

Xavier Franch Gutiérrez

Estructuras de datos

Especificación, diseño e implementación

POLITEXT

Xavier Franch Gutiérrez

Estructuras de datos

Especificación, diseño e implementación

EDICIONS UPC

A mis padres, por todo lo que me han dado
A Cristina, por lo que nos espera juntos
A Miguel Angel, presente en mis recuerdos

Índice

Presentación	11
---------------------------	-----------

Prólogo	13
----------------------	-----------

Capítulo 1 Especificación de tipos abstractos de datos

Presentación.....	19
1.1 Introducción a los tipos abstractos de datos	19
1.2 Modelo de un tipo abstracto de datos.....	25
1.2.1 Signaturas y términos.....	26
1.2.2 Modelos asociados a una signatura.....	29
1.2.3 Evaluación de un término dentro de un álgebra	32
1.2.4 Ecuaciones y especificaciones algebraicas.....	34
1.2.5 Modelo inicial de una especificación.....	37
1.2.6 Otros modelos posibles	43
1.3 Construcción sistemática de especificaciones.....	45
1.3.1 Introducción al uso de especificaciones	45
1.3.2 Clasificación de las operaciones de una especificación.....	46
1.3.3 Método general de construcción de especificaciones.....	47
1.4 Ecuaciones condicionales, símbolos auxiliares y errores.....	48
1.4.1 Ecuaciones condicionales.....	48
1.4.2 Tipos y operaciones auxiliares	50
1.4.3 Tratamiento de errores	51
1.5 Estudio de casos	53
1.5.1 Especificación de algunos tipos de datos clásicos.....	54
1.5.2 Especificación de una tabla de símbolos	60
1.5.3 Especificación de un sistema de reservas de vuelos	63
1.6 Estructuración de especificaciones.....	66
1.6.1 Uso de especificaciones	66
1.6.2 Ocultación de símbolos.....	67
1.6.3 Renombramiento de símbolos.....	68
1.6.4 Parametrización e instanciación	69
1.6.5 Combinación de los mecanismos.....	75
1.7 Ejecución de especificaciones.....	76
1.7.1 La deducción ecuacional.....	77
1.7.2 La reescritura.....	78
Ejercicios	80

Capítulo 2 Implementación de tipos abstractos de datos

Presentación.....	89
2.1 El lenguaje de implementación	89
2.1.1 Representación de tipos.....	91
2.1.2 Sentencias.....	93
2.1.3 Funciones y acciones	95
2.1.4 Ejemplo: una implementación para los conjuntos.....	97
2.2 Corrección de una implementación	98
2.3 Estudio de la eficiencia de las implementaciones.....	108
2.3.1 Notaciones asintóticas	111
2.3.2 Órdenes de magnitud más habituales	116
2.3.3 Análisis asintótico de la eficiencia temporal	118
2.3.4 Análisis asintótico de la eficiencia espacial	121
2.4 Conflicto entre eficiencia y modularidad.....	124
2.4.1 Falta de funcionalidad en la signatura	125
2.4.2 Tipos abstractos de datos recorribles	128
2.4.3 Tipos abstractos de datos abiertos.....	136
Ejercicios	148

Capítulo 3 Secuencias

Presentación.....	151
3.1 Pilas.....	151
3.1.1 Especificación.....	153
3.1.2 Implementación.....	154
3.2 Colas.....	158
3.2.1 Especificación.....	158
3.2.2 Implementación.....	159
3.3 Listas	162
3.3.1 Especificación de las listas con punto de interés.....	162
3.3.2 Implementación de las listas con punto de interés.....	166
3.3.3 Implementación de estructuras de datos con punteros.....	173
3.3.4 Transparencia de la representación usando punteros	178
3.3.5 Implementaciones encadenadas y TAD abiertos	186
Ejercicios	189

Capítulo 4 Árboles

Presentación.....	195
4.1 Modelo y especificación	196
4.1.1 Modelo de árbol general.....	196
4.1.2 Modelo de árbol binario.....	201
4.1.3 Modelo de árbol con punto de interés.....	202

4.2	Implementación	204
4.2.1	Implementación de los árboles binarios	204
4.2.2	Implementación de los árboles generales	213
4.2.3	Variaciones en los otros modelos de árboles	218
4.2.4	Estudio de eficiencia espacial	218
4.3	Recorridos	219
4.3.1	Recorridos en profundidad de los árboles binarios	220
4.3.2	Árboles binarios enhebrados	224
4.3.3	Recorrido por niveles de los árboles binarios	228
4.4	Colas prioritarias	231
4.4.1	Implementación por árboles parcialmente ordenados y casi completos	233
4.4.2	Aplicación: un algoritmo de ordenación	238
	Ejercicios	243

Capítulo 5 Tablas

	Presentación	249
5.1	Especificación	250
5.1.1	Funciones totales	250
5.1.2	Conjuntos	252
5.1.3	Tablas y conjuntos recorribles	252
5.2	Implementación	254
5.2.1	Implementación por listas desordenadas	254
5.2.2	Implementación por listas ordenadas	255
5.2.3	Implementación por vectores de acceso directo	257
5.2.4	Implementación por tablas de dispersión	258
5.3	Funciones de dispersión	259
5.3.1	Funciones de traducción de cadenas a enteros	260
5.3.2	Funciones de restricción de un entero en un intervalo	263
5.3.3	Funciones de traducción de cadenas a enteros en un intervalo	265
5.3.4	Caracterización e implementación de las funciones de dispersión	266
5.4	Organizaciones de las tablas de dispersión	270
5.4.1	Tablas de dispersión encadenadas	270
5.4.2	Tablas de dispersión de direccionamiento abierto	278
5.4.3	Caracterización e implementación de los métodos de redispersión	285
5.4.4	Variantes de las tablas de dispersión de direccionamiento abierto	288
5.4.5	Tablas de dispersión coalescentes	289
5.4.6	Evaluación de las diferentes organizaciones	291
5.4.7	Elección de una organización de dispersión	292
5.4.8	Las organizaciones de dispersión en tablas recorribles y tablas abiertas	295
5.4.9	Inconvenientes de la dispersión	296
5.5	Árboles binarios de búsqueda	297
5.6	Árboles AVL	303
	Ejercicios	315

Capítulo 6 Relaciones binarias y grafos

Presentación.....	319
6.1 Relaciones binarias	320
6.1.1 Especificación.....	320
6.1.2 Implementación.....	324
6.2 Relaciones de equivalencia	332
6.2.1 Implementaciones lineales	335
6.2.2 Implementación arborescente	340
6.2.3 Compresión de caminos.....	342
6.3 Grafos	346
6.3.1 Modelo y especificación.....	348
6.3.2 Implementación.....	352
6.4 Recorridos de grafos.....	360
6.4.1 Recorrido en profundidad	361
6.4.2 Recorrido en anchura.....	364
6.4.3 Recorrido en ordenación topológica	365
6.5 Búsqueda de caminos mínimos	369
6.5.1 Camino más corto de un nodo al resto.....	370
6.5.2 Camino más corto entre todo par de nodos.....	376
6.6 Árboles de expansión minimales	379
6.6.1 Algoritmo de Prim	381
6.6.2 Algoritmo de Kruskal.....	384
Ejercicios	387

Capítulo 7 Uso y diseño de tipos abstractos de datos

Presentación.....	397
7.1 Uso de tipos abstractos de datos existentes	398
7.1.1 Un evaluador de expresiones.....	399
7.1.2 Un gestor de memoria dinámica	405
7.1.3 Un planificador de soluciones.....	412
7.2 Diseño de nuevos tipos abstractos de datos.....	420
7.2.1 Una tabla de símbolos.....	420
7.2.2 Una cola compartida.....	423
7.2.3 Una emisora de televisión.....	430
Ejercicios	439

Bibliografía.....	453
-------------------	-----

Índice temático.....	455
----------------------	-----

Índice de universos.....	461
--------------------------	-----

Presentación

Cuando me piden que escriba el prólogo de un libro, me da un poco de vergüenza, ya que se trata de una de mis asignaturas pendientes: he tenido hijos y he plantado árboles, y también he escrito muchas líneas, pero nunca un libro. Así que hacer de prologuista sin haber sido autor me provoca un cierto sentimiento de jubilación anticipada. En este caso, no obstante, este sentimiento se confunde con una fuerte sensación de orgullo y satisfacción, provocada por el excelente trabajo de alguien que, en parte, me permito considerar discípulo mío en el sentido ancestral de la palabra. Xavier Franch, autor de este libro, ha sido alumno mío durante sus estudios en la Facultat d'Informàtica de Barcelona, colaborador becario mientras era estudiante, después alumno de doctorado y compañero de departamento y, para terminar, siempre hemos trabajado juntos en proyectos de investigación y he dirigido su tesis doctoral. Tengo motivos, pues, para sentir esta satisfacción.

El texto en cuestión, además de actualizar el contenido de las materias ya clásicas de estructuras de datos, se adapta perfectamente al temario de una asignatura de los planes de estudio vigentes en la Facultat d'Informàtica de Barcelona, lo cual justificaría de por sí su existencia. Pero, además, por su actualización del tema puede servir, total o parcialmente, para otros estudios de informática o para cualquier asignatura sobre estructuras de datos de otros planes de estudios en la Universitat Politècnica de Catalunya o en otras universidades. Y, como valor añadido, es destacable la experiencia del autor en la docencia de la asignatura “Estructuras de Datos y Algoritmos”, de los nuevos planes estudio vigentes en la Facultat d'Informàtica de Barcelona.

La notación empleada tanto en las especificaciones como en las implementaciones de las estructuras de datos es Merlí, lenguaje emblemático del proyecto Excalibur y notación que, desde hace ya muchos años, ha caracterizado las diversas enseñanzas algorítmicas en nuestra facultad.

Por todo lo dicho es obvio que no soy nada imparcial a la hora de juzgar el trabajo del profesor Xavier Franch, pero también tengo claro que la parcialidad es una pequeña licencia que, en una presentación, nos podemos permitir.

Como ya he dicho, un excelente texto, que pone al día un tema clásico en informática. Mi enhorabuena al autor. Y también al lector, que encontrará una muestra de aquello que el profesor Turski decía hace muchos años: “no hay nada más práctico que una buena teoría”. Sobre todo si se explica desde un conocimiento sólido de la práctica.

Pere Botella i López

Catedrático del Departamento de Lenguajes y Sistemas Informáticos (U.P.C.)

Decano de la Facultat d'Informàtica de Barcelona (U.P.C.)

Prólogo

El estudio de las estructuras de datos es casi tan antiguo como el nacimiento de la programación, y se convirtió en un tema capital en este ámbito desde finales de la década de los 60. Como es lógico, una de las consecuencias de este estudio es la aparición de una serie de libros de gran interés sobre el tema, algunos de ellos ciertamente excelentes y que se han convertido en piedras angulares dentro de la ciencia de la programación (citemos, por ejemplo, los textos de D.E. Knuth; de A.V. Aho, J. Hopcroft y J.D. Ullman; de E. Horowitz y D. Sahni; de N. Wirth; y, recientemente, de T.H. Cormen, C.E. Leiserson i R.L. Rivest).

Ahora bien, el progreso en el campo de la programación ha dado como resultado la aparición de nuevos conceptos, algunos de los cuales no se han consolidado hasta la segunda mitad de la década de los 80. Muchos de estos conceptos están íntimamente interrelacionados con el ámbito de las estructuras de datos, y ésta es la razón por la cual los libros antes citados han quedado actualmente un poco desfasados en lo que respecta al método de desarrollo de programas que siguen, incluso en sus reediciones más recientes.

En este contexto, he confeccionado el libro "Estructuras de datos. Especificación, diseño e implementación", que trata el estudio de las estructuras de datos dentro del marco de los tipos abstractos de datos. La adopción de este enfoque se inscribe en una metodología de desarrollo modular de programas, que abunda en diferentes propiedades interesantes en la producción industrial de aplicaciones (corrección, mantenimiento, etc.), y permite enfatizar diversos aspectos importantes hoy en día: la necesidad de especificar el *software*, la separación entre la especificación y la implementación, la construcción de bibliotecas de componentes, la reusabilidad del *software*, etc. Diversos autores han explorado esta metodología (sobre todo, desde las aportaciones de B. Liskov y J.V. Guttag), pero sin aplicarla en el contexto de las estructuras de datos.

Destinatario

El libro ha sido concebido sobre todo como un texto de ayuda para alumnos de una asignatura típica de estructura de datos en un primer ciclo de ingeniería en informática; también se puede considerar adecuado para cualquier otra titulación técnica superior o media con contenido informático. A tal efecto, cubre el temario habitual de esta asignatura en tono autoexplicativo, y se ilustra con numerosas figuras, especificaciones y programas.

Dependiendo de los objetivos de la asignatura, el formalismo asociado al estudio de estos temas puede ser más o menos acusado; sea como sea, el libro puede usarse como texto básico de consulta.

Ahora bien, los temas que aparecen en el libro se han desarrollado con más profundidad que la estrictamente requerida por el alumno y, por ello, hay más posibles destinatarios. Por un lado, el mismo profesor de la asignatura, porque puede encontrar en un único volumen los aspectos de especificación y de diseño que no acostumbran a aparecer en los libros de estructuras de datos; además, la inclusión de especificaciones y de programas libera al docente de la necesidad de detallarlos en sus clases. Por otro lado, cualquier informático que quiera profundizar en el estudio de las estructuras de datos más allá de su aspecto puramente de programación, puede encontrar aquí una primera referencia.

Contenido

En el primer capítulo se introduce el concepto de tipo abstracto de datos. Después de analizar su repercusión en el diseño de programas, nos centramos en el estudio de su especificación formal, que es la descripción exacta de su comportamiento. De entre las diferentes opciones existentes de especificación formal, se sigue la llamada especificación ecuacional interpretada con semántica inicial. El capítulo muestra un método general para construir especificaciones para los tipos, les otorga un significado matemático (como álgebras heterogéneas) y también estudia su estructuración, y aquí destaca la posibilidad de definir tipos genéricos, profusamente utilizados a lo largo del libro.

En el segundo capítulo se estudian diversos aspectos sobre la implementación de los tipos de datos. El proceso de implementación se lleva a cabo cuando existe una especificación para el tipo; la segunda sección insiste precisamente en la relación formal entre los dos conceptos, especificación e implementación. También se introduce un punto clave en el análisis de los algoritmos y las estructuras de datos que se desarrollarán posteriormente: el estudio de su eficiencia a través de las denominadas notaciones asintóticas. Por último, se muestran algunas situaciones de la programación con tipos abstractos de datos que pueden causar problemas de eficiencia, y se formulan algunos patrones de comportamiento para solucionar estos problemas.

Las diversas familias de estructuras de datos se introducen en los cuatro capítulos siguientes: se estudian las secuencias; los árboles; las tablas y los conjuntos; y las relaciones binarias y los grafos. Para todas ellas se sigue el mismo método: descripción informal, formulación de un modelo, especificación algebraica del tipo e implementaciones más habituales. Por lo que se refiere a estas últimas, se detalla la representación del tipo y la codificación de las operaciones (hasta el último detalle y buscando la máxima legibilidad posible mediante el uso de funciones auxiliares, diseño descendente, comentarios, etc.), siempre en el caso de implementación en memoria interna; a continuación, se estudia su eficiencia tanto temporal como espacial y se proponen varios ejercicios.

Por último, el capítulo final muestra la integración del concepto de tipo abstracto de datos dentro del desarrollo modular de programas, y lo hace bajo dos vertientes: el uso de los tipos abstractos previamente introducidos y el diseño de nuevos tipos de datos. El estudio se hace a partir de seis ejemplos escogidos cuidadosamente, que muestran la confrontación de los criterios de modularidad y eficiencia en el diseño de programas.

Para leer el texto, son necesarios unos conocimientos fundamentales en los campos de las matemáticas, de la lógica y de la programación. De las matemáticas, los conceptos básicos de conjunto, producto cartesiano, relación, función y otros similares. De la lógica, el concepto de predicado, los operadores booleanos y las cuantificaciones universal y existencial. De la programación, la habilidad de codificar usando un lenguaje imperativo cualquiera (Pascal, C, Ada o similares) que conlleve el conocimiento de los constructores de tipos de datos (tuplas y vectores), de las estructuras de control de flujo (asignaciones, secuencias, alternativas y bucles) y de los mecanismos de encapsulamiento de código (acciones y funciones).

Es importante destacar algunos puntos que el libro no trata, si bien por su temática se podría haber considerado la posibilidad de incluirlos. Primero, no aparecen algunas estructuras de datos especialmente eficientes que, por su complejidad, superan el nivel de una asignatura de primer ciclo de ingeniería; por ejemplo, diversas variantes de montículos y de árboles de búsqueda (*Fibonacci Heaps*, *Red-Black Trees*, *Splay Trees*, etc.) y de dispersión (*Perfect Hashing*, principalmente). También se excluyen algunas otras estructuras que se aplican principalmente a la memoria secundaria, como pueden ser las diversas variantes de árboles B y también los esquemas de dispersión incremental (*Extendible Hashing*, *Linear Hashing*, etc.). Tampoco se tratan en el libro algunos temas característicos de la programación, como pueden ser el estudio de diversas familias de algoritmos (*Greedy Algorithms*, *Dynamic Programming*, etc.) de los cuales constan algunos casos particulares en el capítulo de grafos; o como las técnicas de derivación y de verificación formal de programas, si bien se usan algunos elementos (invariantes de bucles, precondiciones y postcondiciones de funciones, etc.). Hay diversos libros de gran interés que sí tratan en profundidad estos temas, cuyas referencias aparecen convenientemente en este texto. Por último, no se utilizan los conceptos propios de la programación orientada a objetos (básicamente, herencia y vinculación dinámica) para estructurar los tipos de datos formando jerarquías; se ha preferido el enfoque tradicional para simplificar el volumen de la obra y no vernos obligados a introducir la problemática inherente a este paradigma de la programación.

Bibliografía

Las referencias bibliográficas del libro se pueden dividir en dos grandes apartados. Por un lado se citan todos aquellos artículos que son de utilidad para temas muy concretos, cuya referencia aparece integrada en el texto en el mismo lugar en que se aplican. Por el otro, hay diversos textos de interés general que cubren uno o más capítulos del libro y que aparecen dentro del apartado de bibliografía; estos libros han de considerarse como los más destacables en la confección de esta obra y no excluye que haya otros, igualmente buenos,

que no se citan, bien porque su temática es muy similar a alguno de los que sí aparecen, bien porque el desarrollo de los temas es diferente al que se sigue aquí.

Lenguaje

En cualquier texto sobre programación, es fundamental la elección del lenguaje utilizado como vehículo para codificar (y, en este libro, también para especificar) los esquemas que se introducen. En vez de especificar y programar usando algún lenguaje existente, he preferido emplear la notación *Merlí*, diseñada por diversos miembros del Departament de Llenguatges i Sistemes Informàtics (antiguamente, Departament de Programació) de la Universitat Politècnica de Catalunya. Esta notación ha sido utilizada desde principios de los años 80 por los profesores del departamento en la impartición de las asignaturas de programación de los primeros niveles de las titulaciones en informática y ha demostrado su validez como herramienta para el aprendizaje de la programación. Las razones de esta elección son básicamente dos: por un lado, disponer de una notación abstracta que permita expresar fácilmente los diferentes esquemas que se introducen sin ningún tipo de restricción impuesta por el lenguaje; por otro, usar una sintaxis muy parecida tanto para especificar como para implementar los tipos de datos (el hecho de que el mismo lenguaje se pueda usar desde estos dos niveles diferentes refuerza la relación entre la especificación y la implementación de los tipos de datos, que es uno de los objetivos del texto). El inconveniente principal es la necesidad de traducir las especificaciones y los programas que aparecen en este texto a los lenguajes que el lector tenga a su disposición; ahora bien, este inconveniente no parece muy importante, dado que *Merlí* es fácilmente traducible a cualquier lenguaje comercial (a algunos mejor que a otros, eso sí), y que podría haber aparecido el mismo problema fuera cual fuera el lenguaje de trabajo elegido.

Terminología

Dado que, hoy en día, el idioma dominante en el ámbito de la informática es el inglés, he hecho constar las acepciones inglesas junto a aquellos vocablos que denotan conceptos básicos y universalmente aceptados; de esta manera, el lector puede relacionar rápidamente estos conceptos dentro de su conocimiento de la materia o, en el caso de que sea el primer libro que lee sobre estructuras de datos, adquirir el vocabulario básico para la lectura posterior de textos ingleses. Los términos ingleses se escriben siempre en singular independientemente del género con el que se usen en castellano.

Por el mismo motivo, se utilizan de manera consciente varios anglicismos usuales en el ámbito de la programación para traducir algunos términos ingleses. Dichos anglicismos se limitan a lo estrictamente imprescindible, pero he creído conveniente seguir la terminología técnica habitual en vez de introducir vocablos más correctos desde el punto de vista lingüístico pero no tan profusamente usados.

Agradecimientos

Este libro es el resultado de una experiencia personal de varios años de docencia en las asignaturas de estructuras de datos en los planes de licenciatura e ingeniería de la Facultat d'Informàtica de Barcelona de la Universitat Politècnica de Catalunya, por lo que refleja un gran número de comentarios y aportaciones de todos los profesores que, a lo largo de este período, han sido compañeros de asignatura. Quizás el ejemplo más paradigmático sea la colección de ejercicios propuestos en el texto, muchos de ellos provenientes de las listas de ejercicios y exámenes de las asignaturas citadas. Para ellos mi más sincero agradecimiento. En particular, quiero citar al profesor Ricardo Peña por su ayuda durante el primer año que impartí la asignatura "Estructuras de la Información"; a los profesores y profesoras M.T. Abad, J.L. Balcázar, J. Larrosa, J. Marco, C. Martínez, P. Meseguer, T. Moreno, P. Nivela, R. Nieuwenhuis y F. Orejas por la revisión de secciones, versiones preliminares y capítulos enteros del texto y por la detección de errores; y, sobre todo, al profesor Xavier Burgués por todos los años de continuos intercambios de opinión, sugerencias y críticas. A todos ellos, gracias.

Contacto

El lector interesado puede contactar con el autor en la dirección electrónica *franch@lsi.upc.es*, o bien dirigiéndose al departamento de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya. En especial, el autor agradecerá la notificación de cualquier errata detectada en el texto, así como toda sugerencia o crítica a la obra. También existe una página web con información sobre el libro, que se intenta mantener actualizada, cuya dirección es <http://www-lsi.upc.es/~franch/publis/libro-eds.html>.

Barcelona, 10 de Junio de 1996 (primera edición)
12 de Noviembre de 2001 (última edición)

Capítulo 1 Especificación de tipos abstractos de datos

El concepto de tipo abstracto de datos será el marco de estudio de las estructuras de datos que se presentan en el libro. Por ello, dedicamos el primer capítulo a estudiar su significado a partir de lo que se denomina una especificación algebraica, que es la descripción precisa de su comportamiento. También se introducen en profundidad los mecanismos que ofrece la notación Merlí para escribirlas y que serán usados a lo largo del texto en la descripción preliminar de las diferentes estructuras de datos que en él aparecen.

1.1 Introducción a los tipos abstractos de datos

Con la aparición de los lenguajes de programación estructurados en la década de los 60, surge el concepto de *tipo de datos* (ing., *data type*), definido como un conjunto de valores que sirve de dominio de ciertas operaciones. En estos lenguajes (C, Pascal y similares, derivados todos ellos -de forma más o menos directa- de Algol), los tipos de datos sirven sobre todo para clasificar los objetos de los programas (variables, parámetros y constantes) y determinar qué valores pueden tomar y qué operaciones se les pueden aplicar.

Esta noción, no obstante, se reveló insuficiente en el desarrollo de *software* a gran escala, dado que el uso de los datos dentro de los programas no conocía más restricciones que las impuestas por el compilador, lo que era muy inconveniente en los nuevos tipos de datos definidos por el usuario, sobre todo porque no se restringía de ninguna manera su ámbito de manipulación. Para solucionar esta carencia, resumida por J.B. Morris en "Types are not Sets" (*Proceedings ACM Symposium on Principles of Programming Languages, POPL*, 1973), diversos investigadores (citamos como pioneros a S.N. Zilles, J.V. Guttag y el grupo ADJ, formado por J.A. Goguen, J.W. Thatcher, E.G. Wagner y J.B. Wright [ADJ78]) introdujeron a mediados de la década de los 70 el concepto de *tipo abstracto de datos* (ing., *abstract data type*; abreviadamente, *TAD*), que considera un tipo de datos no sólo como el conjunto de valores que lo caracteriza sino también como las operaciones que sobre él se pueden aplicar, juntamente con las diversas propiedades que determinan inequívocamente su comportamiento. Todos estos autores coincidieron en la necesidad de emplear una notación formal para describir el comportamiento de las operaciones, no sólo para impedir cualquier interpretación ambigua sino para identificar claramente el modelo matemático denotado por el TAD.

En realidad, el concepto de TAD ya existe en los lenguajes de programación estructurados bajo la forma de los tipos predefinidos, que se pueden considerar como tipos abstractos sin mucho esfuerzo. Por ejemplo, consideremos el tipo de datos de los enteros que ofrece el lenguaje Pascal; la definición del TAD correspondiente consiste en determinar:

- Cuáles son sus valores. Los números enteros dentro del intervalo [*minint*, *maxint*].
- Cuáles son sus operaciones. La suma, la resta, el producto, y el cociente y el resto de la división.
- Cuáles son las propiedades que cumplen estas operaciones. Hay muchas; por ejemplo: $a+b = b+a$, $a*0 = 0$, etc.

Resumiendo, se puede definir un tipo abstracto de datos como un conjunto de valores sobre los que se aplica un conjunto dado de operaciones que cumplen determinadas propiedades. ¿Por qué "abstracto"? Éste es un punto clave en la metodología que se presentará y se aplicará en todo el libro. El calificativo "abstracto" no significa "surrealista" sino que proviene de "abstracción", y responde al hecho de que los valores de un tipo pueden ser manipulados mediante sus operaciones si se saben las propiedades que éstas cumplen, sin que sea necesario ningún conocimiento ulterior sobre el tipo; en concreto, su implementación en la máquina es absolutamente irrelevante. En el caso de los enteros de Pascal, cualquier programa escrito en este lenguaje puede efectuar la operación $x+y$ (siendo x e y dos variables enteras) con la certeza de que siempre calculará la suma de los enteros x e y , independientemente de su representación interna en la máquina que está ejecutando el programa (complemento a 2, signo y magnitud, etc.) porque, sea ésta cual sea, la definición de los enteros de Pascal asegura que la suma se comporta de una manera determinada. En otras palabras, la manipulación de los objetos de un tipo sólo depende del comportamiento descrito en su *especificación* (ing., *specification*) y es independiente de su *implementación* (ing., *implementation*):

- La especificación de un TAD consiste en establecer las propiedades que lo definen. Para que sea útil, una especificación ha de ser precisa (sólo tiene que decir aquello realmente imprescindible), general (adaptable a diferentes contextos), legible (que sirva como instrumento de comunicación entre el especificador y los usuarios del tipo, por un lado; y entre el especificador y el implementador, por el otro) y no ambigua (que evite posteriores problemas de interpretación). La especificación del tipo, que es única, define totalmente su comportamiento a cualquier usuario que lo necesite. Según su grado de formalismo, será más o menos fácil de escribir y de leer, y más o menos propensa a ser ambigua o incompleta.
- La implementación de un TAD consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación, todo ello usando un lenguaje de programación convencional. Para que sea útil, una implementación ha de ser estructurada (para facilitar su desarrollo), eficiente (para optimizar el uso de recursos del computador) y legible (para facilitar su mantenimiento).

Una implementación del TAD (puede haber muchas, cada una de ellas pensada para un contexto de uso diferente) es totalmente transparente a los usuarios del tipo y no se puede escribir hasta haber determinado claramente su especificación; el cambio de una implementación por otra que respete el comportamiento deseado del tipo no ha de cambiar en absoluto la especificación ni, por consiguiente, la visión que de él tienen sus usuarios, que se limitarán a recompilar la aplicación correspondiente.

La verdadera utilidad de los TAD aparece en el diseño de nuevos tipos de datos. Imaginemos que se quiere construir un programa Pascal que calcule la suma de una secuencia de números complejos introducida por el terminal, acabada por el valor $0 + 0i$ (para simplificar la escritura de algunos detalles irrelevantes, supondremos que tanto la parte real como la parte imaginaria de los números complejos son enteros en vez de reales), escribiendo el resultado en la pantalla. Fieles a la metodología que acabamos de esbozar, enfocamos el caso como un ejercicio resoluble a partir de la especificación de un TAD para los números complejos, definible de la siguiente forma:

- Cuáles son sus valores. Todos aquellos números complejos de partes real e imaginaria enteras y dentro del intervalo $[minint, maxint]$.
- Cuáles son sus operaciones. Como mínimo, y dada la funcionalidad del programa, se necesita una operación para *sumar* complejos, otra para *crear* un complejo a partir de dos enteros, y dos más para obtener las partes *real* e *imaginaria*.
- Cuáles son las propiedades que cumplen las operaciones. Las propias de los complejos.

Una vez definido el TAD para los complejos es posible utilizarlo desde un programa Pascal: se pueden declarar variables del tipo, usar objetos del tipo como parámetros de funciones, utilizar el tipo para construir otros más complicados, etc. Dicho de otra forma, el (nuevo) TAD de los complejos tiene las mismas características de uso que el TAD (predefinido) de los enteros y desde un programa Pascal sus diferencias son exclusivamente notacionales. Como consecuencia, se puede escribir el programa principal que calcula la suma de los complejos sin implementar el TAD, tal y como se muestra en la fig. 1.

```
program suma_complejos;  
var res: complejo; a, b: integer;  
begin  
  res := crear(0, 0); read(a, b);  
  while (a <> 0) or (b <> 0) do begin  
    res := sumar(res, crear(a, b)); read(a, b)  
  end;  
  writeln('El resultado es: ', real(res), ' + ', imaginaria(res), 'i.')  
end.
```

Fig. 1.1: programa Pascal para sumar una secuencia de números complejos.

Es decir, el programa resultante es independiente de la implementación del tipo de los complejos en Pascal. Una vez determinadas las operaciones, para completar la aplicación se escoge una representación para el TAD y se implementa; por ejemplo, en la fig. 1.2 se da una representación que mantiene los complejos en notación binómica.

```

type complejo = record re, im: integer end;

function crear (a, b: integer): complejo;
var c: complejo;
begin
    c.re := a; c.im := b; crear := c
end;

function sumar (a, b: complejo): complejo;
begin
    a.re := a.re + b.re; a.im := a.im + b.im; sumar := a
end;

function real (c: complejo): integer;
begin
    real := c.re
end;

function imaginaria (c: complejo): integer;
begin
    imaginaria := c.im
end;

```

Fig. 1.2: codificación en Pascal de una representación binómica para los números complejos.

La extrapolación de esta técnica a sistemas de gran tamaño conduce al llamado *diseño modular* de las aplicaciones (ing., *modular design*), formulado por D.L. Parnas en 1972 en el artículo "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12); también lo llamaremos *diseño con TAD* (v. [LiG86] para un estudio en profundidad). El diseño modular es una generalización del *diseño descendente* de programas (ing., *stepwise design*; llamado también *diseño por refinamientos sucesivos*) introducido a finales de los años 60 por diversos autores, entre los que destacan O.-J. Dahl, E.W. Dijkstra y C.A.R. Hoare (v. *Structured Programming*, Academic Press Inc., 1972). El diseño descendente se caracteriza por el hecho de dividir el problema original en varios subproblemas más pequeños, cada uno de ellos con una misión bien determinada dentro del marco general del proyecto, que interaccionan de manera clara y mínima y de tal forma que todos ellos juntos solucionan el problema inicial; si algunos subproblemas siguen siendo demasiado complicados, se les aplica el mismo proceso, y así

sucesivamente hasta llegar al estado en que todos los subproblemas son lo bastante sencillos como para detener el proceso. El resultado es una estructura jerárquica que refleja las descomposiciones efectuadas; cada descomposición es el resultado de abstraer las características más relevantes del problema que se está tratando de los detalles irrelevantes en el nivel de razonamiento actual, los cuales adquieren importancia en descomposiciones sucesivas. Desde el punto de vista de su gestión, cada subproblema es un TAD que se encapsula en lo que se denomina un *módulo*¹ (ing., *module*); precisamente, la mejora respecto al diseño descendente proviene de la ocultación de la representación de los datos y de la limitación de su manipulación al ámbito del módulo que define el tipo.

A primera vista, puede parecer costoso, e incluso absurdo, dividir una aplicación en módulos y escribir procedimientos y funciones para controlar el acceso a la estructura que implementa un TAD. Es decir, ¿por qué no escribir directamente la fórmula de la suma de complejos allí donde se necesite, en vez de encapsular el código dentro de una función? La respuesta es que esta metodología abunda en diversas propiedades interesantes:

- Abstracción. Los usuarios de un TAD no necesitan conocer detalles de implementación (tanto en lo que se refiere a la representación del tipo como a los algoritmos y a las técnicas de codificación de las operaciones), por lo que pueden trabajar con un grado muy alto de abstracción. Como resultado, la complejidad de un programa queda diluida entre sus diversos componentes.
- Corrección. Un TAD puede servir como unidad indivisible en las pruebas de programas, de manera que en una aplicación que conste de diversos tipos abstractos no tengan que probarse todos a la vez, sino que es factible y recomendable probarlos por separado e integrarlos más adelante. Es mucho más fácil detectar los errores de esta segunda manera, porque las entidades a probar son más pequeñas y las pruebas pueden ser más exhaustivas. Por otro lado, la adopción de una técnica formal de especificación como la que se explica en el resto del capítulo posibilita la verificación formal de la aplicación de manera que, eventualmente, se puede demostrar la corrección de un programa; ésta es una mejora considerable, porque la prueba empírica muestra la ausencia de errores en determinados contextos, pero no asegura la corrección absoluta. Hay que decir, no obstante, que la complejidad intrínseca de los métodos formales, junto con el volumen de los TAD que aparecen en aplicaciones reales, y también la inexistencia de herramientas totalmente automáticas de verificación, dificultan (y, hoy en día, casi imposibilitan) la verificación formal completa.
- Eficiencia. La separación entre un programa y los TAD que usa favorece la eficiencia, ya que la implementación de un tipo se retrasa hasta conocer las restricciones de eficiencia sobre sus operaciones, y así se pueden elegir los algoritmos óptimos².

¹ En realidad, el diseño modular identifica no sólo TAD (encapsulados en los llamados *módulos de datos*) sino también los llamados *módulos funcionales*, que se pueden catalogar como algoritmos no triviales que operan sobre diversos TAD (v. [LiG86] para más detalles).

² Aunque, como veremos a lo largo del texto, la inaccesibilidad de la implementación fuera de los módulos de definición de los TAD comporta a veces problemas de eficiencia.

- Legibilidad. Por un lado, la estructuración de la información en varios TAD permite estudiar los programas como un conjunto de subprogramas con significado propio y de más fácil comprensión, porque la especificación de un TAD es suficiente para entender su significado. Por lo que respecta a la implementación, los programas que usan los diferentes TAD resultan más fáciles de entender, dado que no manipulan directamente estructuras de datos sino que llaman a las operaciones definidas para el tipo, que ya se encargarán de gestionar las estructuras subyacentes de manera transparente³.
- Mantenimiento. Cualquier modificación que se tenga que efectuar sobre un programa provocada por cambios en sus requerimientos, por ampliaciones si se desarrollan versiones sucesivas (prototipos), o por su funcionamiento incorrecto no requiere normalmente examinar y modificar el programa entero sino sólo algunas partes. La identificación de estas partes queda facilitada por la estructuración lógica en TAD. Una vez más, es importante destacar que los cambios en la codificación de un TAD no afectan a la implementación de los módulos que lo usan, siempre que el comportamiento externo de las operaciones no cambie.
- Organización. La visión de una aplicación como un conjunto de TAD con significado propio permite una óptima repartición de tareas entre los diferentes componentes de un equipo de trabajo, que pueden desarrollar cada TAD independientemente y comunicarse sólo en los puntos en que necesiten interaccionar. Esta comunicación, además, es sencilla, ya que consiste simplemente en saber qué operaciones ofrecen los TAD y qué propiedades cumplen (es decir, en conocer su especificación).
- Reusabilidad. Los TAD diseñados en una especificación pueden ser reutilizables a veces en otros contextos con pocos cambios (lo ideal es que no haya ninguno). En este sentido es importante, por un lado, disponer de un soporte para acceder rápidamente a los TAD (mediante bibliotecas de módulos reusables) y, por otro, escoger las operaciones adecuadas para el tipo en el momento de su definición, incluso añadiendo algunas operaciones sin utilidad inmediata, pero que puedan ser usadas en otros contextos futuros.
- Seguridad. La imposibilidad de manipular directamente la representación evita el mal uso de los objetos del tipo y, en particular, la generación de valores incorrectos. Idealmente los lenguajes de programación deberían reforzar esta prohibición limitando el ámbito de manipulación de la representación. A pesar de que algunos de ellos lo hacen (Ada-83, Modula-2 y la familia de lenguajes orientados a objetos, incluyendo Java, C++, Eiffel, Ada-95 y algunas versiones no estándares de Pascal), hay muchos que no, y es necesario un sobreesfuerzo y autodisciplina por parte del programador para adaptar los conceptos del diseño con TAD a las carencias del lenguaje de programación.

³ Evidentemente, sin olvidar la adopción de técnicas clásicas para mejorar la legibilidad, como por ejemplo el uso de diseño descendente en el cuerpo de las funciones y acciones, la inserción de comentarios, y la aserción de predicados que especifiquen la misión de las funciones, bucles, etc.

Una vez vistos los conceptos de tipo de datos y tipo abstracto de datos, queda claro que el primero de ellos es una limitación respecto al segundo y por ello lo rechazamos; en el resto del texto, cualquier referencia al término "tipo de datos" se ha de interpretar como una abreviatura de "tipo abstracto de datos". Por otro lado, notemos que todavía no ha sido definida la noción de estructura de datos que da título al libro. A pesar de que no se puede decir que haya una definición estándar, de ahora en adelante consideraremos que una *estructura de datos* (ing., *data structure*) es la representación de un tipo abstracto de datos que combina los constructores de tipo y los tipos predefinidos habituales en los lenguajes de programación (por lo que respecta a los primeros, tuplas, vectores y punteros principalmente; en lo referente a los segundos, booleanos, enteros, reales y caracteres normalmente). Determinadas combinaciones presentan propiedades que las hacen interesantes para implementar ciertos TAD, y dan lugar a unas familias ya clásicas en el campo de la programación: estructuras lineales para implementar secuencias, árboles para implementar jerarquías, tablas de dispersión para implementar funciones y conjuntos, multilistas para implementar relaciones binarias, etc. La distinción entre el modelo de un TAD y su implementación mediante una estructura de datos es fundamental, y se refleja a lo largo del texto en la especificación del TAD previa al estudio de la implementación.

1.2 Modelo de un tipo abstracto de datos

El lenguaje natural por sí solo no es una buena opción para describir el comportamiento de un TAD, dada su falta de precisión. Se requiere, pues, una notación formal que permita expresar sin ningún atisbo de ambigüedad las propiedades que cumplen las operaciones de un tipo. Desde que apareció esta necesidad se han desarrollado diversos estilos de especificación formal, cada uno de ellos con sus peculiaridades propias, que determinan su contexto de uso. A lo largo del texto seguiremos la llamada *especificación algebraica* (también denominada *ecuacional*), que establece las propiedades del TAD mediante ecuaciones con variables cuantificadas universalmente. La notación Merlí recoge las construcciones más habituales en este campo.

El estudio del significado de estas especificaciones se hará dentro del ámbito de unos objetos matemáticos llamados *álgebras* (ing., *algebra*). El grupo ADJ fue el pionero y máximo impulsor en la búsqueda del modelo matemático de un tipo abstracto (ya desde [ADJ78], que recoge los resultados obtenidos en la primera mitad de la década de los 70, la mayoría de ellos publicados como *reports* de investigación de los laboratorios IBM Watson Research Center), y son incontables las aportaciones posteriores de otros autores. En el año 1985 se publicó el texto [EhM85], que constituye una compilación de todos los conceptos formales sobre el tema, y al que se ha de considerar como la referencia principal de esta sección; en él se formulan una serie de teoremas y propiedades de gran interés, que aquí se introducen sin demostrar. Posteriormente, los mismos autores presentaron [EhM90], que estudia el modelo formal de los TAD respetando su estructura interna, aspecto éste no tratado aquí.

1.2.1 Signaturas y términos

El primer paso al especificar un TAD consiste en identificar claramente sus objetos y operaciones. En Merlí se encapsula la especificación dentro de una estructura llamada *universo* (equivalente al concepto tradicional de módulo) donde, para empezar, se escribe el nombre del TAD en definición tras la palabra clave "tipo" y se establecen las operaciones detrás de la palabra clave "ops". Para cada operación se indica su nombre, el número y el tipo de sus parámetros y el tipo de su resultado; es lo que se llama la *signatura* (ing., *signature*) de la operación. Al universo se le da un nombre que se usará para referenciarlo.

En la fig. 1.3 se muestra un universo para el TAD de los booleanos. Por omisión, las operaciones de una signatura se invocarán con los parámetros (si los hay) separados por comas y encerrados entre paréntesis; para indicar una sintaxis diferente, se usa el carácter de subrayado para determinar la ubicación de todos los parámetros (por ejemplo \neg _, $_ \vee _$ y $_ \wedge _$). Por otro lado, diversos operadores con la misma signatura se pueden introducir en la misma línea (como *cierto* y *falso*, o $_ \vee _$ y $_ \wedge _$).

```

universo BOOL es
  tipo bool
  ops
    cierto, falso:  $\rightarrow$  bool
     $\neg$ _: bool  $\rightarrow$  bool
     $\_ \vee \_$ ,  $\_ \wedge \_$ : bool bool  $\rightarrow$  bool
  funiverso

```

Fig. 1.3: signatura de un TAD para los booleanos.

En la fig. 1.4 se ofrece un TAD para los naturales; es necesario, no obstante, introducir también el tipo de los booleanos porque, aunque el objetivo principal es especificar el tipo *nat*, hay un símbolo de operación, *ig*, que involucra *bool*⁴. Las operaciones sobre *bool* son las estrictamente imprescindibles para especificar más adelante los naturales. La signatura se puede presentar gráficamente encerrando en círculos los nombres de los tipos y representando las operaciones como flechas que salen de los tipos de los parámetros y van a parar al tipo del resultado (v. fig. 1.5).

Resumiendo, en un universo se establece, para empezar, qué objetos y qué operaciones intervienen en la definición del TAD; es lo que se conoce como *signatura de un tipo abstracto de datos*⁵. Al escribir la signatura, todavía no se da ninguna propiedad sobre los símbolos de operación; además, tampoco se les proporciona ningún significado, aparte de la información puramente subjetiva de sus nombres (que son totalmente arbitrarios).

⁴ En el apartado 1.3.1 veremos cómo aprovechar especificaciones ya construidas.

⁵ Es decir, la palabra "signatura" puede usarse refiriéndose a operaciones individuales o a todo un TAD.

universo NAT es
tipo nat, bool
ops
 cero: $\rightarrow \text{nat}$
 suc: $\text{nat} \rightarrow \text{nat}$
 suma: $\text{nat nat} \rightarrow \text{nat}$
 ig: $\text{nat nat} \rightarrow \text{bool}$
 cierto, falso: $\rightarrow \text{bool}$
funiverso

Fig. 1.4: *signatura de un TAD para los naturales.*

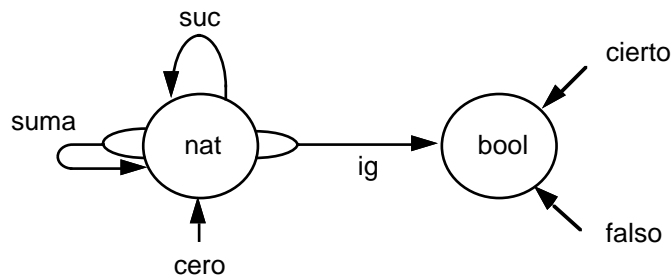


Fig. 1.5: *representación gráfica de la signatura de la fig. 1.4.*

A continuación, se quiere formalizar el concepto de signatura como primer paso hacia la búsqueda del modelo asociado a una especificación. Previamente hay que introducir una definición auxiliar: dado un conjunto S , definimos un S -conjunto A como una familia de conjuntos indexada por los elementos de S , $A = (A_s)_{s \in S}$; el calificativo "indexada" significa que cada uno de los elementos de S sirve como medio de referencia para un conjunto de A . Definimos el S -conjunto vacío \emptyset como aquel S -conjunto que cumple $\forall s: s \in S: \emptyset_s = \emptyset$. Sobre los S -conjuntos se definen operaciones de intersección, unión, pertenencia e inclusión, cuya definición es, para dos S -conjuntos $A = (A_s)_{s \in S}$ y $B = (B_s)_{s \in S}$:

- $A \cup B = (A_s \cup B_s)_{s \in S}$, $A \cap B = (A_s \cap B_s)_{s \in S}$
- $A \subseteq B \equiv \forall s: s \in S: A_s \subseteq B_s$
- $v \in A \equiv \exists s: s \in S: v \in A_s$

Ahora, se puede definir una signatura SIG como un par $SIG = (S_{SIG}, OP_{SIG})$ o, para reducir subíndices, $SIG = (S, OP)$, donde:

- S es un conjunto de *géneros* (ing., *sort*); cada género representa el conjunto de valores que caracteriza el tipo.

- OP es un conjunto de *símbolos de operaciones* (ing., *operation symbol*) o, más exactamente, una familia de conjuntos indexada por la signatura de las operaciones, es decir, un $\langle S^* \times S \rangle$ -conjunto⁶, $OP = (OP_{w \rightarrow s})_{w \in S^*, s \in S}$. Cada uno de estos conjuntos agrupa los símbolos de operaciones que tienen la misma signatura (exceptuando el nombre). La longitud de w , denotada como $\|w\|$, recibe el nombre de *aridad* (ing., *arity*) de la operación. Los símbolos de aridad 0 se llaman *símbolos de constantes* (ing., *constant symbol*); su tratamiento no difiere del que se da al resto de símbolos (a pesar de que, a veces, habrá que distinguirlos al formular ciertas definiciones recursivas).

Por ejemplo, la signatura $NAT = (S, OP)$ de la fig. 1.4 queda:

$$\begin{aligned} S &= \{nat, bool\} \\ OP_{\rightarrow nat} &= \{cero\} & OP_{nat \rightarrow nat} &= \{suc\} & OP_{nat \ nat \rightarrow nat} &= \{suma\} \\ OP_{\rightarrow bool} &= \{cierto, falso\} & OP_{nat \ nat \rightarrow bool} &= \{ig\} \end{aligned}$$

y el resto de conjuntos $OP_{w \rightarrow s}$ son vacíos.

Mediante la aplicación sucesiva y correcta de símbolos de operaciones de una signatura se pueden construir *términos* (ing., *term*) sobre ella; por ejemplo, $suc(suc(suc(cero)))$ es un término sobre la signatura NAT . El conjunto de términos (generalmente infinito) que se puede construir sobre una signatura $SIG = (S, OP)$ es un S -conjunto denotado con T_{SIG} , $T_{SIG} = (T_{SIG,s})_{s \in S}$, definido recursivamente como:

- $\forall c: c \in OP_{\rightarrow s}: c \in T_{SIG,s}$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s} \wedge n > 0: \forall t_1: t_1 \in T_{SIG,s_1}; \dots \forall t_n: t_n \in T_{SIG,s_n}: op(t_1, \dots, t_n) \in T_{SIG,s}$
- No hay nada más en T_{SIG}

Es decir, el conjunto de términos contiene todas las combinaciones posibles de aplicaciones de operaciones, respetando aridades y tipos. Los términos que pertenecen a $T_{SIG,s}$ se llaman *términos sobre SIG de género s*. Por ejemplo, unos cuantos términos correctos para el conjunto de términos $T_{NAT} = (T_{NAT,nat}, T_{NAT,bool})$ son *cero*, *suc(cero)* y también *suma(suc(cero), suma(cero, cero))*, que están dentro de $T_{NAT,nat}$ y *falso* e *ig(suc(cero), cero)*, que están dentro de $T_{NAT,bool}$. En cambio, no son términos las expresiones *suc(cierto)* y *suc(cero, cero)*, porque violan las reglas de formación dadas. La estructura de los términos queda patente mediante una representación gráfica como la fig. 1.6.

La definición de término no considera la existencia de operaciones de invocación no funcional, como \neg o $_ \vee _$. Este hecho es irrelevante al ser la diferencia puramente sintáctica; si se quieren incorporar, es necesario incluir un mecanismo de parentización en la definición.

⁶ S^* representa el monoide libre sobre S , es decir, secuencias de elementos de S (v. sección 1.5 para una introducción y el capítulo 3 para más detalles). Por su parte, $\langle S \times T \rangle$ representa el producto cartesiano de los conjuntos S y T .

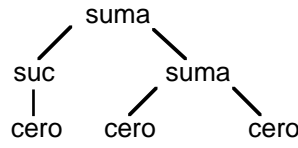


Fig. 1.6: representación gráfica del término $\text{suma}(\text{suc}(\text{cero}), \text{suma}(\text{cero}, \text{cero}))$.

El siguiente paso consiste en incorporar *variables* (ing., *variable*) en los términos. Las variables son símbolos de aridad 0 (como las constantes) que pueden tomar cualquier valor y, por ello, permiten expresar propiedades universales que han de cumplir todos los elementos de un tipo. Dada la signatura $SIG = (S, OP)$, el S -conjunto V será un *conjunto de variables sobre SIG*, si no hay variables repetidas ni ninguna variable con el mismo nombre que una constante; entonces, el S -conjunto $T_{SIG}(V)$ de todos los términos sobre SIG con variables de V , $T_{SIG}(V) = (T_{SIG,s}(V))_{s \in S}$ se define como:

- $\forall c: c \in OP_{\rightarrow s}: c \in T_{SIG,s}(V)$
- $\forall v: v \in V_s: v \in T_{SIG,s}(V)$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s} \wedge n > 0$:
 $\forall t_1: t_1 \in T_{SIG,s_1}(V): \dots \forall t_n: t_n \in T_{SIG,s_n}(V): op(t_1, \dots, t_n) \in T_{SIG,s}(V)$
- No hay nada más en $T_{SIG}(V)$

Denotaremos con Vars_{SIG} todos los posibles conjuntos de variables sobre SIG . Por ejemplo, dado $V \in \text{Vars}_{NAT}$ tal que $V_{nat} = \{x, y\}$ y $V_{bool} = \{z\}$, algunos términos (con variables) válidos son cero , $\text{suc}(x)$ y $\text{suma}(\text{suc}(\text{cero}), x)$, que están dentro de $T_{NAT,nat}(V)$, y también $\text{ig}(y, x)$, que está dentro de $T_{NAT,bool}(V)$; en cambio, las expresiones $\text{suc}(z)$ e $\text{ig}(x, z)$ no son términos.

1.2.2 Modelos asociados a una signatura

Hasta ahora, los TAD han sido tratados a nivel sintáctico. El siguiente objetivo consiste en traducir los símbolos de un universo a los objetos que se quiere modelizar. Así, si se quiere que la signatura $NAT = (S, OP)$, $S = \{nat, bool\}$, $OP = \{\text{cero}, \text{suc}, \text{suma}, \text{ig}, \text{cierto}, \text{falso}\}$, tenga el significado esperado dados los nombres de los géneros y de las operaciones, hay que asociar los números naturales \mathcal{N} a *nat*, el álgebra de Boole \mathcal{B} de valores \mathcal{C} y \mathcal{F} (cierto y falso) a *bool*, el valor 0 de los naturales a *cero*, la operación de sumar el uno a un natural a *suc*, la operación + de sumar naturales a *suma*, la igualdad = de los naturales a *ig*, el valor \mathcal{C} a *cierto* y el valor \mathcal{F} a *falso*. Para establecer claramente esta correspondencia pueden estructurarse los naturales y los booleanos y asociar uno a uno los símbolos de la signatura:

$$\begin{aligned}
\mathcal{NAT} &= (S_{\mathcal{NAT}}, OP_{\mathcal{NAT}}), \\
S_{\mathcal{NAT}} &= \{nat_{\mathcal{NAT}}, bool_{\mathcal{NAT}}\}, \text{ donde } nat_{\mathcal{NAT}} \equiv \mathcal{N}, bool_{\mathcal{NAT}} \equiv \mathcal{B}^7 \\
OP_{\mathcal{NAT}} &= \{cero_{\mathcal{NAT}}, suc_{\mathcal{NAT}}, suma_{\mathcal{NAT}}, ig_{\mathcal{NAT}}, cierto_{\mathcal{NAT}}, falso_{\mathcal{NAT}}\}, \text{ donde} \\
& \quad cero_{\mathcal{NAT}} \equiv 0, suc_{\mathcal{NAT}} \equiv +1, ig_{\mathcal{NAT}} \equiv =, suma_{\mathcal{NAT}} \equiv +, cierto_{\mathcal{NAT}} \equiv C, falso_{\mathcal{NAT}} \equiv \mathcal{F}
\end{aligned}$$

Esta estructura se llama *álgebra respecto de la signatura SIG* o, para abreviar, *SIG-álgebra* (ing., *SIG-algebra*), y se caracteriza porque da una interpretación de cada uno de los símbolos de la signatura. En el ejemplo anterior, \mathcal{NAT} es una *NAT-álgebra* y el subíndice " \mathcal{NAT} " se puede leer como "interpretación dentro del modelo \mathcal{NAT} ".

Más formalmente, dada una signatura $SIG = (S, OP)$, una *SIG-álgebra* \mathcal{A} es un par ordenado, $\mathcal{A} = (S_{\mathcal{A}}, OP_{\mathcal{A}})$, siendo $S_{\mathcal{A}}$ un S -conjunto y $OP_{\mathcal{A}}$ un $\langle S^* \times S \rangle$ -conjunto, definida como:

- $\forall s: s \in S: s_{\mathcal{A}} \in S_{\mathcal{A}}; s_{\mathcal{A}}$ son los *conjuntos de base* (ing., *carrier set*) de \mathcal{A} .
- $\forall c: c \in OP_{\rightarrow S}: c_{\mathcal{A}}: \rightarrow s_{\mathcal{A}} \in OP_{\mathcal{A}}; c_{\mathcal{A}}$ son las *constantes* de \mathcal{A} .
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow S} \wedge n > 0: op_{\mathcal{A}}: s_{1_{\mathcal{A}}} \times \dots \times s_{n_{\mathcal{A}}} \rightarrow s_{\mathcal{A}} \in OP_{\mathcal{A}}; op_{\mathcal{A}}$ son las *operaciones* de \mathcal{A} .

Así pues, una álgebra es un conjunto de valores sobre el que se aplican operaciones; esta definición es idéntica a la de TAD y por este motivo normalmente se estudian los TAD dentro del marco de las *álgebras heterogéneas*, es decir, álgebras que implican varios géneros.

Otras veces, las asociaciones a efectuar con los símbolos de la signatura no son tan obvias. Por ejemplo, consideremos la signatura de la fig. 1.7; supongamos que los símbolos *nat*, *bool*, *cero*, *suc*, *ig*, *cierto* y *falso* tienen el significado esperado dado su nombre, y planteémonos qué significa el género *x* y qué significan las operaciones *crea*, *convierte*, *fusiona* y *está?*. Una primera solución consiste en pensar que *x* representa los conjuntos de naturales y que las operaciones significan:

- crea* $\equiv \emptyset$, el conjunto vacío.
- convierte*(*m*) $\equiv \{m\}$, el conjunto que sólo contiene el elemento *m*.
- fusiona* $\equiv \cup$, la unión de conjuntos.
- está?* $\equiv \in$, la operación de pertenencia de un elemento a un conjunto.

Todo encaja. Ahora bien, es igualmente lícito conjeturar que *x* es el monoide libre sobre los naturales (secuencias de naturales, v. sección 1.5) y que las operaciones significan:

- crea* $\equiv \lambda$, la secuencia vacía
- convierte*(*m*) $\equiv m$, la secuencia que sólo contiene el elemento *m*.
- fusiona*(*r*, *s*) $\equiv r.s$, la concatenación de dos secuencias.
- está?* $\equiv \in$, la operación de pertenencia de un elemento a una secuencia.

⁷ A lo largo de este texto, el símbolo \equiv se usa con el significado "se define como".

```

universo X es
  tipo x, nat, bool
  ops
    crea: → x
    convierte: nat → x
    fusiona: x x → x
    está?: nat x → bool
    cero, suc, ig, cierto y falso: v. fig. 1.4
funiverso

```

Fig. 1.7: una signatura misteriosa.

Incluso se puede interpretar alguna operación de manera atípica respecto a su nombre; en el primer modelo dado es válido asociar la diferencia entre conjuntos a *fusiona*, porque también se respeta la signatura de la operación.

Con la información disponible hasta el momento no se puede decidir cuál es el modelo asociado a la signatura, pues aún no se han especificado las propiedades de sus símbolos; es decir, existen infinitas *SIG*-álgebras asociadas a una signatura *SIG*, todas ellas igualmente válidas, que pueden variar en los conjuntos de base o, a conjuntos de base idénticos, en la interpretación de los símbolos de las operaciones. El conjunto de todas las *SIG*-álgebras se denota mediante Alg_{SIG} . Desde este punto de vista, se puede contemplar una signatura como una plantilla que define la forma que deben tener todos sus posibles modelos.

Más adelante se dan ciertos criterios para elegir un modelo como significado de una especificación; a tal efecto, se presenta a continuación una *SIG*-álgebra particular, fácil de construir: la *SIG*-álgebra resultante de considerar como álgebra el conjunto de términos. Sea $\text{SIG} = (S, OP)$ una signatura y $V \in \text{Vars}_{\text{SIG}}$, el *álgebra de términos* (ing., *term-algebra*) sobre *SIG* y V , denotada por $T_{\text{SIG}}(V)$, es el álgebra $T_{\text{SIG}}(V) = (S_T, OP_T)^8$, siendo el S -conjunto S_T , $S_T = (s_T)_{s \in S}$, los conjuntos de base y el $\langle S^* \times S \rangle$ -conjunto OP_T , $OP_T = ((op_T)_{W \rightarrow S})_{W \in S^*, S \in S}$, las operaciones, definidos como:

- $\forall s: s \in S: s_T \in S_T$, definido como: $s_T \equiv T_{\text{SIG},s}(V)$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s}: op_T: T_{\text{SIG},s_1}(V) \times \dots \times T_{\text{SIG},s_n}(V) \rightarrow T_{\text{SIG},s}(V) \in OP_T$, definida como:
 $op_T(t_1, \dots, t_n) \equiv op(t_1, \dots, t_n)$.

Resumiendo, los conjuntos de base son todos los términos que se pueden construir a partir de los símbolos de operaciones y de las variables, y las operaciones son las reglas de formación de términos considerando la signatura concreta. De manera análoga, se puede definir el álgebra de términos sin variables. Como ejemplo, en la fig. 1.8 se construye T_{NAT} .

⁸ Para mayor claridad, se omiten algunos subíndices.

$$\begin{aligned}
S_T &= (nat_T, bool_T) \\
nat_T &\equiv T_{NAT,nat} = \{cero\} \cup \{suc(x) / x \in T_{NAT,nat}\} \cup \{suma(x, y) / x, y \in T_{NAT,nat}\} \\
bool_T &\equiv T_{NAT,bool} = \{cierto\} \cup \{falso\} \cup \{ig(x, y) / x, y \in T_{NAT,nat}\} \\
OP_T: & \quad cero_T: \rightarrow T_{NAT,nat}; \quad cero_T \equiv cero \\
& \quad suc_T: T_{NAT,nat} \rightarrow T_{NAT,nat}; \quad suc_T(x) \equiv suc(x) \\
& \quad suma_T: T_{NAT,nat} \times T_{NAT,nat} \rightarrow T_{NAT,nat}; \quad suma_T(x, y) \equiv suma(x, y) \\
& \quad ig_T: T_{NAT,nat} \times T_{NAT,nat} \rightarrow T_{NAT,bool}; \quad ig_T(x, y) \equiv ig(x, y) \\
& \quad cierto_T, falso_T: \rightarrow T_{NAT,bool}; \quad cierto_T \equiv cierto, falso_T \equiv falso
\end{aligned}$$

Fig. 1.8: álgebra de términos $T_{NAT} = (S_T, OP_T)$ para la signatura NAT.

1.2.3 Evaluación de un término dentro de un álgebra

Una vez determinada la interpretación de cada símbolo de una signatura SIG dentro de una SIG -álgebra \mathcal{A} , es inmediato definir un mecanismo para calcular el valor representado por un término de T_{SIG} en \mathcal{A} . Previamente, es necesario introducir el concepto de homomorfismo entre álgebras.

Dados dos S -conjuntos A y B , definimos una S -aplicación $f: A \rightarrow B$ como una familia de aplicaciones indexada por los elementos de S , $f = (f_s: A_s \rightarrow B_s)_{s \in S}$. Clasificamos f como inyectiva, exhaustiva o biyectiva, si y sólo si todas las f_s lo son a la vez. Entonces, dadas la signatura $SIG = (S, OP)$ y dos SIG -álgebras \mathcal{A}_1 y \mathcal{A}_2 , un (S) -homomorfismo (ing., *homomorphism*) de \mathcal{A}_1 a \mathcal{A}_2 es una S -aplicación $f: \mathcal{A}_1 \rightarrow \mathcal{A}_2$ que cumple:

$$\begin{aligned}
- \forall c: c \in OP_{\rightarrow S}: f_s(c_{\mathcal{A}_1}) &= c_{\mathcal{A}_2} \\
- \forall op: op \in OP_{s_1 \dots s_n \rightarrow S} \wedge n > 0: \forall t_1: t_1 \in s_1: \dots \forall t_n: t_n \in s_n: \\
& f_s(op_{\mathcal{A}_1}(t_1, \dots, t_n)) = op_{\mathcal{A}_2}(f_{s_1}(t_1), \dots, f_{s_n}(t_n))
\end{aligned}$$

Si el homomorfismo es biyectivo se llama *isomorfismo*; si entre dos álgebras \mathcal{A}_1 y \mathcal{A}_2 se puede establecer un isomorfismo⁹, entonces decimos que \mathcal{A}_1 y \mathcal{A}_2 son *isomorfas* (ing., *isomorphic*) y lo denotamos por $\mathcal{A}_1 \approx \mathcal{A}_2$. Destaquemos que si $\mathcal{A}_1 \approx \mathcal{A}_2$ y $\mathcal{A}_2 \approx \mathcal{A}_3$, entonces $\mathcal{A}_1 \approx \mathcal{A}_3$.

Por ejemplo, sean la signatura NAT y el álgebra \mathcal{NAT} presentadas en el apartado anterior, y sea la NAT-álgebra $\mathcal{NAT}_2 = (S_{\mathcal{NAT}_2}, OP_{\mathcal{NAT}_2})$ de los naturales módulo 2 definida como:

$$S_{\mathcal{NAT}_2} = (nat_{\mathcal{NAT}_2}, bool_{\mathcal{NAT}_2}); \quad nat_{\mathcal{NAT}_2} \equiv \{0, 1\}, \quad bool_{\mathcal{NAT}_2} \equiv \mathcal{B}$$

⁹ Notemos que en los isomorfismos es indiferente el sentido de la función: si existe un isomorfismo de \mathcal{A} a \mathcal{A}' existe también un isomorfismo de \mathcal{A}' a \mathcal{A} .

$$\begin{aligned}
OP_{\mathcal{N}(\mathcal{A}T2)} &= \{cero_{\mathcal{N}(\mathcal{A}T2)}, suc_{\mathcal{N}(\mathcal{A}T2)}, suma_{\mathcal{N}(\mathcal{A}T2)}, ig_{\mathcal{N}(\mathcal{A}T2)}, cierto_{\mathcal{N}(\mathcal{A}T2)}, falso_{\mathcal{N}(\mathcal{A}T2)}\} \\
cero_{\mathcal{N}(\mathcal{A}T2)} &\equiv 0 \\
suc_{\mathcal{N}(\mathcal{A}T2)}(0) &\equiv 1, suc_{\mathcal{N}(\mathcal{A}T2)}(1) \equiv 0 \\
suma_{\mathcal{N}(\mathcal{A}T2)}(0, 0) &= suma_{\mathcal{N}(\mathcal{A}T2)}(1, 1) \equiv 0, suma_{\mathcal{N}(\mathcal{A}T2)}(1, 0) = suma_{\mathcal{N}(\mathcal{A}T2)}(0, 1) \equiv 1 \\
ig_{\mathcal{N}(\mathcal{A}T2)} &\equiv = \\
cierto_{\mathcal{N}(\mathcal{A}T2)} &\equiv C, falso_{\mathcal{N}(\mathcal{A}T2)} \equiv \mathcal{F}
\end{aligned}$$

Entonces, la aplicación $f: \mathcal{N}(\mathcal{A}T) \rightarrow \mathcal{N}(\mathcal{A}T2)$, definida por $f(n) = 0$, si n es par, y $f(n) = 1$, si n es impar, es un homomorfismo, dado que, por ejemplo:

$$\begin{aligned}
f(cero_{\mathcal{N}(\mathcal{A}T)}) &= f(0) = 0 = cero_{\mathcal{N}(\mathcal{A}T2)} \\
f(suc_{\mathcal{N}(\mathcal{A}T)}(2n)) &= f(2n+1) = 1 = suc_{\mathcal{N}(\mathcal{A}T2)}(0) = suc_{\mathcal{N}(\mathcal{A}T2)}(f(2n)) \\
f(suc_{\mathcal{N}(\mathcal{A}T)}(2n+1)) &= f(2(n+1)) = 0 = suc_{\mathcal{N}(\mathcal{A}T2)}(1) = suc_{\mathcal{N}(\mathcal{A}T2)}(f(2n+1))
\end{aligned}$$

y así para el resto de operaciones.

Una vez introducido el concepto de homomorfismo, puede definirse la evaluación de un término sobre su signatura: dados una signatura $SIG = (S, OP)$, una SIG -álgebra \mathcal{A} y un conjunto de variables V sobre SIG , definimos:

- La *función de evaluación de términos sin variables* dentro de \mathcal{A} , $eval_{\mathcal{A}}: T_{SIG} \rightarrow \mathcal{A}$, que es única dada una interpretación de los símbolos de la signatura dentro del álgebra, como un S -homomorfismo:

$$\begin{aligned}
\Diamond \forall c: c \in T_{SIG}: eval_{\mathcal{A}}(c) &\equiv c_{\mathcal{A}} \\
\Diamond \forall op(t_1, \dots, t_n): op(t_1, \dots, t_n) \in T_{SIG}: eval_{\mathcal{A}}(op(t_1, \dots, t_n)) &\equiv op_{\mathcal{A}}(eval_{\mathcal{A}}(t_1), \dots, eval_{\mathcal{A}}(t_n))
\end{aligned}$$

$eval_{\mathcal{A}}$ da la interpretación de los términos sin variables dentro de una álgebra \mathcal{A} .

- Una *función de asignación* de V dentro de \mathcal{A} , $as_{V, \mathcal{A}}: V \rightarrow \mathcal{A}$, como una S -aplicación. $as_{V, \mathcal{A}}$ da la interpretación de un conjunto de variables dentro del álgebra \mathcal{A} .
- La *función de evaluación de términos con variables* de V dentro del álgebra \mathcal{A} , $eval_{as_{V, \mathcal{A}}}: T_{SIG}(V) \rightarrow \mathcal{A}$, como un homomorfismo:

$$\begin{aligned}
\Diamond \forall v: v \in V: eval_{as_{V, \mathcal{A}}}(v) &\equiv as_{V, \mathcal{A}}(v) \\
\Diamond \forall c: c \in T_{SIG}(V): eval_{as_{V, \mathcal{A}}}(c) &\equiv eval_{\mathcal{A}}(c) \\
\Diamond \forall op(t_1, \dots, t_n): op(t_1, \dots, t_n) \in T_{SIG}(V): \\
eval_{as_{V, \mathcal{A}}}(op(t_1, \dots, t_n)) &\equiv op_{\mathcal{A}}(eval_{as_{V, \mathcal{A}}}(t_1), \dots, eval_{as_{V, \mathcal{A}}}(t_n))
\end{aligned}$$

Esta función es única, dada una interpretación $eval_{\mathcal{A}}$ de los símbolos de la signatura dentro del álgebra y dada una asignación $as_{V, \mathcal{A}}$ de variables de V dentro del álgebra.

Para simplificar subíndices, abreviaremos $eval_{as_{V, \mathcal{A}}}$ por $eval_{V, \mathcal{A}}$.

Por ejemplo, dadas la signatura NAT y la NAT -álgebra $\mathcal{N}AT$ introducidas anteriormente, la función de evaluación correspondiente es $eval_{\mathcal{N}AT} : T_{NAT} \rightarrow \mathcal{N}AT$ definida como:

$$\begin{aligned} eval_{\mathcal{N}AT}(cero_T) &\equiv 0, \quad eval_{\mathcal{N}AT}(suc_T(x)) \equiv eval_{\mathcal{N}AT}(x) + 1 \\ eval_{\mathcal{N}AT}(suma_T(x, y)) &\equiv eval_{\mathcal{N}AT}(x) + eval_{\mathcal{N}AT}(y) \\ eval_{\mathcal{N}AT}(ig_T(x, y)) &\equiv (eval_{\mathcal{N}AT}(x) = eval_{\mathcal{N}AT}(y)) \\ eval_{\mathcal{N}AT}(cierto_T) &\equiv C, \quad eval_{\mathcal{N}AT}(falso_T) \equiv \mathcal{F} \end{aligned}$$

Ahora se puede evaluar dentro del álgebra $\mathcal{N}AT$ el término $suc(suma(cero, suc(cero)))$ construido a partir de la signatura NAT :

$$\begin{aligned} eval_{\mathcal{N}AT}(suc(suma(cero, suc(cero)))) &= \\ eval_{\mathcal{N}AT}(suma(cero, suc(cero))) + 1 &= \\ (eval_{\mathcal{N}AT}(cero) + eval_{\mathcal{N}AT}(suc(cero))) + 1 &= \\ (0 + (eval_{\mathcal{N}AT}(cero) + 1)) + 1 &= (0 + (0 + 1)) + 1 = 2 \end{aligned}$$

Dado el conjunto de variables (de género *nat*) $V = \{x, y\}$, definimos una función de asignación dentro de $\mathcal{N}AT$ como $as_{V, \mathcal{N}AT}(x) = 3$ y $as_{V, \mathcal{N}AT}(y) = 4$. Dada esta función y la función de evaluación $eval_{\mathcal{N}AT}$, la evaluación $eval_{V, \mathcal{N}AT}$ dentro de $\mathcal{N}AT$ de $suc(suma(x, suc(y)))$ queda:

$$\begin{aligned} eval_{V, \mathcal{N}AT}(suc(suma(x, suc(y)))) &= eval_{V, \mathcal{N}AT}(suma(x, suc(y))) + 1 = \\ (eval_{V, \mathcal{N}AT}(x) + eval_{V, \mathcal{N}AT}(suc(y))) + 1 &= \\ (3 + (eval_{V, \mathcal{N}AT}(y) + 1)) + 1 &= (3 + (4 + 1)) + 1 = 9 \end{aligned}$$

1.2.4 Ecuaciones y especificaciones algebraicas

Dados los géneros y los símbolos de operación que forman la signatura de un tipo, es necesario introducir a continuación las propiedades que cumplen, de manera que se pueda determinar posteriormente el significado del TAD; para ello, se añaden a la signatura unas *ecuaciones* o *axiomas* (ing., *equation* o *axiom*), que forman la llamada *especificación algebraica* o *ecuacional* del TAD (ing., *algebraic* o *equational specification*).

Actualmente, la utilidad de las especificaciones formales es indiscutible dentro de los métodos modernos de desarrollo de *software*. Una especificación, ya sea ecuacional o de otra índole, no sólo proporciona un significado preciso a un tipo de datos asociándole un modelo matemático a partir de su descripción formal, sino que, como ya se ha dicho en la primera sección, responde a cualquier cuestión sobre el comportamiento observable del tipo sin necesidad de consultar el código, y por ello clarifica la misión de un tipo dentro de una aplicación. Además, las especificaciones ecuacionales pueden usarse como un primer prototipo de la aplicación siempre que cumplan determinadas condiciones (v. sección 1.7).

Una especificación algebraica $SPEC = (SIG_{SPEC}, E_{SPEC})$ se compone de:

- Una signatura $SIG_{SPEC} = (S, OP)$.
- Un conjunto E_{SPEC} de ecuaciones que expresan relaciones entre los símbolos de la signatura. Cada ecuación tiene la forma sintáctica $t = t'$, siendo t y t' términos con variables sobre SIG_{SPEC} ; se dice que t es la *parte izquierda* de la ecuación y t' la *parte derecha* (ing., *left-hand side* y *right-hand side*, respectivamente); definimos $Vars_{t=t'}$ como la unión de las variables de los dos términos. La ecuación $t = t'$ representa la fórmula universal de primer orden $\forall x_1 \forall x_2 \dots \forall x_n: t = t'$, siendo $\{x_1, x_2, \dots, x_n\} = Vars_{t=t'}$.

Para simplificar subíndices, de ahora en adelante abreviaremos SIG_{SPEC} por SIG y E_{SPEC} por E . Asimismo, denotaremos $SPEC = (SIG, E)$ por $SPEC = (S, OP, E)$ cuando sea preferible.

Las ecuaciones se incluyen en los universos de Merlí precedidas por la palabra clave "ecns" y, opcionalmente, por la declaración de sus variables; cuando sea necesario, a estos universos los llamaremos *universos de especificación* o también *universos de definición* para distinguirlos de otras clases de universos. Las ecuaciones se escriben en líneas diferentes o, de lo contrario, se separan con el carácter ';'. La fig. 1.9 presenta un universo de definición de los booleanos.

```

universo BOOL es
  tipo bool
  ops cierto, falso: → bool
  ¬_: bool → bool
  _∨_, _∧_: bool bool → bool
  ecns ∀b∈bool
    ¬ cierto = falso; ¬ falso = cierto
    b ∨ cierto = cierto; b ∨ falso = b
    b ∧ cierto = b; b ∧ falso = falso
  funiverso

```

Fig. 1.9: especificación algebraica para el TAD de los booleanos.

Las ecuaciones definen el comportamiento de las operaciones de la signatura; consideraremos que una operación está definida si las ecuaciones determinan su comportamiento respecto a todas las combinaciones posibles de valores (de los géneros correctos) que pueden tomar sus parámetros. Por ejemplo, por lo que se refiere a la suma en la especificación de los naturales de la fig. 1.10, se escribe su comportamiento respecto a cualquier par de naturales con dos ecuaciones: la ecuación 1 trata el caso de que el segundo operando sea el natural cero y la 2, que sea un natural positivo. Por lo que respecta a la especificación de la igualdad, se estudian los cuatro casos resultantes de considerar todas

las posibles combinaciones de dos naturales, que pueden ser o bien cero o bien positivos. En la sección siguiente se da un método de escritura de especificaciones basado en esta idea intuitiva.

```

universo NAT es
  tipo nat, bool
  ops cero: → nat
      suc: nat → nat
      suma: nat nat → nat
      ig: nat nat → bool
      cierto, falso: → bool
  ecns ∀n,m∈nat
      1) suma(n, cero) = n
      2) suma(n, suc(m)) = suc(suma(n, m))
      3) ig(cero, cero) = cierto
      4) ig(cero, suc(m)) = falso
      5) ig(suc(n), cero) = falso
      6) ig(suc(n), suc(m)) = ig(n, m)
funiverso

```

Fig. 1.10: especificación algebraica para el TAD de los naturales.

También es posible demostrar formalmente que la suma está correctamente definida de la siguiente forma. Dado que todo natural n puede representarse mediante la aplicación n veces de suc sobre $cero$, abreviadamente $suc^n(cero)$, basta con demostrar el enunciado: cualquier término de tipo nat que contenga un número arbitrario de sumas puede reducirse a un único término de la forma $suc^n(cero)$, interpretando las operaciones dentro del modelo de los naturales. Previamente, introducimos un lema auxiliar.

Lema. Todo término t de la forma $t = suc^X(cero) + suc^Y(cero)$ es equivalente (por las ecuaciones del tipo) a otro de la forma $suc^{X+Y}(cero)$.

Demostración. Por inducción sobre y .

$y = 0$. El término queda $t = suc^X(cero) + cero$, igual a $suc^X(cero)$ aplicando la ecuación 1.

$y = k$. Hipótesis de inducción: $t = suc^X(cero) + suc^k(cero)$ se transforma en $suc^{X+k}(cero)$.

$y = k+1$. Debe demostrarse que $t = suc^X(cero) + suc^{k+1}(cero)$ cumple el lema. t también puede escribirse como $t = suc^X(cero) + suc(suc^k(cero))$, y aplicando la ecuación 2 se transforma en $t = suc(suc^X(cero) + suc^k(cero))$, que es igual a $suc(suc^{X+k}(cero))$ aplicando la hipótesis de inducción, con lo que t ha podido transformarse finalmente en el término $suc^{X+k+1}(cero)$, que cumple el enunciado del lema.

Teorema. Todo término de tipo *nat* que contenga r operaciones de suma y s operaciones *suc* es equivalente por las ecuaciones a otro término de la forma $suc^s(cero)$.

Demostración. Por inducción sobre r .

$r = 0$. El término es de la forma $suc^s(cero)$ y cumple el enunciado.

$r = k$. Hipótesis de inducción: el término t de tipo *nat* con k operaciones de suma y s operaciones *suc* se transforma en $suc^s(cero)$.

$r = k+1$. Sea $\alpha = suc^x(cero) + suc^y(cero)$ un subtérmino de t que no contiene ninguna suma (siempre existirá al menos uno pues $r = k + 1 > 0$). Aplicando el lema sobre α se obtiene otro término $\beta = suc^{x+y}(cero)$ que elimina la (única) suma y conserva el número de apariciones de *suc*, y sustituyendo α por β dentro de t , resulta en un término con k operaciones de suma y s operaciones *suc*, siendo posible aplicar la hipótesis de inducción y así obtener el término $suc^s(cero)$.

En el apartado siguiente se estudia con mayor detalle el significado de esta demostración. Básicamente, se enuncia una biyección entre los términos de la forma $suc^n(cero)$ y los naturales, definida por $suc^n(cero) \leftrightarrow n$, y a continuación se generaliza la biyección a isomorfismo considerando no el conjunto de términos sino el álgebra de términos. En el caso general, será necesario una manipulación adicional que dará lugar a la denominada álgebra cociente de términos que también se define más adelante.

Por último, se introduce la noción de satisfacción de una ecuación. Sea e la ecuación $t_1 = t_2$, siendo t_1 y t_2 términos con variables sobre una signatura SIG , y sea $V = \text{Vars}_e$. Decimos que e es *válida* dentro de una SIG -álgebra \mathcal{A} (también se dice que \mathcal{A} *satisface* e) si, para toda función $as_{V,\mathcal{A}}: V \rightarrow \mathcal{A}$ de asignación, se cumple $\text{eval}_{V,\mathcal{A}}(t_1) =_{\mathcal{A}} \text{eval}_{V,\mathcal{A}}(t_2)$, siendo $=_{\mathcal{A}}$ la igualdad dentro del álgebra \mathcal{A} y $\text{eval}_{\mathcal{A}}$ la función de evaluación correspondiente. Por extensión, una SIG -álgebra \mathcal{A} *satisface* una especificación $SPEC = (SIG, E)$ (también se dice que \mathcal{A} es una *SPEC-álgebra*) si \mathcal{A} satisface todas las ecuaciones de E ; el conjunto de álgebras que satisfacen una especificación $SPEC$ se denota mediante Alg_{SPEC} .

1.2.5 Modelo inicial de una especificación

Dada una especificación ecuacional es imprescindible determinar con exactitud cuál o cuáles son los modelos por ella representados. Por ejemplo, dado el universo *NAT* de la fig. 1.10, está claro que el álgebra \mathcal{NAT} cumple sus ecuaciones (con la interpretación $\text{eval}_{\mathcal{NAT}}$ de los símbolos de la signatura), pero hay más modelos posibles: los enteros, o las matrices de naturales de dos dimensiones, donde se interpreta la operación *suc* como sumar el uno a todos sus elementos. Así, hay que establecer ciertos criterios que determinen claramente cuál o cuáles de estas álgebras son el modelo asociado a una especificación, y lo haremos con un ejemplo: investiguemos cuál de las álgebras siguientes:

$M_1 = (\mathcal{N}, 0, +1, +)$, naturales con cero, incremento en uno y suma.
 $M_2 = (\mathbb{Z}, 0, +1, +)$, enteros con cero, incremento en uno y suma.
 $M_3 = (\mathcal{M}_{2 \times 2}(\mathcal{N}), (0, 0; 0, 0), (+1, +1; +1, +1), +)$, matrices 2×2 de naturales con matriz cero, incremento en uno de todos los componentes y suma de matrices.
 $M_4 = (\mathcal{N}, 0, +1, \cdot)$, naturales con cero, incremento en uno y producto.
 $M_5 = (\mathcal{N}, 0, +1, +, -)$, naturales con cero, incremento en uno, suma y resta.
 $M_6 = (\mathcal{N}, 0, +1, \text{mod } 2)$, naturales con cero, incremento en uno, suma y resto de la división por 2.
 $M_7 = (\mathcal{N}, \mathbb{Z}, 0, +1, | \cdot |)$ naturales y enteros con cero, incremento en uno y valor absoluto.
 $M_8 = (\{*\}, f, g, h)$, modelo con un único elemento en su conjunto base, con las operaciones definidas por $f \equiv *, g(*) \equiv *$ y $h(*, *) \equiv *$.

es el modelo de la especificación $Y = (SY, EY)$:

universo Y es
tipo y
ops cero: $\rightarrow y$
suc: $y \rightarrow y$
op: $y y \rightarrow y$
ecns $\forall n, m \in y$
1) $op(n, \text{cero}) = n$; 2) $op(n, suc(m)) = suc(op(n, m))$
funiverso

Dado que buscamos el modelo asociado a una especificación, es imprescindible que este modelo presente unos conjuntos de base y unas operaciones que respondan a la plantilla determinada por la signatura. Así, se puede hacer una primera criba de las álgebras introducidas, porque hay algunas que no son SY-álgebras: M_5 tiene cuatro símbolos de operación (tendríamos que olvidar la suma o la resta para obtener una SY-álgebra), M_6 tiene tres, pero las aridades no coinciden (no hay ninguna operación que se pueda asociar a op), y M_7 define dos géneros (tendríamos que olvidar uno para obtener una SY-álgebra). El resto de álgebras son SY-álgebras y la interpretación de los símbolos de la signatura es inmediata en cada caso, como también la función de evaluación resultante (v. fig. 1.11).

Por lo que concierne a la satisfacción de las ecuaciones en las SY-álgebras, es fácil ver que M_1 , M_2 , M_3 y M_8 son Y-álgebras, pero que M_4 no lo es, porque no cumple la propiedad $m \cdot 0 = 0$. Por ello, puede descartarse M_4 como posible modelo, dado que ni siquiera cumple las ecuaciones; quedan pues cuatro candidatos. Para determinar cuál o cuáles son los buenos, construiremos una nueva SY-álgebra a partir del álgebra de términos, incorporando la información que proporcionan las ecuaciones, y estudiaremos la isomorfía entre ella y las álgebras. Concretamente, consideramos que dos términos son equivalentes si y sólo si se deduce su igualdad sintáctica manipulándolos con las ecuaciones de la especificación; el resultado es la llamada álgebra cociente de términos, que se introduce a continuación.

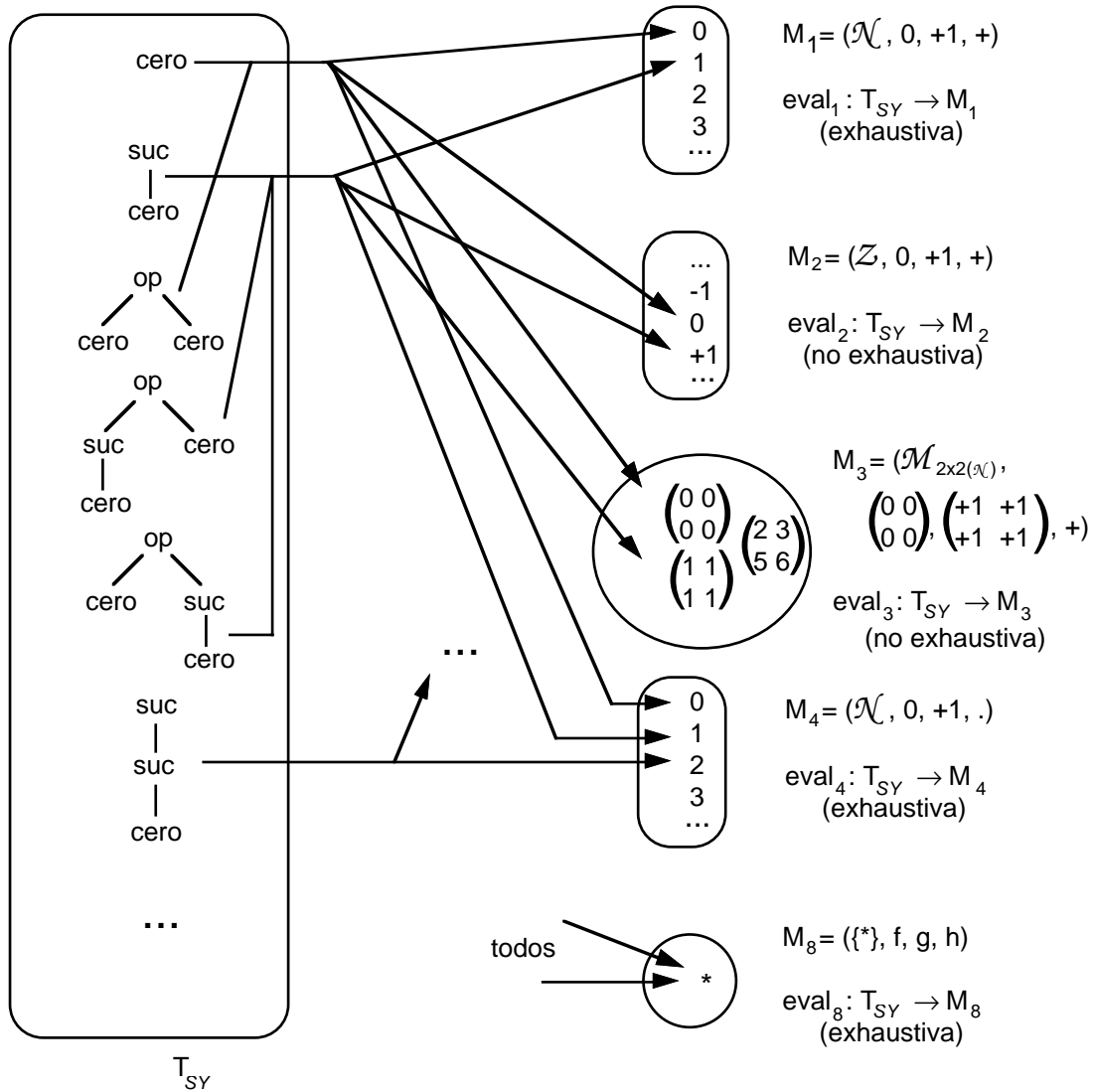


Fig. 1.11: algunas SY-álgebras y las funciones de evaluación de T_{SY} dentro de ellas.

Dada la especificación $SPEC = (SIG, E)$, siendo $SIG = (S, OP)$, se define la *congruencia* \equiv_E inducida por las ecuaciones de E como la menor relación que cumple:

- \equiv_E es una relación de equivalencia.
- Están dentro de una misma clase de equivalencia todos aquellos términos que puede demostrarse por las ecuaciones que son iguales:

$$\forall e \equiv t_1 = t_2: e \in E: \forall as_{Vars_e, T_{SIG}}: Vars_e \rightarrow T_{SIG}: eval_{Vars_e, T_{SIG}}(t_1) \equiv_E eval_{Vars_e, T_{SIG}}(t_2).$$

- Cualquier operación aplicada sobre diversas parejas de términos congruentes da como resultado dos términos igualmente congruentes:

$$\forall op: op \in OP_{s_1 \dots s_n \rightarrow s}: \forall t_1, t'_1 \in T_{SIG, s_1} \dots \forall t_n, t'_n \in T_{SIG, s_n}: \\ t_1 \equiv_E t'_1 \wedge \dots \wedge t_n \equiv_E t'_n \Rightarrow op(t_1, \dots, t_n) \equiv_E op(t'_1, \dots, t'_n).$$

Entonces se define el *álgebra cociente de términos* (ing., *quotient-term algebra*), denotada por T_{SPEC} , como el resultado de particionar T_{SIG} usando \equiv_E , $T_{SPEC} \equiv T_{SIG} / \equiv_E$; concretamente, $T_{SPEC} = (S_Q, OP_Q)$, siendo S_Q un S -conjunto y OP_Q un $\langle S^* \times S \rangle$ -conjunto, definidos:

- $\forall s: s \in S: s_Q \in S_Q$, siendo $s_Q \equiv \{ [t] / t \in T_{SIG, s} \}$, siendo $[t] \equiv \{ t' \in T_{SIG} / t' \equiv_E t \}$
- $\forall c: c \in OP_{\rightarrow s}: c_Q \in OP_Q$, donde $c_Q \equiv [c]$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s}: op_Q: s_{1Q} \times \dots \times s_{nQ} \rightarrow s_Q \in OP_Q$, donde $op_Q([t_1], \dots, [t_n]) \equiv [op(t_1, \dots, t_n)]$

Los elementos de los conjuntos de base de esta álgebra son las clases de equivalencia resultado de particionar T_{SPEC} usando \equiv_E , compuestas por términos sin variables cuya igualdad es deducible por las ecuaciones. En la fig. 1.12 se muestra el álgebra cociente de términos T_Y para la especificación Y ; las correspondencias (1), (2) y (3) se calculan:

- (1) $op(cero, cero) = cero$, aplicando la ecuación 1 con $n = cero$.
- (2) $op(suc(cero), cero) = suc(cero)$, aplicando la ecuación 1 con $n = suc(cero)$.
- (3) $op(cero, suc(cero)) = suc(op(cero, cero)) = suc(cero)$, aplicando la ecuación 2 con $m = n = cero$ y después la ecuación 1 con $n = cero$.

La importancia del álgebra cociente de términos radica en que, si consideramos cada una de las clases componentes como un objeto del TAD que se quiere especificar, entonces T_{SPEC} es el modelo. Para ser más concretos, definimos como modelo del tipo cualquier álgebra isomorfa a T_{SPEC} ; en el ejemplo, M_1 es isomorfa a T_Y , mientras que M_2 , M_3 y M_8 no lo son (como queda claro en la fig. 1.12), y por ello se puede decir que el modelo del TAD son los naturales con operaciones cero, incremento en uno y suma. La isomorfía establece la insensibilidad a los cambios de nombre de los símbolos de la signatura: es lo mismo escribir $[cero]$ que 0, $[suc(cero)]$ que 1, etc., siempre que las propiedades que cumplan los símbolos sean las mismas.

El álgebra cociente de términos asociada a la especificación $SPEC$ cumple ciertas propiedades:

- T_{SPEC} es *generada*: todos sus valores son generados por las operaciones del tipo (no contiene datos inalcanzables desde las operaciones).
- T_{SPEC} es *típica*: dos términos están dentro de la misma clase si y sólo si por las ecuaciones se demuestra su igualdad (se dice que T_{SPEC} no *confunde* los términos).
- T_{SPEC} es *inicial* dentro de la clase de $SPEC$ -álgebras: para toda $SPEC$ -álgebra \mathcal{A} , se puede encontrar un único homomorfismo de T_{SPEC} en \mathcal{A} (v. fig. 1.13).

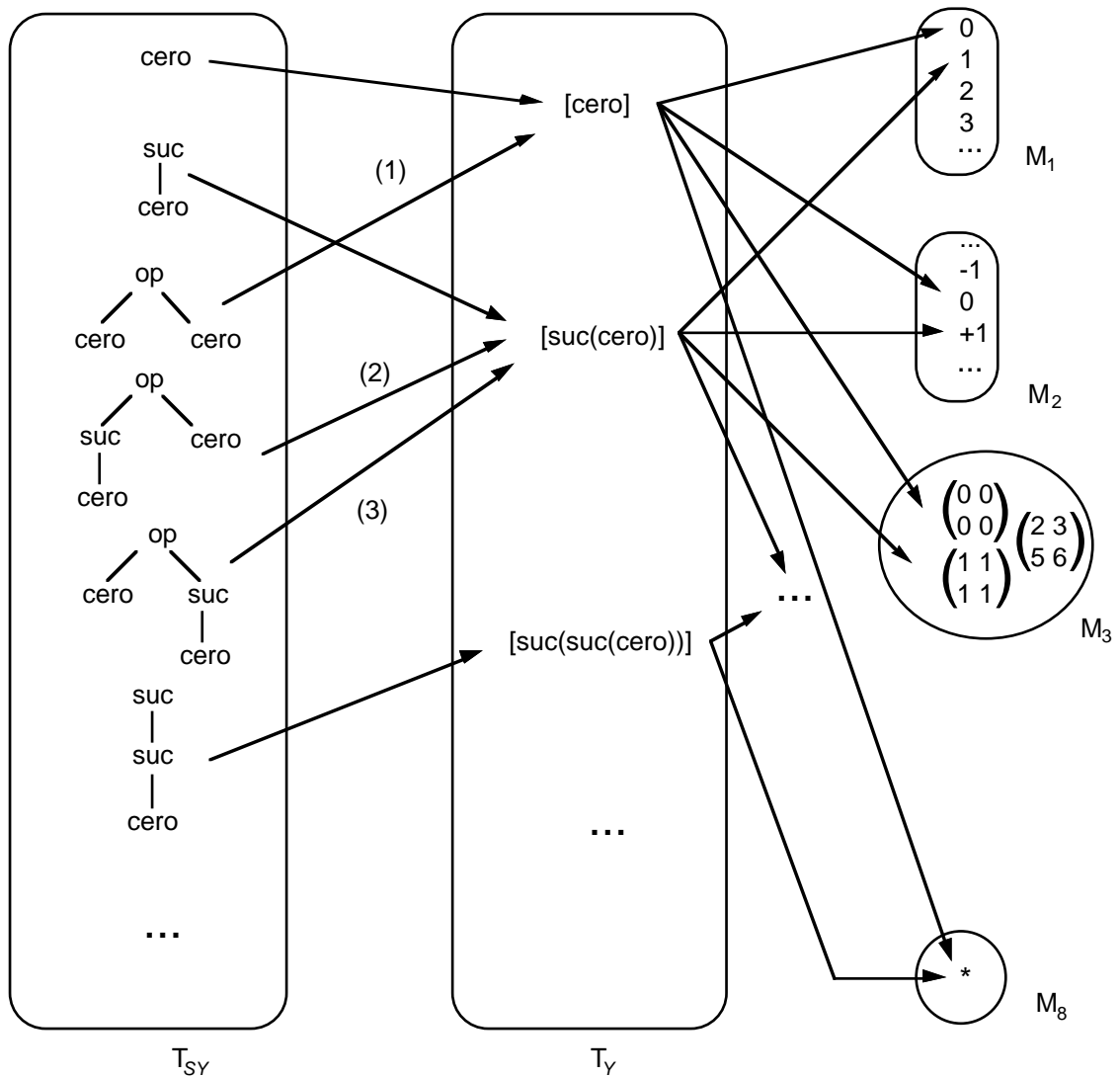


Fig. 1.12: el álgebra cociente de términos T_Y para la especificación Y .

- Las diversas álgebras isomorfas al álgebra de términos también pueden tomarse como modelos; por eso se dice que el modelo no es un álgebra concreta, sino una *clase de isomorfía* (la clase de todas las álgebras isomorfas a T_{SPEC} ; v. fig. 1.13). En concreto, toda *SPEC*-álgebra generada y típica es isomorfa a T_{SPEC} .
- Los elementos de T_{SPEC} están caracterizados por un término llamado *representante canónico* de la clase; además, se pueden escoger los representantes canónicos de manera que, considerados como términos, todos estén formados por combinaciones de un subconjunto de operaciones de la signatura (v. sección siguiente).

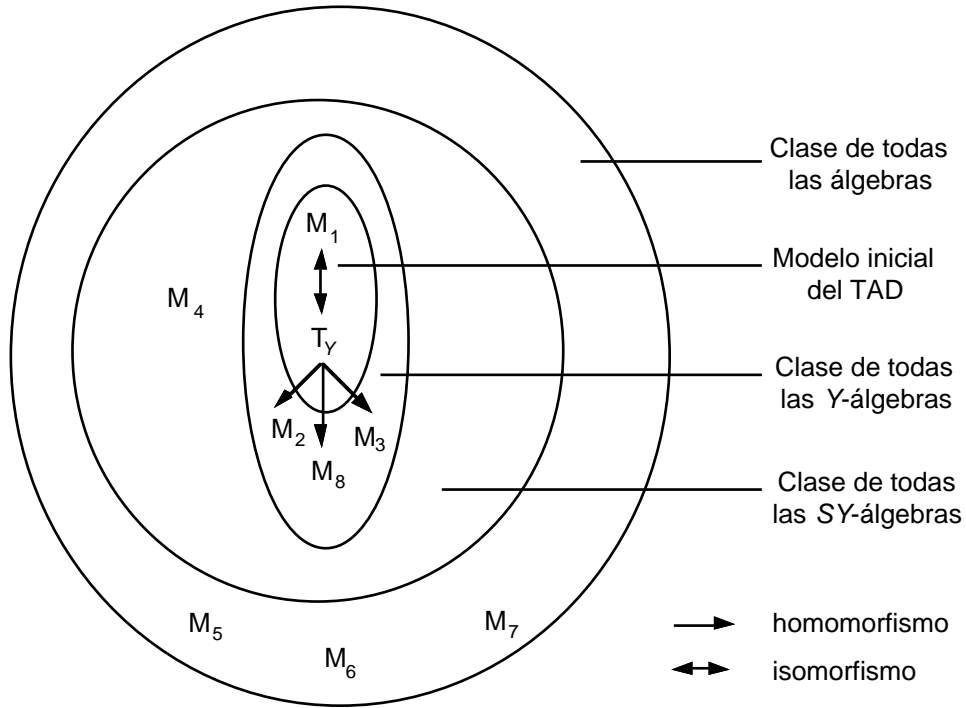


Fig. 1.13: clasificación de todas las álgebras existentes respecto a la especificación Y .

En la fig. 1.14 se muestra el álgebra cociente de términos T_{ENTERO} asociada a la especificación $ENTERO$. Se podría demostrar que T_{ENTERO} es isomorfa a los enteros $(\mathbb{Z}, 0, +1, -1, +)$ tomando $entero_Q \approx \mathbb{Z}$, $[suc^n(cero)] \approx n$, $[pred^n(cero)] \approx -n$, $[cero] \approx 0$ y la asociación intuitiva de los símbolos de operación con las operaciones de $(\mathbb{Z}, 0, +1, -1, +)$.

universo ENTERO es

tipo entero

ops cero: \rightarrow entero

suc, pred: entero \rightarrow entero

suma: entero entero \rightarrow entero

ecns $\forall n, m \in \text{entero}$

suc(pred(m)) = m

pred(suc(m)) = m

suma(n, cero) = n

suma(n, suc(m)) = suc(suma(n, m))

suma(n, pred(m)) = pred(suma(n, m))

funiverso

$T_{ENTERO} = ((entero_Q), \{cero_Q, suc_Q, pred_Q, suma_Q\})$

$entero_Q \equiv \{ [suc^n(cero)] / n \geq 1 \} \cup \{ [cero] \} \cup \{ [pred^n(cero)] / n \geq 1 \}$

$cero_Q \equiv [cero]$

$suc_Q([suc^n(cero)]) \equiv [suc^{n+1}(cero)], n \geq 0$

$suc_Q([pred^n(cero)]) \equiv [pred^{n-1}(cero)], n \geq 1$

$pred_Q([suc^n(cero)]) \equiv [suc^{n-1}(cero)], n \geq 1$

$pred_Q([pred^n(cero)]) \equiv [pred^{n+1}(cero)], n \geq 0$

$suma_Q([suc^n(cero)], [suc^m(cero)]) \equiv$

$\equiv [suc^{n+m}(cero)], n, m \geq 0$

$suma_Q([suc^n(cero)], [pred^m(cero)]) \equiv$

$\equiv [suc^{n-m}(cero)], n \geq m \geq 1 \dots \text{etc.}$

Fig. 1.14: un álgebra cociente de términos para los enteros.

1.2.6 Otros modelos posibles

En el apartado anterior se ha asociado como modelo (semántica) de un TAD la clase isomorfa al álgebra cociente de términos. Es lo que se conoce como *semántica inicial* (ing., *initial semantics*) de un TAD (llamada así porque hay un homomorfismo único de T_{SPEC} a todas las demás álgebras de Alg_{SPEC}), que se caracteriza porque todos los valores son alcanzables a partir de las operaciones y porque dos términos son iguales si y sólo si puede deducirse de las ecuaciones. En el resto de este libro se interpretará la especificación de un TAD con este significado; no obstante, conviene saber que el enfoque inicial no es el único y en este apartado mencionamos otras propuestas existentes.

Una crítica que recibe frecuentemente el modelo inicial es su bajo grado de abstracción. Por ejemplo, consideremos la especificación de la fig. 1.15, de la que se pretende que represente los conjuntos de naturales y dos términos sobre su signatura, $t_1 = \text{añade}(\text{añade}(\emptyset, \text{cero}), \text{cero})$ y $t_2 = \text{añade}(\emptyset, \text{cero})$.

```

universo CJT_NAT es
  tipo cjt, nat, bool
  ops  $\emptyset$ :  $\rightarrow$  cjt
      añade: cjt nat  $\rightarrow$  cjt
       $\_ \in \_$ : nat cjt  $\rightarrow$  bool
      cero:  $\rightarrow$  nat
      suc: nat  $\rightarrow$  nat
      ig: nat nat  $\rightarrow$  bool
      cierto, falso:  $\rightarrow$  bool
       $\_ \vee \_$ : bool bool  $\rightarrow$  bool
  ecns ... { especificación de  $\vee$  e  $ig$  }
       $n \in \emptyset = \text{falso}$ 
       $n_2 \in \text{añade}(s, n_1) = ig(n_1, n_2) \vee n_2 \in s$ 
  funiverso

```

Fig. 1.15: ¿una especificación para los conjuntos?

Según el enfoque inicial, t_1 y t_2 denotan valores diferentes porque no puede deducirse por las ecuaciones que sean iguales; ahora bien, ambos términos se comportan exactamente igual, porque todo natural que pertenece al conjunto representado por t_1 también pertenece al conjunto representado por t_2 . Podríamos interpretar, pues, que t_1 y t_2 son el mismo valor si rechazamos el principio de tipicidad del modelo y afirmamos que dos términos son iguales, salvo que se pueda deducir por las ecuaciones que son diferentes. Esta es la *semántica final* (ing., *final semantics*) de un TAD (llamada así porque hay un homomorfismo único desde

todas las demás álgebras de Alg_{SPEC} hacia el modelo), que es igualmente una clase de isomorfía y que también es generada, como la semántica inicial.

Notemos que la semántica inicial de la especificación de la fig. 1.15 es \mathcal{N}^* (las secuencias de naturales) y la semántica final es $\mathcal{P}(\mathcal{N})$ (los conjuntos de naturales). Para conseguir que $\mathcal{P}(\mathcal{N})$ sea el modelo inicial, hay que añadir las ecuaciones:

$$\begin{aligned}\text{añade}(\text{añade}(s, n_1), n_2) &= \text{añade}(\text{añade}(s, n_2), n_1) \\ \text{añade}(\text{añade}(s, n), n) &= \text{añade}(s, n)\end{aligned}$$

Es decir, la semántica inicial, a veces, obliga a sobreespecificar los universos introduciendo ecuaciones que no son importantes para el uso del universo dentro de un programa. Este problema aparecerá con cierta frecuencia en las especificaciones de TAD que nos encontraremos en el libro.

Para considerar la igualdad dentro de la semántica final (es decir, la congruencia inducida por las ecuaciones), se hacen *experimentos* sobre unos géneros especiales llamados *géneros observables*. A tal efecto, se cambia la definición de una especificación *SPEC* y se define $\text{SPEC} = (V, S, OP, E)$, siendo V (subconjunto de S) los géneros observables. Intuitivamente, dos términos de un género no observable son iguales si, al aplicarles cualquier operación que devuelva un valor observable, el resultado es el mismo. En el ejemplo de los conjuntos, si definimos *bool* y *nat* como los géneros observables, la única operación sobre términos no observables que da resultado observable es la de pertenencia, por lo que dos términos no observables t_1 y t_2 son iguales dentro de la semántica final si $\forall n: n \in \text{nat}: n \in t_1 = n \in t_2$.

Una tercera opción es la *semántica de comportamiento* (ing., *behavioural semantics*), que tiene un principio de funcionamiento parecido a la semántica final obviando la restricción de isomorfía de la clase de modelos. Por ejemplo, en el caso de la especificación *CJT_NAT*, se toman como modelo \mathcal{N}^* y $\mathcal{P}(\mathcal{N})$ a la vez y, además, todas las álgebras con el mismo comportamiento observable, como los conjuntos con repeticiones y las secuencias sin repeticiones. Todos estos modelos forman parte del álgebra de comportamiento por *CJT_NAT*, a pesar de que sus conjuntos base no son isomorfos entre sí.

Finalmente, también se puede considerar toda la clase Alg_{SPEC} como la clase de modelos; es la llamada *semántica laxa* (ing., *loose semantics*). En este caso, el modelo se toma como punto de partida y se va restringiendo poco a poco (hay diversos tipos de restricciones) hasta llegar a un modelo ya aceptable como resultado. La ventaja sobre otros tipos de semánticas es que se adapta mejor al concepto de desarrollo de aplicaciones. Por ejemplo, supongamos que la especificación de los conjuntos se añade una operación para elegir un elemento cualquiera del conjunto, *elige*: $\text{cjt} \rightarrow \text{nat}$; si no importa cuál es el elemento concreto elegido, se puede añadir la ecuación $\text{elige}(s) \in s = \text{cierto}$, que no impone ninguna estrategia; en algún momento, no obstante, debe restringirse el modelo añadiendo las ecuaciones necesarias para obtener un tipo de comportamiento conocido, por ejemplo, $\text{elige}(s) = \text{mínimo}(s)$.

1.3 Construcción sistemática de especificaciones

En la sección anterior hemos estudiado las características de una especificación dada y hemos encontrado su modelo *a posteriori*. Ahora bien, en el proceso de desarrollo de *software* la situación acostumbra a ser la contraria: a partir de un TAD que formará parte de un programa, del que se conoce su comportamiento (quizás del todo, quizás sólo se tiene una idea intuitiva), el problema consiste en encontrar una especificación que lo represente y es por ello que en esta sección se estudia un método general para la construcción de especificaciones, basado en una clasificación previa de las operaciones del TAD. Antes introducimos en el primer apartado un concepto útil para escribir especificaciones con mayor comodidad, que se estudiará en profundidad en la sección 1.6.

1.3.1 Introducción al uso de especificaciones

Hasta ahora, al especificar un TAD no hay manera de aprovechar la especificación de otro tipo que se pudiera necesitar, aunque ya esté definida en otro universo; es un ejemplo conocido la especificación de los naturales con operación de igualdad, que precisa repetir la especificación de los booleanos. Es evidente que se necesita un medio para evitar esta redundancia; por tanto, se incluye una nueva cláusula en el lenguaje que permite usar especificaciones ya existentes desde un universo cualquiera. En la fig. 1.16 se muestra una nueva especificación de los naturales que "usa" el universo *BOOL* de los booleanos y, consecuentemente, puede utilizar todos los símbolos en él definidos.

```

universo NAT es
  usa BOOL
  tipo nat
  ops cero: → nat
      suc: nat → nat
      suma: nat nat → nat
      ig: nat nat → bool
  ecns ∀n,m∈nat
      1) suma(n, cero) = n
      2) suma(n, suc(m)) = suc(suma(n, m))
      3) ig(cero, cero) = cierto
      4) ig(cero, suc(m)) = falso
      5) ig(suc(n), cero) = falso
      6) ig(suc(n), suc(m)) = ig(n, m)
funiverso

```

Fig. 1.16: un universo para los naturales con igualdad usando los booleanos.

1.3.2 Clasificación de las operaciones de una especificación

Como paso previo a la formulación de un método general de construcción de especificaciones, se precisa clasificar las operaciones de la signatura de una especificación respecto a cada género que en ella aparece: dada una especificación $SPEC = (S, OP, E)$ y un género $s \in S$, se define el conjunto de operaciones *constructoras* de OP respecto a s , $constr_{OP_s}$, como el conjunto de operaciones de OP que devuelven un valor de género s ; y el conjunto de operaciones *consultoras* de OP respecto a s , $consul_{OP_s}$, como el conjunto de operaciones de OP que devuelven un valor de género diferente de s :

$$constr_{OP_s} \equiv \{op \in OP_{w \rightarrow s}\}.$$

$$consul_{OP_s} \equiv \{op \in OP_{s_1 \dots s_n \rightarrow s'} / s \neq s' \wedge \exists i : 1 \leq i \leq n : s_i \neq s\}.$$

Dentro de las operaciones constructoras, destaca especialmente el conjunto de operaciones *constructoras generadoras*, que es un subconjunto mínimo de las operaciones constructoras que permite generar, por aplicaciones sucesivas, los representantes canónicos de las clases de T_{SPEC} (es decir, todos los valores del TAD que queremos especificar); en el caso general, puede haber más de un subconjunto de constructoras generadoras, de los que hay que escoger uno. Las constructoras que no forman parte del conjunto de constructoras generadoras escogidas se llaman (*constructoras*) *modificadoras*. Notaremos los dos conjuntos con gen_{OP_s} y $modif_{OP_s}$. El conjunto gen_{OP_s} es *puro* si no hay relaciones entre las operaciones que lo forman (dicho de otra manera, todo par de términos diferentes formados sólo por constructoras generadoras denota valores diferentes) o *impuro* si hay relaciones; en este último caso, a las ecuaciones que expresan las relaciones entre las operaciones constructoras generadoras las llamamos *impurificadoras*.

Por ejemplo, dado el universo *BOOL* de la fig. 1.9, el conjunto de consultoras es vacío porque todas son constructoras; en lo que se refiere a éstas, parece lógico elegir como representantes canónicos los términos *cierto* y *falso* y entonces quedarán los conjuntos $gen_{OP_{bool}} = \{cierto, falso\}$ y $modif_{OP_{bool}} = \{\neg, \vee, \wedge\}$. Ahora bien, también es lícito coger como representantes canónicos los términos *cierto* y $\neg cierto$, siendo $gen_{OP_{bool}} = \{cierto, \neg\}$; la propiedad $\neg \neg x = x$ determina la impureza de este conjunto de constructoras generadoras.

No es normal que haya un único género en el universo; por ejemplo, en la especificación de los naturales con igualdad de la fig. 1.16 aparecen *nat* y *bool*. No obstante, con vistas a la formulación de un método general de especificación (en el siguiente apartado), solamente es necesario clasificar las operaciones respecto a los nuevos géneros que se introducen y no respecto a los que se usan; si sólo se define un nuevo género, como es habitual, se hablará de operaciones constructoras o consultoras sin explicitar respecto a qué género. Volviendo a la especificación de los naturales $NAT = (S, OP, E)$ de la fig. 1.16 y tomando como representantes canónicos los términos (de género *nat*) $suc^n(cero)$, $n \geq 0$, la clasificación de las operaciones es: $gen_{OP} = \{cero, suc\}$, $modif_{OP} = \{suma\}$ y $consul_{OP} = \{ig\}$.

1.3.3 Método general de construcción de especificaciones

Para escribir la especificación de un TAD no se dispone de un método exacto, sino que hay que fiarse de la experiencia y la intuición a partes iguales; a veces, una especificación no acaba de resolverse si no se tiene una idea feliz. A continuación se expone un método que da resultados satisfactorios en un elevado número de casos y que consta de tres pasos.

- Elección de un conjunto de operaciones como constructoras generadoras. Se buscan las operaciones que son suficientes para generar todos los valores del álgebra cociente de términos del tipo (es decir, para formar los representantes canónicos de las clases). Como ya se ha dicho, puede haber más de uno; en este caso, se puede escoger el conjunto menos impuro o bien el mínimo (criterios que muy frecuentemente conducen al mismo conjunto).
- Aserción de las relaciones entre las constructoras generadoras. Se escriben las ecuaciones impurificadoras del tipo. Una posibilidad para conseguir estas ecuaciones consiste en pensar en la forma que ha de tomar el representante canónico de la clase y de qué manera se puede llegar a él.
- Especificación del resto de operaciones, una a una, respecto a las constructoras generadoras (es decir, definición de los efectos de aplicar las operaciones sobre términos formados exclusivamente por constructoras generadoras, pero sin suponer que estos términos son representantes canónicos). Recordemos que el objetivo final es especificar cada operación respecto a todas las combinaciones posibles de valores a los que se puede aplicar, y una manera sencilla de conseguir este objetivo consiste en estudiar los efectos de la aplicación de la operación sobre las constructoras generadoras, que permiten generar por aplicaciones sucesivas todos los valores del género.

En este último paso se debe ser especialmente cuidadoso al especificar las operaciones consultoras para asegurar dos propiedades denominadas *consistencia* y *completitud suficiente*: si se ponen ecuaciones de más, se pueden igualar términos que han de estar en clases de equivalencia diferentes del género correspondiente, mientras que si se ponen de menos, se puede generar un número indeterminado de términos (potencialmente infinitos) incongruentes con los representantes de las clases existentes hasta aquel momento y que dan lugar a nuevas clases de equivalencia; en cualquiera de los dos casos, se está modificando incorrectamente la semántica del tipo afectado.

Como ejemplo, se presenta una especificación para el tipo *cjt* de los conjuntos de naturales con operaciones $\emptyset: \rightarrow \text{cjt}$, *añade*: $\text{cjt nat} \rightarrow \text{cjt}$, $_ \cup _: \text{cjt cjt} \rightarrow \text{cjt}$ y $_ \in _: \text{nat cjt} \rightarrow \text{bool}$. El conjunto $\{\emptyset, \text{añade}\}$ juega el papel de conjunto de constructoras generadoras, porque las operaciones permiten construir todos los valores posibles del tipo (la constructora $_ \cup _$ en particular no es necesaria). La forma general del representante canónico de las clases es *añade*(*añade*(...(*añade*(\emptyset , n_1), ..., n_k), tomando por ejemplo $n_1 < \dots < n_k$.

Para modelizar realmente los conjuntos, es necesario incluir dos ecuaciones impurificadoras que expresen que el orden de añadir los naturales al conjunto no es significativo y que dentro del conjunto no hay elementos repetidos:

$$\begin{aligned} \text{añade}(\text{añade}(s, n_1), n_2) &= \text{añade}(\text{añade}(s, n_2), n_1) \\ \text{añade}(\text{añade}(s, n), n) &= \text{añade}(s, n) \end{aligned}$$

No hay ningún otro tipo de relación entre las constructoras generadoras; mediante estas dos ecuaciones todo término formado exclusivamente por constructoras generadoras puede convertirse, eventualmente, en su representante canónico.

Por lo que respecta al resto de operaciones, la aplicación del método da como resultado:

$$\begin{aligned} n \in \emptyset &= \text{falso} & n_1 \in \text{añade}(s, n_2) &= \text{ig}(n_1, n_2) \vee (n_1 \in s) \\ \emptyset \cup s &= s & \text{añade}(s_1, n) \cup s_2 &= \text{añade}(s_1 \cup s_2, n) \end{aligned}$$

Notemos que hay parámetros que se dejan como una variable, en lugar de descomponerlos en las diversas formas que pueden tomar como combinación de constructoras generadoras, por lo que el parámetro toma igualmente todos los valores posibles del género.

A partir de este enfoque, se puede decir que especificar es dar unas reglas para pasar de un término cualquiera a su representante canónico y así evaluar cualquier expresión sobre un objeto del TAD correspondiente. En la sección 1.7 se insiste en este planteamiento.

1.4 Ecuaciones condicionales, símbolos auxiliares y errores

Introducimos en esta sección tres conceptos adicionales necesarios para poder construir especificaciones que resuelvan problemas no triviales: las ecuaciones condicionales, los tipos y las operaciones auxiliares, y el mecanismo de tratamiento de errores.

1.4.1 Ecuaciones condicionales

Hasta ahora, las ecuaciones expresan propiedades que se cumplen incondicionalmente; a veces, no obstante, un axioma se cumple sólo en determinadas condiciones. Esto sucede, por ejemplo, al añadir la operación *saca*: $cjt\ nat \rightarrow cjt$ a la especificación de los conjuntos de la sección anterior; su especificación siguiendo el método da como resultado:

$$\begin{aligned} \text{saca}(\emptyset, n) &= \emptyset \\ \text{saca}(\text{añade}(s, n_1), n_2) &= ?? \end{aligned}$$

donde la parte derecha de la segunda ecuación depende de si n_1 es igual a n_2 y por ello resultan infinitas ecuaciones:

$$\begin{aligned} \text{saca}(\text{añade}(s, \text{cero}), \text{cero}) &= \text{saca}(s, \text{cero}) \\ \text{saca}(\text{añade}(s, \text{suc}(\text{cero})), \text{cero}) &= \text{añade}(\text{suc}(\text{cero}), \text{saca}(s, \text{cero})) \\ \text{saca}(\text{añade}(s, \text{suc}(\text{cero})), \text{suc}(\text{cero})) &= \text{saca}(s, \text{suc}(\text{cero})) \dots \text{etc.} \end{aligned}$$

El problema se puede solucionar introduciendo *ecuaciones condicionales* (ing., *conditional equation*):

- 1) $\text{saca}(\emptyset, n) = \emptyset$
- 2) $[\text{ig}(n_1, n_2) = \text{cierto}] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 3) $[\text{ig}(n_1, n_2) = \text{falso}] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{añade}(\text{saca}(s, n_2), n_1)$

(Notemos que la parte derecha de la ecuación 2 no puede ser simplemente s , porque no se puede asegurar que dentro de s no haya más apariciones de n_2 -es decir, no se puede asegurar que el término sea canónico. Es más, incluir esta ecuación en sustitución de 2 provocaría inconsistencias en el tipo, pues se podrían deducir igualdades incorrectas, como $\text{añade}(\emptyset, 1) = \text{saca}(\text{añade}(\text{añade}(\emptyset, 1), 1), 1) = \text{saca}(\text{añade}(\emptyset, 1), 1) = \emptyset$.)

Podemos decir, pues, que una ecuación condicional es equivalente a un número infinito de ecuaciones no condicionales, así como una ecuación con variables es equivalente a un número infinito de ecuaciones sin variables. La sintaxis de Merlí encierra la condición entre corchetes a la izquierda de la ecuación, también en forma de ecuación¹⁰: $[t_0 = t_0'] \Rightarrow t_1 = t_1'$. La ecuación $t_0 = t_0'$ se denomina *premisa* y $t_1 = t_1'$ se denomina *conclusión*. Para abreviar, las condiciones de la forma $[t = \text{cierto}]$ o $[t = \text{falso}]$ las escribiremos simplemente $[t]$ o $[\neg t]$, respectivamente; también para abreviar, varias ecuaciones condicionales con idéntica premisa se pueden agrupar en una sola, separando las conclusiones mediante comas. Dada una SPEC-álgebra \mathcal{A} con la correspondiente función de evaluación $\text{eval}_{\mathcal{A}}$ y una igualdad $=_{\mathcal{A}}$ y, siendo V la unión de las variables de la premisa, diremos que \mathcal{A} satisface una ecuación condicional si:

$$\forall \text{as}_{V, \mathcal{A}}: V \rightarrow \mathcal{A} : \{ \text{eval}_{V, \mathcal{A}}(t_0) =_{\mathcal{A}} \text{eval}_{V, \mathcal{A}}(t_0') \} \Rightarrow \text{eval}_{V, \mathcal{A}}(t_1) =_{\mathcal{A}} \text{eval}_{V, \mathcal{A}}(t_1').$$

Es necesario destacar un par de cuestiones importantes referentes a las ecuaciones condicionales. Primero, al especificar una operación no constructora generadora usando ecuaciones condicionales hay que asegurarse de que, para una misma parte izquierda, se cubren todos los casos posibles; en otras palabras, para toda asignación de variables de la ecuación debe haber como mínimo una condición que se cumpla. Segundo, notemos que la operación de igualdad sobre los valores de los tipos es necesaria para comprobar la desigualdad pero no la igualdad; así, las tres ecuaciones siguientes son equivalentes:

- 2a) $[\text{ig}(n_1, n_2)] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 2b) $[n_1 = n_2] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 2c) $\text{saca}(\text{añade}(s, n), n) = \text{saca}(s, n)$

¹⁰ En el marco de la semántica inicial, la premisa no se define normalmente como una única ecuación, sino como la conjunción de varias ecuaciones. No obstante, la forma simplificada aquí adoptada cubre todos los ejemplos que aparecen en el texto y, en realidad, no presenta carencia alguna.

Esto es debido a la existencia de la deducción ecuacional, introducida en la sección 1.7. Lo que nunca se puede suponer es que dos variables diferentes n_1 y n_2 denotan forzosamente dos valores diferentes, porque siempre puede ocurrir que n_1 y n_2 valgan lo mismo (recordemos que la interpretación de una ecuación cuantifica universalmente sus variables).

La redefinición de la congruencia \equiv_E con ecuaciones condicionales queda como sigue:

- Se define \equiv_0 como aquella congruencia en la que todo término del álgebra de términos forma una clase de equivalencia por sí mismo.

- Dada \equiv_i , se define \equiv_{i+1} de la siguiente manera:

$$t \equiv_{i+1} t' \Leftrightarrow t \equiv_i t' \vee$$

$$\exists e \in E, e = [t_0 = t_0'] \Rightarrow t_1 = t_1' \wedge \exists \text{as}_{\text{Vars}_e, T_{SIG}}: \text{Vars}_e \rightarrow T_{SIG} \text{ tal que:}$$

$$1) \text{eval}_{\text{Vars}_e, T_{SIG}}(t_0) \equiv_i \text{eval}_{\text{Vars}_e, T_{SIG}}(t_0'), \text{ y}$$

$$2) (\text{eval}_{\text{Vars}_e, T_{SIG}}(t_1) = t \wedge \text{eval}_{\text{Vars}_e, T_{SIG}}(t_1') = t') \vee$$

$$(\text{eval}_{\text{Vars}_e, T_{SIG}}(t_1) = t' \wedge \text{eval}_{\text{Vars}_e, T_{SIG}}(t_1') = t).$$

- Finalmente, se define \equiv_E como \equiv_∞ .

Vemos que en cada nueva congruencia se fusionan aquellas clases de equivalencia que pueden identificarse como la parte derecha y la parte izquierda de la conclusión de una ecuación condicional, siempre que, para la asignación de variables a la que induce esta identificación, todas las premisas se cumplan (es decir, las dos partes de cada premisa estén en una misma clase de equivalencia, dada la sustitución de variables).

1.4.2 Tipos y operaciones auxiliares

Los *tipos* y las *operaciones auxiliares* se introducen dentro de una especificación para facilitar su escritura y legibilidad; algunas veces son incluso imprescindibles para poder especificar las operaciones que configuran una signature dada. Estos símbolos auxiliares son invisibles para los usuarios del TAD (su ámbito es exclusivamente la especificación donde se definen), por lo que también se conocen como *tipos y operaciones ocultos* o *privados* (ing., *hidden* o *private*).

Por ejemplo, supongamos una signature para el TAD de los naturales con operaciones cero, sucesor y producto, pero sin operación de suma. Por lo que respecta a la especificación del producto, hay que determinar los valores de $\text{prod}(\text{cero}, n)$ y de $\text{prod}(\text{suc}(m), n)$; el segundo término es fácil de igualar aplicando la propiedad distributiva:

$$\text{prod}(\text{suc}(m), n) = \text{suma}(\text{prod}(m, n), n)$$

Es necesario introducir, pues, una función auxiliar *suma* que no aparece en la signature

inicial; *suma* se declara en la cláusula "*ops*" de la manera habitual pero precedida de la palabra clave "*privada*" para establecer su ocultación a los universos que usen los naturales:

privada suma: nat nat \rightarrow nat

El último paso consiste en especificar *suma* como cualquier otra operación de la signatura; por ello, al definir una operación privada es conveniente plantearse si puede ser de interés general, para dejarla pública y que otros universos puedan usarla libremente.

El efecto de la declaración de símbolos auxiliares en el modelo asociado a una especificación *ESP* es el siguiente. Se considera la signatura $SIG' \subseteq SIG$ que contiene todos los símbolos no auxiliares de la especificación, y se define la *restricción de ESP con SIG'*, denotada por $\langle\langle T_{ESP} \rangle\rangle_{SIG'}$, como el resultado de olvidar todas las clases del álgebra cociente T_{ESP} cuyo tipo sea auxiliar en *SIG* y todas las operaciones de T_{ESP} asociadas a operaciones auxiliares de *SIG*. Entonces, el modelo es la clase de todas las álgebras isomorfas a $\langle\langle T_{ESP} \rangle\rangle_{SIG'}$. La construcción detallada de este nuevo modelo se encuentra en [EhM85, pp. 145-151].

1.4.3 Tratamiento de los errores

Hasta el momento, se ha considerado que las operaciones de una signatura están bien definidas para cualquier combinación de valores sobre la que se apliquen. Sin embargo, normalmente una o más operaciones de una especificación serán funciones parciales, que no se podrán aplicar sobre ciertos valores del dominio de los datos. Veremos en este apartado cómo tratar estas operaciones, y lo haremos con el ejemplo de la fig. 1.17 de los naturales con operaciones de igualdad y obtención del predecesor.

La especificación no define completamente el comportamiento de la operación *pred*, porque no determina el resultado de *pred(cero)*, que es un error. Dada la construcción del modelo de un TAD presentada en la sección 1.2, se puede introducir una nueva clase de equivalencia dentro del álgebra cociente de términos que represente este valor erróneo; para facilitar su descripción y la evolución posterior de la especificación, se añade una constante a la signatura, $error_{nat}: \rightarrow nat$, que modeliza un valor de error dentro del conjunto base de *nat*, que representaremos con $[error_{nat}]$. Entonces se puede completar la especificación de *pred* con la ecuación $pred(cero) = error_{nat}$.

Notemos que el representante canónico de la clase $[error_{nat}]$ no es un término formado íntegramente por aplicaciones de *cero* y *suc*; por ello, se debe modificar el conjunto de constructoras generadoras incluyendo en él $error_{nat}$. Este cambio es importante, porque la definición del método general obliga a determinar el comportamiento del resto de operaciones de la signatura respecto a $error_{nat}$. Entre diversas opciones posibles adoptamos la estrategia de *propagación de los errores*: siempre que uno de los operandos de una operación sea un error, el resultado también es error. Así, deben añadirse algunas ecuaciones a la especificación, entre ellas:

universo NAT es
usa BOOL
tipo nat
ops cero: \rightarrow nat
suc, pred: nat \rightarrow nat
suma, mult: nat nat \rightarrow nat
ig: nat nat \rightarrow bool
ecns $\forall n, m \in \text{nat}$
1) suma(cero, n) = n
2) suma(suc(m), n) = suc(suma(m, n))
3) mult(cero, n) = cero
4) mult(suc(m), n) = suma(mult(m, n), n)
5) ig(cero, cero) = cierto
6) ig(suc(m), cero) = falso
7) ig(cero, suc(n)) = falso
8) ig(suc(m), suc(n)) = ig(m, n)
9) pred(suc(m)) = m
funiverso

Fig. 1.17: un universo para los naturales con predecesor.

$$\begin{array}{ll}
\text{E1) } \text{suc}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}} & \text{E2) } \text{pred}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}} \\
\text{E3) } \text{suma}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{nat}} & \text{E4) } \text{suma}(n, \text{error}_{\text{nat}}) = \text{error}_{\text{nat}} \\
\text{E5) } \text{mult}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{nat}} & \text{E6) } \text{mult}(n, \text{error}_{\text{nat}}) = \text{error}_{\text{nat}}
\end{array}$$

Hay varios problemas en esta solución. Para empezar, las ecuaciones han introducido inconsistencias: por ejemplo, dado el término $\text{mult}(\text{cero}, \text{error}_{\text{nat}})$, se le pueden aplicar dos ecuaciones, 3 y E6; aplicando E6, con $n = \text{cero}$, se obtiene como resultado $\text{error}_{\text{nat}}$, mientras que aplicando 3, con $n = \text{error}_{\text{nat}}$, se obtiene como resultado cero . Es decir, que el mismo término lleva a dos valores diferentes dentro del álgebra en función de la ecuación que se le aplique. Para evitar este problema intolerable, deben protegerse las ecuaciones normales con una comprobación de que los términos utilizados no son erróneos; en la especificación del producto, por ejemplo, se cambian las ecuaciones 3 y 4 por:

$$\begin{aligned}
& [\text{correcto}_{\text{nat}}(n)] \Rightarrow \text{mult}(\text{cero}, n) = \text{cero} \\
& [\text{correcto}_{\text{nat}}(\text{mult}(m, n)) \wedge \text{correcto}_{\text{nat}}(\text{suc}(m))] \Rightarrow \\
& \quad \Rightarrow \text{mult}(\text{suc}(m), n) = \text{suma}(\text{mult}(m, n), n)
\end{aligned}$$

y lo mismo para el resto de operaciones.

La operación $\text{correcto}_{\text{nat}}: \text{nat} \rightarrow \text{bool}$, que garantiza que un término no representa $\text{error}_{\text{nat}}$, se puede especificar en función de las operaciones constructoras generadoras:

- E7) $\text{correcto}_{\text{nat}}(\text{cero}) = \text{cierto}$
 E8) $\text{correcto}_{\text{nat}}(\text{error}_{\text{nat}}) = \text{falso}$
 E9) $\text{correcto}_{\text{nat}}(\text{suc}(m)) = \text{correcto}_{\text{nat}}(m)$

Un segundo problema surge al modificar la especificación de la igualdad:

- E10) $\text{ig}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{bool}}$ E11) $\text{ig}(m, \text{error}_{\text{nat}}) = \text{error}_{\text{bool}}$

$\text{error}_{\text{bool}}$ es un error de tipo diferente, pues ig devuelve un booleano; es decir, especificando NAT debe modificarse un universo ya existente, introduciendo en él una constructora generadora que obliga a repetir el proceso (y, eventualmente, a modificar otros universos).

Queda claro, pues, que el tratamiento de los errores expande el número de ecuaciones de un universo para solucionar todos los problemas citados; por ejemplo, la especificación de los naturales con predecesor (cuya finalización queda como ejercicio para el lector) pasa de 9 ecuaciones a 21. Este resultado es inevitable con el esquema de trabajo adoptado; ahora bien, podemos tomar algunas convenciones para simplificar la escritura de la especificación, pero sin que varíe el resultado final (v. [ADJ78] para una explicación detallada):

- Todo género introducido en la especificación ofrece implícitamente dos operaciones visibles, la constante de error y el predicado de corrección convenientemente especificado; en el ejemplo de los naturales, $\text{error}_{\text{nat}}: \rightarrow \text{nat}$ y $\text{correcto}_{\text{nat}}: \text{nat} \rightarrow \text{bool}$.
- Todas las ecuaciones de error que no sean de propagación se escribirán en una cláusula aparte y sin explicitar la parte derecha, que es implícitamente el valor de error del tipo adecuado. En Merlí se escriben los errores antes que las otras ecuaciones, a las que por contraposición denominamos ecuaciones *correctas* o *normales*.
- Durante la evaluación de términos, los errores se propagan automáticamente; así, no es necesario introducir las ecuaciones de propagación como $\text{suc}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}}$.
- En el resto de ecuaciones, tanto la parte izquierda como la parte derecha están libres de error (es decir, existen condiciones implícitas); por ello, la ecuación normal $\text{mult}(\text{cero}, n) = \text{cero}$ en realidad significa $[\text{correcto}_{\text{nat}}(n)] \Rightarrow \text{mult}(\text{cero}, n) = \text{cero}$. Con esta suposición, no es necesario comprobar explícitamente que los valores sobre los que se aplica una operación en una ecuación normal están libres de error.

Como resultado, para completar la especificación ejemplo basta con escribir la cláusula $\text{error}_{\text{pred}}(\text{cero})$, con la certeza que las convenciones dadas la expanden correctamente.

1.5 Estudio de casos

Se presentan a continuación algunos ejemplos que permiten ejercitar el método general de especificación introducido en el apartado 1.3.3. y que, sobre todo, muestran algunas

excepciones bastante habituales. La metodología de desarrollo que se sigue en esta sección es la base de la especificación de los diversos TAD que se introducirán en el resto del texto. Los ejemplos han sido elegidos para mostrar la especificación de: a) un par de modelos matemáticos clásicos; b) un módulo auxiliar para la construcción de una aplicación; c) una aplicación entera, de dimensión necesariamente reducida. Pueden encontrarse más ejemplos resueltos en el trabajo "Especificació Algebraica de Tipus Abstractes de Dades: Estudi de Casos", escrito por el autor de este libro y publicado en el año 1991 como *report* LSI-91-5 del Dept. de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya (disponible en lengua catalana).

1.5.1 Especificación de algunos tipos de datos clásicos

a) Polinomios

Queremos especificar los polinomios $\mathbb{Z}[X]$ de una variable con coeficientes enteros, es decir, $p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n$, $n \in \mathbb{N}$, $\forall i: 0 \leq i \leq n: a_i \in \mathbb{Z}$, con operaciones: *cero*: $\rightarrow poli$, que representa el polinomio $p(x) = 0$; *añadir*: $poli \text{ entero } nat \rightarrow poli$, que añade una pareja coeficiente-exponente (abreviadamente, *monomio*) a un término que representa un polinomio, y la operación *evaluar*: $poli \text{ entero} \rightarrow entero$, que calcula el valor del polinomio en un punto. Está claro que las operaciones constructoras generadoras son *cero* y *añadir*, porque cualquier polinomio se puede expresar como un término compuesto íntegramente por ellas. De esta manera, el polinomio $p(x)$ tiene como representante canónico el término *añadir*(*añadir*(...(*añadir*(*cero*, a_{k_0} , k_0), a_{k_1} , k_1), ...), a_{k_r} , k_r), donde sólo aparecen los términos de coeficiente diferente de cero y donde se cumple, por ejemplo, que $k_i < k_{i+1}$.

Es obvio que las constructoras generadoras exhiben ciertas interrelaciones; así, el polinomio $p(x) = 8x$ puede representarse, entre otros, mediante los términos *añadir*(*cero*, 8, 1), *añadir*(*añadir*(*cero*, 8, 1), 0, 5), *añadir*(*añadir*(*cero*, 3, 1), 5, 1) y *añadir*(*añadir*(*cero*, 5, 1), 3, 1), aunque estén contruidos con monomios diferentes, los tengan en diferente orden, o presenten monomios de coeficiente cero. Estas propiedades dan lugar a las ecuaciones:

- 1) *añadir*(*añadir*(*p*, a_1 , n_1), a_2 , n_2) = *añadir*(*añadir*(*p*, a_2 , n_2), a_1 , n_1)
- 2) *añadir*(*añadir*(*p*, a_1 , n), a_2 , n) = *añadir*(*p*, ENTERO.suma(a_1 , a_2), n)^{11,12}
- 3) *añadir*(*p*, ENTERO.cero, n) = *p*

Para demostrar la corrección de estas ecuaciones, se podría establecer una biyección entre el subconjunto del álgebra cociente de términos resultado de agrupar los términos de género *poli* contruidos sobre la signatura de los polinomios y $\mathbb{Z}[X]$.

Por último, se especifica la operación *evaluar* respecto a las constructoras generadoras, es

¹¹ Cuando a una operación como *suma* la precede un nombre de universo como ENTERO, se está explicitando, por motivos de legibilidad, en qué universo está definida la operación.

¹² A lo largo del ejemplo, el universo ENTERO define los enteros con las operaciones necesarias.

decir, se evalúa cada monomio sobre el punto dado (se eleva a la potencia y el resultado se multiplica por el coeficiente), y se suman los resultados parciales:

- 4) evaluar(cero, b) = ENTERO.cero
- 5) evaluar(añadir(p, a, n), b) =
ENTERO.suma(avaluar(p, b), ENTERO.mult(a, ENTERO.eleva(b, n)))

Para seguir con el ejercicio, se enriquece la especificación con algunas operaciones más:

- coeficiente: poli nat \rightarrow entero
- suma, mult: poli poli \rightarrow poli

donde la operación *coeficiente* devuelve el coeficiente asociado a un término de exponente dado dentro un polinomio, y *suma* y *mult* se comportan como su nombre indica. Una primera versión de la especificación de la operación *coeficiente* podría ser:

- 6) coeficiente(cero, n) = ENTERO.cero
- 7) coeficiente(añadir(p, a, n), n) = a
- 8) $[\neg \text{NAT.ig}(n_1, n_2)] \Rightarrow \text{coeficiente}(\text{añadir}(p, a, n_1), n_2) = \text{coeficiente}(p, n_2)$

Es decir, se examinan los monomios que forman el término hasta encontrar el que tiene el exponente dado, y se devuelve el coeficiente asociado; en caso de que no aparezca, la operación acabará aplicándose sobre el término *cero*, y dará 0. Esta versión, no obstante, presenta una equivocación muy frecuente en la construcción de especificaciones: la ecuación está diseñada para aplicarse sobre representantes canónicos, pero no se comporta correctamente al aplicarse sobre cualquier otro término. Así, el coeficiente del monomio de exponente 1 de los términos *añadir(añadir(cero, 3, 1), 5, 1)* y *añadir(cero, 8, 1)* (que son demostrablemente equivalentes) puede ser diferente en función del orden de aplicación de las ecuaciones. La conclusión es que, al especificar cualquier operación de una signatura, no se puede suponer nunca que los términos utilizados en las ecuaciones son canónicos; dicho de otra manera, no es lícito suponer que las ecuaciones de una signatura se aplicarán en un orden determinado (excepto las ecuaciones de error). En este caso, hay que buscar por todo el término los diversos *añadir* del exponente dado y sumarlos:

- 7) coeficiente(añadir(p, a, n), n) = ENTERO.suma(a, coeficiente(p, n))

La especificación de la suma de polinomios es:

- 9) suma(cero, p) = p
- 10) suma(añadir(q, a, n), p) = añadir(suma(q, p), a, n)

Es decir, se añaden los monomios del primer polinomio sobre el segundo; las cuestiones relativas al orden de escritura, a la existencia de diversos monomios para un mismo exponente y de monomios de coeficiente cero no deben tenerse en cuenta, porque las ecuaciones impurificadoras ya las tratan. Notemos que no es necesario descomponer el segundo parámetro en función de las constructoras generadoras, porque el uso de una variable permite especificar lo mismo con menos ecuaciones.

Finalmente, la especificación de la multiplicación puede realizarse de diferentes maneras; la más sencilla consiste en introducir una operación auxiliar *mult_un*: *poli entero nat* \rightarrow *poli* que calcula el producto de un monomio con un polinomio, de manera que la multiplicación de polinomios consiste en aplicar *mult_un* reiteradamente sobre todos los monomios de uno de los dos polinomios, y sumar la secuencia resultante de polinomios:

- 11) $\text{mult}(\text{cero}, p) = \text{cero}$
- 12) $\text{mult}(\text{añadir}(q, a, n), p) = \text{POLI.suma}(\text{mult}(q, p), \text{mult_un}(p, a, n))$
- 13) $\text{mult_un}(\text{cero}, a, n) = \text{cero}$
- 14) $\text{mult_un}(\text{añadir}(p, a_1, n_1), a_2, n_2) =$
 $\text{añadir}(\text{mult_un}(p, a_2, n_2), \text{ENTERO.mult}(a_1, a_2), \text{NAT.suma}(n_1, n_2))$

Una segunda posibilidad no precisa de la operación privada, pero a cambio exige descomponer más los parámetros:

- 11) $\text{mult}(\text{cero}, p) = \text{cero}$
- 12) $\text{mult}(\text{añadir}(\text{cero}, a, n), \text{cero}) = \text{cero}$ (también $\text{mult}(p, \text{cero}) = \text{cero}$)
- 13) $\text{mult}(\text{añadir}(\text{cero}, a_1, n_1), \text{añadir}(q, a_2, n_2)) =$
 $\text{añadir}(\text{mult}(\text{añadir}(\text{cero}, a_1, n_1), q), \text{mult}(a_1, a_2), \text{suma}(n_1, n_2))$
- 14) $\text{mult}(\text{añadir}(\text{añadir}(q, a_1, n_1), a_2, n_2), p) =$
 $\text{suma}(\text{mult}(\text{añadir}(q, a_1, n_1), p), \text{mult}(\text{añadir}(\text{cero}, a_2, n_2), p))$

En realidad, se sigue la misma estrategia, pero en vez de la operación auxiliar se explicita el caso *añadir(cero, a, n)*, de manera que se especifica *mult* respecto a términos sin ningún monomio, términos con un único monomio y términos con dos o más monomios. Notemos que el resultado es más complicado que la anterior versión, lo que confirma que el uso de operaciones auxiliares adecuadas simplifica frecuentemente la especificación resultante.

b) Secuencias

Se propone la especificación del TAD de las *secuencias* o *cadena*s de elementos provenientes de un alfabeto: dado un alfabeto $V = \{a_1, \dots, a_n\}$, las secuencias V^* se definen:

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

λ representa la secuencia vacía, mientras que el resto de cadenas se pueden considerar como el añadido de un elemento (por la derecha o por la izquierda) a una cadena no vacía.

En la fig. 1.18 se da una especificación para el alfabeto. Como no hay ninguna ecuación impurificadora, todas las constantes denotan valores distintos del conjunto base del modelo inicial; se requiere una operación de igualdad para comparar los elementos. Por lo que respecta las cadenas, se propone una signatura de partida con las siguientes operaciones: λ , la cadena vacía; $[v]$, que, dado un elemento v , lo transforma en una cadena que sólo consta de v ; $v.c$, que añade por la izquierda el elemento v a una cadena c ; $c_1.c_2$, que concatena dos cadenas c_1 y c_2 (es decir, coloca los elementos de c_2 a continuación de los

elementos de c_1); $\|c\|$, que devuelve el número de elementos de la cadena c ; y $v \in c$, que comprueba si el elemento v aparece o no en la cadena c . A la vista de esta signatura, puede verse que hay dos conjuntos posibles de constructoras generadoras, $\{\lambda, _._ \}$ y $\{\lambda, _[], _ \cdot _ \}$. La fig. 1.19 muestra las especificaciones resultantes en cada caso, que indican claramente la conveniencia de seleccionar el primer conjunto, que es puro; con éste, el representante canónico es de la forma $v_1.v_2. \dots .v_n.\lambda$, que abreviaremos por $v_1v_2 \dots v_n$ cuando convenga.

universo ALFABETO es
usa BOOL
tipo alf
ops $a_1, \dots, a_n: \rightarrow \text{alf}$
 $_ = _$: $\text{alf alf} \rightarrow \text{bool}$
ecns $(a_1 = a_1) = \text{cierto}$; $(a_1 = a_2) = \text{falso}$; ...
funiverso

Fig. 1.18: especificación del alfabeto.

universo CADENA es
usa ALFABETO, NAT, BOOL
tipo cadena
ops
 λ : $\rightarrow \text{cadena}$
 $_[]$: $\text{alf} \rightarrow \text{cadena}$
 $_ \cdot _$: $\text{alf cadena} \rightarrow \text{cadena}$
 $_ \cdot _$: $\text{cadena cadena} \rightarrow \text{cadena}$
 $\| _ \|$: $\text{cadena} \rightarrow \text{nat}$
 $_ \in _$: $\text{alf cadena} \rightarrow \text{bool}$
funiverso

- | | |
|---|--|
| 1) $[v] = v.\lambda$ | 1) $\lambda \cdot c = c$ |
| 2) $\lambda \cdot c = c$ | 2) $c \cdot \lambda = c$ |
| 3) $(v.c_1) \cdot c_2 = v.(c_1 \cdot c_2)$ | 3) $(c_1 \cdot c_2) \cdot c_3 = c_1 \cdot (c_2 \cdot c_3)$ |
| 4) $\ \lambda\ = \text{cero}$ | 4) $v.c = [v] \cdot c$ |
| 5) $\ v.c\ = \text{suc}(\ c\)$ | 5) $\ \lambda\ = 0$ |
| 6) $v \in \lambda = \text{falso}$ | 6) $\ [v]\ = \text{suc}(\text{cero})$ |
| 7) $v_1 \in (v_2.c) = (v_1 = v_2) \vee v_1 \in c$ | 7) $\ c_1 \cdot c_2\ = \text{suma}(\ c_1\ , \ c_2\)$ |
| | 8) $v \in \lambda = \text{falso}$ |
| | 9) $v_1 \in [v_2] = (v_1 = v_2)$ |
| | 10) $v \in (c_1 \cdot c_2) = v \in c_1 \vee v \in c_2$ |

Fig. 1.19: signatura (arriba) y especificación (abajo) de las cadenas con los dos conjuntos posibles de constructoras generadoras: $\{\lambda, _._ \}$ (izquierda) y $\{\lambda, _[], _ \cdot _ \}$ (derecha).

A continuación se incorporan diversas operaciones a la signatura inicial. Para empezar, se añade una operación de comparación de cadenas, $_=_: cadena\ cadena \rightarrow bool$, cuya especificación respecto del conjunto de constructoras generadoras es sencilla (no lo sería tanto respecto el conjunto que se ha descartado previamente):

$$\begin{array}{ll} 8) (\lambda = \lambda) = \text{cierto} & 9) (v.c = \lambda) = \text{falso} \\ 10) (\lambda = v.c) = \text{falso} & 11) (v_1.c_1 = v_2.c_2) = (v_1 = v_2) \wedge (c_1 = c_2) \end{array}$$

Una operación interesante sobre las cadenas es $i_ésimo: cadena\ nat \rightarrow alf$, que obtiene el elemento i -ésimo de la cadena definido como $i_ésimo(v_1 \dots v_{i-1} v_i v_{i+1} \dots v_n, i) = v_i$. Primero, se controlan los errores de pedir el elemento que ocupa la posición cero o una posición mayor que la longitud de la cadena:

$$12) \text{error } [(||c|| < i) \vee (\text{NAT.ig}(i, \text{cero}))] \Rightarrow i_ésimo(c, i)^{13}$$

Para el resto de ecuaciones, notemos que el elemento i -ésimo no es el que se ha insertado en i -ésima posición en la cadena, sino que el orden de inserción es inverso a la numeración asignada a los caracteres dentro de la cadena. Por ello, las ecuaciones resultantes son:

$$\begin{array}{l} 13) i_ésimo(v.c, \text{suc}(\text{cero})) = v \\ 14) i_ésimo(v.c, \text{suc}(\text{suc}(i))) = i_ésimo(c, \text{suc}(i)) \end{array}$$

Destacamos que la operación ha sido especificada, no sólo respecto a las constructoras generadoras de *cadena*, sino también respecto a las constructoras generadoras de *nat*, distinguiendo el comportamiento para el cero, el uno y el resto de naturales.

Introducimos ahora la operación $\text{rotar_dr}: cadena \rightarrow cadena$ para rotar los elementos una posición a la derecha, $\text{rotar_dr}(v_1 \dots v_{n-1} v_n) = v_n v_1 \dots v_{n-1}$; en vez de especificarla respecto a λ y $_=_$, se incorpora una nueva operación privada para añadir elementos por la derecha, de signatura $_._: cadena\ alf \rightarrow cadena$ (la sobrecarga del identificador $_._$ no provoca ningún problema), que define un nuevo conjunto de constructoras generadoras. La especificación de rotar_dr respecto este nuevo conjunto es:

$$\begin{array}{l} 15) \text{rotar_dr}(\lambda) = \lambda \\ 16) \text{rotar_dr}(c.v) = v.c \end{array}$$

Es decir, y este es un hecho realmente importante, no es obligatorio especificar todas las operaciones del tipo respecto al mismo conjunto de constructoras generadoras. Es más, en realidad no es obligatorio especificar las operaciones respecto a un conjunto de constructoras generadoras, sino que sólo hay que asegurarse que cada operación se especifica respecto a absolutamente todos los términos posibles del álgebra de términos; ahora bien, siendo la especificación que usa las constructoras generadoras una manera clara y sencilla de conseguir este objetivo, se sigue esta estrategia siempre que sea posible.

¹³ Suponemos que los naturales definen las operaciones de comparación que se necesiten.

La especificación de la operación privada queda:

$$17) \lambda.v = v.\lambda$$

$$18) (v_1.c).v_2 = v_1.(c.v_2)$$

Consideramos a continuación la función *medio*: *cadena* \rightarrow *alf*, que devuelve el elemento central de una cadena. Aplicando el razonamiento anterior, en vez de especificarla respecto a las constructoras generadoras, se hace un estudio por casos según la longitud de la cadena (0, 1, 2 o mayor que 2), y el último caso se expresa de la manera más conveniente. La especificación del comportamiento de la operación respecto a todos estos casos garantiza que el comportamiento de *medio* está definido para cualquier cadena:

$$19) \text{error medio}(\lambda)$$

$$20) \text{medio}([v]) = v$$

$$21) \text{medio}(v_1.\lambda.v_2) = v_2 \text{ (o bien } v_1)$$

$$22) [|c| > \text{cero}] \Rightarrow \text{medio}(v_1.c.v_2) = \text{medio}(c)$$

En la ecuación 22 es necesario proteger con la condición, de lo contrario, dada una cadena de dos elementos se podría aplicar tanto 21 como 22, y se provocaría error en este caso.

Para aquellas operaciones que, como *rotar_dr* y *medio*, se han especificado sin seguir el método general, puede ser conveniente demostrar que su definición es correcta. Para ello, aplicaremos la técnica empleada en el apartado 1.2.4 para verificar la corrección de la suma según se explica a continuación. Sea $\{\lambda, _._ \}$ el conjunto de operaciones constructoras generadoras, siendo $_.$ la operación de añadir un carácter por la izquierda. Ya se ha dicho que este conjunto es puro, y es posible establecer una biyección entre el conjunto de base de las cadenas en el álgebra cociente y el modelo matemático de las cadenas. Para demostrar la corrección de *rotar_dr* y *medio*, debe comprobarse que:

$$\text{rotar_dr}_Q([v_n.v_{n-1} \dots v_2.v_1.\lambda]) = [v_1.v_n.v_{n-1} \dots v_2.\lambda],$$

$$\text{medio}_Q([\lambda]) = [\text{error}_{\text{alf}}], \text{ y}$$

$$\text{medio}_Q([v_n.v_{n-1} \dots v_1.\lambda]) = [v_{\lfloor (n+1)/2 \rfloor}], n > 0$$

significando el subíndice "Q" la interpretación de la operación en el álgebra cociente. Ambas demostraciones se desarrollan con la ayuda de un lema que demuestra la equivalencia de los añadidos por la derecha y por la izquierda si se aplican en el orden correcto.

Lema. Todo término representando una cadena no vacía de la forma $v_n.v_{n-1} \dots v_2.v_1.\lambda$, $n > 0$, es equivalente por las ecuaciones al término $v_n.v_{n-1} \dots v_2.\lambda.v_1$.

Demostración. Por inducción sobre n .

$n = 1$. Queda el término $v_1.\lambda$, que es igual a $\lambda.v_1$ aplicando la ecuación 17.

$n = k$. Hipótesis de inducción: $v_k.v_{k-1} \dots v_1.\lambda$ se transforma en $v_k.v_{k-1} \dots v_2.\lambda.v_1$.

$n = k+1$. Debe demostrarse que $t = v_{k+1}.v_k \dots v_1.\lambda$ cumple el lema. Ello es inmediato ya que puede aplicarse la hipótesis de inducción sobre el subtérmino $t_0 = v_k \dots v_1.\lambda$, obteniendo $t_1 = v_k.v_{k-1} \dots \lambda.v_1$. Sustituyendo t_0 por t_1 dentro de t se obtiene el término $v_{k+1}.v_k.v_{k-1} \dots \lambda.v_1$, que cumple el enunciado del lema.

Teorema. Todo término de la forma $rotar_dr(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda)$ es equivalente por las ecuaciones al término $v_1 \cdot v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$.

Demostración. Por análisis de casos sobre n .

$n = 0$. El término es $rotar_dr(\lambda)$, que se transforma en λ aplicando 15, y se cumple el enunciado.

$n > 0$. Sea $t = rotar_dr(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda)$. Como $n > 0$, se aplica el lema y t queda igual a $rotar_dr(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda \cdot v_1)$. A continuación, se aplica la ecuación 16 con $c = v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$ y queda $t = v_1 \cdot v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$, como se quería demostrar.

Teorema. Todo término de la forma $medio(v_n \cdot v_{n-1} \cdot \dots \cdot v_1 \cdot \lambda)$ es equivalente por las ecuaciones al término $v_{\lfloor (n+1)/2 \rfloor}$ si $n > 0$, o bien es un error si $n = 0$.

Demostración. Por análisis de casos sobre n .

$n = 0$. El término es de la forma $medio(\lambda)$, que es igual al error de tipo *alf* aplicando 19.

$n > 0$. Por inducción sobre n .

$n = 1$. Queda el término $medio(v_1 \cdot \lambda)$, que es igual a v_1 aplicando 20. Como la expresión $\lfloor (n+1) / 2 \rfloor$ vale 1 con $n = 1$, se cumple el enunciado.

$n = k$. Hipótesis de inducción: $medio(v_k \cdot v_{k-1} \cdot \dots \cdot v_1 \cdot \lambda)$ se transforma en $v_{\lfloor (k+1)/2 \rfloor}$.

$n = k+1$. Debe demostrarse que $t = medio(v_{k+1} \cdot v_k \cdot \dots \cdot v_1 \cdot \lambda)$ cumple el teorema. Es necesario primero aplicar el lema sobre el subtérmino $v_{k+1} \cdot v_k \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda$ (lo que es posible ya que $k+1 > 0$), resultando en $v_{k+1} \cdot v_k \cdot \dots \cdot v_2 \cdot \lambda \cdot v_1$. Si reescribimos t con esta sustitución y con el cambio $w_i = v_{i+1}$, obtenemos el término $t = medio(w_k \cdot w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda \cdot v_1)$. A continuación, deben distinguirse dos casos:

$k = 1$: queda $t = medio(w_1 \cdot \lambda \cdot v_1)$, con lo que se aplica la ecuación 21 y se obtiene v_1 , que cumple el enunciado ya que $\lfloor (n+1) / 2 \rfloor = \lfloor (k+1+1) / 2 \rfloor = 1$.

$k > 1$: se aplica la ecuación 22 sobre el término t mediante la asignación $c = w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda$, cuya longitud $\|c\|$ es mayor que 0 (por lo que se cumple la premisa de la ecuación), resultando en $t = medio(w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda)$. A continuación, se aplica la hipótesis de inducción y se obtiene $w_{\lfloor k/2 \rfloor}$, que al deshacer el cambio se convierte en $v_{\lfloor k/2 \rfloor + 1} = v_{\lfloor (k+2)/2 \rfloor} = v_{\lfloor (k+1)+1/2 \rfloor}$, y se cumple el enunciado del teorema.

1.5.2 Especificación de una tabla de símbolos

Durante la compilación de un programa es necesario construir una estructura que asocie a cada objeto que en él aparece, identificado con un nombre, un cierto número de características, como su tipo, la dirección física en memoria, etc. Esta estructura se denomina *tabla de símbolos* (ing., *symbol table*) y se construye a medida que se examina el texto fuente. Su estudio es interesante tanto desde el punto de vista clásico de la implementación como desde la vertiente ecuacional. Diversos tipos de lenguajes pueden exigir tablas que

presenten características ligeramente diferentes; en este apartado nos referimos a un caso concreto: la especificación de una tabla de símbolos para un lenguaje de bloques.

Los lenguajes de bloques, como Pascal o C, definen ámbitos de existencia para los objetos que corresponden a los diferentes *bloques* que forman el programa principal y que normalmente se asocian a la idea de subprograma. Como los bloques pueden anidarse, el lenguaje ha de definir exactamente las reglas de visibilidad de los identificadores de los objetos, de manera que el compilador sepa resolver posibles ambigüedades durante la traducción del bloque en tratamiento, que denominaremos *bloque en curso*. Normalmente, las dos reglas principales son: a) no pueden declararse dos objetos con el mismo nombre dentro del mismo bloque y, b) una referencia a un identificador denota el objeto más cercano con este nombre, consultando los bloques de dentro a fuera a partir del bloque en curso. Bajo estas reglas, se propone la siguiente signatura para el tipo *ts* de las tablas de símbolos, donde *cadena* representa el género de las cadenas de caracteres (con la especificación que hemos definido en el punto anterior) y *caracts* representa las características de los objetos:

crea: $\rightarrow ts$, crea la tabla vacía, siendo el programa principal el bloque en curso
entra: $ts \rightarrow ts$, registra que el compilador entra en un nuevo bloque
sal: $ts \rightarrow ts$, registra que el compilador abandona el bloque en curso; el nuevo bloque en curso pasa a ser aquel que lo englobaba
declara: $ts\ cadena\ caracts \rightarrow ts$, declara dentro del bloque en curso un objeto identificado por la cadena y con las características dadas
consulta: $ts\ cadena \rightarrow caracts$, devuelve las características del objeto identificado por la cadena dada, correspondientes a su declaración dentro del bloque más próximo que englobe el bloque en curso
declarado?: $ts\ cadena \rightarrow bool$, indica si la cadena identifica un objeto que ha sido declarado dentro del bloque en curso

En la fig. 1.20 se presenta un programa Pascal y la tabla de símbolos *T* asociada, justo en el momento en que el compilador está en el punto *writeln(c)*. Como características se dan la categoría del identificador, su tipo (si es el caso) y su dimensión; la raya gruesa delimita los dos bloques existentes, el programa principal y el bloque correspondiente al procedimiento *Q* en curso. En esta situación, *consulta(T, a)* devuelve la descripción de la variable local *a*, que oculta la variable global *a*, mientras que *consulta(T, b)* devuelve la descripción de la variable global *b*. La variable global *a* volverá a ser visible cuando el compilador salga del bloque *Q*.

Aplicando el método, primero se eligen las operaciones constructoras generadoras. Además de la operación *crea*, está claro que *declara* también es constructora generadora, ya que es la única operación que permite declarar identificadores dentro de la tabla. Además, la tabla de símbolos necesita incorporar la noción de bloque y, por ello, es necesario tomar como constructora generadora la operación *entra*. Con este conjunto mínimo de tres operaciones se puede generar cualquier tabla de símbolos, eso sí, hay dos interrelaciones claras que se

```

program P;
  var a, b: integer;
    procedure Q (c: integer);
      var a, d: real;
    begin
      writeln(c)
    end;
  begin
    Q(a)
  end.

```



P	program		10230
a	var	int	1024
b	var	int	1026
Q	proc		10500
c	param	int	1028
a	var	real	1030
d	var	real	1032

Fig. 1.20: un programa Pascal y su tabla de símbolos asociada.

deben explicitar: dentro de un mismo bloque, es un error repetir la declaración de un identificador y, además, no importa el orden de la declaración de identificadores:

- 1) error [declarado?(t, id)] \Rightarrow declara(t, id, c)
- 2) declara(declara(t, id₁, c₁), id₂, c₂) = declara(declara(t, id₂, c₂), id₁, c₁)

La operación *sal* ha de eliminar todas las declaraciones realizadas en el bloque en curso, y por eso se obtienen tres ecuaciones aplicando directamente el método:

- 3) error sal(crea)
- 4) sal(entra(t)) = t
- 5) sal(declara(t, id, c)) = sal(t)

Se ha considerado un error intentar salir del programa principal. Notemos que 4 realmente se corresponde con la idea de salida del bloque, mientras que 5 elimina todos los posibles identificadores declarados en el bloque en curso.

Por lo que a *consulta* se refiere, el esquema es idéntico; la búsqueda aprovecha que la estructura en bloques del programa se refleja dentro del término, de manera que se para al encontrar la primera declaración:

- 6) error consulta(crea, id)
- 7) consulta(entra(t), id) = consulta(t, id)
- 8) consulta(declara(t, id, c), id) = c
- 9) $[\neg (id_1 = id_2)] \Rightarrow$ consulta(declara(t, id₁, c₁), id₂) = consulta(t, id₂)

Por último, *declarado?* no presenta más dificultades:

- 10) declarado?(crea, id) = falso
- 11) declarado?(entra(t), id) = falso
- 12) declarado?(declara(t, id₁, c₁), id₂) = (id₁ = id₂) \vee declarado?(t, id₂)

1.5.3 Especificación de un sistema de reservas de vuelos

Se quiere especificar el sistema de reservas de vuelos de la compañía "Averia". Se propone una signatura que permita añadir y cancelar vuelos dentro de la oferta de la compañía, realizar reservas de asientos y consultar información diversa. En concreto, las operaciones son:

crea: $\rightarrow averia$, devuelve un sistema de reservas sin información
añade: $averia\ vuelo \rightarrow averia$, añade un nuevo vuelo al sistema de reservas
reserva: $averia\ vuelo\ pasajero \rightarrow averia$, registra una reserva hecha por un pasajero dentro de un vuelo determinado; no puede haber más de una reserva a nombre de un pasajero en un mismo vuelo
cancela: $averia\ vuelo \rightarrow averia$, anula un vuelo del sistema de reservas; todas las reservas de pasajeros para este vuelo, si las hay, han de ser eliminadas
asiento?: $averia\ vuelo\ pasajero \rightarrow nat$, determina el número de asiento asignado a un pasajero dentro de un vuelo; se supone que esta asignación se realiza por orden de reserva, y que los asientos se numeran en orden ascendente a partir del uno; devuelve 0 si el pasajero no tiene ninguna reserva
lista: $averia \rightarrow cadena_vuelos$, obtiene todos los vuelos en orden de hora de salida
a_suspender: $averia\ aeropuerto \rightarrow cadena_vuelos$, dado un aeropuerto, proporciona la lista de todos los vuelos que se dirigen a este aeropuerto

Supondremos que los tipos *vuelo*, *aeropuerto*, *hora*, *pasajero* y *nat*, especificados en los universos *VUELO*, *AEROPUERTO*, *HORA*, *PASAJERO* y *NAT*, respectivamente, disponen de operaciones de comparación *ig*; además, el universo *VUELO* presenta las operaciones:

origen?, *destino?*: $vuelo \rightarrow aeropuerto$, da los aeropuertos del vuelo
hora_salida?: $vuelo \rightarrow hora$, da la hora de salida del vuelo
capacidad?: $vuelo \rightarrow nat$, da el número máximo de reservas que admite el vuelo

Las cadenas de vuelos tendrán las operaciones sobre cadenas vistas en el punto 1.5.1.

El sistema de vuelos necesita información tanto sobre vuelos como sobre reservas; por ello, son constructoras generadoras las operaciones *añade* y *reserva*, y también la típica *crea*. Se debe ser especialmente cuidadoso al establecer los errores que presentan estas operaciones: añadir un vuelo repetido, reservar dentro un vuelo que no existe, reservar para un pasajero más de un asiento en el mismo vuelo y reservar en un vuelo lleno. Hay también varias relaciones conmutativas:

- 1) error $[existe?(s, v)] \Rightarrow añade(s, v)$
- 2) error $[\neg existe?(s, v) \vee lleno?(s, v) \vee \neg ig(asiento(s, v, p), 0)] \Rightarrow reserva(s, v, p)$
- 3) $añade(añade(s, v_1), v_2) = añade(añade(s, v_2), v_1)$
- 4) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow reserva(reserva(s, v_1, p_1), v_2, p_2) =$
 $reserva(reserva(s, v_2, p_2), v_1, p_1)$
- 5) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow añade(reserva(s, v_1, p), v_2) = reserva(añade(s, v_2), v_1, p)$

Para expresar estas relaciones hemos introducido dos operaciones auxiliares:

privada lleno?: *averia vuelo* \rightarrow *bool*, comprueba si el vuelo no admite más reservas

6) $\text{lleno?}(s, v) = \text{NAT.ig}(\text{cuenta}(s, v), \text{VUELO.capacidad}(v))$

privada existe?: *averia vuelo* \rightarrow *bool*, comprueba si el vuelo ya existía

7) $\text{existe?}(\text{crea}, v) = \text{falso}$

8) $\text{existe?}(\text{añade}(s, v_1), v_2) = \text{VUELO.ig}(v_1, v_2) \vee \text{existe?}(s, v_2)$

9) $\text{existe?}(\text{reserva}(s, v_1, p), v_2) = \text{existe?}(s, v_2)$

La operación *cuenta*: *averia vuelo* \rightarrow *nat* da el número de reservas realizadas en el vuelo y se especifica más adelante, así como el resto de operaciones.

a) cancela

Además del error de cancelar en un vuelo no existente, el comportamiento de *cancela* depende de si el vuelo que se anula es el mismo que se está examinando o no; en el primer caso, desaparece la información sobre el vuelo, y en el segundo caso debe conservarse:

10) *error* $\text{cancela}(\text{crea}, v)$

11) $\text{cancela}(\text{añade}(s, v), v) = s$

12) $[\neg \text{VUELO.ig}(v_1, v_2)] \Rightarrow \text{cancela}(\text{añade}(s, v_1), v_2) = \text{añade}(\text{cancela}(s, v_2), v_1)$

13) $\text{cancela}(\text{reserva}(s, v, p), v) = \text{cancela}(s, v)$

14) $[\neg \text{VUELO.ig}(v_1, v_2)] \Rightarrow \text{cancela}(\text{reserva}(s, v_1, p), v_2) = \text{reserva}(\text{cancela}(s, v_2), v_1, p)$

Notemos que la segunda ecuación no precisa controlar posibles repeticiones del vuelo dentro del término, porque en este caso habrían actuado las ecuaciones de error.

b) asiento?

Para especificar esta operación, notemos que el orden de numeración de los asientos es inverso al orden de exploración del término; lo que debe hacerse, pues, es saltar todas las reservas realizadas posteriormente y contar las reservas del mismo vuelo que todavía quedan.

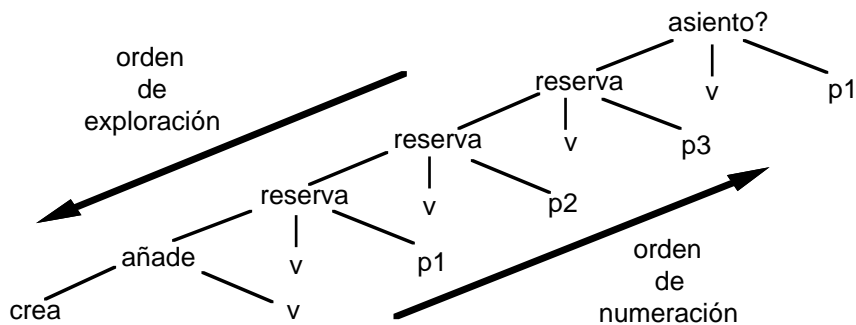


Fig. 1.21: relación entre la numeración y la exploración de las reservas.

Es decir, la operación ha de cambiar de comportamiento al encontrar la reserva; por ello, usamos la operación privada *cuenta* ya introducida, y comparamos si los vuelos y los pasajeros que se van encontrando en el término son los utilizados en la operación (controlando su existencia):

- 15) error asiento?(crea, v, p)
- 16) $[VUELO.ig(v_1, v_2)] \Rightarrow asiento?(añade(s, v_1), v_2, p) = \text{cero}$
- 17) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow asiento?(añade(s, v_1), v_2, p) = asiento?(s, v_2, p)$
- 18) $asiento?(reserva(s, v, p), v, p) = suc(cuenta(s, v))$
- 19) $[\neg(VUELO.ig(v_1, v_2) \wedge PASAJERO.ig(p_1, p_2))] \Rightarrow$
 $asiento?(reserva(s, v_1, p_1), v_2, p_2) = asiento?(s, v_2, p_2)$

La especificación de *cuenta* es realmente sencilla: se incrementa en uno el resultado de la función siempre que se encuentre una reserva en el vuelo correspondiente:

- 20) error cuenta?(crea, v)
- 21) $cuenta(añade(s, v), v) = \text{cero}$
- 22) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow cuenta(añade(s, v_1), v_2) = cuenta(s, v_2)$
- 23) $cuenta(reserva(s, v, p), v) = suc(cuenta(s, v))$
- 24) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow cuenta(reserva(s, v_1, p), v_2) = cuenta(s, v_2)$

c) lista

La resolución presenta diversas alternativas; por ejemplo, se puede considerar la existencia de una operación de ordenación sobre las cadenas, *ordena* (que queda como ejercicio para el lector), y entonces escribir la especificación de *lista* como:

$lista(s) = ordena(enumera(s))$

donde *enumera* es una operación auxiliar del sistema de vuelos que los enumera todos:

privada enumera: avería \rightarrow cadena_vuelos

- 25) $enumera(crea) = \lambda$
- 26) $enumera(reserva(s, v, p)) = enumera(s)$
- 27) $enumera(añade(s, v)) = v . enumera(s)$

d) a_suspender

No hay ninguna situación especial; si el sistema de reservas está vacío, devuelve la cadena vacía, y el resultado sólo se modifica al encontrar un vuelo con el mismo aeropuerto destino:

- 28) $a_suspender(crea, a) = \lambda$
- 29) $[AEROPUERTO.ig(a, VUELO.destino?(v))] \Rightarrow$
 $a_suspender(añade(s, v), a) = v . a_suspender(s, a)$
- 30) $[\neg AEROPUERTO.ig(a, VUELO.destino?(v))] \Rightarrow$
 $a_suspender(añade(s, v), a) = a_suspender(s, a)$
- 31) $a_suspender(reserva(s, v, p), a) = a_suspender(s, a)$

1.6 Estructuración de especificaciones

Hasta ahora, nos hemos centrado en la construcción de especificaciones simples, universos dentro los cuales se definen todos los géneros que forman parte de la especificación; es obvio que se necesitan mecanismos que permitan estructurar estos universos para poder especificar aplicaciones enteras como la composición de especificaciones más simples.

Hay varios lenguajes de especificación que definen mecanismos de estructuración. Las primeras ideas fueron formuladas por R.M. Burstall y J.A. Goguen en "Putting Theories together to make Specifications" (*Proceedings of 5th International Joint Conference on Artificial Intelligence*, Cambridge M.A., 1977), y dieron como resultado el lenguaje CLEAR por ellos desarrollado. Hay otros lenguajes igualmente conocidos: OBJ (inicialmente definido por el mismo J.A. Goguen), ACT ONE (desarrollado en la Universidad de Berlín y descrito en [EhM85]), el más moderno MAUDE (diseñado por José Meseguer y su grupo en el SRI), etc. En Merlí se incorporan las características más interesantes de estos lenguajes, si bien aquí sólo se definen las construcciones que son estrictamente necesarias para especificar los TAD que en él aparecen. Debe destacarse la ausencia de construcciones relativas a la orientación a objetos (principalmente, un mecanismo para declarar subtipos mediante la relación de herencia) porque su inclusión aumentaría la complejidad de este texto y no es imprescindible para el desarrollo de los temas; una referencia interesante en este campo es el libro de R. Breu titulado *Algebraic Specification Techniques in Object Oriented Programming Environments*, Springer-Verlag, serie LNCS nº 562, 1991.

1.6.1 Uso de especificaciones

Mecanismo necesario para usar desde una especificación los géneros, las operaciones y las ecuaciones escritos en otra; se puede considerar una simple copia literal. Si la especificación $SPEC_1 = (S_1, OP_1, E_1)$ usa la especificación $SPEC_2 = (S_2, OP_2, E_2)$, se puede decir de forma equivalente que $SPEC_1 = (S_1 \cup S_2, OP_1 \cup OP_2, E_1 \cup E_2)$ (considerando la unión de un A -conjunto y un B -conjunto como un $A \cup B$ -conjunto). La relación de uso es transitiva por omisión: si A usa B y B usa C , A está usando C implícitamente; ahora bien, esta dependencia se puede establecer explícitamente para favorecer la legibilidad y la modificabilidad. Alternativamente, la palabra "privado" puede preceder al nombre del universo usado, con lo que se evita la transitividad (dicho de otra manera, el uso introduce los símbolos de $SPEC_2$ como privados de $SPEC_1$, el cual no los exporta).

En el apartado 1.3.1 ya se introdujo este mecanismo como una herramienta para la definición de nuevos TAD que necesiten operaciones y géneros ya existentes; por ejemplo, para definir el TAD de los naturales con igualdad se usa el TAD de los booleanos, que define el género *bool* y diversas operaciones. Normalmente, un universo declara un único tipo nuevo, denominado *tipo (género) de interés*.

Otra situación común es el *enriquecimiento* (ing., *enrichment*) de uno o varios tipos añadiendo nuevas operaciones. Un escenario habitual es la existencia de una biblioteca de universos donde se definen algunos TAD de interés general que, como serán manipulados por diferentes usuarios con diferentes propósitos, incluyen operaciones no orientadas hacia ninguna utilización particular. En una biblioteca de este estilo se puede incluir, por ejemplo, un TAD para los conjuntos con operaciones habituales (v. fig. 1.22, izq.). En una aplicación particular, no obstante, puede ser necesario definir nuevas operaciones sobre el tipo; por ejemplo, una que elimine del conjunto todos los elementos mayores que cierta cota. Es suficiente con escribir un nuevo universo que, además de las operaciones sobre conjuntos de la biblioteca, también ofrezca esta operación (v. fig. 1.22, derecha); cualquier universo que lo use, implícitamente está utilizando *CJT_NAT* por la transitividad de los usos.

<u>universo CJT_NAT es</u>	<u>universo MI_CJT_NAT es</u>
<u>usa</u> NAT, BOOL	<u>usa</u> CJT_NAT, NAT, BOOL
<u>tipo</u> cjt_nat	<u>ops</u>
<u>ops</u> \emptyset : \rightarrow cjt_nat	filtra: cjt_nat nat \rightarrow cjt_nat
$\{_ \}$: nat \rightarrow cjt_nat	<u>ecns</u> ...
<u>elige</u> : cjt_nat \rightarrow nat	<u>funiverso</u>
$_ \cup _, _ \cap _, \dots$: cjt_nat cjt_nat \rightarrow cjt_nat	
<u>ecns</u> ...	
<u>funiverso</u>	

Fig. 1.22: especificación de los conjuntos de naturales (izq.) y un enriquecimiento (der.).

1.6.2 Ocultación de símbolos

Así como el mecanismo de uso se caracteriza por hacer visibles los símbolos residentes en una especificación, es útil disponer de una construcción complementaria que se ocupe de ocultar los símbolos de una especificación a todas aquéllas que la usen. De esta manera, los especificadores pueden controlar el ámbito de existencia de los diferentes símbolos de un universo en las diversas partes de una aplicación, y evitar así usos indiscriminados.

Hay varias situaciones en las que la ocultación de símbolos es útil; una de ellas es la redefinición de símbolos, que consiste en cambiar el comportamiento de una operación. En la fig. 1.23, se redefine la operación *elige* introducida en el universo *CJT_NAT* de la fig. 1.22 por otra del mismo nombre que escoge el elemento más grande del conjunto; para ello, se oculta la operación usada y, a continuación, se especifica la operación como si fuera nueva. Cualquier uso del universo *MI_CJT_NAT* contempla la nueva definición de *elige*.

Otra situación consiste en restringir el conjunto de operaciones válidas sobre un TAD. En el capítulo 3 se especifican de forma independiente los tipos de las pilas y las colas, ambos

variantes de las secuencias vistas en la sección anterior. Una alternativa, que queda como ejercicio para el lector, consiste en especificar ambos tipos a partir de un TAD general para las secuencias, restringiendo las operaciones que sobre ellos se pueden aplicar: inserciones, supresiones y consultas siempre por el mismo extremo en las pilas, e inserciones por un extremo y supresiones y consultas por el otro, en las colas.

```

universo MI_CJT_NAT es
  usa CJT_NAT, NAT, BOOL
  esconde elige
  ops elige: cjt_nat → nat
  error elige(∅)
  ecns elige({n}) = n; ...
funiverso

```

Fig. 1.23: redefinición de la operación elige de los conjuntos.

Por lo que respecta al modelo, los símbolos ocultados se pueden considerar parte de la signatura privada usada para restringir el álgebra cociente de términos (v. apartado 1.4.2).

1.6.3 Renombramiento de símbolos

El *renombramiento* (ing., *renaming*) de símbolos consiste en cambiar el nombre de algunos géneros y operaciones de un universo sin modificar su semántica inicial (porque el álgebra inicial de una especificación es una clase de álgebras isomorfas, insensible a los cambios de nombre). Se utiliza para obtener universos más legibles y para evitar conflictos de tipo al instanciar universos genéricos (v. apartado siguiente).

Por ejemplo, supongamos la existencia de un universo que especifique las tuplas de dos enteros (v. fig. 1.24). Si en una nueva aplicación se necesita introducir el TAD de los racionales, hay dos opciones: por un lado, definir todo el tipo partiendo de la nada; por el otro, aprovechar la existencia del universo *DOS_ENTEROS* considerando los racionales como parejas de enteros numerador-denominador. En la fig. 1.25 (izq.) se muestra la segunda opción: primero, se renombra el género y las operaciones de las parejas de enteros dentro del universo *RAC_DEF* y, seguidamente, se introducen las operaciones del álgebra de los racionales que se crean necesarias, con las ecuaciones correspondientes, dentro del universo *RACIONALES*. El proceso se puede repetir en otros contextos; así, en la fig. 1.25 (derecha) se muestra la definición de las coordenadas de un espacio bidimensional donde se renombra el género, pero no las operaciones, de las parejas de enteros, por lo que las operaciones de construcción y consulta de coordenadas tienen el mismo nombre que las operaciones correspondientes de las parejas de enteros.

universo DOS_ENTEROS es
usa ENTERO
tipo 2enteros
ops $<_, _>$: entero entero \rightarrow 2enteros
 $_.c_1, _.c_2$: 2enteros \rightarrow entero
ecns $\forall z_1, z_2 \in \text{entero}$
 $<z_1, z_2>.c_1 = z_1; <z_1, z_2>.c_2 = z_2$
funiverso

Fig. 1.24: especificación para las tuplas de dos enteros.

<u>universo</u> RAC_DEF es	<u>universo</u> COORD_DEF es
<u>usa</u> DOS_ENTEROS	<u>usa</u> DOS_ENTEROS
<u>renombra</u>	<u>renombra</u> 2enteros <u>por</u> coord
2enteros <u>por</u> rac	<u>funiverso</u>
$<_, _>$ <u>por</u> $_/_, _.c_1$ <u>por</u> num, $_.c_2$ <u>por</u> den	
<u>funiverso</u>	
<u>universo</u> RACIONALES es	<u>universo</u> COORDENADAS es
<u>usa</u> RAC_DEF	<u>usa</u> COORD_DEF
<u>ops</u> $(_)^{-1}, -_$: rac \rightarrow rac	<u>ops</u> dist: coord coord \rightarrow coord
$+_-, -_-, *_$: rac rac \rightarrow rac	<u>ecns</u> ...
<u>ecns</u> ...	<u>funiverso</u>
<u>funiverso</u>	

Fig. 1.25: especificación de los racionales (izq.) y de las coordenadas (derecha).

1.6.4 Parametrización e instanciación

La parametrización permite formular una descripción genérica, que puede dar lugar a diversas especificaciones concretas mediante las asociaciones de símbolos adecuadas. Por ejemplo, sean dos especificaciones para los conjuntos de naturales y de enteros (v. fig. 1.26); se puede observar que los universos resultantes son prácticamente idénticos, excepto algunos nombres (*nat* por *entero*, *cjt_nat* por *cjt_ent*), por lo que el comportamiento de las operaciones de los conjuntos es independiente del tipo de sus elementos. Por esto, en vez de especificar unos conjuntos concretos es preferible especificar los conjuntos de cualquier tipo de elementos (v. fig. 1.27). Esta especificación se llama *especificación parametrizada* o *genérica* (ing., *parameterised* o *generic specification*) de los conjuntos; los símbolos que condicionan el comportamiento del universo genérico se llaman *parámetros formales* (ing., *formal parameter*) y en Merlí se definen en *universos de caracterización*, cuyo nombre aparece en la cabecera del universo genérico. En el caso de los conjuntos, el parámetro formal es el género *elem*, que está definido dentro del universo de caracterización *ELEM*.

<u>universo CJT_NAT es</u>	<u>universo CJT_ENT es</u>
<u>usa</u> NAT	<u>usa</u> ENTER
<u>tipo</u> cjt_nat	<u>tipo</u> cjt_ent
<u>ops</u> $\emptyset: \rightarrow \text{cjt_nat}$	<u>ops</u> $\emptyset: \rightarrow \text{cjt_ent}$
<u>\cup</u> : cjt_nat nat \rightarrow cjt_nat	<u>\cup</u> : cjt_ent enter \rightarrow cjt_ent
<u>ecns</u> $\forall s \in \text{cjt_nat}; \forall n, m \in \text{nat}$	<u>ecns</u> $\forall s \in \text{cjt_ent}; \forall n, m \in \text{enter}$
$s \cup \{n\} \cup \{n\} = s \cup \{n\}$	$s \cup \{n\} \cup \{n\} = s \cup \{n\}$
$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$
<u>funiverso</u>	<u>funiverso</u>

Fig. 1.26: especificación de los conjuntos de naturales y de enteros.

<u>universo CJT (ELEM) es</u>	<u>universo ELEM caracteriza</u>
<u>tipo</u> cjt	<u>tipo</u> elem
<u>ops</u> $\emptyset: \rightarrow \text{cjt}$	<u>funiverso</u>
<u>\cup</u> : cjt elem \rightarrow cjt	
<u>ecns</u> $\forall s \in \text{cjt}; \forall n, m \in \text{elem}$	
$s \cup \{n\} \cup \{n\} = s \cup \{n\}; s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	
<u>funiverso</u>	

Fig. 1.27: especificación de los conjuntos genéricos (izq.) y caracterización de los elementos (der.).

Para crear un universo para los conjuntos de naturales debe efectuarse lo que se denomina una *instancia* (ing., *instantiation* o *actualisation*) del universo genérico (v. fig. 1.28), que consiste en asociar unos *parámetros reales* (ing., *actual parameter*) a los formales; por este motivo, la instancia también se denomina *paso de parámetros* (ing., *parameter passing*). La cláusula "*instancia*" indica que se crea un nuevo tipo a partir de la descripción genérica dada en *CJT(ELEM)* a través de la asociación de *nat* a *elem*. Además, se renombra el símbolo correspondiente al género que se había definido, porque, de lo contrario, toda instancia de *CJT(ELEM)* definiría un género con el mismo nombre; opcionalmente, y para mejorar la legibilidad, se podría renombrar algún otro símbolo. El resultado de la instancia es idéntico a la especificación de los conjuntos de naturales de la fig. 1.26; nótese, sin embargo, la potencia del mecanismo de parametrización que permite establecer el comportamiento de más de un TAD en un único universo.

universo CJT_NAT es
usa NAT
instancia CJT(ELEM) donde elem es nat
renombra cjt por cjt_nat
funiverso

Fig. 1.28: instancia de los conjuntos genéricos para obtener conjuntos de naturales.

Este caso de genericidad es el más simple posible, porque el parámetro es un símbolo que no ha de cumplir ninguna condición; de hecho, se podría haber simulado mediante renombramientos adecuados. No obstante, los parámetros formales pueden ser símbolos de operación a los que se exija cumplir ciertas propiedades. Por ejemplo, consideremos los conjuntos con operación de pertenencia de la fig. 1.29; en este caso se introduce un nuevo parámetro formal, la operación de igualdad $_=_$ sobre los elementos de género *elem* (no confundir con el símbolo '=' de las ecuaciones). Es necesario, pues, construir un nuevo universo de caracterización, $ELEM_ =$, que defina el género de los elementos con operación de igualdad. Para mayor comodidad en posteriores usos del universo, se requiere también la operación $_ \neq _$ definida como la negación de la igualdad; para abreviar, y dada la ecuación que define totalmente su comportamiento, consideraremos que $_ \neq _$ se establecerá automáticamente a partir de $_ = _$ en toda instancia de $ELEM_ =$. Notemos que las ecuaciones de $_ = _$ no definen su comportamiento preciso, sino que establecen las propiedades que cualquier parámetro real asociado ha de cumplir (reflexividad, simetría y transitividad). Al efectuar una instancia de $CJT_ \in$, sólo se puede tomar como tipo base de los conjuntos un género que tenga una operación con la signatura correcta y que cumpla las propiedades citadas; por ejemplo, la definición de los enteros de la fig. 1.14 no permite crear conjuntos de enteros, porque no hay ninguna operación que pueda desempeñar el papel de la igualdad, mientras que sí que lo permiten los naturales de la fig. 1.10 (v. fig. 1.30). Digamos por último que la instancia puede asociar expresiones (que representan operaciones anónimas) a los parámetros formales que sean operaciones, siempre y cuando cada expresión cumpla las ecuaciones correspondientes.

<u>universo</u> $CJT_ \in$ ($ELEM_ =$) es	<u>universo</u> $ELEM_ =$ caracteriza
<u>usa</u> BOOL	<u>usa</u> BOOL
<u>tipo</u> cjt	<u>tipo</u> elem
<u>ops</u> $\emptyset: \rightarrow cjt$	<u>ops</u> $_ = _, _ \neq _: elem\ elem \rightarrow bool$
$_ \cup \{ _ \}: cjt\ elem \rightarrow cjt$	<u>ecns</u> $\forall v, v_1, v_2, v_3 \in elem$
$_ \in _ : elem\ cjt \rightarrow bool$	$(v = v) = \text{cierto}$
<u>ecns</u> $\forall s \in cjt; \forall n, m \in elem$	$[v_1 = v_2] \Rightarrow (v_2 = v_1) = \text{cierto}$
$s \cup \{n\} \cup \{n\} = s \cup \{n\}$	$[(v_1 = v_2) \wedge (v_2 = v_3)] \Rightarrow (v_1 = v_3) = \text{cierto}$
$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	$(v_1 \neq v_2) = \neg (v_1 = v_2)$
$n \in \emptyset = \text{falso}$	<u>funiverso</u>
$n \in s \cup \{m\} = (n = m) \vee (n \in s)$	
<u>funiverso</u>	

Fig. 1.29: especificación de los conjuntos genéricos con operación de pertenencia (izquierda) y caracterización de sus elementos con la operación de igualdad (derecha).

Destaquemos que tanto $CJT_ \in$ como $ELEM_ =$ se podrían haber creado como enriquecimientos de CJT y $ELEM$, respectivamente, con la cláusula "usa". Precisamente, el mecanismo de uso se ve afectado por la existencia de los universos de caracterización:

```

universo CJT_∈_NAT es
  usa NAT
  instancia CJT_∈ (ELEM_=) donde elem es nat, = es NAT.ig
  renombra cjt por cjt_nat
funiverso

```

Fig. 1.30: instancia correcta de los conjuntos genéricos con operación de pertenencia.

- Si un universo A de caracterización usa otro B de definición, todos los símbolos de B están incluidos en A , pero no son parámetros. Es el caso de $ELEM_ =$ usando $BOOL$.
- Si un universo A de caracterización usa otro B , también de caracterización, todos los símbolos de B están incluidos en A y además se consideran parámetros. Es el caso de $ELEM_ =$ definido como enriquecimiento de $ELEM$.
- Un universo de definición no puede usar uno de caracterización.

La sintaxis introducida puede producir ambigüedades en algunos contextos. Por ejemplo, supongamos que queremos especificar los pares de elementos cualesquiera dentro de un universo parametrizado y preguntémonos cómo se pueden caracterizar los dos parámetros formales resultantes (los tipos de los componentes de los pares). Una primera opción es definir un universo de caracterización DOS_ELEMS que simplemente incluya los dos géneros; otra posibilidad consiste en aprovechar la existencia del universo $ELEM$ y repetirlo dos veces en la cabecera, indicando que se necesitan dos apariciones de los parámetros que en él residen. Ahora bien, en este caso los dos parámetros formales son indistinguibles: ¿cómo referenciar el tipo de los primeros elementos si tiene el mismo nombre que el de los segundos? Por ello, al nombre del universo de caracterización le debe preceder un identificador, que después se puede usar para distinguir los símbolos, tal como se muestra en la fig. 1.31; notemos que la instancia para definir los pares de enteros da como resultado el mismo TAD especificado directamente en la fig. 1.24. En general, siempre se puede preceder el nombre de un universo de caracterización de otro identificador, si se considera que la especificación resultante queda más legible.

<pre> universo PAR (A, B son ELEM) es tipo par ops <_ , _>: A.elem B.elem → par _ .c₁: par → A.elem _ .c₂: par → B.elem ecns <v₁, v₂> .c₁ = v₁; <v₁, v₂> .c₂ = v₂ funiverso </pre>	<pre> universo DOS_ENTEROS es usa ENTERO instancia PAR (A, B son ELEM) donde A.elem es entero, B.elem es entero renombra par por 2enteros funiverso </pre>
--	--

Fig. 1.31: especificación de los pares de elementos (izq.) y una posible instancia (derecha).

A veces, una instancia sólo se necesita localmente en el universo que la efectúa; en este caso, la palabra clave "instancia" va seguida del calificativo "privada", indicando que los símbolos definidos en el universo parametrizado no son exportables tras la instancia.

Estudiamos a continuación el concepto de *instancia parcial* que generaliza el mecanismo de genericidad presentado hasta ahora. Lo hacemos con un ejemplo de futura utilidad: los conjuntos estudiados en esta sección son conjuntos infinitos, sin ninguna restricción sobre su capacidad; si posteriormente se implementan usando un vector dimensionado con un máximo, y no tratamos ecuacionalmente esta limitación sobre el modelo, obtenemos una implementación que contradice la especificación. Para explicitar los requerimientos de espacio, se incorpora al universo un nuevo parámetro formal que dé el número máximo de elementos que puede tener un conjunto, definiéndose dentro de un nuevo universo de caracterización, VAL_NAT. El resultado aparece en la fig. 1.32; destaca la operación *cuántos* para contar el número de elementos del conjunto, que ha de controlar siempre la aparición de repetidos. Se define una nueva operación visible, *lleno?*, para que exista un medio para saber si un conjunto está lleno sin tener que provocar el error correspondiente.

```

universo CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) es
  usa NAT, BOOL
  tipo cjt
  ops ∅: → cjt
    _∪_: cjt elem → cjt
    _∈_: elem cjt → bool
    lleno?: cjt → bool
    privada cuántos: cjt → nat
  errores ∀s∈cjt; ∀n∈elem: [lleno?(s) ∧ ¬ n∈s] ⇒ s∪{n}
  ecns ∀s∈cjt; ∀n,m∈elem
    s∪{n}∪{n} = s∪{n}; s∪{n}∪{m} = s∪{m}∪{n}
    n∈∅ = falso; n∈s∪{m} = (m = n) ∨ (n ∈ s)
    lleno?(s) = NAT.ig(cuántos(s), val)
    cuántos(∅) = cero
    [n∈s] ⇒ cuántos(s∪{n}) = cuántos(s)
    [¬ n∈s] ⇒ cuántos(s∪{n}) = suc(cuántos(s))
funiverso

universo VAL_NAT caracteriza
  usa NAT, BOOL
  ops val: → nat
  ecns val > cero = cierto
funiverso

```

Fig. 1.32: especificación de los conjuntos acotados (arriba) y del parámetro formal val (abajo).

Ahora pueden realizarse instancias parciales. Por ejemplo, se puede definir un universo para los conjuntos de naturales sin determinar todavía su capacidad máxima, tal como se muestra en la fig. 1.33; el resultado es otro universo genérico que, a su vez, puede instanciarse con un valor concreto de máximo para obtener finalmente un conjunto concreto.

```

universo CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) es ...
universo CJT_NAT_∈_ACOTADO (VAL_NAT) es
  usa NAT
  instancia CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) donde
    elem es nat, = es NAT.ig
  renombra cjt por cjt_nat
funiverso
universo CJT_50_NAT_∈ es
  usa NAT
  instancia CJT_NAT_∈_ACOTADO (VAL_NAT) donde
    val es suc(...suc(cero)...) {50 veces suc}
  renombra cjt_nat por cjt_50_nat
funiverso

```

Fig. 1.33: cabecera para los conjuntos acotados (arriba), instancia parcial para los conjuntos acotados de naturales (en el medio) e instancia determinada a partir de la misma (abajo).

Por último, comentamos el impacto de la parametrización en el significado de una especificación. El modelo de los universos genéricos deja de ser una clase de álgebras para ser una función entre clases de álgebras; concretamente, un morfismo que asocia a las clases de álgebras correspondientes a la especificación de los parámetros formales (es decir, todas aquellas álgebras que cumplen las propiedades establecidas sobre los parámetros formales) las clases de álgebras correspondientes a todos los resultados posibles de una instancia. Así, el paso de parámetros es fundamentalmente un morfismo que tiene como dominio la especificación correspondiente a los parámetros formales y como codominio la especificación correspondiente a los parámetros reales; este morfismo induce otro, que va de la especificación parametrizada a la especificación resultante de la instancia. La principal condición de corrección de la instancia es la protección de los parámetros reales, es decir, que su semántica no se vea afectada en el universo resultado de la instancia.

Técnicamente hablando, el modelo de una especificación parametrizada no es una correspondencia entre clases de álgebras sino entre *categorías* (ing., *category*) de álgebras; la diferencia consiste en que una categoría incluye no sólo álgebras sino también morfismos entre ellas (y alguna cosa más). El modelo se convierte en lo que se denomina un

functor (ing., *functor*), que asocia clases a clases y morfismos a morfismos. Los conceptos teóricos dentro del campo de las categorías y los funtores son realmente complejos y quedan fuera del ámbito de este libro; en [EhM85, caps. 7 y 8] se puede encontrar una recopilación de la teoría de categorías aplicada al campo de la especificación algebraica, así como la construcción del álgebra cociente de términos para especificaciones parametrizadas, que se usa también para caracterizar el modelo.

1.6.5 Combinación de los mecanismos

En los apartados anteriores ha surgido la necesidad de combinar dentro de un mismo universo el mecanismo de instanciación y el de renombramiento. Se puede generalizar esta situación y permitir combinar dentro de un universo todos los mecanismos vistos, en el orden y el número que sea conveniente para la construcción de un universo dado. Por ejemplo, en el apartado 1.6.3 se han definido los racionales en dos pasos: primero se ha escrito *RAC_DEF* como renombramiento de *DOS_ENTEROS* y después se ha enriquecido aquél para formar el universo definitivo, *RACIONALES*. Esta solución presenta el inconveniente de la existencia del universo *RAC_DEF*, que no sirve para nada. En la fig. 1.34 se muestra una alternativa que combina el renombramiento y la definición de las nuevas operaciones directamente sobre *RACIONALES*, sin ningún universo intermedio.

```

universo RACIONALES es
  usa DOS_ENTEROS
  renombra 2enteros por racional, <_ , _> por _/_
           _ .c1 por num, _ .c2 por den
  ops (_)-1, -_: racional → racional
      _+_ , _- , _* : racional racional → racional
  ecns ...
funiverso

```

Fig. 1.34: definición de los racionales a partir de las tuplas de enteros en un único universo.

Más compleja es la siguiente situación: el mecanismo de parametrización ha de poder usarse imbricadamente, de manera que un parámetro formal dentro un universo pueda ser, a su vez, parámetro real de otro universo. Así, supongamos una especificación parametrizada para las cadenas de elementos reducida a las operaciones λ y $_.$ (v. el apartado 1.5.1) y una nueva, *menores*, que dada una cadena y un elemento devuelve un conjunto de todos los elementos de la cadena más pequeños que el elemento dado. El universo genérico resultante (v. fig. 1.35) tiene dos parámetros formales, el género *elem* y una operación de orden, $_<_$, que compara dos elementos y dice cuál es menor; los dos parámetros se

encapsulan en el universo de caracterización $ELEM_<$ (v. fig. 1.36). Dado que la operación *menores* devuelve un conjunto de elementos del mismo tipo de los de la cadena, será necesario efectuar una instancia de una especificación para los conjuntos (por ejemplo, elegimos la de la fig. 1.27); los parámetros reales de esta instancia son los parámetros formales de las cadenas, para garantizar que los elementos de ambos TAD sean los mismos.

```

universo CADENA (A es ELEM_<) es
  tipo cad
  instancia CJT(B es ELEM) donde B.elem es A.elem
  renombra cjt por cjt_elem
  ops  $\lambda$ :  $\rightarrow$  cad
     $\_.$  : A.elem cad  $\rightarrow$  cad
    menores: cad A.elem  $\rightarrow$  cjt_elem
  ecns  $\forall c \in \text{cad}; \forall n, m \in \text{A.elem}$ 
    menores( $\lambda$ , n) =  $\emptyset$ 
     $[m < n] \Rightarrow \text{menores}(m.c, n) = \text{menores}(c, n) \cup \{m\}$ 
     $[\neg m < n] \Rightarrow \text{menores}(m.c, n) = \text{menores}(c, n)$ 
  funiverso

```

Fig. 1.35: especificación de las cadenas con operación menores.

```

universo ELEM_< caracteriza
  usa BOOL
  tipo elem
  ops  $\_<\_$ : elem elem  $\rightarrow$  bool
  ecns  $\forall v, v_1, v_2, v_3 \in \text{elem}$ 
     $v < v = \text{falso}$ 
     $((v_1 < v_2) \Rightarrow \neg(v_2 < v_1)) = \text{cierto}$ 
     $((v_1 < v_2) \wedge (v_2 < v_3) \Rightarrow v_1 < v_3) = \text{cierto}$ 
  funiverso

```

Fig. 1.36: definición de los elementos con operación de orden.

1.7 Ejecución de especificaciones

En las secciones anteriores y en los capítulos sucesivos, se considera una especificación como una herramienta de descripción de tipo de datos, que vendrá seguida por una implementación posterior. Para acabar este capítulo, se estudia la posibilidad de aprovechar el trabajo de especificar para obtener un primer prototipo ejecutable del programa (altamente ineficiente, eso sí) mediante un mecanismo llamado reescritura, que es una evolución de unas reglas de cálculo conocidas con el nombre de deducción ecuacional.

1.7.1 La deducción ecuacional

Dado que la especificación de un TAD se puede considerar como la aserción de propiedades elementales, parece lógico plantearse la posibilidad de demostrar propiedades de estos TAD. Por ejemplo, dada la especificación de la fig. 1.16, se puede demostrar la propiedad de los naturales $\text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{suc}(\text{cero})) = \text{suc}(\text{suc}(\text{suc}(\text{cero})))$ aplicando las ecuaciones adecuadas previa asignación de las variables:

$$\begin{aligned} \text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{suc}(\text{cero})) &= \{\text{aplicando 2 con } n = \text{suc}(\text{suc}(\text{cero})) \text{ y } m = \text{cero}\} \\ &= \text{suc}(\text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{cero})) = \{\text{aplicando 1 con } n = \text{suc}(\text{suc}(\text{cero}))\} \\ &= \text{suc}(\text{suc}(\text{suc}(\text{cero}))) \end{aligned}$$

Este proceso se denomina *deducción ecuacional* (ing., *equational calculus*). La deducción ecuacional permite construir, mediante una manipulación sistemática de los términos, la llamada *teoría ecuacional* de una especificación *SPEC*, abreviadamente Teo_{SPEC} , que es el conjunto de propiedades válidas en todas las álgebras de esta especificación; llamaremos *teorema ecuacional* a cada una de las propiedades $t_1 = t_2$ que forman Teo_{SPEC} y lo denotaremos mediante $\text{SPEC} \mapsto t_1 = t_2$.

Dada la especificación $\text{SPEC} = (\text{SIG}, E)$, $\text{SIG} = (S, OP)$, y los conjuntos V y W de variables sobre SIG , $V, W \in \text{Vars}_{\text{SIG}}$, la teoría ecuacional Teo_{SPEC} se construye según las reglas siguientes:

- Todas las ecuaciones básicas están dentro de Teo_{SPEC} : $\forall t=t' \in E: t=t' \in \text{Teo}_{\text{SPEC}}$.
- Los teoremas ecuacionales son reflexivos, simétricos y transitivos:
 - $\forall t, t_1, t_2, t_3: t, t_1, t_2, t_3 \in T_{\text{SIG}}(V)$:
 - $\Diamond t=t \in \text{Teo}_{\text{SPEC}}$.
 - $\Diamond t_1=t_2 \in \text{Teo}_{\text{SPEC}} \Rightarrow t_2=t_1 \in \text{Teo}_{\text{SPEC}}$.
 - $\Diamond t_1=t_2 \in \text{Teo}_{\text{SPEC}} \wedge t_2=t_3 \in \text{Teo}_{\text{SPEC}} \Rightarrow t_1=t_3 \in \text{Teo}_{\text{SPEC}}$.
- Toda asignación de términos a las variables de un teorema ecuacional da como resultado otro teorema ecuacional:
 - $\forall t_1, t_2: t_1, t_2 \in T_{\text{SIG}}(V). \forall \text{as}_V: \text{as}_V: V \rightarrow T_{\text{SIG}}(W)$:
 - $t_1=t_2 \in \text{Teo}_{\text{SPEC}} \Rightarrow \text{eval}_{V, T_{\text{SIG}}(W)}(t_1) = \text{eval}_{V, T_{\text{SIG}}(W)}(t_2) \in \text{Teo}_{\text{SPEC}}$.
- Una operación aplicada sobre dos conjuntos diferentes de términos que, tomados por parejas, formen parte de Teo_{SPEC} , da como resultado un nuevo teorema ecuacional:
 - $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s}: \forall t_1, t'_1: t_1, t'_1 \in T_{\text{SIG}, s_1}(V) \dots \forall t_n, t'_n: t_n, t'_n \in T_{\text{SIG}, s_n}(V)$
 - $\{ \forall i: 1 \leq i \leq n: t_i = t'_i \in \text{Teo}_{\text{SPEC}} \} \Rightarrow op(t_1, \dots, t_n) = op(t'_1, \dots, t'_n) \in \text{Teo}_{\text{SPEC}}$
- Nada más pertenece a Teo_{SPEC} .

La definición de Teo_{SPEC} recuerda a la de la congruencia \equiv_E inducida por las ecuaciones; la diferencia es que los teoremas son propiedades universales que todavía pueden contener variables, mientras que en el álgebra cociente de términos sólo hay términos sin variables. El nexo queda claro con la formulación del *teorema de Birkhoff*, que afirma: $\text{SPEC} \vdash t_1 = t_2$ si y sólo si $t_1 = t_2$ es válida dentro de todas las SPEC -álgebras.

En general, dada una especificación SPEC , no existe un algoritmo para demostrar que $\text{SPEC} \vdash t_1 = t_2$; el motivo principal es que las ecuaciones se pueden aplicar en cualquier sentido y por eso pueden formarse bucles durante la demostración del teorema. El tratamiento de este problema conduce al concepto de reescritura de términos.

1.7.2 La reescritura

La *reescritura de términos* (ing., *term-rewriting*) o, simplemente, *reescritura*, es una derivación del mecanismo de deducción ecuacional que, en determinadas condiciones, permite usar una especificación como primer prototipo ejecutable de un TAD. Consiste en la manipulación de un término t usando las ecuaciones de la especificación hasta llegar a otro término lo más simplificado posible; en el caso de que t no tenga variables, el resultado estará dentro de la misma clase de equivalencia que t en el álgebra cociente (es decir, representa el mismo valor), y se denominará *forma normal de t* , denotada por $t\downarrow$; usualmente, la forma normal coincide con el representante canónico de la clase del término.

Dada una especificación $\text{SPEC} = (S, OP, E)$, se puede construir un *sistema de reescritura* (ing., *term-rewriting system*) asociado a SPEC orientando las ecuaciones de E para obtener *reglas de reescritura* (ing., *term-rewriting rules*), que indican el sentido de la sustitución de los términos; el punto clave estriba en que esta orientación ha de preservar un orden en el "tamaño" de los términos, de manera que la parte izquierda de la regla sea "mayor" que la parte derecha. Por ejemplo, dada la especificación habitual de la suma de naturales:

- E1) $\text{suma}(\text{cero}, n) = n$
- E2) $\text{suma}(\text{suc}(m), n) = \text{suc}(\text{suma}(m, n))$

es necesario orientar las ecuaciones de izquierda a derecha para obtener las reglas:

- R1) $\text{suma}(\text{cero}, n) \rightarrow n$
- R2) $\text{suma}(\text{suc}(m), n) \rightarrow \text{suc}(\text{suma}(m, n))$

Este sistema de reescritura permite ejecutar el término $\text{suma}(\text{suc}(\text{cero}), \text{suc}(\text{cero}))$ aplicando primero la regla 2 con $m = \text{cero}$ y $n = \text{suc}(\text{cero})$ y, a continuación, la regla 1 con $n = \text{suc}(\text{cero})$ para obtener la forma normal $\text{suc}(\text{suc}(\text{cero}))$ que denota el valor 2 de los naturales.

Dado un sistema de reescritura R , se plantean dos cuestiones fundamentales:

- ¿Todo término tiene alguna forma normal usando las reglas de R ? En tal caso, se dice que R es de *terminación finita* o *noetheriano*. Algunas situaciones son especialmente inconvenientes para la terminación finita, como la conmutatividad y la asociatividad; así, es imposible orientar la ecuación $\text{suma}(m, n) = \text{suma}(n, m)$. Por ello, existen algunas técnicas para la generación de sistemas de reescritura que tratan estas propiedades.
- ¿Todo término tiene una única forma normal según R ? En tal caso, se dice que R es *confluente*, y cumple el lema del diamante: si un término se puede reescribir en otros dos diferentes usando las reglas de R , la forma normal de estos dos es la misma.

Todo sistema de reescritura R asociado a una especificación $SPEC$ que sea confluente y noetheriano se llama *canónico* (ing., *canonical*) y cumple dos propiedades fundamentales: todo término sin variables tiene una única forma normal usando las reglas de R , y la demostración de teoremas ecuacionales es decidible, siendo $SPEC \mapsto t_1 = t_2 \Leftrightarrow t_1 \downarrow_t = t_2 \downarrow_t$, donde la igualdad de las formas normales $=_t$ representa la igualdad sintáctica de los términos. Si una especificación puede convertirse en un sistema de reescritura canónico, la ejecución de especificaciones se reduce al cálculo de formas normales: dado un término t sobre una signatura, su manipulación por un sistema de reescritura canónico resulta en su forma normal (única y siempre existente), que es su valor dentro del álgebra cociente de términos.

Debe tenerse en cuenta que, generalmente, la terminación finita es indecidible, a pesar de que existen condiciones suficientes sobre la forma de las ecuaciones para garantizarla. Además, la confluencia tampoco suele ser decidible; ahora bien, si un sistema de reescritura es de terminación finita entonces sí que lo es. En este contexto, es importante conocer la existencia de un algoritmo llamado algoritmo de *complección* o de *Knuth-Bendix*, que intenta convertir una especificación en un sistema canónico de reescritura. La aplicación de este algoritmo puede no acabar, en cuyo caso no se puede afirmar nada sobre la confluencia o noetherianidad (simplemente el método ha fracasado en esta especificación concreta), o bien acabar de dos maneras posibles: correctamente, obteniéndose el sistema de reescritura canónico resultado de orientar las reglas, o bien sin éxito, porque ha sido imposible orientar alguna ecuación y entonces no se puede asegurar la terminación finita. En el caso de que finalice con éxito, el algoritmo puede haber añadido más ecuaciones para garantizar la confluencia del sistema, que serán deducibles de las ecuaciones iniciales.

Se han publicado varios artículos de introducción al tema de la reescritura; se puede consultar, por ejemplo, la panorámica ofrecida por J.P. Jouannaud y P. Lescanne en "Rewriting Systems" (*Technique et Science Informatiques*, 6(3), 1987), o bien el capítulo 6 del libro *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Editorial Elsevier, 1990, capítulo escrito por N. Dershowitz y J.-P. Jouannaud.

Ejercicios

1.1 Dada la signatura:

universo Y es
tipo y, nat
ops $\text{crea}: \rightarrow y$
 $\text{modif}: y \text{ nat nat} \rightarrow y$
 $f: y \text{ nat} \rightarrow \text{nat}$
 $h: y \text{ nat} \rightarrow y$
 $\text{cero}: \rightarrow \text{nat}$
 $\text{suc}: \text{nat} \rightarrow \text{nat}$
funiverso

- a) Clasificar las operaciones por su signatura (es decir, construir los conjuntos $OP_{W \rightarrow S}$).
- b) Escribir unos cuantos términos de género y .
- c) Repetir b) considerando un conjunto de variables $X = (X_y, X_{\text{nat}}, X_y = \{z\}, X_{\text{nat}} = \{x, y\})$.
- d) Escribir al menos tres Y -álgebras que respondan a modelos matemáticos o informáticos conocidos, y dar la interpretación de los géneros y los símbolos de operación de Y .
- e) Describir el álgebra de términos T_Y .
- f) Evaluar el término $t = f(h(\text{crea}, \text{modif}(\text{crea}, \text{suc}(\text{cero}), \text{suc}(\text{suc}(\text{cero}))))$, crea) dentro de todas las álgebras de d).
- g) Dar una función de asignación de las variables de c) dentro de las álgebras de d).
- h) Evaluar el término $t' = f(h(z, \text{modif}(\text{crea}, x, \text{suc}(\text{cero}))), \text{crea})$ dentro de las álgebras de d), usando las asignaciones de g).

1.2 a) Dada la signatura:

universo Y es
tipo y, nat
ops $\text{crea}: \rightarrow y$
 $\text{añ}: y \text{ nat} \rightarrow y$
 $f: y \rightarrow \text{nat}$
 $\text{cero}: \rightarrow \text{nat}$
 $\text{suc}: \text{nat} \rightarrow \text{nat}$
funiverso

- i) Clasificar las operaciones por su signatura.
- ii) Escribir al menos cinco Y -álgebras que respondan a modelos matemáticos o informáticos conocidos, y dar la interpretación de los géneros y los símbolos de operación de Y .
- iii) Escribir el álgebra de términos T_Y , encontrando una expresión general recursiva para

los términos de los conjuntos base.

iv) Evaluar el término $t = f(añ(añ(añ(crea, suc(cero)), cero), suc(suc(cero))))$ dentro de todas las álgebras de ii).

b) Dada la especificación $SY = (Y, EY, EY = \{ f(añ(s, n)) = suc(f(s)), f(crea) = cero \})$, se pide:

i) Decir si las ecuaciones se satisfacen o no en las álgebras de a.ii).

ii) Encontrar el álgebra cociente de términos T_{SY} , y escoger representantes canónicos para las clases.

iii) Determinar si una o más de una de las álgebras de a.ii) es isomorfa a T_{SY} (en caso contrario, buscar otra Y -álgebra que lo sea).

iv) Clasificar las álgebras de a.ii) en: modelos iniciales de SY , SY -álgebras e Y -álgebras.

1.3 Dada la especificación:

universo MISTERIO es
tipo misterio
ops $z: \rightarrow \text{misterio}$
 $f: \text{misterio} \rightarrow \text{misterio}$
funiverso

a) ¿Cuál es su modelo inicial?

b) ¿Cuál es el modelo inicial de *MISTERIO* si añadimos la ecuación $f(z) = z$?

c) ¿Cuál es el modelo inicial de *MISTERIO* si añadimos la ecuación $f(f(z)) = z$?

1.4 Dada la especificación *QUI_LO_SA*:

universo QUI_LO_SA es
tipo qui_lo_sa
ops $z: \rightarrow \text{qui_lo_sa}$
 $f: \text{qui_lo_sa} \rightarrow \text{qui_lo_sa}$
 $h: \text{qui_lo_sa} \text{ qui_lo_sa} \rightarrow \text{qui_lo_sa}$
ecns $\forall r, s \in \text{qui_lo_sa}$
 $h(r, z) = r$
 $h(r, f(s)) = f(h(r, s))$
funiverso

justificar brevemente si los modelos siguientes son o no son iniciales:

a) Secuencias de ceros ($0^*, \lambda, añ, conc$); *añ* es añadir un "0" y *conc* es la concatenación.

b) Naturales con operación de suma ($\mathcal{N}, 0, +1, +$).

c) Matrices 2x2 de naturales ($\mathcal{M}_{2 \times 2}(\mathcal{N}), (0 \ 0; 0 \ 0), (+1 \ +1; +1 \ +1), +$).

d) Igual que c), con la restricción de que todos los elementos de la matriz deben ser iguales.

1.5 Sea la especificación de la figura siguiente, donde se supone que las especificaciones *BOOL* y *NAT* especifican el álgebra de Bool \mathcal{B} y los naturales \mathcal{N} , respectivamente:

universo QUÉ_SOY_YO es
usa BOOL, NAT
tipo qué_soy_yo
ops λ : \rightarrow qué_soy_yo
 añ: qué_soy_yo nat \rightarrow qué_soy_yo
 está?: qué_soy_yo nat \rightarrow bool
ecns $\forall n, m \in \text{nat}; \forall r \in \text{qué_soy_yo}$
 está?(λ , n) = falso
 está?(añ(r , n), m) = NAT.ig(m , n) \vee está?(r , m)
funiverso

a) Demostrar que las álgebras siguientes son QUÉ_SOY_YO-álgebras y comprobar que no son el modelo inicial.

- i) $A_{bool} \cong \mathcal{B}$, $A_{nat} \cong \mathcal{N}$, $A_{qué_soy_yo} \cong \mathcal{P}(\mathcal{N})$ {conjuntos de naturales}
 $\lambda_A \equiv \emptyset$ {conjunto vacío}
 $añ_A: \mathcal{P}(\mathcal{N}) \times \mathcal{N} \rightarrow \mathcal{P}(\mathcal{N})$
 $(C, n) \rightarrow añ_A(C, n) \equiv C \cup \{n\}$
 $está?_A: \mathcal{P}(\mathcal{N}) \times \mathcal{N} \rightarrow \mathcal{B}$
 $(C, n) \rightarrow está?_A(C, n) \equiv n \in C$
- ii) $B_{bool} \cong \mathcal{B}$, $B_{nat} \cong \mathcal{N}$, $B_{qué_soy_yo} \cong \mathcal{N}^*_{<}$ {secuencias ordenadas de naturales}
 $\lambda_B \equiv \lambda$ {secuencia vacía}
 $añ_B: \mathcal{N}^*_{<} \times \mathcal{N} \rightarrow \mathcal{N}^*_{<}$
 $(l, n) \rightarrow añ_B(l, n) \equiv \text{añadir } n \text{ a } l \text{ ordenadamente}$
 $está?_B: \mathcal{N}^*_{<} \times \mathcal{N} \rightarrow \mathcal{B}$
 $(l, n) \rightarrow está?_B(l, n) \equiv \text{si } n \text{ está dentro de } l \text{ entonces cierto, sino falso}$
- iii) $C_{bool} \cong \mathcal{B}$, $C_{nat} \cong \mathcal{N}$, $C_{qué_soy_yo} \cong \mathcal{M}(\mathcal{N})$ {multiconjuntos de naturales¹⁴}
 $\lambda_C \equiv \emptyset$ {multiconjunto vacío}
 $añ_C: \mathcal{M}(\mathcal{N}) \times \mathcal{N} \rightarrow \mathcal{M}(\mathcal{N})$
 $(M, n) \rightarrow añ_C(M, n) \equiv M \cup_M \{n\}$ $\{\cup_M \text{ es la unión de multiconjuntos}^{12}\}$
 $está?_C: \mathcal{M}(\mathcal{N}) \times \mathcal{N} \rightarrow \mathcal{B}$
 $(M, n) \rightarrow está?_C(M, n) \equiv n \in C$

b) Comprobar que $T_{QUÉ_SOY_YO, qué_soy_yo} \approx \mathcal{N}^*$ y que el modelo inicial de esta especificación es $T_{QUÉ_SOY_YO} = (\mathcal{B}, \mathcal{N}, \mathcal{N}^*)$ con las operaciones adecuadas.

c) Determinar el modelo inicial de la especificación QUÉ_SOY_YO al añadirle la ecuación $añ(añ(r, x), y) = añ(añ(r, y), x)$. Justificar la elección dando el isomorfismo con $T_{QUÉ_SÓY_YO}$.

¹⁴ Un multiconjunto es un conjunto que admite repeticiones; es decir, el multiconjunto $\{1\}$ es diferente del multiconjunto $\{1, 1\}$. La unión de multiconjuntos conserva, pues, las repeticiones.

d) Determinar y justificar el modelo inicial de la especificación *QUÉ_SOY_YO* cuando se le añade la ecuación $a\tilde{n}(a\tilde{n}(r, x), x) = a\tilde{n}(r, x)$.

e) Determinar y justificar el modelo inicial de la especificación *QUÉ_SOY_YO* cuando se le añaden las dos ecuaciones de c) y d) a la vez.

1.6 Sea la especificación de la teoría de grupos:

universo *TEORIA_GRUPOS* es
tipo *g*
ops *e*: $\rightarrow g$ {elemento neutro}
+: $g\ g \rightarrow g$
i: $g \rightarrow g$ {inverso}
ecns $\forall x, y, z \in g: x + e = x; x + i(x) = e; (x + y) + z = x + (y + z)$
funiverso

Usando deducción ecuacional, determinar si las ecuaciones siguientes son teoremas ecuacionales de este modelo, especificando claramente los pasos de las demostraciones.

a) $(x + y) + i(y) = x$. **b)** $x + i(e) = x$. **c)** $e + i(i(x)) = x$. **d)** $x + i(i(y)) = x + y$. **e)** $e + x = x$. **f)** $i(e) = e$. **g)** $i(x) + x = e$.

1.7 Especificar los vectores de índices los naturales de 1 a n que ofrecen los lenguajes de programación imperativos (Pascal, C, etc.) con operaciones de: *crear* un vector con todas las posiciones indefinidas, *modificar* el valor del vector en una posición dada y *consultar* el valor del vector en una posición dada; si se consulta el valor de una posición todavía indefinida, dar error. Expresar el resultado dentro de un universo parametrizado. Hacer una instancia.

1.8 Especificar las matrices $n \times m$ de naturales con las operaciones habituales. Expresar el resultado dentro de un universo parametrizado. Efectuar una instancia.

1.9 Dada la especificación de los conjuntos genéricos con operación de pertenencia y un número máximo de elementos, *CONJUNTO_ε_ACOTADO(ELEM_=, VAL_NAT)*, escribir otro universo de cabecera *CONJUNTO2(ELEM_=, VAL_NAT)* que use el anterior y defina las operaciones \cap (intersección de conjuntos), \cup (unión de conjuntos) y \supset (inclusión de conjuntos). Hacer una instancia para los conjuntos de, como máximo, 50 naturales, y otra para los conjuntos de, como máximo, 30 conjuntos de, como máximo, 20 naturales.

1.10 Especificar una calculadora de polinomios, que disponga de n memorias para guardar resultados intermedios, y de un polinomio actual (que se ve por el visualizador de la calculadora). Las operaciones de la calculadora son: *crear* la calculadora con todas las memorias indefinidas y de polinomio actual cero, *añadir* un término al polinomio actual, *recuperar* el contenido de una memoria sobre el polinomio actual, *guardar* el polinomio actual sobre una memoria, *sumar*, *restar* y *multiplicar* los polinomios residentes en dos memorias,

derivar el polinomio residente en una memoria (estas cuatro últimas operaciones dejan el resultado sobre el polinomio actual) y *evaluar* el polinomio residente en una memoria en un punto dado. Las operaciones han de controlar convenientemente los errores típicos, básicamente referencias a memorias indefinidas o inexistentes. Usar la especificación de los polinomios del apartado 1.5.1, y enriquecerla si es necesario.

1.11 a) Enriquecer la especificación de las cadenas dada en el apartado 1.5.1 (considerándola parametrizada por el tipo de los elementos) con las operaciones:

elimina: cadena \rightarrow *cadena*, elimina de la cadena todas las apariciones consecutivas de un mismo elemento y deja sólo una

selecciona: cadena \rightarrow *cadena*, forma una nueva cadena con los elementos que aparecen en una posición par de la cadena; por ejemplo, *selecciona(especificación)* = *seiiain*

es_prefijo?: cadena cadena \rightarrow *bool*, dice si la primera cadena es prefijo de la segunda; considerar que toda cadena es prefijo de sí misma

es_subcadena?: cadena cadena \rightarrow *bool*, dice si la primera cadena es subcadena de la segunda; considerar que toda cadena es subcadena de sí misma

especula: cadena \rightarrow *cadena*, obtiene la imagen especular de una cadena; así, *especular(especificación)* = *nóicacifcepse*

b) A continuación, hacer una instancia del universo del apartado anterior con cadenas de cualquier cosa, de manera que el resultado sea un género de nombre *cad_cad* que represente las cadenas de cadenas de cualquier cosa.

c) Enriquecer el universo del apartado b) con las operaciones siguientes:

aplana: cad_cad \rightarrow *cadena*, concatena los elementos de la cadena de cadenas dada; así, *aplana([hola, que, λ , tal])* = *holaquetal*, donde los corchetes denotan una cadena de cadenas y λ denota la cadena vacía

distr_izq: elem cad_cad \rightarrow *cad_cad*, dado un elemento *v* y una cadena de cadenas *L*, añade *v* por la izquierda a todas las cadenas de *L*; por ejemplo, *distr_izq(x, [hola, que, λ , tal])* = [*xhola, xque, x, xtal*]

subcadenas: cadena \rightarrow *cad_cad*, dada una cadena *l*, devuelve la cadena de todas las subcadenas de *l* conservando el orden de aparición de los elementos; por ejemplo, *subcadenas(abc)* = [*abc, ab, ac, bc, a, b, c, λ*]

inserta: elem cad_cad \rightarrow *cad_cad*, dado un elemento *v* y una cadena de cadenas *L*, devuelve la cadena con todas las cadenas resultantes de insertar *v* en todas las posiciones posibles de las cadenas de *L*; por ejemplo, *inserta(x, [abc, d])* = [*xabc, axbc, abxc, abcx, xd, dx*]

1.12 a) Especificar un universo que describa el comportamiento de una urna que pueda ser utilizada por diferentes colectivos (sociedades anónimas, partidos políticos, asambleas de alumnos, etc.) para aceptar o rechazar propuestas a través de votación, cuando les falle la vía del consenso o el engaño. Este universo ha de definir un TAD *urna* con operaciones:

inic: colectivo \rightarrow *urna*, prepara la urna para ser usada por un colectivo

vota: urna miembro bool \rightarrow *urna*, un miembro emite un voto favorable o desfavorable

may_absoluta: urna \rightarrow *bool*, dice si el número de votos favorables supera la mitad de la totalidad de votos del colectivo

may_simple: urna \rightarrow *bool*, dice si el número de votos favorables supera la mitad de la totalidad de votos emitidos por el colectivo

El universo ha de poder ser instanciado con cualquier clase de miembro y colectivo, y ha de prever que hay colectivos en los que miembros diferentes pueden disponer de un número diferente de votos. De todas formas, para garantizar un cariz mínimamente democrático, la urna sólo ha de poder ser usada si no hay ningún miembro que tenga más del 50% de los votos totales del colectivo. Especificar la urna en un universo genérico con los parámetros que se consideren oportunos.

b) Construir una especificación que sirva para modelizar una sociedad especuladora, fundada por tres personas y regida por las siguientes directrices:

- Los miembros de la sociedad (socios, entre los cuales están los fundadores) se identifican mediante una cadena de caracteres.
- Los tres fundadores disponen de dos votos cada uno a la hora de tomar decisiones; el resto de socios sólo dispone de un voto por cabeza.

La sociedad estará dotada de un mecanismo de toma de decisiones por mayoría simple.

El universo donde se encapsule esta especificación ha de tener las operaciones:

fundación: socio socio socio \rightarrow *sociedad*, crea la sociedad con los tres socios fundadores

añadir: sociedad socio \rightarrow *sociedad*, añade un nuevo socio; error si ya estaba

es_socio?: sociedad socio \rightarrow *bool*, dice si el socio está en la sociedad

inicio_votación: sociedad \rightarrow *sociedad*, prepara la futura votación

añade_voto: sociedad socio bool \rightarrow *sociedad*, cuenta el voto del socio

prospera: sociedad \rightarrow *bool*, contesta si la propuesta se acepta

Hay un universo *SOCIO* que define el género *socio* y la operación *ident: socio* \rightarrow *cadena* (entre otras). También se puede utilizar el resultado del apartado a) y el universo *PAR* de la fig. 1.31.

1.13 Una hoja de cálculo puede considerarse como una matriz rectangular de celdas, las cuáles se identifican por su coordenada (par de naturales que representan fila y columna). Dentro de una celda puede no haber nada, puede haber un valor entero o puede haber una fórmula. De momento, para simplificar la especificación, supondremos que las fórmulas son de la forma $a + b$, donde a y b son coordenadas de celdas. Definimos la evaluación de una celda recursivamente del siguiente modo: si la celda está vacía, la evaluación es 0; si la celda contiene el entero n , la evaluación es n ; en cualquier otro caso, la celda contiene una fórmula $c_1 + c_2$ y el resultado es la evaluación de la celda de coordenada c_1 , sumado a la evaluación de la celda de coordenada c_2 . La signatura de este tipo es:

crea: nat nat \rightarrow *hoja*, crea una hoja de cálculo con las filas y las columnas numeradas del uno a los naturales dados; inicialmente, todas las celdas están vacías

ins_valor: hoja coordenada enter \rightarrow *hoja*, inserta el entero en la celda de coordenada dada, sobrescribiendo el contenido anterior de la celda

ins_fórmula: hoja coordenada coordenada coordenada \rightarrow *hoja*, siendo c_1 , c_2 y c_3 las coordenadas dadas, inserta la fórmula $c_2 + c_3$ en la celda de coordenada c_1 de la hoja, sobrescribiendo el contenido anterior

valor: hoja coordenada \rightarrow *entero*, devuelve la evaluación de la celda dada

Las coordenadas se definen como instancia del universo *PAR* (v. fig. 1.31), siendo los dos parámetros formales el tipo de los naturales.

a) Especificar el tipo *hoja*, controlando todos los errores posibles (referencias a coordenadas fuera de rango, ciclos en las fórmulas, etc.).

b) Pensar en la extensión en caso de permitir fórmulas arbitrarias (quizás se podría definir el tipo *fórmula*, con algunas operaciones adecuadas...).

1.14 Se quiere especificar una tabla de símbolos para lenguajes modulares (como Merlí) que organice los programas como jerarquías de módulos. Las operaciones escogidas son:

crea: \rightarrow ts, crea la tabla vacía

declara: ts módulo cadena caracts \rightarrow *ts*, registra que el identificador ha sido definido dentro del módulo con una descripción dada

usa: ts módulo módulo \rightarrow *ts*, registra que el primer módulo usa el segundo, es decir, dentro del módulo usador son visibles los objetos declarados dentro del módulo usado

usa?: ts módulo \rightarrow *cjt_módulos*, da todos los módulos que usa el módulo dado. Hay que considerar que el uso de módulos es transitivo, es decir, si m_1 usa m_2 y m_2 usa m_3 , entonces m_1 usa m_3 , aunque no esté explícitamente declarado. Para simplificar, puede considerarse que todo módulo se usa a sí mismo

consulta: ts módulo cadena \rightarrow *cjt_caracts*, devuelve el conjunto de todas las descripciones para el identificador que sean válidas dentro del módulo dado (es decir, las descripciones declaradas en todos los módulos usados por el módulo dado)

(Notar que no hay una operación explícita de declaración de módulos; un módulo existe si hay alguna declaración -de uso o de identificador- que así lo indique.)

Se puede suponer la existencia de las operaciones necesarias sobre los conjuntos y las cadenas.

1.15 Se quiere especificar un diccionario de palabras compuestas de n partes como máximo, $n < 10$. Cada parte es una cadena de caracteres. El diccionario ha de presentar operaciones de acceso individual a las palabras y de recorrido alfabético; concretamente:

crea: \rightarrow dicc, crea el diccionario vacío

inserta: dicc cadena ... cadena (10 cadenas) \rightarrow *dicc*, registra la inserción de una palabra

compuesta de k partes, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía; da error si la palabra ya estaba en el diccionario

borra: dicc cadena ... cadena (10 cadenas) \rightarrow *dicc*, borra una palabra de k partes, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía; error si la palabra no está en el diccionario

está?: dicc cadena ... cadena (10 cadenas) \rightarrow *bool*, comprueba si una palabra de k partes está o no en el diccionario, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía

lista: dicc \rightarrow *lista_cadenas*, devuelve la lista ordenada alfabéticamente de todas las palabras del diccionario sin considerar las partes que las componen

Es importante definir correctamente la igualdad de palabras: dos palabras son iguales si y sólo si están compuestas por las mismas partes; así, la palabra *méd/ic/amente* es diferente de la palabra *médic/a/mente* porque, a pesar de tener los mismos caracteres, su distribución en partes no concuerda (considerando '/' como separador de partes).

Se puede suponer la existencia de las operaciones necesarias sobre las cadenas y las listas.

1.16 a) En el juego del *mastermín* participan dos jugadores A y B con fichas de colores. A imagina una combinación "objetivo" de cinco fichas (sin repetir colores) y B debe adivinarla. El jugador B va proponiendo jugadas (combinaciones de cinco fichas de colores no repetidos; se dice que cada ficha va en una posición determinada), y el jugador A responde con el número de aciertos y de aproximaciones que contiene respecto al objetivo. Se produce acierto en cada posición de la jugada propuesta que contenga una ficha del mismo color que la que hay en la misma posición del objetivo. Se produce aproximación en las posiciones de la jugada propuesta que no sean aciertos, pero sí del mismo color que alguna otra posición del objetivo. En el siguiente ejemplo, la propuesta obtiene un acierto y dos aproximaciones:

Posiciones	1	2	3	4	5
OBJETIVO:	Amarillo	Verde	Rojo	Azul	Fucsia
PROPUESTA:	Azul	Verde	Amarillo	Marrón	Naranja

La secuencia de jugadas hechas hasta un momento dado proporciona una pista para la siguiente propuesta del jugador B , de manera que, cuando B propone una jugada, puede comprobar previamente que no se contradiga con los resultados de las jugadas anteriores. Una propuesta coherente a partir del ejemplo anterior (donde B supone que el acierto era el marrón y las aproximaciones el amarillo y el verde) sería:

Posiciones	1	2	3	4	5
PROPUESTA:	Amarillo	Púrpura	Gris	Marrón	Verde

Especificar este juego con la signatura siguiente:

crea: jugada $\rightarrow mm$, deposita en el tablero la jugada objetivo

juega: mm jugada $\rightarrow mm$, deposita en el tablero la jugada siguiente propuesta

aciertos, aproxs: mm $\rightarrow nat$, da el número de aciertos y aproximaciones de la última jugada propuesta, respectivamente

coherente: mm jugada $\rightarrow bool$, comprueba si una jugada es coherente con las anteriores

Suponer que se dispone de una constante para cada color: *blanco, amarillo, ...* $\rightarrow color$, así como de una operación de comparación, *ig: color color* $\rightarrow bool$. Se dispone, además, de la definición del género *jugada*, con unas operaciones básicas que se pueden suponer correctamente especificadas:

crea: color color color color color $\rightarrow jugada$, constructora de jugadas

qué_color: jugada nat $\rightarrow color$, devuelve el color de la ficha que hay en la posición dada de la jugada

b) Modificar la especificación permitiendo la repetición de colores en las jugadas.

Capítulo 2 Implementación de tipos abstractos de datos

La especificación de un tipo abstracto es el paso previo a la escritura de cualquier programa que lo manipule, porque permite describir inequívocamente su comportamiento; una vez establecida, se pueden construir diversas implementaciones usando un lenguaje de programación convencional. En este capítulo se introducen los mecanismos que proporciona la notación Merlí para codificar los tipos de datos, que son los habituales en los lenguajes de programación imperativos más utilizados; se estudian las relaciones que se pueden establecer entre la implementación y la especificación de un TAD; se definen diversas estrategias de medida de la eficiencia de las implementaciones que permiten compararlas y determinar bajo qué circunstancias una implementación es más adecuada que otra; y se ponen de manifiesto algunas situaciones recurrentes que exigen un tratamiento específico para asegurar que la programación con TAD no genera programas ineficientes.

A lo largo del capítulo se supone que el lector conoce los constructores habituales que ofrecen los lenguajes de programación imperativos más habituales, como Modula-2, Pascal o C y, en general, todos aquellos que se pueden considerar derivados del Algol; por ello, no se explicarán en detalle los esquemas adoptados en Merlí, sino que tan sólo se mostrará su sintaxis, fácilmente traducible a cualquiera de ellos. Eso sí, se insistirá en los aspectos propios de la metodología de uso de tipos abstractos, derivados básicamente de la distribución en universos de las aplicaciones.

2.1 El lenguaje de implementación

En el primer capítulo se ha destacado la diferencia entre la especificación y la implementación de un TAD, que resulta en su visión a dos niveles diferentes. En concreto, dada la especificación (única) de un tipo se pueden construir diversas implementaciones y así escoger la más adecuada en cada contexto de uso. Cada una de estas implementaciones se encierra en un universo diferente, llamado *universo de implementación*, en contraposición a los universos de especificación y de caracterización explicados detalladamente en el capítulo anterior. Notemos que un universo de implementación no puede existir de forma independiente, sino que siempre se referirá a una especificación ya existente; para

establecer claramente el nexo entre un universo de implementación y el universo de especificación correspondiente, se consignará siempre en la cabecera del primero el nombre del segundo precedido por la palabra clave "implementa".

La construcción de un universo de implementación consta de dos fases (v. fig. 2.1):

- Elección de una representación para los diferentes géneros definidos en la especificación. Es habitual el uso de tipos auxiliares para estructurar el resultado, que son invisibles a los usuarios del universo. Aunque no sea estrictamente necesario, se usará la palabra clave "privado" para distinguirlos. También se pueden definir constantes, implícitamente privadas.
- Codificación de todas y cada una de las diversas operaciones visibles de la especificación, en términos de las representaciones que se acaban de escribir. Igualmente puede ser necesario, e incluso recomendable, usar una o más operaciones auxiliares que surjan de la implementación de las operaciones usando la técnica de diseño descendente; las operaciones auxiliares de la implementación no han de ser forzosamente las mismas de la especificación, y en general no lo son.

```

universo U_IMP implementa U
    usa U1, ..., Un
    const c1 vale v1, ... fconst
    tipo T1 es ... ftipo
    ...
    tipo privado Taux1 es ... ftipo
    ...
    función F1 ...
    ...
    función / acción privada Faux1 ...
    ...
funiverso

```

Fig. 2.1: esquema general de los universos de implementación.

Tanto en la representación como en la codificación de las operaciones pueden usarse identificadores de tipo y de operaciones visibles, definidos en otros universos cuyos nombres se escriben en una cláusula de usos al inicio del universo. En principio, los nombres se referirán a universos de especificación, ya que los programas dependen del comportamiento de los tipos que usen y no de su implementación; ahora bien, ya veremos a lo largo del texto que, en algunas situaciones y por razones de eficiencia, se exigirá que uno o más tipos usados en un universo estén implementados según una estrategia

determinada¹, en cuyo caso se consigna la implementación escogida mediante la cláusula "implementado con". Además de los usos, en un universo de implementación pueden aparecer ocultaciones, renombramientos e instancias; por lo que respecta a estas últimas, pueden implementar alguno de los tipos definidos en la especificación, en cuyo caso se determina la implementación elegida mediante la cláusula "implementado con".

Si el universo de especificación define un TAD parametrizado, los universos de implementación correspondientes también lo serán y en la implementación aparecerán, como mínimo, todos los parámetros formales de la especificación. En ocasiones, alguna técnica de implementación puede requerir parámetros formales adicionales que no afecten la especificación del tipo (por ejemplo, las técnicas de dispersión vistas en el capítulo 5).

2.1.1 Representación de tipos

Para representar un TAD, Merlí ofrece diversos tipos predefinidos, y también algunos constructores de tipo que permiten al usuario construir sus propias estructuras. Los tipos predefinidos del lenguaje son los booleanos, los naturales, los enteros, los caracteres y los reales, todos ellos con las operaciones habituales. Por lo que respecta a los constructores, se dispone de los siguientes (v. fig. 2.2):

- Por enumeración de sus valores: se escriben todos los valores del tipo uno detrás del otro. Se dispone de operaciones de comparación $=$ y \neq .
- Tuplas: permiten la definición de un tipo como producto cartesiano. Cada componente o *campo* (ing., *field*) de la tupla se identifica mediante un nombre y se declara de un tipo determinado; dado un objeto v de tipo tupla, la referencia a su campo c se denota por $v.c$. El lenguaje permite la escritura de constantes tuplas: dada una tupla de n campos c_1, \dots, c_n , el literal $\langle v_1, \dots, v_n \rangle$ representa la tupla tal que su campo c_i vale v_i , $1 \leq i \leq n$; evidentemente, los tipos de los v_i han de concordar con los tipos de los c_i . Las tuplas pueden ser *tuplas variantes*, es decir, tuplas tales que el nombre y el tipo de campos componentes varía en función del valor de uno de ellos, llamado *discriminante*.
- Vectores (ing., *array*): permiten la definición de tipos como funciones $f: I \rightarrow V$; los elementos de I se llaman *índices* y los de V simplemente *valores*. Gráficamente se puede considerar un vector como una tabla que muestra el valor asociado a cada índice y cada celda se llama *componente* o *posición*. Dado un objeto A de tipo vector, la referencia al valor asociado al índice i se hace mediante $A[i]$. Se exige que el tipo de los índices sea natural, entero, carácter o bien un tipo por enumeración; generalmente no se usará todo el tipo para indexar el vector, sino que se especificará una cota inferior y una cota superior de manera que la referencia a un índice fuera del intervalo definido por estas cotas da error. El lenguaje permite la escritura de vectores literales: dado un

¹Incluso en este caso, la representación del tipo usado quedará inaccesible en virtud del principio de la transparencia de la representación propia del diseño modular con TAD.

vector de n componentes, el literal $[v_1, \dots, v_n]$ representa el vector tal que su posición i -ésima vale v_i , $1 \leq i \leq n$; si en la constante tan sólo se quiere destacar el valor de una posición i concreta, pueden escribirse los literales $[\dots, v_i, \dots]$ o $[\dots, i \rightarrow x, \dots]$.

tipo semana es (lu, ma, mi, ju, vi, sa, do) ftipo

(a) *tipo por enumeración de sus valores*

tipo fecha es tupla

día_s es semana

día_m, mes, año son natural

ftupla

ftipo

(b.1) *tipo tupla*

tipo trabajador es

tupla

nombre, apellido son cadena

caso categoría de tipo (jefe, analista, programador) igual a

jefe entonces #subordinados es natural

analista entonces #proyectos, #programadores son natural

programador entonces #líneas_código es natural; senior? es bool

fcaso

ftupla

ftipo

(b.2) *tipo tupla con partes variantes (el campo categoría es el discriminante)*

tipo estadística es vector [semana] de natural ftipo

(c.1) *tipo vector; su índice es un tipo por enumeración*

tipo cadena es vector [de 1 a máx_cadena] de carácter ftipo

(c.2) *tipo vector; su índice es un intervalo de un tipo predefinido*

tipo empresa es vector [de 1 a máx_empresa] de tupla

tr es trabajador

sueldo es real

ftupla

ftipo

(c.3) *tipo vector; sus valores usan otro tipo definido por el usuario*

Fig. 2.2: ejemplos de uso de los constructores de tipo de Merlí.

2.1.2 Sentencias

Se describe a continuación el conjunto de sentencias o instrucciones que ofrece Merlí, cuya sintaxis se puede consultar en la fig. 2.3.

- Asignación: modificación del valor de un objeto de cualquier tipo.
- Secuencia: ejecución incondicional de las instrucciones que la forman, una tras otra.
- Sentencias condicionales: por un lado, se define la sentencia condicional simple, que especifica una instrucción que se ha de ejecutar cuando se cumple una condición booleana, y otra, opcional, para el caso de que no se cumpla. Por otro, se dispone de la sentencia condicional compuesta, que generaliza la anterior especificando una serie de condiciones y, para cada una, una sentencia para ejecutar en caso de que se cumpla. Las condiciones han de ser mutuamente exclusivas; también se puede incluir una sentencia para ejecutar en caso en que ninguna condición se cumpla.
- Sentencias iterativas o bucles. Existen diversos tipos:
 - ◊ Simple: se especifica el número de veces que se debe repetir una sentencia.
 - ◊ Gobernada por una expresión booleana: se repite la ejecución de una sentencia mientras se cumpla una condición de entrada al bucle, o bien hasta que se cumpla una condición de salida del bucle.
 - ◊ Gobernada por una variable: se define una variable, llamada *variable de control*, que toma valores consecutivos dentro de los naturales, de los enteros, de un tipo por enumeración o bien de un intervalo de cualquiera de ellos; el número de iteraciones se fija inicialmente y no cambia. En los tipos por enumeración, los valores que toma la variable siguen el orden de escritura en la declaración del tipo.

Cada ejecución de las instrucciones que aparecen dentro de la sentencia iterativa se llama *vuelta* o también *iteración*.

Al hablar de sentencias iterativas son básicos los conceptos de *invariante* y de *función de cota*, que abreviaremos por I y \mathcal{F} , respectivamente. El invariante es un predicado que se cumple tanto antes como después de cada vuelta y que describe la misión del bucle. La función de cota implementa el concepto de terminación del bucle, por lo que se define como una función estrictamente decreciente que tiene como codominio los naturales positivos y que, generalmente, se formula a partir de la condición del bucle. Tanto el uno como la otra se escriben usando las variables que intervienen en la sentencia iterativa. Es conveniente que ambos se determinen previamente a la escritura del código, porque definen inequívocamente el comportamiento del bucle. En este texto, no obstante, sólo se presentarán en aquellos algoritmos que, por su dificultad, se considere que aportan información relevante para su comprensión (por ello, se puede preferir una explicación informal a su expresión enteramente formal). Para profundizar en el tema, v. por ejemplo [Bal93] o [Peñ98], que proporcionan varias fuentes bibliográficas adicionales.

El lenguaje no admite ningún tipo de instrucción para romper la estructura dada por el uso de estas construcciones; es decir, no existen las sentencias *goto*, *skip*, *break* o similares que, a cambio de pequeñas reducciones del código resultante, complican la comprensibilidad, la depuración y el mantenimiento de los programas. Tampoco se definen las instrucciones habituales de entrada/salida, ya que no se necesitarán en el texto.

objeto := expresión

(a) asignación

sent₁; sent₂; ... ; sent_n

(b) secuencia; en lugar de ';', se pueden separar por saltos de línea

si expresión booleana entonces sent₁

si no sent₂ {opcional}

fsi

(c.1) sentencia alternativa simple

opción

caso expresión booleana₁ hacer sent₁

...

caso expresión booleana_n hacer sent_n

en cualquier otro caso sent_{n+1} {opcional}

fopción

(c.2) sentencia alternativa compuesta

repetir expresión natural veces sent frepedir

(d.1) sentencia iterativa simple

mientras expresión booleana hacer sent fmientras

repetir sent hasta que expresión booleana

(d.2) sentencias iterativas gobernadas por una expresión

para todo variable dentro de tipo hacer sent fpara todo

para todo variable desde cota inf hasta cota sup hacer sent fpara todo

para todo variable desde cota sup bajando hasta cota inf hacer sent fpara todo

(d.3) sentencias iterativas gobernadas por una variable

Fig. 2.3: forma general de las sentencias de Merlí.

2.1.3 Funciones y acciones

Son los dos mecanismos de encapsulamiento de las instrucciones que codifican las operaciones de un tipo. Las funciones son la correspondencia directa del concepto de operación de un TAD dentro del lenguaje: a partir de unos parámetros de entrada se devuelve un valor de salida y los parámetros, que tienen un nombre que los identifica y se declaran de un tipo dado, mantienen el mismo valor después de la ejecución de la función, aunque se modifiquen. En caso de que se devuelva más de un valor, se pueden encerrar entre '<' y '>' los nombres de los tipos de los valores que se devuelvan, e indicar así que en realidad se está devolviendo una tupla de valores. La función sólo puede devolver su valor una sola vez, al final del cuerpo. La recursividad está permitida e incluso recomendada como herramienta de diseño e implementación de algoritmos.

Aparte de los parámetros, dentro de la función pueden declararse *variables locales*, cuya existencia está limitada al ámbito de la función. Notemos que en los programas Merlí no existe el concepto de *variable global*, ya que una función sólo tiene acceso a sus parámetros y puede usar variables locales para cálculos intermedios. Este resultado se deriva del hecho de que no pueden imbricarse funciones ni declararse objetos compartidos entre diferentes módulos². También es interesante notar que no hay un programa principal y unas funciones subordinadas, sino que una aplicación consiste en un conjunto de funciones de las cuales el usuario escogerá una, que podrá invocar las otras en tiempo de ejecución; el esquema resultante es altamente simétrico (no da preferencia a ningún componente) y favorece el desarrollo independiente y la integración posterior de diferentes partes de la aplicación.

La existencia de acciones dentro del lenguaje proviene de la necesidad de encapsular operaciones auxiliares que modifican diversos objetos; la definición de estas operaciones como funciones daría como resultado cabeceras e invocaciones engorrosas. Las acciones también presentan el mecanismo de variables locales. La incorporación de acciones en el lenguaje obliga a definir diversos tipos de parámetros: los parámetros sólo de entrada (precedidos por la palabra clave "ent") en la cabecera), que se comportan como los de las funciones; los parámetros sólo de salida (precedidos por "sal"), correspondientes a objetos que no tienen un valor significativo en la entrada y almacenan un resultado en la salida; y los parámetros de entrada y de salida (precedidos por "ent/sal"), que tienen un valor significativo en la entrada que puede cambiar en la salida³. En este texto no nos preocuparemos por los posibles problemas de eficiencia en el uso de funciones en vez de acciones (v. sección 2.3), sino que usaremos siempre la notación que clarifique el código al máximo; se supone que el programador podrá adaptar sin problemas la notación funcional a las características de su instalación concreta.

² En realidad, el mecanismo de memoria dinámica del lenguaje (v. apartado 3.3.3) rompe esta regla general, porque actúa como una variable global implícita.

³ Existen otras clasificaciones de tipos de parámetros, destacando la que distingue entre *parámetros por valor*, aproximadamente equivalentes a los parámetros sólo de entrada, y *parámetros por variable* (también *por referencia*) que se corresponden a grandes rasgos con los otros dos tipos.

En la fig. 2.4 aparecen tres implementaciones posibles de una operación para el cálculo del máximo común divisor de dos números, las dos primeras como funciones, en versiones recursiva e iterativa, y la tercera como una acción que devuelve el resultado en el primero de sus parámetros. También se muestra una función que, dado un vector de naturales y un elemento, devuelve una tupla de dos valores: un booleano que indica si el elemento está dentro del vector o no y, en caso de que esté, el índice de la primera aparición.

```

función MCD (a, b son nat) devuelve nat es
  var res es nat fvar
    opción
      caso a = b hacer res := a
      caso a > b hacer res := MCD(a - b, b)
      caso a < b hacer res := MCD(a, b - a)
    fopción
  devuelve res

```

(a.1) *función recursiva para el cálculo del máximo común divisor de dos naturales*

```

función MCD (a, b son nat) devuelve nat es
  mientras a ≠ b hacer
    si a > b entonces a := a - b si no b := b - a fsi
  fmientras
  devuelve a

```

(a.2) *función no recursiva para el cálculo del máximo común divisor de dos naturales*

```

acción MCD (ent / sal a es nat; ent b es nat) es
  mientras a ≠ b hacer
    si a > b entonces a := a - b si no b := b - a fsi
  fmientras
  facción

```

(a.3) *acción no recursiva para el cálculo del máximo común divisor de dos naturales*

```

función está? (A es vector [de 1 a máx] de nat; x es nat) devuelve <bool, nat> es
  var i es nat; encontrado es bool fvar
    i := 1; encontrado := falso
    mientras (i ≤ máx) ∧ ¬encontrado hacer
      si A[i] = x entonces encontrado := cierto si no i := i + 1 fsi
    fmientras
  devuelve <encontrado, i>

```

(b) *función para la búsqueda de un elemento dentro de un vector*

Fig. 2.4: ejemplos de funciones y acciones.

Por último, introducimos los conceptos de *precondición* y de *postcondición* de una operación. La precondición, abreviadamente P , es un predicado lógico que debe satisfacerse al comenzar la ejecución de una operación; la postcondición, abreviadamente Q , es otro predicado lógico que debe satisfacerse al acabar la ejecución de una operación siempre que se cumpliera previamente la precondición correspondiente. Ambos predicados dependen exclusivamente de los parámetros de la operación. La invocación de una función o acción sin que se cumpla su precondición es incorrecta y da como resultado un valor aleatorio. Así como la especificación algebraica es un paso previo a la implementación de un tipo de datos, el establecimiento de la precondición y la postcondición (llamada *especificación pre-post*) ha de preceder, sin duda, a la codificación de cualquier operación.

Debe distinguirse el uso de las especificaciones algebraicas de los TAD y de las especificaciones pre-post de las operaciones. Por un lado, la especificación algebraica da una visión global del tipo de datos y define el comportamiento de las operaciones para todos los valores; por el otro, la especificación pre-post describe individualmente el comportamiento de cada operación en términos de la representación escogida para el tipo, y es el punto de partida de una verificación posterior del código de la operación.

En este texto se escriben sólo las precondiciones y postcondiciones de las funciones y acciones que no implementan operaciones visibles de los TAD (es decir, funciones auxiliares y algoritmos que usan los TAD), que aparecen juntamente con el código mismo, a menudo en estilo informal para mayor claridad en su lectura y comprensión, o bien simplemente como comentario si se considera que la misión de la operación es suficientemente clara. En lo que respecta a las funciones de los TAD, se formula el modelo matemático correspondiente al tipo previamente a su especificación algebraica y se describe, aunque sólo sea informalmente, el comportamiento de las operaciones en términos de este modelo, lo que puede considerarse como una especificación pre-post de alto nivel. Para profundizar en el tema de las especificaciones pre-post y su uso dentro de la verificación de programas, se puede consultar [Ba93] y [Peñ98].

2.1.4 Ejemplo: una implementación para los conjuntos

Como aplicación de los diversos mecanismos introducidos en esta sección, se quiere implementar el TAD de los conjuntos con operación de pertenencia y un número máximo de elementos, para los cuales se ha presentado una especificación en la fig. 1.32. Las representaciones eficientes de los conjuntos serán estudiadas a lo largo del texto como aplicación de diferentes estructuras de datos que en él se presentan; de momento, no nos preocupa que la implementación escogida sea eficiente, sino que simplemente se quiere ilustrar la aplicación de los constructores del lenguaje, por lo que la estrategia consiste en usar un vector dentro del cual se almacenan los elementos en posiciones consecutivas, y se añade un entero que apunta a la posición que se ocupará cuando se inserte un nuevo

elemento. A un objeto (entero, natural, puntero -v. apartado 3.3.3-, etc.) que apunta a otro se lo denomina *apuntador* (también *cursor*, si el objeto apuntado es una posición de vector), y se empleará bajo diversas variantes en la práctica totalidad de las estructuras de datos estudiadas en el libro; en este caso, lo llamamos *apuntador de sitio libre*, por razones obvias.

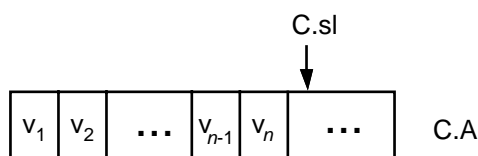


Fig. 2.5: representación del conjunto $C = \{v_1, v_2, \dots, v_n\}$.

El universo parametrizado de la fig. 2.6 sigue esta estrategia. En primer lugar se establece cuál es la especificación que se está implementando. A continuación se escriben los universos de especificación que definen símbolos usados en la implementación. La representación del tipo no presenta ningún problema. Por último, se codifican las operaciones; notemos el uso de una operación auxiliar, *índice*, que se introduce para simplificar la escritura del universo y que se especifica según se ha explicado anteriormente. Destacamos que los errores que se establecen en la especificación han de tratarse en la implementación, ya sea incluyendo su negación en la precondition o bien detectándolos en el código de la función, como ocurre en la operación de añadir. En el segundo caso no se detalla el comportamiento del programa y se deja indicado por la palabra "error", porque su tratamiento depende del lenguaje de programación en que finalmente se implementará el programa, que puede ofrecer o no un mecanismo de excepciones.

2.2 Corrección de una implementación

Los elementos que ofrece el lenguaje para escribir universos de implementación permiten obtener programas ejecutables de la manera tradicional (mediante un compilador que genere código o bien a través de un intérprete); además, el programador puede verificar si el código de cada operación individual cumple su especificación pre-post. Ahora bien, es necesario definir algún método para comprobar si la implementación de un TAD es correcta respecto a su especificación ecuacional formulada previamente y éste es el objetivo de la sección⁴. Así, al igual que la programación de un módulo que use un TAD es independiente de la implementación del mismo, también su verificación podrá realizarse sin necesidad de recurrir a la representación del tipo, confiriendo un alto nivel de abstracción a las demostraciones.

⁴ Una estrategia diferente consiste en derivar programas a partir de la especificación; si la derivación se hace aplicando técnicas formales, el resultado es correcto y no es preciso verificar el código. Esta técnica no se introduce en el presente texto; puede consultarse, por ejemplo, en [Bal93].

```

universo CJT_∈_ACOTADO_POR_VECT (ELEM_ =, VAL_NAT) es
  implementa CJT_∈_ACOTADO (ELEM_ =, VAL_NAT)
  usa NAT, BOOL
  tipo cjt es
    tupla
      A es vector [de 0 a val-1] de elem
      sl es nat
    ftupla
  ftipo
  función Ø devuelve cjt es
  var C es cjt fvar
    C.sl := 0
  devuelve C
  función _∪_{ } (C es cjt; v es elem) devuelve cjt es
    si índice(C, v) = C.sl entonces {se evita la inserción de repetidos}
      si C.sl = val entonces error {conjunto lleno}
      si no C.A[C.sl] := v; C.sl := C.sl + 1
    fsi
  fsi
  devuelve C
  función _∈_ (v es elem; C es cjt) devuelve bool es
  devuelve índice(C, v) < C.sl
  función lleno? (C es cjt) devuelve bool es
  devuelve C.sl = val
  {Función auxiliar índice(C, v) → k: devuelve la posición k que ocupa v en
  C.A; si v no aparece entre las posiciones de la 0 a la C.sl-1, devuelve C.sl.
  
$$\mathcal{P} \equiv (C.sl \leq val) \wedge (\forall i, j: 0 \leq i, j \leq C.sl-1: i \neq j \Rightarrow C.A[i] \neq C.A[j])$$

  
$$\mathcal{Q} \equiv (está?(C, v) \Rightarrow C.A[k] = v) \wedge (\neg está?(C, v) \Rightarrow k = C.sl)$$

  siendo está? el predicado:  $está?(C, v) \equiv \exists i: 0 \leq i \leq C.sl-1: C.A[i] = v$  }
  función privada índice (C es cjt; v es elem) devuelve nat es
  var k es nat; encontrado es bool fvar
    k := 0; encontrado := falso
    mientras (k < C.sl) ∧ ¬encontrado hacer
      {  $I \equiv \forall j: 0 \leq j \leq k-1: C.A[j] \neq v \wedge (encontrado \Rightarrow C.A[k] = v)$ .  $\mathcal{F} \equiv C.sl - k$  }
      si C.A[k] = v entonces encontrado := cierto si no k := k + 1 fsi
    fmientras
  devuelve k
funiverso

```

Fig. 2.6: implementación para los conjuntos con pertenencia.

El punto clave consiste en considerar las implementaciones de los tipos abstractos también como álgebras, y entonces se podrá definir una implementación como la representación de los valores y operaciones del TAD en términos de los valores y las operaciones de otros TAD más simples. Para ello, se determina la *función de abstracción* de una implementación (ing., *abstraction function*, también conocida como *función de representación*), que transforma un valor v de la implementación del TAD en la clase correspondiente del álgebra cociente de términos; el estudio de esta función permite demostrar la corrección de la implementación.

La función de abstracción fue definida por primera vez por C.A.R. Hoare en "Proofs of Correctness of Data Representation", *Acta Informatica*, 1(1), 1972. Presenta una serie de propiedades que son fundamentales para entender su significado:

- Es parcial. Puede haber valores de la implementación que no correspondan a ningún valor del TAD. La condición I_t que ha de cumplir una representación para denotar un valor válido del tipo t se llama *invariante de la representación* (ing., *representation invariant*) y la han de respetar todas las operaciones del tipo.
- Es exhaustiva. Todas las clases de equivalencia del álgebra cociente de términos (que, recordemos, denotan los diferentes valores del TAD) se pueden representar con la implementación escogida.
- No es forzosamente inyectiva. Valores diferentes de la implementación pueden denotar el mismo valor del TAD. Consecuentemente, la función de abstracción induce una relación de equivalencia \mathcal{R}_t que llamamos *relación de igualdad* (*representation equivalence* en el artículo de Hoare) que agrupa en una clase los valores de la implementación que se corresponden con el mismo valor del TAD t .
- Es un homomorfismo respecto a las operaciones de la signatura del TAD. Así, para toda operación $op \in OP_{t_1 \dots t_n \rightarrow t}$ se cumple: $abs_t(op(v_1, \dots, v_n)) = op_t(abs_{t_1}(v_1), \dots, abs_{t_n}(v_n))$, donde abs_t es la función de abstracción del tipo t , op_t la interpretación de op dentro del álgebra del tipo, y la función de abstracción que se aplica sobre cada v_i es la correspondiente a su tipo.

En la fig. 2.7 se muestra la función de abstracción abs_{cjt} para el tipo de los conjuntos, $abs_{cjt}: \mathcal{A}_{cjt} \rightarrow T_{cjt}$, siendo \mathcal{A}_{cjt} el álgebra de la implementación de los conjuntos y T_{cjt} el álgebra cociente de términos (en realidad, la función devuelve un término que identifica la clase resultado); se puede observar que realmente se cumplen las propiedades enumeradas. Así, pueden establecerse el invariante de la representación I_{cjt} y la relación de igualdad \mathcal{R}_{cjt} de la representación de los conjuntos: I_{cjt} acota el número de posiciones ocupadas del vector y prohíbe la aparición de elementos repetidos, mientras que \mathcal{R}_{cjt} define la igualdad de dos estructuras si los sitios libres valen igual y hay los mismos elementos en las posiciones ocupadas de los vectores:

$$I_{cjt}(C \text{ es } cjt) \equiv C.sl \leq val \wedge \forall i, j: 0 \leq i, j \leq C.sl-1: i \neq j \Rightarrow C.A[i] \neq C.A[j]$$

$$\mathcal{R}_{cjt}(C_1, C_2 \text{ son } cjt) \equiv C_1.sl = C_2.sl \wedge (\forall i: 0 \leq i \leq C_1.sl-1: \exists j: 0 \leq j \leq C_2.sl-1: C_1.A[i] = C_2.A[j])$$

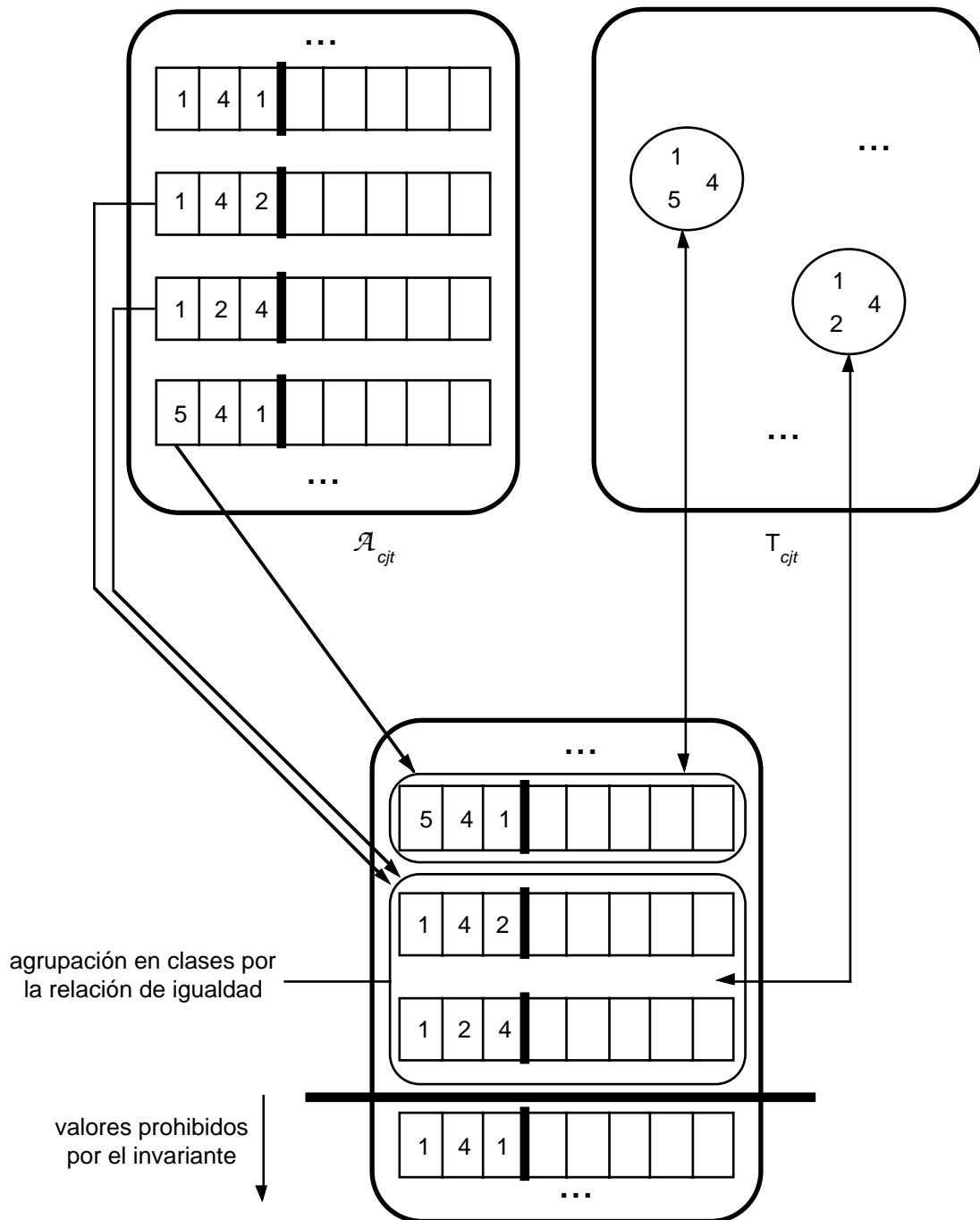


Fig. 2.7: función de abstracción aplicada sobre algunos valores de la representación de los conjuntos (las clases del álgebra cociente se han dibujado directamente como conjuntos).

Notemos que, al establecer la relación de igualdad sobre la representación, se trabaja directamente sobre el dominio de la función de abstracción, es decir, sólo se deben estudiar aquellos valores de la implementación que cumplan el invariante de la representación. Por este motivo en \mathcal{R}_{cjt} no se tiene en cuenta la posibilidad de que haya elementos repetidos porque es una situación prohibida por I_{cjt} . Destaquemos también que la relación exacta entre estos dos predicados y la función de abstracción es (en el ejemplo de los conjuntos):

$$I_{cjt}(\langle A, sl \rangle) \equiv \langle A, sl \rangle \in \text{dom}(\text{abs}_{cjt}), \text{ y}$$

$$\mathcal{R}_{cjt}(\langle A_1, sl_1 \rangle, \langle A_2, sl_2 \rangle) \equiv \text{abs}_{cjt}(\langle A_1, sl_1 \rangle) = \text{abs}_{cjt}(\langle A_2, sl_2 \rangle).$$

Es decir, la relación de igualdad sobre la representación es equivalente a la deducción ecuacional en el marco de la semántica inicial.

A continuación, ha de establecerse claramente cuál es el método de verificación de una implementación con respecto a su especificación, y es aquí donde aparecen varias alternativas propuestas por diversos autores, entre las que destacamos tres que estudiamos a continuación. Los formalismos que presentamos también son válidos si la especificación define más de un género, en cuyo caso cada género del universo de implementación dispondrá de sus propios predicados y, eventualmente, algunas demostraciones implicarán invariantes y relaciones de igualdad de diversos géneros.

Cronológicamente, el primer enfoque propuesto se basa en el uso de las pre y postcondiciones de las operaciones implementadas, y por ello se denomina comúnmente método *axiomático*. Este enfoque, formulado por el mismo C.A.R. Hoare en el artículo ya mencionado, requiere la escritura explícita de la función de abstracción, además de los predicados \mathcal{R}_t e I_t ; a continuación, puede establecerse de manera casi automática la especificación pre-post de las operaciones mediante las siguientes reglas:

- Toda operación tiene como precondition el invariante de la representación, así como la negación de todas aquellas condiciones de error que no se detecten en el cuerpo de la propia función; si la operación es una constante, la precondition es *cierto*.
- Las postcondiciones consisten en igualar la función de abstracción aplicada sobre la representación resultante con la aplicación de la operación sobre los parámetros de entrada, amén del propio invariante de la representación.

Así, por ejemplo, la operación de añadido de un elemento queda con la especificación:

$$\{I_{cjt}(s_1)\} \quad s_2 := s_1 \cup \{v\} \quad \{\text{abs}_{cjt}(s_2) = \text{abs}_{cjt}(s_1) \cup \{v\} \wedge I_{cjt}(s_2)\}^6$$

donde abs_{cjt} se puede escribir recursivamente como:

$$\text{abs}_{cjt}(\langle A, 0 \rangle) = \emptyset$$

$$\text{abs}_{cjt}(\langle A, x+1 \rangle) = \text{abs}_{cjt}(\langle A, x \rangle) \cup \{A[x]\}$$

⁶ Notemos que el símbolo \cup está sobrecargado; por ello, algunos autores distinguen notacionalmente las referencias a la operación del TAD de las referencias a su implementación.

Una vez se dispone de la especificación pre-post, basta con verificar una a una todas las operaciones del tipo para concluir la corrección de la implementación. En dicha verificación, puede ser necesario utilizar la relación de igualdad. Como ejemplo ilustrativo, en el apartado 3.1 se presenta la verificación de la operación de añadir un elemento a una pila.

El segundo método, propuesto por J.V. Guttag, E. Horowitz y D.R. Muser en "Abstract Data Types and Software Validation", *Communications ACM*, 21(12), 1978, es un enfoque básicamente algebraico. Una vez escritos los predicados I_t y \mathcal{R}_t , basta con restringir los valores del álgebra de la implementación mediante I_t , agrupar los valores resultantes en clases de equivalencia según \mathcal{R}_t y, finalmente, demostrar la validez del modelo en la implementación. Es decir, es preciso efectuar cuatro pasos:

- Reescribir la implementación del tipo en forma de ecuaciones.
- Demostrar que \mathcal{R}_t es efectivamente una relación de equivalencia y que se mantiene bajo transformaciones idénticas que involucren las operaciones de la signatura.
- Demostrar que I_t es efectivamente un invariante de la representación respetado por todas las operaciones constructoras del tipo.
- Demostrar que las ecuaciones de la especificación se satisfacen en la implementación.

Estudiemos el ejemplo de los conjuntos. Como resultado del primer paso, habrá un conjunto de ecuaciones por cada operación visible y, en ellas, toda variable de tipo *cjt* se habrá sustituido por la aplicación de la función de abstracción sobre la representación del tipo. Es necesario controlar que el invariante de la representación proteja de alguna manera las ecuaciones (ya sea explícitamente como premisa en una ecuación condicional, ya sea implícitamente, por construcción). En la fig. 2.8 se muestra el universo resultante, donde se aprecia el uso de operaciones sobre vectores, *as* y *cons*, cuyo significado es asignar un valor a una posición y consultarlo, y que se suponen especificadas en el universo *VECTOR*; además, se instancian los pares para poder representar los conjuntos mediante el vector y el apuntador. Notemos que las condiciones en el código se han traducido trivialmente a ecuaciones condicionales, mientras que los bucles se han transformado en ecuaciones recursivas. El universo resultante es utilizado en las tres fases siguientes de la verificación.

Para demostrar que la relación de igualdad se mantiene, basta con estudiar la igualdad de todas las parejas de términos posibles cuya operación más externa sea un símbolo de operación de la signatura, aplicadas sobre dos representaciones iguales según \mathcal{R}_{cjt} . Así, siendo $\text{abs}_{cjt}(\langle A_1, sl_1 \rangle) = \text{abs}_{cjt}(\langle A_2, sl_2 \rangle)$ por la relación de igualdad, debe cumplirse que $\text{abs}_{cjt}(\langle A_1, sl_1 \rangle) \cup \{v\} = \text{abs}_{cjt}(\langle A_2, sl_2 \rangle) \cup \{v\}$, $v \in \text{abs}_{cjt}(\langle A_1, sl_1 \rangle) = v \in \text{abs}_{cjt}(\langle A_2, sl_2 \rangle)$ y $\text{lento}(\text{abs}_{cjt}(\langle A_1, sl_1 \rangle)) = \text{lento}(\text{abs}_{cjt}(\langle A_2, sl_2 \rangle))$. Por lo que respecta a la conservación del invariante, se debe demostrar que, para toda operación constructora *f*, el resultado protege el invariante, suponiendo que los parámetros de tipo *cjt* lo cumplen inicialmente. En el caso de que *f* no tenga parámetros de tipo *cjt*, basta con comprobar que la estructura se crea cumpliendo el invariante. En el último paso, se comprueba la validez de las ecuaciones del

tipo implementado una por una, previa sustitución de las operaciones que en ellas aparecen por su implementación, usando \mathcal{R}_{cjt} , I_{cjt} las ecuaciones de la implementación y las de los tipos que aparecen en la implementación (vectores, tuplas, y similares, incluyendo posibles TAD definidos por el usuario); así, por ejemplo, debe demostrarse que $s \cup \{v\} \cup \{v\} = s \cup \{v\}$, utilizando, entre otras, las ecuaciones de los vectores tales como $cons(as(A, k, v), k) = v$.

```

universo CJT_∈_ACOTADO_POR_VECT (ELEM_∈, VAL_NAT)
  implementa CJT_∈_ACOTADO (ELEM_∈, VAL_NAT)
  usa NAT, BOOL
  instancia VECTOR(ÍNDICE es ELEM_∈, VALOR es ELEM) donde
    ÍNDICE.elem es nat, ÍNDICE.∈ es NAT.∈, VALOR.elem es ELEM.elem
  instancia PAR (A, B son ELEM) donde A.elem es vector, B.elem es nat
  tipo cjt es par
  error ∀A∈vector; ∀sl∈nat: [NAT.ig(sl, val) ∧ ¬v∈ abs_cjt(<A, sl>)] ⇒ abs_cjt(<A, sl>) ∪ {v}
  ecns ∀A∈vector; ∀sl∈nat; ∀v∈elem
    Ø = abs_cjt(<VECTOR.crea, 0>)
    [v∈ abs_cjt(<A, sl>)] ⇒ abs_cjt(<A, sl>) ∪ {v} = abs_cjt(<A, sl>)
    [¬ v∈ abs_cjt(<A, sl>)] ⇒ abs_cjt(<A, sl>) ∪ {v} = abs_cjt(<VECTOR.as(A, sl, v), sl+1>)
    v∈ abs_cjt(<A, 0>) = falso
    [sl ≠ 0] ⇒ v∈ abs_cjt(<A, sl>) = (v = VECTOR.cons(A, sl-1)) ∨ (v∈ abs_cjt(<A, sl-1>))
    lleno?(abs_cjt(<A, sl>)) = NAT.ig(sl, val)
  funiverso

```

Fig. 2.8: implementación por ecuaciones para los conjuntos usando vectores.

El tercer método que se presenta fue propuesto por H. Ehrig, H.-J. Kreowski, B. Mahr y P. Padawitz en "Algebraic Implementation of Abstract Data Types", *Theoretical Computer Science*, 20, 1982, y se basa en el uso de la especificación algebraica del tipo que se utiliza para implementar. Concretamente, siendo t_{spec} el tipo que se ha de implementar definido en el universo $SPEC = (\{t_{spec}\}, OP_{spec}, E_{spec})$, y siendo t_{impl} el tipo implementador definido en el universo $IMPL = (\{t_{impl}\}, OP_{impl}, E_{impl})$, se trata de establecer un isomorfismo entre el álgebra cociente de términos T_{SPEC} y otro objeto algebraico construido a partir de la implementación de t_{spec} por t_{impl} . Sea $ABS = (S_{abs}, OP_{abs}, E_{abs})$ el universo que define esta nueva álgebra; su construcción sigue las fases siguientes:

- Síntesis: inicialmente, se define $ABS = (\{t_{spec}, t_{impl}\}, OP_{spec} \cup OP_{impl}, E_{impl})$. A continuación se efectúa la implementación de las operaciones de OP_{spec} en términos de las de OP_{impl} . Al escribirlas, se necesitan, además, una o más operaciones para convertir un objeto de género t_{impl} a género t_{spec} .

- Restricción: en este paso se olvidan todos aquellos valores de la implementación que no son alcanzables y, además, se esconden los géneros y las operaciones de OP_{impl} para que el usuario de la implementación no pueda utilizarlas.
- Identificación: finalmente, se incorporan las ecuaciones de la especificación a E_{abs} , para asegurar que se igualen todas aquellas representaciones realmente equivalentes.

(Notemos la similitud de los pasos de restricción y de identificación con el invariante de la representación y la relación de equivalencia de los enfoques anteriores.) Finalmente, se considera que la implementación es correcta si el álgebra cociente de términos T_{ABS} de este universo es isomorfa a T_{SPEC} .

Si aplicamos este método al ejemplo de los conjuntos, se podría pensar en tomar como punto de partida la implementación por ecuaciones del universo de la fig. 2.8. Ahora bien, hay diversas situaciones anómalas; por ejemplo, dados dos elementos v_1 y v_2 , las tuplas resultantes de añadir primero v_1 y después v_2 y viceversa son diferentes y, al particionar el álgebra de la implementación, van a parar a clases diferentes e impiden establecer un isomorfismo con el álgebra cociente de la especificación de los conjuntos. Estas situaciones eran irrelevantes en el método de J.V. Guttag *et al.*, donde no se necesitaba definir ningún isomorfismo, sino que simplemente se comprobaba que el comportamiento determinado por la implementación "simulaba" correctamente el comportamiento establecido en la especificación. Estos y otros inconvenientes se solucionan en la fig. 2.9, que sí sigue el método y que se explica a continuación.

En el paso de síntesis se define el nuevo género *cjt* y una operación, *abs*, que convierte un valor de la implementación en otro de la especificación y que es la constructora generadora (una especie de función de abstracción que se limita a "copiar" el valor de la implementación en la especificación, y que es necesaria para no tener problemas con los tipos de los parámetros y los resultados de las operaciones). A continuación, se implementan por ecuaciones las operaciones de los conjuntos en términos de las tuplas. En la fase de restricción, se prohíben repeticiones en los vectores, se limita el valor máximo permitido del sitio libre (en realidad, la última condición no es estrictamente necesaria tal como se ha sintetizado previamente el tipo) y se olvidan todos aquellos símbolos invisibles para el usuario del TAD. Por último, se aplican las ecuaciones de los conjuntos para asegurar la equivalencia de los valores. En la fig. 2.10 se muestra todo el proceso aplicado sobre diversas representaciones que denotan el mismo valor del tipo.

Una vez presentados estos tres métodos, cabe preguntarse en qué contexto puede ser más adecuado uno u otro. En el marco de desarrollo de aplicaciones modulares, una implementación puede constar de diversas etapas; por ejemplo, un conjunto se podría representar mediante una secuencia y, a su vez, una secuencia se podría representar mediante el esquema de vector y apuntador que acabamos de ver. Esta situación se habría dado en caso de haber existido la implementación para las secuencias y de que se hubiera

considerado que su uso facilitaba la implementación de los conjuntos. La corrección de esta implementación se puede verificar usando el enfoque de H. Ehrig *et al.* para la primera etapa y el enfoque de J.V. Guttag *et al.* o de C.A.R. Hoare para la segunda, dado que la implementación de los conjuntos se basa en la especificación algebraica que definen las secuencias, mientras que la implementación de las secuencias se hace directamente usando los constructores del lenguaje.

universo CJT_∈_ACOTADO_POR_VECT (ELEM_∈, VAL_NAT) es
usa NAT, BOOL
instancia VECTOR(ÍNDICE es ELEM_∈, VALOR es ELEM) donde
 ÍNDICE.elem es nat, ÍNDICE.∈ es NAT.∈, VALOR.elem es ELEM.elem
instancia PAR(A, B son ELEM) donde A.elem es vector, B.elem es nat
 {Paso 1: síntesis}
tipo cjt
ops abs: par → cjt
 ∅: → cjt
 ∪: cjt ELEM.elem → cjt
 ∈: ELEM.elem cjt → bool
 lleno?: cjt → bool
errores [lleno?(abs(<A, sl>)) ∧ ¬v ∈ abs(<A, sl>)] ⇒ abs(<A, sl>) ∪ {v}
ecns ∅ = abs(<VECTOR.crea, 0>)
 [v ∈ abs(<A, sl>)] ⇒ abs(<A, sl>) ∪ {v} = abs(<A, sl>)
 [¬v ∈ abs(<A, sl>)] ⇒ abs(<A, sl>) ∪ {v} = abs(<VECTOR.as(A, sl, v), sl+1>)
 v ∈ abs(<A, NAT.0>) = falso
 [sl ≠ 0] ⇒ v ∈ abs(<A, sl>) = (v = VECTOR.cons(A, sl-1)) ∨ (v ∈ abs(<A, sl-1>))
 lleno?(abs(<A, sl>)) = NAT.ig(sl, val)
 {Paso 2: restricción}
errores as(as(A, i, v), i, v); [sl > val] ⇒ <A, sl>
esconde VECTOR.vector, VECTOR.crea, VECTOR.as, VECTOR.cons
 PAR.par, PAR._.c1, PAR._.c2
 {Paso 3: identificación}
errores [lleno?(C) ∧ ¬v ∈ C] ⇒ C ∪ {v}
ecns C ∪ {v} ∪ {v} = C ∪ {v}
 C ∪ {v₁} ∪ {v₂} = C ∪ {v₂} ∪ {v₁}
 ...etc. {v. fig. 1.32}
funiverso

Fig. 2.9: aplicación de la técnica de tres pasos: síntesis, restricción e identificación.

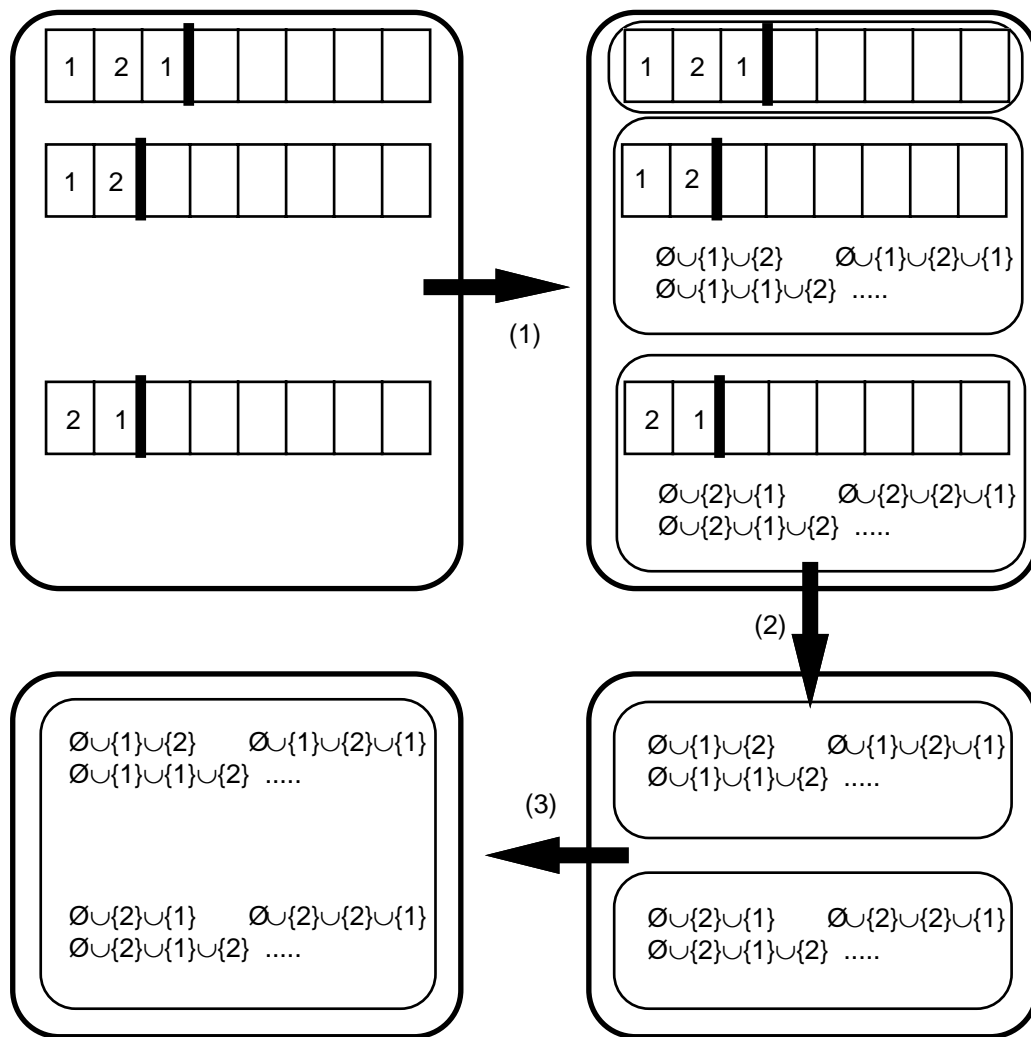


Fig. 2.10: implementación de los conjuntos: (1) síntesis; (2) restricción; (3) identificación.

A partir de la introducción dada aquí, queda claro que las demostraciones de corrección de las implementaciones son ciertamente complicadas sobrepasando los objetivos de este texto, y por ello no se adjuntan. Ahora bien, para mayor legibilidad incluiremos siempre los invariantes de la representación en forma de predicados booleanos que acompañarán a las representaciones del tipo; su escritura complementa las explicaciones que aparecen.

2.3 Estudio de la eficiencia de las implementaciones

Como ya se ha dicho, un tipo de datos descrito mediante una especificación algebraica se puede implementar de varias maneras, que diferirán en la representación del tipo y/o en los algoritmos y detalles de codificación de algunas de sus funciones. La razón de que existan múltiples implementaciones de un único TAD es que, generalmente, diversas aplicaciones exigirán diferentes requisitos de eficiencia y, por ello, no se puede afirmar que una implementación concreta es "la mejor implementación posible" de un TAD (a pesar de que muchas veces es posible rechazar completamente una estrategia concreta de implementación, porque no es la mejor en ningún contexto). En el ejemplo de los conjuntos, una aplicación con pocas inserciones y muchas consultas reclamará la optimización de la operación de pertenencia por encima de la operación de inserción, pero hay diversos algoritmos en este libro que exigen que ambas operaciones sean igualmente rápidas, y entonces es necesario aplicar estrategias que consuman espacio adicional. Idealmente, el conjunto de todas las implementaciones existentes de un tipo de datos ha de ser lo suficientemente versátil como para ofrecer la máxima eficiencia al integrarlo dentro de cualquier entorno⁷.

Se podría argumentar que el estudio de la eficiencia de los programas resulta irrelevante desde el momento en que la potencia y la memoria de los computadores crece y crece sin fin. Ahora bien, este crecimiento, en realidad, incita a la resolución de problemas cada vez más complejos que exigen administrar cuidadosamente los recursos del computador y, por ello, el estudio de la eficiencia sigue vigente. Además, existe una clase de problemas (que aquí llamaremos "de orden de magnitud exponencial", v. apartado 2.3.2) cuya complejidad intrínseca no parece que pueda ser solventada por las mejoras del *hardware*.

Para estudiar la integración correcta de una implementación dentro de una aplicación primero es necesario determinar los criterios que definen su eficiencia y después formular una estrategia de medida de los recursos consumidos; de esta manera, será posible clasificar los algoritmos y, también, compararlos cuando son funcionalmente equivalentes. Por lo que respecta al primer punto, los dos factores más obvios ya se han citado más arriba y son los que usaremos a lo largo del texto: el tiempo de ejecución de las diversas funciones del tipo y el espacio necesario para representar los valores. Normalmente son criterios confrontados (siempre habrá que sacrificar la velocidad de una o más operaciones del tipo y/o espacio para favorecer otras o bien ahorrar espacio) y por ello su estudio tendrá que ser cuidadoso. Hay otros factores que pueden llegar a ser igualmente importantes, sobre todo en un entorno industrial de desarrollo de programas a gran escala, pero que raramente citaremos en este texto a causa de la dificultad para medirlos. Destacan: el tiempo de desarrollo de las aplicaciones; el dinero invertido en escribirlas; el posible *hardware* adicional que exijan; y los

⁷ Ahora bien, no es necesario construir *a priori* todas las implementaciones que se crean oportunas para un tipo, sino que generalmente se escribirán a medida que se precisen (a no ser que se quiera organizar una biblioteca de módulos reusables de interés general).

llamados *factores de calidad del software*, entre los que podemos citar: la portabilidad, la facilidad de mantenimiento, la fiabilidad, etc.

Centrémonos en la elección de las estrategias de medida de la eficiencia. Se podría considerar la posibilidad de medir el tiempo de ejecución del programa en segundos (o fracciones de segundo) y el espacio que ocupa en *bytes* o similares, pero estas opciones ocasionan una serie de problemas que las hacen claramente impracticables:

- Son poco precisas. Un programa que ocupa 1 Mbyte y tarda 10 segundos en ejecutarse, ¿es eficiente o no? Es más, comparado con otro programa que resuelva el mismo enunciado, que ocupe 0.9 Mbytes y tarde 9 segundos en ejecutarse, ¿es significativamente mejor el primero que el segundo o la diferencia es desdeñable y fácilmente subsanable, o incluso imputable a factores externos al programa?
- Son *a posteriori*. Hasta que no se dispone de código ejecutable no se puede estudiar la eficiencia.
- Dependen de parámetros diversos:
 - ◊ De bajo nivel: del *hardware* subyacente, del sistema operativo de la máquina, del compilador del lenguaje, etc. Esta dependencia es nefasta, porque dificulta la extrapolación de los resultados obtenidos a otros entornos de ejecución.
 - ◊ De los datos de entrada: el comportamiento del programa puede depender y, generalmente, dependerá del volumen de datos para procesar y posiblemente de su configuración.

En consecuencia, la medida de la eficiencia de los programas en general y de las estructuras de datos en particular sigue un enfoque diferente, que consiste en centrarse precisamente en el estudio de los datos de entrada del programa. Para conseguir que este método sea realmente previo al desarrollo total del código, se caracterizan las operaciones del TAD mediante el algoritmo subyacente y, a partir de éste, se deduce una función que proporciona la eficiencia tanto en tiempo como en espacio auxiliar, parametrizada por el volumen y/o configuración de los datos que procesa; además, se caracteriza el espacio de la representación del tipo mediante la estructura de datos empleada y se deduce una función similar. Para independizar el método de los parámetros de bajo nivel citados antes, la eficiencia se medirá con las denominadas notaciones asintóticas que se introducen más adelante.

Es necesario precisar el significado que puede tener el parámetro del cálculo de la eficiencia. Básicamente, distinguimos dos casos:

- Representa el volumen de los datos. Por ejemplo, la mayoría de algoritmos sobre vectores y, por extensión, la mayoría de algoritmos que manipulan las estructuras de datos que veremos en este libro, tienen un comportamiento que depende del número de posiciones del vector, o elementos de la estructura. Es el caso de un recorrido o una búsqueda en un vector, o de los algoritmos de ordenación, etc.

- Representa el valor de los datos. Es un caso habitual en los algoritmos que trabajan sobre números enteros: el valor de estos números determinan la eficiencia del algoritmo. Por ejemplo, algoritmos para el cálculo del factorial o del máximo común divisor⁸.

En cualquiera de los dos casos, aparece un problema. Una vez fijados los parámetros de la eficiencia, el comportamiento del programa puede variar substancialmente según el dominio concreto que se maneje. Por ejemplo, la búsqueda de un elemento dentro de un vector de n posiciones exige un número de comparaciones que oscila entre 1 y n , según donde se encuentre el elemento (si está); o bien el cálculo del máximo común divisor utilizando el algoritmo de Euclides requiere un número de iteraciones que será función de la relación entre los dos enteros implicados. Por este motivo, parecería adecuado formular un caso medio a partir de la frecuencia de aparición esperada de los datos. Ahora bien, normalmente es imposible caracterizar la distribución de probabilidad de los datos de entrada, ya sea porque se desconoce, o bien porque se conoce, pero su análisis se vuelve matemáticamente impracticable. Debido a esto y a que, habitualmente, no se busca tanto una eficiencia determinada como una cota inferior que asegure que bajo ningún concepto el comportamiento del programa será inaceptable, se opta tradicionalmente por estudiar el caso peor en la distribución de los datos de entrada. El *caso peor* del tiempo de ejecución de la función F respecto su parámetro de eficiencia n se define como:

$$T_F(n) \equiv \max s : s \text{ es entrada de } F \wedge s \mathcal{R} n : t_F(s),$$

siendo $t_F(s)$ el tiempo de ejecución de la función F con la entrada s , y siendo \mathcal{R} la relación entre la entrada y su parámetro de eficiencia; por ejemplo, en el caso que represente el volumen de los datos, se tiene que $s \mathcal{R} n$ es equivalente a $\|s\| = n$. La definición puede adaptarse para el cálculo del espacio en el caso peor.

Para formular la eficiencia de un algoritmo, una primera opción consiste en contar el número de instrucciones de diverso tipo que intervienen en él; es el enfoque seguido por ejemplo en el libro clásico de D.E. Knuth [Knu68, Knu73], que cuenta el número de instrucciones en un ordenador idealizado. Consideremos por ejemplo la función *está?* que aparece en la fig. 2.4, (b). Observamos que en su interior aparecen diversas comparaciones, operadores lógicos, asignaciones e incrementos, en el contexto de instrucciones de composición secuencial, alternativas e iterativas. Si denotamos por T_{comp} el tiempo de efectuar una comparación entre dos números, por T_{log} el tiempo de aplicar un operador lógico, por T_{asig} el tiempo de efectuar una asignación de tipo básico y por T_{incr} el tiempo de incrementar el valor de una variable numérica, puede calcularse el tiempo de ejecución del algoritmo en el caso peor⁹ respecto el parámetro de eficiencia *máx*, $T_{\text{está?}}(\textit{máx})$, como:

⁸ Este segundo caso normalmente puede reducirse al primero, pero la expresión de eficiencia resultante no es en general tan ilustrativa.

⁹ El caso peor se produce cuando el elemento no está en el vector y se realizan *máx* vueltas de bucle. En este caso, se ejecuta siempre el "si no" de la sentencia condicional.

$$\begin{aligned}
T_{\text{está?}}(\text{máx}) &= 2 * T_{\text{asig}} && \text{(asignaciones iniciales)} \\
&+ (\text{máx}+1) * (T_{\text{comp}} + 2 * T_{\text{log}}) && \text{(número máximo de comparaciones)} \\
&+ \text{máx} * (T_{\text{comp}} + T_{\text{asig}} + T_{\text{incr}}) && \text{(número máximo de condicionales)} \\
&= (2 * \text{máx}+1) * T_{\text{comp}} + 2 * (\text{máx}+1) * T_{\text{log}} + (\text{máx}+2) * T_{\text{asig}} + \text{máx} * T_{\text{incr}}
\end{aligned}$$

Puede observarse que el resultado, incluso para una función pequeña como ésta, es ciertamente complejo de analizar, y sigue siendo dependiente de factores de bajo nivel, concretamente la relación entre T_{comp} , T_{log} , T_{asig} y T_{incr} . Y eso que hemos efectuado algunas simplificaciones que tal vez deberían justificarse adecuadamente; por ejemplo, no hemos considerado el tiempo del paso de parámetros, ni la indexación del vector, etc. Por ello, buscamos otros enfoques que nos permitan deducir resultados más fáciles de interpretar e igual de significativos.

2.3.1 Notaciones asintóticas

Las *notaciones asintóticas* (ing., *asymptotic notation*) son las estrategias de medida de la eficiencia seguidas en este texto. A partir de una función que tiene como dominio y codominio los naturales positivos, $f: \mathcal{N}^+ \rightarrow \mathcal{N}^+$, función que caracteriza la eficiencia espacial de la representación de un tipo o bien la eficiencia temporal de una operación en función del volumen o el valor de los datos utilizados, una notación asintótica define un conjunto de funciones que acotan de alguna manera el crecimiento de f . El calificativo "asintótica" significa que la eficiencia se estudia para volúmenes de datos grandes por ser éste el caso en que la ineficiencia de un programa es realmente crítica. El estudio del caso asintótico permite desdeñar los factores de bajo nivel citados al principio de esta sección; la independencia del entorno de ejecución se traduce en un estudio, no tanto de los valores de f , como de su ritmo de crecimiento, y da como resultado unas definiciones de las notaciones asintóticas que desdeñan, por un lado, los valores más bajos del dominio por poco significativos y, por el otro, las constantes multiplicativas y aditivas que pueden efectivamente considerarse irrelevantes a medida que el volumen de datos aumenta.

Hay varias notaciones asintóticas, explicadas en detalle en diferentes libros y artículos. El artículo pionero en la aplicación de estas notaciones en el campo de la programación fue escrito por D.E. Knuth en el año 1976 ("Big Omicron and Big Omega and Big Theta", *SIGACT News*, 8(2)), que presenta, además, una descripción histórica (ampliada por P.M.B. Vitányi y L.G.L.T. Meertens en "Big Omega versus the Wild Functions", *SIGACT News*, 16(4), 1985). G. Brassard mejoró la propuesta de Knuth definiendo las notaciones como conjuntos de funciones ("Crusade for a Better Notation", *SIGACT News*, 17(1), 1985), que es el enfoque que se sigue actualmente; en [BrB97], el mismo Brassard, junto con P. Bratley, profundiza en el estudio de las notaciones asintóticas.

En este texto introducimos las tres notaciones asintóticas más habituales. Las dos primeras definen cotas superiores y inferiores de los programas: la notación *O grande* (ing., *big Oh* o *big Omicron*), o simplemente *O*, para la búsqueda de cotas superiores y la notación *omega grande* (ing., *big Omega*), o simplemente Ω , para la búsqueda de cotas inferiores. Concretamente, dada una función $f: \mathcal{N}^+ \rightarrow \mathcal{N}^+$, la *O de f*, denotada por $O(f)$, es el conjunto de funciones que crecen como máximo con la misma rapidez que f y la *omega de f*, denotada por $\Omega(f)$, es el conjunto de funciones que crecen con mayor o igual rapidez que f . Dicho en otras palabras, f es una cota superior de todas las funciones que hay en $O(f)$ y una cota inferior de todas las funciones que hay en $\Omega(f)$.

$$O(f) \equiv \{ g: g \in \mathcal{N}^+ \rightarrow \mathcal{N}^+ : (\exists c_0, n_0: c_0 \in \mathcal{R}^+ \wedge n_0 \in \mathcal{N}^+ : (\forall n: n \geq n_0: g(n) \leq c_0 f(n))) \}$$

$$\Omega(f) \equiv \{ g: g \in \mathcal{N}^+ \rightarrow \mathcal{N}^+ : (\exists c_0, n_0: c_0 \in \mathcal{R}^+ \wedge n_0 \in \mathcal{N}^+ : (\forall n: n \geq n_0: g(n) \geq c_0 f(n))) \}^{10}$$

La constante n_0 identifica el punto a partir del cual f es cota de g , mientras que c_0 es la constante multiplicativa (v. fig. 2.11); n_0 permite olvidar los primeros valores de la función por poco significativos, y c_0 desprecia los cambios constantes dentro de un mismo orden debidos a variaciones en los factores de bajo nivel y también a la aparición reiterada en los algoritmos de construcciones que tienen la misma eficiencia.

Por ejemplo, dadas las funciones $f(n) = 4n$, $g(n) = n+5$ y $h(n) = n^2$, se puede comprobar aplicando la definición que f y g están dentro de $O(f)$, $O(g)$ y $O(h)$ y también dentro de $\Omega(f)$ y $\Omega(g)$ (en concreto, $O(f) = O(g)$ y $\Omega(f) = \Omega(g)$, porque ambas funciones presentan el mismo ritmo de crecimiento despreciando la constante multiplicativa), mientras que h no está dentro de $O(f)$ ni de $O(g)$, pero sí dentro de $\Omega(f)$ y $\Omega(g)$. Por ejemplo, para verificar que g está dentro de $O(f)$, podemos dar los valores $c_0 = 1$ y $n_0 = 2$; para verificar que f está dentro de $O(g)$, la asignación de valores puede ser $c_0 = 4$ y $n_0 = 1$.

A partir de la definición, y como ilustra este ejemplo, puede comprobarse que las dos notaciones están fuertemente relacionadas según la fórmula $f \in O(g) \Leftrightarrow g \in \Omega(f)$, cuya demostración queda como ejercicio para el lector.

Por lo que respecta a su utilidad como herramienta de medida de la eficiencia de los programas, las notaciones O y Ω presentan varias propiedades especialmente interesantes que se enumeran a continuación. Así, el cálculo de la eficiencia de un algoritmo se basará en la aplicación de estas propiedades y no directamente en la definición, facilitando el proceso. Para abreviar se dan solamente las propiedades de O ; las de Ω son idénticas.

¹⁰ Hay una definición diferente de la notación Ω (v. ejercicio 2.6) que amplía el conjunto de funciones que pertenecen a $\Omega(f)$. No obstante, las funciones "especiales" que allí aparecen raramente surgen en los análisis de eficiencia de algoritmos y, por este motivo y también por su simplicidad, preferimos la definición aquí dada.

1) La notación O es una relación reflexiva y transitiva, pero no simétrica:

$$\forall f, g, h: \mathcal{N}^+ \rightarrow \mathcal{N}^+: f \in O(f) \wedge \neg (f \in O(g) \Rightarrow g \in O(f)) \wedge (f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h))$$

2) La notación O es resistente a las multiplicaciones por, y adiciones de, constantes:

$$\forall f, g: \mathcal{N}^+ \rightarrow \mathcal{N}^+: \forall c \in \mathcal{R}^+: \{ g \in O(f) \Leftrightarrow c + g \in O(f) \} \wedge \{ g \in O(f) \Leftrightarrow c \cdot g \in O(f) \},$$

siendo $(c+g)(n) \equiv c+g(n)$, y $(c \cdot g)(n) \equiv c \cdot g(n)$

3) Reglas de la notación O respecto a la suma y el producto de funciones:

$$\forall f_1, g_1, f_2, g_2: \mathcal{N}^+ \rightarrow \mathcal{N}^+: g_1 \in O(f_1) \wedge g_2 \in O(f_2):$$

$$3.a) g_1 + g_2 \in O(\max(f_1, f_2)), \text{ siendo } (g_1 + g_2)(n) \equiv g_1(n) + g_2(n), \text{ y}$$

$$\max(f_1, f_2)(n) \equiv \max(f_1(n), f_2(n))$$

$$3.b) g_1 \cdot g_2 \in O(f_1 \cdot f_2), \text{ siendo } (g_1 \cdot g_2)(n) \equiv g_1(n) \cdot g_2(n)$$

Otra manera de enunciar estas reglas es:

$$3.a) O(f_1) + O(f_2) = O(f_1 + f_2) = O(\max(f_1, f_2)), \text{ y}$$

$$3.b) O(f_1) \cdot O(f_2) = O(f_1 \cdot f_2)$$

La caracterización de los conjuntos $O(f_1) + O(f_2)$ y $O(f_1) \cdot O(f_2)$ aparece en [Peñ98], y puede definirse como:

$$O(f) + O(g) \equiv \{h: \mathcal{N}^+ \rightarrow \mathcal{N}^+: \exists a, b, n_0: a \in O(f) \wedge b \in O(g) \wedge n_0 \in \mathcal{N}^+: (\forall n: n \geq n_0: h(n) = a(n) + b(n))\}$$

$$O(f) \cdot O(g) \equiv \{h: \mathcal{N}^+ \rightarrow \mathcal{N}^+: \exists a, b, n_0: a \in O(f) \wedge b \in O(g) \wedge n_0 \in \mathcal{N}^+: (\forall n: n \geq n_0: h(n) = a(n) \cdot b(n))\}$$

Notemos que la definición de las notaciones asintóticas es independiente de si se trata el caso peor o cualquier otro; en el caso peor, no obstante, se produce una asimetría entre ellas: si $f \in O(g)$, entonces g es una cota superior del caso peor y por ello no puede haber ningún valor de la entrada que se comporte peor que g , mientras que si $f \in \Omega(g)$, entonces g es una cota inferior del caso peor, pero puede haber diversos valores de la entrada que se comporten mejor que g , si el algoritmo trabaja en un caso que no sea el peor.

Las dos notaciones hasta ahora presentadas permiten buscar cotas superiores e inferiores de programas; ahora bien, la formulación de una de estas cotas no proporciona suficiente información como para determinar la eficiencia exacta de un programa. Por ello, introducimos una última notación que será la más usada en este texto, llamada *theta grande* (ing., *big Theta*) o, simplemente, Θ . La *theta de f* , notada mediante $\Theta(f)$, es el conjunto de funciones que crecen exactamente al mismo ritmo que f ; dicho de otra manera, $f \in \Theta(g)$ si y sólo si g es a la vez cota inferior y superior de f . Este conjunto se puede formular a partir de $O(f)$ y $\Omega(f)$ como $\Theta(f) = O(f) \cap \Omega(f)$ (v. fig. 2.11) y así la definición queda:

$$\Theta(f) \equiv \{g: g \in \mathcal{N}^+ \rightarrow \mathcal{N}^+: \exists c_0, c_1, n_0: c_0, c_1 \in \mathcal{R}^+ \wedge n_0 \in \mathcal{N}^+: (\forall n: n \geq n_0: c_1 f(n) \geq g(n) \geq c_0 f(n))\}$$

En el ejemplo de la página anterior se cumple que $f \in \Theta(g)$, ya que $f \in O(g)$ y $f \in \Omega(g)$ y, simétricamente, $g \in \Theta(f)$, dado que $g \in O(f)$ y $g \in \Omega(f)$. Precisamente, la simetría de la notación Θ es un rasgo diferencial respecto a O y Ω , porque determina que Θ es una relación de equivalencia; el resto de propiedades antes enunciadas son, no obstante, idénticas. Destaca

también una propiedad referente a límites de funciones: $\lim_{n \rightarrow \infty} f(n)/g(n) \in \mathcal{R} \setminus \{0\} \Rightarrow f \in \Theta(g)$. Por lo que respecta a la nomenclatura, dadas dos funciones $f, g: \mathcal{N}^+ \rightarrow \mathcal{N}^+$ tales que $f \in \Theta(g)$, decimos que f es de orden (de magnitud) g o, simplemente, que f es g (siempre en el contexto del estudio de la eficiencia, claro); también se dice que f es de (o tiene) coste (o complejidad; ing., cost o complexity) g ¹¹.

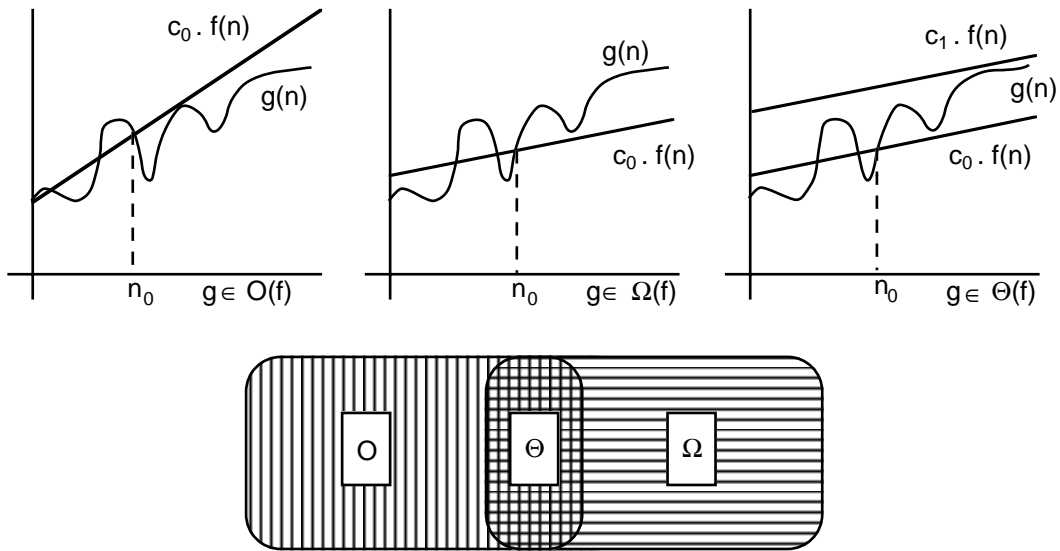


Fig. 2.11: notaciones O , Ω y Θ .

Es importante destacar la mayor precisión de la notación Θ respecto las otras dos. Por ello, preferimos decir que un algoritmo tiene coste $\Theta(n)$ en el caso peor, que decir que un algoritmo tiene coste $O(n)$. La primera afirmación nos proporciona más información, siempre y cuando precisemos que se entiende por "el caso peor" en el análisis en cuestión. Si decimos simplemente que un algoritmo tiene coste $\Theta(n)$, sin especificar en qué caso, se sobreentenderá que dicho coste es común a todos los casos posibles.

Hasta ahora, las notaciones asintóticas se han definido para funciones de una sola variable. No obstante, en el caso general, la eficiencia de un algoritmo puede depender de diferentes magnitudes, cada una de las cuáles representa normalmente el tamaño de un dominio de datos determinado. La definición de O , Ω y Θ en esta nueva situación introduce tantas constantes como variables tenga la función para determinar los valores a partir de los que la cota dada es válida. Por ejemplo, la definición de Θ para una función f de k variables queda:

¹¹ Algunos autores proponen la terminología "orden exacto"; en el caso de que $f \in O(g)$, dicen que f tiene "como máximo" orden g , y si $f \in \Omega(g)$ dicen que f tiene "como mínimo" orden g .

$$\Theta(f)^{12} \equiv \{g: \mathcal{N}^+ \times \dots \times \mathcal{N}^+ \rightarrow \mathcal{N}^+ / \exists c_0, c_1: c_0, c_1 \in \mathcal{R}^+: \exists x_1, \dots, x_k: x_1, \dots, x_k \in \mathcal{N}^+: \\ \forall n_1, \dots, n_k: n_1 \geq x_1 \wedge \dots \wedge n_k \geq x_k: c_1 f(n_1, \dots, n_k) \geq g(n_1, \dots, n_k) \geq c_0 f(n_1, \dots, n_k)\}$$

La existencia de diferentes variables genera órdenes de magnitud incomparables cuando no es posible relacionarlas completamente. Así, $\Theta(a^2.n) \not\leq \Theta(a.n^2)$ y $\Theta(a.n^2) \not\leq \Theta(a^2.n)$. Esta situación dificulta el análisis de la eficiencia y provoca que algunos resultados queden pendientes del contexto de utilización del algoritmo, que será el que determinará todas las relaciones entre los diferentes parámetros de la eficiencia. Concretamente, para asegurar que todos los órdenes de magnitud se puedan comparar, estas relaciones deberían definir un orden total sobre los parámetros que permitiera expresarlos en función de uno solo. Asimismo, la regla de la suma de las notaciones asintóticas se ve afectada porque puede ser imposible determinar el máximo de dos funciones tal como exige su definición.

Por último, es conveniente constatar que el análisis asintótico de la eficiencia de los programas a veces esconde e, incluso, distorsiona otros hechos igualmente importantes, y puede conducir a conclusiones erróneas. Destacamos los puntos siguientes:

- Como ya se ha dicho, el comportamiento asintótico se manifiesta a medida que crece el parámetro de eficiencia, ya sea el volumen de los datos, ya sea el valor de los datos de entrada. Eventualmente, el contexto de utilización de un programa puede asegurar que el volumen de datos es siempre reducido, y entonces el análisis asintótico es engañoso: una función asintóticamente costosa puede llegar a ser más rápida que otra de orden inferior si las diversas constantes aditivas y, sobre todo, multiplicativas, olvidadas en el proceso de cálculo de la eficiencia, son mucho mayores en la segunda que en la primera. Por ejemplo, hay diversos algoritmos de productos de enteros o matrices que son asintóticamente más eficientes que los tradicionales, pero que no se empiezan a comportar mejor hasta dimensiones de los datos demasiado grandes para que sean realmente prácticos en la mayoría de los casos. Esta situación también puede producirse si las constantes multiplicativas desdeñadas durante el cálculo asintótico son muy grandes, aunque en general, las constantes que surgen en el análisis de la eficiencia tienen un valor pequeño.
- En algunos contextos es necesario un análisis más depurado del comportamiento de un algoritmo si realmente se quieren extraer conclusiones pertinentes. Por ejemplo, en la familia de los *algoritmos de ordenación* (ing., *sorting algorithm*) hay varios con el mismo coste asintótico y, por ello, también se consideran las constantes y se estudia en qué situaciones (es decir, para qué configuraciones de los datos de entrada) son más apropiados unos u otros.

¹² En este caso, y por motivos de claridad, puede ser conveniente explicitar las variables en la definición misma, y así escribir $\Theta(f(n_1, \dots, n_k))$.

- Una versión muy eficiente de un programa puede dar como resultado un trozo de código excesivamente complicado y rígido que vulnere los principios de la programación modular enumerados en la sección 1.1, especialmente en lo que se refiere a su mantenimiento y reusabilidad.
- El estudio asintótico muestra claramente que el ahorro de una variable o instrucción que complique el código resultante no aporta ninguna ganancia considerable y reduce la legibilidad; por tanto, dicho ahorro es claramente desaconsejable. Sin embargo, no es justificable la ineficiencia causada por la programación descuidada, por mucho que no afecte al coste asintótico del programa; un ejemplo habitual es el cálculo reiterado de un valor que se podría haber guardado en una variable, si dicho cálculo tiene un coste apreciable.
- Si la vida de un programa se prevé breve priman otros factores, siendo especialmente importante el coste de desarrollo y depuración. En el mismo sentido, si se tiene la certeza de que un trozo de programa se usará sólo en situaciones excepcionales sin que sea necesario que cumpla requisitos de eficiencia exigentes, el estudio de la eficiencia del programa puede prescindir de él y centrarse en las partes realmente determinantes en el tiempo de ejecución. Cualquiera de estas dos suposiciones, evidentemente, ha de estar realmente justificada.
- Otros factores, como la precisión y la estabilidad en los algoritmos de cálculo numérico, o bien el número de procesadores en algoritmos paralelos, puede ser tan importantes (o más) como la eficiencia en el análisis de los programas.

2.3.2 Órdenes de magnitud más habituales

Toda función $f: \mathcal{N}^+ \rightarrow \mathcal{N}^+$ cae dentro de una de las clases de equivalencia determinadas por la relación Θ . Para poder hablar con claridad de la eficiencia de los programas se identifican a continuación las clases más usuales que surgen de su análisis. Todas ellas se caracterizan mediante un representante lo más simple posible, se les da un nombre para referirnos a ellas en el resto del libro y se enumeran algunas funciones que son congruentes.

- $\Theta(1)$: *constante*. Incluye todas aquellas funciones independientes del volumen de los datos que se comportan siempre de la misma manera; por ejemplo, secuencias de instrucciones sin iteraciones ni llamadas a funciones o acciones, y tuplas de campos no estructurados. Dentro de estas clases se encuentran las funciones 4, $8+\log 2$, etc.
- $\Theta(\log n)$: *logarítmico*¹³. Aparecerá en algoritmos que descarten muchos valores en un único paso (generalmente, la mitad) durante un proceso de búsqueda; las estructuras arborescentes del capítulo 4 son un ejemplo paradigmático. Se incluyen funciones como $\log n+k$, $\log n+k \log n$, etc., siendo k una constante no negativa cualquiera.

¹³ No se especifica la base, ya que un cambio de base en los logaritmos se concreta en una multiplicación por una constante (v. ejercicio 2.5).

- $\Theta(n^k)$, para k constante: *polinómico*. Serán habituales los casos $k = 1, 2$ y 3 llamados, respectivamente, *lineal*, *cuadrático* y *cúbico*. Notemos que el caso $k = 0$ es el coste constante. El análisis de una función o estructura de datos en la que haya vectores y la aparición de bucles comporta normalmente un factor polinómico dentro del coste. Son funciones congruentes $n^k + \log n$, $n^k + 7$, $n^k + n^{k-1}$, etc.; consultar también el ejercicio 2.5.
- $\Theta(n^k \log n)$, para k constante: *casi-polinómico*. Serán habituales los casos $k = 1$ y $k = 2$ llamados, respectivamente, *casi-lineal* y *casi-cuadrático*. Generalmente aparecen en bucles tales que cada paso comporta un coste logarítmico o casi-lineal, respectivamente. Son funciones en esta clase $3n^k \log n$, $n^k \log n + n^{k-1}$, y $n^k \log' n$, para r constante. Destacamos que un algoritmo de coste $\Theta(n^k \log n)$ es (bastante) más eficiente que otro de coste $\Theta(n^{k+1})$ y (un poco menos) menos que otro de coste $\Theta(n^k)$.
- $\Theta(k^n)$, para k constante: *exponencial*; es el caso peor de todos, asociado generalmente a problemas que se han de resolver mediante el ensayo reiterado de soluciones.

En la fig. 2.12 se muestra la evaluación de estas funciones para algunos valores representativos de n . En los órdenes donde interviene el logaritmo, el caso $n = 1$ se toma igual a 1 porque el tiempo de ejecución no puede considerarse 0; para fijar unos valores concretos tomamos logaritmos en base 2 (que es el caso habitual). Notemos la similitud del comportamiento de los casos constante y logarítmico, lineal y casi-lineal, y cuadrático y casi-cuadrático, que en memoria interna se pueden considerar prácticamente idénticos, sobre todo teniendo en cuenta que en el cálculo de los costes se desprecian constantes y factores aditivos más pequeños, que pueden llegar a hacer un algoritmo asintóticamente logarítmico mejor que otro constante, como ya se ha dicho en el punto anterior.

n	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2 \log n)$	$\Theta(n^3)$	$\Theta(2^n)$
1	1	1	1	1	1	1	1	2
10	1	2.3	10	23	100	230	1000	1024
100	1	4.6	100	460	10000	46000	10^6	1.26×10^{30}
1000	1	6.9	1000	6900	10^6	6.9×10^6	10^9	$\rightarrow \infty$
10000	1	9.2	10000	92000	10^8	9.2×10^8	10^{12}	$\rightarrow \infty$
100000	1	11.5	10^5	11.5×10^5	10^{10}	11.5×10^{10}	10^{15}	$\rightarrow \infty$

Fig. 2.12: algunos valores de los costes asintóticos más frecuentes.

En caso de que la eficiencia dependa de más de un parámetro, será necesario explicitar éstos al catalogar la función. Frecuentemente, para evitar ambigüedades, en vez de usar los nombres que se acaban de dar, leeremos directamente la función representante de la clase; así, un algoritmo que tenga un coste $2a^2.3 \log n$ lo describiremos como "a cuadrado por el logaritmo de n ".

2.3.3 Análisis asintótico de la eficiencia temporal

Una vez formuladas las diferentes notaciones asintóticas de interés e identificadas sus propiedades más relevantes, es indispensable determinar cómo se puede deducir la eficiencia de una función o representación a partir del algoritmo o la estructura de datos correspondiente. Estudiamos en este apartado la eficiencia temporal y en el siguiente la eficiencia espacial.

El cálculo de la eficiencia temporal de un programa se puede realizar aplicando las reglas que se enumeran a continuación, donde se muestra el tipo de construcción del lenguaje y también las construcciones Merlí concretas para así poder analizar más fácilmente los programas que aparecen en este texto completamente codificados. Destacamos que la mayoría de las reglas dan como resultado sumatorios que, aplicando las propiedades de las notaciones asintóticas, son equivalentes al orden de magnitud del sumando más costoso. Denotaremos por $T(S)$ la eficiencia temporal del trozo S de código; como ya hemos visto, cuando el trozo sea una función, acción u operador f , denotaremos su eficiencia por T_f .

- Referencia a valores y operadores predefinidos. Se toma como coste constante cualquier referencia a un objeto así como la aplicación de operaciones aritméticas, relacionales y lógicas sobre los tipos predefinidos. También se consideran constantes las indexaciones de vectores (sin incluir el cálculo del índice mismo, que tendrá su propio coste) y las referencias a tuplas. Los vectores y las tuplas podrán compararse con $=$ y \neq ; en el caso de los vectores, el coste de la comparación es el producto de la dimensión del vector por el coste de comparar un elemento; en el caso de las tuplas, el resultado es igual al máximo del coste de las comparaciones de los campos individuales.
- Evaluación de una expresión. Suma de los costes de las subexpresiones que la componen y de los operadores que las combinan:

$$T(E_1 \text{ op}_1 E_2 \text{ op}_2 \dots E_n \text{ op}_n E_{n+1}) \equiv [\sum i: 1 \leq i \leq n+1: T(E_i)] + [\sum i: 1 \leq i \leq n: T(\text{op}_i)]$$

Aplicando la propiedad 3.a de las notaciones asintóticas, este coste es equivalente al máximo de los costes individuales $T(\text{op}_i)$ y $T(E_i)$. Los costes $T(\text{op}_i)$ y $T(E_i)$ vienen dados por la regla anterior y, en el caso de que aparezcan invocaciones a funciones, por la regla siguiente.

- Invocación a una función. Suma del tiempo empleado en evaluar y, posiblemente, duplicar los diversos parámetros reales más el tiempo empleado en ejecutar la función; si alguno de los parámetros es una expresión, se debe aplicar la regla anterior.

$$T(f(E_1, \dots, E_n)) \equiv [\sum i: 1 \leq i \leq n: T(E_i) + T(\text{duplicación } E_i)] + T_f$$

Una vez más, la propiedad 3.a nos permitirá simplificar el resultado.

Aunque los parámetros de la función sean implícitamente parámetros sólo de entrada, se considera que su paso sólo exige hacer una copia cuando la función los modifica; es decir, las categorías de los parámetros se consideran exclusivamente como información relativa a su uso, al contrario que los lenguajes imperativos tradicionales, que las tratan también como información orientada a la generación de código. Como resultado, el tiempo de paso de parámetros será constante en muchos casos. No consideramos el posible tiempo de duplicar el resultado, ya que supondremos que éste será disponible implícitamente en el ámbito de la llamada.

De momento, se puede considerar el tiempo de duplicación igual al espacio asintótico que ocupa el tipo correspondiente, calculado según las reglas que se dan en el apartado siguiente. En el apartado 3.3.4 introduciremos en los TAD (por otros motivos) una función de asignación de valores y veremos entonces que la duplicación de los parámetros consistirá en llamar a esta función, que tendrá su propio coste.

Por otro lado, no consideramos la formulación de ecuaciones de recurrencia típicas del caso de las funciones recursivas (v., por ejemplo, [Peñ98]) dado que, en este texto, y considerando la simplicidad de los algoritmos recursivos que aparecen, el cálculo de la eficiencia de este tipo de funciones se hará usando otros razonamientos.

- Composición secuencial de instrucciones. Suma de cada una de las instrucciones que componen la secuencia.

$$T(S_1; \dots; S_n) \cong \sum_{i: 1 \leq i \leq n} T(S_i)$$

que, de nuevo, será igual al máximo de los costes $T(S_i)$.

- Asignación. Suma de los tiempos de evaluación de la parte izquierda y de la parte derecha. Una vez más, no obstante, la asignación puede ser de vectores o tuplas completos, en cuyo caso ni la parte izquierda ni la parte derecha presentarán expresiones. En la asignación de vectores se multiplica el coste de la asignación a un componente por la dimensión, y en la de tuplas, al ser el coste la suma, domina la asignación de campos más costosa.

$$T(E_1 := E_2) \cong T(E_1) + T(E_2), \text{ si el tipo es predefinido o por enumeración}$$

$$T(v_1 := v_2) \cong n \cdot T(v_1[i] := v_2[i]), \text{ para vectores de } n \text{ componentes}$$

$$T(v_1 := v_2) \cong \sum_{i: 1 \leq i \leq n} T(v_1.c_i := v_2.c_i), \text{ para tuplas de campos } c_1, \dots, c_n$$

La expresión $n \cdot \Theta(f)$ es igual a $\Theta(n \cdot f)$, cuya definición aparece en el apartado 2.3.1.

- Retorno de valor. Tiempo de evaluar la expresión: $T(\text{devuelve } E) \cong T(E)$.
- Errores. Supondremos que el tratamiento de errores es de tiempo constante.
- Sentencias alternativas. Suma de las diferentes expresiones condicionales e instrucciones asociadas. A causa del cálculo del caso peor, se examinan todas las ramas posibles y predomina el coste de la condición o la expresión más costosa.

$$T(\text{si } E \text{ entonces } S_1 \text{ sino } S_2 \text{ fsi}) \equiv T(E) + T(S_1) + T(S_2)^{14}$$

$$T(\text{opción caso } E_1 \text{ hacer } S_1 \dots \text{ caso } E_n \text{ hacer } S_n \text{ en cualquier otro caso } S_{n+1} \text{ fopción}) \\ \equiv [\sum i: 1 \leq i \leq n+1: T(S_i)] + [\sum i: 1 \leq i \leq n: T(E_i)]$$

- Sentencias iterativas. Suma de la condición del bucle y de las instrucciones que forman el cuerpo, multiplicadas por el número n de veces que se ejecuta el bucle. Hay diversas modalidades de bucle que sólo evalúan la condición una única vez, al inicio.

$$T(\text{hacer } E \text{ veces } S \text{ fhacer}) \equiv T(E) + (n \cdot T(S))$$

$$T(\text{mientras } E \text{ hacer } S \text{ fmientras}), T(\text{repetir } S \text{ hasta que } E) \equiv n \cdot (T(E) + T(S))$$

$$T(\text{para todo } v \text{ desde } E_1 \text{ hasta } E_2 \text{ hacer } S \text{ fpara todo}) \equiv T(E_1) + T(E_2) + (n \cdot T(S))$$

$$T(\text{para todo } v \text{ dentro de } T \text{ hacer } S \text{ fpara todo}) \equiv n \cdot T(S)$$

En los dos últimos casos, la referencia a v no afecta al coste porque la variable ha de ser de tipo natural, entero o por enumeración y su asignación, por tanto, constante.

Hay que destacar que, a veces, n es una magnitud desconocida y entonces se asume el valor más grande que puede tomar (hipótesis de caso peor), a pesar de que también se puede conjeturar un valor fijo si se pretende comparar diversos algoritmos más que formular una eficiencia concreta. También es frecuente que el coste de ejecución de un bucle no sea siempre el mismo; si se dan los dos casos, se puede multiplicar el número máximo de vueltas por el coste más grande de un bucle y así seguir moviéndonos en el caso peor, aunque el resultado obtenido puede ser incluso peor que el caso peor, debido a la poca precisión del análisis.

Para evitar esta falta de precisión, el análisis de un bucle puede no seguir las reglas de cálculo vistas, sino que se emplea una estrategia diferente que exige un análisis más profundo del significado del código. Así lo haremos en este mismo capítulo al analizar el algoritmo de ordenación por el método de la inserción, y también en el capítulo 6 al estudiar diversos algoritmos sobre grafos. En estos casos la regla se define como:

$$T(\text{mientras } E \text{ hacer } S \text{ fmientras}) \equiv \sum i: 1 \leq i \leq n: (T(E) + T(S) \text{ en la vuelta } i\text{-ésima})$$

y de manera similar en el resto de modalidades de bucles.

- Invocación a una acción. Por los mismos razonamientos que la invocación a funciones:

$$T(P(E_1, \dots, E_n)) \equiv [\sum i: 1 \leq i \leq n: T(E_i) + T(\text{duplicación } E_i)] + T_P$$

Como en las funciones, sólo se duplican los parámetros de entrada que se modifican.

¹⁴ Para ser más exactos, el coste del algoritmo se puede precisar en términos de las notaciones O y Ω , y obtener $O(\max(T(E), T(S_1), T(S_2)))$ como cota superior y $\Omega(\max(T(E), \min(T(S_1), T(S_2))))$ como cota inferior. Si ambos valores son iguales, se puede afirmar efectivamente que Θ tiene el coste dado; si no lo son, realmente se está tomando como coste la cota superior del caso peor, dentro del contexto pesimista en que nos movemos.

2.3.4 Análisis asintótico de la eficiencia espacial

También el espacio que ocupa una estructura de datos se puede calcular automáticamente a partir de unas reglas formuladas sobre los diferentes constructores de tipo del lenguaje, tomando como coste $\Theta(1)$ el espacio que ocupa un objeto de tipo predefinido o por enumeración. Concretamente, si T es un tipo de datos representado en Merlí dentro de un universo de implementación, el espacio $E(T)$ que ocupa un objeto de tipo T se calcula según las reglas:

- Vectores. Producto del espacio que ocupa cada componente por la dimensión n .

$$E(T) \equiv E(\text{vector } [\text{índice}] \text{ de } t) = n.E(t)$$

- Tuplas (tanto variantes como normales). Suma del espacio de los campos.

$$E(T) \equiv E(\text{tupla } c_1 \text{ es } t_1; \dots; c_n \text{ es } t_n \text{ ftupla}) = \sum i: 1 \leq i \leq n: E(t_i)$$

Sin embargo, a menudo, cuando se estudia el espacio de una estructura de datos, no sólo interesa el coste asintótico sino también su coste "real" sin despreciar las constantes multiplicativas ni los factores aditivos de orden inferior. La razón es doble. Por un lado, no será habitual que dos estrategias de representación de un tipo de datos sean tan radicalmente diferentes como para resultar en costes asintóticos desiguales, sino que las diferencias estarán en el número de campos adicionales empleados para su gestión eficiente; estas variaciones no influyen en el coste asintótico, pero pueden ser suficientemente significativas como para considerarlas importantes en la elección de la representación concreta. Por otro lado, al contrario que en el análisis de la eficiencia temporal, las expresiones resultantes de este cálculo dependerán de pocos parámetros de bajo nivel, y por ello son más fáciles de analizar.

Para el cálculo correcto de este coste "real" se debe determinar el espacio que ocupa un objeto de cada uno de los tipos predefinidos, un objeto de tipo por enumeración y un objeto de tipo puntero (los punteros se introducen en el apartado 3.3.3), y aplicar las mismas reglas en los vectores y en las tuplas, poniendo especial atención en el caso de las tuplas variantes: dada una parte variante de n conjuntos de campos $\{c_1\}, \dots, \{c_n\}$, el espacio que ocupa es la suma del campo discriminante más el máximo del espacio de los $\{c_i\}$. El espacio se puede dar en términos absolutos (en número de bytes o palabras) o bien relativos, eligiendo el espacio de alguno de los tipos como unidad y expresando el resto en función suya.

Por último, cabe destacar que el estudio de la eficiencia espacial se aplica, no sólo a las representaciones de estructuras de datos, sino también a las variables auxiliares que declara una función o acción. Es decir, se considera que toda función o acción, aparte de sus parámetros, utiliza un espacio adicional que puede depender de la estrategia concreta y que, eventualmente, puede llegar a cotas lineales o cuadráticas. En este libro, el espacio adicional se destacará cuando sea especialmente voluminoso o cuando sea un criterio de elección u optimización de una operación.

Para ilustrar las reglas definidas en estos dos últimos puntos, se determina a continuación la eficiencia de la implementación de los conjuntos de la fig. 2.6. La eficiencia espacial queda:

- Asintótica: la representación de los conjuntos es una tupla de un vector de elementos *elem* de *val* posiciones y un entero *sl*. Por ello, queda $E(cjt) = val.E(elem) + \Theta(1) = \Theta(val.n)$, siendo $E(elem) \equiv \Theta(n)$ y $\Theta(1)$ el coste del campo *sl*.
- Real: sea *x* el espacio que ocupa un natural y sea *n* el espacio que ocupa un objeto de tipo *elem*, el espacio real resultante es $val.n + x$.

Es decir, el coste espacial depende de lo que ocupe un objeto de tipo *elem*; ésta será la situación habitual de los tipos parametrizados. Por otro lado, notemos que todas las funciones son lo más óptimas posible dado que sólo requieren un espacio adicional constante, incluso la operación auxiliar¹⁵. Por último, destaquemos que la eficiencia espacial de un conjunto es independiente del número de elementos que realmente contiene.

En lo que respecta al coste temporal, empezamos por calcular la eficiencia de *índice*, que sirve de base para el resto. El cuerpo de la función es una secuencia de cuatro instrucciones (dos asignaciones, un bucle y el retorno) y el coste total será su suma. Las asignaciones y la instrucción de retorno tienen un coste constante, porque son valores o referencias a variables de tipos predefinidos. Por lo que se refiere al bucle, el coste se calcula así:

- La evaluación de la condición de entrada queda constante, porque consiste en aplicar una operación booleana (por definición, $\Theta(1)$) sobre dos operandos, siendo cada uno de ellos una operación de coste constante aplicada sobre unos operandos, que son valores o bien referencias a variables de tipos predefinidos.
- El cuerpo consiste en una sentencia condicional y su eficiencia es la suma de tres factores: la evaluación de la condición y dos asignaciones de coste constante igual que las examinadas anteriormente. El coste de la evaluación es igual al coste $T_{=}$ de la operación de igualdad, que es desconocido por depender del parámetro formal.
- El número de iteraciones es desconocido *a priori*. En el caso peor, el elemento que se busca no está dentro de la parte ocupada del vector y entonces el número de iteraciones para un conjunto de *k* elementos es igual a *k*.

Considerando todos estos factores, el coste total del bucle para un conjunto de *k* elementos es $k.(\Theta(1)+T_{=}) = \Theta(k.T_{=})$. El coste total de *índice* es, pues, $\Theta(1)+\Theta(1)+\Theta(k.T_{=})+\Theta(1) = \Theta(k.T_{=})$. El coste del resto de funciones se calcula aplicando razonamientos parecidos y queda $\Theta(k.T_{=})$ igualmente, excepto *crea*, que no invoca *índice* y es constante.

Para cerrar el estudio de la eficiencia asintótica de las implementaciones, consideremos un algoritmo algo más complicado, como es el algoritmo de ordenación de un vector *A*, usando el método de inserción (ing., *insertion sorting*). El método de inserción (v. fig. 2.13) sitúa los elementos en su lugar uno a uno, empezando por la posición 1 hasta el máximo: al inicio del

¹⁵ Por mucho que una función no declare ninguna variable, se considera un coste constante.

paso i -ésimo del bucle principal, las posiciones entre la $A[1]$ y la $A[i-1]$ contienen las mismas posiciones que al empezar, pero ordenadas (como establece el invariante) y entonces el elemento $A[i]$ se intercambia reiteradamente con elementos de este trozo ordenado hasta situarlo en su lugar. Para facilitar la codificación, se supone la existencia de una posición 0 adicional, que se emplea para almacenar el elemento más pequeño posible del tipo (valor que sería un parámetro formal del universo) y que simplifica el algoritmo; asimismo, supondremos que los elementos presentan dos operaciones de comparación, $_<_$ y $_ \leq _$, con el significado habitual.

{Acción *ordena_inserción*: dado un vector A , lo ordena por el método de la inserción.

$\mathcal{P} \equiv A = A_0$ (se fija un valor de inicio para referirse a él en Q)

$Q \equiv \text{elems}(A, 1, \text{máx}) = \text{elems}(A_0, 1, \text{máx}) \wedge \text{ordenado}(A, 1, \text{máx})$, donde

$\text{elems}(A, i, j) \equiv \{A[i], A[i+1], \dots, A[j]\}$, como multiconjunto¹⁶, y

$\text{ordenado}(A, i, j) \equiv \forall k: i \leq k \leq j-1: A[k] \leq A[k+1]$ }

acción *ordena_inserción* (ent/sort A es vector [de 0 a máx] de elem) es

var i, j son nat; temp es elem fvar

$A[0] := \text{el_menor_de_todos}$

para todo i desde 2 hasta máx hacer

$\{ I \equiv \text{elems}(A, 1, i-1) = \text{elems}(A_0, 1, i-1) \wedge \text{ordenado}(A, 1, i-1) \wedge$

$\wedge \forall k: 1 \leq k \leq \text{máx}: A[k] > A[0]$

$\mathcal{F} \equiv \text{máx}-i \}$

$j := i$

mientras $A[j] < A[j-1]$ hacer

$\{ I \equiv 1 \leq j \leq i \wedge \text{ordenado}(A, j, i) \wedge \forall k: 1 \leq k \leq \text{máx}: A[k] > A[0]$

$\mathcal{F} \equiv j \}$

$\text{temp} := A[j]; A[j] := A[j-1]; A[j-1] := \text{temp}; j := j-1$

fmientras

fpara todo

facción

Fig. 2.13: algoritmo de ordenación de un vector por el método de inserción.

Estudiemos el coste temporal en el caso peor del algoritmo (el coste espacial adicional es $\Theta(E(\text{elem}))$), que abreviamos por T_{ord} . El coste es igual a la suma del coste de la primera asignación y del bucle; dado que dentro del bucle aparecen más asignaciones del mismo tipo, se puede decir simplemente que T_{ord} es igual al coste del bucle principal, que se ejecuta $\text{máx}-1$ veces. Dentro de este bucle se encuentran las expresiones que determinan los valores que toma la variable de control, una asignación de naturales (ambas construcciones de coste constante) y un segundo bucle que se ejecuta un número

¹⁶ Un multiconjunto es un conjunto que conserva las repeticiones. Así, los multiconjuntos $\{1\}$ y $\{1,1\}$ son diferentes.

indeterminado de veces, que es función del valor i y de la suerte que tengamos. Precisamente es esta indeterminación la que nos conduce a aplicar la segunda modalidad de regla de cálculo de la eficiencia del bucle: se calcula el número N de veces que se ejecutan las instrucciones del cuerpo del bucle interno a lo largo de todo el algoritmo, se suma el resultado al coste $\Theta(\text{máx})$ de ejecutar la asignación $j := i$ exactamente $\text{máx}-1$ veces y así se obtiene el coste total del bucle principal.

El valor N es la resolución de dos sumatorios imbricados; el más interno surge de la suposición del caso peor que lleva a i iteraciones del bucle interno:

$$N = \sum_{i=2}^{\text{máx}} \sum_{j=i}^1 1 = \sum_{i=2}^{\text{máx}} i = \frac{\text{máx}(\text{máx}+1)}{2} - 1$$

Ahora se puede determinar el coste total del bucle principal que queda parametrizado por los costes de comparar y de asignar elementos, T_c y T_{as} , respectivamente, y da como resultado:

$$\Theta((\text{máx}(\text{máx}+1)/2 - 1) \cdot T_c \cdot T_{as}) = \Theta(\text{máx}^2 \cdot T_c \cdot T_{as}),$$

cuadrático sobre el número de elementos. No es un coste demasiado bueno, dado que hay algoritmos de ordenación de coste casi-lineal (en el capítulo 4 se presenta uno, conocido como *heapsort*).

Precisamente, el desconocimiento del coste T_{as} esconde una posible ineficiencia del algoritmo dado en la fig. 2.13, porque la colocación del elemento $A[i]$ exige tres asignaciones a cada paso del bucle más interno para implementar el intercambio. En lugar de intercambiar físicamente los elementos se puede simplemente guardar $A[i]$ en una variable auxiliar *temp*, hacer sitio desplazando elementos dentro del vector y copiar *temp* directamente en la posición que le corresponda. El resultado es una nueva versión asintóticamente equivalente a la anterior, pero que simplifica las constantes multiplicativas con poco esfuerzo y sin complicar el código, y que por tanto es preferible a la anterior.

2.4 Conflicto entre eficiencia y modularidad

La técnica de diseño modular de programas empleada en este texto presenta diversas ventajas ya comentadas en la sección 1.1, pero puede dar como resultado aplicaciones ineficientes precisamente a causa del principio de transparencia de la implementación, que impide la manipulación de la representación de un tipo desde fuera de sus universos de implementación. En esta sección vamos a enumerar las tres situaciones más comunes en este contexto: falta de funcionalidad en la signatura, necesidad de recorrer los elementos de una estructura y necesidad de acceder a los elementos de una estructura mediante apuntadores externos. Introduciremos los TAD recorribles y los TAD abiertos para tratar adecuadamente los problemas de eficiencia causados por las dos últimas situaciones.

2.4.1 Falta de funcionalidad en la signatura

Consideremos la especificación habitual de los conjuntos y su implementación vista en la sección anterior. Dado que los conjuntos son un TAD que aparece en multitud de aplicaciones, parece una decisión acertada incluir dicho TAD en una hipotética biblioteca de módulos reusables que contenga tipos y algoritmos de interés general. Notemos, no obstante, que el estado actual del tipo incluye muy pocas operaciones, y para que sea realmente útil se deben añadir algunas nuevas; como mínimo, se necesita alguna operación para sacar elementos y seguramente la unión, la intersección y similares. En concreto, supongamos que añadimos una operación *sacar_uno_cualquiera* que, dado un conjunto, selecciona y borra un elemento cualquiera, que se implementa con coste $\Theta(1)$. Ahora, sea U un universo que precisa una nueva operación, *cuántos*, que cuenta el número de elementos que hay dentro de un conjunto y que no aparece en la especificación residente en la biblioteca. Usando los mecanismos de estructuración de especificaciones, se definirá U como un enriquecimiento del universo de especificación de los conjuntos con esta nueva operación (v. fig. 2.14). Su codificación dentro de un universo de implementación es trivial, altamente fiable e independiente de la representación escogida, pero presenta un problema evidente: para un conjunto de n elementos, el coste temporal de la operación es $\Theta(n)$ en todos los casos, a causa de la imposibilidad de manipular la representación del tipo.

```

universo U(ELEM_ =, VAL_ NAT) es
  usa CJT_ ∈ _ACOTADO(ELEM_ =, VAL_ NAT), NAT, BOOL
  ops cuántos: cjt → nat...
  ecns cuántos(∅) = 0...
funiverso

universo IMPL_U(ELEM_ =, VAL_ NAT) es
  implementa U(ELEM_ =, VAL_ NAT)
  usa CJT_ ∈ _ACOTADO(ELEM_ =, VAL_ NAT), NAT, BOOL
  función cuántos (c es conjunto) devuelve nat es
  var cnt es nat; v es elem fvar
    cnt := 0
    mientras ¬vacío?(c) hacer
      <c, v> := sacar_uno_cualquiera(c); cnt := cnt + 1
    fmientras
  devuelve cnt
funiverso

```

Fig. 2.14: especificación (arriba) e implementación (abajo) de un enriquecimiento de los conjuntos.

Una opción para reducir el coste de la operación *cuántos* consiste en redefinir el tipo *cjt* dentro de *IMPL_U* añadiendo explícitamente un contador (v. fig. 2.15). Sin embargo, con este enfoque deben redefinirse las operaciones del tipo para implementarlas sobre el nuevo género, apareciendo nuevos problemas:

- Si bien se mejora la eficiencia de *cuántos*, también puede empeorarse la eficiencia de alguna otra operación. En la fig. 2.15, vemos que al añadir el elemento es necesario comprobar si ya estaba o no. Esta comprobación vuelve a realizarse al invocar la operación de inserción del universo enriquecido. Si bien asintóticamente el coste no varía, el tiempo de añadido de elementos se duplica.
- Si el módulo reside en una biblioteca de módulos reusables de interés general, seguramente ofrecerá bastantes operaciones, y su redefinición es una tarea engorrosa que puede provocar errores.

```

universo IMPL_U(ELEM_ =, VAL_NAT) es
  implementa U(ELEM_ =, VAL_NAT)
  usa CJT_ ∈ _ACOTADO(ELEM_ =, VAL_NAT), NAT, BOOL
  tipo cjt es tupla
    cbase es CJT_ ∈ _ACOTADO.cjt
    cnt es nat
  ftupla
  ftipo
  función Ø devuelve cjt es
  var c es cjt fvar
    c.cbase := CJT_ ∈ _ACOTADO.crea; c.cnt := 0
  devuelve c
  función _ ∪ { } (c es cjt; v es elem) devuelve cjt es
    si ¬CJT_ ∈ _ACOTADO.(v ∈ c.cbase) entonces c.cnt := c.cnt + 1 fsi
    c.cbase := CJT_ ∈ _ACOTADO.(c.cbase ∪ {v})
  devuelve c
  función _ ∈ _ (v es elem; c es conjunto) devuelve bool es
  devuelve CJT_ ∈ _ACOTADO.(v ∈ c.cbase)
  .....
  función cuántos (c es conjunto) devuelve nat es
  devuelve c.cnt
funiverso

```

Fig. 2.15: implementación de un enriquecimiento para los conjuntos con redefinición del tipo.

- El nuevo tipo no es compatible con el anterior. Así, dadas dos variables r y s de tipo cjt la primera implementada con $CJT_∈_ACOTADO_POR_VECT$ y la segunda con $IMPL_U$, la llamada a una operación de unión de conjuntos mediante $union(r, s)$ no puede ejecutarse directamente.

Destacamos que no hay ninguna otra alternativa que permita reducir el coste de la operación *cuántos* (como resultado del bucle que forzosamente dará n vueltas) sin violar el principio de transparencia de la representación. Si se quiere reducir el coste de la operación a orden constante sin sufrir los problemas vistos en la solución anterior, no queda más remedio que introducir la función *cuántos* dentro de la especificación básica de los conjuntos (v. fig. 2.16) y adquirir así el derecho de manipular la representación del tipo.

```

universo CJT_∈_ACOTADO(ELEM_=, VAL_NAT) es
  usa NAT, BOOL
  tipo cjt
  ops  $\emptyset$ ,  $\cup$ {_}, ...
      cuántos: cjt  $\rightarrow$  nat
  ecns ...
      cuántos( $\emptyset$ ) = 0 ...

funiverso

universo CJT_∈_ACOTADO_POR_VECT(ELEM_=, VAL_NAT) es
  implementa CJT_∈_ACOTADO(ELEM_=, VAL_NAT)
  tipo cjt es ... {v. fig. 2.6}
  función cuántos (c es conjunto) devuelve nat es
    devuelve c.sl

funiverso

```

Fig. 2.16: especificación e implementación de la operación para contar los elementos de un conjunto como parte del TAD que los define.

Este enfoque, no obstante, también presenta diversos inconvenientes:

- Se introduce en la definición de un tipo de interés general (que residirá dentro una biblioteca de módulos reusables) una operación que inicialmente no estaba prevista; si esta operación no es utilizada por ninguna otra aplicación, se complica la definición y el uso del tipo sin obtener demasiado provecho a cambio. En el ejemplo que nos ocupa, puede argüirse que una operación para obtener el número de elementos no es tan extraña como para no incluirla en un módulo de interés general, pero el problema puede aparecer de nuevo en otras operaciones realmente particulares.

- Si el universo ya existía previamente, hay que modificarlo y esto es problemático:
 - ◊ Es necesario implementar la nueva operación en todas las implementaciones existentes para el tipo. En cada una de ellas, si el implementador original (persona o equipo de trabajo) está disponible y puede encargarse de la tarea, ha de recordar el funcionamiento para poder programar la nueva operación (¡y puede hacer meses que la escribió!). En caso contrario, todavía peor, pues alguna otra persona ha de entender y modificar la implementación, siendo esta tarea más o menos complicada dependiendo de la calidad del código (sobre todo por lo que respecta a la legibilidad). Incluso es posible que el código fuente no esté disponible y sea imposible modificarlo.
 - ◊ La sustitución de la versión anterior del universo (residente en una biblioteca de módulos) por la nueva puede provocar una catástrofe si se introduce inadvertidamente algún error. Si, por prudencia, se prefiere conservar las dos versiones, se duplicarán módulos en la biblioteca.

Resumiendo, se pueden seguir dos enfoques diferentes al definir un tipo: pensar en los universos de definición como un suministro de las operaciones indispensables para construir otras más complicadas en universos de enriquecimiento, incluso a costa de la eficiencia, o bien incluir en el universo de definición del tipo todas aquellas operaciones que se puedan necesitar en el futuro, buscando mayor eficiencia en la implementación.

2.4.2 Tipos abstractos de datos recorribles

La fig. 2.14 puede servir de ejemplo para ilustrar una situación particular que surge continuamente en los programas que utilizan estructuras de datos: la necesidad de recorrer la estructura para obtener sus elementos y aplicarles algún tratamiento. Esta necesidad aparece en prácticamente todos los TAD que veremos a lo largo del libro, independientemente del modelo matemático subyacente. La solución dada en el apartado anterior no es en general satisfactoria, pues para recorrer el conjunto se ha utilizado una operación que va eliminando sus elementos uno a uno, con los problemas no sólo de integridad (la destrucción del conjunto puede obligar a una duplicación previa) sino también de eficiencia que conlleva esta estrategia (además del coste de esta posible duplicación, la operación de supresión a menudo tiene un coste no constante).

Una solución menos agresiva y, normalmente, tanto o más eficiente nos la proporciona el concepto de *iterador* (ing. *iterator*). Un iterador es un mecanismo de acceso a los elementos de la estructura de datos que define la existencia de un *punto de acceso* que puede moverse secuencialmente según algún criterio determinado (que puede ser completamente aleatorio). Este concepto, ya ideado en los 80 (por ejemplo, el concepto de *iteration abstraction* de CLU [LiG86]) aparece con frecuencia en algunas bibliotecas de módulos

reusables de amplia difusión como LEDA, STL, Booch Components o Java Collection Framework.

Los iteradores pueden integrarse de diferentes maneras en los TAD. Por ejemplo, en algunas propuestas efectuadas dentro del marco de la orientación a objetos, un iterador es un objeto que se asocia a otro, que es el objeto que se quiere examinar. Así, se pueden tener diversos iteradores asociados sobre un mismo objeto y mantener diversos recorridos paralelos. Este esquema puede ser peligroso y, en el marco imperativo, es además más complicado de implantar. Dadas nuestras necesidades más habituales, el enfoque que seguimos consiste en añadir al TAD una operaciones nuevas que posibiliten la obtención de todos los elementos que almacena. Concretamente, dado el TAD T cuyo tipo de interés es s y que almacena elementos de tipo v , introducimos las operaciones siguientes:

- *principio* : $s \rightarrow s$, prepara la estructura para su recorrido, es decir, situa al punto de acceso sobre el primer elemento de la estructura.
- *actual* : $s \rightarrow v$, devuelve el elemento (de género v) actual del recorrido, es decir, aquél sobre el que se encuentra el punto de acceso.
- *avanza* : $s \rightarrow s$, avanza un elemento dentro del recorrido, es decir, mueve el punto de acceso al siguiente elemento.
- *final?* : $s \rightarrow \text{bool}$, indica si se han examinado todos los elementos, es decir, si el punto de acceso ha llegado al final de la estructura.

En la fig. 2.17 se muestra una signatura para los conjuntos con estas operaciones, a los que denominamos conjuntos recorribles. Así, dado un TAD T , definimos su *TAD recorrible* asociado, que denotamos por $T_{\text{recorrible}}$ y que encapsulamos en el universo $T_{\text{RECORRIBLE}}$.

```

universo CJT_∈_ACOTADO_RECORRIBLE (ELEM_∈, VAL_NAT) es
  usa NAT, BOOL
  tipo cjt
  ops ∅: → cjt
    _∪{_: cjt elem → cjt
    _∈_: elem cjt → bool
    principio: cjt → cjt
    actual: cjt → elem
    avanza: cjt → cjt
    final?: cjt → bool
  ecns ...
funiverso

```

Fig. 2.17: signatura para los conjuntos recorribles con operaciones de iterador.

En la fig. 2.18 mostramos una nueva versión del universo *IMPL_U* de la fig. 2.14 que usa los conjuntos recorribles. Notemos que el orden de obtención de los elementos del conjunto es irrelevante; esta situación es muy común. Ahora bien, en el marco de la semántica inicial, una vez más nos vemos abocados a la sobre-especificación, pues se requiere que este orden esté bien definido. Este hecho afecta: a la especificación, porque obliga a fijar una política de recorrido que puede ser artificiosa y que puede derogar alguna propiedad del modelo de partida, típicamente la conmutatividad; a la implementación, porque obtener los elementos en cualquier orden acostumbra a ser rápido, mientras que definir un orden de visita puede impactar en el coste de las actualizaciones de la estructura, que deben asegurar que dicho orden se mantiene. El seguimiento estricto de la semántica inicial en este contexto y, por ende, de los TAD recorribles tal y como se definen aquí, queda a discreción del programador.

```

universo IMPL_U(ELEM_ =, VAL_NAT) es
  implementa U(ELEM_ =, VAL_NAT)
  usa CJT_∈_ACOTADO_RECORRIBLE(ELEM_ =, VAL_NAT), NAT, BOOL
  función cuántos (c es conjunto) devuelve nat es
    var cnt es nat fvar
      cnt := 0; c := principio(c)
      mientras ¬final?(c) hacer
        cnt := cnt + 1; c := avanza(c)
      fmientras
    devuelve cnt
  funiverso

```

Fig. 2.18: implementación de un enriquecimiento para los conjuntos con iteradores.

Si nos decidimos por imponer una estrategia de recorrido, ya sea para no seguir dentro del marco de la semántica inicial, ya sea porque el contexto de uso del iterador así lo exige, obtenemos *TAD recorribles ordenadamente*. El criterio de ordenación obedece a alguno de los dos motivos siguientes:

- A las relaciones estructurales de los elementos. Por ejemplo, podemos querer obtener los elementos en el mismo orden en que se han insertado.
- A los valores mismos de los elementos. Por ejemplo, podemos querer obtener los elementos en orden creciente.

La semántica inicial recogerá el orden de recorrido mediante ecuaciones (en la sección 3.3 puede encontrarse un ejemplo). Además, en el segundo caso, el universo de caracterización de los parámetros puede ser diferente, típicamente exigiendo una relación de orden. En el ejemplo de los conjuntos, para recorrer los elementos en orden creciente, el universo de caracterización debería ser *ELEM_<_* en vez de *ELEM_ =* (v. fig. 2.19)

<u>universo</u> CJT_∈_ACOTADO_RECORRIBLE_ORD (ELEM_<_ =, VAL_NAT) <u>es</u>	
<u>usa</u> NAT, BOOL	
<u>tipo</u> cjt	
<u>ops</u> ∅: → cjt	
<u> </u> ∪ {_}: cjt elem → cjt	<u>universo</u> ELEM_<_ = <u>caracteriza</u>
<u> </u> ∈ _: elem cjt → bool	<u> </u> <u>usa</u> ELEM_ =
<u> </u> principio: cjt → cjt	<u> </u> <u>ops</u>
<u> </u> actual: cjt → elem	<u> </u> _<_: elem elem → bool
<u> </u> avanza: cjt → cjt	<u> </u> <u>ecns</u>
<u> </u> final?: cjt → bool	<u> </u>
<u>funiverso</u>	<u>funiverso</u>

Fig. 2.19: *signatura de los conjuntos recorribles ordenadamente por el valor sus elementos.*

Por el hecho de disponer de ecuaciones diferentes, el modelo del nuevo TAD es diferente del anterior. Por ejemplo, dos conjuntos con los mismos elementos pero en un estado diferente de recorrido deben considerarse diferentes, pues la operación *actual* devolverá valores diferentes en ambos casos. Concretamente, si denotamos por A el modelo del TAD original, por V el dominio de los elementos almacenados, y por $<$ una operación de ordenación de los elementos de A (ora por las relaciones estructurales, ora por los valores de sus elementos), el nuevo modelo puede definirse según dos estrategias distintas:

- El producto cartesiano $A \times A$. Entonces, el valor $\langle a_1, a_2 \rangle$ del TAD se interpreta como: todos los elementos de a_1 han sido accedidos en el recorrido y los elementos de a_2 , no. Los elementos almacenados en $\langle a_1, a_2 \rangle$ son la unión de elementos de a_1 y a_2 (sea cual sea el concepto de unión). Si se avanza el iterador, el nuevo valor del TAD recorrible es $\langle a_1 \cup \{x\}, a_2 - \{x\} \rangle$ tal que $x \in a_2$ cumple $\forall w: w \in a_2: x < w \vee x = w$.
- El producto cartesiano $A \times V$. En este caso, el valor $\langle a, x \rangle$ del TAD indica que x es el elemento actual en el recorrido. En algunos casos, y en particular cuando el modelo admite repeticiones de elementos, puede identificarse de alguna manera la posición que ocupa el elemento en vez del elemento mismo (sea cual sea el concepto de posición). Si se avanza el iterador, el nuevo valor del TAD recorrible es $\langle a, w \rangle$ tal que $w \in a$ cumple $\forall z: z \in a \wedge x < z: w < z \vee w = z$.

Estas signaturas permiten implementar de manera sencilla los dos esquemas de programación más usuales que exigen una obtención consecutiva de elementos: el esquema de *recorrido* y el esquema de *búsqueda* (v. fig. 2.20). En el esquema de recorrido se aplica un tratamiento T sobre todos los elementos de la estructura, mientras que en el esquema de búsqueda se explora la estructura hasta encontrar un elemento que cumpla una propiedad P o bien hasta llegar al final; en caso de encontrarse dicho elemento, el punto de acceso queda situado sobre él. Por lo que respecta a la búsqueda, la situación más habitual consiste en comprobar si un elemento particular está o no dentro de la estructura.

$c := \text{principio}(c)$	$c := \text{principio}(c); \text{está} := \text{falso}$
<u>mientras</u> $\neg \text{final?}(c)$ <u>hacer</u>	<u>mientras</u> $\neg \text{final?}(c) \wedge \neg \text{está}$ <u>hacer</u>
$T(\text{actual}(c))$	<u>si</u> $P(\text{actual}(c))$ <u>entonces</u> $\text{está} := \text{cierto}$
$c := \text{avanza}(c)$	<u>si no</u> $c := \text{avanza}(c)$
<u>fmientras</u>	<u>fsi</u>
	<u>fmientras</u>

Fig. 2.20: esquemas de recorrido (izq.) y búsqueda (der.) sobre TAD recorribles.

Como el esquema de recorrido se usa con mucha frecuencia en los algoritmos de este texto, adoptaremos a partir de ahora una notación abreviada que clarificará su escritura: siendo c un valor de un TAD recorrible de elementos de tipo v y siendo x una variable de tipo v , el bucle:

para todo x dentro de c hacer $\text{tratar } x$ fpara todo

es equivalente a la secuencia de instrucciones:

```

c := principio(c)
mientras  $\neg \text{final?}(c)$  hacer
     $\text{tratar actual}(c); c := \text{avanza}(c)$ 
fmientras
  
```

Debe destacarse que la utilización de los iteradores según cualquiera de estos dos esquemas asegura un comportamiento fiable del TAD recorrible. No pasa lo mismo si se permiten actualizaciones durante el recorrido o la búsqueda. Por ejemplo, consideremos la posibilidad de dar de alta elementos durante el recorrido:

- Si no existe orden de recorrido, debe decidirse si nuevo elemento ha de ser accesible o no durante el recorrido actual.
- Si existe orden de recorrido, el elemento será o no accesible dependiendo de su relación con el resto de elementos. Por ejemplo, si se efectúa un recorrido ordenado de un conjunto de enteros, el elemento actual es el 100 y se insertan los elementos 50 y 200, el primero no será accesible en el recorrido actual y el segundo sí, y este comportamiento no uniforme probablemente se considerará un inconveniente.

Las diversas propuestas existentes de iteradores tampoco adoptan una posición unánime respecto este punto. En este texto, y dado el contexto habitual de uso de los recorridos, contemplamos éstos de manera disociada a las actualizaciones, a no ser que el TAD se diseñe explícitamente con este comportamiento en mente, como es el caso de las listas con punto de interés (v. sección 3.3). Por ello, no nos preocupa especialmente detallar el comportamiento del TAD en este caso y dejamos que sean las ecuaciones ya existentes las que determinen la política seguida, con la seguridad que los algoritmos habituales sobre TAD no van a actualizar las estructuras durante un recorrido.

La fig. 2.21 muestra una implementación posible de los conjuntos recorribles. Se introduce un campo nuevo en la representación del tipo que apunta siempre al elemento actual del recorrido y que se incorpora al invariante. Debe destacarse el hecho que, al añadir por el final, si durante un recorrido se añade un elemento, éste siempre se encontrará.

```

universo CJT_∈_ACOTADO_RECORRIBLE_POR_VECT (ELEM_∈_, VAL_NAT) es
  implementa CJT_∈_ACOTADO_RECORRIBLE (ELEM_∈_, VAL_NAT)
  usa NAT, BOOL
  tipo cjt es tupla
    A es vector [de 0 a val-1] de elem
    act, sl es nat
  ftupla
  ftipo
  invariante (c es cjt):  $0 \leq c.act \leq c.sl \leq val \wedge \forall i, j: 0 \leq i, j \leq c.sl-1: i \neq j \Rightarrow c.A[i] \neq c.A[j]$ 
  función Ø devuelve cjt es
  var c es cjt fvar
    c.sl := 0; c.act := 0
  devuelve c
  funciones _∪_, _∈_ y lleno?: idénticas a la fig. 2.6
  función principio (c es cjt) devuelve cjt es
    c.act := 0
  devuelve c
  función actual (c es cjt) devuelve elem es
  var v es elem fvar
    si c.act = c.sl entonces error {se ha terminado el recorrido}
    si no v := c.A[c.act]
    fsi
  devuelve v
  función avanza (c es cjt) devuelve cjt es
    si c.act = c.sl entonces error {se ha terminado el recorrido}
    si no c.act := c.act + 1
    fsi
  devuelve c
  función final? (c es cjt) devuelve cjt es
  devuelve c.act = c.sl
funiverso

```

Fig. 2.21: implementación para los conjuntos recorribles.

El coste de las operaciones ya existentes no varía, mientras que las operaciones de recorrido son todas ellas $\Theta(1)$; no es usual que se cumplan ambas condiciones a la vez. Como conclusión, el coste del esquema de recorrido sobre esta representación es de $\Theta(n \cdot T_{\text{trat}})$, siendo n el número de elementos del conjunto y siendo T_{trat} el coste en el caso peor del tratamiento individual de cada elemento durante el recorrido.

Comparemos esta implementación con los universos de la fig. 2.22, que implementan los conjuntos recorribles ordenadamente según el valor de sus elementos. Se ofrecen dos alternativas. El primer universo ordena los elementos cuando comienza el recorrido (en la operación *principio*), y las inserciones son por el final, de manera que pueden obtenerse elementos en un orden incorrecto si se realizan inserciones durante el recorrido; podría remediarse el problema determinando al iniciar el recorrido cuál es el último elemento al que debe llegarse. El segundo universo mantiene el conjunto siempre ordenado y, así, si se inserta durante un recorrido, el elemento insertado se encontrará o no dependiendo de su valor respecto al elemento actual; debe cuidarse además de actualizar el apuntador.

Por último, debe destacarse que el uso de TAD recorribles desde los algoritmos debe justificarse adecuadamente porque, en el caso general, el uso de iteradores perjudica la eficiencia del TAD, especialmente en el caso de los recorridos ordenados por contenido de los elementos, donde las actualizaciones deben preservar el orden y pueden exigir hasta costes lineales sobre el número de elementos (v. ejercicio 2.10).

```

universo CJT_∈_ACOTADO_RECORRIBLE_ORD_VECT_1 (ELEM_<_≠, VAL_NAT)
  implementa CJT_∈_ACOTADO_RECORRIBLE_ORD (ELEM_<_≠, VAL_NAT)
  usa NAT, BOOL
  tipo cjt es tupla
    A es vector [de 0 a val-1] de elem
    act, sl es nat
    ftupla
  ftipo
  invariante (c es cjt):  $0 \leq c.act \leq c.sl \leq val \wedge \forall i, j: 0 \leq i, j \leq c.sl-1: i \neq j \Rightarrow c.A[i] \neq c.A[j]$ 
  funciones  $\emptyset, \cup, \_ \in \_, \text{lleno?}, \text{actual}, \text{avanza}$  y final?: idénticas a la fig. 2.21
  función principio (c es cjt) devuelve cjt es
    ... aplicar algún buen algoritmo de ordenación sobre el vector c.A
    c.act := 0
  devuelve c
funiverso

```

Fig. 2.22 (a): implementación para los conjuntos recorribles ordenadamente con vector desordenado.

universo CJT \in _ACOTADO_RECORRIBLE_ORD_VECT_2 (ELEM \leq _, VAL_NAT)
implementa CJT \in _ACOTADO_RECORRIBLE_ORD (ELEM \leq _, VAL_NAT)
usa NAT, BOOL
tipo cjt es tupla
 A es vector [de 0 a val-1] de elem
 act, sl es nat
 ftupla
ftipo
invariante (c es cjt): $0 \leq c.act \leq c.sl \leq val \wedge \forall i, j: 0 \leq i < j \leq c.sl-1: c.A[i] < c.A[j]$
función \emptyset : idéntica a la fig. 2.20
función $\cup \{ _ \}$ (c es cjt; v es elem) devuelve cjt es
var i, k son nat; encontrado, final son bool fvar
 i := 0; encontrado := falso; final := falso
 mientras i < c.sl \wedge \neg final hacer
 si c.A[i] \geq v entonces final := cierto; encontrado := (c.A[i] = v)
 si no i := i + 1
 fsi
 fmientras
 si \neg encontrado entonces {se evita la inserción de repetidos}
 si c.sl = val entonces error {conjunto lleno}
 si no {desplazamiento de los elementos a la derecha del punto de interés
 una posición a la derecha, para hacer sitio}
 para todo k desde c.sl bajando hasta i+1 hacer c.A[k] := c.A[k-1] fpara todo
 c.A[i] := v; c.sl := c.sl+1
 si i \leq c.act llavors c.act := c.act + 1 fsi
 fsi
 fsi
devuelve c
función \in _ : podría usarse búsqueda dicotómica (v. sección 5.2)
función lleno?: idéntica a la fig. 2.6
funciones principio, actual, avanza y final?: idénticas a la fig. 2.21
funiverso

Fig. 2.22 (b): implementación para los conjuntos recorribles ordenadamente con vector ordenado.

2.4.3 Tipos abstractos de datos abiertos

Los iteradores solucionan uno de los problemas más comunes en el marco de la programación con TAD, pero existen otros, y entre ellos destacamos la necesidad de acceder a los elementos de una estructura de datos de manera eficiente. Veamos mediante un ejemplo en qué consiste este problema.

Sea $CJT_ \in_ ACOTADO_ IND$ una nueva versión de $CJT_ \in_ ACOTADO$ en el que los elementos disponen de una clave que los identifica y hay una operación *indexa* tal que, dada la clave, devuelve el elemento asociado en el conjunto (v. fig. 2.23); cambiamos el parámetro formal del universo para dotar al género de los elementos de una operación que proporciona esta clave. Supongamos que disponemos de una instancia de $CJT_ \in_ ACOTADO_ IND$ para almacenar los empleados de una empresa, donde para los empleados se registra como información: el nif (que consideramos una cadena de nueve caracteres), el nombre y apellidos, el domicilio, la formación académica, etc. (v. fig. 2.24, izq.). A continuación, supongamos que se quiere disponer de un conjunto de pólizas correspondiente a una mutua sanitaria que tiene un convenio con la susodicha empresa; en este conjunto los elementos se consideran identificados por número de póliza y, para cada póliza, se registran los datos del empleado que la ha suscrito y cierta información adicional, como la modalidad de contrato, un historial actualizado, etc. (v. fig. 2.24, der.). Para simplificar el estudio, suponemos que ambos conjuntos se instancian del mismo tamaño *max*.

Al implementar estos dos conjuntos, supongamos que utilizamos en ambos casos una implementación por vectores. Analicemos con un poco más de detalle la posibilidades de las que disponemos. Sean $E_{empleado}$ el espacio que ocupan los datos personales de un empleado, E_{mutua} el espacio necesario para registrar los datos de las pólizas en la mutua sin incluir los datos personales del empleado, E_{cadena} el espacio ocupado por una cadena de nueve caracteres (que son los necesarios para representar el nif de un empleado) y E_{entero} el espacio ocupado por un entero. Se cumplen las siguientes relaciones: $E_{empleado} \gg E_{cadena}$, $E_{mutua} \gg E_{cadena}$ y $E_{cadena} > E_{entero}$, incluso normalmente $E_{cadena} > 2 * E_{entero}$.

La primera opción consiste en mantener dos estructuras completamente desligadas, donde los datos personales de los empleados aparecen duplicados (v. fig. 2.25, arriba izq.). Observamos en este caso un desperdicio de espacio considerable, pues el espacio total de ambos conjuntos, E_{total} , es igual a $(máx * E_{empleado}) + (máx * (E_{empleado} + E_{mutua}))$. Además, la redundancia provoca un problema de gestión de la consistencia: aunque las estructuras se hayan definido independientemente, es necesario que los datos personales de los empleados coincidan en ambas, lo que debe ser asegurado por el programa que las usa. Estos inconvenientes apuntan a la necesidad de buscar otras alternativas.

<u>universo</u> CJT_ε_ACOTADO_IND (ELEM_CLAVE, VAL_NAT) <u>es</u> <u>usa</u> NAT, BOOL <u>tipo</u> cjt <u>ops</u> Ø: → cjt <u> </u> ∪ {_}: cjt elem → cjt <u> </u> ∈ _: clave cjt → bool <u> </u> indexa: cjt clave → elem <u>ecns</u> ... <u>funiverso</u>	<u>universo</u> ELEM_CLAVE <u>caracteriza</u> <u>tipo</u> elem, clave <u>ops</u> <u> </u> id: elem → clave <u> </u> =_, ≠_: clave clave → bool <u>ecns</u> ... <u>funiverso</u>
--	---

Fig. 2.23: *signatura para los conjuntos indexables y caracterización de sus elementos.*

<u>universo</u> CJT_EMPLEADOS <u>es</u> <u>usa</u> EMPLEADO, ... <u>instancia</u> CJT_ε_ACOTADO_IND (ELEM_CLAVE, VAL_NAT) <u>donde</u> elem <u>es</u> empleado, clave <u>es</u> cadena id <u>es</u> nif, val <u>es</u> máx, ... <u>renombra</u> cjt <u>por</u> cjt_empleado ... <u>funiverso</u> <u>universo</u> EMPLEADO <u>es</u> <u>tipo</u> empleado <u>ops</u> ... <u> </u> nif: empleado → cadena ... <u>funiverso</u>	<u>universo</u> CJT_POLIZAS <u>es</u> <u>usa</u> POLIZA, ... <u>instancia</u> CJT_ε_ACOTADO_IND (ELEM_CLAVE, VAL_NAT) <u>donde</u> elem <u>es</u> poliza, clave <u>es</u> entero id <u>es</u> npoliza, val <u>es</u> máx, ... <u>renombra</u> cjt <u>por</u> cjt_póliza ... <u>funiverso</u> <u>universo</u> CJT_POLIZAS <u>es</u> <u>tipo</u> poliza <u>ops</u> ... <u> </u> npoliza: poliza → entero ... <u>funiverso</u>
---	--

Fig. 2.24: *dos instancias de los conjuntos indexables.*

La solución más evidente al problema visto consiste en implementar el conjunto de empleados como repositorio total de información, y en el de pólizas guardar la información estrictamente necesaria para acceder al primero, es decir, el nif de los empleados (v. fig. 2.25, arriba derecha) y así el espacio total queda $E_{\text{total}} = (\text{máx} * E_{\text{empleado}}) + (\text{máx} * (E_{\text{cadena}} + E_{\text{mutua}}))$, con un ahorro de espacio igual a $\text{máx} * (E_{\text{empleado}} - E_{\text{cadena}})$. Además, se evita la necesidad de controlar la consistencia.

Sin embargo, todavía existe un problema que tiene a ver con la eficiencia de las operaciones. Supongamos una operación *datos_personales*: *cjt_póliza entero* → *empleado*, que devuelve los datos personales del empleado que ha suscrito una póliza con un número determinado. En la primera implementación, esta operación tenía como coste el de localizar

la póliza en el conjunto de pólizas, que denotamos por $T_{CJT_PÓLIZAS,indexa}$; no se requería acceder al conjunto de empleados pues la información estaba duplicada. En cambio, en esta nueva implementación se requiere primero localizar la póliza con coste $T_{CJT_PÓLIZAS,indexa}$ y luego localizar el empleado dado su nif, con coste $T_{CJT_EMPLEADOS,indexa}$. A lo máximo que podemos aspirar es que el coste asintótico no se incremente, lo que ocurre si en ambos casos la indexación (que consistirá en buscar la clave en el vector) es del mismo coste, por ejemplo $\Theta(máx)$ en la representación vista en la fig. 2.6; eso sí, el tiempo no asintótico se incrementa (en el caso peor, se duplica). Pero el problema puede ser más grave. Imaginemos el caso en que los números de póliza son enteros consecutivos entre 1 y $máx$, de manera que se pueden usar directamente para indexar el vector subyacente. En este caso, se tiene que $T_{CJT_PÓLIZAS,indexa}$ es $\Theta(1)$ mientras que $T_{CJT_EMPLEADOS,indexa}$ no, y ahora sí que se incrementa el coste asintótico en el caso peor¹⁷.

De hecho, la solución más eficiente consiste en representar explícitamente la relación entre las dos estructuras de manera que desde cada póliza del conjunto se identifique la posición que ocupa el empleado correspondiente en el vector que implementa el conjunto de empleados (v. fig. 2.25, abajo). De esta manera, una vez localizada la póliza en tiempo $T_{CJT_PÓLIZAS,indexa}$, el acceso a los datos del empleado se concreta en un acceso al vector de coste $\Theta(1)$, y así el tiempo final es $T_{CJT_PÓLIZAS,indexa}$ como en la primera solución. Y no sólo esto, sino que también se reduce el espacio de la estructura dado que se cambia el nif por un entero (índice del vector), y queda $E_{total} = (máx * E_{empleado}) + (máx * (E_{entero} + E_{mutua}))$. No obstante, esta solución adolece de diversos problemas que estudiamos más adelante detalladamente, todos ellos derivados del hecho de que se permite al conjunto de pólizas conocer la implementación usada para el conjunto de empleados, violando el principio de ocultación de la información enunciado en la sección 1.1.

Esta situación, que en este texto denotamos como el problema del *acoplamiento* de las estructuras de datos, es bien conocido y en cierto modo es el exponente máximo de los conflictos entre modularidad y eficiencia propios de la programación con TAD. Cuando disponemos de dos estructuras de datos que comparten lógicamente un dominio real de elementos, la programación con TAD frecuentemente no permite disponer de versiones altamente eficientes debido a la ocultación de la representación del tipo. Hasta la pasada década, la opción generalmente más seguida consistía en sacrificar la modularidad cuando era necesario y permitir el acceso a la representación del tipo desde fuera del módulo de implementación correspondiente. En los últimos años, las bibliotecas de módulos reusables ya citadas en el apartado anterior han intentado con mayor o menor fortuna racionalizar este acceso eficiente mediante el concepto de *posición*, que en algunos casos incluso coincide con el de iterador. Dado el enfoque de este texto, se requiere que dicho concepto se incorpore a la especificación formal del tipo y, tal como hicimos con los iteradores, definiremos un patrón general de añadido de las posiciones a cualquier TAD.

¹⁷ Este problema se mantiene incluso usando organizaciones más astutas para la representación de conjuntos, como las que se estudian en el capítulo 5.

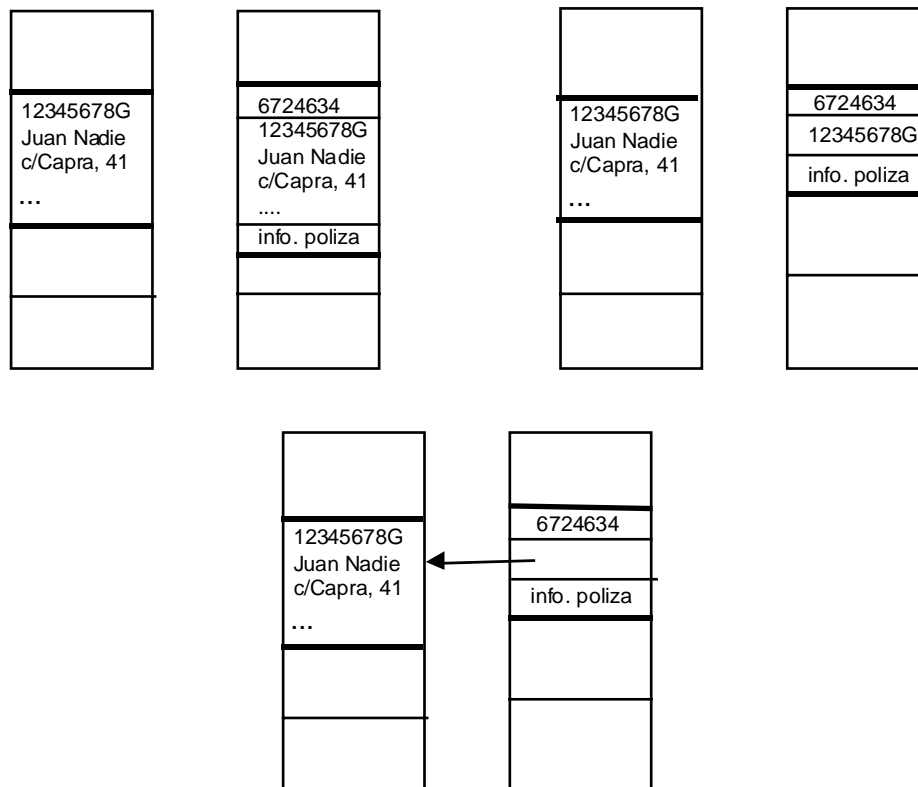


Fig. 2.25: acoplamiento de dos estructuras de datos: con replicación de información (arriba izq.); con replicación de clave (arriba der.); con apuntadores externos (abajo).

(a) TAD parcialmente abiertos

El ejemplo que hemos utilizado para introducir el problema ilustra una situación típica: se requiere acceder a los elementos almacenados en una estructura mediante *apuntadores externos*, es decir, apuntadores a dichos elementos que se almacenan en otras estructuras de datos diferentes (incluso en variables simples), pero no se requiere usar estos apuntadores externos para modificar la estructura. Llamaremos *TAD parcialmente abiertos* a los TAD que presentan esta característica, y *atajos* a dichos apuntadores (porque ofrecen un camino alternativo de acceso a los datos generalmente "más corto", es decir, más rápido, que el camino normal).

Dado un TAD T que define un tipo de interés t , que almacena elementos de tipo v , definimos el nuevo TAD $T_PARC_ABIERTO$ de la manera siguiente:

- Declaramos dentro de T un nuevo tipo denominado $atajo_t$, el tipo de los atajos a elementos de v almacenados en una estructura que implemente t . Este tipo formaliza la noción de apuntador y permite ocultar la forma concreta que toma este concepto a

nivel de implementación, donde puede añadirse información de control e incluso puede dejar de ser un entero si usamos el mecanismo de memoria dinámica para implementar la estructura (v. sección 3.3).

- Introducimos una nueva operación $\text{último_atajo} : t \rightarrow \text{atajo_t}$ que devuelve el atajo al último elemento insertado en la estructura. Da error si la estructura está vacía. Esta operación puede invocarse inmediatamente después de una inserción para obtener el atajo de todo elemento que entre en la estructura¹⁸.
- Introducimos una nueva operación $\text{dato} : t \text{ atajo_t} \rightarrow v$ que devuelve el elemento apuntado por el atajo. Da error si el atajo no da acceso sobre ningún elemento en la estructura. Para poder detectar esta situación, también añadimos la operación $\text{definido} : t \text{ atajo_t} \rightarrow \text{bool}$ ¹⁹.
- Además, definimos una operación para comparar atajos, $\text{ig} : t \text{ atajo_t atajo_t} \rightarrow \text{bool}$. Esta operación puede usarse para comparar elementos de manera eficiente en determinados contextos: si dos atajos son iguales, también lo son los elementos accesibles a través de ellos²⁰. Asimismo, los atajos podrán asignarse libremente mediante la instrucción de asignación.

La fig. 2.26 muestra la especificación ecuacional de la versión parcialmente abierta del TAD de los conjuntos acotados indexables. Introducimos dos operaciones privadas constructoras generadoras del género de los atajos; es importante que sean privadas para evitar la generación indiscriminada de atajos: los atajos sólo se generan cuando hay inserciones de elementos. Puede comprobarse que el modelo inicial restringido a este género es isomorfo a los naturales con el cero y la operación de sucesor, lo que se corresponde con la visión operativa que tenemos hasta ahora de los atajos como índices de vector.

En cuanto a los conjuntos, cabe señalar de entrada que se pierde la ecuación de conmutatividad, dado que el orden de entrada de los elementos es relevante de cara a la asignación de atajos. Por lo tanto, el modelo del TAD respecto este género deja de ser un conjunto y pasa a ser una secuencia. Para facilitar la especificación, se añade igualmente una operación privada añadir_con_atajo , constructora generadora junto a \emptyset (forman un conjunto puro); el atajo que se asigna al nuevo elemento es el siguiente del último asignado. Se controla en las ecuaciones de error tanto la asignación repetida de atajos (de hecho, este error no puede darse nunca dadas las ecuaciones) como el acceso por atajo inexistente.

¹⁸ También podría haberse modificado el resultado de las operaciones de inserción para que devolvieran directamente el atajo asignado, como por ejemplo hacen las bibliotecas LEDA y STL.

¹⁹ Aunque la propia dinámica del entorno de uso del TAD debería asegurar que nunca se intenta acceder por un atajo indefinido, sin necesidad de utilizar esta operación, se ha decidido dejarla pública.

²⁰ Si en la estructura pueden haber elementos repetidos, la comparación de atajos no proporciona suficiente información: dos elementos iguales pueden estar asociados a atajos diferentes.

universo CJT_ \in _ACOTADO_IND_PARC_ABIERTO (ELEM_CLAVE, VAL_NAT) es

usa BOOL

tipo cjt, atajo_cjt

ops \emptyset : \rightarrow cjt

$_ \cup \{ _ \}$: cjt elem \rightarrow cjt

$_ \in _$: clave cjt \rightarrow bool

indexa: cjt clave \rightarrow elem

último_atajo: cjt \rightarrow atajo_cjt

dato: cjt atajo_cjt \rightarrow elem

definido: cjt atajo_cjt \rightarrow bool

ig: cjt atajo_cjt atajo_cjt \rightarrow bool

privada añadir_con_atajo: cjt elem atajo \rightarrow cjt

privada primer_atajo: \rightarrow atajo_cjt

privada sig_atajo: atajo_cjt \rightarrow atajo_cjt

errores $\forall s \in \text{cjt}; \forall a \in \text{atajo}; \forall v \in \text{elem}; \forall k \in \text{clave}$

$[\text{definido}(s, a)] \Rightarrow \text{añadir_con_atajo}(s, v, a)$

$[\neg k \in s] \Rightarrow \text{indexa}(s, k)$

$[\neg \text{definido}(s, a)] \Rightarrow \text{dato}(s, a)$

$\text{último_atajo}(\emptyset)$

ecns $\forall s \in \text{cjt}; \forall a, b \in \text{atajo}; \forall v \in \text{elem}; \forall k \in \text{clave}$

$[v \in s] \Rightarrow s \cup \{v\} = s$

$[\neg v \in s] \Rightarrow s \cup \{v\} = \text{añadir_con_atajo}(s, v, \text{sig_atajo}(\text{último_atajo}(s)))$

$k \in \emptyset = \text{falso}$

$k \in \text{añadir_con_atajo}(s, v, a) = (\text{id}(v) = k) \vee (k \in s)$

$[\text{id}(v) = k] \Rightarrow \text{indexa}(\text{añadir_con_atajo}(s, v, a), k) = v$

$[\text{id}(v) \neq k] \Rightarrow \text{indexa}(\text{añadir_con_atajo}(s, v, a), k) = \text{indexa}(s, k)$

$\text{último_atajo}(\text{añadir_con_atajo}(s, v, a)) = a$

$[\text{ig}(s, a, b)] \Rightarrow \text{dato}(\text{añadir_con_atajo}(s, v, a), b) = v$

$[\neg \text{ig}(s, a, b)] \Rightarrow \text{dato}(\text{añadir_con_atajo}(s, v, a), b) = \text{dato}(s, b)$

$\text{definido}(\emptyset, a) = \text{falso}$

$\text{definido}(\text{añadir_con_atajo}(s, v, a), b) = (\text{ig}(s, a, b)) \vee (\text{definido}(s, b))$

$\text{ig}(s, \text{prim_atajo}, \text{prim_atajo}) = \text{cierto}$

$\text{ig}(s, \text{prim_atajo}, \text{sig_atajo}(a)) = \text{falso}; \text{ig}(s, \text{sig_atajo}(a), \text{prim_atajo}) = \text{falso}$

$\text{ig}(s, \text{sig_atajo}(a), \text{sig_atajo}(b)) = \text{ig}(s, a, b)$

funiverso

Fig. 2.26: especificación ecuacional de los conjuntos indexables parcialmente abiertos.

La política de asignación de atajos ha quedado simplificada por el hecho de que el TAD visto de los conjuntos no tiene operación de supresión. Éste no es el caso habitual. Si suponemos que en los conjuntos disponemos de una operación para borrar elementos, *borrar: cjt elem* \rightarrow *cjt*, el esquema se complica. Por un lado, para ver si un atajo está definido, debe comprobarse no sólo si se ha asignado sino también si el elemento al que se asignó no se ha borrado. Por otro lado, debe decidirse qué hacer con los atajos que se van liberando. Si bien podría pensarse en olvidarlos, es evidente que pensando en la implementación del tipo, los atajos liberados deben reaprovecharse; y si queremos que la implementación sea correcta respecto la especificación en el marco de la semántica inicial, también ésta debe reaprovechar los atajos a medida que se liberan. La especificación se complica un poco, pero puede modificarse para recoger este hecho; al insertar un elemento, si hay algún atajo liberado, se asigna el último que quedo libre y, si no lo hay, se asigna el siguiente al último asignado (para más detalles, v. el artículo de X. Franch y J. Marco "Adding Alternative Access Paths to Abstract Data Types", en *Proceedings Information Resource Management Association (IRMA) International Conference*, 2000).

El procedimiento aquí dado dota de un marco general para solucionar el problema de acceso directo a los elementos de una estructura, aunque cabe señalar que la solución no es universal y puede exigir algunos cambios en contextos concretos. Por ejemplo:

- Algunos TAD pueden exigir operaciones específicas. Por ejemplo, si en vez de conjuntos tuviéramos multiconjuntos (conjuntos con repetición), podría añadirse una operación para contar el número de repeticiones de un elemento accediendo mediante su atajo asociado, *naps: cjt_reps atajo_cjt_reps* \rightarrow *nat*.
- Algunos TAD tienen más de un tipo de interés. Por ejemplo, las diversas variantes de relaciones que presentamos en el capítulo 6. En este caso, deben duplicarse la mayoría de géneros y operaciones vistos.
- Algunas situaciones específicas pueden no encajar bien en este esquema. Sería el caso de los propios conjuntos con una operación de unión, en los que debería asegurarse la unicidad de los atajos de los dos conjuntos involucrados, o redefinir los atajos de los elementos que cambian de conjunto.

Consideremos ahora la implementación de los TAD parcialmente abiertos, y lo haremos mediante el ejemplo de los conjuntos. La solución más obvia consiste en representar los atajos mediante simples números naturales, de manera que el acceso por atajo se concrete en la indexación del vector usando dicho natural. Básicamente, pues, estamos usando el esquema de acoplamiento directo de estructuras encapsulando la información, y por ello, si bien esta solución simple funciona en algunos casos, en la mayoría de TAD adolece de algunos problemas que ya aparecían anteriormente, siendo el más importante la *obsolescencia de los atajos*. Para estudiar este problema, no tenemos más que considerar de nuevo la existencia de una operación de supresión en los conjuntos.

La obsolescencia de los atajos se manifiesta por dos motivos diferentes:

- El elemento al que apunta el atajo es eliminado de la estructura. Dado que los atajos no se eliminan junto con el elemento, la estructura debería controlar todo intento de acceder a un elemento no existente mediante un atajo. Se trata pues de tener en cuenta a nivel de implementación una situación que la especificación ya controla.
- Un elemento cambia de posición a causa de la estrategia de la implementación. Estas reubicaciones físicas se producen generalmente debido a inserciones y supresiones de elementos; en el caso de los conjuntos implementados mediante un vector ordenado (v. fig. 2.22), la supresión de un elemento implica el movimiento de los elementos que residen en posiciones mayores que la suya (v. fig. 2.27). En este caso, los atajos a estos elementos movidos quedan desactualizados.

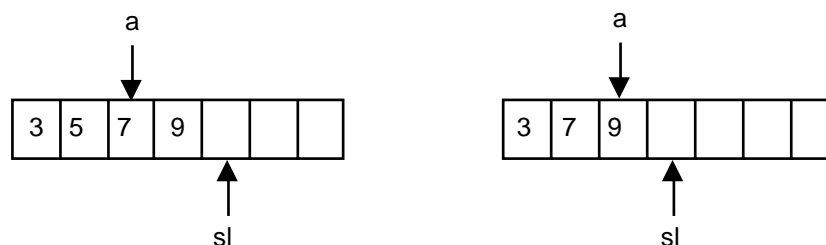


Fig. 2.27: conjunto implementado con vector ordenado con un atajo a al elemento 7 (izq.) y el mismo conjunto después de borrar el elemento 5 (der.), mostrando la obsolescencia de a.

Probablemente nos interese que los TAD parcialmente abiertos controlen la obsolescencia de atajos. Precisamente este control de seguridad es un elemento diferenciador más de este tipo de TAD respecto el uso directo de apuntadores incontrolados a los elementos. Eso sí, como puede intuirse este control va a tener un coste en eficiencia y deberemos evaluar si lo efectuamos o no. Precisamente este coste adicional provoca que bibliotecas que presentan el concepto de atajo, como LEDA o STL, no controlen la obsolescencia o lo hagan sólo de forma parcial. Una solución posible es tener dos versiones de TAD parcialmente abiertos, en una primando la seguridad y en otra la eficiencia.

Una primera solución sería que el TAD parcialmente abierto guardara para cada atajo el conjunto de (identificadores de) estructuras de datos que lo usan. Esta opción es complicada de implementar y además ocuparía mucho espacio. Por lo tanto, estudiamos un esquema en el que el propio TAD gestiona la integridad de la información, controlando que los parámetros de tipo atajo de las operaciones estén actualizados.

Por lo que se refiere al primer motivo de obsolescencia, podría pensarse en identificar los atajos obsoletos mediante *marcas*, es decir, valores especiales en algún campo ya existente

para registrar si la posición está ocupada o libre; si no es posible usar algún campo existente, se introduce uno booleano. Por ejemplo, si el atajo a es un camino de acceso al elemento v y este elemento v se borra, el acceso a la estructura mediante a resultaría un error. Pero como los atajos correspondientes a elementos borrados se deben reaprovechar, de manera que en el futuro el atajo a puede asociarse a un elemento w diferente, insertado en algún momento posterior a la solución. Si nos limitamos a cambiar el valor del campo de marca, el acceso por a encuentra un dato asociado, pero no tenemos la seguridad que se corresponda al valor correcto: si el acceso se realiza a partir de una estructura diferente que obtuvo el atajo a como vía de acceso a v , el elemento w no debería haberse obtenido.

Las soluciones a este problema son dos. Por un lado, podemos complicar la definición del TAD para controlar el uso de atajos en estructuras externas, incluyendo las operaciones *nuevo_uso*: $t \text{ atajo } t \rightarrow t$, que registra el momento en que una estructura almacena una copia de un atajo, y la simétrica *elimina_uso*: $t \text{ atajo } t \rightarrow t$. De esta manera, podría tenerse un contador de referencias para los atajos y éstos tan sólo se reaprovecharían cuando el contador estuviera a 0. Los problemas de integridad de esta solución son evidentes, pues el correcto estado de la estructura depende de la disciplina de sus usuarios (que, por ejemplo, deben evitar el uso directo de la instrucción de asignación aplicada sobre atajos), que además están obligados a llevar a cabo más trabajo. La fig. 2.28, izq., ilustra esta opción.

Alternativamente, podemos cambiar la definición de atajo añadiéndole información de control. Concretamente, un atajo será un par <apuntador, número de asignación> de manera que cuando se reaprovecha un atajo, se incrementa el segundo componente del TAD. El atajo pues es diferente cuando el elemento asociado lo es, y pueden detectarse los intentos de acceso incorrecto. La fig. 2.28, der., muestra este esquema; el atajo con número de asignación 2 está obsoleto.

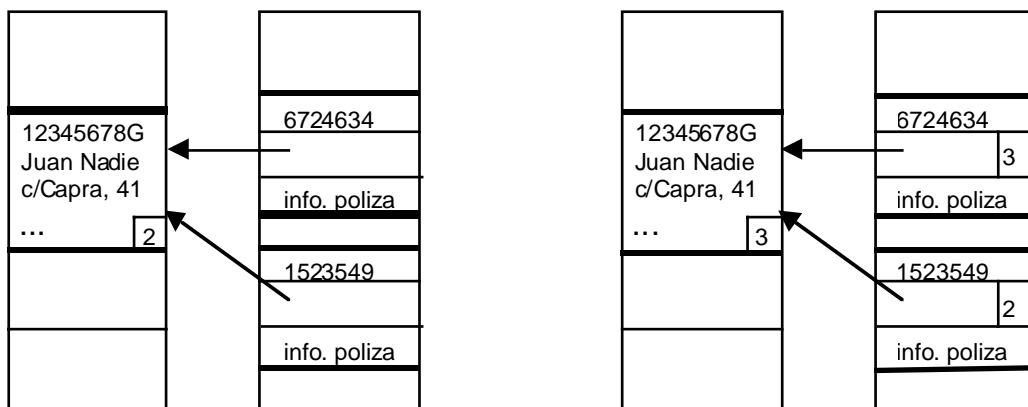


Fig. 2.28: control de obsolescencia con contador de referencias (izq.) y con información de control en el atajo (der.).

También cabe la posibilidad de considerar que, en realidad, esta modalidad de obsolescencia no tiene porqué ser controlada; parece razonable argumentar que no necesitamos ser más estrictos trabajando con atajos que sin ellos. Si consideramos la representación vista en la fig. 2.25, arriba der., la codificación de *datos_personales* probablemente no estará diseñada para controlar si ha habido algún cambio en la información asociada al nif del empleado de la póliza desde que la estructura almacenó dicho nif.

Queda pendiente el problema de las reubicaciones. Si queremos mantener los atajos actualizados cuando hay reubicaciones, no queda más remedio que desligar el concepto de atajo del de posición del vector original, de manera que añadimos al vector de elementos un segundo vector, indexado por atajos. De hecho, los elementos dejan de residir en el vector original y pasan a almacenarse en el nuevo vector; en el vector original pasa a haber los atajos a los elementos (v. fig. 2.29). Así, cuando se accede por atajo a la estructura, se encuentra directamente el elemento asociado; cuando se accede por las operaciones propias del conjunto original (por ejemplo, *indexa*), se obtienen atajos a elementos y es necesario usarlos para encontrar el elemento en sí. Finalmente, vemos que en las reubicaciones se actualizan los valores del vector original igual que antes, sólo que ahora en lugar de mover elementos se mueven atajos; los elementos no cambian su posición en el nuevo vector. El espacio ocupado por esta solución es mayor, no sólo por el vector índice, sino porque cada elemento del vector de elementos debe identificar cuál es su atajo, pero el precio a pagar es inevitable si realmente buscamos una implementación segura del concepto de atajo, más allá de la simple cosmética de la noción de apuntador. El coste asintótico temporal de las operaciones no varía, pues los algoritmos subyacentes son idénticos y cada acceso adicional a vector que se requiere tiene coste constante.

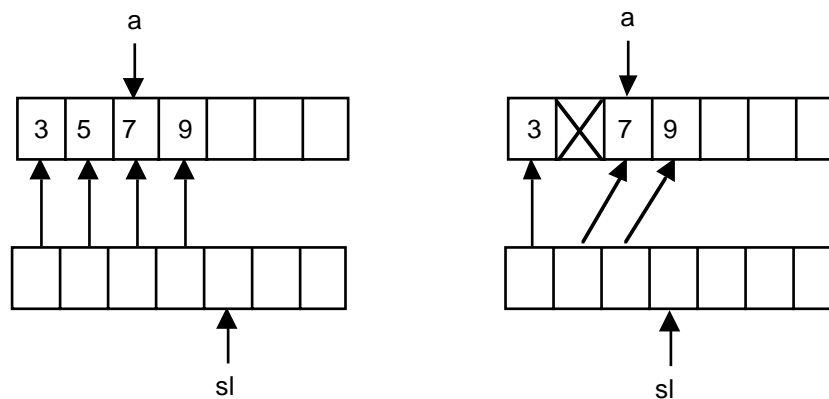


Fig. 2.29: Ídem fig. 2.27, con control de obsolescencia mediante vector auxiliar.

No olvidemos, no obstante, las supresiones. Si un elemento se borra de la estructura, el

atajo correspondiente debe quedar disponible para su reaprovechamiento. Para ello, podemos optar por marcar las posiciones correspondientes a atajos libres, con la consiguiente búsqueda (ineficiente) del próximo atajo a asignar cuando hay una nueva inserción. Otra posibilidad consiste en que añadamos un apuntador al último atajo liberado, desde la posición correspondiente a este atajo se apunta al siguiente en orden inverso de supresión, y así sucesivamente (v. fig. 2.30)²¹. En este caso, los atajos se obtienen de manera más rápida y de acuerdo con la política definida en la especificación. Notemos que esta solución exige más espacio para encadenar los atajos libres.

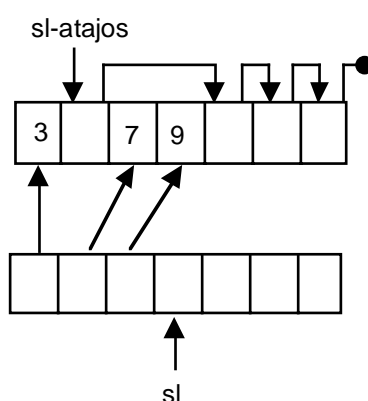


Fig. 2.30: control de atajos libres en la implementación de un TAD parcialmente abierto.

(b) TAD totalmente abiertos

En los *TAD totalmente abiertos* o, para abreviar, *TAD abiertos*, se permite el uso de atajos no sólo para acceder a los elementos de la estructura, sino para actualizar su contenido. Esta posibilidad se concreta en la existencia de operaciones de supresión y de modificación de elementos dado su atajo asociado, aunque eventualmente puede haber también operaciones de inserción de nuevos elementos relativas a atajos a elementos ya existentes (por ejemplo, inserción detrás de un atajo).

Por lo que se refiere a la implementación, la supresión por atajo puede colisionar con la estrategia de representación de los elementos, como queda claro con el ejemplo mismo de los conjuntos. Podemos comprobar en la representación de la fig. 2.30 que, en caso de suprimir el 7 accediendo directamente por atajo, debe eliminarse del vector base el apuntador que hay en la segunda posición. Por ello, debe ser posible acceder desde el vector indexado por atajos al vector original, y para ello podemos usar el campo de

²¹ En realidad, esta estrategia ya nos está induciendo el concepto de representación encadenada que se introduce en la sección 3.3, donde se explica este concepto en más detalle. En concreto, la representación que aquí se está proponiendo se denomina pila de sitios libres.

encadenamiento de los atajos libres, que en el caso de los atajos desocupados estaba desocupado (v. fig. 2.31).

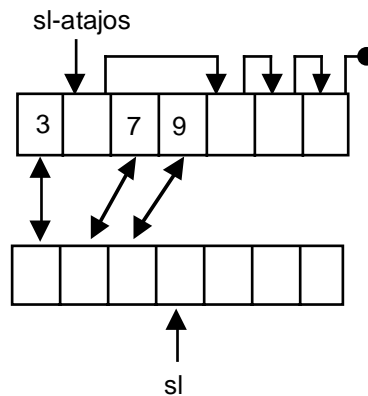


Fig. 2.31: representación de los TAD totalmente abiertos.

c) Algunas consideraciones finales

Hemos estudiado en este apartado y en anterior dos conceptos, los iteradores y los atajos, que permiten reconciliar modularidad y eficiencia de manera satisfactoria. Ambos conceptos se han plasmado de diversas maneras, tanto a nivel de modelo como de implementación. Todas las variantes vistas a ambos niveles pueden ser aplicadas a la mayoría de TAD que se estudiarán en los cuatro próximos capítulos, y se comentarán las especificidades de su uso en cada caso, sin entrar en demasiados detalles por motivo de espacio.

La uniformidad del tratamiento de iteradores y atajos da pie a preguntarnos si es posible mecanizar el añadido de estas facilidades a los TAD. La respuesta es afirmativa. Podemos formular las propuestas mediante universos parametrizados que añadan ambos mecanismos a TAD cualesquiera, siempre y cuando cumplan unos requisitos mínimos de similitud (por ejemplo, no podremos tratar de la misma manera TAD relativos a secuencias y a relaciones porque el número de tipos de interés involucrados en cada caso es diferente). Esta propuesta genérica ha sido aplicada a una biblioteca de componentes de amplia difusión, la Booch Components para Ada-95, con resultados satisfactorios (v. "Reengineering the Booch Components Library, de J. Marco y X. Franch, en *Proceedings 5th Ada-Europe International Conference*, LNCS 1845, 2000).

Eso sí, tal y como ya habíamos comentado, el uso de atajos, así como el de los iteradores presentados en esta misma sección, puede tener un coste importante en la eficiencia de la implementación del TAD. Por ello, la utilización de estos tipos debe quedar restringida a los ámbitos en que los requisitos de eficiencia realmente exigen estas versiones de TAD (porque prima el recorrido o acceso eficiente sobre otras consideraciones).

Ejercicios

2.1 Escribir una implementación para los conjuntos acotados con pertenencia usando el tipo V de los elementos como índice de un vector A de booleanos, tal que $A[v]$ vale cierto si v está en el conjunto. Suponer que la cardinalidad de V es n y que se dispone de una función inyectiva $h: V \rightarrow [1, n]$. Escribir la especificación pre-post de las operaciones. Verificar la corrección de la implementación respecto a la especificación usando alguna de las estrategias vistas en la sección 2.2.

2.2 a) Escribir una implementación para los polinomios $\mathbb{Z}[X]$ especificados en el apartado 1.5.1, de modo que se almacenen en un vector todos los monomios de coeficiente diferente de cero sin requerir ninguna condición adicional. Escribir la especificación pre-post de las diferentes operaciones y también los invariantes y las funciones de acotamiento de los diversos bucles que en ella aparezcan. Verificar la corrección de la implementación respecto a la especificación usando alguna de las estrategias vistas en la sección 2.2. Calcular el coste asintótico de la implementación.

b) Repetir el apartado anterior requiriendo que los pares del vector estén ordenados en forma creciente por exponente.

2.3 Sea el TAD de los multiconjuntos (conjuntos que admiten elementos repetidos; es decir, el multiconjunto $\{x, x\}$ es válido y diferente del multiconjunto $\{x\}$) sobre un dominio V con operaciones de crear el multiconjunto vacío y añadir un elemento al multiconjunto. Proponer una representación (que incluya la función de abstracción, el invariante de la representación y la relación de igualdad) para las dos situaciones siguientes:

a) El número total de elementos del multiconjunto está acotado por n .

b) El número total de elementos del multiconjunto no está acotado, pero, en cambio, se sabe que la cardinalidad de V es n y que se dispone de una función inyectiva $h: V \rightarrow [1, n]$.

2.4 Sean las funciones $f_1(n) = n^3$, $f_2(n) = 90000n^2 + 70000n$, $f_3(n) = n^2 \log n$, $f_4(n) = n^3 \log n$, $f_5(n) = n^3 + \log n$. Estudiar, para todo par de valores diferentes i, j entre 1 y 5, si $f_i \in O(f_j)$, si $f_i \in \Omega(f_j)$ y si $f_i \in \Theta(f_j)$. Representar gráficamente los conjuntos $O(f_j)$, $\Omega(f_j)$ y $\Theta(f_j)$.

2.5 a) Sea el polinomio $p(n) = c_k n^k + c_{k-1} n^{k-1} + \dots + c_1 n + c_0$, con $\forall i: 0 \leq i \leq k-1: c_i \geq 0$, y además $c_k > 0$. Demostrar que $p(n) \in \Theta(n^k)$. **b)** Demostrar que $\sum i: 1 \leq i \leq n: i^k \in \Theta(n^{k+1})$, $k > 0$. **c)** Demostrar que $\forall a, b: a > 1 \wedge b > 1: \log_a n \in \Theta(\log_b n)$.

2.6 Las notaciones O y Ω tienen asociadas otras dos, *o pequeña* (ing., *small o*) y *ω pequeña* (ing., *small omega*), abreviadamente o y ω , que definen cotas significativamente más fuertes que las primeras y que se definen, para $f: \mathcal{N}^+ \rightarrow \mathcal{N}^+$:

$$o(f) = \{g: \mathcal{N}^+ \rightarrow \mathcal{N}^+ / \forall c_0: c_0 \in \mathcal{R}^+: \exists n_0: n_0 \in \mathcal{N}^+: \forall n: n \geq n_0: g(n) \leq c_0 f(n)\}$$

$$\omega(f) = \{g: \mathcal{N}^+ \rightarrow \mathcal{N}^+ / \forall c_0: c_0 \in \mathcal{R}^+: \exists n_0: n_0 \in \mathcal{N}^+: \forall n: n \geq n_0: g(n) \geq c_0 f(n)\}$$

Estudiar si estas notaciones son reflexivas, simétricas o transitivas, así como sus relaciones con las notaciones O y Ω . Mostrarlas gráficamente con un esquema similar al de la fig. 2.11.

2.7 Dados los siguientes algoritmos, formular su especificación pre-post, establecer los invariantes y las funciones de acotamiento de sus bucles y calcular su coste.

a) Producto de matrices.

```

acción producto (ent A, B son vectores [de 1 a n, de 1 a n] de enteros;
                    sal C es vector [de 1 a n, de 1 a n] de enteros) es
var fil, col, índice son enteros fvar
    para todo fil desde 1 hasta n hacer
        para todo col desde 1 hasta n hacer {cálculo de C[fil, col]}
            C[fil, col] := 0
            para todo índice desde 1 hasta n hacer
                C[fil, col] := C[fil, col] + (A[fil, índice]*B[indice, col])
            fpara todo
        fpara todo
    facción

```

b) Ordenación de un vector por el método de la burbuja.

```

acción burbuja (ent/sal A es vector [de 1 a n] de enteros) es
var i, j, aux son enteros fvar
    para todo i desde 1 hasta n-1 hacer {localización del i-ésimo menor elemento}
        para todo j bajando desde n hasta i+1 hacer
            {examen de la parte del vector todavía no ordenada}
            si A[j-1] > A[j] entonces {intercambio de los elementos}
                aux := A[j]; A[j] := A[j-1]; A[j-1] := aux
            fsi
        fpara todo
        {el i-ésimo menor elemento ya está en A[i]}
    fpara todo
facción

```

c) Búsqueda dicotómica en un vector.

función búsqueda_dicot (A es vector [de 1 a n] de enteros; x es entero) devuelve bool es
var izq, der, med son enteros; encontrado es bool fvar
 {se determinan los extremos de la porción del vector donde puede estar x}
 izq := 0; der := n+1
 {se busca mientras quede vector por explorar}
mientras (izq < der-1) hacer
 med := (izq + der) div 2
 opción
 caso x < A[med] hacer der := med
 caso x ≥ A[med] hacer izq := med
 fopción
fmientras
si izq = 0 entonces encontrado := falso sino encontrado := (A[izq] = x) fsi
devuelve encontrado

2.8 En el artículo de Vitányi y Meertens citado en el inicio de la sección 2.3 se propone una definición alternativa de la notación Ω :

$$\Omega(f) = \{g: \mathcal{N}^+ \rightarrow \mathcal{N}^+ / \exists c_0: c_0 \in \mathcal{R}^+: \forall n_0: n_0 \in \mathcal{N}^+: \exists n: n \geq n_0: g(n) \geq c_0 f(n)\}$$

Esta definición sirve también para el caso en que el algoritmo a analizar presente oscilaciones en su comportamiento, porque no exige que g sea una cota superior de f a partir de un cierto punto sino sólo que g sobrepase o iguale f un número infinito de veces. La utilidad de este caso se hace evidente en algoritmos que tienen un coste para algunas entradas y otro diferente para el resto. Estudiar su reflexividad, simetría y transitividad y establecer la relación de esta Ω con la dada en la sección 2.3. Imaginar un algoritmo que tenga un coste diferente según las dos definiciones dadas de Ω .

2.9 En algunos casos, es interesante disponer de TAD recorribles ordenadamente por más de un criterio. Formular una signatura y su correspondiente especificación adecuadas a este requisito para el caso de los conjuntos. Estudiar el coste de las diferentes implementaciones aplicadas a este contexto.

2.10 Sea el TAD de los multiconjuntos propuesto en el ejercicio 2.3. Establecer la signatura e implementar las versiones recorrible y recorrible ordenadamente en los dos casos que se mencionan en dicho ejercicio. Calcular la eficiencia resultante.

2.11 En el apartado 2.5.3 se han descrito diversas alternativas para el acceso directo a los datos almacenados en una estructura mediante atajos. Construir una tabla comparativa que sumarice los resultados de estas alternativas respecto: la eficiencia de las operaciones; el espacio real empleado; la integridad de la estructura; y la facilidad de uso.

Capítulo 3 Secuencias

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

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

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

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

3.1 Pilas

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

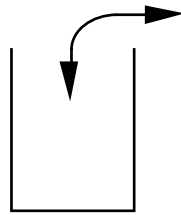


Fig. 3.1: representación gráfica de las pilas, dada su gestión.

Este comportamiento es habitual en la vida cotidiana; por ejemplo, si apilamos todos los platos de un banquete uno encima de otro, cualquier intento de obtener un plato del medio de la pila acaba irremediabilmente con la ruptura de la vajilla. En la programación, las pilas no sólo se utilizan en el diseño de estructuras de datos, sino también como estructura auxiliar en diversos algoritmos y esquemas de programación; por ejemplo, en la transformación de algunas clases de funciones recursivas a iterativas (v. recorridos de árboles y grafos en los capítulos 4 y 6), o bien en la evaluación de expresiones (v. sección 7.1). Otra situación típica es la gestión de las direcciones de retorno que hace el sistema operativo en las invocaciones a procedimientos de un programa imperativo (v. [HoS94, pp. 97-99]). Sea un programa P que contiene tres acciones $A1$, $A2$ y $A3$, que se invocan tal como se muestra en la fig. 3.2 (r , s y t representan la dirección de retorno de las llamadas a los procedimientos). Durante la ejecución de $A3$ se disponen los puntos de retorno de los procedimientos dentro de una pila, tal como se muestra en la misma figura (m representa la dirección a la que el programa ha de devolver el control al finalizar su ejecución), de manera que el sistema operativo siempre sabe a dónde devolver el control del programa al acabar la ejecución del procedimiento en curso, obteniendo el elemento superior de la pila y eliminándolo seguidamente.

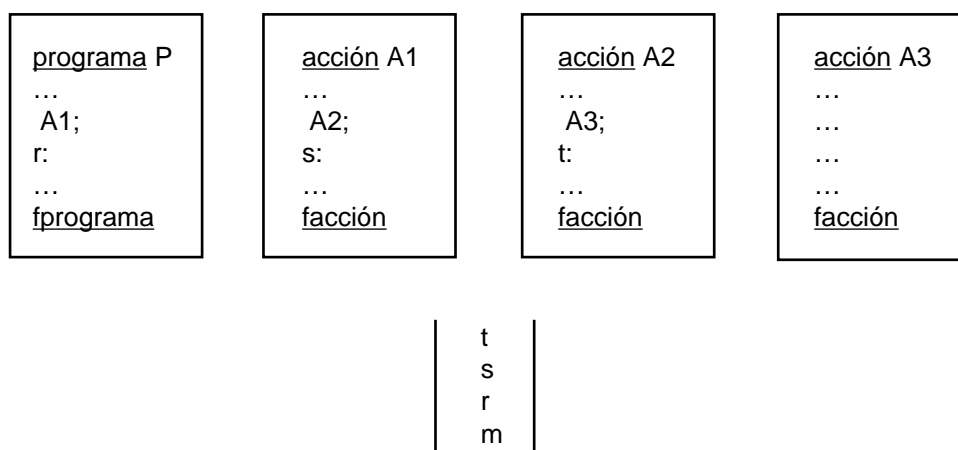


Fig. 3.2: un programa con tres acciones (arriba) y las direcciones de retorno dentro de una pila (abajo).

3.1.1 Especificación

Dada una pila correspondiente a la secuencia $p \in V^*$, $p = v_0 v_1 \dots v_n$, con $\forall i: 0 \leq i \leq n: v_i \in V$, y dado un elemento $v \in V$, se pueden definir diversas operaciones de interés:

- Crear la pila vacía: *crea* (ing., *create*), devuelve la pila λ .
- Insertar un elemento: *empila*(p, v) (ing., *push*), devuelve $v_0 v_1 \dots v_n v$.
- Sacar un elemento: *desempila*(p) (ing., *pop*), devuelve $v_0 v_1 \dots v_{n-1}$.
- Consultar un elemento: *cima*(p) (ing., *top*), devuelve v_n .
- Decidir si la pila está vacía: *vacía?*(p) (ing., *empty?*), devuelve cierto si $p = \lambda$ y falso en caso contrario.

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

```

universo PILA(ELEM) define
  usa BOOL
  tipo pila
  ops
    crea: → pila
    empila: pila elem → pila
    desempila: pila → pila
    cima: pila → elem
    vacía?: pila → bool
  errores desempila(crea); cima(crea)
  ecns  $\forall p \in \text{pila}; \forall v \in \text{elem}$ 
    desempila(empila(p, v)) = p; cima(empila(p, v)) = v
    vacía?(crea) = cierto; vacía?(empila(p, v)) = falso
  funiverso

```

Fig. 3.3: especificación del TAD de las pilas.

3.1.2 Implementación

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

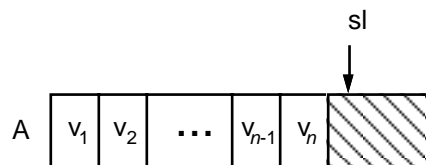


Fig. 3.4: representación secuencial de la pila $v_1v_2\dots v_n$.

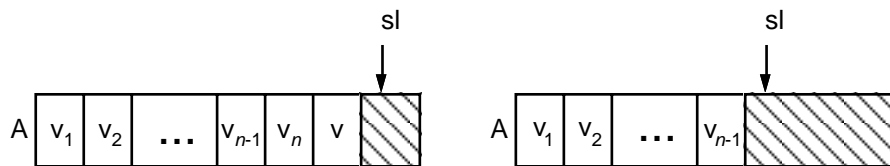


Fig. 3.5: apilamiento de un elemento (izq.) y desapilamiento (der.) en la pila de la fig. 3.4.

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

```

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

```

Fig. 3.6: implementación secuencial de las pilas.

El coste temporal de las operaciones es óptimo, $\Theta(1)$ en todos los casos (siendo ésta una característica común en las estructuras lineales más simples), siempre y cuando el coste de duplicar elementos sea negligible¹. Por lo que se refiere al coste espacial, una pila tiene un espacio reservado $\Theta(val)$, independientemente del número de elementos que la formen en un momento dado², considerando negligible a su vez el espacio ocupado por los elementos.

Por último, comentar que las versiones recorribles y abiertas se implementan igual que en los conjuntos. Sin embargo, en el caso de pilas recorribles ordenadamente según el valor de los elementos, no podemos mantener el vector ordenado y es necesario ordenarlo antes del recorrido, trabajando sobre una copia para no destruir el contenido; el coste del recorrido queda $\Theta(n \log n)$ en tiempo y $\Theta(n)$ en espacio auxiliar, siendo n el número de elementos.

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

$$\begin{aligned} & \{ sl_1 \leq val \} \\ & \quad \langle A_2, sl_2 \rangle := empila(\langle A_1, sl_1 \rangle, v) \\ & \{ sl_2 \leq val \wedge abs_{pila}(\langle A_2, sl_2 \rangle) = empila(abs_{pila}(\langle A_1, sl_1 \rangle), v) \} \end{aligned}$$

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

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

$$\begin{aligned} (A1) \quad & abs_{pila}(\langle A, 0 \rangle) = crea \\ (A2) \quad & abs_{pila}(\langle A, x+1 \rangle) = empila(abs_{pila}(\langle A, x \rangle), A[x]) \end{aligned}$$

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

$$(R) \quad \mathcal{R}_{pila}(\langle A_1, sl_1 \rangle, \langle A_2, sl_2 \rangle) = (sl_1 = sl_2) \wedge (\forall k: 0 \leq k \leq sl_1-1: A_1[k] = A_2[k])$$

¹ Como ya se ha dicho en el capítulo 2, para simplificar el estudio de la eficiencia, en el resto del libro nos centraremos en el coste intrínseco de las operaciones del TAD, sin considerar el coste de las duplicaciones, comparaciones y otras operaciones elementales de los parámetros formales de los universos que los definen. Si dichos costes no fueran negligibles, deberían considerarse para obtener la eficiencia auténtica del TAD.

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

universo VECTOR(ELEM, VAL_NAT) és
usa NAT, BOOL
tipo vector
ops crea: \rightarrow vector
as: vector nat elem \rightarrow vector
cons: vector nat \rightarrow elem
errores $\forall A \in \text{vector}; \forall i \in \text{nat}; \forall v \in \text{elem}$
 $[i \geq \text{val}] \Rightarrow \text{as}(A, i, v), \text{cons}(A, i); \text{cons}(\text{crea}, i)$
ecns $\forall A \in \text{vector}; \forall i, j \in \text{nat}; \forall v, w \in \text{elem}$
(E1) $\text{as}(\text{as}(A, i, v), i, w) = \text{as}(A, i, w)$
(E2) $[\neg \text{NAT.ig}(i, j)] \Rightarrow \text{as}(\text{as}(A, i, v), j, w) = \text{as}(\text{as}(A, j, w), i, v)$
(E3) $\text{cons}(\text{as}(A, i, v), i) = v$
(E4) $[\neg \text{NAT.ig}(i, j)] \Rightarrow \text{cons}(\text{as}(A, i, v), j) = \text{cons}(A, j)$
funiverso

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

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

$$sl_1 \leq \text{val} \Rightarrow (sl_1 + 1 \leq \text{val} \wedge \text{abs}_{\text{pila}}(\langle \text{as}(A_1, sl_1, v), sl_1 + 1 \rangle) = \text{empila}(\text{abs}_{\text{pila}}(\langle A_1, sl_1 \rangle), v))$$

Estudiamos primero la satisfacción de la segunda fórmula de la conclusión:

$$\begin{aligned}
&\text{abs}_{\text{pila}}(\langle \text{as}(A_1, sl_1, v), sl_1 + 1 \rangle) = \{ \text{aplicando (A2)} \} \\
&\text{empila}(\text{abs}_{\text{pila}}(\langle \text{as}(A_1, sl_1, v), sl_1 \rangle), \text{cons}(\text{as}(A_1, sl_1, v), sl_1)) = \{ \text{aplicando (E3)} \} \\
&\text{empila}(\text{abs}_{\text{pila}}(\langle \text{as}(A_1, sl_1, v), sl_1 \rangle), v) = \{ \text{conjetura} \} \\
&\text{empila}(\text{abs}_{\text{pila}}(\langle A_1, sl_1 \rangle), v)
\end{aligned}$$

La conjetura que aparece establece que una asignación en una posición libre del vector no afecta el resultado de la función de abstracción. La demostración es obvia aplicando (R) ya que el predicado $\mathcal{R}_{\text{pila}}(\langle \text{as}(A_1, sl_1, v), sl_1 \rangle, \langle A_1, sl_1 \rangle)$ efectivamente evalúa a cierto.

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

3.2 Colas

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

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



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

3.2.1 Especificación

Dada una cola correspondiente a la secuencia $c \in V^*$, $c = v_0 v_1 \dots v_n$, con $\forall i: 0 \leq i \leq n: v_i \in V$, y dado un elemento $v \in V$, definimos las operaciones siguientes:

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

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

```

universo COLA (ELEM) define
  usa BOOL
  tipo cola
  ops
    crea: → cola
    encola: cola elem → cola
    desencola: cola → cola
    cabeza: cola → elem
    vacía?: cola → bool
  errores desencola(crea); cabeza(crea)
  ecns  $\forall c \in \text{cua}; \forall v \in \text{elem}$ 
    desencola(encola(crea, v)) = crea
     $[\neg \text{vacía?}(c)] \Rightarrow \text{desencola}(\text{encola}(c, v)) =$ 
      encola(desencola(c), v)
    cabeza(encola(crea, v)) = v
     $[\neg \text{vacía?}(c)] \Rightarrow \text{cabeza}(\text{encola}(c, v)) = \text{cabeza}(c)$ 
    vacía?(crea) = cierto
    vacía?(encola(c, v)) = falso
  funiverso

```

Fig. 3.9: especificación del TAD de las colas.

3.2.2 Implementación

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

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

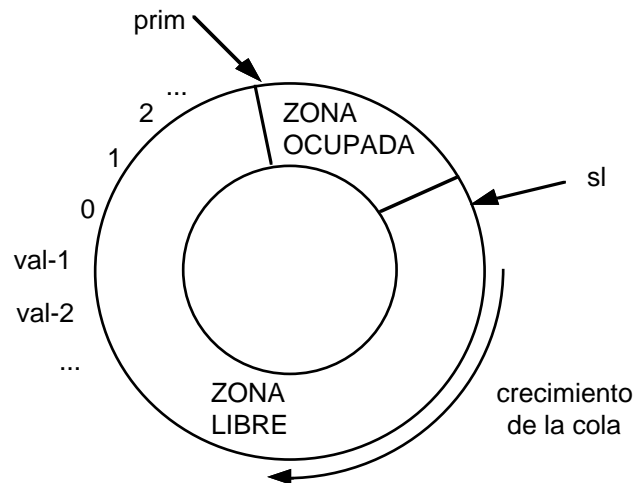


Fig. 3.10: implementación de una cola dentro de un vector circular de val posiciones.

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

En la fig. 3.11 se presenta una implementación en Merlí de las colas acotadas que sigue todas estas consideraciones, estableciendo el invariante de la representación que acota los valores válidos de los apuntadores y del contador y, además, los relaciona. Notemos el uso de mod (la operación que devuelve el resto de la división de dos números naturales) para tratar el vector circularmente. El coste temporal de las operaciones es $\Theta(1)$ en todos los casos, y el coste espacial de la estructura queda $\Theta(val)$. Por lo que respecta a las variantes recorribles y abiertas del TAD, se aplican los comentarios efectuados sobre las pilas.

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

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

```

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

```

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

3.3 Listas

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

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

3.3.1 Especificación de las listas con punto de interés

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

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

Destaquemos la similitud de este TAD con un TAD recorrible, que se traducirá incluso en la existencia de las cuatro operaciones que caracterizan a éstos. La diferencia fundamental es que en las listas la interacción entre actualizaciones y recorridos está contemplada inequívocamente a nivel modelo, y por ello su comportamiento es claramente previsible. Una consecuencia de esta similitud es que raramente nos interesará tener un TAD de las listas con punto de interés y recorribles, por lo que no lo estudiaremos en este texto.

La existencia del punto de interés conduce a un modelo de secuencia diferente a los vistos hasta ahora, similar a los modelos de TAD recorribles. Se distingue el trozo de secuencia a la izquierda del punto de interés y el trozo que va del elemento distinguido al de la derecha del

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

todo, y por eso el modelo es realmente $V^* \times V^*$. Notemos que cualquiera de las dos partes puede ser λ según la situación del punto de interés. En consecuencia, dada una lista l , $l = \langle s, t \rangle \in V^* \times V^*$, $s = s_0 s_1 \dots s_n$, $t = t_0 t_1 \dots t_m$, $\forall i, j: 0 \leq i \leq n \wedge 0 \leq j \leq m: s_i, t_j \in V$, y dado un elemento $v \in V$, se definen las operaciones:

- Crear la lista vacía: *crea*, devuelve la lista $\langle \lambda, \lambda \rangle$; es decir, devuelve una lista sin elementos y el punto de interés queda a la derecha del todo.
- Insertar un elemento: *inserta*(l, v) (ing., *insert*), devuelve $\langle s_0 s_1 \dots s_n v, t_0 t_1 \dots t_m \rangle$; es decir, inserta delante del punto de interés, que no cambia.
- Sacar un elemento: *borra*(l) (ing., *delete* o *remove*), devuelve $\langle s_0 s_1 \dots s_n, t_1 \dots t_m \rangle$; es decir, borra el elemento distinguido (en caso de que el punto de interés esté a la derecha del todo, da error) y el punto de interés queda sobre el siguiente elemento (o a la derecha del todo, si el borrado es el último).
- Consultar un elemento: *actual*(l) (ing., *current* o *get*), devuelve t_0 ; es decir, devuelve el elemento distinguido (o da error si el punto de interés está a la derecha del todo).
- Decidir si la lista está vacía o no: *vacía?*(l), devuelve cierto si $l = \langle \lambda, \lambda \rangle$ y falso en caso contrario.
- Poner al inicio el punto de interés: *principio*(l) (ing., *reset*), devuelve $\langle \lambda, s \cdot t \rangle$; es decir, el punto de interés queda sobre el primer elemento, si hay, o bien a la derecha del todo, si no hay ninguno.
- Avanzar el punto de interés: *avanza*(l) (ing., *next*), devuelve $\langle s_0 s_1 \dots s_n t_0, t_1 \dots t_m \rangle$; es decir, el elemento distinguido pasa a ser el siguiente al actual (o da error si el punto de interés está a la derecha del todo).
- Mirar si el punto de interés está a la derecha del todo: *final?*(l) (ing., *end?* o *eol?*), devuelve cierto si $t = \lambda$ y falso en caso contrario.

Como presenta las cuatro operaciones que caracterizan los TAD recorribles, la signatura vista nos permite implementar los conocidos esquemas de recorrido y de búsqueda. En este contexto, podemos ser más precisos que cuando los formulamos en general en el apartado 2.4.2 y establecer así pre y postcondiciones e invariantes (v. fig. 3.12). En las pre y postcondiciones y también en los invariantes de estas y otras funciones, usaremos una operación $_ \in _$ tal que, dado un elemento v y una lista l , $v \in l$ comprueba si v está dentro de l y también dos funciones pi y pd que, dada una lista $l = \langle s, t \rangle$, devuelven s y t , respectivamente.

$\{l = \langle v_0 v_1 \dots v_{k-1}, v_k \dots v_n \rangle\}$	$\{l = \langle v_0 v_1 \dots v_{k-1}, v_k \dots v_n \rangle\}$
$l := \text{principio}(l)$	$l := \text{principio}(l); \text{está} := \text{falso}$
<u>mientras</u> $\neg \text{final?}(l)$ <u>hacer</u>	<u>mientras</u> $\neg \text{final?}(l) \wedge \neg \text{está}$ <u>hacer</u>
$\{I \equiv \forall i: 0 \leq i \leq \text{pi}(l) : T(v_i)\}$	$\{I \equiv (\forall i: 0 \leq i \leq \text{pi}(l) : \neg A(v_i)) \wedge \text{está} \Rightarrow A(\text{actual}(l))\}$
$T(\text{actual}(l))$	<u>si</u> $A(\text{actual}(l))$ <u>entonces</u> $\text{está} := \text{cierto}$
$l := \text{avanza}(l)$	<u>si no</u> $l := \text{avanza}(l)$
<u>fmientras</u>	<u>fsi</u>
$\{\forall i: 0 \leq i \leq n: T(v_i)\}$	<u>fmientras</u>
	$\{\text{está} \equiv \exists i: 0 \leq i \leq n: A(v_i) \wedge \text{está} \Rightarrow A(\text{actual}(l))\}$

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

La especificación algebraica de este tipo de listas se basa precisamente en el modelo que se acaba de introducir, donde se considera que una lista es un par de secuencias, concretamente pilas dadas las operaciones que se necesitan⁴. Estas dos pilas están confrontadas, de manera que la pila de la izquierda tiene la cima justo antes del punto de interés y la pila de la derecha tiene como cima el punto de interés, y sus sentidos de crecimiento son opuestos. Sobre estas pilas se necesita un enriquecimiento, *concat*, que pasa todos los elementos de la pila de la izquierda a la pila de la derecha; la operación se especifica convenientemente en el mismo universo de las listas. El universo genérico (v. fig. 3.13) queda finalmente definido sobre los pares de dos pilas (la secuencia de la izquierda y la secuencia de la derecha) con una operación privada que es la constructora generadora del género. Observemos que las operaciones del TAD *lista* han sido especificadas respecto a las constructoras generadoras del TAD *pila* donde ha sido necesario. En concreto, se ha distinguido la pila vacía (representada por *crea*) de la pila no vacía (representada por *empila(p, v)*) allí donde ha convenido. El resultado es correcto porque toda operación ha quedado definida para todas las listas posibles. Destaquemos también que la instancia de las pilas ha sido efectuada con el parámetro formal de las listas, y como los dos parámetros se definen en el mismo universo de caracterización, es necesario bautizarlos en la cabecera de los universos genéricos para poder distinguir el tipo *elem* en ambos contextos. La instancia es privada porque el usuario de las listas no ha de tener acceso a los símbolos resultantes.

Cabe destacar que esta estrategia de especificación puede generalizarse a la mayoría de TAD recorribles, tanto ordenadamente como no ordenadamente.

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

universo LISTA_INTERÉS (A es ELEM) es

usa BOOL

instancia privada PILA (B es ELEM) donde B.elem es A.elem

ops privada concat: pila pila \rightarrow pila

ecns $\forall p, p_1, p_2 \in \text{pila}; \forall v \in \text{elem}$

concat(crea, p) = p

concat(empila(p₁, v), p₂) = concat(p₁, empila(p₂, v))

tipo lista

ops

privada <_, _>: pila pila \rightarrow lista

crea: \rightarrow lista

inserta: lista elem \rightarrow lista

borra, principio, avanza: lista \rightarrow lista

actual: lista \rightarrow elem

final?, vacía?: lista \rightarrow bool

errores $\forall p \in \text{pila}$

borra(<p, PILA.crea>); avanza(<p, PILA.crea>)

actual(<p, PILA.crea>)

ecns $\forall p, p_1, p_2 \in \text{pila}; \forall v \in \text{elem}$

crea = <PILA.crea, PILA.crea>

inserta(<p₁, p₂>, v) = <PILA.empila(p₁, v), p₂>

borra(<p₁, PILA.empila(p₂, v)>) = <p₁, p₂>

principio(<p₁, p₂>) = <PILA.crea, concat(p₁, p₂)>

avanza(<p₁, PILA.empila(p₂, v)>) = <PILA.empila(p₁, v), p₂>

actual(<p₁, PILA.empila(p₂, v)>) = v

final?(<p₁, p₂>) = PILA.vacía?(p₂)

vacía?(<p₁, p₂>) = PILA.vacía?(p₁) \wedge PILA.vacía?(p₂)

funiverso

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

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

3.3.2 Implementación de las listas con punto de interés

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

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

a) Representación secuencial

Mostrada en las figs. 3.14 y 3.15, su filosofía es idéntica a la representación de las pilas y las colas; sólo es necesario un apuntador adicional al elemento distinguido para poder aplicar las operaciones propias del modelo de lista. El problema principal de esta implementación es evidente: una inserción o supresión en cualquier posición del vector obliga a desplazar algunos elementos para no tener posiciones desaprovechadas, lo que resulta en un coste lineal sobre el número n de elementos, $\Theta(n)$ en el caso peor (al insertar al principio o borrar el primer elemento) inadmisibles cuando estas operaciones son frecuentes, la dimensión de los elementos es muy grande o en las versiones abiertas del TAD, en este último caso debido a la desactualización de los atajos. La resolución de estos problemas lleva a la representación encadenada.

```

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

```

Fig. 3.14: implementación secuencial de las listas.

```

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

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

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

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

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

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

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

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

funiverso

```

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

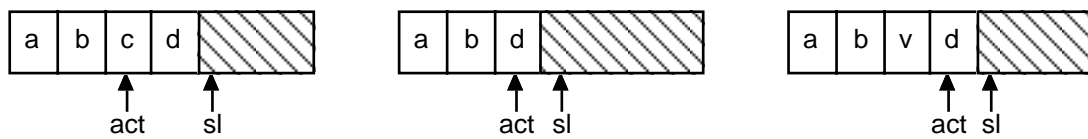


Fig. 3.15: operaciones sobre listas con punto de interés: izquierda, la lista $\langle ab, cd \rangle$; centro, supresión del elemento distinguido; derecha, inserción del elemento v en la lista resultante.

b) Representación encadenada

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

El concepto de representación encadenada es extensible a todas las estructuras lineales estudiadas hasta ahora.

```

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

```

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

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

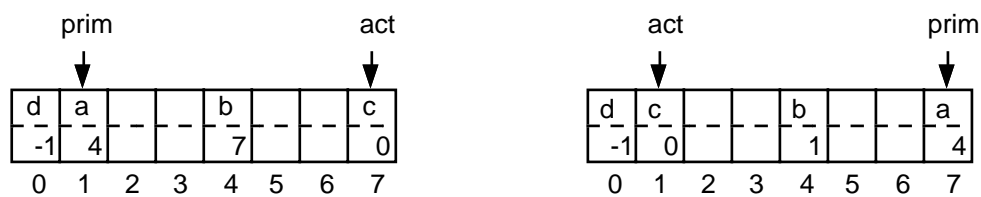


Fig. 3.17(a): dos posibles representaciones encadenadas de la lista <ab, cd>.

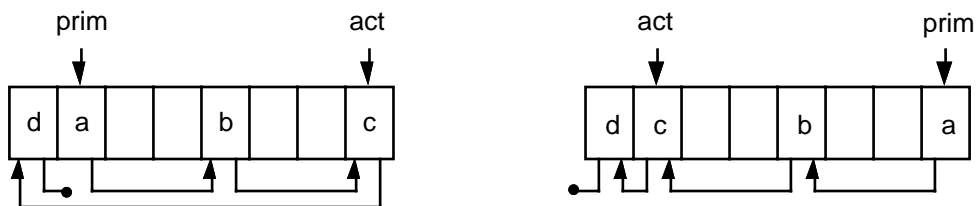


Fig. 3.17(b): ídem, pero mostrando gráficamente los encadenamientos.

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

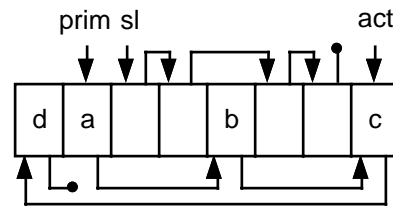


Fig. 3.18: Ídem fig. 3.17(b), izquierda, con gestión del espacio libre.

La representación resultante presenta todavía un problema más: al insertar y borrar elementos es necesario modificar el encadenamiento del elemento anterior al distinguido. Actualmente, la única manera de encontrar el elemento anterior al actual es recorrer la lista desde el inicio hasta llegar al elemento en cuestión, por lo que la inserción y la supresión tendrían un coste temporal lineal. La solución obvia consiste en no tener un apuntador al elemento actual, sino al anterior al actual y, así, dado que el anterior permite acceder al actual siguiendo un único encadenamiento, tanto la inserción como la supresión tendrán un coste constante. De todos modos, esta modificación introduce un nuevo problema: ¿cuál es el elemento anterior al primero? Este caso especial obliga a escribir unos algoritmos más complicados, que actúan dependiendo de si la lista está o no vacía, de si el elemento a suprimir es o no es el primero, etc. Para evitar estos problemas, existe una técnica sencilla pero de gran utilidad, que consiste en la inclusión dentro de la lista de un *elemento fantasma* o *centinela* (ing., *sentinel*), es decir, un elemento ficticio que siempre está a la izquierda de la lista, desde que se crea, de manera que el elemento actual siempre tiene un predecesor⁵. Dentro del vector, este elemento ocupará la posición 0, y como los algoritmos no lo moverán nunca de sitio, deja de ser necesario el apuntador al primer elemento.

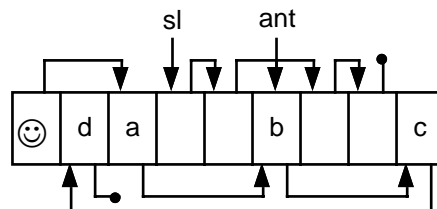


Fig. 3.19: Ídem fig. 3.18, con un elemento fantasma y apuntador al anterior.

En la fig. 3.21 se da un universo de implementación de las listas encadenadas con elemento fantasma y apuntador al anterior del actual, que deja todas las operaciones de coste constante en todos los casos, excepto *crea* que queda $\Theta(val)$. En la fig. 3.20 se muestra la mecánica de inserción y supresión de elementos en listas encadenadas (para que los

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

esquemas sean más claros, no se dibujan los vectores ni el elemento fantasma, sino directamente la secuencia de elementos y la pila de sitios libres). Notemos que el invariante de la representación es más complejo que los que hemos visto hasta ahora, porque es necesario establecer ciertas propiedades que cumplan los encadenamientos. En concreto, en la primera línea se dan los valores permitidos para *ant* y *sl*, y en la segunda y tercera líneas se establece que toda posición del vector forma parte, o bien sólo de la lista de elementos, o bien sólo de la pila de sitios libres. Para escribir este predicado utilizamos una función auxiliar definida recursivamente, *cadena*, que devuelve el conjunto de posiciones que cuelga de una posición determinada del vector siguiendo los encadenamientos; así, *cadena(l, 0)* devuelve el conjunto de posiciones ocupadas dentro de *l*, y *cadena(l, l.sl)* el conjunto de posiciones que forman la pila de sitios libres dentro de *l*. La mayoría de estas propiedades del invariante se repetirán en las representaciones encadenadas que estudiaremos a lo largo del texto, así como la función *cadena*, en esta forma o con pequeñas variaciones.

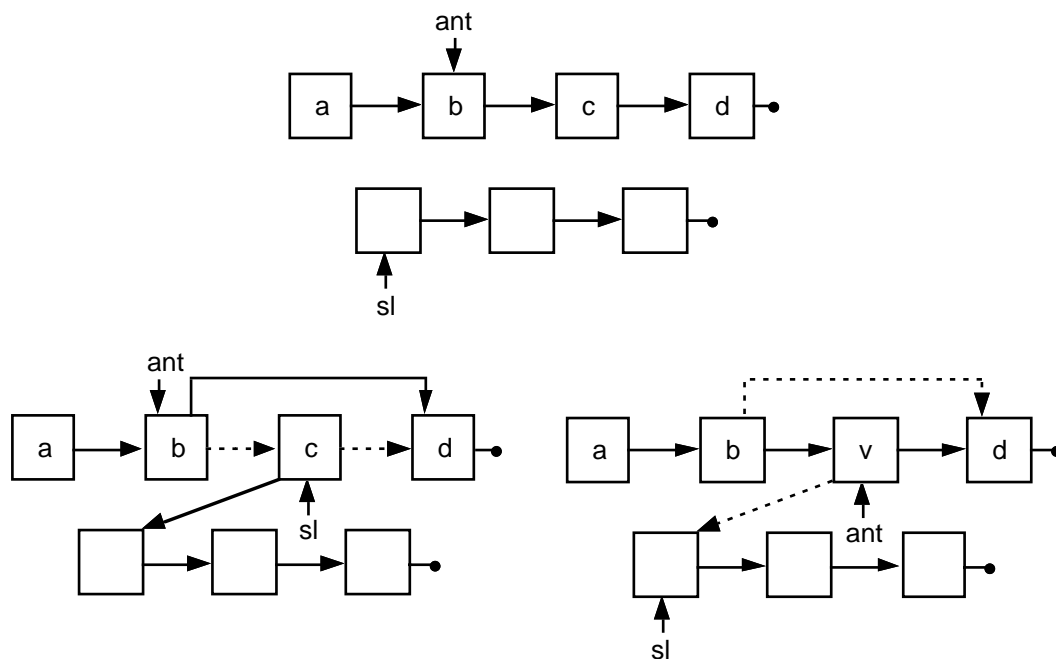


Fig. 3.20: representación encadenada de la lista $\langle ab, cd \rangle$ (arriba), supresión del elemento distinguido (abajo, a la izquierda) e inserción del elemento v (abajo, a la derecha).

universo LISTA_INTERÉS_ENC(ELEM, VAL_NAT) es
implementa LISTA_INTERÉS(ELEM, VAL_NAT)
tipo lista es tupla
 A es vector [de 0 a val] de
 tupla v es elem; enc es entero ftupla
 ant, sl son entero
 ftupla
ftipo
invariante (l es lista): $(1 \leq l.sl \leq val \vee l.sl = -1) \wedge l.ant \in \text{cadena}(l, 0) - \{-1\} \wedge$
 $\text{cadena}(l, 0) \cap \text{cadena}(l, l.sl) = \{-1\} \wedge$
 $\text{cadena}(l, 0) \cup \text{cadena}(l, l.sl) = [-1, val]$
 donde se define *cadena*: $\text{lista nat} \rightarrow \mathcal{P}(\text{nat})$ como:
 $\text{cadena}(l, -1) = \{-1\}$
 $n \neq -1 \Rightarrow \text{cadena}(l, n) = \{n\} \cup \text{cadena}(l, l.A[n].enc)$
 {En todas las operaciones, debe recordarse que el punto de
 interés es *l.A[l.ant].enc* y que el fantasma reside en la posición 0}
función crea devuelve lista es
var l es lista; i es entero fvar
 l.ant := 0; l.A[0].enc := -1 {primero, se inserta el elemento fantasma}
 {después, se forma la pila de sitios libres}
 para todo i desde 1 hasta val-1 hacer l.A[i].enc := i+1 fpara todo
 l.A[val].enc := -1; l.sl := 1
devuelve l
función inserta (l es lista; v es elem) devuelve lista es
var temp es entero fvar
 si l.sl = -1 entonces error {lista llena}
 si no {v. fig. 3.21, abajo, derecha}
 temp := l.A[l.sl].enc; l.A[l.sl] := <v, l.A[l.ant].enc>
 l.A[l.ant].enc := l.sl; l.ant := l.sl; l.sl := temp
 fsi
devuelve l
función borra (l es lista) devuelve lista es
 si l.A[l.ant].enc = -1 entonces error {lista vacía o final de lista}
 si no {v. fig. 3.21, abajo, izquierda}
 temp := l.A[l.ant].enc; l.A[l.ant].enc := l.A[temp].enc
 l.A[temp].enc := l.sl; l.sl := temp
 fsi
devuelve l

Fig. 3.21: implementación encadenada de las listas.

<u>función principio</u> (l <u>es</u> lista) <u>devuelve</u> lista <u>es</u> l.ant := 0 <u>devuelve</u> l	<u>función avanza</u> (l <u>es</u> lista) <u>devuelve</u> lista <u>es</u> <u>si</u> l.A[l.ant].enc = -1 <u>entonces</u> error <u>si no</u> l.ant := l.A[l.ant].enc <u>fsi</u> <u>devuelve</u> l
<u>función actual</u> (l <u>es</u> lista) <u>devuelve</u> elem <u>es</u> var v <u>es</u> elem <u>fvar</u> <u>si</u> l.A[l.ant].enc = -1 <u>entonces</u> error <u>si no</u> v := l.A[l.A[l.ant].enc].v <u>fsi</u> <u>devuelve</u> v	<u>función final?</u> (l <u>es</u> lista) <u>devuelve</u> bool <u>es</u> <u>devuelve</u> l.A[l.ant].enc = -1
<u>función vacía?</u> (l <u>es</u> lista) <u>devuelve</u> bool <u>es</u> <u>devuelve</u> l.A[0].enc = -1	<u>función llena?</u> (l <u>es</u> lista) <u>devuelve</u> bool <u>es</u> <u>devuelve</u> l.sl = -1

funiverso

Fig. 3.21: implementación encadenada de las listas (cont.).

3.3.3 Implementación de estructuras de datos con punteros

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

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

Los lenguajes imperativos comerciales presentan un mecanismo incorporado de gestión de espacio para liberar al programador de estas limitaciones, que se concreta en la existencia de un nuevo tipo de datos (en realidad, un constructor de tipo) llamado *puntero* (ing., *pointer*): una variable de tipo *puntero a T*, donde *T* es un tipo de datos cualquiera, representa un apuntador a un objeto de tipo *T*. En términos de implementación, el espacio libre es gestionado por el sistema operativo, que responde a las peticiones del programa, y un

puntero no es más que una dirección de memoria que representa la posición de inicio del objeto de tipo T , con el añadido de un mecanismo de control de tipo para impedir su uso indebido. El espacio así gestionado se denomina *memoria dinámica* (ing., *dynamic storage*).

Usando punteros, se puede obtener espacio para guardar objetos de un tipo determinado y posteriormente devolver este espacio cuando el programa ya no necesita más el objeto. Cada lenguaje de programación imperativo implementa este esquema de manera diferente. En este libro, definimos el constructor de tipos "puntero" de manera similar a como lo hacen Pascal o Modula; la política de estos lenguajes es más segura y cómoda que la que utiliza C, por ejemplo. Como resultado, en Merlí definimos dos primitivas sobre el constructor de tipo:

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

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

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

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

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

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

La codificación con punteros del tipo *lista* muestra las diversas ventajas de la representación por punteros sobre los vectores: no es necesario predeterminedar un número máximo de elementos en la estructura, no es necesario gestionar el espacio libre y, en cada momento, el espacio ocupado por la estructura es estrictamente el necesario exceptuando el espacio requerido por los encadenamientos. Destacamos asimismo el coste temporal constante de las operaciones en todos los casos. De todas formas, el uso de punteros también presenta algunas desventajas:

universo LISTA_INTERÉS_ENC_PUNT(ELEM) implementa LISTA_INTERÉS(ELEM)

tipo lista es tupla prim, ant son ^nodo ftupla ftipo

tipo privado nodo es tupla v es elem; enc es ^nodo ftupla ftipo

invariante (l es lista): l.prim \neq NULO \wedge NULO \in cadena(l.prim) \wedge l.ant \in cadena(l.prim) - {NULO}
donde se define cadena: ^nodo $\rightarrow \mathcal{P}(\text{^nodo})$ como:
cadena(NULO) = {NULO}
 $p \neq \text{NULO} \Rightarrow \text{cadena}(p) = \{p\} \cup \text{cadena}(p^{\text{enc}})$

función crea devuelve lista es

var l es lista fvar
l.prim := obtener_espacio {para el elemento fantasma}
si l.prim = NULO entonces error {no hay espacio libre}
si no l.ant := l.prim; l.ant^.enc := NULO
fsi
devuelve l

función inserta (l es lista; v es elem) devuelve lista es

var p es ^nodo fvar
p := obtener_espacio
si p = NULO entonces error {no hay espacio libre}
si no p^.v := v; p^.enc := l.ant^.enc; l.ant^.enc := p; l.ant := p
fsi
devuelve l

función borra (l es lista) devuelve lista es

var temp es ^nodo fvar
si l.ant^.enc = NULO entonces error {lista vacía o final de lista}
si no temp := l.ant^.enc; l.ant^.enc := temp^.enc; liberar_espacio(temp)
fsi
devuelve l

función principio (l es lista) devuelve lista es
devuelve <l.prim, l.prim>

función actual (l es lista) dev elem es

si l.ant^.enc = NULO entonces error
si no v := l.ant^.enc^.v
fsi
devuelve v

función avanza (l es lista) devuelve lista es

si l.ant^.enc = NULO entonces error
si no l.ant := l.ant^.enc
fsi
devuelve l

función final? (l es lista) devuelve bool es
devuelve l.ant^.enc = NULO

función vacía? (l es lista) devuelve bool es
devuelve l.prim^.enc = NULO

funiverso

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

- No es verdad que la memoria sea infinita: en cualquier momento durante la ejecución del programa se puede agotar el espacio, con el agravante de no conocer *a priori* la capacidad máxima de la estructura.
- Los punteros son referencias directas a memoria, por lo que es posible que se modifiquen insospechadamente datos (e incluso código, si el sistema operativo no lo controla) en otros puntos del programa a causa de errores algorítmicos.
- La depuración del programa es más difícil, porque puede ser necesario estudiar la ocupación de la memoria. Y a tal efecto, debe considerarse la memoria dinámica como una especie de variable global accesible desde cualquier acción o función del programa, sin que se declare como parámetro, a diferencia de cualquier otro dato manipulado por el programa.
- Que el sistema operativo gestione los sitios libres es cómodo, pero a veces ineficiente (por ejemplo, v. ejercicio 3.17).
- Algunos esquemas típicos de los lenguajes imperativos (por ejemplo, el mecanismo de parámetros, la asignación y la entrada/salida) no funcionan de la manera esperada: dados los punteros p y q a objetos de tipo T , dado el fichero f de registros de tipo T y dada la función *llamada* con un parámetro de entrada de tipo T , destacamos:
 - ◊ *escribe*(f, p) no escribe el objeto designado por p , sino el valor del puntero, que es una dirección de memoria, y es necesario invocar *escribe*(f, p^{\wedge}). Ahora bien, esta llamada también grabará en el fichero aquellos campos de T que sean punteros (si los hay), que serán direcciones de memoria particulares de la ejecución actual del programa. Si en otra ejecución posterior se leen los datos del fichero, los punteros obtenidos no tendrán ningún valor y se habrá perdido la estructuración lógica de los datos.
 - ◊ $p := q$ no asigna el valor del objeto apuntado por q al objeto apuntado por p , sino que asigna la dirección q a p .
 - ◊ *llamada*(p) protege el valor del puntero p , pero no el valor del objeto apuntado por p , que puede modificarse aunque el parámetro se haya declarado sólo de entrada.
- Diversos punteros pueden designar un mismo objeto, en cuyo caso, la modificación del objeto usando un puntero determinado tiene como efecto lateral la modificación del objeto apuntado por el resto de punteros.
- Una particularización del problema anterior son las llamadas *referencias colgadas* (ing., *dangling reference*). Por ejemplo, dado el trozo de código:

```

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

```

se dice que q queda "colgado", en el sentido de que su valor es diferente de NULO, pero no apunta a ningún objeto válido. Una referencia a q^{\wedge} tiene efectos impredecibles.

- El problema simétrico al anterior es la creación de *basura* (ing., *garbage*); dado:

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

observemos que el objeto inicialmente asociado a p queda inaccesible después de la segunda asignación, a pesar de que existe porque no ha sido explícitamente destruido. Eventualmente, esta basura puede provocar errores de ejecución o, como mínimo, dificultar la depuración del programa. Muchos sistemas operativos tienen incorporado un proceso de recuperación de estos objetos inaccesibles llamado *recogida de basura* (ing., *garbage collection*), cuya programación ha dado lugar a diversos algoritmos clásicos dentro del ámbito de las estructuras de datos (v., por ejemplo, [AHU83, cap. 12])⁶.

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

3.3.4 Transparencia de la representación usando punteros

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

a) El problema de la asignación

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

Para solucionar este problema, todo TAD T ha de ofrecer una operación *duplica* que, dado un objeto de tipo T , simplemente hace una réplica exacta del mismo. Entonces, un usuario

⁶ En algunos lenguajes de programación orientados a objetos, la existencia del recolector de basura viene dada por la definición del propio lenguaje, como es el caso de Java o Eiffel. En estos lenguajes, pues, no es necesario disponer de una primitiva para liberar espacio.

del TAD empleará *duplica* para copiar un objeto en lugar de la asignación estándar. Eso sí, dentro del TAD que implementa T se puede usar la asignación normal con su comportamiento tradicional, pues la representación es accesible.

b) El problema de la comparación

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

Así, todo TAD T ha de ofrecer una operación *ig* que, dados dos objetos de tipo T , los compare (es decir, compruebe si están en la misma clase de equivalencia según la relación de igualdad del tipo)⁷. Como en el caso de la asignación, dentro del TAD que implementa T sí se puede usar la comparación normal con su comportamiento tradicional.

c) El problema de los parámetros de entrada

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

```

función concatena ( $l_1, l_2$  son lista) devuelve lista es
  mientras  $\neg$  final?( $l_1$ ) hacer  $l_1 := avanza(l_1)$  fmientras
   $l_2 := principio(l_2)$ 
  mientras  $\neg$  final?( $l_2$ ) hacer
     $l_1 := inserta(l_1, actual(l_2)); l_2 := avanza(l_2)$ 
  fmientras
devuelve  $l_1$ 

```

Fig. 3.24: una función que concatena dos listas.

⁷ Eventualmente, en los TAD recorribles puede decidirse que el estado del recorrido no afecta el resultado de la comparación, aunque en este caso la implementación no sería correcta respecto de la especificación usando las técnicas de verificación introducidas en la sección 2.2.

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

En la fig. 3.25 se presenta la adaptación de la función *concatena* a estas normas. Notemos que la función pasa a ser una acción donde se declaran dos parámetros: el primero, de entrada y de salida, almacena el resultado, y el segundo, sólo de entrada, que se duplica al empezar la acción porque se modifica (aunque en este ejemplo concreto no era necesario, dado que los elementos de l_2 no cambian, excepto el punto de interés, que sí habrá sido duplicado por el mecanismo de llamada a procedimientos propio de los lenguajes de programación imperativos). Opcionalmente, se podría haber respetado la estructura de función, y habría sido necesario declarar una lista auxiliar para generar el resultado.

```

acción concatena (ent/sal  $l_1$  es lista; ent  $l_2$  es lista) es
var laux es lista fvar
    duplica( $l_2$ , laux); laux := principio(laux)
    mientras  $\neg$  final?( $l_1$ ) hacer  $l_1$  := avanza( $l_1$ ) fmientras
    mientras  $\neg$  final?(laux) hacer
         $l_1$  := inserta( $l_1$ , actual(laux)); laux := avanza(laux)
    fmientras
facción

```

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

d) El problema de las variables auxiliares

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

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

e) El problema de la reinicialización

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

```

acción concatena (ent/sal  $l_1$  es lista; ent  $l_2$  es lista) es
var laux es lista fvar
    duplica( $l_2$ , laux); laux := principio(laux)
    mientras  $\neg$  final?( $l_1$ ) hacer  $l_1$  := avanza( $l_1$ ) fmientras
    mientras  $\neg$  final?(laux) hacer
         $l_1$  := inserta( $l_1$ , actual(laux)); laux := avanza(laux)
    fmientras
    destruye(laux)
facción

```

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

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

Resumiendo, en el resto del texto convendremos que todo tipo abstracto presenta, no sólo las operaciones que específicamente aparecen en la signatura, sino también las introducidas en este apartado (orientadas a la transparencia total de la información), y que las funciones y los algoritmos que utilizan los tipos siguen las normas aquí expuestas para asegurar el

funcionamiento correcto en todos los contextos conflictivos citados en este apartado. Como ejemplo ilustrativo, en la figura 3.27 se presentan las dos representaciones encadenadas de las listas con punto de interés, una con vectores y la otra con punteros, que siguen todas las convenciones dadas. Destacamos:

- Las operaciones de inserción y borrado se ven modificadas por este esquema; en la inserción, debe duplicarse el elemento almacenado en el vector mediante *duplica*, mientras que en la segunda debe destruirse el espacio ocupado usando *destruye*; al haber acordado que todo TAD presenta estas operaciones, tomamos como convención que no es necesario requerirlas como parámetros formales. Igualmente, la consulta del elemento actual debe duplicar el valor obtenido. El algoritmo de *duplica* asegura no sólo que los elementos son los mismos sino además que su posición se conserva, así como la situación del punto de interés.
- La operación de comparación también usa la igualdad de elementos, definida en cada una de las implementaciones del tipo; notemos que los elementos fantasma de las listas involucradas no son comparados y que la posición ocupada por el punto de interés es significativa de cara al resultado.
- La destrucción de la lista obliga a destruir el espacio individual que ocupa cada elemento y, además, en el caso de implementación por punteros, el espacio de cada celda de la lista.
- Presentamos las dos políticas existentes para la creación de las listas. En la representación por vectores, el usuario es el responsable de destruir la estructura antes de recrearla, si lo considera necesario. En la representación por punteros, la destrucción se realiza al comenzar la creación, incondicionalmente, de forma transparente al usuario del tipo. Estas dos políticas pueden utilizarse indistintamente en cualquier implementación, y simplemente deben tenerse en cuenta las consecuencias que para el usuario tienen cada una de ellas.

universo LISTA_INTERÉS_ENC(ELEM, VAL_NAT) es
implementa LISTA_INTERÉS(ELEM, VAL_NAT)

... la representación del tipo no cambia

acción crea (sal | es lista) es

var i es entero fvar

i.ant := 0; i.A[0].enc := -1

para todo i desde 1 hasta val-1 hacer i.A[i].enc := i+1 fpara todo

i.A[val].enc := -1; i.sl := 1

facción

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

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

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

función actual (l₁ es lista) devuelve elem es
var v es elem fvar
 si l.A[l.ant].enc = -1 entonces error
 si no duplica(l.A[l.A[l.ant].enc].v, v)
 fsi
devuelve v

función *final?*: idéntica a la fig. 3.20

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

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

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

```

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

```

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

```

universo LISTA_INTERÉS_ENC_PUNT(ELEM) implementa LISTA_INTERÉS(ELEM)
... la representación del tipo no cambia
acción crea (sal l es lista) es
    {obtiene espacio para el fantasma, liberando previamente si es necesario, en
     cuyo caso se liberan todas las celdas menos una, precisamente para el fantasma}
    si l.prim ≠ NULO entonces libera(l.prim^.enc) si no l.prim := obtener_espacio fsi
    si l.prim = NULO entonces error si no l.prim^.enc := NULO; l.ant := l.prim fsi
devuelve l

```

Acciones *inserta*, *borra*, *principio* y *avanza*: siguen los algoritmos de la fig. 3.23, pero adaptando las funciones a acciones de manera similar a la función *inserta* de la fig. 3.27(a). La función *actual* debe duplicar el resultado al igual que en la fig. 3.27(a). La función *final?* es idéntica a la de la fig. 3.23.

Función *ig* y acción *duplica*: idénticas a la operaciones *ig* y *duplica* de la fig. 3.27(a), adaptando la notación de vectores a punteros

```

acción destruye (ent/sal l es lista) es libera(l.prim) facción

```

{Acción *libera(p)*: devuelve al sistema las celdas que cuelgan a partir del puntero *p* siguiendo el campo de encadenamiento (v. ejercicio 3.17 para una variante)}

```

acción privada libera (ent/sal p es ^nodo) es
var q es ^nodo fvar
    mientras p ≠ NULO hacer
        ELEM.destruye(p^.v)    { se libera el espacio del elemento }
        q := p; p := p^.enc
        liberar_espacio(q)     { se libera el espacio de la celda de la lista }
    fmientras
facción
funiverso

```

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

Para evitar una sobrecarga del código de los TAD con estas tareas mecánicas, en el resto del texto se toman una serie de convenciones que las llevan a término implícitamente⁸. En concreto, dado un tipo abstracto T ⁹ se cumple:

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

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

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

⁸ Obviamente, sería mejor disponer de un lenguaje de programación que no presentara estos problemas, pero como éste no es el caso habitual (en el paradigma imperativo, se entiende), se ha optado por dar aquí unas convenciones fácilmente aplicables a los lenguajes más comunes y que, además, encaja bastante bien con la política seguida por la mayoría de lenguajes orientados a objetos.

⁹ Exceptuando los tipos predefinidos del lenguaje (booleanos, naturales, etc.).

3.3.5 Implementaciones encadenadas y TAD abiertos

En los TAD abiertos, las implementaciones vistas hasta ahora pueden presentar ineficiencias, sobre todo con respecto a su coste temporal. Introducimos dos variantes que permiten resolver algunas situaciones habituales.

a) Estructuras encadenadas circulares

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

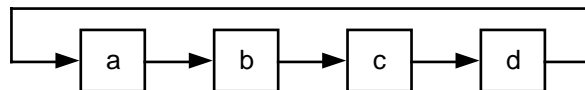


Fig. 3.28: estructura encadenada circular.

El concepto de circularidad es útil especialmente en los TAD abiertos. En una estructura encadenada circular, partiendo de un elemento accedido mediante su atajo asociado puede accederse a cualquier otro de la estructura, lo que es especialmente útil si hay un elemento de la cadena con información diferenciada (típicamente el primero; v. la implementación de relaciones por multilistas en la sección 6.1).

Como curiosidad, destacaremos que la representación encadenada circular permite que la representación de colas con encadenamientos sólo necesite un apuntador al último elemento, porque el primero siempre será el apuntado por el último (v. fig. 3.29). Puede añadirse también un elemento fantasma para simplificar los algoritmos.

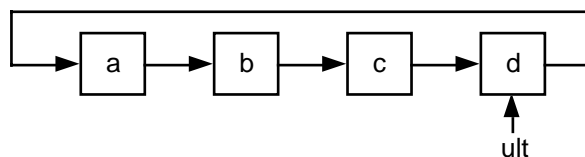


Fig. 3.29: representación encadenada circular de la cola abcd, donde a es la cabeza.

b) Estructuras doblemente encadenadas

Las estructuras encadenadas en un único sentido exhiben como propiedad característica que, dado un elemento, la única manera de saber cuál es el anterior consiste en recorrer la estructura desde el inicio y el coste resultante será lineal. Hay dos situaciones donde es imprescindible saber qué elemento es el anterior a uno dado:

- Cuando el TAD ha de ofrecer recorridos tanto hacia adelante como hacia atrás. Notemos que este modelo exige tener más operaciones en la signatura simétricas a las habituales en los TAD recorribles: una para situarse al final, otra para retroceder un elemento y una última para saber si ya estamos al principio.
- Cuando el TAD es totalmente abierto y sus elementos se pueden suprimir mediante atajos, sin conocer la posición que ocupa su predecesor. La única manera de asegurar coste constante en esta supresión es tener un apuntador al elemento anterior para poder actualizar el encadenamiento de éste.

Si el coste lineal de estas operaciones es inaceptable, es necesario modificar la representación para obtener estructuras doblemente encadenadas, que en el caso de las listas se denominan *listas doblemente encadenadas* (ing., *doubly-linked list*). Así, las operaciones vuelven a ser constantes, a costa de utilizar más espacio.

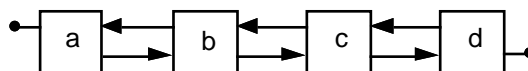


Fig. 3.30: estructura doblemente encadenada.

Una vez más, para simplificar los algoritmos de inserción y de supresión (v. fig. 3.31) se añade al principio un elemento fantasma, de modo que la estructura no esté nunca vacía; además, es aconsejable implementar la estructura circularmente.

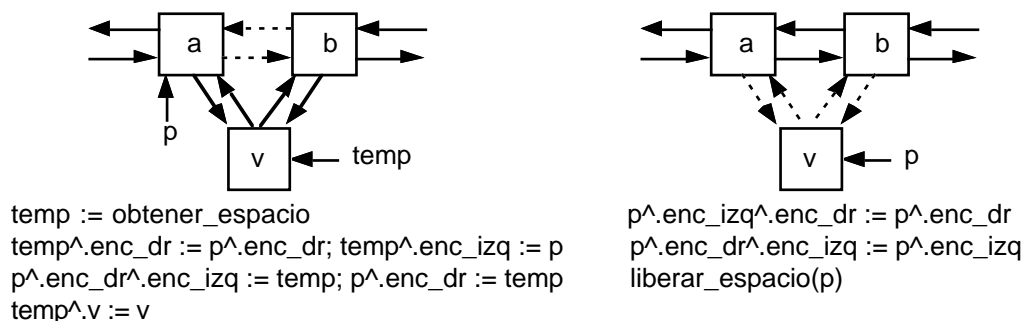


Fig. 3.31: inserción (izq.) y supresión (der.) en una estructura doblemente encadenada.

3.3.6 Implementaciones encadenadas y TAD recorribles ordenadamente

En el apartado 2.4.2 hemos comprobado que los recorridos ordenados con representación secuencial pueden provocar problemas de eficiencia en las operaciones de actualización, así como movimientos de elementos, incompatibles con las versiones abiertas de los TAD. Estudiamos a continuación la adecuación de las estructuras encadenadas a este contexto.

Consideremos el caso habitual de disponer de un único criterio de ordenación. Las dos estrategias vistas para el caso de los conjuntos en el apartado 2.4.2 son aplicables al caso de estructuras encadenadas con los efectos siguientes:

- Se puede tener la estructura desordenada mientras se insertan y se borran elementos y ordenarla justo antes de empezar el recorrido. La ventaja de este esquema es que las inserciones y supresiones no varían de coste (por ejemplo, en el caso de las listas con punto de interés se mantienen en orden constante), mientras que la única operación de recorrido que tiene una eficiencia baja es *principio*, que ha de ordenar los elementos de la estructura. Con un buen algoritmo de ordenación, esta ordenación quedaría $\Theta(n \log n)$, donde n es el número de elementos de la estructura, aunque también puede exigir espacio auxiliar $\Theta(n)$. La ordenación se basará en el orden de los encadenamientos por lo que, a diferencia de la representación secuencial, no habrá movimientos de elementos. El recorrido queda pues $\Theta(n \log n)$.
- Una alternativa es mantener la estructura siempre ordenada. Esto implica que, para cada actualización, es necesario localizar la posición donde insertar ordenadamente, o el elemento a borrar, y entonces el coste $\Theta(n)$ en el caso peor es ineludible, independientemente de la eficiencia de partida de estas operaciones de actualización y aunque no haya movimientos de elementos. Sin embargo, las operaciones de recorrido quedan de orden constante, resultando pues el recorrido de orden $\Theta(n)$ en todos los casos.

La elección de una alternativa concreta dependerá de la relación esperada entre modificaciones y recorridos ordenados de la estructura. Por ejemplo, para poder aplicar el primer esquema es aconsejable que la gestión de la lista presente dos fases diferenciadas, una primera de actualizaciones y una segunda de consultas.

La representación encadenada permite implementar TAD recorribles ordenadamente de manera eficiente por más de un concepto, empleando tantos encadenamientos como campos de ordenación haya. Hemos visto un caso particular en el apartado anterior, con las lista doblemente encadenadas que permiten recorrer la estructura en los dos sentidos. Notemos que, con una representación secuencial, los recorridos ordenados eficientes (es decir, de orden lineal sobre el número de elementos) por más de un concepto no son posibles pues la ordenación física en el vector tan solo permitiría reflejar una de las diversas ordenaciones.

Ejercicios

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

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

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

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

b) Repetir el apartado a) enriqueciendo las pilas recorribles.

c) Dado el universo $PILA(ELEM)$ que encapsula la especificación habitual de las pilas, modificarlo para que incluya todas las operaciones citadas en el apartado a) (no es necesario volverlas a especificar). A continuación, modificar el universo que implementa $PILA(ELEM)$ para que también se implementen estas nuevas operaciones.

d) Comparar los resultados de los tres apartados anteriores en cuanto a los criterios de corrección, eficiencia, modificabilidad y reusabilidad de los programas.

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

3.5 Una *doble cola* (ing., abreviadamente, *deque*) es una cola en la cual los elementos se pueden insertar, eliminar y consultar en ambos extremos. Determinar la signatura de este tipo de datos y especificarlo dentro de un universo parametrizado. Implementar el tipo según una

estrategia secuencial con gestión circular del vector.

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

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

$$Z = Y[1].Y[n].Y[2].Y[n-1]..., \text{ donde } Y[i] \text{ representa el carácter } i\text{-ésimo de } Y.$$

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

3.7 Modificar la representación encadenada sobre el vector de las listas con punto de interés para que la operación de crear la lista tenga coste constante sin cambiar el coste de las demás. Razonar si esta modificación afecta al coste total de un algoritmo que inserte n elementos en la lista después de crearla.

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

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

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

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

3.12 a) Escribir en un universo parametrizado *LISTA_2* un enriquecimiento de las listas con punto de interés consistente en las operaciones: *contar* el número de elementos de una lista, *eliminar* todas las apariciones de un elemento dado, *invertir* una lista, *intercambiar* el elemento de interés con su sucesor y *concatenar* dos listas. En la operación *invertir*, el

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

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

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

crea: $\rightarrow lista$, devuelve la lista vacía

inserta: $lista \text{ nat elem} \rightarrow lista$, inserta delante del elemento *i*-ésimo o bien a la derecha del todo si se especifica como parámetro la longitud de la lista más uno; la numeración de los elementos cambia convenientemente

borra: $lista \text{ nat} \rightarrow lista$, borra el elemento *i*-ésimo; la numeración de los elementos cambia convenientemente

modifica: $lista \text{ nat elem} \rightarrow lista$, sustituye el elemento *i*-ésimo por el valor dado

consulta: $lista \text{ nat} \rightarrow lista$, consulta el elemento *i*-ésimo

longitud: $lista \rightarrow nat$, devuelve el número de elementos de la lista

Especificar este TAD controlando las posibles referencias a posiciones inexistentes. A continuación determinar su implementación. ¿Es necesario en este nuevo tipo de lista disponer de versiones recorribles? ¿Y abiertas?

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

3.15 Especificar el TAD *colapr* de las *colas con prioridad* (ing., *priority queue*; se estudian detalladamente en el capítulo 4) en las que el primer elemento es siempre el que tiene una prioridad más alta independientemente de cuándo se insertó; dos elementos con la misma prioridad conservan el orden de inserción dentro de la cola. Como operaciones se pueden considerar la cola *vacía*, *insertar* un elemento en la cola, obtener el *menor* elemento (que es el que tiene la prioridad más alta), *desencolar* el menor elemento y averiguar si la cola está

vacía?. Determinar las posibles implementaciones para el tipo.

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

vacío: \rightarrow *montón*, crea el montón sin cadenas

inserta: *montón cadena* \rightarrow \langle *montón*, *nat* \rangle , añade una cadena nueva a la estructura y devuelve, además, el identificador con el que se puede referir la cadena

borra: *montón nat* \rightarrow *montón*, borra la cadena identificada por el natural; esta función no ha de intentar comprimir el espacio del montón

consulta: *montón nat* \rightarrow *cadena*, obtiene la cadena identificada por el natural

listar: *montón* \rightarrow *lista_cadenas*, lista todas las cadenas del montón en orden alfabético

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

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

3.18 a) Sea un TAD para las relaciones de equivalencia sobre elementos cualesquiera (que se estudia en el capítulo 6) con operaciones: relación *vacía*, *añadir* una nueva clase con un único elemento, *fusionar* dos clases identificadas a partir de dos elementos que pertenecen a ellas, averiguar *cuántas* clases hay en la relación y si dos elementos dados son *congruentes?* (es decir, si están dentro de la misma clase). Determinar la signatura del tipo, especificarlo e implementarlo según dos estrategias: usando la especificación de alguna estructura lineal y representando el tipo directamente con los constructores del lenguaje.

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

3.19 Considerar un TAD para las matrices de enteros cuadradas dispersas (es decir, con la mayoría de elementos a cero) y con operaciones: matriz *cero*, *definir* el valor en una posición dada, *obtener* el valor de una posición dada, *sumar* y *multiplicar* dos matrices y calcular la matriz *traspuesta*. Implementar el tipo para todas las estrategias propuestas a continuación:

a) Implementación con un vector bidimensional. **b)** Implementación con una lista de 3-tuplas $\langle \text{fila}, \text{columna}, \text{valor} \rangle$ donde sólo aparecen las posiciones no nulas. Discutir la conveniencia de ordenar la lista siguiendo algún criterio. **c)** Considerar el caso de matrices triangulares inferiores (todos los elementos por encima de la diagonal principal son nulos). En este caso, el número de elementos del triángulo es fácilmente calculable. Implementar el tipo sin operación *traspuesta* usando un vector unidimensional. **d)** Repetir el apartado b) considerando matrices tridiagonales (todos los elementos son nulos excepto los de la diagonal principal y sus dos diagonales adyacentes), pero sin *multiplicar* y con *traspuesta*.

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

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

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

Capítulo 4 Árboles

Los *árboles* (ing., *tree*) son objetos que organizan sus elementos, denominados *nodos*, formando jerarquías, tal y como se refleja en su representación gráfica (v. fig. 4.1). Su definición inductiva es:

- Un único nodo es un árbol; a este nodo se le llama *raíz* del árbol.
- Dados n árboles a_1, \dots, a_n , se puede construir uno nuevo como resultado de *enraizar* un nuevo nodo con los n árboles a_i . Generalmente, el orden de los a_i es significativo y así lo consideramos en el resto del capítulo; algunos autores lo subrayan llamándolos *árboles ordenados* (ing., *ordered trees*). Los árboles a_i pasan a ser *subárboles* del nuevo árbol y el nuevo nodo se convierte en raíz del nuevo árbol.

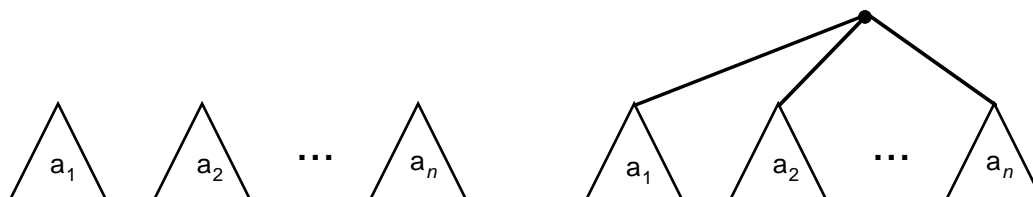


Fig. 4.1: n árboles (a la izquierda) y su enraizamiento formando uno nuevo (a la derecha).

El concepto matemático de árbol data de mediados del siglo XIX y su uso en el mundo de la programación es habitual. Ya ha sido introducido en la sección 1.2 como una representación intuitiva de la idea de término; en general, los árboles permiten almacenar cómodamente cualquier tipo de expresión dado que la parentización, la asociatividad y la prioridad de los operadores quedan implícitos, y la evaluación de la expresión consiste en la simple aplicación de una estrategia de recorrido. Suelen utilizarse también para representar jerarquías de todo tipo: taxonomías, relaciones entre módulos, etc. Igualmente es destacable su contribución a diversos campos de la informática: árboles de decisión en inteligencia artificial, representación de gramáticas y programas en el ámbito de la compilación, ayuda en técnicas de programación (transformación de programas recursivos en iterativos, etc.), índices en ficheros de acceso por clave, bases de datos jerárquicas, etc.

4.1 Modelo y especificación

En esta sección introducimos el modelo formal asociado a un árbol, que emplearemos para formular diversas definiciones útiles, y también su especificación ecuacional. Hablando con propiedad, hay varios modelos de árboles que surgen a partir de todas las combinaciones posibles de las características siguientes:

- El proceso de enraizar puede involucrar un número indeterminado de subárboles o bien un número fijo n . En el primer caso hablamos de *árboles generales* (ing., *general tree*) y en el segundo caso de *árboles n -arios* (ing., *n -ary tree*)¹; entre estos últimos destaca el caso de $n = 2$, los denominados *árboles binarios* (ing., *binary tree*), sobre el que nos centraremos a lo largo del texto dada su importancia, siendo inmediata la extensión a cualquier otro n . En los árboles n -arios se definirá el concepto de *árbol vacío* que ampliará el conjunto de valores del tipo.
- Los nodos del árbol pueden contener información o no. El primer caso es el habitual y responde al uso del árbol como almacén de información, pero también se puede dar la situación en que sólo interese reflejar la jerarquía, sin necesidad de guardar ningún dato adicional. En el resto del capítulo nos centraremos en el primer tipo de árbol, porque generaliza el segundo. La información en los nodos se denominará *etiqueta* (ing., *label*) y los árboles con información, *árboles etiquetados* (ing., *labelled tree*).
- La signatura definida en los árboles puede ser muy variada. Destacamos dos familias: en la primera se utilizan los árboles tal como se ha explicado en la introducción, con operaciones de enraizar diversos árboles, obtener un subárbol y, si el árbol es etiquetado, obtener la etiqueta de la raíz; la segunda familia define un nodo de referencia para las actualizaciones y consultas, que puede cambiarse con otras operaciones. Llamaremos al segundo tipo *árboles con punto de interés*, por su similitud respecto a las listas de las mismas características. En el primer tipo, existe también la opción de añadir iteradores para obtener los nodos en algún orden.

4.1.1 Modelo de árbol general

Un árbol general se caracteriza por su forma, que determina una relación jerárquica, y por el valor de la etiqueta asociada a cada uno de los nodos del árbol. La *forma* del árbol se puede describir como un conjunto de cadenas de los naturales sin el cero, $\mathcal{P}(\mathcal{N}_0^*)$, de manera que cada cadena del conjunto identifica un nodo del árbol según una numeración consecutiva de izquierda a derecha comenzando por el uno, tal como se muestra en la fig. 4.2. Hay que resaltar que este conjunto es *cerrado por prefijo* y *compacto*: cerrado por prefijo, porque si la cadena $\alpha.\beta$ está dentro del conjunto que denota la forma de un árbol, para $\alpha, \beta \in \mathcal{N}_0^*$, también lo está la cadena α ; y compacto, porque si la cadena $\alpha.k$ está dentro del conjunto,

¹ La nomenclatura no es estándar; es más, diferentes textos pueden dar los mismos nombres a diferentes modelos y contradecirse entre sí.

para $\alpha \in \mathcal{N}_0^*$, $k \in \mathcal{N}_0$, también lo están las cadenas $\alpha.1$, $\alpha.2$, ..., $\alpha.(k-1)$.

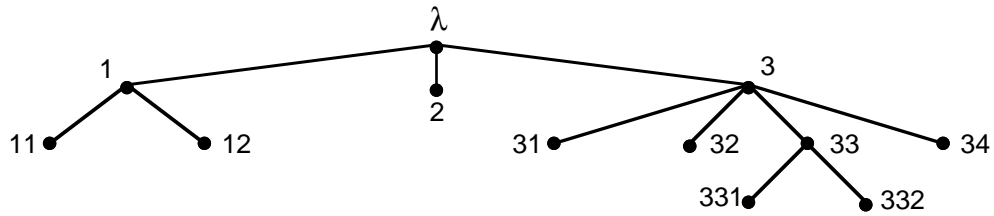


Fig. 4.2: representación del árbol de forma $\{\lambda, 1, 2, 3, 11, 12, 31, 32, 33, 34, 331, 332\}$.

Por lo que respecta a los valores de las etiquetas, se puede definir una función que asocie a cada cadena de la forma un valor del conjunto de base de las etiquetas. Evidentemente, esta función es parcial y su dominio es justamente el conjunto de cadenas que da la forma del árbol. En la fig. 4.3 se muestra gráficamente el árbol correspondiente a la función $\{\lambda \rightarrow \text{enero}, 1 \rightarrow \text{agosto}, 2 \rightarrow \text{febrero}, 3 \rightarrow \text{junio}, 11 \rightarrow \text{abril}, 12 \rightarrow \text{diciembre}, 31 \rightarrow \text{julio}, 32 \rightarrow \text{marzo}, 33 \rightarrow \text{noviembre}, 34 \rightarrow \text{setiembre}, 331 \rightarrow \text{mayo}, 332 \rightarrow \text{octubre}\}$.

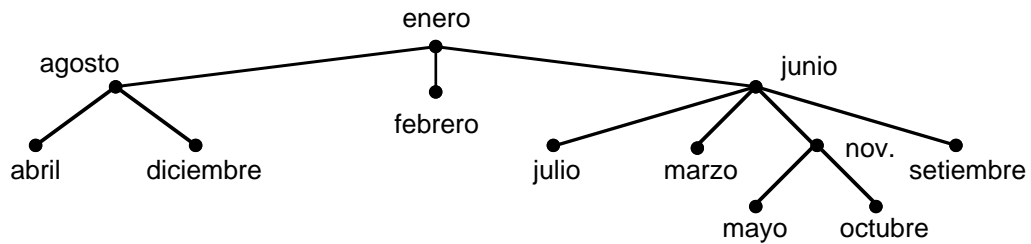


Fig. 4.3: el mismo árbol que en la fig. 4.2, asignando etiquetas a las cadenas de su forma.

Así, dado el conjunto base de las etiquetas V , definimos el modelo \mathcal{A}_V de los árboles generales y etiquetados como las funciones que tienen como dominio las cadenas de naturales \mathcal{N}_0^* y como codominio las etiquetas V , $f: \mathcal{N}_0^* \rightarrow V$, tales que su dominio $\text{dom}(f)$ cumple que no es vacío, es cerrado por prefijo y compacto. A $\text{dom}(f)$ le llamaremos la *forma* del árbol. A partir de este modelo se formulan una serie de definiciones útiles para el resto del capítulo, cuya especificación ecuacional queda como ejercicio para el lector.

a) Definiciones sobre los nodos

Sea $a \in \mathcal{A}_V$ un árbol general y etiquetado cualquiera.

- Un *nodo* del árbol (ing., *node*) es un componente determinado por la *posición* que

ocupa y por la etiqueta; es decir, $n = \langle s, v \rangle$ es un nodo de a si $s \in \text{dom}(a)$ y $a(s) = v$. En la fig. 4.3, $\langle 11, \text{abril} \rangle$ es un nodo pero no lo son ni $\langle 7, \text{julio} \rangle$ ni $\langle 11, \text{setiembre} \rangle$. El conjunto $N_a = \{n / n \text{ es un nodo de } a\}$ es el conjunto de todos los nodos del árbol a .

- La *raíz* del árbol (ing., *root*) es el nodo que está más arriba, $\langle \lambda, a(\lambda) \rangle$. Por ejemplo, la raíz del árbol de la fig. 4.3 es el nodo $\langle \lambda, \text{enero} \rangle$. Notemos que, a causa de las propiedades del modelo, todo árbol a presenta una raíz. A veces, por comodidad y cuando quede claro por el contexto, denominaremos raíz simplemente a la etiqueta del nodo raíz.
- Una *hoja* del árbol (ing., *leaf*) es un nodo que está abajo de todo; es decir, el nodo $n = \langle s, v \rangle \in N_a$ es hoja dentro de a si no existe $\alpha \in \mathcal{N}_0^+$ tal que $s, \alpha \in \text{dom}(a)$. En el árbol de la fig. 4.3, son hojas los nodos $\langle 2, \text{febrero} \rangle$ y $\langle 331, \text{mayo} \rangle$, pero no lo es $\langle 1, \text{agosto} \rangle$. El conjunto $\mathcal{F}_a = \{n / n \text{ es hoja dentro de } a\}$ es el conjunto de todas las hojas del árbol a .
- Un *camino* dentro del árbol (ing., *path*) es una secuencia de nodos conectados dentro del árbol; o sea, la secuencia $c = n_1 \dots n_r, \forall i: 1 \leq i \leq r: n_i = \langle s_i, v_i \rangle \in N_a$, es camino dentro de a si cada nodo cuelga del anterior, es decir, si $\forall i: 1 \leq i \leq r-1: (\exists k: k \in \mathcal{N}_0: s_{i+1} = s_i.k)$. La *longitud* del camino c (ing., *length*) es el número de nodos menos uno, $r-1$; si $r > 1$, se dice que el camino es *propio* (ing., *proper*). En el árbol de la fig. 4.3, la secuencia formada por los nodos $\langle \lambda, \text{enero} \rangle \langle 3, \text{julio} \rangle \langle 32, \text{marzo} \rangle$ es un camino propio de longitud 2. El conjunto $C_a = \{c / c \text{ es camino dentro de } a\}$ es el conjunto de todos los caminos del árbol a .
- La *altura* de un nodo del árbol (ing., *height*) es uno más que la longitud del camino desde el nodo a la hoja más lejana que sea alcanzable desde él²; es decir, dado el nodo $n \in N_a$ y el conjunto de caminos C_a^n que salen de un nodo n y van a parar a una hoja, $C_a^n = \{c = n_1 \dots n_k: c \in C_a: n_1 = n \wedge n_k \in \mathcal{F}_a\}$, definimos la altura de n dentro de a como la longitud más uno del camino más largo de C_a^n , $\text{altura}_a(n) = (\max c: c \in C_a^n: \|c\|) + 1$. En el árbol de la fig. 4.3, el nodo $\langle 3, \text{julio} \rangle$ tiene altura 3. La altura de un árbol a se define como la altura de su raíz, $\text{altura}(a) = \text{altura}(\langle \lambda, a(\lambda) \rangle)$; la altura del árbol de la fig. 4.3 es 4.
- El *nivel* o *profundidad* de un nodo del árbol (ing., *level* o *depth*) es el número de nodos que se encuentran entre él y la raíz; es decir, dado el nodo $n = \langle s, v \rangle \in N_a$ definimos el nivel de n dentro de a como $\text{nivel}_a(n) = \|s\| + 1$, siendo $\|s\|$ la longitud de s ; también se dice que n está en el nivel $\|s\| + 1$. En el árbol de la fig. 4.3, el nodo $\langle 3, \text{julio} \rangle$ está en el nivel 2. Por extensión, cuando se habla del nivel i de un árbol a (también primer nivel, segundo nivel, y así sucesivamente) se hace referencia al conjunto de nodos que están en el nivel i , $\text{nivel}_a^i = \{n: n \in N_a: \text{nivel}_a(n) = i\}$. El número de niveles de un árbol a se define como el nivel de la hoja más profunda, es decir,

² Las definiciones de altura, profundidad y nivel son también contradictorias en algunos textos. Hay autores que no definen alguno de estos conceptos, o los definen sólo para un árbol y no para sus nodos, o bien empiezan a numerar a partir del cero y no del uno, etc. En nuestro caso concreto, preferimos numerar a partir del uno para que el árbol vacío (que aparece en el caso n -ario) tenga una altura diferente a la altura del árbol con un único nodo.

$\text{nivel}(a) = \max n: n \in N_a: \text{nivel}_a(n)$; así, el número de niveles del árbol de la fig. 4.3 es 4. Notemos que, por definición, el número de niveles de un árbol es igual a su altura, por lo que pueden usarse ambas magnitudes indistintamente.

b) Definiciones sobre las relaciones entre nodos

A partir del concepto de camino, damos diversas definiciones para establecer relaciones entre nodos. Sean $a \in \mathcal{A}_V$ un árbol general y etiquetado y $n, n' \in N_a$ dos nodos del árbol, $n = \langle s, v \rangle$ y $n' = \langle s', v' \rangle$; si hay un camino $c = n_1 \dots n_r \in C_a$ tal que $n_1 = n$ y $n_r = n'$, decimos que:

- El nodo n' es *descendiente* (ing., *descendant*) de n y el nodo n es *antecesor* (ing., *ancestor*) de n' . Si, además, $r > 1$, los llamamos descendiente o antecesor *propio*.
- Si, además, $r = 2$ (es decir, si n' cuelga directamente de n), entonces:
 - ◊ El nodo n' es *hijo* (ing., *child*) de n ; en concreto, si $s' = s.k$, $k \in \mathcal{N}_0$, decimos que n' es *hijo k -ésimo* de n (o también *primer hijo*, *segundo hijo*, y así hasta el *último hijo*).
 - ◊ El nodo n es *padre* (ing., *parent*) de n' .
 - ◊ Hay una *rama* (ing., *branch*) entre n y n' .
- El *grado* o *aridad* (ing., *degree* o *arity*) de n es el número de hijos del nodo; es decir, $\text{grado}_a(n) = |\{ \alpha: \alpha \in \text{dom}(a): (\exists k: k \in \mathcal{N}_0: \alpha = s.k) \}|$. El grado de un árbol a es el máximo de los grados de sus nodos, $\text{grado}(a) = \max n: n \in N_a: \text{grado}_a(n)$.

Además, dos nodos n y n' son *hermanos* (ing., *sibling*) si tienen el mismo padre, es decir, si se cumple que $s = \alpha.k$ y $s' = \alpha.k'$, para $k, k' \in \mathcal{N}_0$ y $\alpha \in \mathcal{N}_0^*$. Si, además, $k = k'+1$, a n' lo llamaremos *hermano izquierdo* de n y a n , *hermano derecho* de n' . En el árbol de la fig. 4.3, el nodo $\langle \lambda, \text{enero} \rangle$ es antecesor propio del nodo $\langle 3, \text{junio} \rangle$, también del $\langle 32, \text{marzo} \rangle$ y del $\langle 1, \text{agosto} \rangle$ (y, consecuentemente, estos tres nodos son descendientes propios del primero). Además, $\langle 3, \text{junio} \rangle$ y $\langle 1, \text{agosto} \rangle$ son hijos de $\langle \lambda, \text{enero} \rangle$ (y, en consecuencia, $\langle \lambda, \text{enero} \rangle$ es padre de los nodos $\langle 3, \text{junio} \rangle$ y $\langle 1, \text{agosto} \rangle$, que son hermanos, y hay una rama entre $\langle \lambda, \text{enero} \rangle$ y $\langle 3, \text{junio} \rangle$ y entre $\langle \lambda, \text{enero} \rangle$ y $\langle 1, \text{agosto} \rangle$). El grado del nodo raíz es 3 porque tiene tres hijos, mientras que el grado de todas las hojas es 0 porque no tienen ninguno, siendo ésta una propiedad común a todos los árboles del modelo.

c) Definiciones sobre subárboles

Sean $a, a' \in \mathcal{A}_V$ dos árboles generales y etiquetados. Decimos que a es *subárbol hijo* de a' , o simplemente *subárbol* (ing., *subtree*), si está contenido en él, o sea, si $\exists \alpha \in \mathcal{N}_0^*$ tal que:

- Las formas se superponen: $\alpha \# \text{dom}(a) = \{ \gamma: \gamma \in \text{dom}(a'): (\exists \beta: \beta \in \mathcal{N}_0^*: \gamma = \alpha \cdot \beta) \}$, siendo $\#: \mathcal{N}_0^* \times \mathcal{P}(\mathcal{N}_0^*) \rightarrow \mathcal{P}(\mathcal{N}_0^*)$ definida como: $\alpha \# \{s_1, \dots, s_n\} = \{\alpha.s_1, \dots, \alpha.s_n\}$.
- Las etiquetas de los nodos se mantienen: $\forall \langle s, v \rangle: \langle s, v \rangle \in N_a: a'(\alpha.s) = v$.

Concretamente, en este caso decimos que a es α -*subárbol* de a' ; si, además, $|\alpha| = 1$, a se denomina *primer subárbol* de a' si $\alpha = 1$, *segundo subárbol* de a' si $\alpha = 2$, etc., hasta el *último subárbol*. Notemos que todo árbol a es subárbol (concretamente, λ -subárbol) de sí mismo. Por ejemplo, en la fig. 4.4 se presentan un árbol que es subárbol del árbol de la fig. 4.3 (a la izquierda, en concreto, su tercer subárbol) y otro que no lo es (a la derecha).

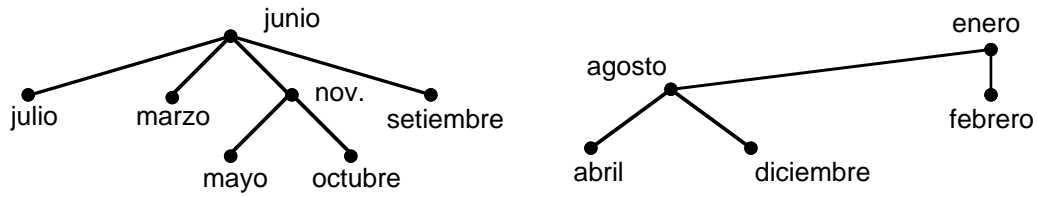


Fig. 4.4: un subárbol del árbol de la fig. 4.3 (a la izquierda) y otro que no lo es (a la derecha).

Una vez determinado completamente el modelo y las definiciones necesarias para hablar con propiedad de los árboles generales, vamos a determinar su signatura y especificación. Tal como se ha dicho previamente, la operación básica de la signatura de este modelo es enraizar un número indeterminado de árboles para formar uno nuevo, que tendrá en la raíz una etiqueta dada, *enraiza*: $\mathcal{A}_V^* \times V \rightarrow \mathcal{A}_V$. Ahora bien, la sintaxis de las especificaciones de Merlí (y de la mayoría de lenguajes de especificación o implementación) no permite declarar un número indeterminado de parámetros como parece requerir esta signatura y, por tanto, adoptamos un enfoque diferente. En concreto, introducimos el concepto de *bosque* (ing., *forest*), definido como una secuencia³ de árboles, de manera que *enraiza* tendrá dos parámetros, el bosque que contiene los árboles para enraizar y la etiqueta de la raíz⁴. Así, dados el árbol $a \in \mathcal{A}_V$, el bosque $b \in \mathcal{A}_V^*$, $b = a_1 \dots a_n$, y la etiqueta $v \in V$, la signatura es:

- Crear el bosque vacío: *crea*, devuelve el bosque λ sin árboles.
- Añadir un árbol al final del bosque: *inserta*(b, a), devuelve el bosque $b.a$.
- Averiguar el número de árboles que hay en el bosque: *cuántos*(b), devuelve n .
- Obtener el árbol i -ésimo del bosque: *i_ésimo*(b, i), devuelve a_i ; da error si $i > n$ o vale 0.
- Enraizar diversos árboles para formar uno nuevo: *enraiza*(b, v), devuelve un nuevo árbol a tal que la etiqueta de su raíz es v y cada uno de los a_i es el subárbol i -ésimo de a , es decir, devuelve el árbol a tal que:
 - $\diamond \text{dom}(a) = \{ \cup i: 1 \leq i \leq n: i \# \text{dom}(a_i) \} \cup \{ \lambda \}$.
 - $\diamond a(\lambda) = v$.
 - $\diamond \forall i: 1 \leq i \leq n: \forall \alpha: \alpha \in \text{dom}(a_i): a(i.\alpha) = a_i(\alpha)$.
- Averiguar la aridad de la raíz del árbol: *nhijos*(a), devuelve $\text{grado}_a(<\lambda, a(\lambda)>)$.
- Obtener el subárbol i -ésimo del árbol: *subárbol*(a, i) devuelve el subárbol i -ésimo de a o da error si i es mayor que la aridad de la raíz o cero; es decir, devuelve el árbol a' tal que:
 - $\diamond \text{dom}(a') = \{ \alpha : \alpha \in \mathcal{N}_0: i.\alpha \in \text{dom}(a) \}$.
 - $\diamond \forall \alpha: \alpha \in \text{dom}(a'): a'(\alpha) = a(i.\alpha)$.
- Consultar el valor de la etiqueta de la raíz: *raíz*(a), devuelve $a(\lambda)$.

³ No un conjunto, porque el orden de los elementos es significativo.

⁴ También podríamos haber optado por un conjunto de operaciones $\{ \text{enraiza}_1, \dots, \text{enraiza}_n \}$, donde n fuera un número lo bastante grande, de manera que *enraiza_i* enraizase i árboles con una etiqueta.

En la fig. 4.5 se muestra una especificación para este modelo. Definimos los géneros del bosque y del árbol en el mismo universo, porque se necesitan recíprocamente. Notemos que el bosque se crea fácilmente a partir de una instancia de las secuencias tal como fueron definidas en el apartado 1.5.1, pero considerando el género del alfabeto como parámetro formal (definido en el universo de caracterización *ELEM*). La instancia va seguida de varios renombramientos para hacer corresponder los identificadores de un tipo con los del otro, y de la ocultación de los símbolos que no deben ser visibles por el usuario del universo. La especificación del tipo *árbol* es trivial tomando *{enraiza}* como conjunto de constructoras generadoras (puro) y usando las operaciones sobre bosques. Como hay dos apariciones diferentes del universo *ELEM*, se prefija cada una de ellas con un identificador.

```

universo ÁRBOL_GENERAL (A es ELEM) es
  usa NAT, BOOL
  tipo árbol
  instancia CADENA (B es ELEM) donde B.elem es árbol
  renombra _._ por inserta, ||_| por cuántos, cadena por bosque
  esconde _._, [ ], etc.
  ops enraiza: bosque A.elem → árbol
      nhijos: árbol → nat
      subárbol: árbol nat → árbol
      raíz: árbol → A.elem
  error  $\forall a \in \text{árbol}; \forall i \in \text{nat}: [i > \text{nhijos}(a) \vee \text{NAT.ig}(i, 0)] \Rightarrow \text{subárbol}(a, i)$ 
  ecns  $\forall b \in \text{bosque}; \forall i \in \text{nat}; \forall v \in A.\text{elem}$ 
       $\text{nhijos}(\text{enraiza}(b, v)) = \text{cuántos}(b)$ 
       $\text{subárbol}(\text{enraiza}(b, v), i) = \text{CADENA.i-ésimo}(b, i)$ 
       $\text{raíz}(\text{enraiza}(b, v)) = v$ 
  funiverso

```

Fig. 4.5: especificación del TAD de los árboles generales.

4.1.2 Modelo de árbol binario

El modelo de árbol binario, denotado por \mathcal{A}^2_V , son las funciones $f : \{1, 2\}^* \rightarrow V$, dado que un nodo puede tener, como mucho, dos hijos. Como en el caso de los árboles generales, el dominio de la función ha de ser cerrado por prefijo, pero, en cambio, se permite la existencia del árbol vacío y, como consecuencia, la forma del árbol binario puede no ser compacta, presentando discontinuidades cuando la operación de enraizamiento involucre un árbol vacío como primer hijo y un árbol no vacío como segundo hijo. En la fig. 4.6 se muestran algunos árboles binarios válidos; notemos que si interpretamos los árboles como generales, el de la derecha y el del medio son idénticos.

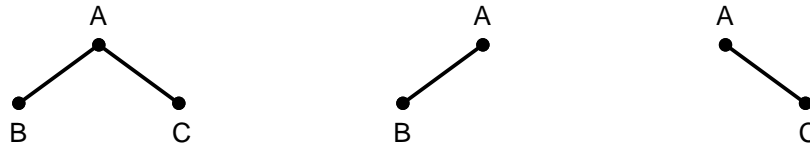


Fig. 4.6: tres árboles binarios tales que la raíz tiene: primer y segundo hijo (izquierda), primer hijo, pero no segundo (centro), y segundo hijo, pero no primero (derecha).

El resto del modelo no varía, siempre que se recuerde que el grado de todo nodo del árbol es, como mucho, igual a dos. Las definiciones dadas también son válidas respetando esta regla y considerando que la altura y la profundidad de un árbol vacío valen cero. Para mayor claridad, el primer hijo de un nodo de un árbol binario se llama *hijo izquierdo* (ing., *left child*) y el segundo se llama *hijo derecho* (ing., *right child*). De un árbol vacío también se puede decir que no existe; refiriéndose a un nodo, si un hijo (o hermano o cualquier otra relación de parentesco) de un nodo es vacío también podemos decir que el nodo no tiene este hijo (o hermano, etc.). Todas estas definiciones se pueden aplicar al caso de los subárboles.

Dados los árboles $a, a_1, a_2 \in \mathcal{A}^2_V$ y la etiqueta $v \in V$, definimos las siguientes operaciones (especificadas en la fig. 4.7)⁵:

- Crear el árbol vacío: *crea*, devuelve la función f_\emptyset de dominio vacío, $\text{dom}(f_\emptyset) = \emptyset$.
- Enraizar dos árboles y una etiqueta para formar un árbol: *enraiza*(a_1, v, a_2), devuelve el árbol a tal que $a(\lambda) = v$, a_1 es el subárbol izquierdo de a y a_2 es su subárbol derecho.
- Obtener los subárboles izquierdo y derecho de un árbol: *izq*(a) y *der*(a), respectivamente; si a es el árbol vacío, dan error.
- Obtener la etiqueta de la raíz: *raíz*(a), devuelve $a(\lambda)$ o da error si a es vacío.
- Averiguar si un árbol es vacío: *vacío?*(a), devuelve cierto si a es el árbol vacío y falso si no.

La generalización al caso de árboles n -arios cualesquiera, \mathcal{A}_{n_V} , es inmediata, considerando el modelo como las funciones $f: [1, n]^* \rightarrow V$, y queda como ejercicio para el lector.

4.1.3 Modelo de árbol con punto de interés

Los árboles con punto de interés, ya sea generales o n -arios, se caracterizan por la existencia de un nodo privilegiado identificado por el punto de interés, que sirve de referencia para diversas operaciones de actualización. La etiqueta de este nodo es la única que puede consultarse y, en consecuencia, también habrá diversas operaciones de modificación del punto de interés. Vamos a estudiar el caso de árboles generales (el caso n -ario es parecido).

⁵ La descripción formal es similar a los árboles generales y queda como ejercicio para el lector.

universo ÁRBOL_BINARIO (ELEM) es
usa BOOL
tipo árbol
ops crea: \rightarrow árbol
 enraiza: árbol elem árbol \rightarrow árbol
 izq, der: árbol \rightarrow árbol
 raíz: árbol \rightarrow elem
 vacío?: árbol \rightarrow bool
errores izq(crea); der(crea); raíz(crea)
ecns $\forall a, a_1, a_2 \in \text{árbol}; \forall v \in \text{elem}$
 izq(enraiza(a_1, v, a_2)) = a_1
 der(enraiza(a_1, v, a_2)) = a_2
 raíz(enraiza(a_1, v, a_2)) = v
 vacío?(crea) = cierto; vacío?(enraiza(a_1, v, a_2)) = falso
funiverso

Fig. 4.7: especificación del TAD de los árboles binarios.

El modelo de este tipo de árboles ha de recoger la posición del punto de interés y, por tanto, podemos considerarlo un par ordenado $\langle f: \mathcal{N}_0^* \rightarrow V, \mathcal{N}_0 \rangle$, donde V es el dominio de las etiquetas y se cumple que el segundo componente, que identifica la posición actual, está dentro del dominio de la función o bien tiene un valor nulo que denota la inexistencia de elemento distinguido. En cuanto a la signatura, el abanico es muy amplio tanto en lo que se refiere a las operaciones de actualización como a las de recorrido, y citamos a continuación algunas posibilidades sin entrar en detalle.

- Actualización. En las inserciones podemos plantearnos dos situaciones: insertar un nodo como hijo i -ésimo del punto de interés, o colgar todo un árbol a partir del punto de interés. En cualquier caso, parece lógico no cambiar el nodo distinguido. En cuanto a las supresiones, se presenta la misma disyuntiva: borrar nodos individualmente (que deberían ser hojas e hijos del nodo distinguido, o bien el mismo nodo distinguido) o bien subárboles enteros. Las diversas opciones no tienen por qué ser mutuamente excluyentes.
- Recorrido. También distinguimos dos situaciones: la primera consiste en ofrecer un conjunto de operaciones de interés general para mover el punto de interés al hijo i -ésimo, al padre o al hermano izquierdo o derecho. También se puede disponer de diversas operaciones de más alto nivel que incorporen las diversas estrategias de recorrido que se introducen en la sección 4.3 de manera que se obtienen recorridos ordenados según la forma del árbol. En cualquier caso, las operaciones de recorrido tendrán que permitir que se examinen los n nodos del árbol, y el esquema resultante será similar al uso de iteradores.

4.2 Implementación

En esta sección se presentan diversas estrategias de implementación de los árboles. Nos centramos en las representaciones más habituales para los modelos binario y general; en [Knu68, pp. 376-388] se presentan algunas variantes adicionales que pueden ser útiles en casos muy concretos. En el último apartado de la sección, se hace una breve referencia a la extensión de binario a n-ario y a los árboles con punto de interés.

4.2.1 Implementación de los árboles binarios

Como es habitual, las representaciones de árboles binarios se subdividen en dos grandes familias: encadenadas (por punteros y vectores) y secuenciales.

a) Representación encadenada

En la fig. 4.8 se muestra un ejemplo de árbol representado por punteros, y en la fig. 4.9 un universo que implementa esta estrategia. Hay un simple apuntador de cada nodo a sus dos hijos; si un hijo no existe, el encadenamiento correspondiente vale NULO. Las operaciones quedan aparentemente de orden constante en todos los casos, mientras que el espacio utilizado para guardar el árbol es lineal respecto al número de nodos que lo forman. El invariante prohíbe que haya más de un encadenamiento apuntando al mismo nodo (lo cual conduciría a una estructura en forma de grafo, v. capítulo 6) mediante el uso de una función auxiliar *correcto* (necesaria para hacer comprobaciones recursivas siguiendo la estructura del árbol) que se define sobre otra, *nodos*, que obtiene el conjunto de punteros a nodos de un árbol, similar a la operación *cadena* ya conocida (v. fig. 3.23).

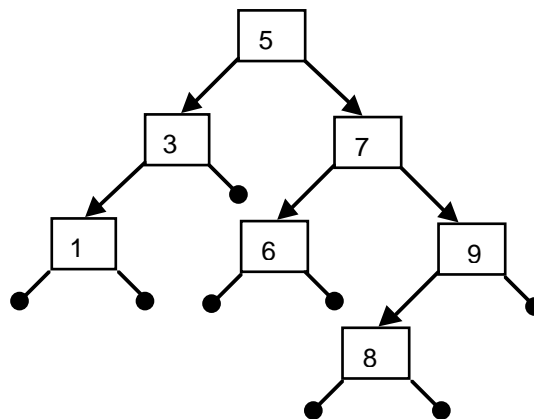


Fig. 4.8: ejemplo de implementación por punteros del TAD de los árboles binarios, siendo sus etiquetas números enteros.

universo ARBOL_BINARIO_ENC_PUNTEROS (ELEM) es
implementa ARBOL_BINARIO (ELEM)
usa BOOL
tipo árbol es \wedge nodo ftipo
tipo privado nodo es
tupla
 etiq es elem; hizq, hder son \wedge nodo
ftupla
ftipo
invariante (a es árbol): correcto(a), donde *correcto*: \wedge nodo \rightarrow bool se define:
 correcto(NULO) = cierto
 $p \neq \text{NULO} \Rightarrow \text{correcto}(p) = \text{correcto}(p^\wedge.\text{hizq}) \wedge \text{correcto}(p^\wedge.\text{hder}) \wedge$
 $\text{nodos}(p^\wedge.\text{hizq}) \cap \text{nodos}(p^\wedge.\text{hder}) = \{\text{NULO}\} \wedge$
 $p \notin \text{nodos}(p^\wedge.\text{hizq}) \cup \text{nodos}(p^\wedge.\text{hder})$
 y donde *nodos*: \wedge nodo $\rightarrow \mathcal{P}(\wedge \text{nodo})$ se define como:
 $\text{nodos}(\text{NULO}) = \{\text{NULO}\}$
 $p \neq \text{NULO} \Rightarrow \text{nodos}(p) = \{p\} \cup \text{nodos}(p^\wedge.\text{hizq}) \cup \text{nodos}(p^\wedge.\text{hder})$
función crea devuelve árbol es
devuelve NULO
función enraiza (a₁ es árbol; v es elem; a₂ es árbol) devuelve árbol es
var p es \wedge nodo fvar
 p := obtener_espacio
 si p = NULO entonces error si no p[^].v := v; p[^].hizq := a₁; p[^].hder := a₂ fsi
devuelve p
función izq (a es árbol) devuelve árbol es
 si a = NULO entonces error si no a := a[^].hizq fsi
devuelve a
función der (a es árbol) devuelve árbol es
 si a = NULO entonces error si no a := a[^].hder fsi
devuelve a
función raíz (a es árbol) devuelve elem es
var v es elem fvar
 si a = NULO entonces error si no v := a[^].v fsi
devuelve v
función vacío? (a es árbol) devuelve bool es
devuelve a = NULO
funiverso

Fig. 4.9: implementación por punteros del TAD de los árboles binarios.

Un examen más detallado de esta representación muestra una característica importante concerniente al funcionamiento de las operaciones *enraiza*, *izq* y *der*. Recordemos que, tal como se expuso en el apartado 3.3.4, la asignación $a_3 := \text{enraiza}(a_1, v, a_2)$ es en realidad una abreviatura de $\text{duplica}(\text{enraiza}(a_1, v, a_2), a_3)$, por lo que la instrucción tiene realmente coste lineal debido a la duplicación del árbol. Para evitar dichas duplicaciones, es necesario codificar la función en forma de acción definiendo, por ejemplo, dos parámetros de entrada de tipo *árbol* y uno de salida que almacene el resultado (v. fig. 4.11, arriba). Ahora bien, con esta estrategia, varios árboles pueden compartir físicamente algunos o todos los nodos que los forman (v. fig. 4.10, arriba a la izquierda), de manera que la modificación o destrucción de uno de los árboles tiene como efecto lateral la modificación de los otros⁶. Si se considera que esta peculiaridad del funcionamiento de la implementación de los árboles binarios es perniciosa, es necesario duplicar los árboles a enraizar (v. fig. 4.10, abajo), ya sea desde el programa que usa el TAD (como ocurre con la implementación de la fig. 4.9), ya sea sustituyendo la simple asignación de punteros por duplicaciones de los árboles implicados (v. fig. 4.11, abajo), con el coste lineal ya comentado. Una última opción consiste en evitar las duplicaciones y poner a NULO los apuntadores a árboles que actúen como parámetros de entrada (que dejarían de ser sólo de entrada) para evitar manipulaciones posteriores (v. fig. 4.10 arriba a la derecha y 4.11, centro); esta solución es viable si los subárboles enraizados pierden su identidad y puede considerarse subsumidos en el árbol resultante.

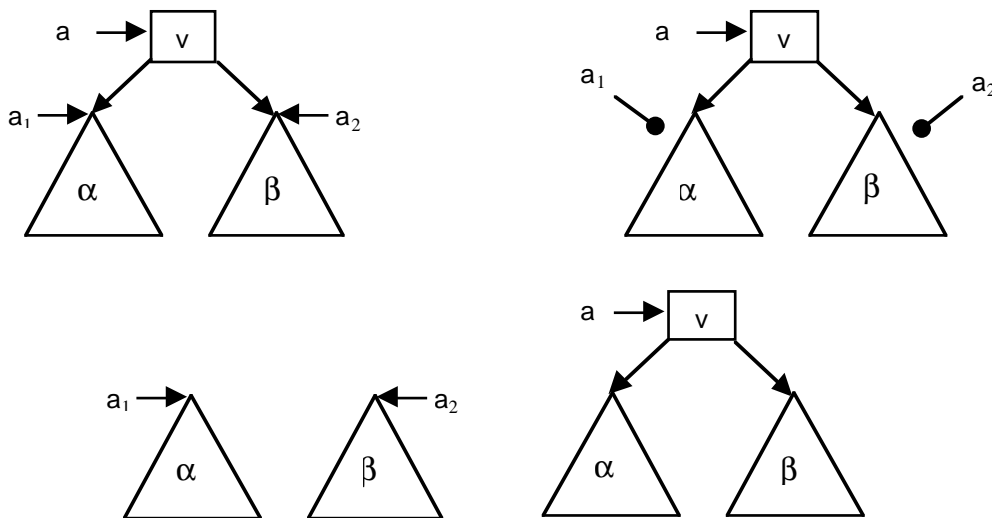


Fig. 4.10: efecto de $a := \text{enraiza}(a_1, v, a_2)$ en tres situaciones: sin duplicación (arriba a la izq.), ídem anulando los parámetros (arriba a la derecha) y con duplicación (abajo) de los árboles.

⁶ En realidad, éste no es un problema exclusivo de los árboles, sino en general de las representaciones por punteros. No lo hemos encontrado antes simplemente por la signatura de las operaciones que hemos definido sobre los diversos tipos de secuencias. Por ejemplo, si considerásemos la operación de concatenación de dos listas, surgiría la misma problemática.

acción enraiza (ent a_1 es árbol; ent v es elem; ent a_2 es árbol; sal a_3 es árbol) es

$a_3 := \text{obtener_espacio}$
si $a_3 = \text{NULO}$ entonces error
si no $a_3.v := v$; $a_3.hizq := a_1$; $a_3.hder := a_2$
fsi

facción

acción enraiza (ent sal a_1 es árbol; ent v es elem; ent sal a_2 es árbol; sal a_3 es árbol) es

$a_3 := \text{obtener_espacio}$
si $a_3 = \text{NULO}$ entonces error
si no $a_3.v := v$; $a_3.hizq := a_1$; $a_3.hder := a_2$; $a_1 := \text{NULO}$; $a_2 := \text{NULO}$
fsi

facción

acción enraiza (ent a_1 es árbol; ent v es elem; ent a_2 es árbol; sal a_3 es árbol) es

var p es ^nodo fvar

$a_3 := \text{obtener_espacio}$
si $a_3 = \text{NULO}$ entonces error
si no $a_3.v := v$; $\text{duplica}(a_1, a_3.hizq)$; $\text{duplica}(a_2, a_3.hder)$
fsi

facción

Fig. 4.11: implementación mediante una acción de enraiza, sin duplicación (arriba), ídem anulando los parámetros (centro), y con duplicación (abajo) de los árboles.

Por último, nótese que las implementaciones mediante acciones puede causar problemas cuando un parámetro real se asocia más de una vez con un parámetro formal, como en $\text{enraiza}(a_1, v, a_1, a_2)$. Una implementación realmente segura debería comprobar en el código estas situaciones especiales.

Los argumentos y alternativas son similares en el caso de las operaciones izq y der . Incluso, la representación sin copia presenta un riesgo añadido: la posible generación de basura. Este problema aparece porque las acciones correspondientes no liberan espacio, de manera que al ejecutar $izq(a, a)$ o $der(a, a)$ el espacio del árbol que no aparece en el resultado quedará inaccesible si no hay otros apuntadores a él. Mantener las operaciones de coste constante, pues, exige poder convivir con este problema (que por ejemplo desaparece si existe un mecanismo de recogida automática de basuras).

La representación encadenada con vectores sigue una estrategia similar. Ahora bien, en el caso de representar cada árbol en un vector, tal como se muestra en la fig. 4.12, las operaciones de enraizar y obtener los subárboles izquierdo y derecho exigen ineludiblemente duplicar todos los nodos del resultado para pasar de un vector a otro y, por

ello, quedan lineales sobre el número de nodos del árbol⁷. Si queremos rebajar el coste a constante se debe buscar la misma situación que con memoria dinámica, es decir, permitir que diferentes árboles compartan los nodos y, así, usar un único vector que los almacene todos y que gestione las posiciones libres en forma de pila, de manera que los árboles serán apuntadores (de tipo entero o natural) a la raíz que reside en el vector.

```

tipo árbol es tupla
    A es vector [de 0 a máx-1] de nodo
    si es nat
        ftupla
    ftipo
tipo privado nodo es tupla eti es elem; hizq, hder son entero ftupla ftipo

```

Fig. 4.12: representación encadenada de los árboles usando un vector para cada árbol.

En la fig. 4.13 se presenta una implementación usando un único vector para todos los árboles. Notemos que es necesario introducir el vector compartido como variable global y externa al universo de definición de los árboles, dado que es una estructura que utilizarán muchos árboles diferentes y que no es propiedad de uno solo. Esta solución viola todas las reglas de modularidad dadas hasta ahora y se puede pensar que es totalmente descartable. No obstante, es necesario tener en cuenta que el vector desempeña exactamente el mismo papel que la memoria dinámica en el caso de los punteros; la única diferencia es que en el segundo caso la memoria (que también es un objeto global y externo a cualquier módulo que lo usa) es un objeto anónimo y implícitamente referible desde cualquier punto de un programa. Una opción diferente consiste en declarar el vector como parámetro de las diferentes funciones de los árboles de manera que se encaje el esquema dentro del método de desarrollo modular habitual. Ahora bien, esta solución debe descartarse taxativamente porque la signatura de las operaciones es diferente en la especificación y en la implementación y, por tanto, se viola el principio de transparencia de la representación, pues un universo que use esta implementación ha de seguir unas convenciones totalmente particulares y no extrapolables a cualquier otra situación. Notemos, eso sí, que esta opción permitiría disponer de más de un vector para almacenar árboles.

Estudiemos el invariante de la memoria. En la primera línea se afirma que todo árbol individual que reside en la memoria está bien formado (en el mismo sentido que la representación de la fig. 4.9) y no incluye ninguna posición de la pila de sitios libres. A continuación, se asegura que toda posición del vector está dentro de algún árbol, o bien en la pila de sitios libres y, por último, se impide la compartición de nodos entre los diferentes árboles. Para escribir cómodamente el predicado se usan diversas funciones auxiliares: las ya conocidas *correcto*

⁷ El coste será lineal aunque se transformen las funciones en acciones con parámetros de entrada y de salida según las reglas del apartado 3.3.4.

y *nodos*, adaptadas a la notación vectorial; una función *cadena_der* que devuelve la cadena de posiciones que cuelga a partir de una posición por los encadenamientos derechos (que se usan en las posiciones libres para formar la pila), y una función, *raíces*, que devuelve el conjunto de posiciones del vector que son raíces de árboles, caracterizadas por el hecho de que no son apuntadas por ninguna otra posición ocupada ni están en la pila de sitios libres.

tipo memoria es tupla

A es vector [de 0 a máx-1] de nodo

sl es nat

ftupla

ftipo

tipo privado nodo es tupla etiq es elem; hizq, hder son entero ftupla ftipo

invariante (M es memoria):

$$\begin{aligned} \forall i: i \in \text{raíces}(M): \text{correcto}(M, i) \wedge \text{nodos}(M, i) \cap \text{cadena_der}(M, M.sl) = \{-1\} \wedge \\ \{ \cup i: i \in \text{raíces}(M): \text{nodos}(M, i) \} \cup \text{cadena_der}(M, M.sl) = [-1, \text{máx}-1] \wedge \\ \{ \cap i: i \in \text{raíces}(M): \text{nodos}(M, i) \} = \{-1\} \end{aligned}$$

donde *correcto*: memoria entero \rightarrow bool se define como:

$$\text{correcto}(M, -1) = \text{cierto}$$

$$\begin{aligned} i \neq -1 \Rightarrow \text{correcto}(M, i) = \text{correcto}(M, M[i].hizq) \wedge \text{correcto}(M, M[i].hder) \wedge \\ \text{nodos}(M, M[i].hizq) \cap \text{nodos}(M, M[i].hder) = \{-1\} \wedge \\ i \notin \text{nodos}(M, M[i].hizq) \cap \text{nodos}(M, M[i].hder), \end{aligned}$$

donde *nodos*: memoria entero $\rightarrow \mathcal{P}(\text{entero})$ se define como:

$$\text{nodos}(M, -1) = \{-1\}$$

$$i \neq -1 \Rightarrow \text{nodos}(M, i) = \{i\} \cup \text{nodos}(M, M[i].hizq) \cup \text{nodos}(M, M[i].hder),$$

donde *cadena_der*: memoria entero $\rightarrow \mathcal{P}(\text{entero})$ se define como:

$$\text{cadena_der}(M, -1) = \{-1\}$$

$$i \neq -1 \Rightarrow \text{cadena_der}(M, i) = \{i\} \cup \text{cadena_der}(M, M[i].hder)$$

y donde *raíces*: memoria $\rightarrow \mathcal{P}(\text{entero})$ se define como:

$$\begin{aligned} \text{raíces}(M) = \{ i: i \in [0, \text{máx}-1] - \text{nodos}(M, M.sl): \\ \neg (\exists j: j \in [0, \text{máx}-1] - \text{nodos}(M, M.sl): M.A[j].hizq = i \vee M.A[j].hder = i) \} \end{aligned}$$

{ *inicializa*(M): acción que es necesario invocar una única vez antes de crear cualquier árbol que use la memoria M, para crear la pila de sitios libres usando *hder*}

acción inicializa (sal M es memoria) es

var i es nat fvar

para todo i desde 0 hasta máx-2 hacer M.A[i].hder := i+1 fpara todo

M.A[máx-1].hder := -1; M.sl := 0

facción

(a) definición de la memoria y rutina de inicialización.

Fig. 4.13: implementación encadenada por vector compartido de los árboles binarios.

```

universo ÁRBOL_BINARIO_ENC_1_VECTOR (ELEM) es
  implementa ÁRBOL_BINARIO (ELEM)
  usa ENTERO, NAT, BOOL
  tipo árbol es entero ftipo
  invariante (a es árbol):  $-1 \leq a \leq \text{máx}-1 \wedge (a \neq -1 \Rightarrow a \notin \text{cadena\_der}(M, M.sl))$ 
  función crea devuelve árbol es
    devuelve -1
  función enraiza ( $a_1$  es árbol; v es elem;  $a_2$  es árbol) devuelve árbol es
    var i es entero fvar
      si  $M.sl = -1$  entonces error {el vector es lleno}
      si no {se obtiene un sitio libre y se deposita el nuevo nodo en él}
        i := M.sl; M.sl := M.A[M.sl].hder
        M.A[i] := <v,  $a_1$ ,  $a_2$ >
      fsi
    devuelve i
  función izq (a es árbol) devuelve árbol es
    si  $a = -1$  entonces error {árbol vacío}
    si no a := M[a].hizq
    fsi
  devuelve a
  función der (a es árbol) devuelve árbol es
    si  $a = -1$  entonces error {árbol vacío}
    si no a := M[a].hder
    fsi
  devuelve a
  función raíz (a es árbol) devuelve elem es
    var v es elem fvar
      si  $a = -1$  entonces error {árbol vacío}
      si no v := M[a].v
      fsi
    devuelve v
  función vacío? (a es árbol) devuelve bool es
    devuelve  $a = -1$ 
universo

```

(b) universo de los árboles.

Fig. 4.13: implementación encadenada por vector compartido de los árboles binarios (cont.).

En la codificación de las operaciones sobre árboles, se asume como precondition que los parámetros realmente son árboles válidos, es decir, los enteros correspondientes no están en la cadena de sitios libres. Como en realidad ésta es una propiedad estructural del tipo, se incluye en el invariante de su representación. La comprobación explícita en el código de esta precondition dotaría de mayor seguridad a la implementación pero incrementaría el coste de ejecución, a no ser que usáramos marcas para identificar las posiciones vacías.

En la implementación se incluye también una rutina de inicialización de la memoria que ha de ser invocada antes de cualquier manipulación y que se limita a crear la pila de sitios libres. Además, se puede incluir otra función para averiguar si queda espacio en el vector.

Por último, destacar que esta representación presenta también los problemas presentados para el caso de la representación por punteros, pudiéndose aplicar los mismos comentarios.

b) Representación secuencial

Siguiendo la filosofía de la implementación de estructuras lineales, una representación secuencial ha de almacenar los nodos del árbol sin usar campos adicionales de encadenamientos. Ahora bien, así como en las secuencias había una ordenación clara de los elementos, en los árboles no ocurre lo mismo. Por ejemplo, en el árbol de la fig. 4.3, ¿qué relación se puede establecer entre los nodos <2, febrero> y <332, octubre>? Intuitivamente, podríamos intentar guardar todos los nodos en el orden lexicográfico dado por las secuencias del dominio del árbol. Sin embargo, esta estrategia es totalmente ambigua porque una única configuración del vector se puede corresponder con diversos árboles (v. fig. 4.14). Otra posibilidad consiste en deducir una fórmula que asigne a cada posible nodo del árbol una posición dentro del vector; en el caso de nodos que no existan, la posición correspondiente del vector estará marcada como desocupada. Estudiemos este enfoque.

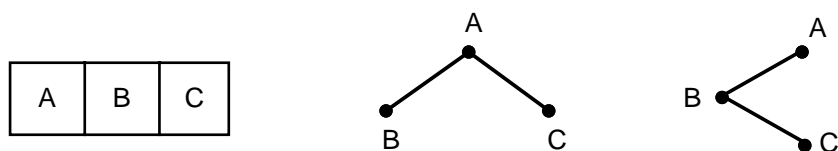


Fig. 4.14: una distribución de elementos dentro de un vector (a la izquierda) y dos posibles árboles que se pueden asociar según el orden lexicográfico (en el centro y a la derecha).

La estrategia adoptada consiste en usar la cadena que identifica la posición del nodo, de manera que la raíz vaya a la primera posición, el hijo izquierdo de la raíz a la segunda, el hijo derecho de la raíz a la tercera, el hijo izquierdo del hijo izquierdo de la raíz a la cuarta, etc. Concretamente, dado un nodo $n = \langle s, v \rangle$ ubicado en el nivel $k+1$ del árbol, $s = s_1 \dots s_k$, se puede definir la función Ψ que determina la posición del vector a la que va a parar cada uno

de los nodos del árbol como:

$$\Psi(s) = (1 + \sum_{i: 0 \leq i \leq k-1: 2^i}) + (\sum_{i: 1 \leq i \leq k: (s_i-1) \cdot 2^{k-i}}) = 2^k + (\sum_{i: 1 \leq i \leq k: (s_i-1) \cdot 2^{k-i}})$$

El primer factor da la posición que ocupa el primer nodo de nivel $k+1$ -ésimo dentro del árbol⁸ y cada uno de los sumandos del segundo término calcula el desplazamiento producido en cada nivel, según se baje por el hijo izquierdo ($s_i = 1$) o el derecho ($s_i = 2$).

A partir de esta fórmula se pueden derivar las diversas relaciones entre la posición que ocupa un nodo y la que ocupan sus hijos, hermano y padre, y que se muestran a continuación. Su cálculo por inducción queda como ejercicio para el lector, pero podemos comprobar su validez de forma intuitiva en el ejemplo dado en la fig. 4.15.

- Hijos. Sea el nodo $n = \langle s, v \rangle$ tal que $\Psi(s) = i$; su hijo izquierdo $n_1 = \langle s.1, v_1 \rangle$ ocupa la posición $\Psi(s.1) = 2i$ y su hijo derecho $n_2 = \langle s.2, v_2 \rangle$ la posición $\Psi(s.2) = 2i+1$.
- Padre. Simétricamente, dado el nodo $n = \langle s.k, v \rangle$ tal que $k \in [1, 2]$ y $\Psi(s.k) = i$, $i > 1$, la posición que ocupa su padre $\langle s, v' \rangle$ es $\Psi(s) = \lfloor i/2 \rfloor$.
- Hermanos. Dado que el nodo $n = \langle s, v \rangle$ es hijo izquierdo si $s = \alpha.1$, e hijo derecho si $s = \alpha.2$, y que en el primer caso la función $\Psi(s)$ da un resultado par y en el segundo uno impar, se cumple que, si $\Psi(s) = 2i$, el hermano derecho de n ocupa la posición $2i+1$, y si $\Psi(s) = 2i+1$, el hermano izquierdo de n ocupa la posición $2i$.

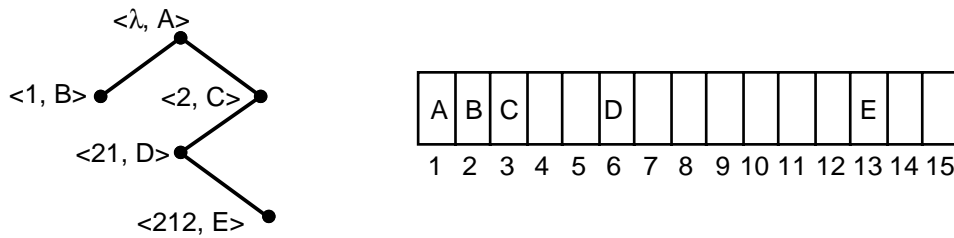


Fig. 4.15: un árbol binario (izquierda) y su representación secuencial (derecha); de cada nodo se da su posición y etiqueta; las posiciones del vector sin etiquetas están vacías.

La representación secuencial aquí presentada se denomina *montículo* (ing., *heap*) y exige que cada árbol se implemente con un único vector, de manera que todas las operaciones de enraizar y obtener los subárboles izquierdo y derecho quedan lineales sobre el número de nodos implicados. Por este motivo, su utilidad principal surge en los árboles con punto de interés con operaciones individuales sobre nodos. No obstante, incluso en este caso es necesario estudiar cuidadosamente si el espacio para almacenar el árbol es realmente menor que usando encadenamientos: si el árbol tiene pocos nodos pero muchos niveles,

⁸ Se puede demostrar por inducción que el número máximo de nodos en el nivel i -ésimo es 2^{i-1} .

probablemente el espacio que se ahorra por la ausencia de apuntadores no compensa las posiciones del vector desocupadas, necesarias para guardar los nodos de los niveles inferiores en la posición dada por Ψ . El caso ideal se da al almacenar árboles completos: un árbol es *completo* (ing., *full*) cuando sus niveles contienen todos los nodos posibles; también es bueno el caso de árboles *casi completos*, que son aquellos que no presentan ninguna discontinuidad en su dominio considerado en orden lexicográfico (el árbol completo es un caso particular de casi completo). Si se puede conjeturar el número de nodos, en ambos casos se aprovechan todas las posiciones del vector; si sólo se puede conjeturar el número de niveles, el aprovechamiento será óptimo en el caso de árboles completos, pero habrá algunas posiciones libres (en la parte derecha del vector) en los árboles casi completos.

4.2.2 Implementación de los árboles generales

Por lo que respecta a la representación encadenada, independientemente de si usamos punteros o vectores, la primera opción puede ser fijar la aridad del árbol y reservar tantos campos de encadenamiento como valga ésta. Esta estrategia, no obstante, presenta el inconveniente de acotar el número de hijos de los nodos (cosa a veces imposible) y, además, incluso cuando esto es posible, el espacio resultante es infrautilizado. Veamos porqué.

Sea n la aridad de un árbol general a , N_a^i el conjunto de nodos de a que tienen i hijos, y X e Y el número de encadenamientos nulos y no nulos de a , respectivamente. Se cumplen las relaciones siguientes:

$$X = \sum i: 0 \leq i \leq n-1: \|N_a^i\| \cdot (n-i) \quad (4.1)$$

$$\|N_a\| = \sum i: 0 \leq i \leq n: \|N_a^i\| \quad (4.2)$$

$$X + Y = n \cdot \|N_a\| \quad (4.3)$$

Ahora, sea B el número de ramas que hay en a . Como que en cada nodo del árbol, exceptuando la raíz, llega una y sólo una rama, queda claro que:

$$\|N_a\| = B + 1 \quad (4.4)$$

y como que de un nodo con i hijos salen i ramas, entonces:

$$B = \sum i: 1 \leq i \leq n: \|N_a^i\| \cdot i \quad (4.5)$$

Combinando (4.4) i (4.5), obtenemos:

$$\|N_a\| = (\sum i: 1 \leq i \leq n: \|N_a^i\| \cdot i) + 1 \quad (4.6)$$

A continuación, desarrollamos la parte derecha de (4.1) desdoblado el sumatorio en dos, manipulando sus cotas y usando (4.2) y (4.6) convenientemente:

$$\begin{aligned}
& \sum i: 0 \leq i \leq n-1: \|N_a^i\| \cdot (n-i) = \\
& (\sum i: 0 \leq i \leq n-1: \|N_a^i\| \cdot n) - (\sum i: 0 \leq i \leq n-1: \|N_a^i\| \cdot i) = \\
& (n \cdot \sum i: 0 \leq i \leq n-1: \|N_a^i\|) - (\sum i: 1 \leq i \leq n-1: \|N_a^i\| \cdot i) = \\
& (n \cdot \sum i: 0 \leq i \leq n-1: \|N_a^i\|) - (\sum i: 1 \leq i \leq n: \|N_a^i\| \cdot i + \|N_a^n\| \cdot n) = \\
& (n \cdot \sum i: 0 \leq i \leq n-1: \|N_a^i\| + \|N_a^n\| \cdot n) - (\sum i: 1 \leq i \leq n: \|N_a^i\| \cdot i) = \\
& (n \cdot \sum i: 0 \leq i \leq n: \|N_a^i\|) - (\sum i: 1 \leq i \leq n: \|N_a^i\| \cdot i) = \text{(usando (4.2) y (4.6))} \\
& (n \cdot \|N_a\|) - (\|N_a\| + 1) = (n-1) \cdot \|N_a\| + 1
\end{aligned}$$

Resumiendo, hemos expresado el número X de encadenamientos nulos en función del número total de nodos, $X = (n-1) \cdot \|N_a\| + 1$, y usando (4.3) podemos comprobar que X es mayor que el número de encadenamientos no nulos en un factor $n-1$, de manera que la representación encadenada propuesta desaprovecha mucho espacio.

La solución intuitiva a este problema consiste en crear, para cada nodo, una lista de apuntadores a sus hijos. Desde el nodo se accede a la celda que contiene el apuntador al primer hijo y, desde cada celda, se puede acceder a la celda que contiene el apuntador al hijo correspondiente y a la que contiene el apuntador al hermano derecho. Ahora bien, esta solución es claramente ineficiente porque introduce muchos apuntadores adicionales. La alternativa más eficiente surge al observar que, dado un número fijo de nodos, mientras más grande es la aridez del árbol que los contiene, mayor es el número de encadenamientos desaprovechados, porque el número total de estos encadenamientos crece mientras que el de ramas no varía. Por esto, buscamos una estrategia que convierta el árbol general en binario, para minimizar el número de encadenamientos y así optimizar el espacio. Tomando como punto de partida la lista de apuntadores a los hijos que acabamos de proponer, se puede modificar encadenando directamente los nodos del árbol de manera que cada nodo tenga dos apuntadores, uno a su primer hijo y otro al hermano derecho. Esta implementación se denomina *hijo izquierdo, hermano derecho* (ing., *leftmost-child, right-sibling*) y, rotando el árbol resultante, se puede considerar como un árbol binario tal que el hijo izquierdo del árbol binario represente al hijo izquierdo del árbol general, pero el hijo derecho del árbol binario represente al hermano derecho del árbol general (v. fig. 4.16). Por esto, la representación también se denomina *representación por árbol binario*.

Notemos que el hijo derecho de la raíz será siempre el árbol vacío, porque la raíz no tiene hermano, por definición, de modo que se puede aprovechar para encadenar todos los árboles que forman un bosque y, así, un bosque también tendrá la apariencia de árbol binario, tal como se muestra en la fig. 4.17.

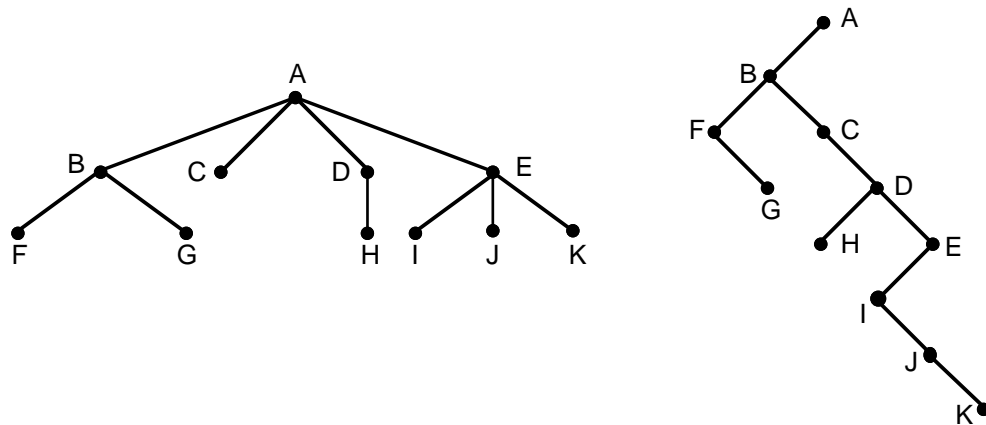


Fig. 4.16: un árbol general (izquierda) y su representación por árbol binario (derecha).

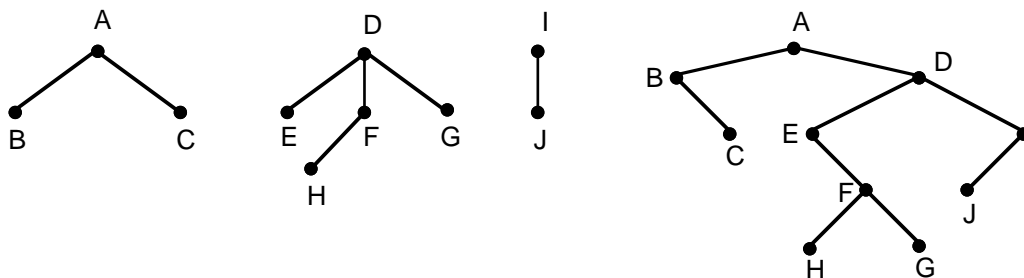


Fig. 4.17: un bosque con tres árboles (izq.) y su representación por árbol binario (der.).

En la fig. 4.18 se muestra la implementación de esta estrategia por punteros. El bosque se representa como una cola de árboles encadenados con dos apuntadores al primer y último elemento (podría también haberse implementado circularmente y así ahorrarnos un apuntador, pero el código resultante es un poco más enrevesado). No se efectúa una instancia de las colas, para optimizar el espacio y aprovechar el encadenamiento a hermano derecho tal como se ha explicado. En la cola se podría poner un elemento fantasma para no distinguir el caso de cola vacía y también un contador, pero no se hace porque probablemente las colas serán cortas en el caso general y el espacio desperdiciado será mayor en términos relativos; de lo contrario, sería necesario reconsiderar ambas decisiones. En lo que respecta al invariante, es necesario dar propiedades para ambos géneros; como los bosques se pueden considerar como árboles, ambos se ayudan de un predicado auxiliar *correcto*: $\wedge \text{nodo} \rightarrow \text{bool}$, que comprueba las propiedades habituales de los árboles binarios. En los árboles no se puede decir que el encadenamiento derecho sea nulo, porque un árbol que pase a formar parte del bosque no cumple esta propiedad.

universo ÁRBOL_GENERAL_POR_BINARIO (ELEM) es
implementa ÁRBOL_GENERAL (ELEM)
usa BOOL
tipo bosque es tupla
 prim, ult son ^nodo
 ftupla
ftipo
tipo árbol es ^nodo ftipo
tipo privado nodo es tupla
 etiq es elem
 hizq, hder son ^nodo
 ftupla
ftipo
invariante (b es bosque): correcto(b.prim) \wedge b.ult \in cadena_der(b.prim)
 (a es árbol): $a \neq \text{NULO} \wedge \text{correcto}(a)$
 donde *correcto*: ^nodo \rightarrow bool se define como en la fig. 4.8 y
 siendo *cadena_der*: ^nodo $\rightarrow \mathcal{P}(\text{^nodo})$ como en el caso binario (v. fig. 4.13)
función crea devuelve bosque es
devuelve <NULO, NULO>
función cuántos (b es bosque) devuelve bosque es
devuelve cuenta(b.prim)
función inserta (b es bosque; a es árbol) devuelve bosque es
 si b.ult = NULO entonces b.prim := a si no b.ult^.hder := a fsi
 b.ult := a
devuelve b
función i_ésimo (b es bosque; i es nat) devuelve árbol es
devuelve hermano_derecho_i_ésimo(b.prim, i)
función enraiza (b es bosque; v es elem) devuelve árbol es
var p es ^nodo fvar
 p := obtener_espacio
 si p = NULO entonces error {falta espacio} si no p^ := <v, b.prim, NULO> fsi
devuelve p
 {Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función nhijos (a es árbol) devuelve nat es
devuelve cuenta(a^.hizq)

Fig. 4.18: implementación hijo izquierdo, hermano derecho de los árboles generales.

```

{Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función subárbol (a es árbol; i es nat) devuelve árbol es
  devuelve hermano_derecho_i_ésimo(ahizq, i)

{Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función raíz (a es árbol) devuelve elem es
  devuelve av

{Función auxiliar cuenta(p): devuelve la longitud de la cadena derecha que
  cuelga de p }
función privada cuenta (p es ^nodo) devuelve nat es
  var cnt es nat fvar
    cnt := 0
    mientras p ≠ NULO hacer cnt := cnt + 1; p := phder fmientras
  devuelve cnt

{Función auxiliar hermano_derecho_i_ésimo(p, i): avanza i encadenamientos
  a la derecha a partir de p; da error si no hay suficientes encadenamientos}
función priv hermano_derecho_i_ésimo (p es ^nodo; i es nat) devuelve árbol es
  var cnt es nat fvar
    cnt := 1
    mientras (p ≠ NULO) ∧ (cnt < i) hacer p := phder; cnt := cnt + 1 fmientras
    si p = NULO entonces error fsi
  devuelve p

funiverso

```

Fig. 4.18: implementación hijo izquierdo, hermano derecho de los árboles generales (cont.).

Si consideramos la eficiencia, las operaciones *cuántos* e *i_ésimo* de los bosques, y *nhijos* y *subárbol* de los árboles, son $\Theta(k)$ en el caso peor, siendo k el grado del árbol. El resto de operaciones son $\Theta(1)$, aunque en las operaciones constructoras depende del tratamiento dado al problema de la compartición de nodos (v. apartado 4.2.1).

La representación secuencial de los árboles generales sigue la misma idea que el caso binario, pero comenzando a numerar por 0 y no por 1 para simplificar las fórmulas de obtención del padre y de los hijos. Es necesario, antes que nada, limitar a un máximo r la aridad mayor permitida en un nodo y también el número máximo de niveles (para poder dimensionar el vector). Entonces, definimos Ψ_r como la función que asigna a cada nodo una posición en el vector tal que, dado $s = s_1 \dots s_k$:

$$\Psi(s) = (1 + \sum i: 0 \leq i \leq k-1: r^i) + (\sum i: 1 \leq i \leq k: (s_i-1).r^{k-i}) = r^k + (\sum i: 1 \leq i \leq k: (s_i-1).r^{k-i})$$

El resto de fórmulas son también una extensión del caso binario y su deducción queda igualmente como ejercicio:

- Hijos. Sea el nodo $n = \langle s, v \rangle$ tal que $\Psi_r(s) = i$; la posición que ocupa su hijo k -ésimo $n' = \langle s.k, v' \rangle$ es $\Psi_r(s.k) = ir+k$. El primer sumando representa la posición anterior al primer hijo de n , y el sumando k es el desplazamiento dentro de los hermanos.
- Padre. Simétricamente, dado el nodo $n = \langle s, v \rangle$ tal que $\Psi_r(s) = i$, $i > 1$, la posición que ocupa su padre es $\lfloor (i-1) / r \rfloor$.
- Hermanos. Sólo es necesario sumar o restar uno a la posición del nodo en cuestión, según se quiera obtener el hermano derecho o izquierdo, respectivamente.

4.2.3 Variaciones en los otros modelos de árboles

Por lo que respecta a la extensión de los árboles binarios a los árboles n -arios, en la representación encadenada se puede optar por poner tantos campos como aridad tenga el árbol, o bien usar la estrategia hijo izquierdo, hermano derecho. En cuanto a la representación secuencial, no hay ninguna diferencia en las fórmulas del caso general tomando como r la aridad del árbol.

El modelo de punto de interés presenta ciertas peculiaridades respecto al anterior. En la representación encadenada, puede ser útil tener apuntadores al padre para navegar convenientemente dentro del árbol. Opcionalmente, si la representación es por hijo izquierdo, hermano derecho, el apuntador al hermano derecho del hijo de la derecha del todo (que es nulo) se puede reciclar como apuntador al padre y, así, todo nodo puede acceder sin necesidad de utilizar más espacio adicional que un campo booleano de marca (en caso de usar punteros), a costa de un recorrido que probablemente será rápido. La representación secuencial queda igual y es necesario notar que es precisamente en este modelo donde es más útil ya que, por un lado, el acceso al padre (y, en consecuencia, a cualquier hermano) se consigue sin necesidad de usar más encadenamientos y, por lo tanto, se puede implementar fácilmente cualquier estrategia de recorrido; por el otro, las operaciones individuales sobre nodos se pueden hacer en tiempo constante, porque consisten simplemente en modificar una posición del vector calculada inmediatamente.

Los modelos recorribles se estudian en la próxima sección. Por lo que respecta a las variantes totalmente abiertas de los árboles, exigen dobles encadenamientos para asegurar supresiones rápidas, lo que en el contexto de los árboles significa que hay que añadir apuntadores a los padres. Las variantes abiertas son costosas de implementar con montículo debido a los movimientos de los elementos.

4.2.4 Estudio de eficiencia espacial

La cuestión primordial consiste en determinar cuando se comporta mejor una representación

encadenada que una secuencial. Un árbol k -ario de n nodos y k campos de encadenamiento ocupa un espacio $(X+k)n$, tomando como unidad el espacio ocupado por un encadenamiento y siendo X la dimensión de las etiquetas. El mismo árbol representado con la estrategia hijo izquierdo, hermano derecho ocupa sólo $(X+2)n$; el precio a pagar por este ahorro son las búsquedas (que generalmente serán cortas) de los subárboles y, por ello, generalmente la segunda implementación será preferible a la primera. Si se usa una representación secuencial, el árbol ocupará un espacio que dependerá de su forma: si los árboles con los que se trabaja son casi completos habrá pocas posiciones desaprovechadas. En general, dada una aridad máxima r y un número máximo de niveles k , el espacio necesario para una representación secuencial es $(\sum_{i: 1 \leq i \leq k: r^i} i)X = [(r^k - 1) / (r - 1)]X$. Esta cifra es muy alta incluso para r y k no demasiado grandes. Por ejemplo, un árbol 4-ario de 10 niveles como máximo, exige dimensionar el vector de 1.398.101 posiciones; suponiendo que las etiquetas sean enteros (caso usual), los árboles deben tener del orden de medio millón de nodos para que la representación secuencial sea mejor. Por ello, esta estrategia sólo se usa con árboles casi completos.

Para ser más precisos, calculamos a continuación cuál es la relación que debe haber entre el número de nodos y el número de niveles para que la representación secuencial realmente supere la encadenada (implementada por punteros). Supongamos que la aridad del árbol es r y fijemos el número k de niveles del árbol. Definimos el *factor de carga* α de un árbol como el cociente entre el número n de nodos y el número total de nodos que puede haber en un árbol de aridad r y altura k , $\alpha = n(r-1) / (r^k - 1)$, que cumple $0 \leq \alpha \leq 1$. Sustituyendo, el espacio de la representación por árbol binario queda $\alpha(X+2)(r^k - 1) / (r - 1)$; si comparamos este valor con el espacio de la representación secuencial, $[(r^k - 1) / (r - 1)]X$, puede verse que el factor de carga debe superar $X/(X+2)$ para que la representación secuencial supere la encadenada. Es decir, mientras más grande sea el espacio necesario para almacenar la etiqueta, menos posiciones vacías pueden quedar en el vector propio de la representación secuencial.

4.3 Recorridos

Usualmente, los programas que trabajan con árboles necesitan aplicar sistemáticamente un tratamiento a todos sus nodos en un orden dado por la forma del árbol; es lo que se denomina *visitar* todos los nodos del árbol. Se distinguen dos categorías básicas de recorridos: los que se basan en las relaciones padre-hijo de los nodos, que denominaremos *recorridos en profundidad*, y los que se basan en la distancia del nodo a la raíz, conocidos como *recorridos en anchura* o *por niveles*. La incorporación de estos recorridos a los árboles resulta en árboles recorribles con las cuatro operaciones habituales. No obstante, para simplificar el estudio, consideramos que los recorridos se plasman en operaciones de *signatura recorrido*: $\text{árbol} \rightarrow \text{lista_etiq}$, donde *lista_etiq* es el resultado de instanciar las listas con punto de interés con el tipo de las etiquetas. Es decir, el recorrido del árbol crea una lista con los nodos ordenados según la estrategia del recorrido.

En esta sección estudiaremos detalladamente el caso de árboles binarios; la extensión al resto de modelos queda como ejercicio para el lector.

4.3.1 Recorridos en profundidad de los árboles binarios

Dado el árbol binario $a \in \mathcal{A}^2_V$, se definen tres recorridos en profundidad (v. fig. 4.19 y 4.20):

- *Recorrido en preorden* (ing., *preorder traversal*). Si a es el árbol vacío, termina el recorrido; si no lo es, primero se visita la raíz de a y, a continuación, se recorren en preorden los subárboles izquierdo y derecho.
- *Recorrido en inorden* (ing., *inorder traversal*). Si a es el árbol vacío, termina el recorrido; si no lo es, primero se recorre en inorden el subárbol izquierdo de a , a continuación, se visita su raíz y, por último, se recorre en inorden el subárbol derecho de a .
- *Recorrido en postorden* (ing., *postorder traversal*). Si a es el árbol vacío, termina el recorrido; si no lo es, primero se recorren en postorden sus subárboles izquierdo y derecho y, a continuación, se visita su raíz.

$\text{preorden}(\text{crea}) = \text{crea}$

$\text{preorden}(\text{enraiza}(a_1, v, a_2)) = \text{concat}(\text{concat}(\text{inserta}(\text{crea}, v), \text{preorden}(a_1)), \text{preorden}(a_2))$

$\text{inorden}(\text{crea}) = \text{crea}$

$\text{inorden}(\text{enraiza}(a_1, v, a_2)) = \text{concat}(\text{inserta}(\text{inorden}(a_1), v), \text{inorden}(a_2))$

$\text{postorden}(\text{crea}) = \text{crea}$

$\text{postorden}(\text{enraiza}(a_1, v, a_2)) = \text{inserta}(\text{concat}(\text{postorden}(a_1), \text{postorden}(a_2)), v)$

Fig. 4.19: especificación ecuacional de los recorridos de árboles binarios, donde *concat* es la concatenación de dos listas que deja el punto de interés a la derecha del todo.

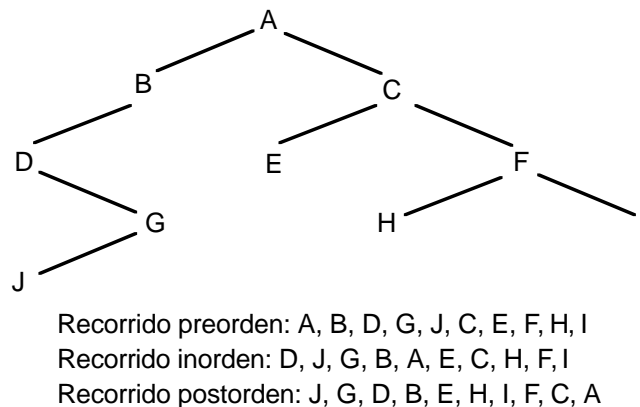


Fig. 4.20: un árbol binario y sus recorridos preorden, inorden y postorden.

Notemos que el recorrido postorden se puede definir como el inverso especular del recorrido preorden, es decir, para obtener el recorrido postorden de un árbol a se puede recorrer a en preorden, pero visitando siempre el subárbol derecho antes que el izquierdo (especular), considerando la lista de etiquetas resultante como el inverso de la solución. Esta visión será útil posteriormente en la formulación de los algoritmos de recorrido postorden.

Estos tres recorridos se usan en diferentes contextos de la informática. Por ejemplo, el recorrido preorden permite el cálculo de atributos heredados de una gramática. Precisamente, el calificativo "heredados" significa que el hijo de una estructura arborescente hereda (y, posiblemente, modifica) el valor del atributo en el padre. El recorrido inorden de un árbol que cumple ciertas relaciones de orden entre los nodos los obtiene ordenados; en concreto, si todos los nodos del subárbol de la izquierda presentan una etiqueta menor que la raíz y todos los del subárbol de la derecha una etiqueta mayor, y los subárboles cumplen recursivamente esta propiedad, el árbol es un árbol de búsqueda, que se caracteriza precisamente por la ordenación de los elementos en un recorrido inorden (v. sección 5.5). Por último, el recorrido postorden de un árbol que represente una expresión es equivalente a su evaluación; en las hojas residen los valores (que pueden ser literales o variables) y en los nodos intermedios los símbolos de operación que se pueden aplicar sobre los operandos, cuando éstos ya han sido evaluados (i.e., visitados).

La implementación más obvia de los recorridos consiste en construir funciones recursivas a partir de la especificación, aplicando técnicas de derivación de programas (en este caso, incluso, por traducción directa). Estas funciones pueden provocar problemas en la ejecución aplicadas sobre árboles grandes y con un soporte *hardware* no demasiado potente; por ello, es bueno disponer también de versiones iterativas, que pueden obtenerse a partir de la aplicación de técnicas de transformación de las versiones recursivas, que se ayudan de una pila que simula los bloques de activación del ordenador durante la ejecución de estas últimas.

En la fig. 4.21 se muestra la transformación de recursivo a iterativo del recorrido en preorden. Se supone que tanto éste como otros recorridos residen en un universo parametrizado por el tipo de las etiquetas, $\text{ÁRBOL_BINARIO_CON_RECORRIDOS}(ELEM)$, que enriquece el universo de definición de los árboles binarios, $\text{ÁRBOL_BINARIO}(ELEM)$, razón por la que no se accede a la representación del tipo, sino que se utilizan las operaciones que ofrece su especificación. Notemos que los árboles se van guardando dentro de la pila siempre que no estén vacíos, y se empila el hijo izquierdo después del derecho para que salga antes. Queda claro que un nodo siempre se inserta en la lista resultado antes de que se inserten sus descendientes.


```

función preorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol fvar
  p := PILA.crea; l := LISTA_INTERÉS.crea
  si ¬vacío?(a) entonces p := empila(p, a) fsi
  mientras ¬vacía?(p) hacer
    aaux := cima(p); p := desempila(p) {aaux no puede ser vacío}
    l := inserta(l, raíz(aaux))
    si ¬ vacío?(hder(aaux)) entonces p := empila(p, der(aaux)) fsi
    si ¬ vacío?(hizq(aaux)) entonces p := empila(p, izq(aaux)) fsi
  fmientras
devuelve l

```

Fig. 4.21: implementación del recorrido preorden con la ayuda de una pila.

En lo que respecta a la eficiencia, suponiendo coste constante en las operaciones sobre pilas y listas, el recorrido de un árbol de n nodos es $\Theta(n)$ en todos los casos, siempre que las operaciones *hizq* y *hder* no dupliquen árboles (parece lógico no hacerlo, porque los árboles no se modifican), puesto que a cada paso se inserta una y sólo una etiqueta en la lista y, además, una misma etiqueta se inserta sólo una vez. En cuanto al espacio auxiliar, el crecimiento de la pila depende de la forma del árbol; como sus elementos serán apuntadores (si realmente estamos trabajando sin copiar los árboles), el espacio total tiene como coste en el caso peor el número esperado de elementos en la pila⁹. El cálculo de la dimensión de la pila sigue el razonamiento siguiente:

- Inicialmente, hay un único árbol en la pila.
- En la ejecución de un paso del bucle, la longitud de la pila puede:
 - ◊ disminuir en una unidad, si el árbol que se desempila sólo tiene un nodo;
 - ◊ quedar igual, si el árbol que se desempila sólo tiene subárbol izquierdo o derecho;
 - ◊ incrementarse en una unidad, si el árbol que se desempila tiene ambos subárboles.

El caso peor, pues, es el árbol sin nodos de grado uno en el que todo nodo que es hijo derecho es a la vez una hoja. Así, al empilar el árbol formado por el único hijo izquierdo que también es hoja, dentro de la pila habrá este árbol y todos los subárboles derechos encontrados en el camino, cada uno de ellos formado por un único nodo (es decir, lo más pequeños posible); para un árbol de n nodos, la pila puede crecer hasta $\lceil n/2 \rceil$ elementos y el coste queda $\Theta(n)$ en el caso peor (v. fig. 4.22). En cambio, el caso mejor es el árbol en el que todo nodo tiene exactamente un subárbol hijo de manera que la pila nunca guarda más de un elemento. En el caso de un árbol completo la pila crece hasta $\lceil \log_2 n \rceil$ al visitar las hojas del último nivel, y así el coste en el caso mejor es $\Theta(\log n)$.

⁹ En realidad esta pila estaba implícita en la versión recursiva y, por ello, no se puede considerar un empeoramiento respecto a ésta.

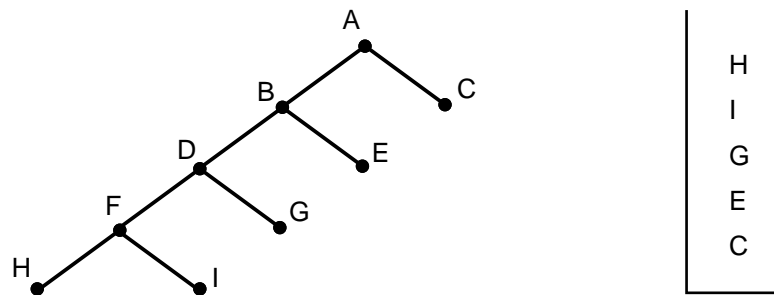


Fig. 4.22: caso peor en el recorrido preorden en cuanto al tamaño de la pila (se identifican los árboles por su raíz dentro de la pila).

El recorrido en postorden sigue el mismo esquema, considerado como un recorrido preorden inverso especular (v. fig. 4.23). Para implementar la noción de especular se empila antes el hijo derecho que el hijo izquierdo, y para obtener la lista en orden inverso se coloca el punto de interés al principio siempre que se inserta un nuevo elemento, de manera que la inserción sea por la izquierda y no por la derecha (se podría haber requerido un modelo de listas con operación de insertar por el principio). De acuerdo con la especificación, al final del recorrido es necesario mover el punto de interés a la derecha del todo, para dejar el resultado en el mismo estado que los otros algoritmos vistos hasta ahora. Pese a esta sobrecarga, el coste asintótico del algoritmo es equivalente al recorrido preorden, $\Theta(n)$ en todos los casos.

```

función postorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol fvar
p := PILA.crea; l := LISTA_INTERÉS.crea
si ¬vacío?(a) entonces p := empila(p, a) fsi
mientras ¬vacía?(p) hacer
    aaux := cima(p); p := desempila(p)
    l := principio(inserta(l, raíz(aaux)))
    si ¬vacío?(hizq(aaux)) entonces p := empila(p, hizq(aaux)) fsi
    si ¬vacío?(hder(aaux)) entonces p := empila(p, hder(aaux)) fsi
fmientras
mientras ¬final?(l) hacer l := avanza(l) fmientras
devuelve l

```

Fig. 4.23: implementación del recorrido postorden con la ayuda de una pila.

Por último, el recorrido en inorden se obtiene a partir del algoritmo de la fig. 4.21: cuando se desempila un árbol, su raíz no se inserta directamente en la lista sino que se vuelve a empilar, en forma de árbol con un único nodo, entre los dos hijos, para obtenerla en el momento

adecuado, tal como se presenta en la fig. 4.24. Consecuentemente, la pila puede crecer todavía más que en los dos recorridos anteriores. En el caso peor, que es el mismo árbol que el de la fig. 4.22, puede llegar a guardar tantos elementos como nodos tiene el árbol. Además, el algoritmo es más lento, porque en ese caso no todas las iteraciones añadirán nodos a la lista solución (eso sí, no se empeorará el coste asintótico, $\Theta(n)$ en todos los casos). En el caso peor de la fig. 4.22, primero se empilan n subárboles de un nodo en $\lfloor n/2 \rfloor$ iteraciones y, a continuación, se visitan sus raíces en n vueltas adicionales del bucle.

```

función inorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol fvar
  p := PILA.crea; l := LISTA_INTERÉS.crea
  si  $\neg$  vacío?(a) entonces p := empila(p, a) fsi
  mientras  $\neg$  vacía?(p) hacer
    aaux := cima(p); p := desempila(p) {aaux no puede ser vacío}
    si vacío?(hizq(aaux))  $\wedge$  vacío?(hder(aaux)) entonces
      l := inserta(l, raíz(aaux)) {hay un único nodo, que se visita}
    si no {se empila la raíz entre los dos hijos}
      si  $\neg$  vacío?(hder(aaux)) entonces p := empila(p, hder(aaux)) fsi
      p := empila(p, enraiza(crea, raíz(aaux), crea))
      si  $\neg$  vacío?(hizq(aaux)) entonces p := empila(p, hizq(aaux)) fsi
    fsi
  fmientras
devuelve l

```

Fig. 4.24: implementación del recorrido inorden con la ayuda de una pila.

4.3.2 Árboles binarios enhebrados

Los *árboles enhebrados* (ing., *threaded tree*) son una representación ideada por A.J. Perlis y C. Thornton en el año 1960 ("Symbol Manipulation by Threaded Lists", *Communications ACM*, 3), que permite implementar los recorridos sin necesidad de espacio adicional (como mucho, un par de booleanos de marca en cada nodo), y se basa en la propiedad ya conocida de que una representación encadenada normal de los árboles desaprovecha muchos apuntadores que no apuntan a ningún nodo. En el caso binario, y a partir de la fórmula calculada en el apartado 4.2.2, de los $2n$ encadenamientos existentes en un árbol de n nodos hay exactamente $n+1$ sin usar. Por ello, es bueno plantearse si se pueden reciclar estos apuntadores para llevar a cabo cualquier otra misión, y de ahí surge el concepto de *hebra* (ing., *thread*): cuando un nodo no tiene hijo derecho, se sustituye el valor nulo del encadenamiento derecho por su sucesor en inorden, y cuando no tiene hijo izquierdo, se sustituye el valor nulo del encadenamiento izquierdo por su predecesor en inorden (v. fig.

4.25); de esta manera, se favorece la implementación de los recorridos sobre el árbol.

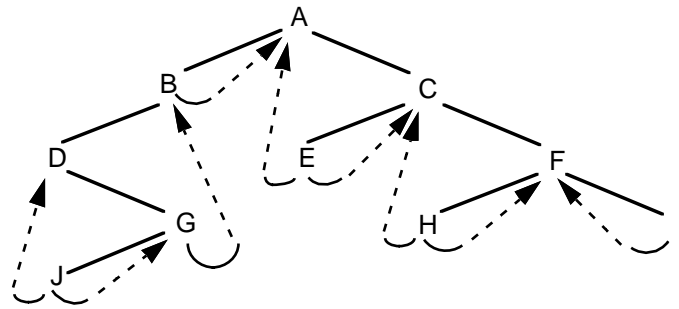


Fig. 4.25: el árbol binario de la fig. 4.20, enhebrado en inorden (las hebras son las rayas discontinuas).

La elección del enhebrado inorden y no de otro se justifica porque favorece la implementación de los diferentes recorridos. El punto clave consiste en definir cuál es el primer nodo del árbol que se ha de visitar y cómo se calcula el siguiente nodo dentro del recorrido:

- Preorden. El primer nodo es la raíz del árbol (si la hay). Dado un nodo n , su sucesor en preorden es el hijo izquierdo, si tiene; si no, el hijo derecho, si tiene. Si es una hoja, es necesario seguir las hebras derechas (que llevan a nodos antecesores de n , ya visitados por definición de recorrido preorden, tales que n forma parte de su subárbol izquierdo) hasta llegar a un nodo que, en lugar de hebra derecha, tenga hijo derecho, que pasa a ser el sucesor en preorden.
- Inorden. El primer nodo (si lo hay) se localiza bajando sucesivamente por la rama de más a la izquierda del árbol. Dado un nodo n (que cumple que los nodos de su árbol izquierdo ya han sido visitados, por definición de recorrido inorden), su sucesor en inorden es el primero en inorden del hijo derecho, si tiene; si no, simplemente se sigue la hebra (que recordamos que apunta precisamente al sucesor en inorden de n).
- Postorden. Es inmediato a partir de la definición como preorden inverso especular.

En la fig. 4.26 se presenta la implementación de estos recorridos. Como se está explotando una característica de la representación de los árboles, los algoritmos se incluyen dentro del mismo universo de definición del tipo porque así pueden acceder a ella. En la figura se ha optado por una representación con punteros (queda como ejercicio la representación por vectores). El esquema de los tres recorridos es casi idéntico y sólo cambia la implementación de la estrategia; notemos el uso de un puntero auxiliar que desempeña el papel de elemento actual en el recorrido. Se introducen funciones auxiliares para obtener el primer elemento y el siguiente según los diferentes recorridos; por ello, la implementación que aquí se ofrece se adapta con mucha facilidad al caso de árboles con punto de interés o con iteradores. El

invariante incorpora el concepto de hebra y garantiza, por un lado, que tanto el hijo izquierdo del primero en inorden como el hijo derecho del último en inorden valen NULO y, por otro, que las hebras apuntan a los nodos correctos según la estrategia adoptada. Los predicados *correcto*, *cadena_izq* y *cadena_der* varían respecto a la fig. 4.13, porque el caso trivial deja de ser el puntero NULO y pasa a ser la existencia de hebra. El predicado *nodos*, en cambio, queda igual, porque la inserción reiterada de algunos nodos no afecta al resultado. Las pre y postcondiciones de las funciones auxiliares, así como los invariantes de los bucles, son inmediatas y quedan como ejercicio para el lector.

tipo árbol es \wedge nodo ftipo

tipo privado nodo es

tupla

v es etiq; hizq, hder son \wedge nodo

\exists hizq, \exists hder son bool¹⁰ {indican si los encadenamientos son hebras}

ftupla

ftipo

invariante (a es árbol): $\text{correcto}(a) \wedge \forall p: p \in \text{nodos}(a) - \{\text{NULO}\}$:

$p^\wedge.\text{hizq} = \text{NULO} \Leftrightarrow (p \in \text{cadena_izq}(a) \wedge \neg p^\wedge.\exists \text{hizq}) \wedge$

$p^\wedge.\text{hder} = \text{NULO} \Leftrightarrow (p \in \text{cadena_der}(a) \wedge \neg p^\wedge.\exists \text{hder}) \wedge$

$(\neg p^\wedge.\exists \text{hizq} \wedge p^\wedge.\text{hizq} \neq \text{NULO}) \Rightarrow p \in \text{cadena_izq}(p^\wedge.\text{hizq}^\wedge.\text{hder}) \wedge$

$(\neg p^\wedge.\exists \text{hder} \wedge p^\wedge.\text{hder} \neq \text{NULO}) \Rightarrow p \in \text{cadena_der}(p^\wedge.\text{hder}^\wedge.\text{hizq})$

donde *correcto*: $\wedge \text{nodo} \rightarrow \text{bool}$ se define como:

1) $\text{correcto}(\text{NULO}) = \text{cierto}$ {caso trivial}

2) $p \neq \text{NULO} \wedge \neg p^\wedge.\exists \text{hizq} \wedge \neg p^\wedge.\exists \text{hder} \Rightarrow \text{correcto}(p) = \text{cierto}$ {hoja}

3) $p \neq \text{NULO} \wedge p^\wedge.\text{hizq} \wedge p^\wedge.\exists \text{hder} \Rightarrow \text{correcto}(p) =$ {tiene los dos hijos}

$\text{correcto}(p^\wedge.\text{hizq}) \wedge \text{correcto}(p^\wedge.\text{hder}) \wedge$

$\text{nodos}(p^\wedge.\text{hizq}) \cap \text{nodos}(p^\wedge.\text{hder}) = \emptyset \wedge$

$p \notin \text{nodos}(p^\wedge.\text{hizq}) \cup \text{nodos}(p^\wedge.\text{hder})$

4) $p \neq \text{NULO} \wedge \neg p^\wedge.\exists \text{hizq} \wedge p^\wedge.\exists \text{hder} \Rightarrow$ {sólo tiene hijo derecho}

$\text{correcto}(p) = \text{correcto}(p^\wedge.\text{hder}) \wedge p \notin \text{nodos}(p^\wedge.\text{hder})$

5) $p \neq \text{NULO} \wedge p^\wedge.\exists \text{hizq} \wedge \neg p^\wedge.\exists \text{hder} \Rightarrow$ {sólo tiene hijo izquierdo}

$\text{correcto}(p) = \text{correcto}(p^\wedge.\text{hizq}) \wedge p \notin \text{nodos}(p^\wedge.\text{hizq}),$

donde *cadena_izq*: $\wedge \text{nodo} \rightarrow \mathcal{P}(\wedge \text{nodo})$ se define (y *cadena_der* simétricamente):

$\neg p^\wedge.\exists \text{hizq} \Rightarrow \text{cadena_izq}(p) = \emptyset$

$p^\wedge.\exists \text{hizq} \Rightarrow \text{cadena_izq}(p) = \{p\} \cup \text{cadena_izq}(p^\wedge.\text{hizq})$

y donde *nodos*: $\wedge \text{nodo} \rightarrow \mathcal{P}(\wedge \text{nodo})$ se define de la manera habitual

Fig. 4.26: implementación de los recorridos en profundidad con árboles enhebrados.

¹⁰ Estos dos booleanos se pueden obviar en caso de una representación por vectores, codificando la existencia o no de hijo con el signo de los encadenamientos: un valor negativo significa que el encadenamiento representa un hijo y es necesario tomar su valor absoluto para acceder a él.

```

función privada siguiente_preorden (act es árbol) devuelve árbol es
  si act^.  $\exists$ hizq entonces act := act^.hizq {si hay hijo izquierdo, ya lo tenemos}
  si no {es necesario remontar por las hebras hasta llegar a un nodo con hijo derecho}
    mientras  $\neg$  act^.  $\exists$ hder hacer act := act^.hder fmientras
    act := act^.hder
  fsi
devuelve act

función privada primero_inorden (act es ^nodo) devuelve ^nodo es
  si act  $\neq$  NULO entonces {es necesario bajar por la rama izquierda, reiteradamente}
    mientras act^.  $\exists$ hizq hacer act := act^.hizq fmientras
  fsi
devuelve act

función privada siguiente_inorden (act es ^nodo) devuelve ^nodo es
  si  $\neg$ act^.  $\exists$ hder entonces act := act^.hder {se sigue la hebra}
  si no act := primero_inorden(act^.hder) {busca primero en inorden del hijo derecho}
  fsi
devuelve act

función privada siguiente_preorden_especular (act es ^nodo) devuelve ^nodo es
  si act^.  $\exists$ hder entonces act := act^.hder {si hay hijo derecho, ya lo tenemos}
  si no {es necesario remontar por las hebras hasta llegar a un nodo con hijo izquierdo}
    mientras  $\neg$  act^.  $\exists$ hizq hacer act := act^.hizq fmientras
    act := act^.hizq
  fsi
devuelve act

función preorden (a es árbol) devuelve lista_etiq es
var l es lista_etiq; act es ^nodo fvar
  l := LLISTA_INTERÉS.crea; act := a
  mientras act  $\neq$  NULO hacer l := inserta(l, act^.v); act := siguiente_preorden(act) fmientras
devuelve l

función inorden (a es árbol) devuelve lista_etiq: como preorden, pero inicializando act
  al primero inorden, y avanzando con siguiente_inorden

función postorden (a es árbol) devuelve lista_etiq: como preorden, pero avanzando
  con siguiente_preorden_especular, colocando la lista al inicio justo antes de insertar
  el elemento actual y, al acabar, colocándola al final

```

Fig. 4.26: implementación de los recorridos en profundidad con árboles enhebrados (cont.).

Dado un árbol enhebrado de n nodos, su recorrido siguiendo cualquier estrategia queda $\Theta(n)$ en todos los casos. En concreto, dado que cada hebra se sigue una única vez y, o bien sólo las hebras de la izquierda, o bien sólo las de la derecha, el número total de accesos adicionales a los nodos del árbol durante el recorrido es $\lfloor (n+1) / 2 \rfloor$.

El último paso consiste en modificar las operaciones habituales de la signatura de los árboles binarios para adaptarlas a la estrategia del enhebrado. En la fig. 4.27 se presenta el esquema general de las operaciones constructoras; notemos que el mantenimiento del enhebrado exige buscar el primero y el último en inorden del árbol y, por tanto, la representación clásica de los árboles binarios no asegura el coste constante de las operaciones que, en el caso peor, pueden llegar a ser incluso lineales. Si este inconveniente se considera importante, se pueden añadir a la representación de los árboles dos apuntadores adicionales a estos elementos.

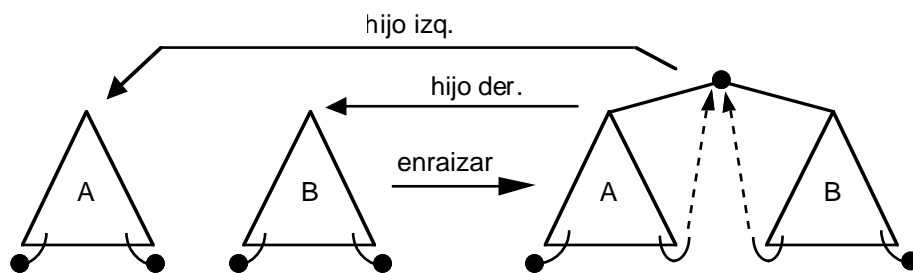


Fig. 4.27: esquema general del funcionamiento de los árboles binarios enhebrados inorden.

En la fig. 4.28 se muestra la nueva representación del tipo y la codificación de *enraiza* con esta estrategia; sería necesario retocar los algoritmos de la fig. 4.26 para adaptarlos a la nueva situación. Si la formación del árbol es previa a cualquier recorrido y se prevé recorrerlo múltiples veces, se puede optar por mantener la representación no enhebrada durante los enraizamientos y enhebrar el árbol antes del primer recorrido.

4.3.3 Recorrido por niveles de los árboles binarios

El *recorrido por niveles* o *por anchura* (ing., *level order* o *breadth traversal*) de un árbol binario consiste en visitar primero la raíz del árbol, después los nodos que están en el nivel 2, después los que están en el nivel 3, etc., hasta visitar los nodos del último nivel del árbol; para cada nivel, los nodos se visitan de izquierda a derecha. Como resultado, el orden de visita se corresponde con la numeración presentada en el punto 4.2.1 b, ya que se basa en la misma idea. Por ejemplo, en el árbol de la fig. 4.20 se obtiene el recorrido A, B, C, D, E, F, G, H, I, J. Esta política de visita por distancia a la raíz es útil en determinados contextos donde las ramas tienen algún significado especialmente significativo.

```

tipo árbol es
  tupla prim, ult, raíz son ^nodo ftupla
ftipo
tipo privado nodo es
  tupla
    v es etiq
     $\exists$ hizq,  $\exists$ hder son bool
    hizq, hder son ^nodo
  ftupla
ftipo
invariante (a es árbol): el mismo que la fig. 4.26 y
   $(a.raíz = NULO \Leftrightarrow a.prim = NULO) \wedge (a.prim = NULO \Leftrightarrow a.ult = NULO) \wedge$ 
   $a.raíz \neq NULO \Rightarrow (a.prim^.hizq = NULO) \wedge (a.ult^.hder = NULO)$ 

función enraiza (a1 es árbol; v es elem; a2 es árbol) devuelve árbol es
var a es árbol fvar
  a.raíz := obtener_espacio
  si a.raíz = NULO entonces error {no hay espacio}
  si no {primero enraizamos normalmente}
    a.raíz^.v := v; a.raíz^.hizq := a1.raíz; a.raíz^.hder := a2.raíz
    a.raíz^. $\exists$ hizq := (a1.raíz  $\neq$  NULO); a.raíz^. $\exists$ hder := (a2.raíz  $\neq$  NULO)
    {a continuación, se enhebra el árbol en inorden}
    si a1.prim  $\neq$  NUL entonces a.prim := a1.prim si no a.prim := a fsi
    si a2.ult  $\neq$  NUL entonces a.ult := a2.ult si no a.ult := a fsi
    si a1.ult  $\neq$  NULO entonces a1.ult^.hder := a.raíz fsi
    si a2.prim  $\neq$  NULO entonces a2.prim^.hizq := a.raíz fsi
  fsi
devuelve a

```

Fig. 4.28: algoritmo de enraizamiento de árboles binarios enhebrados inorden.

La especificación del recorrido por niveles no es tan sencilla como la de los recorridos en profundidad dado que, en este caso, no importa tanto la estructura recursiva del árbol como la distribución de los nodos en los diferentes niveles. En la fig. 4.29 se muestra una propuesta que usa una cola para guardar los subárboles pendientes de recorrer en el orden adecuado. Notemos que la cola mantiene la distribución por niveles de los nodos y obliga a la introducción de una operación auxiliar *niveles_con_cola*. Otra opción es formar listas de pares etiqueta-nivel que, correctamente combinadas, ordenen los nodos según la estrategia. Sea como sea, el resultado peca de una sobre-especificación causada por la adopción de la semántica inicial como marco de trabajo.


```

niveles(a) = niveles_conCola(encola(COLA.crea, a))
niveles_conCola(crea) = LISTA_INTERÉS.crea
[vacío?(cabeza(encola(c, a)))] ⇒
    niveles_conCola(encola(c, a)) = niveles_conCola(desencola(encola(c, a)))
[¬vacío?(cabeza(encola(c, a)))] ⇒
    niveles_conCola(encola(c, a)) =
        concatena(inserta(crea, raíz(cabeza(encola(c, a)))),
            niveles_conCola(encola(desencola(encola(c, a),
                hizq(cabeza(encola(c, a))),
                hder(cabeza(encola(c, a)))))

```

Fig. 4.29: especificación del recorrido por niveles con la ayuda de una cola.

En la fig. 4.30 se presenta la implementación del algoritmo siguiendo la misma estrategia que la especificación, procurando simplemente no encolar árboles vacíos. Observemos la similitud con el algoritmo iterativo del recorrido postorden de la fig. 4.23, cambiando la pila por una cola. Así mismo, su coste asintótico es también $\Theta(n)$ para un árbol de n nodos, en todos los casos.

```

función niveles (a es árbol) devuelve lista_etiq es
var c es cola_árbol; l es lista_etiq; aux es árbol fvar
    c := COLA.crea; l := LISTA_INTERÉS.crea
    si ¬ vacío?(a) entonces c := encola(p, a) fsi
    mientras ¬ vacío?(c) hacer
        aux := cabeza(c); c := desencola(c) {aux no puede ser vacío}
        l := inserta(l, raíz(aux))
        si ¬ vacío?(hizq(aux)) entonces c := encola(c, hizq(aux)) fsi
        si ¬ vacío?(hder(aux)) entonces c := encola(c, hder(aux)) fsi
    fmientras
    devuelve l

```

Fig. 4.30: implementación del recorrido por niveles con la ayuda de una cola.

En lo que respecta a la dimensión de la cola, es necesario notar que, al visitar el nodo $n \in N_a$ residente en el nivel k del árbol a , en la cola sólo habrá los subárboles tales que su raíz x esté en el nivel k de a (x estará más a la derecha que n dentro de a), o bien en el nivel $k+1$ de a (x será hijo de un nodo ya visitado en el nivel k). Por este motivo, el caso peor es el árbol en que, después de visitar el último nodo del penúltimo nivel, en la cola hay todos los subárboles posibles cuya raíz esté en el último nivel. Es decir, el árbol peor es el árbol completo, donde hay $n = 2^k - 1$ nodos dentro del árbol y, así, en el último nivel hay exactamente 2^{k-1} nodos, es decir, $\lceil n/2 \rceil$ elementos y el espacio queda $\Theta(n)$.

4.4 Colas prioritarias

Una aplicación interesante de las estructuras arborescentes consiste en usarlas no tanto como un TAD sino como una estrategia de representación de otros TAD, explotando la idea de jerarquía para conseguir implementaciones eficientes. En esta sección vamos a estudiar un ejemplo concreto, el TAD de las colas prioritarias, y en capítulos posteriores encontraremos estructuras arborescentes para representar tablas y relaciones de equivalencia.

Las colas organizadas por prioridades o, abreviadamente, *colas prioritarias* (ing., *priority queue*) son un tipo especial de cola donde los elementos no se insertan por el final, sino que se ordenan según una prioridad asociada. Así, al principio de la cola tendremos el elemento más prioritario, a continuación el segundo más prioritario, etc. Supondremos para simplificar que no hay dos elementos con la misma prioridad. Para comparar prioridades utilizaremos la operación $_<_$ de *ELEM_ORD* de manera que $x < y$ indica que x es más prioritario que y ¹¹.

Dado un dominio de elementos V y una operación de comparación según la prioridad, $<$, el modelo de las colas prioritarias de elementos de V son las secuencias de elementos de V , V^* , con las operaciones siguientes, para $c \in V^*$ y $v \in V$, siendo $c = v_1 \dots v_n$, $\forall i: 1 \leq i < n: v_i < v_{i+1}$:

- Crear la cola vacía: *crea*, devuelve la secuencia λ .
- Añadir un elemento a la cola: *inserta*(c, v), devuelve la secuencia $v_1 \dots v_{k-1} \vee v_k \dots v_n$ tal que $v_{k-1} < v < v_k$; da error si ya existía algún elemento en c con la misma prioridad que v .
- Obtener el elemento menor de la cola: *menor*(c), devuelve v_1 ; da error si c es vacía.
- Borrar el elemento menor de la cola: *borra*(c), devuelve $v_2 \dots v_n$; da error si c es vacía.
- Consultar si la cola está vacía: *vacía?*(c), devuelve cierto si c es λ , y falso en caso contrario.

En la fig. 4.31 se muestra la especificación del tipo *colapr* de las colas prioritarias; los parámetros formales se definen en el universo de caracterización *ELEM_<*. Recordemos que la comparación se refiere siempre a las prioridades. Destacamos que las ecuaciones no reflejan el modelo de manera evidente, pues en realidad la inserción se define conmutativa como si fueran conjuntos¹², y son la supresión y la consulta las operaciones que buscan el elemento más prioritario dentro de la cola. Se controla la inserción de elementos de idéntica prioridad.

La implementación de las colas prioritarias usando estructuras lineales es costosa en alguna operación. Si mantenemos la lista desordenada, la inserción queda constante (siempre y cuando no comprobemos explícitamente la inexistencia de elementos con la misma prioridad

¹¹ Por ello, también decimos del elemento más prioritario que es el menor elemento (algunos autores lo hacen explícito y hablan de cola prioritaria de mínimos).

¹² De hecho, el modelo de las colas prioritarias podría formularse en términos de conjuntos.

que el insertado), pero la supresión y la consulta exigen una búsqueda lineal; si escogemos ordenar la lista, entonces es la inserción la operación que requiere una búsqueda lineal a cambio del coste constante de la consulta, mientras que la supresión depende de la representación concreta de la lista (si es secuencial, es necesario mover elementos y entonces queda lineal). Una opción intermedia consiste en mantener la lista desordenada durante las inserciones y ordenarla a continuación, siempre que todas las inserciones se hagan al principio y las consultas y supresiones a continuación, como pasa con cierta frecuencia. En este caso, el coste total de mantener una cola de n elementos (es decir, primero insertar los n elementos y después consultarlos y obtenerlos de uno en uno) queda $\Theta(n \log n)$ con un buen algoritmo de ordenación, en comparación con el coste $\Theta(n^2)$ de las dos anteriores.

universo COLA_PRIORITARIA (ELEM_<) es

usa BOOL

tipo colapr

ops crea: \rightarrow colapr

inserta: colapr elem \rightarrow colapr

menor: colapr \rightarrow elem

borra: colapr \rightarrow colapr

vacía?: colapr \rightarrow bool

errores

$[\neg(v_2 < v_1) \wedge \neg(v_1 < v_2)] \Rightarrow \text{inserta}(\text{inserta}(c, v_1), v_2)$

menor(crea); borra(crea)

ecns $\forall c \in \text{cuapr}; \forall v, v_1, v_2 \in \text{elem}$

$\text{inserta}(\text{inserta}(c, v_1), v_2) = \text{inserta}(\text{inserta}(c, v_2), v_1)$

$\text{menor}(\text{inserta}(\text{crea}, v)) = v$

$[v_2 < \text{menor}(\text{inserta}(c, v_1))] \Rightarrow \text{menor}(\text{inserta}(\text{inserta}(c, v_1), v_2)) = v_2$

$[\text{menor}(\text{inserta}(c, v_1)) < v_2] \Rightarrow$

$\text{menor}(\text{inserta}(\text{inserta}(c, v_1), v_2)) = \text{menor}(\text{inserta}(c, v_1))$

$\text{borra}(\text{inserta}(\text{crea}, v)) = \text{crea}$

$[v_2 < \text{menor}(\text{inserta}(c, v_1))] \Rightarrow \text{borra}(\text{inserta}(\text{inserta}(c, v_1), v_2)) = \text{inserta}(c, v_1)$

$[\text{menor}(\text{inserta}(c, v_1)) < v_2] \Rightarrow$

$\text{borra}(\text{inserta}(\text{inserta}(c, v_1), v_2)) = \text{inserta}(\text{borra}(\text{inserta}(c, v_1)), v_2)$

$\text{vacía?}(\text{crea}) = \text{cierto}$

$\text{vacía?}(\text{inserta}(c, v)) = \text{falso}$

funiverso

Fig. 4.31: especificación del TAD de las colas prioritarias.

4.4.1 Implementación por árboles parcialmente ordenados y casi completos

A continuación, introducimos una representación que garantiza el coste logarítmico de las operaciones modificadoras de las colas prioritarias manteniendo el coste constante de la consultora. Esta representación usa una variante de árbol conocida con el nombre de *árbol parcialmente ordenado* (ing., *partially ordered tree*), que cumple que todo nodo es menor que sus hijos, si tiene. Además, nos interesará que el árbol sea casi completo (o sea, que no presente discontinuidades en un recorrido por niveles) para asegurar una buena eficiencia espacial y temporal. En la fig. 4.32 se ilustra este concepto.

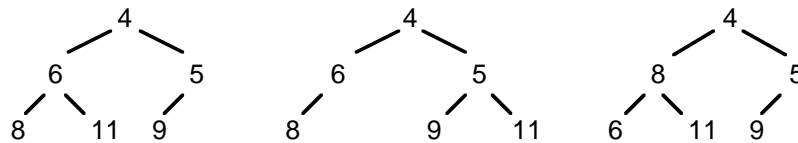


Fig. 4.32: tres árboles binarios con los mismos elementos: *parcialmente ordenado y completo* (izquierda), *sólo parcialmente ordenado* (centro) y *sólo completo* (derecha).

Es obvio que en esta clase de árbol el elemento menor reside en la raíz y, por lo tanto, la ejecución de *menor* es constante si disponemos de acceso directo a la raíz desde la representación del tipo. Es necesario ver, pues, cómo quedan las operaciones de inserción y de supresión. Para hacer este estudio trataremos árboles parcialmente ordenados binarios; v. ejercicio 4.13 para el estudio de la generalización a cualquier aridad¹³.

a) Inserción en un árbol parcialmente ordenado y casi completo

Para que el árbol resultante de una inserción sea completo, insertamos el nuevo elemento v en la primera posición libre en un recorrido por niveles del árbol. Ahora bien, en el caso general, esta inserción no da como resultado la satisfacción de la propiedad de los árboles parcialmente ordenados, porque v puede ser más prioritario que su padre. Debido a ello, es necesario comenzar un proceso de reestructuración del árbol, que consiste en ir intercambiando v con su padre hasta que la relación de orden se cumpla, o bien hasta que v llegue a la raíz; cualquiera de las dos condiciones de parada implica que el árbol vuelve a cumplir la propiedad de ordenación parcial.

¹³ Los árboles parcialmente ordenados n -arios, $n > 2$, pueden ser interesantes en algunos contextos, porque, cuanto mayor es la n , menos comparaciones entre elementos es necesario hacer en las inserciones.

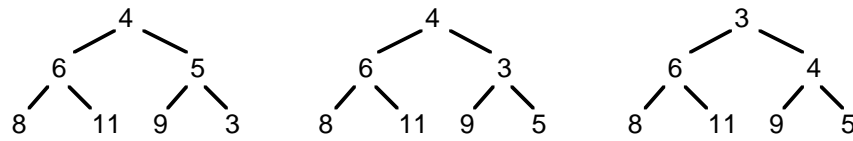


Fig. 4.33: inserción del 3 paso a paso en el árbol de la fig. 4.39, izquierda.

Notemos que, en el caso peor, el número de intercambios de elementos es igual al número de niveles del árbol y, como el árbol es casi completo, el número de niveles es del orden del logaritmo en base 2 del número n de elementos del árbol, exactamente $\lceil \log_2(n+1) \rceil$ (recordemos que en el nivel i caben hasta 2^{i-1} nodos); si la representación de este tipo de árbol permite realizar cada intercambio en tiempo constante, el coste de la operación será efectivamente $\Theta(\log n)$ en el caso peor, y $\Theta(1)$ en el caso mejor.

b) Supresión en un árbol parcialmente ordenado y casi completo

Dado que el elemento menor reside en la raíz, su supresión consiste en eliminar ésta; ahora bien, el resultado no presenta una estructura arborescente y, por ello, es necesario reestructurar el árbol. Para formar un nuevo árbol casi completo simplemente se mueve el último elemento v del árbol en un recorrido por niveles hasta la raíz. Sin embargo, normalmente esta nueva raíz no es más pequeña que sus hijos, lo cual obliga también aquí a reestructurar el árbol con una estrategia muy similar: se va intercambiando v con uno de sus hijos hasta que cumple la relación de orden, o vuelve a ser una hoja. Sólo es necesario notar que, en caso de que los dos hijos sean más pequeños que v , en el intercambio intervendrá el menor de ellos; de lo contrario, el nuevo padre no cumpliría la relación de orden requerida con el hijo no movido. De la misma manera que en la inserción, y por el mismo razonamiento, el coste asintótico de la operación es $\Theta(\log n)$ en el caso peor, y $\Theta(1)$ en el caso mejor.

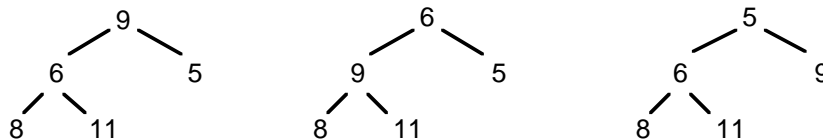


Fig. 4.34: supresión del menor en el árbol de la fig. 4.39, izquierda: se sube el 9 a la raíz (izquierda); se intercambia, erróneamente, el 9 con el 6, violando la relación de orden (centro). En realidad, pues, se intercambia con el menor de los hijos (derecha).

Como queda claro a partir de la explicación de los algoritmos de las operaciones sobre colas prioritarias, dado un árbol parcialmente ordenado es necesario acceder rápidamente a la raíz y al último elemento en un recorrido por niveles; a la raíz, porque en ella reside el elemento menor y al último, porque es el elemento afectado al insertar o borrar. Además, también

queda claro que, a causa de los procesos de reestructuración del árbol, a partir de un nodo es necesario acceder a sus hijos y a su padre. Finalmente, recordemos que el árbol parcialmente ordenado siempre será casi completo. Precisamente, los dos últimos hechos apuntan a una representación secuencial del árbol mediante un montículo. Recordemos que los montículos (v. el apartado 4.2.1) organizan los nodos del árbol dentro de un vector, lo que permite el acceso a los hijos y al padre de un nodo determinado sin necesidad de mantener encadenamientos, sino aplicando una fórmula. En el caso de los árboles binarios, dado un nodo que resida en la posición i del montículo, el padre residirá en la posición $\lfloor i/2 \rfloor$, el hijo izquierdo en la $2i$ y el hijo derecho en la $2i + 1$. Además, el aprovechamiento óptimo del espacio en un montículo se da cuando el árbol que se quiere representar es casi completo, como es el caso que nos ocupa, porque así no hay posiciones intermedias vacías dentro del vector. No obstante, en alguna situación concreta puede considerarse el uso de representaciones encadenadas. En el resto del apartado se estudia principalmente la representación por montículo; la representación encadenada se menciona brevemente al final del apartado y se propone en el ejercicio 4.15.

El universo se presenta en la fig. 4.35. Notemos que en la cabecera aparece el universo de caracterización VAL_NAT , que define una constante natural que representa el número máximo de elementos que caben en la cola, motivo por el que la especificación de las colas no es exactamente la que aparece en la fig. 4.30, sino que también sería necesario controlar el tamaño (de manera similar a todos los TAD ya estudiados). En la representación sólo se necesita el vector para guardar los elementos y un apuntador al primer sitio libre; la raíz estará siempre en la primera posición del vector y así no es necesario ningún apuntador explícito. El invariante de la representación garantiza que la relación de orden entre los nodos se cumple. Por lo que respecta a la codificación de las operaciones del tipo, la creación y la obtención del menor son inmediatas, y la inserción y la supresión del menor siguen la casuística que se acaba de dar, tal como establecen los invariantes esbozados de manera informal. Notemos que, efectivamente, el coste de estas dos últimas operaciones queda logarítmico, dado que los intercambios son de orden constante, y que, además, en realidad no se intercambian los elementos a cada paso, sino que el elemento en proceso de ubicación no se mueve hasta conocer la posición destino, lo que ahorra la mitad de los movimientos (aunque no se mejora el coste asintótico). Eso sí, no se comprueba explícitamente la inexistencia de elementos con la misma prioridad.

universo COLA_PRIORITARIA_POR_MONTÍCULO(ELEM_<, VAL_NAT) es
implementa COLA_PRIORITARIA(ELEM_<, VAL_NAT)
usa BOOL
tipo colapr es tupla
 A es vector [de 1 a val] de elem
 sl es nat
 ftupla
ftipo
invariante (C es colapr): $1 \leq C.sl \leq val+1 \wedge es_cola(C.A, 1, C.sl-1)$,
 siendo el predicado $es_cola(A, i, j)$ definido como:
 $[2r > s] \Rightarrow es_cola(A, r, s) = \text{cierto}$
 $[2r = s] \Rightarrow es_cola(A, r, s) = A[r] < A[2r]$
 $[2r < s] \Rightarrow es_cola(A, r, s) = (A[r] < A[2r]) \wedge (A[r] < A[2r+1]) \wedge$
 $es_cola(A, 2r, s) \wedge es_cola(A, 2r+1, s)$

función crea devuelve colapr es
var C es colapr fvar
 C.sl := 1
devuelve C

 $\{\mathcal{P} \equiv \text{no existe ningún elemento en } C \text{ con la misma prioridad que } v\}$
función inserta (C es colapr; v es elem) devuelve colapr es
var k es nat; fin? es bool fvar
 si C.sl = val+1 entonces error {cola llena}
 sino {se busca la posición destino de v y se mueven los elementos afectados}
 C.sl := C.sl+1 {sitio para el nuevo elemento}
 k := C.sl-1; fin? := falso
 mientras $\neg fin? \wedge (k > 1)$ hacer
 $\{I \equiv es_cola(C.A, 2k, C.sl-1)\}$
 si $v < C.A[k \text{ div } 2]$ entonces $C.A[k] := C.A[k \text{ div } 2]$; k := k div 2
 si no fin? := cierto
 fsi
 fmientras
 C.A[k] := v {inserción del nuevo elemento en su sitio}
 fsi
devuelve C

función menor (C es colapr) devuelve elem es
var res es elem fvar
 si C.sl = 1 entonces error si no res := C.A[1] fsi
devuelve res

Fig. 4.35: implementación de las colas prioritarias usando un montículo.

```

función vacía? (C es colapr) devuelve bool es
devuelve C.sl = 1

función borra (C es colapr) devuelve colapr es
var k, hmp son nat; éxito? es bool fvar
  si C.sl = 1 entonces error {cola vacía}
  si no {se busca la posición destino del último y se mueven los elementos afectados}
    k := 1; éxito? := falso
    mientras (k*2 < C.sl)  $\wedge$   $\neg$  éxito? hacer { k*2  $\geq$  C.sl  $\Rightarrow$  C.A[k] es hoja }
      {I  $\equiv$  despreciando el subárbol de raíz k, el árbol es parcialmente ordenado}
      hmp := menor_hijo(C, k)
      si C.A[hmp] < C.A[C.sl-1] entonces C.A[k] := C.A[hmp]; k := hmp
      si no éxito? := cierto
    fsi
  fmientras
  C.A[k] := C.A[C.sl-1]; C.sl := C.sl-1 {inserción del último en su nuevo sitio}
fsi
devuelve C

{Función auxiliar menor_hijo: dado un nodo, devuelve la posición de su hijo
 menor; si sólo tiene uno, devuelve la posición de este único hijo. Como
 precondition, el nodo tiene como mínimo hijo izquierdo}

función privada menor_hijo (C es colapr; k es nat) devuelve nat es
var i es nat fvar
  si k*2+1 = C.sl entonces i := k*2 {sólo tiene hijo izquierdo}
  si no si C.A[k*2] < C.A[k*2+1] entonces i := k*2 si no i := k*2+1 fsi
  fsi
devuelve i

funiverso

```

Fig. 4.35: implementación de las colas prioritarias usando un montículo (cont.).

Por lo que respecta a las versiones recorribles del TAD, no hay ningún problema en el caso de recorridos no ordenados. En cambio, los recorridos ordenados tienen un coste ineludiblemente más que lineal, ya que los elementos en el montículo no pueden obtenerse de manera simple ordenadamente: es preciso ordenarlos o bien obtener y borrar reiteradamente el menor. En el siguiente apartado veremos un algoritmo de ordenación que puede ser de gran ayuda en este contexto.

El concepto de árbol parcialmente ordenado es independiente de si la representación del árbol es secuencial o encadenada. Ya hemos visto que en el caso secuencial el resultado es óptimo, pero puede ser que necesitemos trabajar con una representación encadenada por

los motivos usuales (principalmente TAD completamente abierto con supresiones mediante apuntadores externos, o desconocimiento absoluto del número de nodos de la estructura). En este caso, parece lógico añadir a los nodos un apuntador al padre, para poder realizar de manera cómoda las inserciones y las supresiones, provocando un incremento de espacio en la estructura del orden del número de nodos; de hecho, este apuntador es imprescindible para el caso de TAD completamente abiertos. Si el TAD no es abierto, puede comprobarse que el encadenamiento al padre no es realmente necesario:

- En la inserción, bajamos por el árbol partiendo de la raíz buscando el nodo del que ha de colgar inicialmente el nuevo elemento. El camino a recorrer puede calcularse si mantenemos un contador de elementos de la estructura. Durante este proceso, guardamos en una pila auxiliar todos los nodos encontrados. A continuación, debemos llevar el elemento a la posición que le corresponde, y para ello vamos obteniendo nodos de la pila; estos nodos son los antecesores del nuevo nodo, y son los que se deben irse comparando en el proceso de reubicación.
- En la supresión, bajamos por el árbol partiendo de la raíz buscando el padre del último nodo, para actualizar sus encadenamientos. Una vez encontrado y actualizado el encadenamiento del padre, partiendo de la raíz de nuevo se efectúan los intercambios pertinentes hasta reubicar el nodo.

Como resultado, para mantener los algoritmos de actualización de coste logarítmico, en vez de los encadenamientos al padre en los nodos basta una pila auxiliar en la operación de inserción. Parece una opción recomendable, pues el tamaño de la pila es $\Theta(\log n)$, es decir, del orden del número de niveles del árbol, y así el espacio adicional requerido es menor.

4.4.2 Aplicación: un algoritmo de ordenación

Los algoritmos de ordenación son una de las familias más clásicas de esquemas de programación y han dado lugar a resoluciones realmente brillantes e ingeniosas. Una de ellas es el algoritmo de *ordenación por montículo* (ing., *heapsort*), presentado en 1964 por J.W.J. Williams en "Heapsort (Algorithm 232)", *Communications ACM*, 7(6), que se basa en el uso de colas prioritarias. En él, se recorre la lista a ordenar y sus elementos se insertan en una cola prioritaria, considerando que la prioridad es el valor mismo del elemento; una vez se han insertados todos, se obtienen uno a uno hasta que la cola queda vacía y cada elemento obtenido, que será el menor de los que queden en la cola en aquel momento, se inserta en la lista resultado y se borra de la cola (v. fig. 4.36; en el invariante, la función *elems* devuelve el conjunto de elementos de una lista o cola, y *máximo* el elemento mayor de la lista). Se supone como precondition que la lista no tiene repetidos, para poder usar el TAD de las colas prioritarias tal como se ha definido.

```

{ $\mathcal{P} \equiv$  la lista  $l$  no tiene elementos repetidos}
función heapsort ( $l$  es lista) devuelve lista es
var  $C$  es cola;  $v$  es elem;  $lres$  es lista fvar
    {primer paso: se construye la cola con los elementos de  $l$ }
     $C := COLA\_PRIORITARIA.crea$ 
    para todo  $v$  dentro de  $l$  hacer  $C := COLA\_PRIORITARIA.inserta(C, v)$  fpara todo
        {segundo paso: se construye la lista ordenada con los elementos de  $C$ }
     $lres := LISTA\_INTERÉS.crea$ 
    mientras  $\neg COLA\_PRIORITARIA.vacía?(C)$  hacer
        { $I \equiv elems(lres) \cup elems(C) = elems(l) \wedge menor(C) > máximo(lres) \wedge ordenada(lres)$ }
         $lres := LISTA\_INTERÉS.inserta(lres, COLA\_PRIORITARIA.menor(C))$ 
         $C := COLA\_PRIORITARIA.borra(C)$ 
    fmientras
devuelve  $lres$ 

```

Fig. 4.36: algoritmo de ordenación por montículo.

Al analizar el coste del algoritmo para una lista de n elementos, observamos que cada uno de los bucles se ejecuta n veces, porque en cada vuelta se trata un elemento. El paso k -ésimo del primer bucle y el paso $(n-k+1)$ -ésimo del segundo tienen un coste $\Theta(\log k)$, de manera que el coste total es igual a $\sum k: 1 \leq k \leq n: \Theta(\log k) = \Theta(n \log n)$, que es la cota inferior de los algoritmos de ordenación, asintóticamente hablando; en este caso, el coste así determinado coincide con el cálculo hecho con las reglas habituales, determinado como el producto del número n de iteraciones multiplicado por el coste individual de la operación en el caso peor, $\Theta(\log n)$. No obstante, la cola exige un espacio adicional lineal. Para evitarlo, R.W. Floyd formuló en el mismo año 1964 una variante del método ("Treesort (Algorithm 243)", *Communications ACM*, 7(12)), aplicable en el caso de que la lista esté representada por un vector directamente manipulable desde el algoritmo (sería el típico caso de ordenar un vector y no una lista). El truco consiste en dividir el vector en dos trozos, uno dedicado a simular la cola y el otro a contener parte de la solución (v. fig. 4.37). Dentro de la cola, los elementos se ordenarán según la relación $>$ (inversa de la ordenación resultado), de manera que en la raíz siempre esté el elemento mayor. Es necesario notar que, en la primera fase del algoritmo, en realidad se dispone de varias colas que se van fusionando para obtener la cola (única) final. El algoritmo no exige ninguna estructura voluminosa auxiliar, pero, en cambio, no es en absoluto modular, pues sólo es aplicable sobre representaciones secuenciales.



Fig. 4.37: heapsort: construcción de la cola (izq.) y ordenación de los elementos (der.); la flecha indica el sentido del desplazamiento de la frontera entre las partes del vector.

Para escribir una versión sencilla y fácilmente legible del algoritmo de ordenación, introducimos una acción auxiliar *hunde* (denominada *heapify* en algunos textos en lengua inglesa), por la que *hunde*(*A*, *pos*, *máx*) reubica el elemento *v* residente en la posición *pos* del vector *A*, intercambiando reiteradamente con el mayor de sus hijos, hasta que *v* sea mayor que sus hijos, o bien hasta que *v* sea una hoja; para saber que un elemento es hoja basta con conocer la posición del último elemento por niveles, que será el parámetro *máx*. Como precondition, se supone que *pos* no es hoja y que sus dos hijos (o sólo uno, si no tiene hijo derecho) son subárboles parcialmente ordenados. La acción correspondiente se presenta en la fig. 4.38; notemos su similitud con la función *borra* de las colas con prioridad, que es una particularización. En las pre y postcondiciones se usa un predicado, *elems*, que da como resultado el conjunto de elementos entre dos posiciones dadas del vector, y que se usa para establecer que los elementos del vector son los mismos y, en particular, que los que había en el trozo afectado por la ordenación también se mantienen. Por lo que respecta a *esCola*, se considera el criterio de ordenación inverso que la fig. 4.35.

```

{ $P \equiv 1 \leq pos \leq \lfloor máx/2 \rfloor \leq n \wedge A = A_0 \wedge es\_cola(A, 2pos, máx) \wedge es\_cola(A, 2pos+1, máx)$ }
acción privada hunde (ent/sal A es vector [de 1 a n] de elem; ent pos, máx son nat) es
var hmp es nat; temp es elem; éxito? es bool fvar
    {se busca la posición que ocupará el nuevo elemento y, mientras, se mueven
    los elementos afectados}
    temp := A[pos]; éxito? := falso
    mientras (pos*2 ≤ máx) ∧ ¬éxito? hacer
    {I ≡ despreciando los nodos que cuelgan de pos, el árbol es parcialmente ordenado}
    hmp := mayor_hijo(A, pos, máx) {inverso de menor_hijo, v. fig. 4.35}
    si A[hmp] > temp entonces A[pos] := A[hmp]; pos := hmp si no éxito? := cierto fsi
    fmientras
    C.A[pos] := temp {inserción del elemento que se está reubicando en su sitio}
facción
{ $Q \equiv elems(A, pos, máx) = elems(A_0, pos, máx) \wedge elems(A, 1, máx) = elems(A_0, 1, máx)$ 
  ∧ esCola(A, pos, máx) }, donde se define elems(A, r, s) ≡ { k: r ≤ k ≤ s: A[k] }

```

Fig. 4.38: algoritmo de ubicación de un elemento en un montículo.

Finalmente, en la fig. 4.39 se codifica el algoritmo de ordenación, y en la fig. 4.40 se muestra un ejemplo. Como en la versión anterior, distinguimos dos partes:

- Formación de la cola. Se van ubicando los nodos en un recorrido inverso por niveles del árbol, de manera que dentro del vector residan diversos árboles parcialmente ordenados, y a cada paso se fusionan dos árboles y un elemento en un único árbol. Notemos que este recorrido no trata las hojas, porque ya son árboles parcialmente ordenados.

- Construcción de la solución. A cada paso se selecciona el elemento mayor de la cola (que reside en la primera posición) y se coloca en la posición i que divide el vector en dos partes, de manera que todos los elementos de su izquierda sean más pequeños y todos los de su derecha más grandes; estos últimos, además, habrán sido ordenados en pasos anteriores. A continuación, el elemento que ocupaba previamente la posición i se sitúa como nueva raíz y se reorganiza el árbol resultante.

```

{ $\mathcal{P} \equiv n \geq 1 \wedge A = A_0$ }
acción heapsort (ent/sal  $A$  es vector [de 1 a  $n$ ] de elem) es
var  $i$  es nat; temp es elem fvar
    {primero, se forma la cola}
    para todo  $i$  desde  $(n \text{ div } 2)$  bajando hasta 1 hacer
        { $I \equiv \forall k: i+1 \leq k \leq n: \text{es\_cola}(A, k, n) \wedge \text{elems}(A, 1, \text{máx}) = \text{elems}(A_0, 1, \text{máx})$ }
        hunde( $A, i, n$ )
    fpara todo
        {a continuación, se extraen ordenadamente los elementos}
        para todo  $i$  desde  $n$  bajando hasta 2 hacer
            { $I \equiv \text{es\_cola}(A, 1, i) \wedge \text{ordenado}(A, i+1, n) \wedge$ 
              $\wedge \text{elems}(A, 1, \text{máx}) = \text{elems}(A_0, 1, \text{máx}) \wedge (i < n \Rightarrow A[1] < A[i+1])$ }
            temp :=  $A[1]$ ;  $A[1] := A[i]$ ;  $A[i] := \text{temp}$     {intercambio de los elementos}
            hunde( $A, 1, i-1$ )                            {reorganización del árbol}
        fpara todo
    facción
    { $Q \equiv \text{elems}(A, 1, n) = \text{elems}(A_0, 1, n) \wedge \text{ordenado}(A, 1, n)$ }

```

Fig. 4.39: algoritmo de ordenación por montículo de un vector.

La versión resultante, efectivamente, no precisa de espacio auxiliar y su coste temporal asintótico se mantiene $\Theta(n \log n)$. Es más, si examinamos detenidamente la primera parte del algoritmo, veremos que su coste no es en realidad $\Theta(n \log n)$, sino simplemente $\Theta(n)$. Ello se debe al hecho de que, al dar una vuelta al bucle que hay dentro de *hunde*, *pos* vale como mínimo el doble que en el paso anterior. Por lo tanto, para *pos* entre $n/2$ y $n/4+1$ el bucle (que tiene un cuerpo de coste constante) se ejecuta como mucho una vez, entre $n/4$ y $n/8+1$ como mucho dos, etc. La suma de estos factores queda, para un árbol de k niveles, siendo $k = \lceil \log_2(n+1) \rceil$, igual a $\sum i: 1 \leq i \leq k-1: 2^{i-1}(k-i)$, donde cada factor del sumatorio es un nivel, el valor absoluto de la potencia de 2 es el número de nodos del nivel, y $k-i$ es el número máximo de movimientos de un nodo dentro del nivel. Esta cantidad está acotada por $2n$ y así el coste asintótico resultante queda $\Theta(n)$. Si bien en el algoritmo de ordenación el coste asintótico no queda afectado, puede ser útil que las colas prioritarias ofrezcan una función *organiza* que implemente esta conversión rápida de un vector en una cola, porque puede

reducir el coste de otros algoritmos (esta función tendría que trabajar sobre la representación del tipo y por ello formaría parte del modelo de las colas). Un ejemplo es la modificación de *heapsort* para que, en lugar de ordenar todo el vector, sólo obtenga los k elementos más grandes, $k < n$, en cuyo caso, el coste del algoritmo modificado sería $\Theta(n+k \log n)$, que podría llegar a quedar $\Theta(n)$ si $k \leq n/\log n$. También el algoritmo de Kruskal sobre grafos (v. sección 6.6) se beneficia de esta reducción de coste.

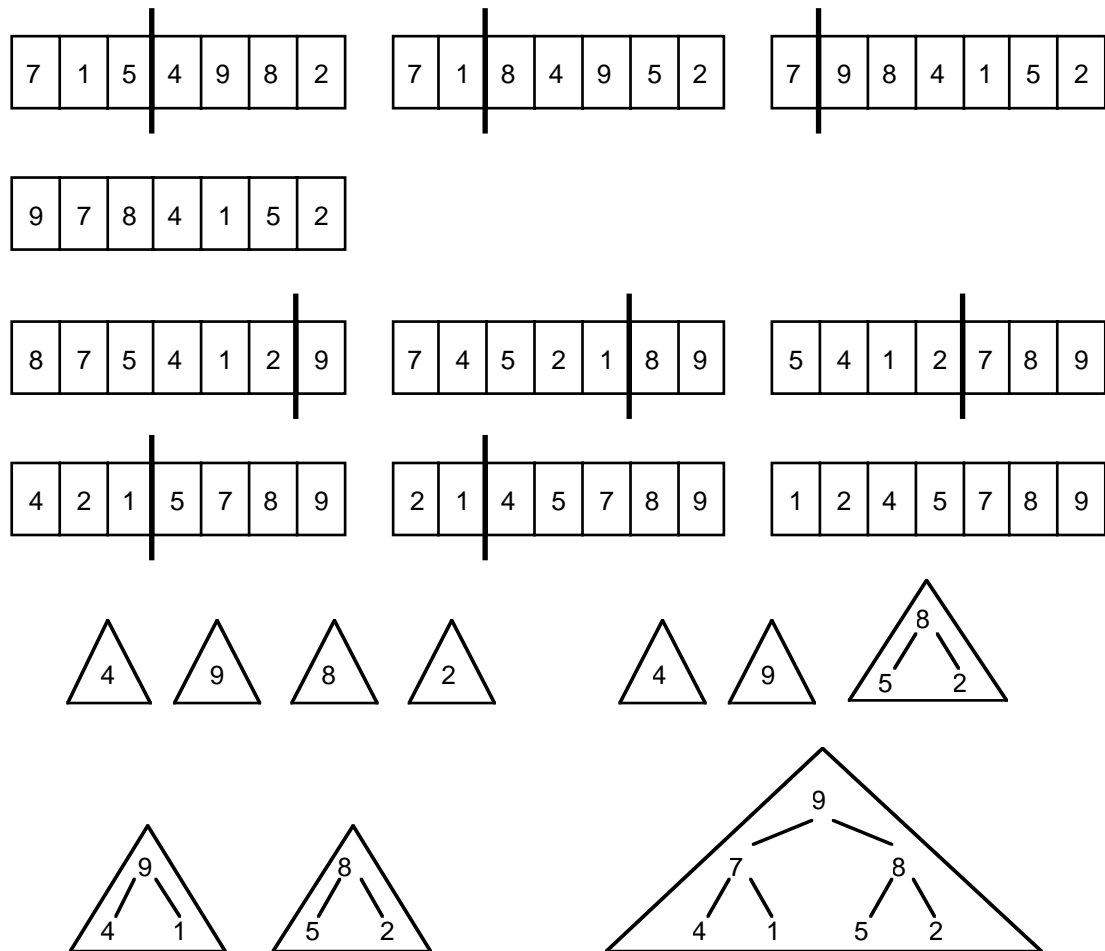
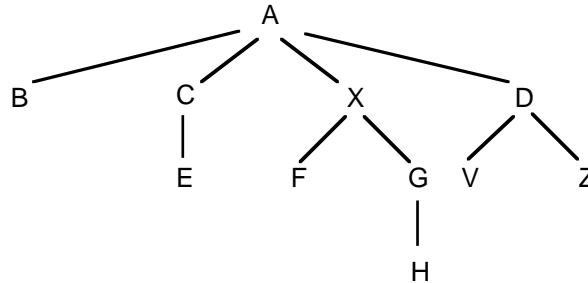


Fig. 4.40: ejemplo de ordenación de un vector con el método del montículo (a) vector: en las dos filas superiores se muestra la formación de la cola y en las dos inferiores la extracción ordenada de elementos; la barra vertical más gruesa indica la partición lógica del vector (b) árboles correspondientes a la formación de la cola; hay un bosque por cada uno de los vectores de la dos primeras filas.

Ejercicios

4.1 Dado el árbol siguiente:



a) describirlo según el modelo asociado a los árboles generales; **b)** decir cuál es el nodo raíz y cuáles son las hojas; **c)** decir qué nodos son padres, hijos, antecesores y descendientes del nodo X; **d)** calcular el nivel y la altura del nodo X; **e)** recorrerlo en preorden, inorden, postorden y por niveles; **f)** transformarlo en un árbol binario que lo represente según la estrategia "hijo izquierdo, hermano derecho".

4.2 Especificar ecuacionalmente el modelo de los árboles con punto de interés tanto binarios como generales, con un conjunto apropiado de operaciones.

4.3 Usando la signatura de los árboles binarios, especificar e implementar una operación que cuente el número de hojas.

4.4 Una expresión aritmética puede representarse como un árbol donde los nodos que son hojas representen operandos y el resto de nodos representen operadores.

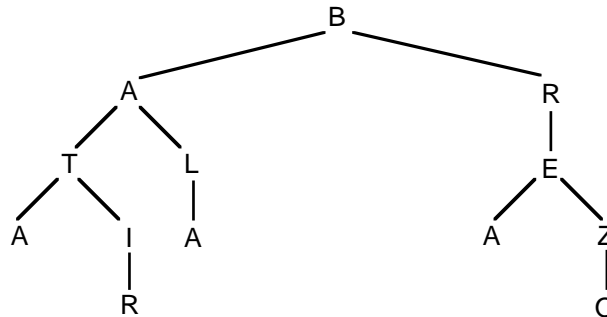
a) Dibujar un árbol representando la expresión $(a+b)*c/d*(a-5)$.

b) ¿Cuál sería el algoritmo de evaluación de la expresión? ¿Se corresponde a algún algoritmo conocido sobre árboles?

c) Escribir un algoritmo que transforme una expresión representada mediante una cola de *símbolos* en una expresión representada mediante un árbol. Suponer que los *símbolos* tienen una operación para averiguar si son operandos, operadores, paréntesis de abrir o paréntesis de cerrar. Considerar las prioridades habituales y la asociatividad por la izquierda. (Una variante de este problema está resuelta en el apartado 7.1.1, usando estructuras lineales.)

4.5 Escribir un algoritmo que transforme un árbol general implementado con apuntadores a los hijos en un árbol binario por la estrategia hijo izquierdo, hermano derecho. Hacer también el algoritmo inverso.

4.6 Sean los árboles generales con la signatura habitual. Implementar un procedimiento que escriba todos los caminos que van de la raíz a las hojas de un árbol de letras. Por ejemplo, del árbol:



han de salir los caminos *BATA*, *BATIR*, *BALA*, *BREA*, *BREZO*.

4.7 Sea un árbol general *A* y un árbol binario *A'* resultado de representar *A* bajo la estrategia hijo izquierdo, hermano derecho. Decir si hay alguna relación entre los recorridos de *A* y *A'*. Justificar la conveniencia de enhebrar o no el árbol binario (v. [Knu68, pp. 361-363]).

4.8 a) ¿Es posible generar un árbol binario a partir únicamente de uno de sus recorridos preorden, inorden o postorden? ¿Y a partir de dos recorridos diferentes (examinar todos los pares posibles)? ¿Por qué?

b) Reconstruir un árbol binario a partir de los recorridos siguientes:

- i) preorden: 2, 1, 4, 7, 8, 9, 3, 6, 5
inorden: 7, 4, 9, 8, 1, 2, 6, 5, 3
- ii) inorden: 4, 6, 5, 1, 2, 12, 7, 3, 9, 8, 11, 10
postorden: 6, 5, 4, 12, 7, 2, 8, 9, 10, 11, 3, 1

c) Diseñar el algoritmo que reconstruya un árbol binario dados sus recorridos preorden e inorden. Hacer lo mismo a partir de los recorridos postorden e inorden. En ambos casos usar el tipo *lista_nodos* para representar los recorridos, definiendo claramente su signatura. Si se necesita alguna operación muy particular de este ejercicio, definirla claramente, especificarla e implementarla.

d) Escribir un algoritmo que, dados dos recorridos preorden, inorden o postorden de un árbol binario y dos nodos suyos cualesquiera, decida si el primero es antecesor del segundo.

4.9 Escribir un algoritmo que transforme un árbol binario representado con apuntadores a los hijos en un árbol binario representado secuencialmente en preorden con apuntador al hijo derecho (es decir, los nodos dentro de un vector almacenados en el orden dado por un recorrido preorden y, para cada uno de ellos, un apuntador adicional al hijo derecho).

4.10 Escribir algoritmos para añadir un nodo como hijo izquierdo o derecho de otro dentro de un árbol binario enhebrado inorden, y borrar el hijo izquierdo o derecho de un nodo.

4.11 Implementar los algoritmos de recorrido directamente sobre la representación secuencial de un árbol, aplicando las ideas del enhebrado inorden.

4.12 Construir un árbol parcialmente ordenado, insertando sucesivamente los valores 64, 41, 10, 3, 9, 1 y 2, y usando $<$ como relación de orden. A continuación, borrar tres veces el mínimo. Mostrar claramente la evolución del árbol paso a paso en cada operación.

4.13 Sea una nueva función sobre las colas prioritarias que borre un elemento cualquiera dada una clave que lo identifique. Implementarla de manera que tenga un coste logarítmico sin empeorar las otras (si es necesario, modificar la representación habitual del tipo).

4.14 Se quieren guardar 10.000 elementos dentro de una cola prioritaria implementada con un montículo que representa el árbol parcialmente ordenado correspondiente. Determinar en los casos siguientes si es mejor un árbol binario o uno cuaternario:

- a) Minimizando el número máximo de comparaciones entre elementos al insertar uno nuevo.
- b) Minimizando el número máximo de comparaciones entre elementos al borrar el mínimo.
- c) Minimizando el número máximo de movimientos de elementos al insertar uno nuevo.
- d) Minimizando el número máximo de movimientos de elementos al borrar el mínimo.

4.15 Implementar un árbol parcialmente ordenado siguiendo la estrategia encadenada.

4.16 Especificar e implementar la función *organiza* mencionada en el algoritmo de ordenación por montículo.

4.17 Especificar e implementar una cola prioritaria que permita la existencia de varios elementos con la misma prioridad. Para hacer honor al nombre del TAD, los elementos con la misma prioridad deben obtenerse y eliminarse en el mismo orden en qué se insertaron.

4.18 Dado un fichero con n números enteros, n muy grande (por ejemplo, $n = 10^8$), construir un algoritmo que obtenga los k nombres más grandes, $k \ll n$ (por ejemplo, $k = 1000$) con la mayor eficiencia posible tanto en tiempo como en espacio. Calcular cuidadosamente su coste.

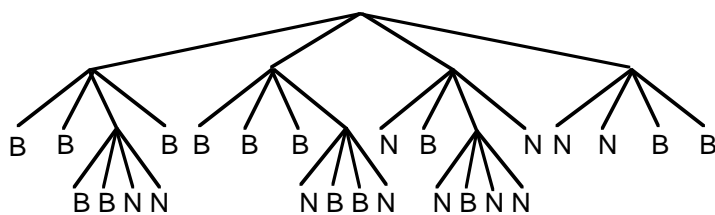
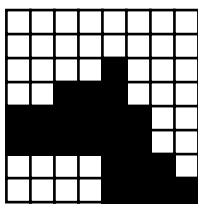
4.19 En lógica, se dice que un predicado (que es una expresión formada por variables y operaciones booleanas; por ejemplo, *cierto*, *falso*, \wedge , \vee y \neg) se satisface si, para alguna asignación de sus variables, el predicado evalúa cierto. A las variables se les pueden asignar los valores *cierto* y *falso* y la evaluación de un predicado es la habitual. Pensar una buena representación de los predicados y un algoritmo que decida si un predicado se satisface. Calcular el coste del algoritmo en tiempo y espacio.

4.20 Nos ocupamos de la construcción de una familia de códigos de compresión de cadenas denominados *códigos de Huffman*. Suponer que tenemos mensajes que consisten en una secuencia de caracteres. En cada mensaje los caracteres aparecen con una probabilidad conocida, independiente de la posición concreta del carácter dentro del mensaje. Por ejemplo, suponer que los mensajes se componen de los caracteres a, b, c, d y e , que aparecen con probabilidades 0.12, 0.40, 0.15, 0.08 y 0.25, respectivamente (obviamente, la suma da 1). Queremos codificar cada carácter en una secuencia de ceros y unos de manera que ningún código de un carácter sea prefijo del código de otro; esta propiedad permite, dado un mensaje, codificarlo y decodificarlo de manera no ambigua. Además, queremos que esta codificación minimice la extensión media esperada de los mensajes, de manera que su compactación sea óptima. En el ejemplo anterior, un código óptimo es $a = 1111$, $b = 0$, $c = 110$, $d = 1110$ y $e = 10$. En concreto:

- a) Diseñar un algoritmo tal que, dado el conjunto V de caracteres y dada la función de probabilidad $pr: V \rightarrow [0,1]$, construya un código de Huffman para V .
- b) Escribir los algoritmos de codificación y decodificación de mensajes suponiendo que el tipo *mensaje* es una instancia adecuada del tipo *cola*.

4.21 La estructura de datos *quad-tree* se usa en informática gráfica para representar figuras planas en blanco y negro. Se trata de un árbol en el cual cada nodo, o bien tiene exactamente cuatro hijos, o bien es una hoja. En este último caso, puede ser una hoja blanca o una hoja negra.

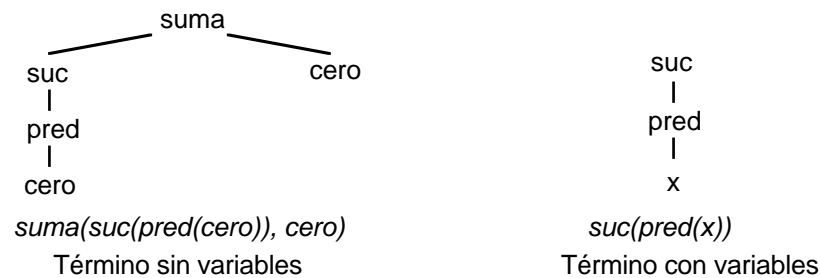
El árbol asociado a una figura dibujada dentro de un plano (que, para simplificar, podemos suponer un cuadrado de lado 2^k) se construye de la forma siguiente: se subdivide el plano en cuatro cuadrantes; los cuadrantes que estén completamente dentro de la figura corresponderán a hojas negras, los que estén completamente fuera de la región, a hojas blancas y los que estén parcialmente dentro y parcialmente fuera, a nodos internos; para estos últimos se aplica recursivamente el mismo algoritmo. Como ejemplo, se muestra una figura en blanco y negro y su árbol asociado (considerando los cuadrantes en el sentido de las agujas del reloj, a partir del cuadrante superior izquierdo):



- a) Determinar la signatura necesaria sobre el tipo *quad-tree* que permita algoritmos para representar una figura como un *quad-tree* y para recuperar la figura a partir de un *quad-tree*. Suponer que existe un tipo *figura* con las operaciones que más convengan.
- b) Escribir una representación para los *quad-trees*.

- c) Implementar las operaciones del tipo obtenidas en a) sobre la representación de b).
- d) Modificar la representación de b) para implementar una nueva operación que obtenga el *negativo* de un *quad-tree*. El resultado ha de ser $\Theta(1)$.

4.22 Ya sabemos que, en el contexto de las especificaciones algebraicas, se define un *término* como la aplicación sucesiva de símbolos de operación de una signatura, que puede tener variables o no tenerlas; gráficamente, un término se puede representar mediante un árbol con símbolos dentro de los nodos.



Consideramos los símbolos definidos dentro de un universo *SÍMBOLO* :

universo *SÍMBOLO* es

usa BOOL

tipo símbolo

ops $op_1, \dots, op_n: \rightarrow \text{símbolo}$

$x_1, \dots, x_k: \rightarrow \text{símbolo}$

$\text{var?}: \text{símbolo} \rightarrow \text{bool}$

$_ = _, _ \neq _: \text{símbolo} \text{ símbolo} \rightarrow \text{bool}$

funiverso

ecns

$\text{var?}(op_1) = \text{falso}; \dots; \text{var?}(x_1) = \text{cierto} \dots$

$(op_1 = op_1) = \text{cierto}; (op_2 = op_2) = \text{falso}; \dots$

$(x \neq y) = \neg (x = y)$

donde *var?* es una función que indica si un símbolo representa una operación.

a) Definir los términos como instancia de árboles. Si, para simplificar, sólo se consideran términos formados con operaciones de aridad 0, 1 ó 2, estos árboles serán binarios.

b) Sean un término *t* sin variables y un término *r* con variables, de modo que en *r* no haya variables repetidas. Diremos que *r* se *superpone con t mediante α* si existe alguna asignación *α* de las variables de *r* que iguale *r* y *t*. Así, los términos $r = \text{suma}(x, \text{cero})$ y $t = \text{suma}(\text{mult}(\text{cero}, \text{cero}), \text{cero})$ se superponen mediante la asignación $x = \text{mult}(\text{cero}, \text{cero})$. En cambio, los términos $t = \text{suma}(\text{mult}(\text{cero}, \text{cero}), \text{cero})$ y $r = \text{mult}(\text{cero}, \text{cero})$ no se superponen. Implementar las operaciones:

superponen?: *término término* $\rightarrow \text{bool}$: siendo *r* un término con variables no repetidas y *t* sin variables, *superponen?*(*t*, *r*) comprueba si *r* se superpone con *t* mediante alguna asignación de sus variables.

as: *término término* $\rightarrow \text{lista_pares_variables_y_términos}$: para *t* y *r* definidos como antes,

$as(t, r)$ devuelve la lista de asignaciones que es necesario hacer a las variables del término r , para que r se superponga con t ; si r no se superpone con t , da error.

c) Sean un término t sin variables y dos términos r y s con variables de modo que ni en r ni en s haya variables repetidas y las variables de s sean un subconjunto de las de r . Si r se superpone con t mediante una asignación α , podemos definir la *transformación de t usando una ecuación $r=s$* como el resultado de aplicar la asignación α sobre s . Así, se puede transformar el término $t = suma(mult(cero, cero), cero)$ usando la ecuación $suma(x, cero) = x$, siendo $\alpha = mult(cero, cero)$, con lo que se obtiene el término $mult(cero, cero)$. Implementar la operación *transforma: término término término \rightarrow término*, que realiza la operación descrita, o da error si el segundo término no se superpone con el primero.

d) Sea un término t sin variables y dos términos r y s con variables de modo que ni en r ni en s haya variables repetidas y las variables de s sean un subconjunto de las de r . Ahora queremos generalizar el apartado c) para que la transformación se pueda realizar sobre cualquier subtérmino t' de t . Así, dado el término $t = suma(mult(cero, cero), cero)$ y la ecuación $r = s: mult(cero, x) = cero$, podemos transformar el subtérmino $t' = mult(cero, cero)$ de t aplicando $r = s$, y obtener como resultado el término $suma(cero, cero)$. Se puede considerar que todo término es subtérmino de sí mismo. Implementar las operaciones:

transformable?: término término término \rightarrow bool: siendo r y s términos con variables y t sin variables, *transformable?(t, r, s)* comprueba si algún subtérmino de t puede transformarse mediante la ecuación $r = s$ (es decir, si r se superpone con algún subtérmino de t).

transfsubt: término término término \rightarrow término: para t, r y s definidos como antes, *transfsubt(t, r, s)* transforma un subtérmino de t mediante la ecuación $r = s$; si no se puede transformar, da error.

e) A continuación se quiere implementar una operación que, dados un término t sin variables y una lista L de ecuaciones de la forma $r = s$, compruebe si puede transformarse algún subtérmino t' de t mediante alguna ecuación de L , según la mecánica descrita en el apartado d). Implementar las operaciones.

se_puede_transformar?: lista_pares_términos término \rightarrow bool: siendo L una lista de ecuaciones y t un término sin variables, *se_puede_transformar?(L, t)* comprueba si algún subtérmino de t puede transformarse mediante alguna ecuación $r = s$ de L .

un_paso: lista_pares_términos término \rightarrow término: para L y t como antes, *un_paso(L, t)* transforma un subtérmino de t mediante alguna ecuación $r = s$ de L ; si no hay ningún subtérmino transformable, da error. Si hay más de una transformación posible, aplica una cualquiera.

f) Implementar la operación *forma_normal: lista_pares_términos término \rightarrow término* que, dado un término t sin variables y una lista L de ecuaciones de la forma $r = s$, transforme t en su forma normal, aplicando sucesivamente ecuaciones de L . Notar que, si se consideran las ecuaciones $r = s$ como reglas de reescritura $r \rightarrow s$, el resultado de este apartado será la implementación del proceso de reescritura.

Capítulo 5 Tablas

Se quiere estudiar el TAD de las funciones $f: K \rightarrow V$ que se aplican sobre un dominio K y dan un resultado sobre un codominio V . A los valores del dominio los llamamos *claves* (ing., *key*; también denominados *índices* o *identificadores*) y a los valores del codominio, *información* o simplemente *valores* (ing., *value*). Si $f(k) = v$, diremos que v es la información *asociada* o *asignada* a k . En el caso general, el dominio K tendrá una cardinalidad muy grande y nuestras implementaciones deberán tratar adecuadamente esta característica. Las funciones que consideramos en este capítulo pueden ser *totales* o *parciales*; en el segundo caso, llamaremos *clave indefinida* a toda clave que no esté en el dominio de la función. Dado este modelo matemático queda claro que, así como las estructuras lineales están orientadas al acceso consecutivo a todos sus elementos (ya sea ordenadamente o no), y los árboles a la estructuración jerárquica de la información, las funciones están orientadas al acceso individual a sus elementos: interesa definir la función en un punto del dominio (es decir, guardar un elemento en la estructura), dejar este punto indefinido (es decir, borrar el elemento) y aplicar la función sobre él (es decir, acceder al elemento dada su clave).

El TAD de las funciones es conocido en el ámbito de las estructuras de la información con el nombre de *tabla* (ing., *lookup table* o *symbol table*, aunque la notación no es totalmente estándar; algunos textos también lo denominan *estructura funcional* o *diccionario*). En este contexto, es frecuente considerar las tablas como conjuntos de pares de clave y de valor, con la restricción de que no haya dos pares con la misma clave, de manera que la clave identifica unívocamente el par. A los pares los llamaremos *elementos definidos* o, simplemente, *elementos* de la tabla.

La aplicación del tipo en el campo de la programación es diversa; por ejemplo, se emplea para implementar tablas de símbolos de compiladores o de sistemas operativos: la clave es el identificador de los símbolos que se almacenan y la información puede ser muy diversa (direcciones de memoria, dimensión, etc.).

En el resto del capítulo se estudia la especificación y la implementación del TAD. Se buscan implementaciones que permitan un acceso individual a los elementos suficientemente eficiente, lo que conseguiremos con las tablas de dispersión y los árboles de búsqueda.

5.1 Especificación

5.1.1 Funciones totales

Sean K y V dos dominios tales que V presenta un valor indefinido que denotaremos por \perp , sea $f: K \rightarrow V$ una tabla y sean $k \in K$ una clave y $v \in V$ un valor. Las operaciones del TAD son:

- Crear la tabla vacía: *crea*, devuelve la función f_\emptyset definida como:
 - $\diamond \text{dom}(f_\emptyset) = K.$
 - $\diamond \forall k: k \in K: f_\emptyset(k) = \perp.$
- Asociar información a una clave: *asigna*(f, k, v), devuelve la función g definida como:
 - $\diamond g(k) = v.$
 - $\diamond \forall k': k' \in K - \{k\}: g(k') = f(k').$
- Separar la información correspondiente a una clave: *borra*(f, k), devuelve la función g definida como:
 - $\diamond g(k) = \perp.$
 - $\diamond \forall k': k' \in K - \{k\}: g(k') = f(k').$
- Consultar el valor asociado a una clave: *consulta*(f, k), devuelve $f(k)$.

Notemos que el modelo aquí presentado se corresponde efectivamente con las funciones totales, ya que toda clave tiene un valor asociado. En caso de que el valor indefinido no se pueda identificar en V , se puede forzar que la función *consulta* devuelva un booleano adicional indicando si la clave estaba o no definida, o bien se puede añadir una operación más, *definida?*, que lo averigüe, de manera que *consulta* o *borra* sobre una clave no definida dé error, y entonces el modelo de las tablas corresponda realmente a las funciones parciales.

En la fig. 5.1 se presenta una especificación parametrizada para el modelo de las funciones totales con la signatura introducida arriba. Sus parámetros formales son: los géneros para las claves y los valores, la igualdad de las claves y el valor indefinido. Por ello, usamos los universos de caracterización *ELEM_* y *ELEM_ESP*: el primero define las claves y el segundo los valores; algunos de estos símbolos se renombran, por motivos de legibilidad¹. Por lo que al resto del universo se refiere, notemos la semejanza con la especificación típica de los conjuntos. Destaquemos que la función *borra* deja la tabla inalterada si la clave no existe, que la operación *consulta* devuelve el valor indefinido si la clave buscada está indefinida y que la operación *asigna*, si la clave ya está definida, le asigna un nuevo valor (otra opción sería dejarla inalterada, o bien que la función *asigna* devolviera un segundo valor tal que, si la clave estuviera definida, devolviera el valor asociado a la clave y dejara la tabla inalterada, o bien asociara a la clave el nuevo valor). Notemos también la ecuación de error que evita la inserción del valor indefinido como pareja de una clave; este valor sólo aparece al consultar una clave sin información asociada.

¹ Los renombramientos de los parámetros formales sólo tienen vigencia en el universo que los efectúa.

universo FUNCION_TOTAL (ELEM_ =, ELEM_ESP) es
renombra ELEM_ =.elem por clave
ELEM_ESP.elem por valor, ELEM_ESP.esp por indef
usa BOOL
tipo tabla
ops crea: \rightarrow tabla
asigna: tabla clave valor \rightarrow tabla
borra: tabla clave \rightarrow tabla
consulta: tabla clave \rightarrow valor
error $\forall t \in \text{tabla}: \forall k \in \text{clave}$
asigna(t, k, indef)
ecns $\forall t \in \text{tabla}; \forall k, k_1, k_2 \in \text{clave}; \forall v, v_1, v_2 \in \text{valor}$
Ecuaciones impurificadoras:
1) $\text{asigna}(\text{asigna}(t, k, v_1), k, v_2) = \text{asigna}(t, k, v_2)$
2) $[k_1 \neq k_2] \Rightarrow \text{asigna}(\text{asigna}(t, k_1, v_1), k_2, v_2) =$
 $\text{asigna}(\text{asigna}(t, k_2, v_2), k_1, v_1)$
Axiomas para *borra*:
1) $\text{borra}(\text{crea}, k) = \text{crea}$
2) $\text{borra}(\text{asigna}(t, k, v), k) = \text{borra}(t, k)$
3) $[k_1 \neq k_2] \Rightarrow \text{borra}(\text{asigna}(t, k_1, v_1), k_2) = \text{asigna}(\text{borra}(t, k_2), k_1, v_1)$
Axiomas para *consulta*:
1) $\text{consulta}(\text{crea}, k) = \text{indef}$
2) $\text{consulta}(\text{asigna}(t, k, v), k) = v$
3) $[k_1 \neq k_2] \Rightarrow \text{consulta}(\text{asigna}(t, k_1, v_1), k_2) = \text{consulta}(t, k_2)$
funiverso
universo ELEM_ESP caracteriza
tipo elem
ops esp: \rightarrow elem
funiverso

Fig. 5.1: especificación del TAD de las funciones totales.

En una instancia de las tablas es necesario asociar símbolos a todos estos parámetros formales; por ejemplo, podemos definir una tabla cuyas claves sean enteros, la información cadenas de caracteres y el valor indefinido una cadena especial, no válida como información normal (algo así como "&%\$#!", por ejemplo).

5.1.2 Conjuntos

Una variante del TAD de las funciones totales consiste en considerar el modelo de los conjuntos de elementos de V , $P(V)$, con operaciones de inserción y supresión individual de elementos y de comprobación de pertenencia, siendo V un dominio que presenta una función $id: V \rightarrow K$, de manera que cada elemento del conjunto esté identificado por una clave que sirva para hacerle referencia. A veces, el identificador será el mismo elemento y entonces V y K serán el mismo género e id será la función identidad, y el resultado será equivalente al universo CJT_{\in} de la fig. 1.29 con el añadido de la supresión. La signatura propuesta para este modelo aparece en la fig. 5.2 y su especificación queda como ejercicio para el lector. Las implementaciones y los algoritmos sobre tablas que aparecen en el resto del capítulo se pueden adaptar a este TAD con muy poco esfuerzo.

<u>universo</u> CONJUNTO (ELEM_CJT) <u>es</u>	<u>universo</u> ELEM_CJT <u>caracteriza</u>
<u>usa</u> BOOL	<u>tipo</u> clave, elem
<u>tipo</u> conjunto	<u>ops</u>
<u>ops</u>	esp: \rightarrow elem
crea: \rightarrow conjunto	id: elem \rightarrow clave
añade: conjunto elem \rightarrow conjunto	$_=_, _ \neq _$: clave clave \rightarrow bool
borra: conjunto clave \rightarrow conjunto	<u>ecns</u> ... las de $_=_$ y $_ \neq _$
consulta: conjunto clave \rightarrow elem	<u>funiverso</u>
está?: conjunto clave \rightarrow bool	
<u>funiverso</u>	

Fig. 5.2: una signatura para el TAD de los conjuntos.

5.1.3 Tablas y conjuntos recorribles

Como es habitual, podemos añadir iteradores a los modelos mencionados y obtener las correspondientes versiones recorribles, ordenadamente o no. Estudiamos a continuación la formulación del modelo de las tablas recorribles.

Optamos por la estrategia habitual de considerar los valores del modelo como pares de valores del modelo recorrible, obteniendo así el modelo $(f: K \rightarrow V) \times (g: K \rightarrow V)$, tales que $\text{dom}(f) \cap \text{dom}(g) = \emptyset$, donde f representa los pares a la izquierda del punto de acceso del iterador y g los pares de la derecha. Si se quiere que el recorrido sea ordenado, lo que es imprescindible para que el modelo sea correcto dentro de la semántica inicial, es necesario requerir que haya una operación de comparación de claves, que denotaremos por $<$, en K . La ordenación exige que el máximo del dominio de la primera función sea menor que el mínimo del dominio de la segunda función. Se definen las operaciones *crea*, *asigna*, *borra* y

consulta propias de las tablas y las operaciones *principio*, *actual*, *avanza* y *final?*, propias de los iteradores. La signature en el caso ordenado se presenta en la fig. 5.3, sin detallar su especificación; las claves se definen en *ELEM_<_* y los valores en *ELEM_ESP*.

En la fig. 5.4, se muestra también una signature para los conjuntos recorribles (no ordenadamente) considerando que la clave es el elemento entero. Notemos que la signature es muy parecida a la de las listas con punto de interés; de hecho, el conjunto recorrible se puede considerar como una lista con punto de interés sin repeticiones, borrando los elementos mediante su clave en vez de suprimir el elemento actual, y con operación de pertenencia.

universo FUNCION_TOTAL_RECORRIBLE_ORD (ELEM_<_, ELEM_ESP) es
usa BOOL
renombra ELEM_<_.elem por clave, ELEM_ESP.elem por valor
instancia PAR(A, B son ELEM) donde A.elem es clave, B.elem es valor
renombra par por elem_tabla
tipo tabla
ops crea: → tabla
 asigna: tabla clave valor → tabla
 borra: tabla clave → tabla
 consulta: tabla clave → valor
 principio, avanza: tabla → tabla
 actual: tabla → elem_tabla
 final?: tabla → bool
funiverso

Fig. 5.3: signature de las tablas recorribles ordenadamente .

universo CJT_RECORRIBLE (ELEM_=) es
usa BOOL
tipo cjt
ops crea: → cjt
 añade, borra: cjt elem → cjt
 está?: cjt elem → bool
 principio, avanza: cjt → cjt
 actual: cjt → elem
 final?: cjt → bool
funiverso

Fig. 5.4: signature de los conjuntos recorribles.

Un modelo igualmente válido de las tablas recorribles consiste en sustituir las operaciones de iterador por una única operación que devuelva directamente la lista de pares clave-valor (ordenada o no). Ya hemos usado esta estrategia para el caso de los árboles. Aunque es evidente que esta especificación tendrá un coste de implementación más elevado (debido al espacio de la lista resultado), en algunas circunstancias puede ser una alternativa adecuada y, por ello, en la fig. 5.5 se introduce la signature correspondiente; además, simplifica el estudio de los recorridos que veremos. Notemos las instancias de los universos de los pares y las listas (públicas, porque los usuarios del tipo puedan servirse de ellas).

```

universo FUNCION_TOTAL_RECORRIBLE_ORD(ELEM_<_ =, ELEM_ESP) es
  renombra ELEM_<_ =.elem por clave, ELEM_ESP.elem por valor
  instancia PAR(A, B son ELEM) donde A.elem es clave, B.elem es valor
  renombra par por elem_tabla, _.c1 por _.clave, _.c2 por _.valor
  instancia LISTA(C es ELEM) donde C.elem es elem_tabla
  renombra lista por lista_elem_tabla
  tipo tabla
  ops crea: → tabla
    asigna: tabla clave valor → tabla
    borra: tabla clave → tabla
    consulta: tabla clave → valor
    todos_ordenados: tabla → lista_elem_tabla
funiverso

```

Fig. 5.5: signature del TAD de las tablas recorribles ordenadamente que devuelve una lista.

5.2 Implementación

Estudiamos en esta sección las posibilidades para implementar los TAD de las funciones y de los conjuntos, buscando un acceso individual de orden constante. Tomaremos como TAD de referencia el de las funciones totales. Comenzamos usando diversas variantes de listas hasta llegar a las tablas de dispersión, que son de la máxima eficiencia para las operaciones del tipo.

5.2.1 Implementación por listas desordenadas

Consiste en organizar una lista no ordenada cuyos elementos sean las claves definidas con su valor asociado. El coste asintótico de las operaciones del TAD en el caso peor es $\Theta(n)$ (cuando el elemento involucrado en la operación no está en la tabla), frente a otros esquemas logarítmicos o constantes que se introducen a continuación. No obstante, su simplicidad hace posible su elección, para n pequeña, o si no hay restricciones de eficiencia.

Es interesante conocer un par de estrategias útiles en determinadas circunstancias (aunque no reducen el coste asintótico):

- Si se dispone *a priori* de las probabilidades de consulta de cada clave y, a partir de un momento dado, la frecuencia de las actualizaciones es baja comparada con la de las consultas, se pueden mantener ordenadas las claves según estas probabilidades dentro de la lista, y el tiempo de localización del elemento queda $\sum_{x: 1 \leq x \leq n} x \cdot p_x$, donde p_x representa la probabilidad de consulta de la x -ésima clave más probable.
- En las búsquedas con éxito puede reorganizarse la lista para que los elementos que se consulten queden cerca del inicio, con la suposición de que las claves no son equiprobablemente consultadas y que las más consultadas hasta al momento son las que se consultarán más en el futuro; es decir, se tiende precisamente a una lista ordenada por probabilidades. Esta estrategia se conoce con el nombre de *búsqueda secuencial autoorganizativa* (ing., *self-organizing sequential search*) y a las listas correspondientes se las llama *lista autoorganizativas* (ing., *self-organizing lists*) y hay varias implementaciones posibles. Entre ellas, destacamos la denominada *búsqueda con movimiento al inicio* (ing., *move to front*), en la que el elemento consultado se lleva al inicio de la lista, y la denominada *búsqueda por transposición* (ing., *transpose*), donde se intercambia el elemento consultado con su predecesor. En general, el segundo método da mejores resultados que el primero cuando las probabilidades de consulta de las diversas claves no varían en el tiempo (los saltos de los elementos dentro la lista son más cortos, con lo que su distribución es más estable); por otro lado, si la distribución es muy sesgada el primer método organiza los elementos más rápidamente. Notemos también que, en el caso de representar las listas secuencialmente, los desplazamientos conllevan los problemas habituales, especialmente en el caso de los TAD abiertos.

5.2.2 Implementación por listas ordenadas

Aplicable sólo si las claves presentan una operación $<$ de comparación que defina un orden total, es una organización útil cuando las consultas predominan sobre las actualizaciones, porque aunque estas últimas queden $\Theta(n)$, las primeras reducen su coste. También pueden usarse si la tabla tiene dos estados bien diferenciados en el tiempo, el primero en el que predominen las actualizaciones y el segundo donde predominen las consultas; entonces se puede mantener la lista desordenada durante el primer estado y aplicar un algoritmo de ordenación de coste $\Theta(n \log n)$ antes de comenzar el segundo. En todo caso, aparte del coste temporal, es necesario recordar que, si la implementación elegida para las listas es secuencial, las actualizaciones exigen movimientos de elementos. A continuación, citamos los tres esquemas de búsqueda habituales.

a) Lineal (secuencial)

Ya presentado en la sección 3.3, consiste en recorrer la lista desde el principio hasta encontrar el elemento buscado o llegar al final. La búsqueda sin éxito acaba antes que en el caso desordenado, exigiendo en media $(n+1)/2$ accesos al vector, aunque el coste asintótico en el caso peor sigue siendo lineal ².

b) Dicotómica

También conocida como *búsqueda binaria* (ing., *dichotomic* o *binary search*), es aplicable sólo cuando la lista se representa secuencialmente dentro de un vector y consiste en descartar sucesivamente la mitad de las posiciones del vector hasta encontrar el elemento buscado, o bien acabar. Para ello, a cada paso se compara el elemento buscado con el elemento contenido en la posición central del trozo de vector en examen; si es más pequeño, se descartan todos los elementos de la derecha, si es más grande, los de la izquierda, y si es igual, ya se ha encontrado.

En la fig. 5.6 se da una versión iterativa del algoritmo de búsqueda dicotómica. Suponemos que las tablas se representan por un vector A dimensionado de uno a un máximo predeterminado y un apuntador sl de sitio libre; suponemos también que cada posición del vector es una tupla de clave y valor. Los apuntadores izq y der delimitan el trozo del vector en examen, tal como establece el invariante; debemos destacar, sin embargo, que para que el invariante sea correcto es necesario imaginar que las posiciones 0 y sl de la tabla están ocupadas por dos elementos fantasma tales que sus claves son, respectivamente, la clave más pequeña posible y la clave más grande posible dentro del dominio de las claves. El análisis del algoritmo revela que el coste de la consulta es, pues, $\Theta(\log n)$ en el caso peor y $\Theta(1)$ en el caso mejor. Existe otra versión del algoritmo que se ha presentado en el ejercicio 2.7, que no corta la búsqueda en caso de éxito de manera que el coste es $\Theta(\log n)$ en todos los casos; a cambio, los bucles son más simples y su ejecución más rápida.

c) Por interpolación

También para listas representadas secuencialmente, la *búsqueda por interpolación* (ing., *interpolation search*; también conocida como *estimated entry search*) sigue la misma estrategia que la anterior con la diferencia de que, en vez de consultar la posición central, se consulta una posición que se interpola a partir del valor de la clave que se quiere buscar y de los valores de la primera y la última clave del trozo del vector en examen. Si suponemos que las claves se distribuyen uniformemente entre sus valores mínimo y máximo, la búsqueda queda $\Theta(\log \log n)$ en el caso mejor, con la función de interpolación F :

$$F(A, k, i, j) = \lfloor \{ (k - A[i]) / (A[j] - A[i]) \} \cdot (j - i) \rfloor + i$$

² Hay un algoritmo que asegura un coste $O(\sqrt{n})$ en el caso medio, pero que queda $\Omega(n)$ en el caso peor, que se basa en la utilización de campos de encadenamiento adicionales, que permiten efectuar saltos más largos.

```

función consulta (t es tabla; k es clave) devuelve valor es
var izq, der, med son nat; encontrado es bool; v es valor fvar
    izq := 0; der := t.sl; encontrado := falso
    mientras (der - izq > 1)  $\wedge$   $\neg$  encontrado hacer
        {  $I \equiv 0 \leq \text{izq} < \text{der} \leq \text{t.sl} \wedge \text{t.A}[\text{izq}].k < k < \text{t.A}[\text{der}].k$ 
           $\wedge$  encontrado  $\Rightarrow \text{t.A}[\text{med}].k = k$  }
        med := (izq + der) div 2
    opción
        caso t.A[med].k = k hacer encontrado := cierto
        caso k < t.A[med].k hacer der := med
        caso t.A[med].k < k hacer izq := med
    fopción
    fmientras
    si encontrado entonces v := t.A[med] si no v := indefinido fsi
devuelve v

```

Fig. 5.6: algoritmo de búsqueda dicotómica.

Si la distribución no es uniforme, aunque teóricamente se podría corregir la desviación para mantener este coste bajo (conociendo *a priori* la distribución real de las claves), en la práctica sería complicado. Por ejemplo, si $\forall p: 1 \leq p \leq n-1: A[p] = p$, y $A[n] = \infty$, en el caso de buscar el elemento $n-1$, el coste resultante es lineal. Para evitar este riesgo, en [GoB91, pp. 42-43] se propone combinar un primer paso de acceso a la tabla usando el algoritmo de interpolación y, a continuación, buscar secuencialmente en la dirección adecuada.

Notemos que la búsqueda por interpolación dentro de una lista sólo funciona cuando las claves son numéricas (o tienen una interpretación numérica), y no están repetidas (como es el caso que aquí nos ocupa, en el que la lista representa una tabla).

5.2.3 Implementación por vectores de acceso directo

Una mejora evidente del coste de las operaciones del TAD es implementar la tabla sobre un vector de manera que cada clave tenga asociada una y sólo una posición del vector. En otras palabras, dada una función $g: K \rightarrow Z_n$, donde Z_n representa el intervalo cerrado $[0, n-1]$, y siendo n la cardinalidad del conjunto de claves, la tabla se puede representar con un vector A tal que, dada una clave k , en $A[g(k)]$ esté el valor asociado a k . Si la clave está indefinida, en $A[g(k)]$ residirá el valor indefinido o, si no existe ningún valor que pueda desempeñar este papel, en cada posición añadiremos un campo booleano adicional que marque la indefinición. Con esta representación, es obvio que todas las operaciones se implementan en orden constante en cualquiera de los casos.

La función g ha de ser inyectiva como mínimo; mejor aún si es biyectiva, porque en caso contrario habrá posiciones del vector que siempre quedarán sin usar. Incluso en el último caso, sin embargo, el esquema es impracticable si n es muy grande y el porcentaje de claves definidas es muy pequeño (lo que implica que la mayoría de las posiciones del vector estarán sin usar), que es lo más habitual.

5.2.4 Implementación por tablas de dispersión

Las *tablas de dispersión* (ing., *hashing table* o también *scatter table*; *to hash* se traduce por "desmenuzar"), también conocidas como *tablas de direccionamiento calculado*, se parecen al esquema anterior en que utilizan una función $h: K \rightarrow Z_r$ que asigna un entero a una clave, pero ahora sin requerir que h sea inyectiva, lo que divide el dominio de las claves en r clases de equivalencia y quedan dentro de cada clase todas las claves tales que, al aplicarles h , dan el mismo valor. A h la llamamos *función de dispersión* (ing., *hashing function*), y los valores de Z_r son los *valores de dispersión* (ing., *hashing values*). Lo más importante es que, a pesar de la no-inyectividad de h (que permite aprovechar mucho más el espacio, porque sólo se guardan los elementos definidos de la tabla), el coste de las operaciones se puede mantener constante si se siguen unas reglas de diseño básicas y no tenemos mala suerte (ya veremos qué queremos decir con esto...).

Se usa $h(k)$ para acceder a un vector donde residan las claves definidas junto con su valor (más otra información de gestión de la tabla de dispersión que ya introduciremos). Como no se requiere que h sea inyectiva, se define el vector con una dimensión más pequeña que en el esquema anterior y se asigna cada clase de equivalencia a una posición del vector. A las posiciones del vector algunos autores las llaman *cubetas* (ing., *bucket*); esta terminología se usa sobre todo al hablar de dispersión sobre memoria secundaria, donde en lugar de vectores se usan ficheros.

Quedan dos cuestiones por resolver:

- ¿Qué forma toma la función de dispersión? Ha de cumplir ciertas propiedades, como distribuir bien las claves y ser relativamente rápida de calcular.
- ¿Qué pasa cuando en la tabla se quieren insertar claves con idéntico valor de dispersión? Si primero se inserta una clave k , tal que $h(k) = i$, y después otra k' , tal que $h(k') = i$, se dice que se produce una *colisión* (ing., *collision*), porque k' encuentra ocupada la posición del vector que le corresponde; a k y k' los llamamos *sinónimos* (ing., *synonym*). El tratamiento de las colisiones da lugar a varias estrategias de implementación de las tablas de dispersión.

5.3 Funciones de dispersión

En esta sección se determinan algunos algoritmos que pueden aplicarse sobre una clave para obtener valores de dispersión. Comencemos por establecer las propiedades que debe cumplir una buena función de dispersión:

- Distribución uniforme. Todos los valores de dispersión deben tener aproximadamente el mismo número de claves asociadas dentro de K , es decir, la partición inducida por la función debe dar lugar a clases de equivalencia de tamaño parecido.
- Independencia de la apariencia de la clave. Pequeños cambios en la clave han de resultar en cambios absolutamente aleatorios del valor de dispersión; además, las variaciones en una parte de la clave no han de ser compensables con variaciones fácilmente calculables en otra parte.
- Exhaustividad. Todo valor de dispersión debe tener como mínimo una clave asociada, de manera que ninguna posición del vector quede desaprovechada *a priori*.
- Rapidez de cálculo. Los algoritmos han de ser sencillos y, si es necesario, programados directamente en lenguaje máquina o bien en lenguajes como C, que permiten manipulación directa de bits; las operaciones necesarias para ejecutar los algoritmos deben ser lo más rápidas posible (idealmente, desplazamientos o rotaciones de bits, operaciones lógicas y similares).

Es necesario señalar que la construcción de computadores cada vez más potentes relativiza la importancia del último criterio al diseñar la función; ahora bien, sea cual sea la potencia de cálculo del computador parece seguro que operaciones como la suma, los desplazamientos y los operadores lógicos serán siempre más rápidas que el producto y la división, por ejemplo, y por esto seguimos considerando la rapidez como un parámetro de diseño.

Por otro lado, y este es un hecho fundamental para entender la estrategia de dispersión, las dos primeras propiedades se refieren al dominio potencial de claves K , pero no aseguran el buen comportamiento de una función de dispersión para un subconjunto A de claves, siendo $A \subseteq K$, porque puede ocurrir que muchas de las claves de A tengan el mismo valor de dispersión y que algunos de estos valores, por el contrario, no tengan ninguna antiimagen en A . Precisamente por esto, una función de dispersión teóricamente buena puede dar resultados malos en un contexto de uso particular (y esto es lo que queríamos decir con "mala suerte"). Si se considera que este problema es grave, en las implementaciones de tablas que se introducen en la próxima sección se puede añadir código para controlar que el número de colisiones no sobrepase una cota máxima. En todo caso, esta propiedad indeseable de la dispersión siempre debe ser tenida en cuenta al diseñar estructuras de datos.

Es natural preguntarse, antes de empezar a estudiar estrategias concretas, qué posibilidades existen de encontrar buenas funciones de dispersión. Si se quiere almacenar n claves en una tabla de dimensión r , hay r^n posibles configuraciones, cada una de ellas obtenida mediante una hipotética función de dispersión; si $n < r$, sólo $r! / (r - n)!$ de estas funciones no provocan ninguna colisión. Para ilustrar el significado de esta fórmula, D.E. Knuth presenta la "paradoja del cumpleaños": si acuden veintitrés personas a una fiesta, es más probable que haya dos de ellas que celebren su cumpleaños el mismo día que no lo contrario [Knu73, pp. 506-507]. Por ello, lo que pretendemos no es buscar funciones que no provoquen ninguna colisión sino funciones que, en el caso medio, no provoquen muchas³. Debe tenerse en cuenta, además, que es teóricamente imposible generar datos aleatorios a partir de claves no aleatorias, pero lo que se busca aquí es una buena simulación, lo que sí que es posible.

Analizaremos las funciones de dispersión según dos enfoques diferentes:

- Considerando las funciones $h: K \rightarrow Z_r$ como la composición de dos funciones diferentes, $f: K \rightarrow Z$ y $g: Z \rightarrow Z_r$.
- Considerando directamente las funciones $h: K \rightarrow Z_r$.

Asimismo, a partir de ahora consideraremos que las claves son, o bien enteros dentro de un intervalo cualquiera, o bien cadenas de caracteres (las cadenas de caracteres las denotaremos por C^* , siendo C el conjunto válido de caracteres de la máquina), porque es el caso usual. A veces, no obstante, nos encontramos con que las claves han de considerarse como la unión o composición de diversos campos, y es necesario adaptar los algoritmos aquí vistos a este caso, sin que generalmente esto nos cause mayores problemas. Cuando las claves sean numéricas, la función h será directamente igual a g .

5.3.1 Funciones de traducción de cadenas a enteros

Normalmente, las funciones $f: C^* \rightarrow Z$ interpretan cada carácter de la cadena como un número, y se combinan todos los números así obtenidos mediante algún algoritmo. Estos números pueden ser naturales entre 0 y 127 ó 255, si se utiliza el código ASCII (o ASCII extendido) o similares, o bien números de 16 bits en caso de utilizar el código UNICODE.

El primer problema que se presenta es que seguramente las claves no usarán todos los caracteres del código. Por ejemplo, imaginemos el caso habitual en que las cadenas de

³ Si la tabla es estática y las claves se conocen *a priori*, hay técnicas que permiten buscar funciones que efectivamente no generen sinónimos; son las denominadas técnicas de *dispersión perfecta* (ing., *perfect hashing*), que no estudiamos aquí. Se puede consultar el artículo de T.G. Lewis y C.R. Cook "Hashing for Dynamic and Static Internal Tables" (publicado en el *Computer IEEE*, 21(10), 1988, es un resumen de la técnica de dispersión en general) para una introducción al tema y [Meh84] para un estudio en profundidad.

caracteres representan nombres (de personas, de lugares, etc.); es evidente que algunos caracteres válidos del código de la máquina no aparecerán nunca (como '&', '%', etc.), por lo que su codificación numérica no se usará. Por ejemplo, si el código de la máquina es ASCII estándar (con valores comprendidos en el intervalo $[0, 127]$) y en los nombres sólo aparecen letras y algunos caracteres especiales más (como '.' y '-'), las cadenas estarán formadas por un conjunto de 54 caracteres del código (suponiendo que se distingan mayúsculas de minúsculas); es decir, más de la mitad del código quedará desaprovechado y por ello los valores generados no serán equiprobables (es más, puede haber valores inalcanzables). Para evitar una primera pérdida de uniformidad en el tratamiento de la clave es conveniente efectuar una traducción de códigos mediante una función $\phi: C \rightarrow [1, b]$, siendo b el número diferente de caracteres que pueden formar parte de las claves, de manera que la función de conversión f se define finalmente como $f(k) = f'(\Phi(k))$, siendo Φ la extensión de ϕ a todos los caracteres de la clave y f' una función $f': [1, b]^* \rightarrow \mathbb{Z}$. Esta conversión será más o menos laboriosa según la distribución de los caracteres dentro del código usado por la máquina.

A continuación, se definen algunas funciones f aplicables sobre una clave $k = c_n \dots c_1$:

- Suma de todos los caracteres. Se define como $f(k) = \sum_{i: 1 \leq i \leq n} \phi(c_i)$. Empíricamente se observa que los resultados de esta función son poco satisfactorios, fundamentalmente porque el valor de un carácter es el mismo independientemente de la posición en que aparezca.
- Suma ponderada de todos los caracteres. Modifica la fórmula anterior considerando que los caracteres tienen un peso asociado que depende de la posición donde aparecen y queda: $f(k) = \sum_{i: 1 \leq i \leq n} \phi(c_i) \cdot b^{i-1}$. Los resultados son espectacularmente mejores que en el método anterior y por ello esta función es muy usada, aunque presenta dos problemas que es necesario conocer:
 - ◊ Es más ineficiente, a causa del cálculo de una potencia y del posterior producto a cada paso del sumatorio. Pueden aplicarse diversos trucos para reducir este coste:
 - * La potencia se puede ahorrar introduciendo un vector auxiliar T tal que $T[i] = b^{i-1}$, que se inicializará una única vez al principio del programa y que generalmente será lo suficientemente pequeño dentro del contexto de los datos del programa como para despreciar el espacio que ocupa y el tiempo de dicha inicialización. Eso sí, el dimensionamiento del vector exige fijar el número máximo de caracteres que puede tener la clave.
 - * Esta idea puede generalizarse mediante un vector bidimensional M de b filas tal que $M[x, i] = x \cdot b^{i-1}$. Es decir, el cálculo de cada término del sumatorio se traduce en un acceso al vector, y ya sólo nos queda sumar los resultados parciales. El vector es ahora más voluminoso y por ello debe evaluarse cuidadosamente su conveniencia, tanto por el espacio ocupado como por el coste de inicialización del vector.

* Otra opción consiste en elevar, no b , sino la menor potencia de 2 más grande que b , de manera que los productos y las potencias puedan implementarse como operaciones de desplazamiento de bits; los resultados de esta variante no son tan malos como se podría temer (dependiendo de la diferencia entre b y la potencia de 2). Es necesario controlar, no obstante, que los desplazamientos no provoquen apariciones reiteradas de ceros dentro de los sumandos parciales ni apariciones cíclicas de la misma secuencia de números.

◊ Se puede producir un desbordamiento en el cálculo, que es necesario detectar antes que se produzca; por ejemplo, si b vale 26, en una máquina que use 32 bits para representar los enteros puede producirse desbordamiento a partir del octavo carácter. Una solución consiste en ignorar los bits que se van generando fuera del tamaño de la palabra del computador, pero esta estrategia ignora cierta cantidad de información reduciendo pues la aleatoriedad de la clave; además, la detección del desbordamiento puede incrementar considerablemente el tiempo de ejecución de la función.

Una alternativa es dividir la clave de entrada en m trozos de s caracteres (excepto uno de ellos si el resto de la división m / s es diferente de cero) aplicando entonces la fórmula de la suma ponderada sobre cada uno de estos trozos y combinando los resultados parciales mediante sumas. La magnitud s debe verificar que el cálculo de cada trozo por separado no provoque desbordamiento, y tampoco su suma final. Es decir, s es el mayor entero que cumple $m * (\sum_{i: 1 \leq i \leq s} b^i) \leq \text{maxent}$, siendo maxent el mayor entero representable en la máquina, y siendo el sumatorio el valor más grande que puede dar la Φ función aplicada sobre una cadena de s caracteres (s apariciones del carácter con valor de conversión más grande); por ejemplo, si las claves son letras minúsculas y la función ϕ asigna valores en orden alfabético, la clave que maximiza el valor de este sumatorio es "zz...z" (s apariciones del carácter 'z'). Esta solución puede combinarse con el uso de vectores comentado en el punto anterior con la ventaja que ahora el número de posiciones del vector (o de columnas en el segundo caso) está acotado no por el número de caracteres de la clave sino por s ; además, podemos asegurar que el valor teórico de las posiciones del vector seguro que es representable (es decir, menor o igual que maxent), cosa que antes no podía aseverarse.

- Cualquier otra variante del método anterior hecha a partir de estudios empíricos, cuando se tiene una idea aproximada de las distribuciones de los datos. Por ejemplo, en el artículo "Selecting a Hashing Algorithm", de B.J. McKenzie, R. Harries y T. Bell, *Software-Practice and Experience*, 20(2), 1990, se presentan algunas funciones de dispersión usadas en la construcción de compiladores, que utilizan otros valores para ponderar los caracteres (potencias de dos, que son rápidas de calcular o números primos, que distribuyen mejor).

5.3.2 Funciones de restricción de un entero en un intervalo

En la literatura sobre el tema se encuentran gran cantidad de funciones; destacamos tres:

- División (ing., *division*). Definida como $g(z) = z \bmod r$, es muy sencilla y rápida de calcular. Evidentemente, el valor r es crucial para una buena distribución cuando los datos de entrada no son uniformes y la función de dispersión debe aleatorizarlos al máximo. Así, por ejemplo, una potencia de 2 favorece la eficiencia, pero la distribución resultante será defectuosa, porque simplemente tomaría los $\log_2 r$ bits menos significativos de la clave. Tampoco es bueno que r sea par, porque la paridad de z se conservaría en el resultado; ni que sea un múltiplo de 3, porque dos valores z y z' que sólo se diferencien en la posición de dos *bytes* tendrían el mismo valor. En general, es necesario evitar los valores de r tales que $(b^s \pm a) \bmod r = 0$, porque un módulo con este valor tiende a ser una superposición de los *bytes* de z , siendo lo más adecuado un r primo tal que $b^s \bmod r \neq \pm a$, para s y a pequeños. En la práctica, es suficiente que r no tenga divisores más pequeños que 20. Se puede encontrar un ejemplo de elección de r en [Meh84, pp. 121-122].
- Plegamiento-desplegamiento (ing., *folding-unfolding*). Se divide la representación binaria de la clave numérica z en m partes de la misma longitud k (excepto posiblemente la última), normalmente un *byte*. A continuación, se combinan las partes mediante algún operador cuya ejecución sea rápida (por ejemplo, con sumas o o-exclusivos) obteniéndose un valor $r = 2^{k+\varepsilon}$, dependiendo ε del método concreto de combinación (por ejemplo, si usamos o-exclusivos, $\varepsilon = 0$).
- Cuadrado. Se define $g(z)$ como la interpretación numérica de los $\lceil \log_2 r \rceil$ bits centrales de z^2 . Conviene que r sea potencia de 2. Esta función distribuye bien porque los bits centrales de un cuadrado no dependen de ninguna parte concreta de la clave, siempre que no haya ceros de relleno al inicio o al final del número. Debe controlarse el posible desbordamiento que pueda provocar el cálculo del cuadrado. En [AHU83, p.131] se presenta una variante interesante.

El problema de todos estos métodos es que pueden distribuir mal una secuencia dada de claves, en cuyo caso es necesario encontrar una nueva función, pero, ¿con qué criterios? ¿Y si esta nueva función tampoco se comporta correctamente? Un enfoque diferente consiste en trabajar no con funciones individuales, sino con familias de funciones de modo que, si una de estas funciones se comporta mal para una distribución dada de la entrada, se elija aleatoriamente otra de la misma familia. Son las denominadas *familias universales* (ing., *universal class*) de funciones de dispersión, definidas por J.L. Carter y M.N. Wegman el año 1979 en "Universal Classes of Hash Functions", *Journal of Computer and System Sciences*, 18. De manera más formal, dada una familia de funciones H , $H \subseteq \{g : Z_a \rightarrow Z_r\}$, siendo a el valor más grande posible después de aplicar f a la clave, diremos que H es una familia *2-universal* si: $\forall x, y : x, y \in Z_a : |\{h \in H / h(x) = h(y)\}| \leq \|H\| / r$ o, en otras palabras, si ningún par de claves colisiona en exceso sobre el conjunto de todas las funciones de H (precisamente,

el prefijo "2" de la definición pone el énfasis en el estudio del comportamiento sobre pares de elementos), de manera que si se toma aleatoriamente una función de H y se aplica sobre un conjunto cualquiera de datos A , $A \subseteq K$, es altamente probable que esta función sea tan buena (es decir, que distribuya las claves igual de bien) como otra función diseñada específicamente para K . Informalmente, se está afirmando que dentro de H hay un número suficiente de "buenas" funciones para esperar que aleatoriamente se elija alguna de ellas.

Una consecuencia de la definición de 2-universal es que, dado un conjunto A cualquiera de claves y dada $k \in K$ una clave cualquiera tal que $k \notin A$, el número esperado de colisiones de k con las claves de A usando una función f de 2-universal está acotado por $\|A\|/r$, que es una magnitud que asegura el buen comportamiento esperado de la función. Es más, la variancia de este valor está acotada para cada consulta individual, lo que permite forzar que dichas consultas cumplan restricciones temporales (v. "Analysis of a Universal Class of Functions", G. Markowsky, J.L. Carter y M.N. Wegman, en los *Proceedings 7th Workshop of Mathematical Foundations of Computer Science*, LNCS 64, Springer-Verlag, 1978).

Carter y Wegman presentan tres clases de 2-universal, adoptadas también por otros autores (principalmente las dos primeras, posiblemente con pequeñas variaciones), que exhiben el mismo comportamiento y que pueden evaluarse de modo eficiente:

- H_1 . Sea p un número primo tal que $p \geq a$, sea $g_1: Z_p \rightarrow Z_r$ una función uniforme cualquiera (por ejemplo, $g_1(z) = z \bmod r$), sea $g_2^{m,n}: Z_a \rightarrow Z_p$ una familia de funciones tales que $m, n \in Z_a$, $m \neq 0$, definida como $g_2^{m,n}(z) = (m \cdot z + n) \bmod p$ y, finalmente, sea $g_{m,n}: Z_a \rightarrow Z_r$ una familia de funciones definida como $g_{m,n}(z) = g_1(g_2^{m,n}(z))$. Entonces se cumple que la clase $H_1 = \{g_{m,n} / m, n \in Z_a \text{ con } m \neq 0\}$ es 2-universal. El artículo original de Carter y Wegman da una propiedad sobre p que, si se cumple, reduce el número de divisiones del algoritmo de dispersión.
- H_3 . En este caso, es necesario que r sea potencia de 2, $r = 2^s$. Sea $j = \lceil \log_2 a \rceil$ (es decir, j es el número mínimo de bits necesarios para codificar cualquier valor de Z_a), sea Ψ el conjunto de matrices $M_{j \times s}$ de componentes 0 ó 1, sea $g_M: Z_a \rightarrow Z_r$ una familia de funciones definida como $g_M(z) = [z_j \otimes_s (m_{j1} \dots m_{js})] \oplus_s \dots \oplus_s [z_1 \otimes_s (m_{11} \dots m_{1s})]$, donde $z_j \dots z_1$ es la representación binaria de z , M es una matriz de Ψ tal que sus filas son de la forma $(m_{i1} \dots m_{is})$, \oplus_s es la operación o-exclusivo sobre operandos de s bits y $z_j \otimes_s (m_{j1} \dots m_{js})$ es igual a s ceros si $z_j = 0$ o a $(m_{j1} \dots m_{js})$ si $z_j = 1$. Entonces se cumple que la clase $H_3 = \{g_M / M \in \Psi\}$ es 2-universal.
- H_2 . Variante de H_3 que permite ganar tiempo a costa de espacio, consiste en interpretar el número $z = z_j \dots z_1$ como un número en base α que se puede expandir a otro número binario $z' = z_j \alpha^{(j\alpha)-1} \dots z_1$, en el que habrá exactamente $\lceil r/\alpha \rceil$ unos; si esta magnitud no es muy grande, el cálculo exige menos o-exclusivos.

5.3.3 Funciones de traducción de cadenas a enteros en un intervalo

La estrategia más sencilla es la combinación de dos métodos de las familias que acabamos de estudiar. La más usual consiste en aplicar la función de suma ponderada a la cadena de caracteres que forma la clave y operar el resultado con una familia universal o un método particular, generalmente el de la división; en este último caso, se puede alternar el módulo con la suma ponderada de manera que se evite el problema del desbordamiento.

También se pueden diseñar funciones que manipulen directamente la representación binaria de la clave. Si la clave es pequeña, puede interpretarse su representación como un número binario; si no, es necesario combinar sus *bytes* con ciertas operaciones (usualmente, sumas y o-exclusivos) de manera parecida al método de desplegamiento visto en el punto anterior. Presentamos aquí uno de estos métodos propuesto por P.K. Pearson en "Fast Hashing of Variable-Length Text Strings", *Communications ACM*, 33(6), 1990, que tiene un funcionamiento parecido a la familia H_3 de funciones universales.

Sea $r = 2^s$ la mínima potencia de 2 más grande que la base b de los caracteres, y sea un vector T [de 0 a $r-1$] que contenga una permutación aleatoria (sin repeticiones) de los elementos de Z_r (sus elementos, pues, se representan con s bits). La función de dispersión $h: C^* \rightarrow Z_r$ aplicada sobre una clave $k = c_n \dots c_1$ se define como:

$$\begin{aligned} A_1(k) &= \phi(c_1) \\ A_i(k) &= T[A_{i-1}(k) \oplus_s \phi(c_i)], i > 1 \\ h(k) &= A_n(k) \end{aligned}$$

siendo \oplus_s la operación o-exclusivo sobre operandos de s bits.

Este esquema se adapta perfectamente al caso habitual de claves de longitud variable o desconocida *a priori*; además, no impone ninguna restricción sobre su longitud y no necesita preocuparse de posibles desbordamientos. Ahora bien, la definición dada presenta un grave inconveniente: la dimensión de la tabla viene determinada por s , lo que normalmente resulta en un valor muy pequeño. Si suponemos el caso inverso, en el cual se diseña la función de dispersión a partir de r , es necesario reformular la definición. Sea t el entero más pequeño tal que $r-1 < 2^t$, siendo t el número de bits necesario para representar valores dentro de Z_r ; entonces, se aplica la función A_n un total de $p = \lceil t/s \rceil$ veces sobre k de la siguiente forma:

- Se define la familia de funciones $A_n^i(k) = A_n(k^i)$, donde $k^i = c_n \dots c_1^i$, siendo c^i el carácter tal que $\phi(c^i) = (\phi(c) + i) \bmod r$.
- Se define la función de dispersión como $h(k) = A_n^{p-1}(k) \cdot \dots \cdot A_n^0(k)$, siendo \cdot la concatenación de bits. Notemos que, efectivamente, $h(k)$ da valores en Z_r .

Se puede demostrar que las p secuencias de valores de $A_n^i(k)$ son independientes entre sí. Si r no es potencia de 2, la última vez que se aplique la función es necesario obtener menos de s bits, por ejemplo, tomando el resto de la división t/s .

5.3.4 Caracterización e implementación de las funciones de dispersión

En la siguiente sección se presentan varias implementaciones de tablas de dispersión. Todas ellas serán independientes de la función de dispersión elegida, que se convertirá, pues, en un parámetro formal de la implementación. Por ello, es necesario encapsular la definición de las funciones de dispersión en un universo de caracterización.

a) Caracterización de las funciones $f: \mathcal{C}^* \rightarrow \mathbb{Z}$

En el universo de caracterización *ELEM_DISP_CONV* (v. fig. 5.7) se define el género *elem* que representa las claves tal como se ha requerido en la especificación (es decir, usando *ELEM_*). Además, para poder aplicarles una función de dispersión, estas claves se consideran como una tira de elementos individuales de tipo *ítem* (que generalmente serán caracteres, como ya se ha dicho), y presentan dos operaciones, una para averiguar el número de ítems de una clave y otra para obtener el ítem *i*-ésimo. Por último, es necesario definir también como parámetros formales la cardinalidad *b* del dominio de los ítems y la función ϕ de conversión de un ítem a un natural dentro del intervalo apropiado, que denotamos respectivamente por *base* y *conv*. El universo resultante contiene, pues, los parámetros formales necesarios para definir las funciones de dispersión *f*. A continuación, se puede enriquecer este universo formando uno nuevo, *FUNCIONES_F*, que añade la caracterización de las funciones $f: \mathcal{C}^* \rightarrow \mathbb{Z}$ (v. fig. 5.8).

```

universo ELEM_DISP_CONV caracteriza
  usa ELEM_ =, NAT, BOOL
  tipo ítem
  ops i_ésimo: elem nat → ítem
      nitems: elem → nat
      conv: ítem → nat
      base: → nat
  errores  $\forall v \in \text{elem}; \forall i \in \text{nat}: [\text{NAT.ig}(i, 0) \vee (i > \text{nitems}(v))] \Rightarrow \text{i\_ésimo}(v, i)$ 
  ecns  $\forall v \in \text{ítem}: (\text{conv}(v) > 0) \wedge (\text{conv}(v) \leq \text{base}) = \text{cierto}$ 
funiverso

```

Fig. 5.7: caracterización de las claves de las funciones de dispersión *f*.

```

universo FUNCIONES_F caracteriza
  usa ELEM_DISP_CONV, NAT
  ops f: elem → nat
funiverso

```

Fig. 5.8: enriquecimiento de *ELEM_DISP_CONV* añadiendo la función *f*.

Los diferentes métodos particulares vistos en 5.3.1 se pueden definir dentro de universos de implementación convenientemente parametrizados por *ELEM_DISP_CONV*. Los algoritmos resultantes son parámetros reales potenciales en aquellos contextos donde se requiera una función *f* (es decir, en universos parametrizados por *FUNCIONES_F*). Por ejemplo, en la fig. 5.9 se muestra el método de la suma ponderada aplicando la regla de Horner; se supone que *mult_desb*(*x*, *y*, *z*) efectúa la operación $x*y + z$ sin que se llegue a producir realmente desbordamiento en ninguna operación.

```

universo SUMA_POND (ELEM_DISP_CONV) implementa ...4
  usa NAT
  función suma_pond (v es elem) devuelve nat es
  var acum, i son nat fvar
    acum := 0
    para todo i desde nitems(v) bajando hasta 1 hacer
      acum := mult_desb(acum, base, conv(i_ésimo(v, i)))
    fpara todo
  devuelve acum
funiverso

```

Fig. 5.9: implementación del método de la suma ponderada.

b) Caracterización de las funciones $g: Z \rightarrow Z_r$

En el universo de caracterización *FUNCIONES_G* se definen el método de dispersión y el valor *r* (v. fig. 5.10). A continuación, se pueden implementar los diversos algoritmos de forma similar a las funciones *f*, parametrizados tan solo por el valor *r* (natural caracterizado en *VAL_NAT*); por ejemplo, en la fig. 5.11 se implementa el método de la división.

```

universo FUNCIONES_G caracteriza
  usa NAT, BOOL
  ops r:  $\rightarrow$  nat
  g: nat  $\rightarrow$  nat
  ecns  $\forall z \in \text{nat}: g(z) < r = \text{cierto}$ 
funiverso

```

Fig. 5.10: caracterización de las claves de las funciones de dispersión *g*.

⁴ La especificación ésta y las restantes funciones de dispersión presenta el problema típico de la sobreespecificación propio de la semántica inicial, porque el universo resultante es en realidad una implementación por ecuaciones. Por ello, se podría considerar la posibilidad de especificarla en otro marco semántico.

```

universo DIVISIÓN (VAL_NAT) implementa ...
  usa NAT
  función división (z es nat) devuelve nat es
    devuelve z mod val
funiverso

```

Fig. 5.11: implementación del método de la división.

c) Caracterización de las funciones $h: \mathcal{C}^* \rightarrow Z_r$

De manera similar, se define un universo de caracterización parametrizado tanto por el tipo de las claves como por el valor r (v. fig. 5.12); este universo caracteriza todos los símbolos que el usuario de una tabla de dispersión ha de determinar, y por ello es el universo que finalmente aparecerá en la cabecera de las diversas implementaciones de las tablas.

```

universo CLAVE_DISPERSIÓN caracteriza
  usa ELEM_ =, VAL_NAT
  ops h: elem  $\rightarrow$  nat
  ecns  $\forall v \in \text{elem}: h(v) < \text{val} = \text{cierto}$ 
funiverso

```

Fig. 5.12: caracterización de las claves en el contexto de las funciones de dispersión h .

En este punto ya es posible definir funciones de dispersión. La primera opción consiste en componer dos funciones f y g , tal como se muestra en la fig. 5.13. Por ejemplo, en la fig. 5.14 se da la composición de la suma ponderada y la división; las dos primeras instancias, privadas, construyen las funciones f y g en el contexto requerido por el tipo de las claves, el valor del módulo, etc. y, a continuación, se efectúa una tercera instancia que define la función h simplemente como composición de las anteriores, con un renombramiento final. La segunda manera de definir funciones de dispersión consiste en trabajar directamente sobre la clave sin pasos intermedios. Por ejemplo, en la fig. 5.15 se muestra la función que alterna la suma ponderada y la división.

```

universo COMPOSICIÓN_F_Y_G (FUNCIONES_F, FUNCIONES_G) implementa ...
  función g_de_f (v es elem) devuelve nat es
    devuelve g(f(v))
funiverso

```

Fig. 5.13: composición de funciones de dispersión.

```

universo SUMA_POND_Y_DIV(E es ELEM_DISP_CONV, V1 es VAL_NAT) impl. ...
  instancia privada SUMA_POND(F es ELEM_DISP_CONV) donde
    F.elem es E.elem, F.item es E.item
    F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
    F.conv es E.conv, F.base es E.base
  instancia privada DIVISIÓN(V2 es VAL_NAT) donde V2.val es V1.val
  instancia COMPOSICIÓN_F_Y_G(F es FUNCIONES_F, G es FUNCIONES_G) donde
    F.elem es E.elem, F.item es E.item
    F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
    F.conv es E.conv, F.base es E.base, F.f es suma_pond
    G.r es V1.val, G.g es división
  renombra f_de_g por suma_pond_y_div
funiverso

```

Fig. 5.14: composición de la suma ponderada y la división.

```

universo SUMA_POND_Y_DIV_SIMULT(ELEM_DISP_CONV, VAL_NAT) impl. ...
  usa NAT
  función suma_pond_y_mod (v es elem) devuelve nat es
  var acum, i son nat fvar
    acum := 0
    para todo i desde nítems(v) bajando hasta 1 hacer
      acum := mult_desb(acum, base, conv(i_ésimo(v, i))) mod val
    fpara todo
  devuelve acum
funiverso

```

Fig. 5.15: aplicación simultánea de la suma ponderada y la división.

Cabe destacar que los universos resultantes de estos pasos deben crearse una única vez y que, a partir de ese momento, el usuario los tendrá disponibles en una biblioteca de funciones de dispersión, y se limitará simplemente a efectuar las instancias adecuadas en aquellos contextos que lo exijan (sin necesidad de conocer la estructuración interna en universos). Estas instancias son muy simples. Por ejemplo, supongamos que se quiere implementar una tabla donde las claves son cadenas de letras mayúsculas, los valores son enteros y la dimensión de la tabla es 1017 y que, como función de dispersión, se componen la suma ponderada y la división; el resultado podría ser la instancia que aparece a continuación, donde *conv_mayúsc* debería estar implementada en la biblioteca de funciones de conversión, y *CADENA* presenta las operaciones definidas en el apartado 1.5.1.

instancia SUMA_POND_Y_DIV(ELEM_DISP_CONV, VAL_NAT) donde
 elem es cadena, ítem es carácter
 = es CADENA.=, nítens es CADENA.long, i_ésimo es CADENA.i_ésimo
 conv es conv_mayúsc, base es 26, val es 1017

5.4 Organizaciones de las tablas de dispersión

Las tablas de dispersión pueden organizarse de varias maneras según el modo en que se gestionen las colisiones. Hay dos grandes familias de tablas dependiendo de si se encadenan o no los sinónimos; además, existen diversas organizaciones que son combinaciones de estas dos, de las que se presentará una. Algunos autores también clasifican las tablas como *abiertas* o *cerradas*, según exista una zona diferenciada para todas o parte de las claves, o se almacenen en un único vector directamente indexado por la función de dispersión. A veces, los nombres que dan diversos autores pueden causar confusión; por ejemplo, algunos textos denominan *closed hashing* a una organización que otros llaman *open addressing*.

A lo largo de la sección, definiremos las diferentes organizaciones como universos parametrizados por los géneros de las claves y los valores, la igualdad de las claves, el valor indefinido, la función de dispersión y el número de valores de dispersión diferentes que hay.

5.4.1 Tablas de dispersión encadenadas

En las *tablas de dispersión encadenadas* (ing., *chaining*) se forman estructuras lineales con los sinónimos. Estudiaremos dos esquemas:

- *Tablas encadenadas indirectas*. Se encadenan los sinónimos de un mismo valor de dispersión, y el primer elemento de la lista es accesible a partir de un vector índice.
- *Tablas encadenadas directas*. Se guarda la primera clave de cada valor de dispersión, con su valor asociado, en una zona diferenciada, y los respectivos sinónimos se encadenan en otra zona, y son accesibles a partir del que reside en la zona diferenciada.

a) Tablas de dispersión encadenadas indirectas

También conocidas con el nombre de *tablas de dispersión encadenadas abiertas* (ing., *separate chaining*), asocian la lista de los sinónimos de un valor de dispersión i a la posición i de un vector índice. En la fig. 5.16 se muestra el esquema de esta representación. Queda claro que si la función distribuye correctamente, las listas de sinónimos serán de longitud similar, concretamente $\lceil n/r \rceil$, donde n es el número de elementos en la tabla. En consecuencia, las operaciones quedan $\Theta(n/r)$. Si aproximadamente hay tantos elementos

como valores de dispersión, este cociente será muy pequeño (es decir, las listas de sinónimos serán muy cortas) y las funciones del TAD, exceptuando la creación (que es $\Theta(r)$) se podrán considerar $\Theta(1)$ en todos los casos, dado que la inserción y la supresión consisten simplemente en buscar el elemento y modificar un par de encadenamientos.

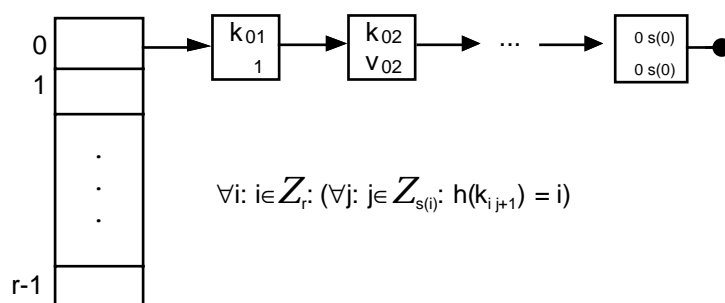


Fig. 5.16: organización de una tabla de dispersión encadenada indirecta.

En la fig. 5.17 se presenta una posible implementación para el tipo *tabla* representando las listas en memoria dinámica (otra alternativa sería guardar los elementos dentro de un único vector compartido por todas las listas). Observemos que el invariante establece que los elementos de la lista que cuelga de la posición i del vector índice son sinónimos del valor de dispersión i , usando la conocida función que devuelve la cadena que cuelga de un elemento inicial (v. fig. 3.23). Asimismo, se refleja la inexistencia de elementos repetidos. Los universos de caracterización definen los parámetros necesarios para aplicar las funciones de dispersión tal como se ha explicado en la sección anterior; notemos que, en particular, todos los parámetros que aparecían en la especificación también están en la implementación, aunque el encapsulamiento en universos de caracterización haya cambiado. Tanto en esta implementación como en posteriores, no se incluye código para controlar el buen comportamiento de la función de dispersión, quedando como ejercicio para el lector.

Los algoritmos son claros: se accede por la función de dispersión al índice y se recorre la lista de sinónimos; las inserciones se efectúan siempre por el inicio (es lo más sencillo) y en las supresiones es necesario controlar si el elemento borrado es el primero, para actualizar el índice (una alternativa sería utilizar elementos fantasmas, pero a costa de aumentar considerablemente el espacio empleado, siendo necesario un estudio muy cuidadoso para justificar su conveniencia). Se usa una función auxiliar, *busca*, que busca una clave en la tabla. Podríamos considerar la posibilidad de ordenar los sinónimos por clave o de disponer de listas autoorganizativas; estas dos variantes reducen el coste de algunas búsquedas a costa de las inserciones. Otra variante consiste en no buscar las claves cuando se insertan sino añadirlas directamente al inicio de la lista, de manera que la consulta siempre encontrará primero la última ocurrencia de la clave. Obviamente, si las claves se redefinen con frecuencia

puede llegar a desaprovecharse mucha memoria; de hecho, esta alternativa es especialmente útil cuando el contexto de utilización de la lista asegura que nunca se insertarán elementos con la misma clave más de una vez, lo que ocurre a veces.

universo TABLA_IND_PUNTEROS(CLAVE_DISPERSIÓN, ELEM_ESP) es
implementa FUNCION_TOTAL(ELEM_-, ELEM_ESP)
usa ENTERO, BOOL
renombra CLAVE_DISPERSIÓN.elem por clave, CLAVE_DISPERSIÓN.val por r
ELEM_ESP.elem por valor, ELEM_ESP.esp por indef
tipo tabla es vector [de 0 a r-1] de ^nodo ftipo
tipo privado nodo es
tupla
k es clave; v es valor; enc es ^nodo
ftupla
ftipo
invariante (T es tabla):
 $\forall i: 0 \leq i \leq r-1:$

$$\text{NULO} \in \text{cadena}(T[i]) \wedge (\forall p: p \in \text{cadena}(T[i]) - \{\text{NULO}\}: h(p.k) = i) \wedge$$

$$\forall p, q: p, q \in \text{cadena}(T[i]) - \{\text{NULO}\}: p \neq q \Rightarrow h(p.k) \neq h(q.k)$$
función crea devuelve tabla es
var i es nat; t es tabla fvar
para todo i desde 0 hasta r-1 hacer t[i] := NULO fpara todo
devuelve t
función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
var i es nat; encontrado es booleano; ant, p son ^nodo fvar
i := h(k); <encontrado, ant, p> := busca(t[i], k)
si encontrado entonces p^.v := v {cambiamos la información asociada}
si no {es necesario crear un nuevo nodo}
p := obtener_espacio
si p = NULO entonces error {no hay espacio}
si no p^ := <k, v, t[i]>; t[i] := p {inserción por el inicio}
fsi
fsi
devuelve t

Fig. 5.17: implementación de las tablas de dispersión encadenadas indirectas.

```

función borra (t es tabla; k es clave) devuelve tabla es
var i es nat; encontrado es booleano; ant, p son ^nodo fvar
  i := h(k); <encontrado, ant, p> := busca(t[i], k)
  si encontrado entonces {es necesario distinguir si es el primero o no}
    si ant = NULO entonces t[i] := p^.enc si no ant^.enc := p^.enc fsi
    liberar_espacio(p)
  fsi
devuelve t

```

```

función consulta (t es tabla; k es clave) devuelve valor es
var encontrado es booleano; res es valor; ant, p son ^nodo fvar
  <encontrado, ant, p> := busca(t[h(k)], k)
  si encontrado entonces res := p^.v si no res := indef fsi
devuelve res

```

{Función $busca(q, k) \rightarrow \langle encontrado, ant, p \rangle$: busca el elemento de clave k a partir de q . Devuelve un booleano indicando el resultado de la búsqueda y, en caso de éxito, un apuntador p a la posición que ocupa y otro ant a su predecesor, que valdrá NULO si p es el primero}

$$P \equiv NULO \in cadena(q)$$

$$Q \equiv encontrado \equiv \exists r: r \in cadena(q) - \{NULO\}: r.k = k \wedge$$

$$encontrado \Rightarrow p.k = k \wedge (p = q \Rightarrow ant = NULO \wedge p \neq q \Rightarrow ant^.enc = p) \}$$

```

función privada busca (q es ^nodo; k es clave) devuelve <bool, ^nodo, ^nodo> es
var encontrado es booleano; ant, p es ^nodo fvar
  ant := NULO; p := q; encontrado := falso
  mientras p ≠ NULO ∧ ¬encontrado hacer
    si p.k = k entonces encontrado := cierto si no ant := p; p := p^.enc fsi
  fmientras
devuelve <encontrado, ant, p>

```

funiverso

Fig. 5.17: implementación de las tablas de dispersión encadenadas indirectas (cont.).

Una alternativa a la implementación del tipo consiste en usar una instancia de alguna implementación de las listas escrita en algún universo ya existente. En la sección 6.3.2 se introduce una representación del TAD de los grafos usando listas con punto de interés que sigue esta estrategia, de manera que el lector puede comparar los resultados.

Por último, es necesario destacar un problema que surge en todas las organizaciones de dispersión: la progresiva degeneración que sufren a medida que crece su tasa de ocupación. Mientras más elementos hay en la tabla, más largas son las listas de sinónimos.

Para evitarla se puede incorporar a la representación un contador de elementos en la tabla, de forma que el cálculo de la tasa de ocupación sea inmediato; al insertar nuevos elementos es necesario comprobar previamente que la tasa no sobrepase una cota máxima determinada (valor que sería un parámetro más, tanto de la especificación como de la implementación), en cuyo caso se da un error conocido con el nombre de *desbordamiento* (ing., *overflow*; evidentemente, la especificación de las tablas debería modificarse para reflejar este comportamiento). La solución más simple posible del desbordamiento (y también la más ineficiente) consiste en redimensionar la tabla, lo cual obliga a definir una nueva función de dispersión y, a continuación reinsertar todos los identificadores presentes en la tabla usando esta nueva función. Alternativamente, hay métodos que aumentan selectivamente el tamaño de la tabla, de manera que sólo es necesario reinsertar un subconjunto de los elementos; estas estrategias, denominadas *incrementales*, difieren en la frecuencia de aplicación y en la parte de la tabla tratada. Entre ellas destacan la *dispersión extensible* y la *dispersión lineal*; ambos métodos son especialmente útiles para tablas implementadas en disco y no se estudian aquí. En el apartado 5.4.6 se presentan unas fórmulas que determinan exactamente cómo afecta la tasa de ocupación a la eficiencia de las operaciones.

b) Tablas de dispersión encadenadas directas

También conocidas con el nombre de *tablas de dispersión con zona de excedentes* (ing., *hashing with cellar*), precisamente por la distinción de la tabla en dos zonas (v. fig. 5.18): la zona principal de r posiciones, donde la posición i , o bien está vacía, o bien contiene un par $\langle k, v \rangle$ que cumple $h(k) = i$; y la zona de excedentes, que ocupa el resto de la tabla y guarda las claves sinónimas. Cuando al insertar un par se produce una colisión, el nuevo sinónimo se almacena en esta zona; todos los sinónimos de un mismo valor están encadenados y el primer sinónimo residente en la zona de excedentes es accesible a partir del que reside en la zona principal. Alternativamente, la zona de excedentes se puede implementar por punteros sin que el esquema general varíe sustancialmente. Las consideraciones sobre la longitud de las listas y el coste de las operaciones son idénticas al caso de tablas encadenadas indirectas; sólo es necesario notar el ahorro de un acceso a causa de la inexistencia del vector índice, que evidentemente no afecta al coste asintótico de las operaciones⁵ (exceptuando la creación, que pasa de $\Theta(r)$ a $\Theta(\text{máx})$).

⁵ Pero que es importante al considerar la implementación sobre ficheros; de hecho, las tablas con zona de excedentes son útiles sobre todo en memoria secundaria, donde en cada cubeta se colocan tantos elementos como se puedan leer del / escribir al disco en una única operación de entrada/salida.

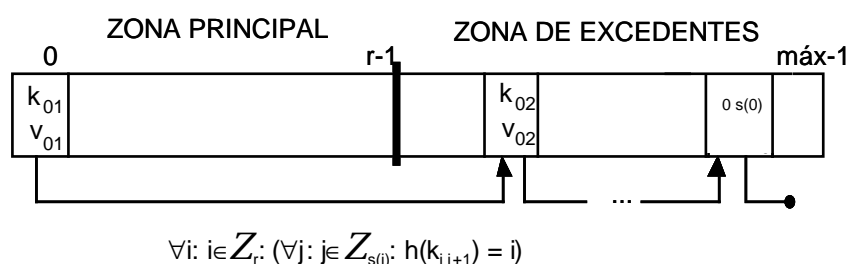


Fig. 5.18: organización de una tabla de dispersión encadenada directa.

En la fig. 5.19 se ofrece una implementación del tipo; en la cabecera aparece un universo de caracterización adicional, *VAL_NAT*, para determinar la dimensión de la zona de excedentes (diversos estudios empíricos apuntan a que esta zona ha de representar aproximadamente el 14% de la tabla, v. artículo de Lewis y Cook ya referenciado). Vuelve a haber una función auxiliar para buscar una clave. Observamos que la codificación de las funciones es un poco más complicada que en el caso indirecto, porque es necesario distinguir si una posición está libre o no, y también el acceso a la zona principal del acceso a la zona de excedentes. Para solucionar el primer punto, se establece una convención para saber si las posiciones de la zona principal están o no vacías, y en caso de que estén vacías, se marcan (en la fig. 5.19, se asigna el valor -2 al campo de encadenamiento, y queda el -1 para el último elemento de cada lista). Las posiciones libres de la zona de excedentes se gestionan en forma de pila.

La supresión de los elementos residentes en la zona de excedentes requiere tan sólo modificar encadenamientos e insertar la posición liberada en la pila de sitios libres; ahora bien, al borrar un elemento residente en la zona principal no se puede simplemente marcar como libre la posición correspondiente, porque el algoritmo de búsqueda no encontraría los hipotéticos sinónimos residentes en la zona de excedentes. Así, se podría introducir otra marca para distinguir las posiciones borradas, de manera que las posiciones marcadas como borradas se interpreten como ocupadas al buscar y como libres al insertar una nueva clave. El mantenimiento de posiciones borradas puede provocar que en la zona principal existan demasiados "agujeros", que la tabla degenere (es decir, que empeore el tiempo de acceso) y que, eventualmente, se llene la zona de excedentes, quedando posiciones borradas en la zona principal y provocando error en la siguiente colisión. Una alternativa consiste en mover un sinónimo de la zona de excedentes a la zona principal, lo que es costoso si el tamaño de los elementos es grande o si se utilizan atajos para el acceso a la estructura. En la fig. 5.19 se implementa esta última opción, quedando la primera como ejercicio. Notemos que el invariante es ciertamente exhaustivo y comprueba los valores de los encadenamientos, la ocupación de las diferentes posiciones de la tabla y que todos elementos que cuelgan de la misma cadena sean sinónimos del valor correspondiente de dispersión; se usa la función *cadena* de la fig. 3.20 aplicada sobre tablas.

universo TABLA_DIRECTA(CLAVE_DISPERSIÓN, ELEM_ESP, VAL_NAT) es
implementa FUNCION_TOTAL(ELEM_ =, ELEM_ESP)
usa ENTERO, BOOL
renombra CLAVE_DISPERSIÓN.elem por clave, CLAVE_DISPERSIÓN.val por r
ELEM_ESP.elem por valor, ELEM_ESP.esp por indef
const máx vale r+VAL_NAT.val
tipo tabla es tupla A es vector [de 0 a máx-1] de
tupla k es clave; v es valor; enc es entero ftupla
sl es entero
ftupla
ftipo
invariante (T es tabla):

$$\begin{aligned}
& (T.sl = -1) \vee (r \leq T.sl < \text{máx}) \\
& \wedge \forall i: 0 \leq i \leq r-1: (T.A[i].enc = -1) \vee (T.A[i].enc = -2) \vee (r \leq T.A[i].enc < \text{máx}) \\
& \wedge \forall i: r \leq i \leq \text{máx}-1: (T.A[i].enc = -1) \vee (r \leq T.A[i].enc < \text{máx}) \\
& \wedge \forall i: 0 \leq i \leq r-1 \wedge T.A[i].enc \neq -2: \{ \text{para toda posición no vacía} \} \\
& \quad \forall p: p \in (\text{cadena}(T.A, i) - \{-1\}): h(T.A[p].k) = i \wedge \\
& \quad \forall p, q: p, q \in \text{cadena}(\text{cadena}(T.A, i) - \{-1\}): p \neq q \Rightarrow h(T.A[p].k) \neq h(T.A[q].k) \wedge \\
& \quad \forall j: 0 \leq j \leq r-1 \wedge j \neq i \wedge A[j].enc \neq -2: \text{cadena}(T.A, i) \cap \text{cadena}(T.A, j) = \{-1\} \wedge \\
& \quad \text{cadena}(T.A, i) \cap \text{cadena}(T.A, T.sl) = \{-1\} \\
& \wedge (\cup i: 0 \leq i \leq r-1 \wedge T.A[i].enc \neq -2: \text{cadena}(T.A, T.A[i].enc)) \cup \\
& \quad \cup \text{cadena}(T.A, T.sl) = [r, \text{máx}-1] \cup \{-1\}
\end{aligned}$$

función crea devuelve tabla es
var i es nat; t es tabla fvar
{zona principal vacía}
para todo i desde 0 hasta r-1 hacer t.A[i].enc := -2 fpara todo
{pila de sitios libres}
para todo i desde r hasta máx-2 hacer t.A[i].enc := i+1 fpara todo
t.A[máx-1].enc := -1; t.sl := r
devuelve t

función consulta (t es tabla; k es clave) devuelve valor es
var encontrado es booleano; res es valor; ant, p son entero fvar
<encontrado, ant, p> := busca(t, k, h(k))
si encontrado entonces res := t.A[p].v si no res := indef fsi
devuelve res

Fig. 5.19: implementación de las tablas de dispersión encadenadas directas.

```

función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
var ant, i, p son enteros; encontrado es booleano fvar
    i := h(k)
    si t.A[i].enc = -2 entonces t.A[i] := <k, v, -1> {posición libre en la zona principal}
    si no {hay sinónimos}
        <encontrado, ant, p> := busca(t, k, i)
        si encontrado entonces t.A[p].v := v {cambio de la información asociada}
        si no si t.sl = -1 entonces error {no hay espacio}
            si no {se deposita en la zona de excedentes}
                p := t.sl; t.sl := t.A[t.sl].enc; t.A[p] := <k, v, t.A[i].enc>; t.A[i].enc := p
            fsi
        fsi
    fsi
devuelve t

función borra (t es tabla; k es clave) devuelve tabla es
var ant, p, nuevo_sl son enteros; encontrado es booleano fvar
    <encontrado, ant, p> := busca(t, k, h(k))
    si encontrado entonces
        si ant = -1 entonces {reside en la zona principal}
            si t.A[p].enc = -1 entonces t.A[p].enc := -2 {no tiene sinónimos}
            si no {se mueve el primer sinónimo a la zona principal}
                nuevo_sl := t.A[p].enc; t.A[p] := t.A[t.A[p].enc]
                t.A[nuevo_sl].enc := t.sl; t.sl := nuevo_sl
            fsi
        si no {reside en la zona de excedentes}
            t.A[ant].enc := t.A[p].enc; t.A[p].enc := t.sl; t.sl := p
        fsi
    fsi
devuelve t

función privada busca (t es tabla; k es clave; i es entero) dev <bool, entero, entero> es
var encontrado es booleano; ant es entero fvar
    encontrado := falso
    si t.A[i].enc ≠ -2 entonces {si no, no hay claves con valor de dispersión i}
        ant := -1
        mientras (i ≠ -1) ∧ ¬encontrado hacer
            si t.A[i].k = k entonces encontrado := cierto si no ant := i; i := t.A[i].enc fsi
        fmientras
    fsi
devuelve <encontrado, ant, i>

```

Fig. 5.19: implementación de las tablas de dispersión encadenadas directas (cont.).

5.4.2 Tablas de dispersión de direccionamiento abierto

En las tablas de direccionamiento abierto (ing., *open addressing*) se dispone de un vector con tantas posiciones como valores de dispersión; dentro del vector, no existe una zona diferenciada para las colisiones ni se encadenan los sinónimos, sino que para cada clave se define una secuencia de posiciones que determina el lugar donde irá. Concretamente, al insertar el par $\langle k, v \rangle$ en la tabla, tal que $h(k) = p_0$, se sigue una secuencia p_0, \dots, p_{r-1} asociada a k hasta que:

- Se encuentra una posición p_s tal que su clave es k . Se sustituye su información asociada por v .
- Se encuentra una posición p_s que está libre. Se coloca el par $\langle k, v \rangle$. Si $s > 0$, a la clave se la llama *invasora* (ing., *invader*), por razones obvias.
- Se explora la secuencia sin encontrar ninguna posición que cumpla alguna de las dos condiciones anteriores. Significa que la tabla está llena y no se puede insertar el par.

El esquema al borrar y consultar es parecido: se siguen las p_0, \dots, p_{r-1} hasta que se encuentra la clave buscada o una posición libre. El punto clave de esta estrategia es que la secuencia p_0, \dots, p_{r-1} asociada a una clave k es fija durante toda la existencia de la tabla de manera que, cuando se añade, se borra o se consulta una clave determinada siempre se examinan las mismas posiciones del vector en el mismo orden; idealmente, claves diferentes tendrán asociadas secuencias diferentes. Este proceso de generación de valores se denomina *redispersión* (ing., *rehashing*), la secuencia generada por una clave se denomina *camino* (ing., *path* o *probe sequence*) de la clave, y cada acceso a la tabla mediante un valor del camino se denomina *ensayo* (ing., *probe*). En otras palabras, el concepto de redispersión consiste en considerar que, en vez de una única función h de dispersión, disponemos de una familia $\{h_i\}$ de funciones, denominadas *funciones de redispersión* (ing., *rehashing function*), tales que, si la aplicación de $h_i(k)$ no lleva al resultado esperado, se aplica $h_{i+1}(k)$; en este esquema, la función h de dispersión se define como h_0 .

Una vez más es necesario estudiar cuidadosamente el problema de la supresión de elementos. La situación es parecida a la organización encadenada directa: cuando una clave k colisiona y se deposita finalmente en la posición determinada por un valor $h_j(k)$ se debe a que todas las posiciones determinadas por los valores $h_i(k)$, para todo $i < j$, están ocupadas; al suprimir una clave que ocupa la posición $h_s(k)$, para algún $s < j$, no se puede simplemente marcar como libre esta posición, porque una búsqueda posterior de k dentro la tabla siguiendo la estrategia antes expuesta fracasaría. En consecuencia, además de las posiciones "ocupadas" o "libres", se distingue un tercer estado que identifica las posiciones "borradas", las cuales se tratan como libres al buscar sitio para insertar un nuevo par y como ocupadas al buscar una clave; al igual que en el esquema encadenado directo ya estudiado, las posiciones borradas provocan la degeneración de la tabla.

En la fig. 5.20 se muestra una implementación de las tablas de direccionamiento abierto sin determinar la familia de funciones de dispersión, que queda como parámetro formal de la implementación sustituyendo a la tradicional función de dispersión; los universos de caracterización de los parámetros formales aparecen en la fig. 5.21. Como en esta implementación no hay campos de encadenamiento para usar como marcas, en la definición de los valores se añade una constante como parámetro formal, que permite implementar el concepto de posición borrada sin emplear ningún campo adicional (las posiciones libres contendrán el valor indefinido, que ya es un parámetro formal). En caso de no poderse identificar este nuevo valor especial, se podría identificar una clave especial y, si tampoco fuera posible, sería imprescindible entonces un campo booleano en las posiciones del vector que codificara el estado. Por lo que respecta al invariante, se usa una función auxiliar *predecesores* que, para una clave k residente en la posición p_i de su camino, devuelve el conjunto de posiciones $\{p_0, \dots, p_{i-1}\}$, las cuales, dada la estrategia de redispersión, no pueden estar libres. Por último, la función auxiliar *busca* da la posición donde se encuentra una clave determinada o, en caso de no encontrarse, la primera posición libre o borrada donde se insertaría, y da prioridad a las segundas para reaprovecharlas; el valor booleano devuelto indica si la clave se ha encontrado o no. Como siempre, se podría incorporar un contador para controlar el factor de ocupación de la tabla, considerando que una posición borrada vale lo mismo que una posición ocupada. La eficiencia de la implementación es la acostumbrada.

En la fig. 5.22 se presenta la evolución de una tabla de direccionamiento abierto siguiendo la estrategia de la fig. 5.20, suponiendo que las funciones de redispersión $\{h_i\}$ simplemente toman el último dígito de la cadena y le suman i (¡es un buen ejemplo de estrategia que nunca debe emplearse!); la disposición de los elementos dentro de la tabla depende de la historia y de los algoritmos concretos de inserción y de supresión, de manera que podríamos encontrar otras configuraciones igualmente válidas al cambiar cualquiera de los dos factores.

Una vez formulada la implementación genérica de las tablas de dispersión de direccionamiento abierto, es necesario estudiar los diferentes métodos de redispersión existentes. Todos ellos se definen a partir de una o dos funciones de redispersión primarias que denotaremos por h y h' . Sea cual sea la familia de funciones de redispersión elegida, hay tres propiedades que deberían cumplirse; la primera de ellas, sobre todo, es ineludible:

- Para una clave dada, su camino es siempre el mismo (las mismas posiciones en el mismo orden).
- Para una clave dada, su camino pasa por todas las posiciones de la tabla.
- Para dos claves diferentes k y k' , si bien es prácticamente inevitable que a veces compartan parte de sus caminos, es conveniente evitar que estos caminos sean idénticos a partir de un punto determinado.

universo TABLA_ABIERTA(CLAVE_REDISPERSIÓN, ELEM_2_ESP_=)
implementa FUNCION_TOTAL(ELEM_=, ELEM_ESP)
renombra
 CLAVE_REDISPERSIÓN.elem por clave, CLAVE_REDISPERSIÓN.val por r
 ELEM_2_ESP_.elem por valor
 ELEM_2_ESP_.esp1 por indef, ELEM_2_ESP_.esp2 por borrada
usa ENTERO, BOOL
tipo tabla es vector [de 0 a r-1] de tupla k es clave; v es valor ftupla ftipo
invariante (T es tabla):

$$\forall p, q: 0 \leq p, q \leq r-1 \wedge T[p].v \neq \text{indef} \wedge T[p].v \neq \text{borrada} \wedge$$

$$T[q].v \neq \text{indef} \wedge T[q].v \neq \text{borrada}: p \neq q \Rightarrow h(T[p].k) \neq h(T[q].k) \wedge$$

$$\forall i: 0 \leq i \leq r-1 \wedge T[i].v \neq \text{indef} \wedge T[i].v \neq \text{borrada}:$$

$$\forall j: j \in \text{predecesores}(T, T[i].k, 0): T[j].v \neq \text{indef}$$
 donde se define *predecesores*: *tabla clave nat* $\rightarrow P(\text{nat})$ como:

$$T[h(i, k)].k = k \Rightarrow \text{predecesores}(T, k, i) = \emptyset$$

$$T[h(i, k)].k \neq k \Rightarrow \text{predecesores}(T, k, i) = \{h(i, k)\} \cup \text{predecesores}(T, k, i+1)$$

función crea devuelve tabla es
var i es nat; t es tabla fvar
para todo i desde 0 hasta r-1 hacer t[i].v := indef fpara todo
devuelve t

función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
var p es entero; encontrado es booleano fvar
 <p, encontrado> := busca(t, k)
si encontrado entonces t[p].v := v {se cambia la información asociada}
si no {se inserta en la posición que le corresponde}
si p = -1 entonces error {no hay espacio} si no t[p] := <k, v> fsi
fsi
devuelve t

función borra (t es tabla; k es clave) devuelve tabla es
var p es nat; encontrado es booleano fvar
 <p, encontrado> := busca(t, k)
si encontrado entonces t[p].v := borrada fsi
devuelve t

función consulta (t es tabla; k es clave) devuelve valor es
var encontrado es booleano; res es valor; p es nat fvar
 <p, encontrado> := busca(t, k)
si encontrado entonces res := t[p].v si no res := indef fsi
devuelve res

Fig. 5.20: implementación de las tablas de dispersión abiertas.

{Función auxiliar $busca(t, k) \rightarrow \langle j, encontrado \rangle$: busca la clave k dentro de t ; si la encuentra, devuelve cierto y la posición que ocupa; si no, devuelve falso y la posición donde insertarla si es el caso (o -1, si no hay ninguna)}

$P \equiv \text{invariante}(t)$

$Q \equiv \text{encontrado} \equiv \exists i: 0 \leq i \leq r-1: (t[i].k = k \wedge t[i].v \neq \text{indef} \wedge t[i].v \neq \text{borrada})$
 $\wedge \text{encontrado} \Rightarrow t[j].k = k$

$\wedge (\neg \text{encontrado} \Rightarrow$

$(j = -1 \wedge \forall i: 0 \leq i \leq r-1: (T[i].v \neq \text{indef} \wedge T[i].v \neq \text{borrada})) \vee$

$(t[j].v = \text{indef} \vee t[j].v = \text{borrada}) \wedge$

$t[j].v = \text{indef} \Rightarrow (\neg \exists s: s \in \text{pos}(T, k, j): T[s].v = \text{borrada})$

donde pos devuelve las posiciones del camino asociado a k entre $T[h(0, k)]$ y j

función privada $busca$ (t es tabla; k es clave) devuelve $\langle \text{nat}, \text{bool} \rangle$ es

var i, p, j son enteros; encontrado , final son booleano fvar

$i := 0$; $\text{final} := \text{falso}$; $j := -1$; $\text{encontrado} := \text{falso}$

mientras $(i < r) \wedge \neg \text{final}$ hacer

$p := h(i, k)$ {se aplica la función i -ésima de la familia de redispersión}

opción

caso $t[p].v = \text{indef}$ hacer {la clave no está}

$\text{final} := \text{cierto}$; si $j = -1$ entonces $j := p$ fsi

caso $t[p].v = \text{borrada}$ hacer $i := i + 1$; $j := p$ {sigue la búsqueda}

en cualquier otro caso {la posición contiene una clave definida}

si $t[p].k = k$ entonces $\text{encontrado} := \text{cierto}$; $\text{final} := \text{cierto}$ si no $i := i + 1$ fsi

fopción

fmientras

devuelve $\langle j, encontrado \rangle$

Fig. 5.20: implementación de las tablas de dispersión abiertas (cont.).

universo ELEM_2_ESP_ caracteriza

usa BOOL

tipo elem

ops $\text{esp1}, \text{esp2} \rightarrow \text{elem}$

$_ = _, _ \neq _ : \text{elem elem} \rightarrow \text{bool}$

funiverso

universo $\text{CLAVE_REDISPERSION}$ caracteriza

usa $\text{ELEM}, \text{VAL_NAT}, \text{NAT}, \text{BOOL}$

ops $h: \text{nat elem} \rightarrow \text{nat}$ $\{h(i, k) \equiv h_i(k)\}$

errores $\forall k \in \text{elem}; \forall i \in \text{nat}: [i \geq \text{val}] \Rightarrow h(i, k)$

ecns $\forall k \in \text{elem}; \forall i \in \text{nat}: h(i, k) < \text{val} = \text{cierto}$

funiverso

Fig. 5.21: caracterización de los parámetros formales de las tablas de dispersión abiertas.

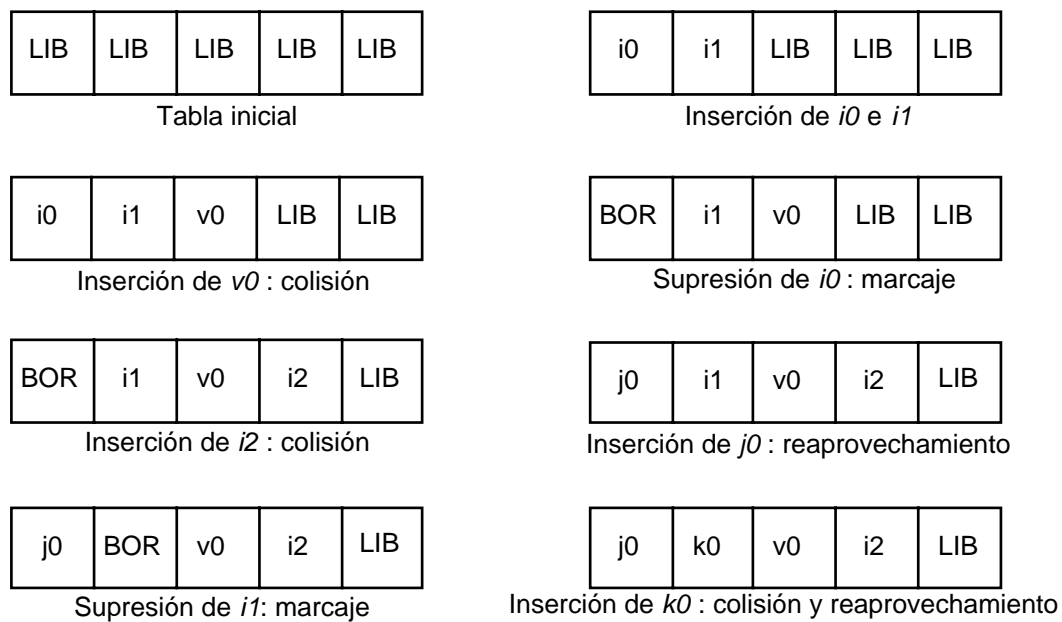


Fig. 5.22: evolución de una tabla de direccionamiento abierto
(LIB: posición libre; BOR: posición borrada).

Examinemos atentamente esta última propiedad. De entrada parece factible que se cumpla, porque en una tabla de dimensión r hay $r!$ caminos diferentes de longitud r . Ahora bien, la mayoría de esquemas de redistribución que estudiamos a continuación usan bastante menos de $r!$ caminos y, por ello, presentan normalmente el fenómeno conocido como *apiñamiento* (ing., *clustering*), que consiste en la formación de largas cadenas de claves en la tabla (denominadas *cadenas de reubicación*) que la hacen degenerar rápidamente. Imaginemos el caso extremo en que $\forall k, k': k, k' \in K: h_i(k) = h_i(k') \Rightarrow h_{i+d}(k) = h_{i+d}(k')$, y sea una secuencia p_1, \dots, p_s de posiciones ocupadas por k_1, \dots, k_s tal que $\forall i: 2 \leq i \leq s: \exists j: 1 \leq j \leq i: h(k_i) = p_j$ y $h(k_1) = p_1$; en esta situación, la inserción de una nueva clave k , tal que $\exists i: 1 \leq i \leq s: h(k) = p_i$, implica que la posición destino de k es $h_{s+1}(k_1)$. Es decir, la posición $h_{s+1}(k_1)$ es el destino de gran cantidad de claves, por lo que su posibilidad de ocupación es muy grande; además, el efecto es progresivo puesto que, cuando se ocupe, la primera posición $h_{s+1+x}(k_1)$ libre tendrá todavía más probabilidad de ocupación que la que tenía $h_{s+1}(k_1)$.

A continuación, enumeramos los principales métodos de redistribución.

a) Redispersión uniforme

Método de interés principalmente teórico, que asocia a cada clave un camino distinto, que es una permutación de los elementos de Z_r ; los $r!$ caminos posibles son equiprobables. Evidentemente, es el mejor método posible; ahora bien, ¿cómo implementarlo?

b) Redispersión aleatoria

De manera parecida, si disponemos de una función capaz de generar aleatoriamente una secuencia de números a partir de una clave, tendremos una familia de funciones de redispersión casi tan buena como la uniforme aunque, a diferencia del esquema anterior, una misma posición puede generarse más de una vez en el camino de una clave.

De nuevo tenemos el problema de definir esta función aleatoria. Normalmente, hemos de conformarnos con un generador de números pseudoaleatorio. Otra opción consiste en construir una función *ad hoc* que manipule con contundencia la clave; por ejemplo, en [AHU83, pp.134-135] se muestra una simulación de la redispersión aleatoria mediante una manipulación de la clave con sumas, desplazamientos y o-exclusivos.

c) Redispersión lineal

Es el método de redispersión más intuitivo y fácil de implementar, se caracteriza porque la distancia entre $h_i(k)$ y $h_{i+1}(k)$ es constante para cualquier i .

$$h_i(k) = (h(k) + c \cdot i) \bmod r$$

Si c y r son primos entre sí, se van ensayando sucesivamente todas las posiciones de la tabla: $h(k)$, $h(k)+c \bmod r$, $h(k)+2c \bmod r$, ..., y si hay alguna posición vacía finalmente se encuentra; para asegurar el cumplimiento de esta propiedad independientemente del valor de r , normalmente se toma $c = 1$ (como en el ejemplo de la fig. 5.22).

Esta estrategia sencilla presenta, no obstante, el llamado *apiñamiento primario*: si hay dos claves k y k' tales que $h(k) = h(k')$, entonces la secuencia de posiciones generadas es la misma para k y para k' . Observamos que, cuanto más largas son las cadenas, más probable es que crezcan; además, la fusión de cadenas agrava el problema porque las hace crecer de golpe. El apiñamiento primario provoca una variancia alta del número esperado de ensayos al acceder a la tabla.

No obstante, la redispersión lineal presenta una propiedad interesante: la supresión no obliga a degenerar la tabla, sino que se pueden mover elementos para llenar espacios vacíos. Concretamente, al borrar el elemento que ocupa la posición p_i dentro de una cadena p_1, \dots, p_n , $i < n$, se pueden mover los elementos k_j que ocupan las posiciones p_j , $i < j \leq n$, a posiciones p_s , $s < j$, siempre que se cumpla que $s \geq i$, siendo $p_i = h(k_j)$. Es decir, al borrar un elemento se examina su cadena desde la posición que ocupa hasta al final y se mueven hacia adelante tantos elementos como sea posible; en concreto, un elemento se mueve siempre que la posición destino no sea anterior a su valor de dispersión (v. fig. 5.23). Esta técnica no sólo evita la degeneración espacial de la tabla sino que acerca los elementos a su valor de dispersión y, además, según la distribución de los elementos, puede partir las cadenas en dos. Ahora bien, la conveniencia del movimiento de elementos ha de ser estudiada cuidadosamente porque puede acarrear los problemas ya conocidos.

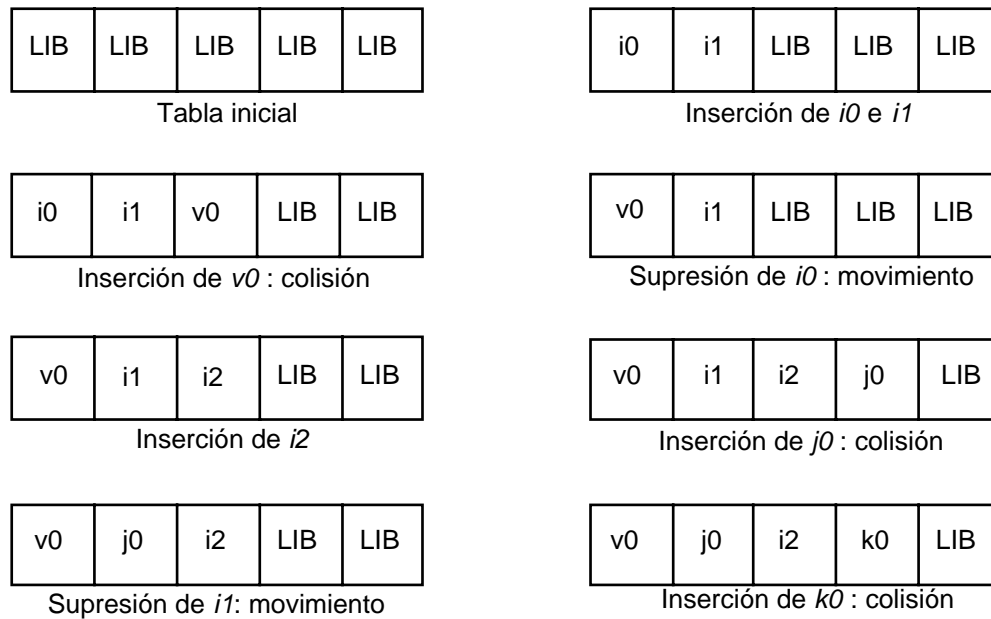


Fig. 5.23: *id.* fig. 5.22, pero con reubicación de elementos en la supresión.

d) Redispersión doble

La familia de redispersión se define a partir de una segunda función primaria de dispersión.

$$h_i(k) = (h(k) + i \cdot h'(k)) \bmod r$$

Si se quiere explorar toda la tabla, es necesario que $h'(k)$ produzca un valor entre 0 y $r-1$ que sea primo con r (lo más sencillo es elegir r primo, o bien $r = 2^s$, y que h' sólo genere números impares). La ventaja respecto el método anterior es que, como h' depende de la clave, es muy improbable que h y h' colisionen a la vez; ahora bien, si esto sucede, vuelve a formarse una cadena de reubicación, aunque en este caso el apiñamiento no es tan grave como en el caso lineal. Los resultados de este método se acercan a la redispersión uniforme o aleatoria de manera que, a efectos prácticos, son indistinguibles.

Tradicionalmente, la forma concreta de h' depende de la elección de h ; así, por ejemplo, si $h(k) = k \bmod r$, puede definirse $h'(k) = 1 + (k \bmod (r-2))$; idealmente, r y $r-2$ deberían ser primos (por ejemplo, 1021 y 1019). Otra posibilidad es tomar h' que no obligue a efectuar nuevas divisiones, como $h(k) = k \bmod r$ y $h'(k) = k / r$ (obligando que el resultado sea diferente de cero). Es importante, eso sí, que h y h' sean independientes entre sí para minimizar la probabilidad de que $h(k) = h(k')$ y a la vez $h'(k) = h'(k')$.

Notemos que, a diferencia del método lineal, no se pueden mover elementos, porque al borrarlos no se sabe trivialmente de qué cadenas forman parte.

e) Redispersión cuadrática

Es una variante del método lineal que usa el cuadrado del ensayo:

$$h_i(k) = (h(k) + i^2) \bmod r$$

Dado que la distancia entre dos valores consecutivos depende del número de ensayos, una colisión secundaria no implica la formación de cadenas siempre que $h(k) \neq h(k')$; ahora bien, si $h(k) = h(k')$, entonces las cadenas son idénticas.

La rapidez de cálculo puede mejorarse transformando el producto en sumas con las siguientes relaciones de recurrencia:

$$\begin{aligned} h_{i+1}(k) &= (h_i(k) + d_i) \bmod r, \text{ con } h_0(k) = 0 \\ d_{i+1} &= d_i + 2, \text{ con } d_0 = 1 \end{aligned}$$

Notemos que la expresión de h_{i+1} en función de h_i aconseja a cambiar la definición del universo *CLAVE_DISPERSIÓN*, pensando en una posterior implementación eficiente.

Tal como se ha definido, no se recorren todas las posiciones de la tabla; como máximo, si r es primo, la secuencia $h_i(k)$ puede recorrer $\lceil r/2 \rceil$ posiciones diferentes (dado que el ensayo i es igual al ensayo $r-i$, módulo r en ambos casos), por lo que la tabla puede tener sitios libres y la técnica de reubicación no los encuentre. En la práctica, este hecho no es demasiado importante, porque si después de $\lceil r/2 \rceil$ ensayos no se encuentra ningún sitio libre significa que la tabla está muy llena y debe regenerarse, pero en todo caso se puede solucionar el problema obligando a r , además de ser primo, a tomar la forma $4s + 3$ y redefiniendo la función como $h_i(k) = (h(k) + (-1)^i i^2) \bmod r$.

f) Otras

Siempre se pueden definir otras familias *ad hoc*, sobre todo aprovechando la forma concreta que tiene la función de dispersión primaria. Por ejemplo, dada la función descrita en el apartado 5.3.3, podemos definir una familia de redispersión tal que $h_i(k)$ se define como la aplicación de h sobre $repr(k)+i$, donde $repr(k)$ es la representación binaria de k (de manera similar al tratamiento de claves largas); esta estrategia genera todas las posiciones de la tabla.

5.4.3 Caracterización e implementación de los métodos de redispersión

Como se hizo con las funciones, es necesario encapsular en universos los diferentes métodos de redispersión que se han presentado. La caracterización de las funciones de redispersión ya ha sido definida en el universo *CLAVE_REDISPERSIÓN*; el siguiente paso consiste en caracterizar los métodos. Como ejemplos, estudiamos las familias lineal y doble.

El método lineal puede considerarse como un algoritmo parametrizado por una función de dispersión, tal como está definida en *CLAVE_DISPERSIÓN*, más la constante multiplicadora (v. fig. 5.24). Para definir una familia de redispersión lineal basta con concretar la función primaria de dispersión; en la fig. 5.25, tomamos la función definida en *SUMA_POND_Y_DIV*.

```

universo REDISPERSIÓN_LINEAL(CLAVE_REDISPERSIÓN_LINEAL) impl ...
  usa NAT
  función redispersión_lineal (i es nat; v es elem) devuelve nat es
    devuelve (h(v) + ct*i) mod val
funiverso

universo CLAVE_REDISPERSIÓN_LINEAL caracteriza
  usa CLAVE_DISPERSIÓN, NAT, BOOL
  ops ct: → nat
  ecns (ct = 0) = falso
        mcd(ct, val) = 1 { siendo mcd el máximo común divisor de dos números }
funiverso

```

Fig. 5.24: caracterización de la redispersión lineal.

```

universo REDISP_LINEAL_SUMA_POND_Y_DIV(E es ELEM_DISP_CONV;
                                         V1, CT son VAL_NAT) impl ...
instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV; V2 es VAL_NAT) donde
  F.elem es E.elem, F.item es E.item
  F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
  F.conv es E.conv, F.base es E.base; V2.val es V1.val
instancia REDISPERSIÓN_LINEAL(K es CLAVE_REDISPERSIÓN_LINEAL) donde
  K.elem es E.elem, K.item es E.item
  K.= es E.=, K.nítems es E.ítems, K.i_ésimo es G.i_ésimo
  K.conv es E.conv, K.base es E.base
  K.val es V1.val, K.h es suma_pond, K.ct es CT.val
renombra redispersión_lineal por redisp_lin_suma_pond_y_div
funiverso

```

Fig. 5.25: una instancia de la redispersión lineal.

Por lo que respecta al método doble, es necesario definir dos funciones de dispersión como parámetros formales (v. fig. 5.26). A continuación, se puede instanciar el universo parametrizado, por ejemplo con la función de suma ponderada y división, y separando el módulo de ambas funciones por dos (v. fig. 5.27).

universo REDISPERSIÓN_DOBLE (CLAVE_REDISPERSIÓN_DOBLE) impl ...
usa NAT
función redispersión_doble (i es nat; v es elem) devuelve nat es
devuelve (h(v) + h2(v)*i) mod val
funiverso
universo CLAVE_REDISPERSIÓN_DOBLE caracteriza
usa CLAVE_DISPERSIÓN, NAT, BOOL
ops h2: elem \rightarrow nat
ecns $\forall v \in \text{elem}: h2(v) < \text{val} = \text{cierto}$
funiverso

Fig. 5.26: caracterización de la redispersión doble.

universo REDISP_DOBLE_SUMA_POND_Y_DIV(E es ELEM_DISP_CONV;
V1, CT son VAL_NAT) impl ...
instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV; V2 es VAL_NAT) donde
F.elem es E.elem, F.item es E.item
F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
F.conv es E.conv, F.base es E.base; V2.val es V1.val
renombra suma_pond por hprim
instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV, V2 es VAL_NAT) donde
F.elem es E.elem, F.item es E.item
F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
F.conv es E.conv, F.base es E.base; V2.val es V1.val+2
renombra suma_pond por hsec
instancia REDISPERSIÓN_DOBLE(K es CLAVE_REDISPERSIÓN_DOBLE) donde
K.elem es E.elem, K.item es E.item
K.= es E.=, K.nítems es E.ítems, K.i_ésimo es E.i_ésimo
K.conv es E.conv, K.base es E.base
K.val es V1.val, K.h es hprim, K.h2 es hsec
renombra redispersión_doble por redisp_doble_suma_pond_y_div
funiverso

Fig. 5.27: una instancia de la redispersión doble.

5.4.4 Variantes de las tablas de direccionamiento abierto

Hay algunas variantes de la estrategia de direccionamiento abierto implementada en la fig. 5.20, que permiten reducir el coste de las búsquedas a costa de mover elementos al insertar. Así, estas variantes se pueden usar para satisfacer mejor los criterios de eficiencia de una aplicación dada, siempre que el coste de mover los elementos no sea elevado (es decir, que su dimensión no sea demasiado grande y que el tratamiento de la obsolescencia de los atajos, si los hay, no incumpla los requerimientos de eficiencia del problema). La implementación de los algoritmos resultantes queda como ejercicio para el lector; se puede consultar, por ejemplo, [TeA86, pp. 531-535].

a) Algoritmo de Brent

Estrategia útil cuando el número de consultas con éxito es significativamente más grande que el número de inserciones, de manera que en estas últimas se trabaja más para favorecer a las primeras. Su comportamiento es comparativamente mejor, especialmente a medida que la tabla está más llena. El algoritmo de Brent ha sido tradicionalmente asociado a la estrategia de redispersión doble, porque es cuando presenta los mejores resultados.

Su funcionamiento es el siguiente: si al insertar un nuevo par $\langle k, v \rangle$ se ha generado el camino p_0, \dots, p_s , definiendo el ensayo $p_i = (h(k) + i \cdot h'(k)) \bmod r$ y tal que p_s es la primera posición vacía de la secuencia, encontramos dos casos:

- Si $s \leq 1$, se actúa como siempre.
- Si no, sea k' la clave que reside en p_0 y sea $c_0 = h'(k')$.
 - ◊ Si la posición $(p_0 + c_0) \bmod r$ (que es la siguiente a p_0 en el camino asociado a k') está vacía, entonces se mueve k' a esta posición (con su información asociada) y se inserta el par $\langle k, v \rangle$ en p_0 . Notemos que, después de la inserción, el algoritmo de Brent determina $z+1$ ensayos para encontrar k' , pero sólo un ensayo para encontrar k , gracias al desalojo del invasor que residía en la posición de dispersión, mientras que sin usar el algoritmo el número de ensayos es z para encontrar k' y $s+1$ para encontrar k , $s \geq 2$. Así, pues, se gana como mínimo un acceso con esta estrategia.
 - ◊ Si la posición $(p_0 + c_0) \bmod r$ está llena, se aplica recursivamente la misma estrategia, considerando p_1 como el nuevo p_0 , p_2 como el nuevo p_1 , etc.

b) Dispersión ordenada

Si las claves de dispersión presentan una operación de comparación (que sería un parámetro formal más del universo de implementación), se pueden ordenar todos los elementos que residen en una misma cadena de reubicación de manera que las búsquedas sin éxito acaben antes. Igual que en el anterior método, esta variante se acostumbra a implementar con redispersión doble. Concretamente, al insertar el par $\langle k, v \rangle$ tal que $h(k) = i$ puede ocurrir que:

- La posición i esté vacía: se almacena el par en ella.
- La posición i contenga el par $\langle k, v' \rangle$: se sustituye v' por v .

- Si no, sea $\langle k', v' \rangle$ el par que reside en la posición i . Si $k' < k$, se repite el proceso examinando la siguiente posición en el camino de k ; si $k' > k$, se guarda $\langle k, v \rangle$ en la posición i y se repite el proceso sobre el par $\langle k', v' \rangle$ a partir del sucesor de i en el camino de k' .

c) Método Robin Hood

Ideado por P. Celis, P.-Å. Larso y J.I. Munro y publicado en *Proceedings 26th Annual Symposium on Foundations of Computer Science* (1985), está orientado a solucionar dos problemas que presentan los métodos de direccionamiento abierto vistos hasta ahora: por un lado, todos ellos aseguran el buen comportamiento esperado de las operaciones sobre tablas, pero no el de una operación individual; por el otro, pierden eficiencia cuando la tabla se llena. Una estrategia de inserción que recuerda vagamente a la variante de Brent ofrece ventajas en estos aspectos.

Consideremos el siguiente algoritmo de inserción: supongamos que los $j-1$ primeros ensayos de la inserción de una clave k no tienen éxito y que se consulta la posición p correspondiente al j -ésimo ensayo. Si esta posición p está vacía o contiene un par $\langle k, v \rangle$, se actúa como siempre; si no, está ocupada por un elemento k' que se depositó en la tabla en el i -ésimo ensayo, y en este caso se sigue la siguiente casuística:

- Si $i > j$, entonces k es rechazada por p y se sigue con el $j+1$ -ésimo ensayo.
- Si $i < j$, entonces k desplaza a k' de la posición p y es necesario insertar k' a partir del $i+1$ -ésimo ensayo.
- Si $i = j$, no importa.

La segunda de estas reglas da nombre al método, porque favorece la clave k más "pobre" (es decir, que exige más ensayos al buscarla) sobre la clave k' más "rica". El resultado es que, aunque el número medio de ensayos necesario para encontrar los elementos en la tabla no varía (al contrario que en el esquema de Brent), no hay claves extraordinariamente favorecidas ni tampoco perjudicadas (la variancia de la variable correspondiente queda $\Theta(1)$), ni siquiera cuando la tabla se llena. Eso sí, el método requiere que se sepa el número de ensayo de todo elemento de la tabla (ya sea con un campo adicional, ya sea implícitamente dada la familia de redispersión).

5.4.5 Tablas de dispersión coalescentes

Las *tablas de dispersión coalescentes* (ing., *coalesced hashing*) son una organización intermedia entre las encadenadas y el direccionamiento abierto. Por un lado, se encadenan los sinónimos y, opcionalmente, los sitios libres; por el otro, todos los elementos ocupan la misma zona dentro de un vector dimensionado de 0 a $r-1$, y la supresión presenta los problemas típicos del direccionamiento abierto.

En concreto, al insertar una clave k en la tabla se siguen los encadenamientos que salen de la posición $h(k)$ hasta que se encuentra la clave (si ya existía), o bien se llega al final de la cadena. En el primer caso se sustituye la información que había por la nueva, mientras que en el segundo se obtiene una posición libre, se inserta el par $\langle k, v \rangle$ y se encadena esta posición con la última de la cadena. La estrategia de inserciones provoca que las cadenas estén formadas por elementos con diferentes valores de dispersión; eso sí, todos los sinónimos de un mismo valor de dispersión forman parte de la misma cadena. Esta propiedad no provoca ninguna situación incorrecta, porque la longitud de las cadenas se mantiene corta incluso cuando los algoritmos provocan fusiones de cadenas, y asegura buenos resultados en las búsquedas. De hecho, los algoritmos podrían modificarse para evitar las fusiones y así tener cadenas sin sinónimos de diferentes claves, pero el beneficio sería tan escaso que no se justifica.

La organización coalescente presenta apiñamiento, porque todas las claves que tienen como valor de dispersión cualquier posición que forme parte de una cadena irán a parar a ella, y las cadenas largas tienen más probabilidad de crecer que las cortas.

Por lo que respecta a la supresión, se puede optar por marcar las posiciones borradas o por reubicar los sinónimos. En el primer caso, conviene reciclar las posiciones borradas en la inserción antes de ocupar nuevas posiciones libres. En la segunda opción, si hay sinónimos de diversos valores de dispersión dentro de una misma cadena, puede que no baste mover uno solo, porque este sinónimo puede dejar otro agujero dentro de la cadena que esconda otras claves posteriores. En consecuencia, en el caso general es necesario efectuar varios movimientos hasta dejar la cadena en buen estado. A veces, estos movimientos acaban fraccionando la cadena original en dos o más diferentes, y reducen el grado de apiñamiento que presenta la tabla. Eso sí, es necesario evitar mover una clave a una posición de la cadena anterior a su valor de dispersión.

Tal como se ha dicho, la gestión del espacio libre se puede hacer o no encadenada. En caso de formar la típica pila de sitios libres, es necesario encadenar doblemente los elementos, porque en todo momento se ha de poder ocupar de manera rápida cualquier posición libre. En caso de gestionarse secuencialmente, se dispondrá de un apuntador que inicialmente valga, por ejemplo, $r-1$ y que se actualize a medida que se inserten y borren elementos. Esta estrategia, aunque pueda parecer más ineficiente, no perjudica significativamente el tiempo de las inserciones o las supresiones.

Destaquemos que este método presenta una variancia muy baja en las búsquedas (v. [GoB91, p.62]; en este texto, se presenta también una variante que implementa las ideas de la dispersión coalescente usando una zona de excedentes para los sinónimos).

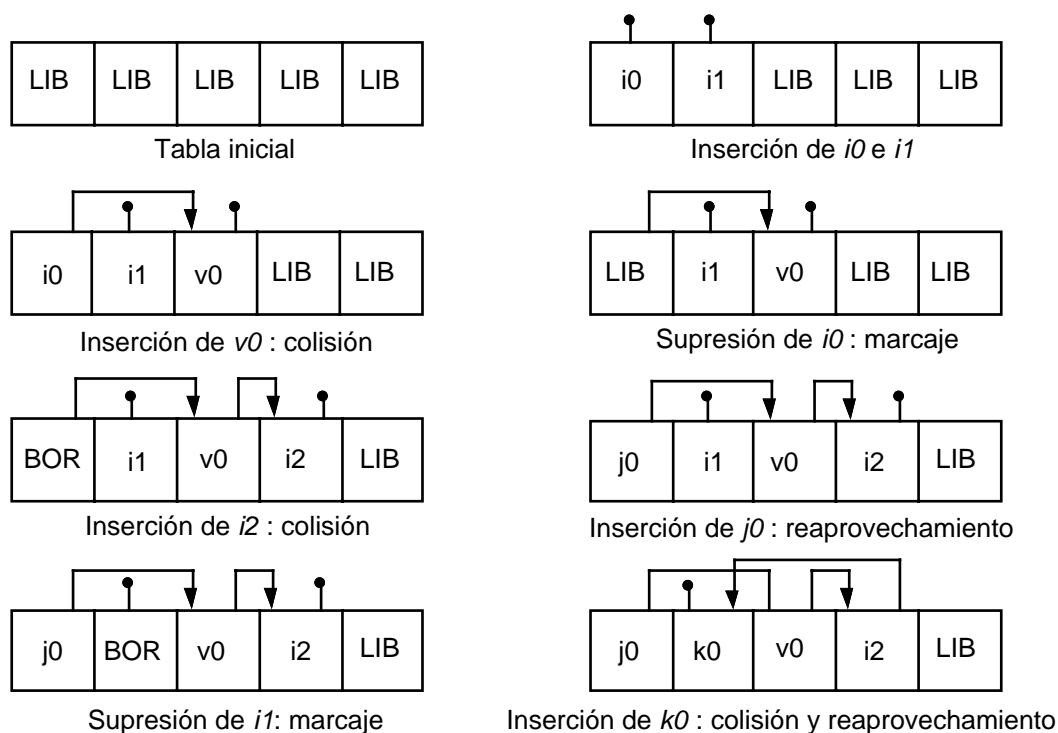


Fig. 5.28: Ídem fig. 5.22 en una tabla coalescente con marcaje en la supresión.

5.4.6 Evaluación de las diferentes organizaciones

Si bien asintóticamente todas las organizaciones presentan el mismo coste, a lo largo de esta sección hemos visto algunas diferencias que impactan en la eficiencia no asintótica, y en este apartado estudiamos este punto. En la fig. 5.29 se presentan las fórmulas que permiten comparar las diferentes estrategias. Para ello, se utilizan las magnitudes siguientes:

- $S_n(k)$: número necesario de comparaciones entre claves en una tabla de n elementos para encontrar el elemento de clave k (ing., *successful search*).
- $U_n(k)$: número necesario de comparaciones entre claves en una tabla de n elementos en una búsqueda sin éxito del elemento de clave k (ing., *unsuccessful search*).

Notemos que $S_n(k)$ es un factor importante en el cálculo del coste de la modificación y la supresión de elementos ya existentes y en la consulta de claves definidas, mientras que $U_n(k)$ determina el coste de la inserción de nuevos elementos dentro de la tabla, de la supresión de elementos dando como referencia una clave indefinida y de la consulta de claves indefinidas. Aunque puede haber otros factores que influyen en la eficiencia (por ejemplo, seguimiento de encadenamientos, movimiento de elementos, etc.), estas dos magnitudes son las más representativas de las diferentes organizaciones.

En concreto, se estudian las esperanzas de las magnitudes $S_n(k)$ y $U_n(k)$. Por motivos de espacio, no se incluye la deducción de las fórmulas, pero hay varios libros y artículos que sí las presentan, entre los que destacan [Knu73] y [GoB91]. Los resultados obtenidos dependen de la tasa de ocupación de la tabla, α , denominada *factor de carga* (ing., *load factor*), definido como el cociente n/r , siendo n el número de elementos en la tabla más las posiciones marcadas como borradas (si la estrategia tiene), y Z_r el codominio de la función de dispersión. Cuanto más grande es α , peor se comporta la tabla, y la mayoría de las organizaciones presentan un punto a partir del cual la degeneración es demasiado acusada para que la tabla valga la pena.

En estos resultados, se ha supuesto que las funciones de dispersión distribuyen uniformemente el dominio de las claves en el intervalo Z_r . Es necesario recordar que todas las organizaciones pueden comportarse tan mal como se pueda temer, para un subconjunto de claves determinado que provoque un mal funcionamiento de la función de dispersión.

5.4.7 Elección de una organización de dispersión

Una vez conocidas las características de las diferentes organizaciones de tablas de dispersión, es lógico preguntarse cuáles son los criterios que conducen a la elección de una u otra en un contexto determinado. Destacan los siguientes puntos:

- Tiempo. A la vista de las fórmulas de la fig. 5.29 está claro que, a igual factor de carga, las estrategias encadenadas dan mejores resultados que las otras. Ahora bien, en algún momento puede merecer la pena incrementar la dimensión de una tabla no encadenada para disminuir este factor y, consecuentemente, el tiempo de acceso. Eso sí, en cualquier caso el coste asintótico será siempre $\Theta(1)$ para factores de carga razonables, suponiendo que la función de dispersión distribuya las claves insertadas en la tabla de manera uniforme.
- Espacio. Si se desconoce el número esperado de elementos, es recomendable usar la estrategia encadenada indirecta implementada con punteros, donde sólo se desaprovecha *a priori* el espacio del índice y, posteriormente, los encadenamientos; además, las estrategias encadenadas soportan factores de carga superiores al 100% sin que la eficiencia temporal se resienta⁶. Ahora bien, si se conoce con cierta exactitud el número de elementos esperado en la tabla, es necesario recordar que las estrategias de direccionamiento abierto no precisan encadenamientos; eso sí, siempre se deberá dimensionarlas en exceso para evitar las casi-asíntotas existentes a partir de factores de carga del orden del 60%.

⁶ Si el desconocimiento es total, puede ser desaconsejable el uso de dispersión, pues esta técnica exige conjeturar el valor de r , y una mala elección lleva a desperdiciar mucho espacio en caso de sobredimensionar vectores, o bien a regenerar la tabla en caso de llegar a un factor de carga prohibitivo. En este mismo capítulo se muestra una alternativa a la dispersión, los árboles de búsqueda, que se adaptan mejor a esta situación, sacrificando algo de eficiencia temporal.

Dispersión encadenada indirecta:

$$\varepsilon[S_n(k)] \approx \alpha/2$$

$$\varepsilon[U_n(k)] \approx \alpha$$

Dispersión encadenada con zona de excedentes:

$$\varepsilon[S_n(k)] \approx \alpha/2$$

$$\varepsilon[U_n(k)] \approx \alpha + e^{-\alpha}$$

Redispersión uniforme:

$$\varepsilon[S_n(k)] \approx -\alpha^{-1} \ln(1-\alpha); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma - 1 + o(1) \quad (\gamma = 0.577\dots)$$

$$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}; \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Redispersión aleatoria:

$$\varepsilon[S_n(k)] \approx -\alpha^{-1} \ln(1-\alpha) + O((r-n)^{-1}); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1/r)$$

$$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}; \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Redispersión lineal:

$$\varepsilon[S_n(k)] \approx 0.5(1+(1-\alpha)^{-1}); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx (\pi r/8)^{1/2}$$

$$\varepsilon[U_n(k)] \approx 0.5(1+(1-\alpha)^{-2}); \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Redispersión doble:

$$\varepsilon[S_n(k)] \approx -\alpha^{-1} \ln(1-\alpha) + O(1); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1)$$

$$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1} + o(1); \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Redispersión cuadrática:

$$\varepsilon[S_n(k)] \approx 1 - \ln(1-\alpha) - \alpha/2$$

$$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1} - \ln(1-\alpha) - \alpha; \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Algoritmo de Brent:

$$\varepsilon[S_n(k)] \approx 1 + \alpha/2 + \alpha^3/3 + \alpha^4/15 - \alpha^5/18 + \dots; \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx 2.49$$

$$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}; \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$$

Dispersión ordenada:

$$\varepsilon[S_n(k)] \approx -\alpha^{-1} \ln(1-\alpha) + o(1); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1)$$

$$\varepsilon[U_n(k)] \approx -\alpha^{-1} \ln(1-\alpha); \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx \ln(r+1) + \gamma + O(1/(r+1))$$

Variante Robin Hood:

$$\varepsilon[S_n(k)] = O(1); \alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] < 2.57 \text{ (usando búsquedas no tradicionales)}$$

$$\varepsilon[U_n(k)] \approx \text{desconocido (pero } < \log r); \alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx \log r \text{ (idem)}$$

Dispersión coalescente:

$$\varepsilon[S_n(k)] \approx 1 + (e^{2\alpha} - 1 - 2\alpha)/4 + O(1/r)$$

$$\varepsilon[U_n(k)] \approx 1 + (e^{2\alpha} - 1 - 2\alpha)/8\alpha + \alpha/4$$

Fig. 5.29: fórmulas de los diversos métodos de dispersión.

- Frecuencia de las supresiones. Si hay muchas supresiones la política encadenada indirecta generalmente es mejor, dado que cualquier otra exige, o bien movimientos de elementos, o bien marcaje de posiciones. Si los movimientos son desdeñables (si no se utilizan atajos para acceder a la estructura y los elementos no son de un tamaño demasiado grande), o bien si las supresiones son escasas o inexistentes, este factor no influye en la elección del tipo de tabla.

A la vista de los resultados, parece claro que la mayoría de las veces podemos decantarnos por la estrategia indirecta, que es rápida, soporta perfectamente las supresiones, no exige un conocimiento exacto del número de elementos y soporta desviaciones en la previsión de ocupación; además, su programación es trivial. Sólo queda por decidir si las listas se implementan en memoria dinámica o, por el contrario, en un único vector. Ahora bien, bajo ciertos requerimientos pueden ser más indicadas otras estrategias; veamos un ejemplo.

Se quiere implementar los conjuntos de naturales con operaciones de conjunto vacío, añadir un elemento y pertenencia (v. fig. 1.29), y con la propiedad de que, a la larga, se espera que vayan a almacenarse n naturales en el conjunto; este tipo de conjuntos puede emplearse en algoritmos que registran los elementos a los que se ha aplicado un tratamiento determinado. Dado que no hay operación de supresión, no se descarta inicialmente ningún método. Por otro lado, como el tamaño de los elementos es muy pequeño y el número que habrá es conocido, la estrategia de direccionamiento abierto puede ser la indicada, puesto que evita el espacio requerido por los encadenamientos. Ahora bien, sería incorrecto dimensionar la tabla de n posiciones porque, al llenarse, el tiempo de acceso a los conjuntos sería muy grande. Consultando las fórmulas, vemos que un factor de carga del 60% da resultados razonables siempre que la función de dispersión distribuya bien.

Para confirmar que la elección es adecuada, es necesario comparar la sobredimensión de la tabla con el ahorro respecto a las otras estrategias. Supongamos que un encadenamiento ocupa el mismo espacio z que un natural; en este caso, las diferentes estrategias necesitan el siguiente espacio:

- Direccionamiento abierto: la tabla tiene $(100/60)n$ posiciones y cada posición ocupa z .
- Coalescente: la tabla tiene, como mínimo, n posiciones y cada posición ocupa $2z$ (natural + encadenamiento).
- Encadenado indirecto: el vector índice ocupa del orden de n posiciones de espacio z cada una de ellas, y cada uno de los n elementos del conjunto ocupará $2z$.
- Encadenado directo: el vector ocupa del orden de $(100/86)n$ posiciones y cada posición ocupa $2z$.

Comparando estas magnitudes puede confirmarse que el direccionamiento abierto presenta mejores resultados; incluso se podría aumentar la dimensión del vector hasta $2n$ para asegurar un buen comportamiento si la función no distribuye del todo bien. Si el contexto de

uso de la tabla lo exige, se puede optar por una de las variantes presentadas en el apartado 5.4.4; no hay problema en mover elementos, porque son pequeños y no hay atajos a ellos.

5.4.8 Las organizaciones de dispersión en tablas recorribles y tablas abiertas

Comentamos brevemente las modificaciones a efectuar sobre las organizaciones de dispersión vistas en el caso de los TAD de tablas recorribles y tablas abiertas.

En el caso de las tablas recorribles, el recorrido no requiere ningún orden de visita de los elementos. La estrategia más sencilla consiste en no modificar la representación más que por el añadido del apuntador al elemento actual dentro del recorrido, de manera que el recorrido sea simplemente un examen secuencial de la tabla y quede de orden $\Theta(t+n)$, siendo t la dimensión del vector que aparece en la tabla (que vale r en todos los casos excepto en la dispersión encadenada directa); el factor n es necesario para cubrir el caso indirecto, donde el número de elementos puede ser superior a t . En esta opción, las inserciones y/o supresiones incontroladas pueden provocar que un recorrido no examine todos los elementos de la tabla. Ahora bien, ya hemos comentado que las operaciones de recorrido no se alternan normalmente con las actualizaciones y, por ello, este problema raramente es relevante.

Esta implementación, conceptualmente tan simple, choca de nuevo con la sobre-especificación que caracteriza la semántica inicial, dado que el orden de recorrido no está en absoluto determinado. Normalmente, este problema es más formal que real; precisamente estamos diciendo que no nos importa el orden de obtención de los elementos de la tabla, tan sólo nos interesa obtenerlos todos. Si aún así nos interesa construir una implementación que sí cumpla la especificación en el marco de la semántica inicial, podemos encadenar los elementos tal como es habitual en las estructuras con punto de interés y mantenerlos en orden de inserción⁷, obteniendo así una tabla recorrible ordenadamente según su estructura. Con esta estrategia, la inserción queda $\Theta(1)$ mientras que en la supresión existen dos alternativas: encadenar los elementos en los dos sentidos, de manera que la operación queda constante, o bien dejar encadenamientos simples, y en este caso la operación queda de orden lineal porque, si bien el elemento a borrar puede localizarse por dispersión, su predecesor en la lista no. Generalmente, esta segunda opción no cumple los requisitos de eficiencia temporal que nos han impulsado a usar tablas de dispersión.

El esquema se complica en el caso de las tablas recorribles ordenadamente según el valor de sus elementos. Es evidente que, para que el recorrido ordenado no sea excesivamente lento, es necesario que los elementos de la tabla estén ordenados según su clave al empezar el recorrido. Existen dos opciones ya conocidas:

⁷ Destaquemos que, en este caso, dos tablas con los mismos elementos pero con estado diferente de recorrido son diferentes en el marco de la semántica inicial.

- Se puede tener la tabla desordenada mientras se insertan y se borran elementos y ordenarla justo antes de comenzar el recorrido. La ventaja de este esquema es que las operaciones propias de las tablas quedan $\Theta(1)$ en todos los casos (exceptuando la creación), aplicando dispersión, y también quedan eficientes el resto de operaciones excepto *principio*, que ha de ordenar los elementos de la tabla; con un buen algoritmo de ordenación, quedaría $\Theta(n \log n)$, siendo n el número de elementos de la estructura, y éste sería el coste total del recorrido. Evidentemente, la ordenación ha de evitar movimientos de elementos para no destruir la estructura de dispersión.
- Como alternativa, se pueden mantener los elementos de la tabla siempre ordenados; esto implica que, para cada inserción, es necesario buscar secuencialmente la posición donde ha de ir el elemento y que resultará un orden lineal, pero las operaciones de recorrido quedarán constantes y así el recorrido entero queda $\Theta(n)$. Por lo que respecta a la supresión, queda igual que en el caso desordenado.

La elección de una alternativa concreta depende de la relación esperada entre altas, bajas y recorridos ordenados de la tabla. En ambos casos, sin embargo, la búsqueda de un elemento dada su clave es inmediata por dispersión.

Por lo que respecta a las variantes abiertas, debe tenerse en cuenta que diversas estrategias de dispersión mueven elementos en las inserciones y supresiones, por lo que es conveniente tratar la posible obsolescencia de los atajos, de manera similar a como se ha comentado en el apartado 2.4.3.

5.4.9 Inconvenientes de la dispersión

Hemos constatado la tremenda utilidad de la estrategia de implementación de las tablas por dispersión, que nos permite acceder a los elementos individualmente con un coste constante. Ahora bien, la organización por dispersión presenta algunas características negativas que han ido apareciendo a lo largo del capítulo y que resumizamos a continuación:

- El acceso a los elementos de la tabla puede dejar de ser constante si la función de dispersión no los distribuye bien en las diferentes cubetas. Para controlar este mal funcionamiento es necesario añadir código de control a la implementación de la tabla y, si se detecta, se ha de redefinir la función y volver a insertar los elementos.
- Se ha de determinar el número aproximado de elementos que se espera guardar en la tabla, aun cuando no se tenga la más remota idea. Si, con el tiempo, la tabla queda pequeña, es necesario agrandarla, redefinir la función de dispersión y reinsertar los elementos (a no ser que se usen métodos incrementales que no se han tratado en este texto); si se revela excesiva, se desperdiciará espacio de la tabla.
- En el contexto de las tablas recorribles ordenadamente hay alguna operación forzosamente lineal: o bien la inserción, si se mantiene siempre una estructura

encadenada ordenada de los elementos, o bien la preparación del recorrido, si se mantiene la estructura sin ningún orden y se ordena previamente al recorrido (en concreto, el coste del último caso es casi lineal).

- Varias organizaciones mueven elementos en las actualizaciones con los problemas que ello provoca como mínimo en las variantes abiertas del TAD.
- Es necesario realizar un estudio cuidadoso para decidir cuál es la organización de dispersión adecuada para el contexto concreto de uso (v. apartado 5.4.7).

En el resto del capítulo, se introduce una técnica de representación de las tablas mediante árboles que permite recorridos ordenados eficientes sin que ninguna operación quede lineal. Cada elemento tendrá asociados como mínimo dos encadenamientos y un campo adicional de control, de manera que ocupará más en esta nueva estructura, pero como no será necesario dimensionar *a priori* ningún vector, es posible que el espacio total resultante no sea demasiado diferente (v. ejercicio 5.16). Estos árboles se denominan árboles binarios de búsqueda, de los que además estudiaremos una mejora, los árboles AVL.

5.5 Árboles binarios de búsqueda

Sea V un dominio de elementos y sea $<$ la operación de comparación que define un orden total (para simplificar las explicaciones, usaremos también el resto de operadores relacionales). Un *árbol binario de búsqueda* (ing., *binary search tree*) es un árbol binario etiquetado con los elementos de V tal que, o bien es el árbol vacío, o bien su raíz es mayor que todos los elementos de su subárbol izquierdo (si tiene) y menor que todos los elementos de su subárbol derecho (si tiene) y, además, sus subárboles izquierdo y derecho son también árboles de búsqueda (si existen). De manera más formal, definimos A^c_V como los árboles binarios de búsqueda sobre V , $A^c_V \subseteq A^2_V$:

- $f_\emptyset \in A^c_V$ (recordemos que el árbol f_\emptyset cumple la condición $\text{dom}(f_\emptyset) = \emptyset$)
- $\forall a, a': a, a' \in A^c_V$:

$$\forall v: v \in V \wedge (\text{dom}(a) \neq \emptyset \Rightarrow (\text{máx } \langle s, n \rangle: \langle s, n \rangle \in N_a: n) < v) \wedge$$

$$(\text{dom}(a') \neq \emptyset \Rightarrow v < (\text{mín } \langle s, n \rangle: \langle s, n \rangle \in N_{a'}: n)): \text{enraiza}(a, v, a') \in A^c_V$$

En la fig. 5.30 se muestran tres árboles binarios con etiquetas naturales; el de la izquierda y el del medio son de búsqueda, pero no el de la derecha, ya que su subárbol izquierdo contiene un elemento, el 9, mayor que la raíz. Los dos árboles de búsqueda contienen los mismos naturales. En general, se pueden formar muchos árboles de búsqueda con los mismos elementos (v. ejercicio 5.9).

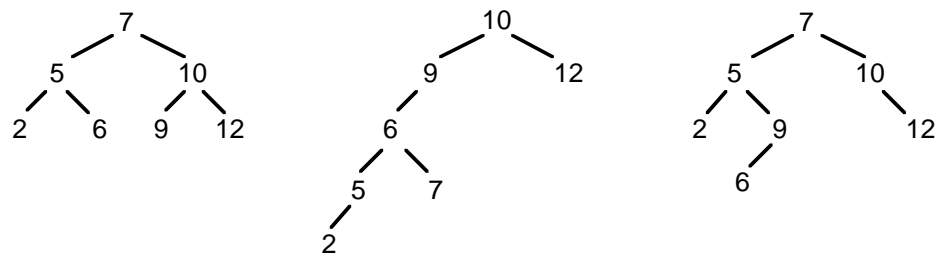


Fig. 5.30: dos árboles de búsqueda con los mismos elementos y uno que no lo es (der.).

La propiedad que caracteriza los árboles de búsqueda es que, independientemente de su forma, su recorrido en inorden proporciona los elementos ordenados precisamente por la relación exigida entre la raíz del árbol y las raíces de sus subárboles. Esta propiedad se constata en los dos árboles de búsqueda de la fig. 5.30, y se podría demostrar fácilmente por inducción sobre la forma de los árboles a partir de su definición recursiva (queda como ejercicio para el lector). La adecuación de los árboles de búsqueda como implementación de las tablas recorribles ordenadamente se basa en este hecho, que permite obtener los n elementos de la tabla en $\Theta(n)$, ya sea con árboles enhebrados o no. A continuación, es necesario estudiar el coste de las operaciones de acceso individual a la tabla para acabar de determinar la eficiencia temporal de la estructura.

Las operaciones de acceso individual se basan en la búsqueda de un elemento en el árbol y se rigen por un esquema bastante evidente. Sea $a \in A^c_V$ un árbol de búsqueda y sea $v \in V$ el elemento a buscar:

- Si a es el árbol vacío f_\emptyset , se puede afirmar que v no está dentro del árbol.
- En caso contrario, se comparará v con la raíz de a y habrá tres resultados posibles:
 - ◊ $v = \text{raíz}(a)$: el elemento ha sido encontrado en el árbol.
 - ◊ $v < \text{raíz}(a)$: se repite el proceso dentro del subárbol izquierdo de a .
 - ◊ $v > \text{raíz}(a)$: se repite el proceso dentro del subárbol derecho de a .

a) Inserción en un árbol binario de búsqueda

Para que el árbol resultante de una inserción en un árbol de búsqueda sea también de búsqueda, se aplica la casuística descrita para localizar el elemento. Si se encuentra, no es necesario hacer nada, si no, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en caso de existir). En la fig. 5.31 se muestra la inserción de dos elementos dentro de un árbol de búsqueda; en los dos casos y como siempre sucede, el nuevo elemento se inserta como una hoja, dado que precisamente la búsqueda acaba sin éxito cuando se accede a un subárbol izquierdo o derecho que está vacío.

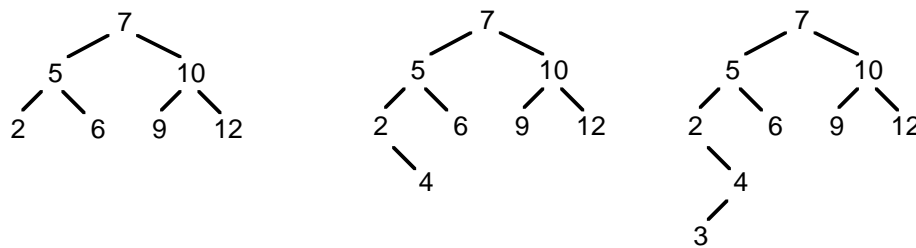


Fig. 5.31: inserción del 4 (centro) y del 3 (derecha) en el árbol de búsqueda de la izquierda.

Si el elemento a insertar no está en el árbol, el algoritmo de inserción recorre un camino desde la raíz hasta una hoja; obviamente, el camino más largo tiene como longitud el número de niveles del árbol. Si tenemos suerte y el árbol es casi completo, el número de niveles es logarítmico. No obstante, en el caso peor el número de niveles de un árbol binario de búsqueda de n elementos es n , ya que no hay ninguna propiedad que restrinja la forma del árbol y, por ello, el coste resultante puede llegar a ser lineal; el caso peor se da en la inserción ordenada de los elementos que resulta en un árbol completamente degenerado, donde cada nodo tiene un hijo y sólo uno. Se puede demostrar que el número esperado de niveles de un árbol binario de búsqueda después de insertar n nodos es asintóticamente logarítmico, aproximadamente $2(\ln n + \gamma + 1)$, siendo $\gamma = 0.577\dots$ la constante de Euler y suponiendo que el orden de inserción de los n nodos es equiprobable [Wir86, pp. 214-217].

b) Supresión en un árbol binario de búsqueda

Para localizar un elemento v dentro del árbol se aplica el algoritmo de búsqueda habitual. Si el elemento no se encuentra, la supresión acaba; de lo contrario, el comportamiento exacto depende del número de hijos que tiene el nodo $n = \langle s, v \rangle$ que contiene el elemento:

- Si n es una hoja, simplemente desaparece.
- Si n tiene un único subárbol, se sube éste a la posición que ocupa n .
- Si n tiene dos hijos, ninguno de los dos comportamientos descritos asegura la obtención de un árbol binario de búsqueda, sino que es necesario mover otro nodo del árbol a la posición s . ¿Cuál? Para conservar la propiedad de los árboles de búsqueda, se mueve el mayor de los elementos más pequeños que v (que está dentro del subárbol izquierdo del árbol que tiene n como raíz), o bien el menor de los elementos más grandes que v (que está dentro del subárbol derecho del árbol que tiene n como raíz). En cualquier caso, el nodo n' que contiene el elemento que responde a esta descripción es, o bien una hoja, o bien un nodo con un único hijo, de manera que a continuación se le aplica el tratamiento correspondiente en estos casos. En la fig. 5.32 se muestran las supresiones de los elementos 6, 2 y 7 en el árbol de la fig. 5.31, derecha. Cada una de las supresiones se corresponde con los tres casos citados; en particular, la supresión del 7 lleva al menor de los mayores al nodo que lo

contiene. Por lo que respecta a la eficiencia temporal, pueden repetirse los razonamientos hechos en la inserción.

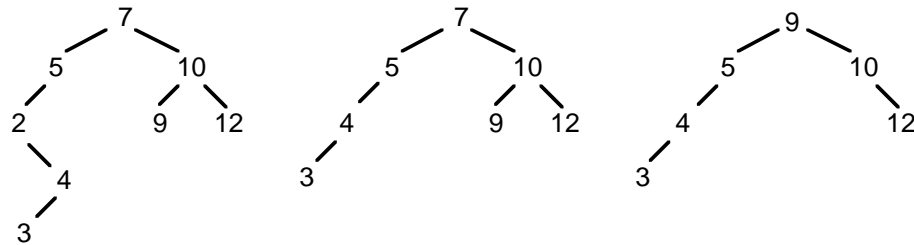


Fig. 5.32: supresión del 6 (izq.), 2 (centro) y 7 (der.) en el árbol de la fig. 5.31, derecha.

En la fig. 5.33 se muestra una implementación por árboles de búsqueda de las tablas recorribles ordenadamente que sigue la estrategia dada. Para simplificar la discusión, se ha escogido la especificación de las tablas recorribles con una única operación que devuelve la lista ordenada de los elementos (v. fig. 5.5); queda como ejercicio para el lector una versión usando iteradores. Observemos que no es necesario manipular directamente la representación de los árboles binarios, sino que simplemente se instancia un universo adecuado que ofrezca operaciones de recorrido y, a continuación, se restringen los valores posibles estableciendo un invariante, que caracteriza la propiedad de los árboles de búsqueda con el uso de dos funciones auxiliares para obtener el máximo y el mínimo de un árbol de búsqueda; notemos que las propiedades sobre la forma del árbol binario son implicadas por las propiedades exigidas a los árboles de búsqueda. La instancia es privada, porque las operaciones propias de los árboles binarios no han de ser accesibles al usuario del tipo. Las instancias de los universos para definir pares y listas ya se hicieron en la especificación y no es necesario repetirlas. Por lo que respecta a las operaciones, se implementan recursivamente, dado que es la manera natural de traducir el comportamiento que se acaba de describir; no obstante, el algoritmo de supresión presenta un par de ineficiencias al borrar un nodo que tiene dos hijos, que no se eliminan del algoritmo para mayor claridad en la exposición: por un lado, para obtener el menor de los mayores se hace un recorrido en inorden del subárbol derecho, cuando se podría simplemente bajar por la rama correspondiente; por otro, para borrar este elemento se vuelve a localizar recursivamente con *borrar*; la solución de estas ineficiencias (que no afectan al coste asintótico) queda como ejercicio para el lector. Notemos que las repetidas invocaciones a *enraizar* provocan cambios en la posición física de las etiquetas (aunque la forma del árbol no varíe), lo cual puede ser un inconveniente; la modificación de la codificación para evitar este movimiento exige manipular las direcciones físicas de los nodos como mínimo en la supresión de nodos con dos hijos, por lo que debería escribirse la representación del tipo explícitamente en el universo.

universo ÁRBOL_BINARIO_DE_BÚSQUEDA (A es ELEM_<_ = , B es ELEM_ESP) es
implementa
 FUNCIÓN_TOTAL_RECORRIBLE_ORD (A es ELEM_<_ = , B es ELEM_ESP)
renombra A.elem por clave, B.elem por valor, B.esp por indef
usa BOOL
instancia privada ÁRBOL_BINARIO_CON_RECORRIDOS (C es ELEM)
donde C.elem es elem_tabla
renombra árbol por árbol_búsqueda
tipo tabla es árbol_búsqueda ftipo
invariante (T es tabla): correcto(T)
 donde el predicado *correcto: árbol_búsqueda → bool* se define:
 correcto(crea) = cierto

$$\text{correcto}(\text{enraiza}(a_1, v, a_2)) = \text{correcto}(a_1) \wedge \text{correcto}(a_2) \wedge$$

$$\{ \neg \text{vacío?}(a_1) \Rightarrow v.\text{clave} > \text{máximo}(a_1) \} \wedge$$

$$\{ \neg \text{vacío?}(a_2) \Rightarrow v.\text{clave} < \text{mínimo}(a_2) \}$$
 y la función *máximo: árbol_búsqueda → elem* (y simétricamente *mínimo*) es:

$$\text{vacío?}(\text{hder}(a)) \Rightarrow \text{máximo}(a) = \text{raíz}(a).\text{clave}$$

$$\neg \text{vacío?}(\text{hder}(a)) \Rightarrow \text{máximo}(a) = \text{máximo}(\text{hder}(a))$$

función crea devuelve tabla es
devuelve ÁRBOL_BINARIO_CON_RECORRIDOS.crea

función todos_ordenados (T es tabla) devuelve lista_elem_tabla es
devuelve ÁRBOL_BINARIO_CON_RECORRIDOS.inorden(T)

función asigna (a es tabla; k es clave; v es valor) devuelve tabla es
 si vacío?(a) entonces {se crea un árbol de un único nodo}
 a := enraiza(crea, <k, v>, crea)
 si no
 opción
 caso raíz(a).clave = k hacer {se sustituye la información asociada}
 a := enraiza(hizq(a), <k, v>, hder(a))
 caso raíz(a).clave < k hacer {debe insertarse en el subárbol derecho}
 a := enraiza(hizq(a), raíz(a), asigna(hder(a), k, v))
 caso k < raíz(a).clave hacer {debe insertarse en el subárbol izquierdo}
 a := enraiza(asigna(hizq(a), k, v), raíz(a), hder(a))
 fopción
 fsi
devuelve a

Fig. 5.33: implementación de las tablas recorribles ordenadamente usando un árbol binario de búsqueda.


```

función borra (a es tabla; k es clave) devuelve tabla es
var temp es elem_tabla fvar
  si  $\neg$ vacío?(a) entonces {si el árbol es vacío, no es necesario hacer nada}
    opción
      caso raíz(a).clave = k hacer {hay cuatro casos posibles}
        opción
          caso vacío?(hizq(a))  $\wedge$  vacío?(hder(a)) hacer {el nodo es una hoja}
            a := crea
          caso  $\neg$ vacío?(hizq(a))  $\wedge$  vacío?(hder(a)) hacer {se sube el hijo izquierdo}
            a := hizq(a)
          caso vacío?(hizq(a))  $\wedge$   $\neg$ vacío?(hder(a)) hacer {se sube el hijo derecho}
            a := hder(a)
          caso  $\neg$ vacío?(hizq(a))  $\wedge$   $\neg$ vacío?(hder(a)) hacer {el nodo tiene dos hijos}
            {se busca el menor de los mayores, temp}
            temp := actual(principio(inorden(hder(a))))
            {se sube a la raíz y se borra del subárbol donde reside}
            a := enraiza(hizq(a), temp, borra(hder(a), temp.clave))
        fopción
      caso raíz(a).clave < k hacer {es necesario suprimir en el subárbol derecho}
        a := enraiza(hizq(a), raíz(a), borra(hder(a), k))
      caso k < raíz(a).clave hacer {es necesario suprimir en el subárbol izquierdo}
        a := enraiza(borra(hizq(a), k), raíz(a), hder(a))
    fopción
  fsi
devuelve a

función consulta (a es tabla; k es clave) devuelve valor es
var v es valor fvar
  si vacío?(a) entonces v := indef {la clave no está definida}
  si no
    opción
      caso raíz(a).clave = k hacer v := raíz(a).valor
      caso raíz(a).clave < k hacer v := consulta(hder(a), k)
      caso raíz(a).clave > k hacer v := consulta(hizq(a), k)
    fopción
  fsi
devuelve v

funiverso

```

Fig. 5.33: implementación de las tablas recorribles ordenadamente usando un árbol binario de búsqueda (cont.).

5.6 Árboles AVL

Como ya se ha explicado, la eficiencia temporal de las operaciones de acceso individual a los elementos del árbol depende exclusivamente de su altura y, en caso de mala suerte, puede llegar a ser lineal. En este apartado se introduce una variante de árbol que asegura un coste logarítmico, ya que reduce el número de niveles de un árbol binario de n nodos a $\Theta(\log n)$, el mínimo posible.

Hay diversas técnicas que aseguran este coste logarítmico sin exigir que el árbol sea casi completo (lo que llevaría a algoritmos demasiado complicados y costosos). Por ejemplo, los denominados *árboles 2-3* son una clase de árboles no binarios, sino ternarios, que obligan a que todas las hojas se encuentren al mismo nivel; su extensión a una aridad cualquiera son los *árboles B* y sus sucesores B^* y B^+ , muy empleados en la implementación de ficheros indexados. En este texto, no obstante, estudiamos otra clase de árboles (estos sí, binarios) que se definen a partir de los conceptos y algoritmos hasta ahora introducidos: los *árboles AVL* (iniciales de sus creadores, G.M. Adel'son-Vel'skii y E.M. Landis, que los presentaron en el año 1962 en una publicación soviética), que son árboles de búsqueda equilibrados.

Diremos que un árbol binario está *equilibrado* (ing., *height balanced* o, simplemente, *balanced*) si el valor absoluto de la diferencia de alturas de sus subárboles es menor o igual que uno y sus subárboles también están equilibrados. Por ejemplo, los árboles izquierdo y derecho de la fig. 5.30 son equilibrados; el de la izquierda, porque es completo, y el de la derecha, porque sus desequilibrios no son lo bastante acusados como para romper la definición dada. En cambio, el árbol central de la misma figura es desequilibrado: por ejemplo, su subárbol izquierdo tiene altura 4 y el derecho 1.

Los árboles AVL aseguran realmente un coste logarítmico a sus operaciones de acceso individual; en [HoS94, p. 520-521] y [Wir86, pp. 218-219] se deduce este coste a partir de una formulación recursiva del número mínimo de nodos de un árbol AVL de altura h , que puede asociarse a la sucesión de Fibonacci. Los mismos Adel'son-Vel'skii y Landis demuestran que la altura máxima de un árbol AVL de n nodos está acotada por $\lceil 1.4404 \log_2(n+2) - 0.328 \rceil$ (aproximadamente $1.5 \log n$), es decir, orden logarítmico; los árboles que presentan esta configuración sesgada se denominan *árboles de Fibonacci*, también debido al parecido con la sucesión correspondiente. Por lo que respecta al caso medio (el caso mejor es, obviamente, el árbol perfectamente completo), estudios empíricos apuntan a que la altura de un árbol AVL de n nodos, usando el algoritmo de inserción que se presenta a continuación, es del orden de $\lceil (\log n) + 0.25 \rceil$, suponiendo que el orden de inserción de los elementos sea aleatorio [Wir86, pp. 223-224].

En la fig. 5.34 se muestra una representación de las tablas con árboles AVL implementados con punteros. Observamos que en este caso no reaprovechamos el TAD de los árboles binarios dado que es necesario guardar información sobre el equilibrio del árbol dentro de los

nodos. Concretamente se introduce un tipo por enumeración, que implementa el concepto de *factor de equilibrio* (ing., *balance factor*) que registra si un árbol AVL tiene el subárbol izquierdo con una altura superior en uno al subárbol derecho, si está perfectamente equilibrado o si el subárbol derecho tiene una altura superior en uno al subárbol izquierdo. El uso del factor de equilibrio en lugar de la altura simplifica la codificación de los algoritmos, porque evita distinguir subárboles vacíos. El invariante del tipo simplemente refuerza el invariante de los árboles de búsqueda con la condición de equilibrio, que tendrá que ser efectivamente mantenida por los algoritmos de inserción y de supresión; para definirla, se introduce una operación auxiliar que calcula la altura de un nodo según se definió en la sección 4.1. Notemos que no se incluyen las comprobaciones sobre los apuntadores que aparecen en los árboles binarios de la fig. 4.9, porque se pueden deducir de la condición de árbol de búsqueda. Para simplificar algoritmos posteriores, el árbol no se enhebra; este caso queda como ejercicio para el lector.

```

tipo tabla es árbol_AVL ftipo
tipo privado árbol_AVL es ^nodo ftipo
tipo privado equilibrio es (IZQ, PERFECTO, DER) ftipo
tipo privado nodo es
  tupla
    k es clave; v es valor
    hizq, hder son ^nodo
    equib es equilibrio {factor de equilibrio del nodo}
  ftupla
ftipo
invariante (T es tabla): correcto(T),
  donde correcto: árbol_AVL  $\rightarrow$  bool se define como:
    correcto(NULO) = cierto
     $p \neq \text{NULO} \Rightarrow \text{correcto}(p) = \text{correcto}(p.\text{hizq}) \wedge \text{correcto}(p.\text{hder}) \wedge$ 
       $p.\text{hizq} \neq \text{NULO} \Rightarrow \text{máximo}(p.\text{hizq}) < p.k \wedge$ 
       $p.\text{hder} \neq \text{NULO} \Rightarrow \text{mínimo}(p.\text{hder}) > p.k \wedge$ 
       $| \text{altura}(p.\text{hizq}) - \text{altura}(p.\text{hder}) | \leq 1 \wedge$ 
       $p.\text{equib} = \text{IZQ} \Leftrightarrow \text{altura}(p.\text{hizq}) > \text{altura}(p.\text{hder}) \wedge$ 
       $p.\text{equib} = \text{PERFECTO} \Leftrightarrow \text{altura}(p.\text{hizq}) = \text{altura}(p.\text{hder}) \wedge$ 
       $p.\text{equib} = \text{DER} \Leftrightarrow \text{altura}(p.\text{hizq}) < \text{altura}(p.\text{hder})$ 
  y donde la función altura: árbol_AVL  $\rightarrow$  nat se define como:
    altura(NULO) = 0
     $p \neq \text{NULO} \Rightarrow \text{altura}(p) = \text{máx}(\text{altura}(p.\text{hizq}), \text{altura}(p.\text{hder})) + 1,$ 
  y máximo y mínimo se definen como en la fig. 5.33

```

Fig. 5.34: representación del tipo de las tablas ordenadas con árboles AVL.

a) Inserción en un árbol AVL

La inserción en un árbol AVL consta de dos etapas diferenciadas: por un lado, es necesario aplicar la casuística de la inserción "normal" para conservar la propiedad de los árboles de búsqueda y, por el otro, es necesario asegurar que el árbol queda equilibrado, reestructurándolo si es necesario. La primera tarea ya ha sido descrita; nos centramos, pues, en la segunda.

Sean a un árbol AVL y $\langle k, v \rangle$ el par clave-valor a insertar; si la clave k ya estaba dentro de a , o bien el nodo que se añade según el algoritmo de inserción no provoca ningún desequilibrio, el proceso acaba sin más problemas. El desequilibrio se produce cuando existe un subárbol a' de a , que se encuentra en cualquiera de los dos casos siguientes:

- El subárbol derecho de a' tiene una altura superior en uno⁸ al subárbol izquierdo de a' , y el nodo correspondiente al par $\langle k, v \rangle$ se inserta en el subárbol derecho de a' y, además, provoca un incremento en uno⁹ de su altura (v. fig. 5.35, izquierda).
- El subárbol izquierdo de a' tiene una altura superior en una unidad al subárbol derecho de a' , y el nodo correspondiente al par $\langle k, v \rangle$ se inserta en el subárbol izquierdo de a' y, además, provoca un incremento en uno de su altura (v. fig. 5.35, derecha).

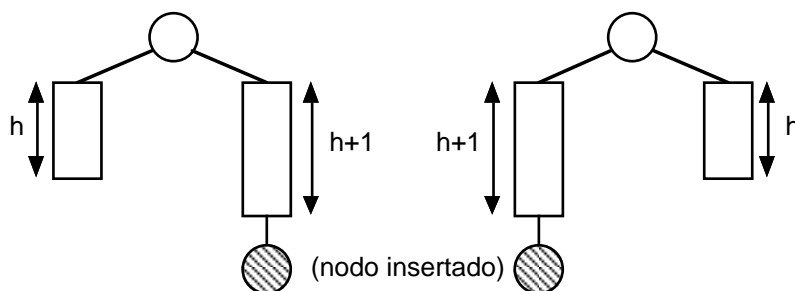


Fig. 5.35: los dos casos posibles de desequilibrio de la inserción en un árbol AVL.

A continuación se estudian las medidas que es necesario tomar para reequilibrar el árbol en estas situaciones. Notemos que ambas son simétricas, razón por la que nos centraremos sólo en la primera, y queda como ejercicio la extensión al segundo caso. Hay dos subcasos:

- Caso *DD* (abreviatura de *Derecha-Derecha*; del inglés *RR*, abreviatura de *Right-Right*): el nodo se inserta en el subárbol derecho del subárbol derecho de a' . En la fig. 5.36 se muestra este caso (las alturas de β y γ son las únicas posibles, una vez fijada la altura de α) y su solución, que es muy simple: la raíz B del subárbol derecho de a' pasa a ser la nueva raíz del subárbol y conserva su hijo derecho, que es el que ha provocado el

⁸ Nunca puede ser superior en más de una unidad, porque a es un árbol AVL antes de la inserción.

⁹ El incremento no puede ser superior a una unidad, porque se inserta un único nodo.

desequilibrio con su incremento de altura y que, con este movimiento, queda a un nivel más cerca de la raíz del árbol, compensando el aumento; la antigua raíz A de a' pasa a ser hijo izquierdo de B y conserva su subárbol izquierdo; finalmente, el anterior subárbol izquierdo de B pasa a ser subárbol derecho de A para conservar la propiedad de ordenación de los árboles de búsqueda. Algunos autores denominan *rotaciones* a estos movimientos de subárboles. Se puede comprobar que la rotación mantiene la ordenación correcta recorriendo inorden el árbol antes y después del proceso. En la fig. 5.36 confirmamos que, efectivamente, el recorrido es idéntico, $\alpha A \beta B \gamma$, tanto en el árbol de la izquierda como en el de la derecha.

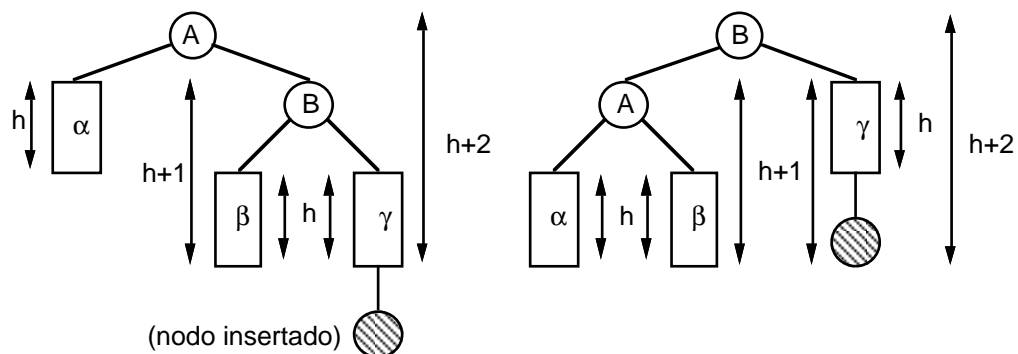


Fig. 5.36: inserción con desequilibrio DD (a la izquierda) y su resolución (a la derecha).

Notemos que la altura del árbol resultante es la misma que tenía el árbol antes de la inserción. Esta propiedad es importantísima, porque asegura que basta con un único reequilibrio del árbol para obtener un árbol AVL después de la inserción, siempre que la búsqueda del primer subárbol que se desequilibra se haga siguiendo el camino que va de la nueva hoja a la raíz (v. fig. 5.37). En el momento en que se reequilibra este subárbol, el resto del árbol queda automáticamente equilibrado, porque ya lo estaba antes de la inserción y su altura no varía. El árbol resultante queda incluso "más equilibrado" que antes, en el sentido de que el proceso de reequilibrio iguala las alturas de los dos subárboles de a' .

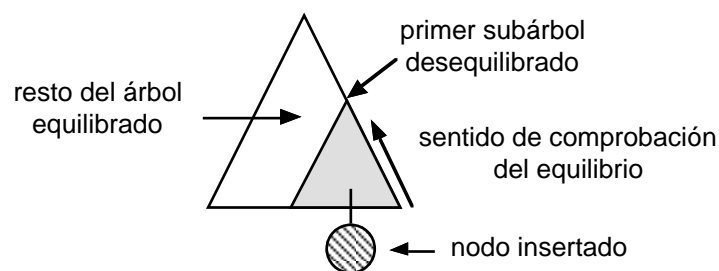


Fig. 5.37: proceso de equilibrado del árbol.

- Caso *DI* (abreviatura de *Derecha-Izquierda*; del inglés *RL*, abreviatura de *Right-Left*): el nodo se inserta en el subárbol izquierdo del subárbol derecho de a' . En la fig. 5.38 se muestra este caso (las alturas de los subárboles de la derecha de la raíz son fijadas a partir de la altura de α) y su solución, que no es tan evidente como antes, porque aplicando las mismas rotaciones de subárboles no se solucionaría el desequilibrio, por lo que es necesario descomponer también el 21-subárbol de a' . Precisamente por este motivo es necesario distinguir el caso trivial en que el 21-subárbol de a' sea vacío y no se pueda descomponer (v. fig. 5.37). La rotación *DI* se puede considerar como la composición de dos rotaciones: la primera, una rotación *II* (simétrica de la *DD*) del subárbol derecho de a' y la segunda, una rotación *DD* del subárbol resultante. Notemos que el nuevo nodo puede ir a parar indistintamente a cualquiera de los dos subárboles que cuelgan del 21-subárbol de a' sin que afecte a las rotaciones definidas. También aquí la altura después de la rotación es igual a la altura previa a la inserción.

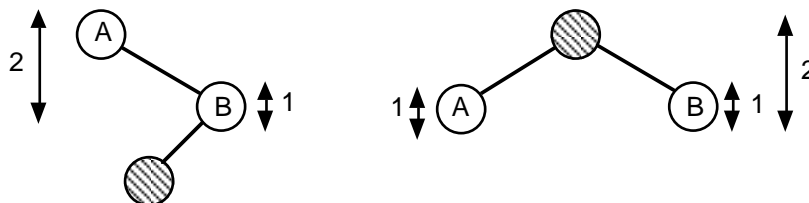


Fig. 5.38: caso trivial de desequilibrio *DI* (a la izquierda) y su resolución (a la derecha).

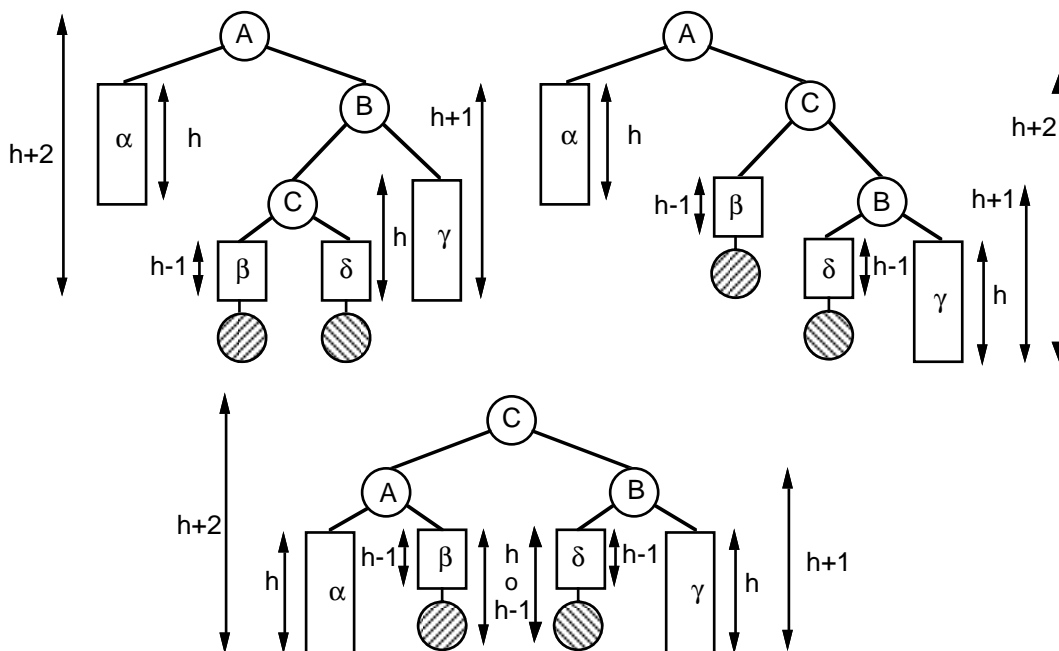


Fig. 5.39: inserción con desequilibrio *DI* (arriba, izquierda), rotación *II* sobre el subárbol izquierdo (arriba, derecha) y rotación *DD* sobre el árbol entero (abajo).

Existen diversos estudios empíricos que intentan establecer la relación entre el número de rotaciones exigidas para una secuencia dada de inserciones [Wir86, p. 227]. Los resultados apuntan que, dada una secuencia aleatoria de inserciones en un árbol AVL, se necesita una rotación para cada dos inserciones, siendo los dos tipos de rotaciones casi equiprobables.

En la fig. 5.41 se muestra la codificación recursiva del algoritmo; la versión iterativa queda como ejercicio para el lector. La tarea más importante recae sobre una función recursiva auxiliar, *inserta_AVL*, y los casos terminales de la recursividad son crear una nueva hoja o bien encontrar algún nodo en el árbol que contenga la clave. En el primer caso, y después de hacer la inserción del nodo siguiendo la misma estrategia que en la fig. 5.33, se estudia el equilibrio desde la hoja, avanzando a la raíz hasta que se llega a ella, o se encuentra algún subárbol que no crece, o bien se encuentra algún subárbol que se desequilibra; este estudio es sencillo usando el factor de equilibrio del nodo. Para equilibrar el árbol, se usan dos operaciones auxiliares más, una de las cuales también se codifica en la figura, que no hacen más que implementar las rotaciones que se acaban de describir como modificaciones del valor de los encadenamientos. Destaquemos que el árbol es un parámetro de entrada y de salida para asegurar que los encadenamientos queden realmente actualizados.

b) Supresión en un árbol AVL

Los dos casos posibles de desequilibrio en la supresión son idénticos al proceso de inserción, pero ahora el desequilibrio se produce porque la altura de un subárbol disminuye por debajo del máximo tolerado. Una vez más, nos centramos en el desequilibrio provocado por la supresión en el subárbol izquierdo; la otra situación es simétrica. Los diferentes algoritmos de rotación que se necesitan dependen exclusivamente de la relación de las alturas de los dos subárboles del subárbol derecho de la raíz (que, a diferencia de lo que ocurría al insertar, nunca pueden ser vacíos):

- Si son iguales, se produce el desequilibrio DD del caso de la inserción, que se resuelve de la misma forma (v. fig. 5.40). El árbol resultante tiene la misma altura antes y después de la supresión, por lo que basta con esta rotación para reestablecer el equilibrio.

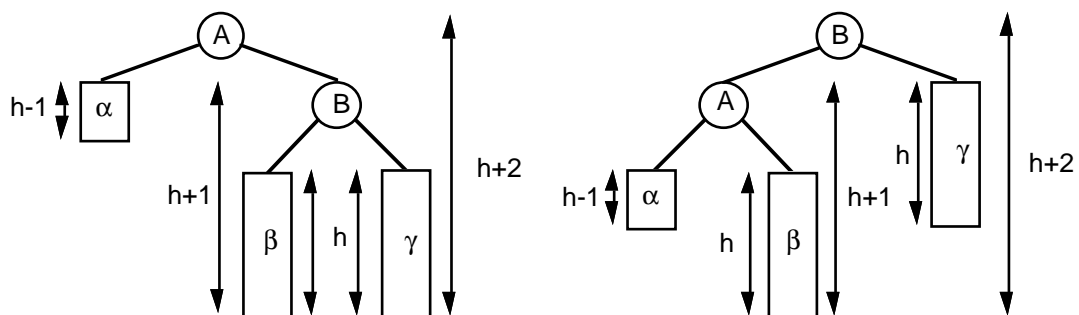


Fig. 5.40: supresión con desequilibrio DD (a la izquierda) y su resolución (a la derecha).

función inserta (a es tabla; k es clave; v es valor) devuelve tabla es
var res es bool fvar
 inserta_AVL(a, k, v, res)
devuelve a

{Función auxiliar *inserta_AVL(a, k, v, crece?)*: dado un árbol AVL *a*, inserta el par $\langle k, v \rangle$, de manera que se conserva el invariante del tipo; *crece?* indica si la altura del árbol aumenta}

acción privada inserta_AVL (ent/sal a es árbol_AVL; ent k es clave; ent v es valor;
 sal crece? es bool) es
 si a = NULO entonces {se crea un árbol de un único nodo}
 a := obtener_espacio
 si a = NULO entonces error
 si no
 a := <k, v, NULO, NULO, PERFECTO>
 crece? := cierto
 fsi
 si no
 opción
 caso a^.k = k hacer a^.v := v; crece? := falso
 caso a^.k > k hacer
 inserta_AVL(a^.hizq, k, v, crece?) {inserción recursiva por la izquierda}
 si crece? entonces {estudio del equilibrio y rotaciones si es necesario}
 opción
 caso a^.equib = DER hacer a^.equib := PERFECTO; crece? := falso
 caso a^.equib = PERFECTO hacer a^.equib := IZQ; crece? := cierto
 caso a^.equib = IZQ hacer a := rotación_izq(a); crece? := falso
 fopción
 fsi
 caso a^.k < k hacer {simétrico al anterior}
 inserta_AVL(a^.hder, k, v, crece?)
 si crece? entonces
 opción
 caso a^.equib = IZQ hacer a^.equib := PERFECTO; crece? := falso
 caso a^.equib = PERFECTO hacer a^.equib := DER; crece? := cierto
 caso a^.equib = DER hacer a := rotación_der(a); crece? := falso
 fopción
 fsi
 fopción
 fsi
 facción

Fig. 5.41: algoritmo de inserción en un árbol AVL.

{Función auxiliar *rotación_der*: dado un subárbol *a* donde todos sus subárboles son AVL, pero *a* está desequilibrado por la derecha, investiga la razón del desequilibrio y efectúa las rotaciones oportunas}

función privada *rotación_der* (*a es árbol_AVL*) devuelve *árbol_AVL es*

var *raíz*, *b*, *beta*, *delta son* *^nodo ftupla*

{*raíz* apuntará a la nueva raíz del árbol}

si (*a^.hizq = NULO*) \wedge (*a^.hder^.hder = NULO*) entonces {caso trivial de la fig. 5.38}

b := a^.hder; raíz := b^.hizq

raíz^.hizq := a; raíz^.hder := b; b^.hizq := NULO; a^.hder := NULO

a^.equib := PERFECTO; b^.equib := PERFECTO

si no {es necesario distinguir tipo de desequilibrio y rotar en consecuencia}

si *a^.hder^.equib = DER* entonces {desequilibrio DD, v. fig. 5.36}

raíz := a^.hder; beta := raíz^.hizq

raíz^.hizq := a; a^.hder := beta; a^.equib := PERFECTO

si no {desequilibrio DI, v. fig. 5.39}

b := a^.hder; raíz := b^.hizq; beta := raíz^.hizq; delta := raíz^.hder

a^.hder := beta; raíz^.hizq := a; b^.hizq := delta; raíz^.hder := b

si *raíz^.equib = IZQ* entonces {el nuevo elemento está dentro de *beta*}

a^.equib := PERFECTO; b^.equib := DER

si no {el nuevo elemento está dentro de *delta*}

a^.equib := IZQ; b^.equib := PERFECTO

fsi

fsi

fsi

raíz^.equib := PERFECTO

devuelve *raíz*

Fig. 5.41: algoritmo de inserción en un árbol AVL (cont.).

- Si la altura del subárbol izquierdo es menor que la altura del subárbol derecho, la rotación es exactamente la misma; ahora bien, la altura del árbol resultante es una unidad más pequeña que antes de la supresión (v. fig. 5.42). Este hecho es significativo, porque obliga a examinar si algún subárbol que lo engloba también se desequilibra.
- Si la altura del subárbol izquierdo es mayor que la altura del subárbol derecho, la rotación es similar al caso DI (v. fig. 5.43); también aquí la altura del árbol resultante es una unidad más pequeña que antes de la supresión. Los árboles β y γ de la figura pueden tener altura $h-1$ ó $h-2$, pero al menos uno de ellos tiene altura $h-1$.

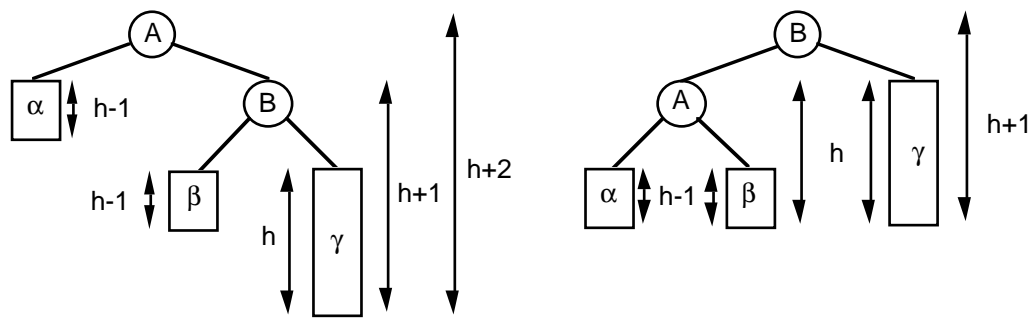


Fig. 5.42: supresión con desequilibrio DD (izq.) y su resolución (der.) con altura variable.

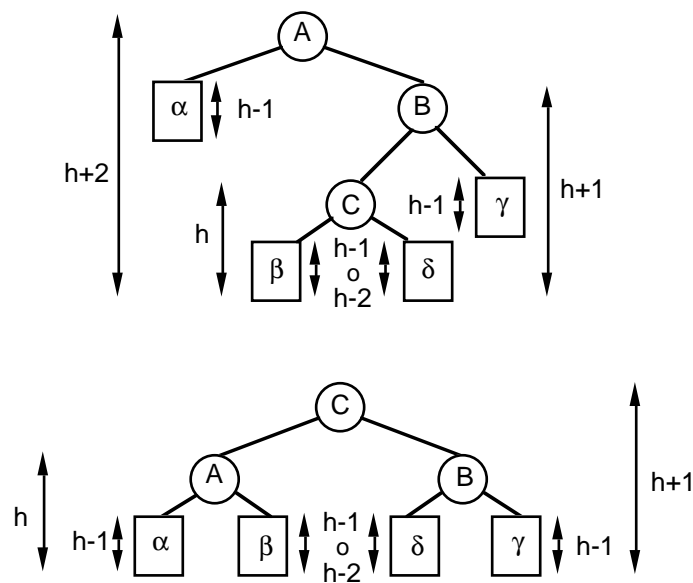


Fig. 5.43: supresión con desequilibrio DI (arriba) y su resolución (abajo).

Igual que sucedía en la inserción, se han hecho estudios empíricos sobre el número de rotaciones exigidas durante una secuencia de supresiones y, curiosamente, los resultados indican que sólo es necesaria una rotación por cada cinco supresiones [Wir86, p. 227].

El algoritmo de supresión se implementa recursivamente en la fig. 5.44. Su similitud con la operación de inserción es evidente, por lo que no se comenta más que aquello estrictamente imprescindible. Queda claro que la supresión de un elemento puede exigir tantas rotaciones como nodos haya en el camino proveniente de la raíz, porque después de cada vuelta de una llamada recursiva puede haber reorganizaciones. Se usa una función

auxiliar para borrar el elemento que aplica la casuística de la supresión en un árbol de búsqueda y comprueba el equilibrio en el caso general; esta función se podría escribir más compacta (v. [Wir86, pp.226-227]), pero se ha expandido por motivos de legibilidad. Una vez más, hay una ineficiencia temporal, porque en el caso de mover elementos se recorre un mismo camino dos veces; de todos modos, el coste asintótico es igualmente logarítmico; también, la codificación cambia la dirección física de los datos en las mismas condiciones que la supresión en árboles binarios de búsqueda.

función borra (a es tabla; k es clave) devuelve tabla es
var res es bool fvar
 borra_AVL(a, k, res)
devuelve a

{Función *borra_AVL(a, k, v, encoge?)*: dado el árbol AVL *a*, borra el nodo que tiene como clave *k*, conservando el invariante del tipo; *encoge?* indica si la altura disminuye}

acción priv borra_AVL (ent/sal a es árbol_AVL; ent k es clave; sal encoge? es bool) es
si a = NULO entonces encoge? := falso {no hay ningún nodo de clave *k*}
si no opción
 caso a.k = k hacer {se borra el nodo, controlando posibles desequilibrios}
 <a, encoge?> := borra_nodo(a)
 caso a.k > k hacer
 borra_AVL(a.hizq, k, encoge?) {se sigue la rama adecuada}
 {estudio del equilibrio y rotaciones si es necesario}
 si encoge? entonces <a, encoge?> := equilibra_derecha(a) fsi
 caso a.k < k hacer {simétrico al anterior}
 borra_AVL(a.hder, k, encoge?)
 si encoge? entonces <a, encoge?> := equilibra_izquierda(a) fsi
fopción
fsi
facción

Fig. 5.44: algoritmo de supresión en un árbol AVL.

{Función auxiliar *equilibra_derecha(a)*: dado un árbol AVL *a* donde se ha borrado un nodo del subárbol izquierdo provocando una disminución de altura, averigua si el árbol se ha desequilibrado y, en caso afirmativo, efectúa las rotaciones oportunas. Además, devuelve un booleano que indica si la altura del árbol ha disminuido}

función privada equilibra_derecha (a es árbol_AVL) devuelve <árbol_AVL, bool> es var encoge? es bool fvar

opción

caso a^.equib = IZQ hacer a^.equib := PERFECTO; encoge? := cierto

caso a^.equib = PERFECTO hacer a^.equib := DER; encoge? := falso

caso a^.equib = DER hacer <a, encoge?> := rotación_derecha(a)

fopción

devuelve <a, encoge?>

{Función auxiliar *rotación_derecha*: dado un subárbol *a* en que todos los subárboles son AVL, pero *a* está desequilibrado por la derecha, investiga la razón del desequilibrio y efectúa las rotaciones oportunas. Además, devuelve un booleano que indica si la altura del árbol ha disminuido}

función privada rotación_derecha (a es árbol_AVL) devuelve <árbol_AVL, bool> es var raíz, b, beta, delta son ^nodo; encoge? es bool fvar

{*raíz* apuntará a la nueva raíz del árbol}

si a^.hder^.equib = IZQ entonces {desequilibrio DI, v. fig. 5.43}

b := a^.hder; raíz := b^.hizq; beta := raíz^.hizq; delta := raíz^.hder

a^.hder := beta; raíz^.hizq := a; b^.hizq := delta; raíz^.hder := b

opción {se actualizan los factores de equilibrio según las alturas de *beta* y *delta*}

caso raíz^.equib = IZQ hacer a^.equib := PERFECTO; b^.equib := DER

caso raíz^.equib = PERFECTO hacer

a^.equib := PERFECTO; b^.equib := PERFECTO

caso raíz^.equib = DER hacer a^.equib := IZQ; b^.equib := PERFECTO

fopción

raíz^.equib := PERFECTO; encoge? := cierto

si no {desequilibrio DD, v. fig. 5.40 y 5.42, de idéntico tratamiento}

raíz := a^.hder; beta := raíz^.hizq; raíz^.hizq := a; a^.hder := beta

{a continuación, se actualizan los factores de equilibrio según las alturas de *beta* y el hijo derecho de la nueva raíz, y se indaga si el árbol ha encogido}

si raíz^.equib = PERFECTO

entonces a^.equib := DER; raíz^.equib := IZQ; encoge? := falso

si no a^.equib := PERFECTO; raíz^.equib := PERFECTO; encoge? := cierto

fsi

fsi

devuelve raíz

Fig. 5.44: algoritmo de supresión en un árbol AVL (cont.).

{Función auxiliar *borra_nodo(a)*: dado un árbol *a* en que la raíz contiene el elemento que se quiere borrar, lo suprime según la casuística de los árboles binarios de búsqueda y libera espacio; en caso de que existan los dos subárboles de *a*, controla el equilibrio y rota si es necesario. Además, devuelve un booleano que indica si la altura del árbol ha disminuido}

función privada *borra_nodo* (*a es árbol_AVL*) devuelve <árbol_AVL, bool> es
var raíz, min son ^nodo; encoge? es bool fvar
 {raíz apunta a la raíz del subárbol resultante}
opción
 caso (*a^.hizq = NULO*) \wedge (*a^.hder = NULO*) hacer
 {es una hoja; la altura disminuye}
 raíz := NULO; encoge? := cierto
 liberar_espacio(*a*)
 caso (*a^.hizq \neq NULO*) \wedge (*a^.hder = NULO*) hacer
 {le cuelga un único nodo por la izquierda, que sube; la altura disminuye}
 raíz := *a^.hizq*; encoge? := cierto
 liberar_espacio(*a*)
 caso (*a^.hizq = NULO*) \wedge (*a^.hder \neq NULO*) hacer
 {le cuelga un único nodo por la derecha, que sube; la altura disminuye}
 raíz := *a^.hder*; encoge? := cierto
 liberar_espacio(*a*)
 caso (*a^.hizq \neq NULO*) \wedge (*a^.hder \neq NULO*) hacer
 {obtiene y copia el mínimo del subárbol derecho a la raíz}
 min := *a^.hder*
 mientras min^.hizq \neq NULO hacer min := min^.hizq fmientras
 a^.k := min^.k; *a*^.v := min^.v
 {a continuación, borra el mínimo y controla el equilibrio}
 borra_AVL(*a^.hder*, min^.k, encoge?)
 si encoge? entonces <*a*, encoge?> := equilibra_izquierda(*a*) fsi
 raíz := *a*
fopción
devuelve <raíz, encoge?>

Fig. 5.44: algoritmo de supresión en un árbol AVL (cont.).

Ejercicios

5.1 Escribir una especificación para el modelo de las funciones parciales.

5.2 Implementar las tablas con todas las variantes de listas que aparecen en la sección 5.2.

5.3 Sean las funciones de dispersión $h(k) = k \bmod 512$ y $h'(k) = k \bmod 557$ aplicadas sobre claves naturales. Enumerar las ventajas y desventajas que presentan.

5.4 Programar en C, al máximo detalle y con la mayor eficiencia posible, las funciones de dispersión presentadas en la sección 5.3, suponiendo una tabla de dimensión de aproximadamente 5.000 elementos (determinar con exactitud la que sea apropiada en cada caso) y siendo las claves cadenas de caracteres compuestas sólo de letras mayúsculas y minúsculas (distinguibles entre ellas).

5.5 a) Se quiere organizar una tabla de dispersión de siete posiciones con estrategia de direccionamiento abierto y redispersión lineal. Sea la función h de dispersión que da los siguientes valores para las siguientes claves: $h(\text{sonia}) = 3$, $h(\text{gemma}) = 5$, $h(\text{paula}) = 2$, $h(\text{anna}) = 3$, $h(\text{ruth}) = 3$, $h(\text{cris}) = 2$.

i) Insertar todas estas claves en el siguiente orden: *paula, anna, cris, ruth, sonia, gemma*. Mostrar claramente cómo va evolucionando la tabla y indicar las colisiones que se producen. A continuación, mostrar el camino generado para buscar la clave *ruth*.

ii) Repetir el proceso cuando el orden de inserción sea: *sonia, anna, ruth, gemma, paula, cris*. Comprobar que las claves ocupan, en general, posiciones diferentes a las de antes dentro de la tabla. A continuación, mostrar el camino generado para buscar la clave *ruth*.

iii) ¿En cuál de las dos configuraciones obtenidas hay menos comparaciones entre claves en el caso peor de búsqueda de un elemento en la tabla?

iv) ¿Hay algún orden de inserción que distribuya mejor las claves según el criterio del apartado anterior? ¿Y peor? ¿Por qué?

b) Repetir el apartado a) con una tabla de dispersión de siete posiciones con estrategia coalescente, suponiendo que los sitios libres se obtienen comenzando a buscarlos desde la última posición de la tabla en dirección a la primera.

5.6 Queremos almacenar en una tabla de dispersión 10.000 elementos de X bits de tamaño y tales que no existe ningún valor especial. Suponiendo una función de dispersión que distribuya los elementos de manera uniforme, y sabiendo que un booleano ocupa un bit y un entero o puntero 32 bits, determinar el espacio (en bits y en función de X) que ocuparía cada una de las representaciones habituales de dispersión, si se queremos que el número esperado de comparaciones entre claves en una búsqueda con éxito sea como mucho de 2.5.

5.7 Implementar la estrategia de direccionamiento abierto con redistribución lineal y con movimiento de elementos en la supresión.

5.8 En el año 1974, O. Amble y D.E. Knuth propusieron en el artículo "Ordered Hash Tables", publicado en *The Computer Journal*, 17, un método de dispersión similar a la estrategia de direccionamiento abierto con redistribución lineal, con la particularidad de que los elementos se podrían obtener ordenados sin necesidad de encadenar los elementos o reorganizar la tabla. La única (y muy restrictiva) condición era que la función de dispersión fuera monótonica, de manera que la gestión lineal de los sinónimos garantizara que los elementos quedaban realmente ordenados dentro del vector.

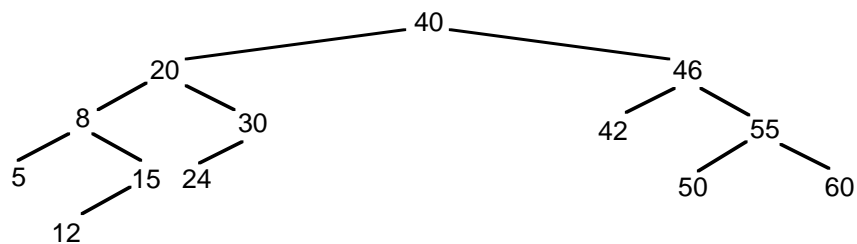
a) Pensar alguna situación donde sea posible definir una función de dispersión monótonica y definirla. (D. Gries muestra un ejemplo en *Communications ACM*, 30(4), 1987.)

b) Implementar la estrategia.

5.9 Dibujar todos los árboles de búsqueda posibles que contengan los naturales 1, 2, 3 y 4 (sin repeticiones). ¿Cuáles de estos árboles son equilibrados? En general, ¿cuántos árboles de búsqueda se pueden formar con n elementos diferentes?

5.10 Dado un árbol de búsqueda inicialmente vacío, insertar los elementos 7, 2, 9, 0, 5, 6, 8 y 1, equilibrando a cada paso.

5.11 Dado el árbol de búsqueda de la figura:



insertar sucesivamente los valores 64, 41, 10, 3, 9, 1 y 2. A continuación, borrar sucesivamente los valores 9, 15, 10 y 55; en caso de borrar un nodo con dos hijos, sustituirlo por el menor de su subárbol derecho. Repetir el proceso equilibrando el árbol después de cada modificación (notar que, inicialmente, ya está equilibrado). Mostrar claramente la evolución del árbol paso a paso en cada operación.

5.12 Proponer una representación de los árboles de búsqueda equilibrados que, además de las operaciones habituales, permita encontrar el k -ésimo elemento menor del árbol en tiempo logarítmico, sin afectar el coste de las operaciones habituales y empleando el mínimo espacio adicional que sea posible. Codificar la operación.

5.13 Proponer una estructura de datos que tenga tiempo logarítmico en la inserción y supresión de elementos y tiempo constante en la consulta de elementos, y que permita listar todos los elementos ordenados en un tiempo lineal.

5.14 Implementar el TAD de las colas prioritarias usando árboles de búsqueda.

5.15 Interesa encontrar una representación de los conjuntos que, aparte de las típicas operaciones de añadir y sacar elementos y de comprobar si un elemento está dentro de un conjunto, ofrezca otras tres que obtengan una lista con la intersección, la unión y la diferencia de dos conjuntos. ¿Cuál es la mejor estructura para favorecer las tres últimas operaciones? ¿Y si, además, queremos que las tres primeras también sean rápidas? ¿Qué ocurre si las tres últimas, en vez de devolver una lista de elementos, han de devolver también un conjunto? Justificar las respuestas en función del coste de las operaciones.

5.16 Se quiere implementar un diccionario con operaciones de acceso directo por palabra y de recorrido alfabético. Suponer que el espacio reservado para cada entrada del diccionario es X , un valor fijo. Suponer que el número esperado de entradas es N . Razonar qué estructura de datos es mejor en los siguientes supuestos: **a)** minimizando el espacio que ocupa el diccionario; **b)** favoreciendo el coste temporal de las operaciones de recorrido alfabético y de consulta directa por palabra; **c)** favoreciendo el coste temporal de las operaciones de recorrido alfabético y de actualización del diccionario (inserciones y supresiones). En cada caso determinar exactamente el coste temporal de las diferentes operaciones del diccionario, así como también el espacio usado (en bits y como función de X y N), suponiendo que los enteros y los punteros ocupan 32 bits.

5.17 La Federación Local de Ajedrez de Catalunya (*FLACA*) ha decidido informatizarse. La *FLACA* tiene registrados los nombres de los diferentes clubes y jugadores de ajedrez de Catalunya, y sabe que un jugador sólo pertenece a un club y que, a causa de la incipiente crisis económica, los clubes se fusionan con cierta frecuencia para formar otros nuevos. El nombre del club resultante es igual al nombre del club que tenga más ajedrecistas y los jugadores de los clubes que se fusionan pasan a ser automáticamente del nuevo club. Determinar la signatura y la implementación de la estructura necesaria para que se puedan fusionar clubes, y a qué club pertenece un jugador concreto con cierta rapidez. ¿Cuál es el coste resultante de las operaciones? ¿Y el espacio empleado?

5.18 A causa de una intensa campaña institucional y de las exenciones fiscales, el número de clubes de ajedrez en Catalunya se ha disparado, es muy grande y continúa creciendo. Para organizar futuros campeonatos, la *FLACA* clasifica los clubes por comarcas (considerar que hay un número pequeño y fijo de comarcas) de manera que, al dar de alta un nuevo club, se dice de qué comarca es. Diseñar una estructura de datos para que sea lo más eficiente posible al sacar los listados, ordenados alfabéticamente, de todos los clubes de Catalunya y de todos los clubes de una comarca dada, y que la operación de añadir un nuevo club no sea

excesivamente lenta. En todos los casos, calcular el coste de las operaciones y justificar que la solución expuesta no es mejorable. ¿Cómo se implementaría la operación de listar ordenadamente todos los clubes de una comarca?

Capítulo 6 Relaciones binarias y grafos

En este capítulo nos centraremos en el modelo de las *relaciones binarias*¹ entre elementos de dos dominios A y B , donde un elemento de A está relacionado con un elemento de B si se cumple el *enunciado* de la relación. Generalmente, un elemento de A estará relacionado con varios de B y viceversa; por ello, en algunos textos informáticos las relaciones binarias se denominan *relaciones $m:n$* como abreviatura de: m elementos de A se relacionan con n de B y viceversa.

Un ejemplo de aplicación de las relaciones binarias es la gestión de la matriculación de alumnos en una universidad (v. fig. 6.1). La estructura necesaria se puede considerar como una relación entre dos conjuntos de elementos: los alumnos y las asignaturas, por la que cada alumno está relacionado con todas las asignaturas que cursa y cada asignatura con todos los alumnos que se han matriculado de la misma. Eventualmente, podríamos decidir almacenar la cualificación que el alumno ha obtenido de las asignaturas, y entonces obtenemos *relaciones binarias etiquetadas*.

	IP	IC	Alg	Fis	PM	ILo	EDA
Abad	x	x	x	x			
Abadía		x		x	x	x	
Aguilar					x	x	x
...							

Fig. 6.1: representación de la relación alumnos-asignaturas
(la cruz significa que el alumno cursa la asignatura).

Dado el contexto de uso esperado, las operaciones sobre el tipo que parecen adecuadas son: matricular un alumno de una asignatura, desmatricularlo (cuando la ha aprobado), averiguar si un alumno cursa una asignatura concreta, y listar las asignaturas que cursa un alumno y los alumnos que cursan una asignatura. Extrapolando estas operaciones a un

¹ Si bien nos podríamos plantear el estudio de relaciones sobre un número arbitrario de dominios, el caso habitual es el binario y por eso nos limitamos a éste.

marco general, podemos formular una especificación para las relaciones y deducir implementaciones eficientes. Concretamente, en la primera sección estudiaremos la especificación y la implementación de las relaciones binarias en general, mientras que en el resto del capítulo nos centraremos en el caso particular de relaciones binarias sobre un único dominio. Distinguiremos dos modelos. Primero, el TAD de las relaciones de equivalencia, en el que las relaciones cumplen las propiedades reflexiva, simétrica y transitiva; las operaciones que normalmente se precisan sobre estas relaciones son de diferente naturaleza que en el resto de relaciones. A continuación, estudiaremos una particularización de la idea de relación binaria denominada *grafo*. Sobre los grafos se definen varios algoritmos de gran interés, cuya resolución será ampliamente comentada y en cuya implementación intervendrán diversos TAD ya conocidos, como las colas prioritarias y las propias relaciones de equivalencia.

6.1 Relaciones binarias

6.1.1 Especificación

Se quiere especificar el TAD de las relaciones binarias $\mathcal{R}_{A \times B}$ definidas sobre dos dominios de datos A y B , tal que todo valor R del TAD, $R \in \mathcal{R}_{A \times B}$, es un conjunto de pares ordenados $\langle a, b \rangle$, $a \in A$ y $b \in B$. Dados $R \in \mathcal{R}_{A \times B}$, $a \in A$ y $b \in B$, la signatura del tipo es:

- Crear la relación vacía: *crea*, devuelve \emptyset .
- Añadir un par a la relación: *inserta*(R, a, b), devuelve $R \cup \{\langle a, b \rangle\}$.
- Borrar un par de la relación: *borra*(R, a, b), devuelve $R - \{\langle a, b \rangle\}$.
- Mirar si un par de elementos están relacionados: *existe?*(R, a, b), averigua si $\langle a, b \rangle \in R$.
- Dos operaciones para determinar el conjunto recorrible de elementos que están relacionados con uno dado: *fila*(R, a), devuelve el conjunto $s \in \mathcal{P}(B)$, definido como $b \in s \Leftrightarrow \langle a, b \rangle \in R$, y *columna*(R, b), que se define simétricamente.

Las dos últimas operaciones mencionadas son necesarias para que el TAD se adapte a la mayoría de contextos de uso. Podríamos optar por añadir iteradores para implementar este concepto, lo que resultaría en implementaciones más eficientes; no obstante, por simplicidad preferimos la opción aquí presentada, sobre todo porque el coste asintótico de los recorridos de fila y columna mediante iteradores en una implementación dada será equivalente al coste de las operaciones *fila* y *columna*, respectivamente.

Alternativamente, se podría considerar la relación como una función de pares de elementos en el álgebra booleana \mathcal{B} de valores cierto y falso, $f: A \times B \rightarrow \mathcal{B}$, en la que $f(a, b)$ vale cierto si a y b están relacionados.

En la figura 6.2 se presenta una especificación para las relaciones. Notemos que aparecen diferentes parámetros formales. Por un lado, los géneros sobre los que se define la relación así como sus operaciones de igualdad, imprescindibles en la especificación. Por otro lado, y para establecer un criterio de ordenación de los elementos al insertarlos en el conjunto recorrible, se requiere que los géneros presenten una operación de comparación adicional, tal como establece el universo $ELEM_<=>$. Se ha optado por definir los conjuntos recorribles como ordenados para que no surjan problemas de consistencia al implementar el tipo (concretamente, se evita que las implementaciones secuenciales inserten los elementos en los conjuntos en un orden diferente que en la especificación). Veremos que el impacto en la eficiencia de la implementación de esta decisión será en general mínimo.

Notemos que, al haber dos operaciones que devuelven dos conjuntos de elementos de tipos diferentes, es necesario efectuar dos instancias de los conjuntos genéricos para determinar su signatura. Los tipos resultantes son implícitamente exportados y podrán ser empleados por los módulos usuarios de las relaciones para declarar variables o parámetros.

Por lo que respecta a las ecuaciones, las relaciones establecidas sobre *inserta* confirman que la relación representa un conjunto de pares. Notemos también que, al borrar, hay que sacar todas las posibles apariciones del par y que si el par no existe la relación queda igual. Por último, cabe destacar que la inserción reiterada de elementos en obtener una fila o columna no afecta el resultado por las ecuaciones propias de los conjuntos recorribles.

El uso de las relaciones exige a veces, no sólo relacionar pares de elementos, sino también explicitar un valor que caracteriza este nexo, en cuyo caso diremos que la relación es *etiquetada*; en el ejemplo de la fig. 6.1, esto sucede si se quiere guardar la nota que un alumno obtiene de cada asignatura que cursa. El modelo resultante exige algunos cambios en la especificación y, posteriormente, en la implementación. Por lo que respecta al modelo, la relación se puede considerar como una función de dos variables (los dominios de la relación) sobre el codominio V de las etiquetas, $f : A \times B \rightarrow V$, de manera que $f(a, b)$ represente la etiqueta de la relación. Supondremos que V presenta un valor especial \perp tal que, si a y b no están relacionados, $consulta(f, a, b) = \perp$ y, así, la función f es total. De lo contrario, debería añadirse una operación *definida?*: *relación* $A.elem\ B.elem \rightarrow bool$ para ver si un par está en el dominio de la función y considerar un error que se consulte la etiqueta de un par indefinido.

En la figura 6.3 se presenta la signatura del TAD (la especificación queda como ejercicio para el lector); notemos que el género de las etiquetas y el valor \perp se definen en el universo $ELEM_ESP$. Las operaciones *fila* y *columna* no sólo devuelven los elementos con que se relacionan sino también la etiqueta, por lo que los conjuntos resultantes son de pares de elementos que se forman como instancia del universo genérico PAR , en las que el concepto de igualdad tan sólo tiene en cuenta el componente elemento y no la etiqueta.

universo RELACIÓN (A, B son ELEM_<_ =) es

usa BOOL

instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_ =) donde

C.elem es A.elem, C.< es A.<, C.= es A.=

renombra cjt por cjt_a

instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_ =) donde

C.elem es B.elem, C.< es B.<, C.= es B.=

renombra cjt por cjt_b

tipo relación

ops crea: \rightarrow relación

inserta, borra: relación A.elem B.elem \rightarrow relación

existe?: relación A.elem B.elem \rightarrow bool

fila: relación A.elem \rightarrow cjt_b

columna: relación B.elem \rightarrow cjt_a

ecns $\forall R \in \text{relación}; \forall a, a_1, a_2 \in A.\text{elem}; \forall b, b_1, b_2 \in B.\text{elem}$

inserta(inserta(R, a, b), a, b) = inserta(R, a, b)

inserta(inserta(R, a₁, b₁), a₂, b₂) = inserta(inserta(R, a₂, b₂), a₁, b₁)

borra(crea, a, b) = crea

borra(inserta(R, a, b), a, b) = borra(R, a, b)

$[(a_1 \neq a_2) \vee (b_1 \neq b_2)] \Rightarrow \text{borra}(\text{inserta}(R, a_1, b_1), a_2, b_2) = \text{inserta}(\text{borra}(R, a_2, b_2), a_1, b_1)$

existe?(crea, a, b) = falso

existe?(inserta(R, a₁, b₁), a₂, b₂) = existe?(R, a₂, b₂) $\vee ((a_1 = a_2) \wedge (b_1 = b_2))$

fila(crea, a) = CJT_ORDENADO_RECORRIBLE.crea

fila(inserta(R, a, b), a) = CJT_ORDENADO_RECORRIBLE.añade(fila(R, a), b)

$[a_1 \neq a_2] \Rightarrow \text{fila}(\text{inserta}(R, a_1, b), a_2) = \text{fila}(R, a_2)$

columna(crea, b) = CJT_ORDENADO_RECORRIBLE.crea

columna(inserta(R, a, b), b) = CJT_ORDENADO_RECORRIBLE.añade(columna(R, b), a)

$[b_1 \neq b_2] \Rightarrow \text{columna}(\text{inserta}(R, a, b_1), b_2) = \text{columna}(R, b_2)$

funiverso

Fig. 6.2: especificación del tipo abstracto de datos de las relaciones.

El modelo aquí presentado considera que los dominios A y B de la relación son fijos. En ocasiones, esto no es así y los elementos de estos dominios aparecen y desaparecen dinámicamente. Dependiendo del contexto, podemos optar por tener operaciones explícitas de añadido y supresión de elementos, o bien dejar que dichos elementos se añadan o borren implícitamente, como efecto lateral del añadido y supresión de relaciones entre ellos.

universo RELACIÓN_ETIQUETADA (A, B son ELEM_<_, V es ELEM_ESP) es
instancia PAR (P1, P2 son ELEM) donde
P1.elem es A.elem, P2.elem es V.elem
renombrar par por a_y_etiq, _c1 por _a, _c2 por _et
instancia PAR (P1, P2 son ELEM) donde
P1.elem es B.elem, P2.elem es V.elem
renombrar par por b_y_etiq, _c1 por _b, _c2 por _et
instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_) donde
C.elem es a_y_etiq, C.= es (_a) A.= (_a), C.< es (_a) A.< (_a)
renombrar cjt por cjt_a_y_etiq
instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_) donde
C.elem es b_y_etiq, C.= es (_b) B.= (_b), C.< es (_b) B.< (_b)
renombrar cjt por cjt_b_y_etiq
tipo relación
ops crea: \rightarrow relación
inserta: relación A.elem B.elem V.elem \rightarrow relación
borra: relación A.elem B.elem \rightarrow relación
consulta: relación A.elem B.elem \rightarrow V.elem
fila: relación A.elem \rightarrow cjt_b_y_etiq
columna: relación B.elem \rightarrow cjt_a_y_etiq
funiverso

Fig. 6.3: *signatura del tipo abstracto de datos de las relaciones etiquetadas.*

Por último, citar que son usuales las variantes recorribles y abiertas del TAD. No obstante, debe precisarse qué se entiende por ellas, pues en este TAD son dos dominios y no uno los que están involucrados.

Por lo que respecta a los TAD recorribles, podemos tener iteradores (ordenados o no) tanto sobre el dominio A , como sobre el B , como sobre el conjunto de relaciones entre elementos que hay en una relación binaria. Si los dominios son fijos, probablemente tanto A como B ya tienen sus propios iteradores y no es necesario volverlos a definir. Si A o B tienen iteradores, ora porque ya los tenían, ora porque se han introducido, pueden obtenerse todos los elementos iterando sobre el dominio A o B y aplicando la función *fila* o *columna* reiteradamente; de esta manera, no es necesario el tercer tipo de iterador mencionado.

En cuanto a los TAD abiertos, hay las mismas opciones: se puede permitir accesos directos tanto a los elementos de A , como a los de B , como a las relaciones individuales entre elementos.

6.1.2 Implementación

En el resto de la sección estudiaremos la implementación del TAD de las relaciones binarias no etiquetadas; la extensión al caso etiquetado queda como ejercicio para el lector. A lo largo del apartado, denotaremos por r el número de elementos del dominio A , $r = \|A\|$, por s el número de elementos del dominio B , $s = \|B\|$, y por n el número de pares de elementos relacionados en la relación R , $n = \|R\|$.

a) Representación secuencial

La representación más intuitiva del tipo de las relaciones binarias es la secuencial, que consiste en usar un vector bidimensional de booleanos, con lo que se registra si el par correspondiente a cada posición pertenece o no a la relación. Si los dominios A y B no permiten indexar directamente el vector, debe añadirse una tabla que asocie a los elementos de estos dominios un índice del vector. Esta tabla se implementará mediante una lista, una tabla de dispersión o un árbol de búsqueda dependiendo de los requisitos de eficiencia.

Si hay muchos elementos interrelacionados esta solución puede ser satisfactoria, porque las operaciones de inserción, supresión y consulta son $\Theta(1)$, pero no lo será en el caso general, principalmente por razones de espacio (la representación del tipo queda $\Theta(r * s)$) y también por la ineficiencia de *fila* y *columna*, que quedan de orden lineal independientemente del número de elementos que en hay en ellas, $\Theta(s)$ y $\Theta(r)$ respectivamente. Por ejemplo, si consideramos una universidad que imparta 400 asignaturas con una media de 6 asignaturas cursadas por alumno, la matriz de la fig. 6.1 estará ocupada sólo en un 1.5% de sus posiciones. Estas matrices se denominan *matrices dispersas* (ing., *sparse matrix*) y es evidente que su representación exige buscar una estrategia alternativa.

b) Representación encadenada

La implementación más razonable sigue la filosofía de las representaciones encadenadas y guarda sólo los elementos no nulos de la matriz dispersa; cada elemento formará parte exactamente de dos listas diferentes: la de elementos de su fila y la de elementos de su columna. En vez de duplicar los elementos en cada lista, se les asocian dos campos de encadenamiento y así pueden insertarse en las dos listas simultáneamente. Esta estrategia será especialmente útil cuando la relación sea etiquetada, porque no hará falta duplicar la etiqueta de la relación en varias listas diferentes. La estructura se denomina *multilista de grado dos*; multilista, porque los elementos forman parte de más de una lista, y de grado dos, porque forman parte exactamente de dos listas. Para abreviar, a las multilistas de grado dos las llamaremos simplemente *multilistas* (ing., *multilist*).

Dada esta política de representación, las operaciones sobre las relaciones quedan así:

- Insertar un par. Se busca la lista de la fila y la columna del elemento y se encadena la celda. Las celdas se han de mantener ordenadas porque la especificación establece que *fila* y *columna* deben devolver un conjunto ordenado y así no es necesario un

proceso explícito de ordenación; además, la obtención ordenada de los elementos puede ser útil en algunos contextos. Notemos que la inserción ordenada no enlentece en exceso la operación porque, igualmente, para evitar las repeticiones hay que recorrer una de las listas para ver si el par existe o no.

- Borrar un par. Se busca el par en la fila o la columna involucradas. Para actualizar las listas hay que conocer la posición que ocupan los elementos que lo preceden, lo que exigirá una exploración adicional en la lista en la que no se haya buscado el par. Esta segunda búsqueda puede evitarse encadenando doblemente las listas correspondientes.
- Comprobar si un par existe. Hay que recorrer una de las dos listas a la que pertenece hasta encontrarlo o llegar al final sin éxito. En el caso de que la longitud esperada de uno de los dos tipos de listas (de filas o de columnas) sea significativamente menor que la del otro, resulta mejor buscar en las primeras.
- Obtener una fila o columna. Hay que recorrer la lista y formar el conjunto resultante.

Existen todavía un par de detalles por concretar:

- Todas las celdas de una fila y de una columna están encadenadas, pero no hay un camino de acceso a la primera celda. Es necesario, pues, añadir una tabla índice que asocie a cada elemento su lista; si el número de filas y columnas es acotado y enumerable, se puede implementar la tabla con un simple vector de apuntadores a la primera celda de cada fila y de cada columna, si no, hay que organizar estos apuntadores en una lista o bien utilizar tablas de dispersión o árboles de búsqueda, si se pide un acceso eficiente.
- Los nodos se almacenan en un único vector o en memoria dinámica según se conozca o no el número de pares que habrá en la relación. En el ejemplo de los alumnos y las asignaturas, puede plantearse el uso de un vector habida cuenta la estimación existente de la ocupación total.
- Dentro de las celdas no hay ninguna información que indique qué columna o fila representan. Hay varias soluciones a este problema:
 - ◊ Añadir a cada celda los identificadores de la fila y la columna a los que pertenece, o bien apuntadores a la posición adecuada de la tabla índice si los identificadores ocupan demasiado espacio (por ejemplo, cadenas no exageradamente cortas); estos apuntadores serán atajos si la tabla se integra modularmente en la estructura. Es la solución óptima en tiempo, a costa de mantener dos campos adicionales (v. fig. 6.4, izq.).
 - ◊ Cerrar circularmente las listas, de forma que el último nodo de cada lista "apunte" a su cabecera; entonces la cabecera tendrá que identificar la fila o columna correspondiente. Hay que resaltar que, en este contexto, la circularidad significa que el último elemento de una lista apunta a la posición correspondiente de la tabla (v. fig. 6.4, der.). Esta opción es más costosa en tiempo, porque para saber la fila o la

columna a la que pertenece un elemento hay que recorrer la lista hasta el final. En cuanto al espacio, puede acarrear más o menos problemas según el uso de vectores o punteros en las diferentes estructuras encadenadas, y dependiendo de que la relación sea etiquetada o no, lo que puede permitir aprovechar campos ya existentes o bien obligar a declarar alguno adicional, posiblemente usando el mecanismo de tuplas variantes, si el lenguaje de programación lo presenta.

- ◊ Se puede optar por una solución intermedia: un tipo de listas con identificadores y el otro circular. Esta solución es útil cuando la longitud esperada de uno de los dos tipos es pequeña. Así, en el ejemplo de alumnos y asignaturas, como la lista de asignaturas por alumno siempre será muy corta, se podrá implementar circularmente, mientras que para las otras parece mas adecuado el uso de identificadores.

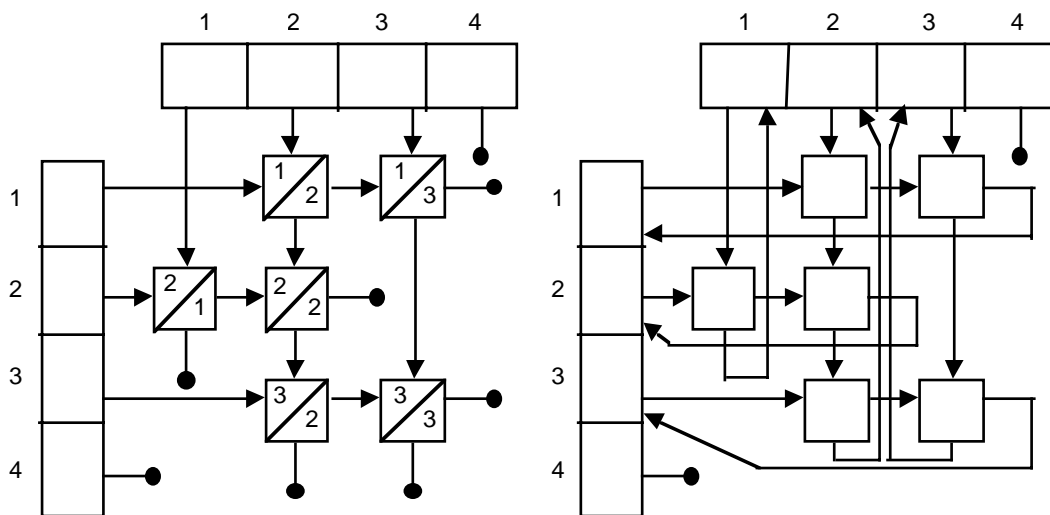


Fig. 6.4: relaciones implementadas por multilistas (a) con identificadores de los elementos; (b) con listas circulares.

El espacio empleado por esta solución es claramente menor si la matriz propia de la representación secuencial realmente era dispersa, pues queda $\Theta(r + s + n)$; es necesario incluir el tamaño de los dominios por si el número de pares de elementos relacionados realmente es muy pequeño y entonces el espacio de las tablas de acceso es relevante. Si suponemos que la tabla nos da acceso en tiempo constante a las listas, tenemos que el coste de las operaciones de inserción y supresión queda $\Theta(r + s)$ en el caso peor (elemento relacionado con todo otro elemento del dominio), lo mismo que la consulta. Además, el coste de las operaciones *fila* y *columna* tampoco se reduce en el caso peor, que es el mismo; ahora bien, debe verse que sí se reducen los otros casos, pues en la implementación encadenada

estas operaciones sólo acceden a tantas posiciones como elementos hay en la lista correspondiente, a diferencia de la representación secuencial que necesita acceder a todas las posiciones de la fila o columna del vector.

Esta propiedad tiene una consecuencia importante. En muchas ocasiones, no interesa tanto el recorrido individual de una fila o columna, sino el de todas ellas conjuntamente, con el objetivo de obtener todos los pares de elementos relacionados, ya sea de golpe, ya sea a medida que un algoritmo va evolucionando. Y en este contexto sí que notamos diferencia en la eficiencia. En el caso de representación secuencial, el coste de la obtención de todos estos pares es ineludiblemente $\Theta(r * s)$. En cambio, con la representación encadenada este coste queda reducido a $\Theta(r + n)$ o $\Theta(s + n)$, dependiendo de si recorremos por filas o por columnas (una vez más, el tamaño del dominio es relevante si hay pocos pares). Obviamente, en el caso peor se tiene $\Theta(n) = \Theta(r * s)$ y el coste es el mismo, pero en el caso habitual de que se cumpla $n < r * s$, y más si $n \ll r * s$, la diferencia es asintóticamente significativa.

En la figura 6.5 se muestra una representación de las multilistas en las que tanto las filas como las columnas forman listas circulares, usando vectores para acceder al inicio e implementando con punteros las estructuras lineales necesarias; se supone pues que los géneros A y B son válidos como índice de vector lo que se traduce en el uso de *ELEM_ORDENADO* como universo de caracterización (v. fig. 6.6). Este universo introduce diversas operaciones y propiedades sobre los elementos: han de ser un tipo con un orden total definido que permita identificar cuál es el primer elemento, *prim*, cuál es el último, *ult*, y cómo se obtiene el sucesor a un elemento dado, *suc*. También se dispondrá de las operaciones de igualdad. Además, para uso futuro, requerimos una constante n que dé el número de elementos dentro del género y una operación de comparación $<$ que representa el orden total.

En el resto de la implementación, destacamos el uso de tuplas variantes para la implementación del tipo de las celdas, y de diversas operaciones privadas cuyo comportamiento se especifica convenientemente. Notemos también que la ausencia de fantasmas complica algunos algoritmos, pero aun así no se incluyen para ahorrar espacio a la estructura. Por último, destaquemos que el invariante evita que dos listas de filas o dos listas de columnas compartan nodos, porque se violaría la condición de igualdad dada.

universo MULTILISTA_TODO_CIRCULAR (A, B son ELEM_ORDENADO)
implementa RELACIÓN (A, B son ELEM_<_=
usa BOOL
tipo relación es tupla
 fil es vector [A.elem] de ^nodo
 col es vector [B.elem] de ^nodo
 ftupla
ftipo
tipo privado nodo es
 tupla
 caso ult_fil? de tipo bool igual a
 cierto entonces punt_fil es A.elem
 falso entonces enc_fil es ^nodo
 caso ult_col? de tipo bool igual a
 cierto entonces punt_col es B.elem
 falso entonces enc_col es ^nodo
 ftupla
ftipo
invariante (r es relación)
 $\forall a: a \in A.elem: r.fil[a] \neq NULO \Rightarrow (cabecera_fil(r.fil[a]) = a \wedge ord_fil(r.fil[a])) \wedge$
 $\forall b: b \in B.elem: r.col[b] \neq NULO \Rightarrow (cabecera_col(r.col[b]) \wedge ord_col(r.col[b])) = b$
 donde se define *cabecera_fil*: ^nodo \rightarrow A.elem como:
 $p.^{ult_fil?} \Rightarrow cabecera_fil(p) = p.^{punt_fil}$
 $\neg p.^{ult_fil?} \Rightarrow cabecera_fil(p) = cabecera_fil(p.^{enc_fil}),$
 ord_fil: ^nodo \rightarrow bool como:
 $p.^{ult_fil?} \Rightarrow ord_fil(p) = p.^{cierto}$
 $\neg p.^{ult_fil?} \Rightarrow ord_fil(p) = (cabecera_col(p) < cabecera_col(p.^{enc_fil})) \wedge$
 $\wedge ord_fil(p.^{enc_fil})$
 y de forma similar *ord_col* y *cabecera_col*
función crea devuelve relación es
var r es relación; a es A.elem; b es B.elem fvar
 para todo a dentro de A.elem hacer r.fil[a] := NULO fpara todo
 para todo b dentro de B.elem hacer r.col[b] := NULO fpara todo
 devuelve r
función existe? (r es relación; a es A.elem; b es B.elem) devuelve bool es
var antf, antc, p son ^nodo; está? es bool fvar
 {se apoya íntegramente en la función auxiliar *busca*}
 <está?, antf, antc, p> := busca(r, a, b) {está? indica si <a, b> está en r}
 devuelve está?

Fig. 6.5: universo para la implementación del tipo abstracto de las relaciones.

```

función inserta (r es relación; a es A.elem; b es B.elem) devuelve relación es
var antf, antc, p son ^nodo; está? es bool fvar
  <está?, antf, antc, p> := busca(r, a, b)
  si ¬está? entonces {si existe, nada}
    p := obtener_espacio
    si p = NULO entonces error {no hay espacio}
    si no {inserción ordenada en la lista de filas, controlando si la fila está vacía}
      si antf = NULO entonces p^.ult_fil := (r.fil[a] = NULO) si no p^.ult_fil := antf^.ult_fil fsi
      si p^.ult_fil entonces p^.punt_fil := a si no p^.enc_fil := antf^.enc_fil fsi
      si antf = NULO entonces r.fil[a] := p si no antf^.ult_fil := falso; antf^.enc_fil := p fsi
      {repetición del proceso en la lista de columnas}
      si antc = NULO entonces p^.ult_col := (r.col[b] = NULO) si no p^.ult_col := antc^.ult_col fsi
      si p^.ult_col entonces p^.punt_col := b si no p^.enc_col := antc^.enc_col fsi
      si antc = NULO entonces r.col[b] := p si no antc^.ult_col := falso; antc^.enc_col := p fsi
    fsi
  fsi
devuelve r

función borra (r es relación; a es A.elem; b es B.elem) devuelve relación es
var antf, antc, p son ^nodo; está? es bool fvar
  {si existe, se obtienen los apuntadores a la celda y a sus predecesores en las listas}
  <está?, antf, antc, p> := busca(r, a, b)
  si está? entonces {si no existe, nada}
    {supresión de la lista de filas, vigilando si es el primero y/o el último}
    opción
      caso antf = NULO  $\wedge$  p^.ult_fil? hacer r.fil[a] := NULO
      caso antf = NULO  $\wedge$  ¬p^.ult_fil? hacer r.fil[a] := p^.enc_fil
      caso antf  $\neq$  NULO  $\wedge$  p^.ult_fil? hacer antf^.ult_fil? := cierto; antf^.punt_fil := a
      caso antf  $\neq$  NULO  $\wedge$  ¬p^.ult_fil? hacer antf^.enc_fil := p^.enc_fil
    fopción
      {repetición del proceso en la lista de la columna}
    opción
      caso antc = NULO  $\wedge$  p^.ult_col? hacer r.col[b] := NULO
      caso antc = NULO  $\wedge$  ¬p^.ult_col? hacer r.col[b] := p^.enc_col
      caso antc  $\neq$  NULO  $\wedge$  p^.ult_col? hacer antc^.ult_col? := cierto; antc^.punt_col := a
      caso antc  $\neq$  NULO  $\wedge$  ¬p^.ult_col? hacer antc^.enc_col := p^.enc_col
    fopción
      liberar_espacio(p)
  fsi
devuelve r

```

Fig. 6.5: universo para la implementación del tipo abstracto de las relaciones (cont.).

{Funciones *fila* y *columna*: siguen el encadenamiento correspondiente y, para cada celda, averiguan la columna o la fila siguiendo el otro campo. Esta indagación se lleva a término usando diversas funciones auxiliares}

<u>función</u> <i>fila</i> (<i>r es</i> relación; <i>a es</i> A.elem) <u>devuelve</u> <i>cjt_b es</i> <u>var</u> <i>s es</i> <i>cjt_b</i> ; <i>p es</i> ^nodo <i>fvar</i> <i>s</i> := crea; <i>p</i> := <i>r.fil</i> [<i>a</i>] <u>si</u> <i>p</i> ≠ NULO <u>entonces</u> <u>mientras</u> ¬ <i>p</i> ^.ult_fil? <u>hacer</u> <i>s</i> := añade(<i>s</i> , qué_col(<i>r</i> , <i>p</i>)) <i>p</i> := <i>p</i> ^.enc_fil <u>fmientras</u> <i>s</i> := añade(<i>s</i> , qué_col(<i>r</i> , <i>p</i>)) <u>fsi</u> <u>devuelve</u> <i>s</i>	<u>función</u> <i>columna</i> (<i>r es</i> relación; <i>b es</i> B.elem) <u>devuelve</u> <i>cjt_a es</i> <u>var</u> <i>s es</i> <i>cjt_a</i> ; <i>p es</i> ^nodo <i>fvar</i> <i>s</i> := crea; <i>p</i> := <i>r.col</i> [<i>b</i>] <u>si</u> <i>p</i> ≠ NULO <u>entonces</u> <u>mientras</u> ¬ <i>p</i> ^.ult_col? <u>hacer</u> <i>s</i> := añade(<i>s</i> , qué_fil(<i>r</i> , <i>p</i>)) <i>p</i> := <i>p</i> ^.enc_col <u>fmientras</u> <i>s</i> := añade(<i>s</i> , qué_fil(<i>r</i> , <i>p</i>)) <u>fsi</u> <u>devuelve</u> <i>s</i>
--	---

{Función *busca_col(r, p) → <qué_b, antc>*: usada en *busca*, busca la columna que representa *p* dentro de la relación *r* y, además, devuelve el apuntador al anterior dentro de la lista de la columna (que valdrá NULO si el elemento es el primero de la columna); existe una función *busca_fil*, similar pero actuando sobre las listas de filas

$$\mathcal{P} \equiv p \neq \text{NULO} \wedge (\exists b: b \in B.\text{elem}: p \in \text{cadena_col}(r.\text{col}[b]))$$

$$Q \equiv \text{qué_b} = \text{cabecera_col}(p) \wedge$$

$$(\text{antc} \neq \text{NULO} \Rightarrow \text{antc}^{\wedge}.\text{enc_col} = p) \wedge (\text{antc} = \text{NULO} \Rightarrow r.\text{col}[\text{qué_b}] = p),$$

siendo *cadena_col* similar a *cadena* siguiendo el encadenamiento de columna}

función privada *busca_col* (*r es* relación; *p es* ^nodo) devuelve <B.elem, ^nodo> es

var *antc*, *p*, *temp son* ^nodo; *qué_b es* bool *fvar*

{primero, se busca la columna}

temp := *p*

mientras ¬*temp*^.ult_col? hacer *temp* := *temp*^.enc_col fmientras

qué_b := *temp*^.punt_col {ya está localizada}

{a continuación, se busca el anterior a *p* en la columna}

antc := NULO; *temp* := *r.col*[*qué_b*]

mientras *temp* ≠ *p* hacer *antc* := *temp*; *temp* := *temp*^.enc_col fmientras

devuelve <*qué_b*, *antc*>

{Función auxiliar *qué_col(r, p)*: busca la columna que representa *p* dentro de la relación *r*; existe una función *qué_fil*, similar pero actuando sobre las filas.

$$\mathcal{P} \equiv p \neq \text{NULO} \wedge (\exists b: b \in B.\text{elem}: p \in \text{cadena_col}(r.\text{col}[b]))$$

$$Q \equiv \text{qué_col}(r, p) = \text{cabecera_col}(p) \}$$

función privada *qué_col* (*r es* relación; *p es* ^nodo) devuelve B.elem es

mientras ¬*p*^.ult_col? hacer *p* := *p*^.enc_col fmientras

devuelve *p*^.punt_col

Fig. 6.5: universo para la implementación del tipo abstracto de las relaciones (cont.).

{Función auxiliar $\text{busca}(r, a, b) \rightarrow \langle \text{encontrado}, \text{antf}, \text{antc}, p \rangle$: en caso que el par $\langle a, b \rangle$ exista, pone *encontrado* a cierto y devuelve: el apuntador p al elemento que lo representa dentro de las listas de filas y de columnas de la relación r , y también los apuntadores *antf* y *antc* a sus antecesores en estas listas. Si el par es el primero de alguna lista devuelve NULO en el puntero correspondiente al anterior en ella. Si el par $\langle a, b \rangle$ no existe pone *encontrado* a falso y *antf* y *antc* apuntan a los anteriores (según el criterio de ordenación correspondiente) de b y de a en las listas correspondientes a la fila a y a la columna b ; si a o b no tienen menores en la lista correspondiente, el puntero devuelto vale NULO

$\mathcal{P} \equiv \text{invariante}(r)$

$Q \equiv \text{encontrado} \equiv \exists q: q \in \text{cadena_fil}(r.\text{fil}[a]): \text{cabecera_col}(q) = b \wedge$
 $\text{encontrado} \Rightarrow (\text{cabecera_fil}(p) = a) \wedge (\text{cabecera_col}(p) = b) \wedge$
 $(\text{antf} \neq \text{NULO} \Rightarrow \text{antf}^{\text{enc_fil}} = p) \wedge (\text{antf} = \text{NULO} \Rightarrow r.\text{fil}[a] = p) \wedge$
 $(\text{antc} \neq \text{NULO} \Rightarrow \text{antc}^{\text{enc_col}} = p) \wedge (\text{antc} = \text{NULO} \Rightarrow r.\text{col}[b] = p)$
 $\neg \text{encontrado} \Rightarrow (\text{antf} \neq \text{NULO} \Rightarrow \text{cabecera_col}(\text{antf}) < b) \wedge$
 $(\text{antc} \neq \text{NULO} \Rightarrow \text{cabecera_fil}(\text{antc}) < a) \wedge$
 $(\text{antf} = \text{NULO} \wedge r.\text{fil}[a] \neq \text{NULO}) \Rightarrow \text{cabecera_col}(r.\text{fil}[a]) > b \wedge$
 $(\text{antc} = \text{NULO} \wedge r.\text{col}[b] \neq \text{NULO}) \Rightarrow \text{cabecera_fil}(r.\text{col}[b]) > a$

siendo *cadena_fil* similar a *cadena_col* siguiendo el encadenamiento de fila)

función privada *busca* (r es relación; a es A.elem; b es B.elem)

devuelve $\langle \text{bool}, \wedge \text{nodo}, \wedge \text{nodo}, \wedge \text{nodo} \rangle$ es

var encontrado, final es bool; antf, antc, p son $\wedge \text{nodo}$; qué_b es B.elem fvar

encontrado := falso; final := falso

si $r.\text{fil}[a] = \text{NULO}$ entonces antf := NULO

si no $p := r.\text{fil}[a]$; antf := NULO {se busca por la fila; antf: anterior en la fila}

mientras $\neg p^{\text{ult_fil}} \wedge \neg \text{final}$ hacer

$\langle \text{qué_b}, \text{antc} \rangle := \text{busca_col}(r, p)$ {antc: anterior en la columna}

si $\text{qué_b} \geq b$ entonces final := cierto; encontrado := ($\text{qué_b} = b$)

si no antf := p; $p := p^{\text{enc_fil}}$

fsi

fmientras

si $\neg \text{final}$ entonces {tratamiento de la última clave}

$\langle \text{qué_b}, \text{antc} \rangle := \text{busca_col}(r, p)$

si $\text{qué_b} = b$ entonces encontrado := cierto

si no si $\text{qué_b} < b$ entonces antf := p fsi

fsi

fsi

fsi

... repetición del proceso en la lista de columnas

devuelve $\langle \text{encontrado}, \text{antf}, \text{antc}, p \rangle$

funiverso

Fig. 6.5: universo para la implementación del tipo abstracto de las relaciones (cont.).

```

universo ELEM_ORDENADO caracteriza
  usa NAT, BOOL
  tipo elem
  ops prim, ult: → elem
    suc: elem → elem
    _=_, _≠_, _<_: elem elem → bool
    n: → nat
  error suc(ult)
  ecns n > 0 = cierto
    ...reflexividad, simetría y transitividad de _=_
    (v ≠ w) = ¬ (v = w)
    ...antireflexividad, antisimetría y transitividad de _<_
  funiverso

```

Fig. 6.6: caracterización de los elementos válidos como índice de vector.

6.2 Relaciones de equivalencia

En el resto del capítulo se estudian las relaciones binarias definidas sobre pares de elementos de un mismo dominio. Para empezar, en esta sección se estudian las *relaciones de equivalencia*, esto es, relaciones que cumplen la propiedad reflexiva, simétrica y transitiva. En muchos otros libros de texto los podemos encontrar bajo otras denominaciones: estructuras de partición, *MFsets*, *union-find sets* y otras que, aun cuando reflejan las operaciones que ofrecen, no denotan tan fielmente su significado.

Hay diversos modelos posibles para el TAD de las relaciones de equivalencia que se pueden adaptar los unos mejor que los otros a determinados contextos. En esta sección definimos un modelo que asocia un identificador a cada clase de equivalencia para poder referirse a ellas desde otras estructuras externas; para simplificar, consideramos que los identificadores son naturales. Dado V el conjunto base de los elementos de la relación, podemos definir las relaciones de equivalencia \mathcal{R}_V como funciones de los elementos a los naturales, $f: V \rightarrow \mathcal{N}$, siendo $f(v)$ el identificador de la clase a la que pertenece el elemento v , $v \in V$.

Para $R \in \mathcal{R}_V$, $v \in V$ e $i_1, i_2 \in \mathcal{N}$, definimos las siguientes operaciones sobre el TAD:

- Crear la relación vacía: *crea*, devuelve la relación I tal que cada elemento forma una clase por sí mismo, es decir, $\forall v, w: v, w \in V: v \neq w \Rightarrow I(v) \neq I(w)$.
- Determinar la clase en la que se halla un elemento: *clase*(R, v), devuelve el natural que identifica la clase de equivalencia dentro de R que contiene v ; es decir, devuelve $R(v)$.

- Establecer la congruencia entre dos clases: *fusiona*(R, i_1, i_2), devuelve la relación resultado de fusionar todos los elementos de las dos clases identificadas por i_1 e i_2 en una misma clase, o deja la relación inalterada si i_1 o i_2 no identifican ninguna clase dentro de la relación, o bien si $i_1 = i_2$; es decir, devuelve la relación R' que cumple:

$$\begin{aligned} &\Diamond \{ \forall v: v \in V: (R(v) = i_1 \vee R(v) = i_2) \Rightarrow R'(v) = i_1 \} \vee \\ &\quad \{ \forall v: v \in V: (R(v) = i_1 \vee R(v) = i_2) \Rightarrow R'(v) = i_2 \} \\ &\Diamond \forall v: v \in V: (R(v) \neq i_1 \wedge R(v) \neq i_2) \Rightarrow R'(v) = R(v). \end{aligned}$$

- Averiguar el número de clases de la relación: *cuántos?*(R), devuelve el número de naturales que son imagen de algún elemento, $||\text{codom}(R)||$.

Notemos que el modelo no determina cuáles son los identificadores iniciales de las clases, porque no afecta al funcionamiento del tipo, sólo obliga a que sean diferentes entre ellos. Se puede decir lo mismo por lo que se refiere al identificador de la clase resultante de hacer una fusión y, por ello, no se obliga a que sea uno u otro, sino que simplemente se establece que todos los elementos de una de las dos clases fusionadas vayan a parar a la otra.

En la fig. 6.7 se muestra el contexto de uso habitual de este TAD de las relaciones. Primero, se forma la relación con tantas clases como elementos, a continuación, se organiza un bucle que selecciona dos elementos según un criterio dependiente del algoritmo concreto y, si están en clases diferentes, las fusiona, y el proceso se repite hasta que la relación consta de una única clase. En el apartado 6.6.2 se muestra un algoritmo sobre grafos, el algoritmo de Kruskal para el cálculo de árboles de expansión minimales, que se basa en este esquema. Notemos que el número máximo de fusiones que se pueden efectuar sobre una relación es $n-1$, siendo $n = ||V||$, pues cada fusión junta dos clases en una, por lo que $n-1$ fusiones conducen a una relación con una única clase. En cambio, el número de ejecuciones de *clase* y *cuántos* no puede determinarse con exactitud, ya que el criterio de selección puede generar pares de elementos congruentes un número elevado de veces. Eso sí, como mínimo el algoritmo ejecutará *clase* $2(n-1)$ veces y *cuántos*, n veces. Además, si la selección asegura que no puede generarse el mismo par de elementos más de una vez, la cota superior del número de ejecuciones de estas operaciones es $n(n-1)$ para *clase* y $n(n-1)/2 + 1$ para *cuántos*, porque el número diferente de pares (no ordenados) es $\sum k: 1 \leq k \leq n: k$.

```

R := crea
mientras cuántos?(R) > 1 hacer
    seleccionar dos elementos  $v_1$  y  $v_2$  según cierto criterio
     $i_1 := \text{clase}(R, v_1); i_2 := \text{clase}(R, v_2)$ 
    si  $i_1 \neq i_2$  entonces R := fusiona(R,  $i_1, i_2$ ) fsi
fmientras

```

Fig. 6.7: algoritmo que usa las relaciones de equivalencia.

A veces este modelo de las relaciones de equivalencia no se adapta totalmente a los requerimientos de una aplicación concreta; las variantes más habituales son:

- Considerar la relación como una función parcial, de manera que haya elementos del conjunto de base que no estén en ninguna clase, en cuyo caso, se acostumbra a sustituir la operación *crea* por otras dos: la primera, para crear la relación "vacía" (es decir, sin ninguna clase), y una segunda para añadir a la relación una nueva clase que contenga un único elemento que se explicita como parámetro. A veces, el identificador de la clase vendrá asignado por el contexto (es decir, será un parámetro más).
- Identificar las clases no por naturales, sino por los elementos que la forman. Según este esquema, la operación *clase* se sustituye por otra que, dados dos elementos, comprueba si son congruentes o no, mientras que la operación *fusiona* tiene como parámetros dos elementos que representan las clases que es necesario fusionar.

En la fig. 6.8 se muestra la especificación del tipo *releq* de las relaciones de equivalencia. Para dotar al TAD de un significado dentro de la semántica inicial (es decir, para construir el álgebra cociente de términos asociada al tipo con la cual se establece un isomorfismo) es necesario resolver las dos indeterminaciones citadas al describir el modelo. Por lo que respecta al identificador de la fusión, se elige arbitrariamente el último parámetro de la función como identificador de la clase resultado. En lo que se refiere a los identificadores iniciales, se toma la opción de numerar las clases de uno al número de elementos. Caracterizamos los elementos de la relación mediante el universo *ELEM_ORDENADO*.

Por lo que respecta a la estrategia de la especificación, se introducen dos operaciones privadas, *vacía* (que representa la relación sin ninguna clase) e *inserta* (que añade un elemento a una clase dada, que puede no existir), que forman un conjunto de constructoras generadoras impuras que facilitan considerablemente la especificación. También surgen otras operaciones auxiliares: *vacía?*, que comprueba si el identificador dado denota una clase existente, y *pon_todos*, que se encarga de construir una a una las clases iniciales añadiendo los elementos desde el primero hasta el último. Los errores que aparecen se escriben por la aplicación estricta del método general de especificación presentado en la sección 1.3, pero es imposible que se produzcan, dado el uso que se hace de las operaciones.

En el resto del apartado nos ocupamos de la implementación del TAD, partiendo de dos representaciones lineales diferentes y llegando a una arborescente que asegura un coste asintótico medio constante en todas las operaciones. Supondremos que el conjunto de base es un tipo por enumeración que permite indexar directamente un vector²; en caso contrario, sería necesario organizar una tabla de dispersión para mantener los costes que aquí se darán o, en el caso que la dispersión no sea aconsejable, cualquier otra estructura que incrementará el coste de las operaciones.

² Esta suposición es lógica, dados los requerimientos sobre los elementos establecidos en la especificación *ELEM_ORDENADO*.

universo RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO) *es*
usa NAT, BOOL
tipo releq
ops privada vacía: \rightarrow releq
 privada inserta: releq elem nat \rightarrow releq
 crea: \rightarrow releq
 clase: releq elem \rightarrow nat
 fusiona: releq nat nat \rightarrow releq
 cuántos?: releq \rightarrow nat
 privada vacía?: releq nat \rightarrow bool
 privada pon_todos: elem nat \rightarrow releq
errores $\forall r \in \text{releq}; \forall i_1, i_2 \in \text{nat}; \forall v \in \text{elem}$
 inserta(inserta(r, v, i₁), v, i₂); clase(vacía); [$i > n \vee \text{NAT.ig}(i, \text{cero})$] \Rightarrow inserta(R, v, i)
ecns $\forall r \in \text{releq}; \forall i, i_1, i_2 \in \text{nat}; \forall v, v_1, v_2 \in \text{elem}$
 inserta(inserta(r, v₁, i₁), v₂, i₂) = inserta(inserta(r, v₂, i₂), v₁, i₁)
 crea = pon_todos(prim, 1)
 clase(inserta(r, v, i), v) = i
 $[v_1 \neq v_2] \Rightarrow \text{clase}(\text{inserta}(r, v_1, i), v_2) = \text{clase}(r, v_2)$
 fusiona(vacía, i₁, i₂) = vacía
 $[\text{NAT.ig}(i, i_1)] \Rightarrow \text{fusiona}(\text{inserta}(r, v, i), i_1, i_2) = \text{inserta}(\text{fusiona}(r, i_1, i_2), v, i_2)$
 $[\neg \text{NAT.ig}(i, i_1)] \Rightarrow \text{fusiona}(\text{inserta}(r, v, i), i_1, i_2) = \text{inserta}(\text{fusiona}(r, i_1, i_2), v, i)$
 cuántos?(vacía) = 0
 $[\text{vacía?}(r, i)] \Rightarrow \text{cuántos?}(\text{inserta}(r, v, i)) = \text{cuántos?}(r) + 1$
 $[\neg \text{vacía?}(r, i)] \Rightarrow \text{cuántos?}(\text{inserta}(r, v, i)) = \text{cuántos?}(r)$
 vacía?(vacía, i) = cierto; vacía?(inserta(c, v, i₁), i₂) = vacía?(c, i₂) \wedge NAT.ig(i₁, i₂)
 pon_todos(ult, n) = inserta(vacía, ult, n)
 $[v \neq \text{ult}] \Rightarrow \text{pon_todos}(v, i) = \text{inserta}(\text{pon_todos}(\text{suc}(v), i+1), v, i)$
funiverso

Fig. 6.8: especificación del TAD de las relaciones de equivalencia.

6.2.1 Implementaciones lineales

Dado que el algoritmo de la fig. 6.7 ejecuta más veces *clase* que *fusiona*, nos planteamos su optimización. Una representación intuitiva consiste en organizar un vector indexado por los elementos, tal que en cada posición se almacene el identificador de la clase en la que resida el elemento. Así, el coste temporal de *clase* queda constante (acceso directo a una posición del vector), mientras que *fusiona* es lineal, porque su ejecución exige modificar el identificador de todos aquellos elementos que cambien de clase mediante un recorrido del

vector; el coste de $n-1$ ejecuciones de *fusiona* es, pues, $\Theta(n^2)$. Por otro lado, el coste de *crea* es lineal (suficientemente bueno, puesto que con toda probabilidad sólo se ejecutará una vez) y el de *cuántos?* será constante si añadimos un contador.

En la fig. 6.9 se muestra una variante de este esquema que puede llegar a mejorar el coste asintótico de *fusiona* sin perjudicar *clase* a costa de emplear espacio adicional. Por un lado, se encadenan todos los elementos que están en la misma clase; así, al fusionar dos clases no es necesario recorrer todo el vector para buscar los elementos que cambian de clase. Por otro lado, se introduce un nuevo vector indexado por identificadores que da acceso al primer elemento de la cadena de cada clase.

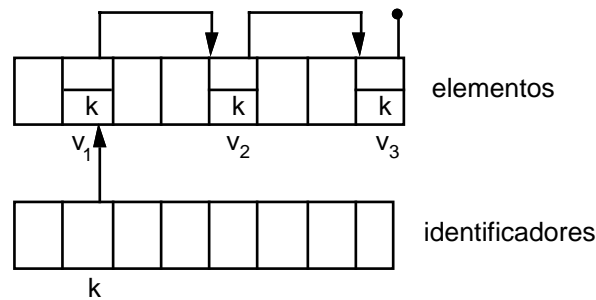


Fig. 6.9: implementación lineal del TAD de las relaciones de equivalencia.

Está claro que la única operación que varía de coste es *fusiona*; concretamente, la ejecución de *fusiona*(R, i_1, i_2) exige modificar sólo las posiciones del vector de elementos que estén dentro de la cadena correspondiente a la clase i_2 : cada posición ha de tomar el valor i_1 y, además, el último elemento de la cadena se encadena al primero de la cadena correspondiente a la clase i_1 . Es necesario preguntarse, pues, cuántas posiciones pueden formar parte de la cadena asociada a i_2 .

Como siempre, buscamos el caso peor. Supongamos que la k -ésima fusión del algoritmo de la fig. 6.7 implica a dos clases i_1 e i_2 de la relación R , tales que en la clase identificada por i_1 sólo hay un elemento, $|\{v : v \in V : R(v) = i_1\}| = 1$, y en la clase identificada por i_2 hay k elementos, $k = |\{v : v \in V : R(v) = i_2\}|$; entonces, el número de posiciones del vector de elementos que es necesario modificar es k . Si esta situación se repite en las $n-1$ fusiones del algoritmo, $n = |V|$, el número total de modificaciones del vector de elementos es $\sum k : 1 \leq k \leq n-1 : k = n(n-1)/2$, es decir, $\Theta(n^2)$. Dicho de otra manera, el coste asintótico de *fusiona* después de las $n-1$ ejecuciones posibles no varía en el caso peor.

La solución es sencilla. Como ya se ha dicho, el identificador de la clase resultante de la fusión no es significativo y, por ello, se puede redefinir de manera que *fusiona*(R, i_1, i_2)

mueva los elementos de la clase más pequeña a la más grande (siempre referido a las clases identificadas por i_1 y i_2). Es decir, siendo $k_1 = \|\{v: v \in V: R(v) = i_1\}\|$ y $k_2 = \|\{v: v \in V: R(v) = i_2\}\|$, $fusiona(R, i_1, i_2)$ devuelve la relación R' que cumple³:

- Si $k_2 \geq k_1$, entonces $\forall v: v \in V: \{ R(v) = i_1 \Rightarrow R'(v) = i_2 \} \wedge \{ R(v) \neq i_1 \Rightarrow R'(v) = R(v) \}$
- Si $k_2 < k_1$, entonces $\forall v: v \in V: \{ R(v) = i_2 \Rightarrow R'(v) = i_1 \} \wedge \{ R(v) \neq i_2 \Rightarrow R'(v) = R(v) \}$

Según esta estrategia, no hay ningún elemento que cambie de clase más de $\lceil \log_2 n \rceil$ veces después de las $n-1$ fusiones posibles, puesto que la clase resultante de una fusión tiene una dimensión como mínimo el doble que la más pequeña de las clases que en ella intervienen. Como el coste total de *fusiona* depende exclusivamente del número de cambios de clase de los elementos (el resto de tareas de la función son comparativamente desdeñables), el coste total de las $n-1$ fusiones es $\Theta(n \log n)$. Notemos, no obstante, que no se puede asegurar que el coste individual de una fusión sea $\Theta(\log n)$, pero en el caso general (y en el algoritmo de la fig. 6.7 en particular) esta característica no es negativa, porque no será habitual tener fusiones aisladas.

Por lo que respecta a la eficiencia espacial de este esquema, para implementar la nueva estrategia de fusiones es necesario añadir contadores de elementos al vector de los identificadores de las clases. El espacio resultante es, pues, más voluminoso que la primera estructura presentada, aunque el coste asintótico no varía.

En la fig. 6.10 se muestra la implementación del tipo. Se dispone de dos vectores *idents* y *elems* que implementan la estrategia descrita y del típico contador de clases. Se adopta la convención de que, si un identificador no denota ninguna clase válida (porque la clase ha desaparecido en alguna fusión), su contador de elementos vale cero. Esta propiedad es fundamental al establecer el invariante (concretamente, para fijar el valor del contador de clases y para identificar el comienzo de las cadenas válidas dentro del vector de elementos). El final de la cadena correspondiente a una clase de equivalencia se identifica por un elemento que se apunta a sí mismo (de esta manera no es necesario definir un elemento especial que sirva de marca, ni emplear un campo booleano adicional). En el invariante, las cadenas se definen de la manera habitual usando esta convención, y no es necesario exigir explícitamente que la intersección sea vacía, ya que se puede deducir de la fórmula dada. La implementación de las operaciones es inmediata: se introduce una función auxiliar *cambia* para cambiar todos los elementos de una clase a otra tal como se ha dicho en el modelo.

³ También se debe adaptar la especificación ecuacional del tipo (queda como ejercicio para el lector).

universo RELACIÓN_DE_EQUIVALENCIA_LINEAL(ELEM_ORDENADO) es
implementa RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO)
usa ENTERO, BOOL
tipo releq es
tupla
 elems es vector [elem] de tupla id es nat; enc es elem ftupla
 idents es vector [de 1 a n] de tupla cnt es nat; prim es elem ftupla
 nclases es nat
ftupla
ftipo
invariante (R es releq):

$$R.nclases = || \{ i: 1 \leq i \leq n: R.idents[i].cnt \neq 0 \} || \wedge$$

$$\forall i: i \in \{ i: 1 \leq i \leq n: R.idents[i].cnt \neq 0 \}:$$

$$R.idents[i].cnt = || cadena(R, R.idents[i].prim) || \wedge$$

$$\forall v: v \in cadena(R, R.idents[i].prim): R.elems[v].id = i \wedge$$

$$\{ \cup i: i \in \{ k: 1 \leq k \leq n: R.idents[k].cnt \neq 0 \}: cadena(R, R.idents[i].prim) \} = elem$$
 donde *cadena*: $releq\ nat \rightarrow \mathcal{P}(elem)$ se define:

$$R.elems[v].enc = v \Rightarrow cadena(R, v) = \{v\}$$

$$R.elems[v].enc \neq v \Rightarrow cadena(R, v) = \{v\} \cup cadena(R, R.elems[v].enc)$$

función crea devuelve releq es
var R es releq; v es elem; i es nat fvar
 i := 1
para todo v dentro de elem hacer
 {se forma la clase i únicamente con v}
 R.elems[v].id := i; R.elems[v].enc := v
 R.idents[i].prim := v; R.idents[i].cnt := 1
 i := i + 1
fpara todo
 R.nclases := n
devuelve R
función clase (R es releq; v es elem) devuelve nat es
devuelve R.elems[v].id
función cuántos? (R es releq) devuelve nat es
devuelve R.nclases

Fig. 6.10: implementación lineal de las relaciones de equivalencia.

función fusiona (R es releq; i₁, i₂ son nat) devuelve releq es
si (i₁ = 0) \vee (i₁ > n) \vee (i₂ = 0) \vee (i₂ > n) \vee (R.idents[i₁].cnt = 0) \vee (R.idents[i₂].cnt = 0)
entonces error
sino si (i₁ \neq i₂) entonces
 {se comprueba qué clase tiene más elementos}
 si R.idents[i₁].cnt \geq R.idents[i₂].cnt entonces R := cambia(R, i₂, i₁)
 sino R := cambia(R, i₁, i₂)
fsi
 R.nclases := R.nclases - 1 {en cualquier caso desaparece una clase}
fsi
fsi
devuelve R

{Función auxiliar *cambia*: dados dos identificadores de clase, *fuelle* y *destino*, implementa el trasvase de elementos de la primera a la segunda. Como precondition, los identificadores denotan dos clases válidas y diferentes

$\mathcal{P} \equiv (\text{fuente} \neq \text{destino}) \wedge (\text{fuente} > 0) \wedge (\text{fuente} \leq n) \wedge (\text{destino} > 0) \wedge (\text{destino} \leq n) \wedge (\text{R.idents}[\text{fuente}].\text{cnt} \leq \text{R.idents}[\text{destino}].\text{cnt}) \wedge (\text{R} = \text{R}_0)$

$\mathcal{Q} \equiv \forall i: i \in \text{cadena}(\text{R}_0, \text{R}_0.\text{idents}[\text{fuente}].\text{prim}): \text{R}.\text{elems}[i].\text{id} = \text{destino} \wedge \forall k: \{ i: 1 \leq i \leq n: \text{R.idents}[i].\text{cnt} \neq 0 \}:$

$\forall i: i \in \text{cadena}(\text{R}_0, \text{R}_0.\text{idents}[k].\text{prim}): \text{R}.\text{elems}[i].\text{id} = k \wedge$

$\text{R.idents}[\text{fuente}].\text{cnt} = 0 \wedge$

$\text{R.idents}[\text{destino}].\text{cnt} = \text{R}_0.\text{idents}[\text{fuente}].\text{cnt} + \text{R}_0.\text{idents}[\text{destino}].\text{cnt} \}$

función privada cambia (R es releq; fuelle, destino son nat) devuelve releq es
var v es elem fvar

v := R.idents[fuelle].prim

repetir {cambiamos los identificadores de los elementos de la clase *fuelle*}

$\{I \equiv \forall i: i \in \text{cadena}(\text{R}_0, \text{R}_0.\text{idents}[\text{fuente}].\text{prim}) - \text{cadena}(\text{R}_0, \text{R}_0.\text{idents}[v].\text{prim}):$
 R.elems[i].id = destino}

R.elems[v].id := destino; v := R.elems[v].enc

hasta que R.elems[v].enc = v {condición de parada: siguiente de $v = v$ }

R.elems[v].id := destino

{luego se encadenan las secuencias correspondientes a las dos clases}

R.elems[v].enc := R.idents[destino].prim

R.idents[destino].prim := R.idents[fuelle].prim

{por último se actualizan la información de las clases}

R.idents[destino].cnt := R.idents[destino].cnt + R.idents[fuelle].cnt

R.idents[fuelle].cnt := 0 {marca de clase vacía}

devuelve R

funiverso

Fig. 6.10: implementación lineal de las relaciones de equivalencia (cont.).

6.2.2 Implementación arborescente

Las implementaciones lineales que acabamos de ver son suficientemente buenas si no se formulan requerimientos de eficiencia sobre el tipo, o bien si el número de elementos dentro de la relación es pequeño. Igualmente, hay usos del tipo diferentes del esquema general presentado en la fig. 6.7 en los cuales la ineficiencia de *fusiona* no es tan importante. Por ejemplo, puede que el objetivo final no sea tener una única clase, sino simplemente procesar una colección de equivalencias para agrupar los elementos bajo determinado criterio y, en este caso, seguramente el coste inicial de construir la relación con la aplicación sucesiva de *fusiona* será irrelevante comparado con el coste posterior de las consultas con *clase*. Es aconsejable, no obstante, buscar una representación que optimice al máximo el coste del algoritmo de la fig. 6.7, y éste es el objetivo del resto de la sección.

La idea básica para mejorar la fusión consiste en evitar cualquier recorrido de listas, por lo que no todos los elementos guardarán directamente el identificador de la clase a la cual pertenecen y no será preciso tratarlos todos en cada fusión en que intervengan. Para ser más exactos, para cada clase habrá un único elemento, que denominamos *representante* de la clase, que guardará el identificador correspondiente; los otros elementos que sean congruentes accederán al representante siguiendo una cadena de apuntadores.

El funcionamiento de la representación es el siguiente:

- Al crear la relación, se forman las n clases diferentes; cada elemento es el representante de su clase y guarda el identificador.
- Al fusionar, se encadena el representante de la clase menor con el representante de la clase mayor, el cual conserva su condición de representante de la nueva clase. Será necesario, pues, asegurar el acceso rápido a los representantes de las clases mediante un vector indexado por identificador de clase.
- Para consultar el identificador de la clase en la que reside un elemento dado, es necesario buscar su representante siguiendo los encadenamientos.

La repetición del proceso de fusión puede hacer que diversos elementos apunten a un mismo representante, dando lugar a una estructura arborescente. Concretamente, cada clase forma un árbol, donde la raíz es el representante y el resto de elementos apuntan cada uno su padre. Un elemento v es padre de otro elemento w , porque en algún momento se han fusionado dos clases A y B , tales que v era el representante de A , w el representante de B y la clase A tenía más elementos que la clase B . En la fig. 6.11 se muestra un ejemplo. Se define $V = \{a, b, \dots, h\}$ tal que inicialmente la clase $[a]$ se identifica con el uno, la clase $[b]$ con el dos, etc., y a continuación se realizan diversas fusiones hasta obtener una única clase. En caso de clases igual de grandes, se elige arbitrariamente el identificador de la segunda clase para denotar el resultado. Para mayor claridad, se dibujan directamente los árboles sin mostrar la distribución de los elementos dentro del vector correspondiente.

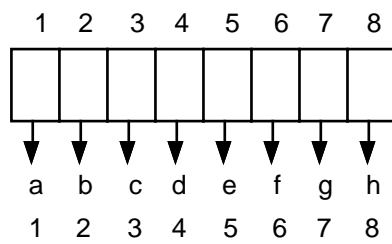
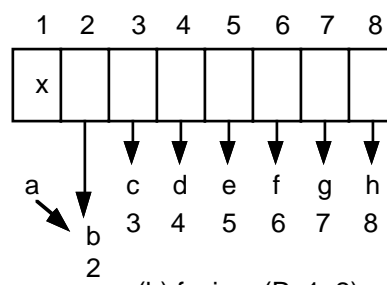
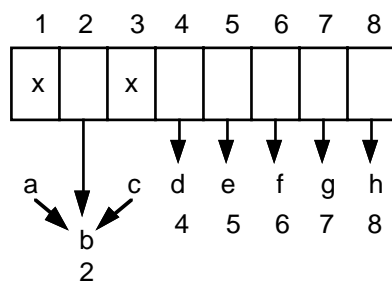
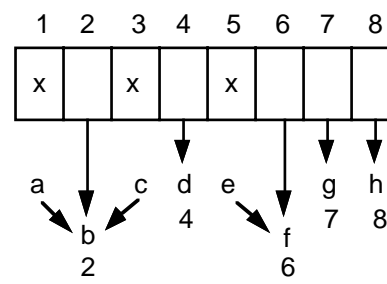
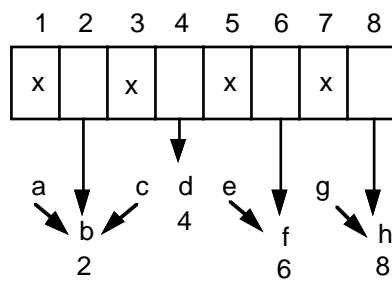
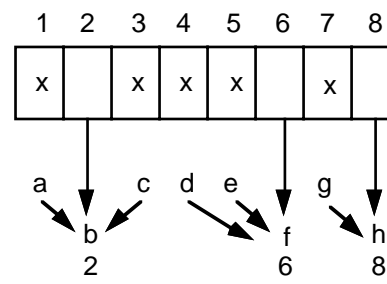
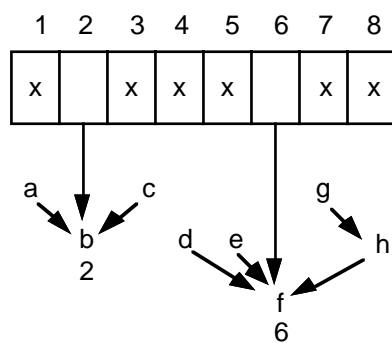
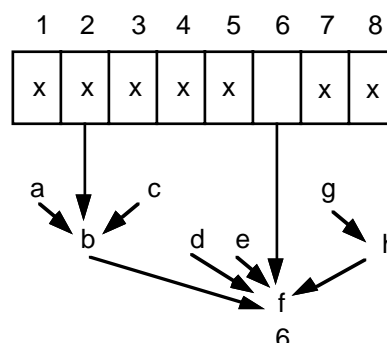
(a) Creación de la relación R (b) $\text{fusiona}(R, 1, 2)$ (c) $\text{fusiona}(R, 2, 3)$ (d) $\text{fusiona}(R, 5, 6)$ (e) $\text{fusiona}(R, 7, 8)$ (f) $\text{fusiona}(R, 4, 6)$ (g) $\text{fusiona}(R, 8, 6)$ (h) $\text{fusiona}(R, 2, 6)$

Fig. 6.11: ejemplo de funcionamiento de la representación arborescente del tipo releq.

En la fig. 6.12 se muestra finalmente la implementación del universo correspondiente. En el vector de elementos, los nodos que no representan ninguna clase se caracterizan por el valor cero en su campo identificador. Igualmente, los naturales que no identifican ninguna clase tienen el contador de elementos a cero en la posición correspondiente del vector de identificadores; opcionalmente, podríamos haber usado tuplas variantes. La codificación de las operaciones es muy parecida: la fusión sigue el mismo esquema pero ahora la función auxiliar da forma arborescente a las clases tal como se ha explicado. Por lo que respecta a *clase*, hay una búsqueda desde el nodo consultado hasta la raíz del árbol. El invariante es muy similar al caso anterior, usando una función que devuelve el representante de la clase correspondiente a un elemento dado.

La eficiencia espacial es idéntica al enfoque lineal. En cuestión de tiempo, *fusiona* queda efectivamente constante, mientras que *clase* pasa a ser función de la profundidad del nodo buscado. Notemos que, cada vez que se hace una fusión, los elementos que cambian de clase aumentan su profundidad en uno; dado que un elemento cambia $\lceil \log_2 n \rceil$ veces de clase como máximo, la altura de los árboles está acotada por este valor y, en consecuencia, la operación queda $\Theta(\log n)$.

6.2.3 Compresión de caminos

Con la estrategia arborescente tal como se ha explicado, el algoritmo sobre relaciones de la fig. 6.7 no es más eficiente que con el uso de la implementación lineal (incluso empeora), porque no sirve de nada mejorar las fusiones sin que las consultas de identificadores sean igualmente rápidas. Por este motivo, introducimos finalmente una técnica llamada *compresión de caminos* (ing., *path compression*) que reduce el coste asintótico de una secuencia de operaciones *fusiona* y *clase* a lo largo del tiempo, aunque alguna ejecución individual pueda salir perjudicada.

En la implementación de la fig. 6.12, al ejecutar *clase*(R, v) se recorre el camino C que hay entre v y la raíz de su árbol, donde se encuentra el identificador de la clase. Si posteriormente se repite el proceso para un nodo w antecesor de v dentro del árbol correspondiente, se examinan otra vez todos los nodos entre w y la raíz que ya se habían visitado antes, y que pueden ser muchos. Para evitarlo, se puede aprovechar la primera búsqueda para llevar todos los nodos del camino C al segundo nivel del árbol (es decir, se cuelgan directamente de la raíz); posteriores búsquedas del representante de cualquier descendente de los nodos de C (incluidos los propios) serán más rápidas que antes de la reorganización; concretamente, si el nodo v situado en el nivel k_v es descendiente de un nodo $u \in C$ situado en el nivel k_u , la búsqueda de v pasa de seguir k_v encadenamientos a seguir sólo $k_v - k_u + 2$.

universo RELACIÓN_DE_EQUIVALENCIA_ARBORESCENTE(ELEM_ORDENADO) es

implementa RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO)

usa ENTERO, BOOL

tipo releq es

tupla

elems es vector [elem] de tupla id es nat; padre es elem ftupla

idents es vector [de 1 a n] de tupla cnt es nat; raíz es elem ftupla

nclases es nat

ftupla

ftipo

invariante (R es releq):

$R.nclases = || \{ i: 1 \leq i \leq n: R.idents[i].cnt \neq 0 \} || \wedge$

$\forall i \in \{ i: 1 \leq i \leq n: R.idents[i].cnt \neq 0 \}:$

$R.idents[i].cnt = || \{ v: v \in \text{elem}: \text{representante}(R, v) = R.idents[i].raíz \} || \wedge$

$R.elems[R.idents[i].raíz].id = i \wedge$

donde *representante*: *releq elem* \rightarrow *elem* se define:

$R.elems[v].id = 0 \Rightarrow \text{representante}(R, v) = \text{representante}(R, R.elems[v].padre)$

$R.elems[v].id \neq 0 \Rightarrow \text{representante}(R, v) = v$

función crea devuelve releq es

var R es releq; v es elem; i es nat fvar

i := 1

para todo v dentro de elem hacer

{se forma la clase i únicamente con v}

R.elems[v].id := i

R.idents[i].raíz := v; R.idents[i].cnt := 1

i := i + 1

fpara todo

R.nclases := n

devuelve R

función clase (R es releq; v es elem) devuelve nat es

mientras R.elems[v].id = 0 hacer v := R.elems[v].padre fmientras {se busca la raíz}

devuelve R.elems[v].id

función cuántos? (R es releq) devuelve nat es

devuelve R.nclases

Fig. 6.12: implementación arborescente de las relaciones de equivalencia.

```

función fusiona (R es releq;  $i_1, i_2$  son nat) devuelve releq es
  si ( $i_1 = 0$ )  $\vee$  ( $i_1 > n$ )  $\vee$  ( $i_2 = 0$ )  $\vee$  ( $i_2 > n$ )  $\vee$  (R.idents[ $i_1$ ].cnt = 0)  $\vee$  (R.idents[ $i_2$ ].cnt = 0)
    entonces error
  sino si ( $i_1 \neq i_2$ ) entonces
    {se comprueba qué clase tiene más elementos}
    si R.idents[ $i_1$ ].cnt  $\geq$  R.idents[ $i_2$ ].cnt llavors R := cambia(R,  $i_2, i_1$ )
      si no R := cambia(R,  $i_1, i_2$ )
    fsi
    R.nclases := R.nclases - 1 {en cualquier caso desaparece una clase}
  fsi
devuelve R

{Función auxiliar cambia: dados dos identificadores de clase, fuelle y destino,
cuelga el árbol asociado a fuelle como hijo del árbol asociado a destino}
 $\mathcal{P} \equiv (\text{fuente} \neq \text{destino}) \wedge (\text{fuente} > 0) \wedge (\text{fuente} \leq n) \wedge (\text{destino} > 0) \wedge$ 
 $(\text{destino} \leq n) \wedge (\text{R.idents}[\text{fuente}].\text{cnt} \leq \text{R.idents}[\text{destino}].\text{cnt}) \wedge (\text{R} = \text{R}_0)$ 
 $\mathcal{Q} \equiv \forall v: \text{representante}(\text{R}_0, v) = \text{fuente}: \text{representante}(\text{R}, v) = \text{R.idents}[\text{destino}].\text{raíz}$ 
 $\wedge \forall v: \text{representante}(\text{R}_0, v) \neq \text{fuente}: \text{representante}(\text{R}, v) = \text{representante}(\text{R}_0, v)$ 
 $\wedge \text{R.idents}[\text{fuente}].\text{cnt} = 0$ 
 $\wedge \text{R.idents}[\text{destino}].\text{cnt} = \text{R}_0.\text{idents}[\text{fuente}].\text{cnt} + \text{R}_0.\text{idents}[\text{destino}].\text{cnt}$  }
función privada cambia (R es releq; fuelle, destino son nat) devuelve releq es
var v es elem fvar
  {primero se cuelga el árbol fuelle del árbol destino}
  R.elems[R.idents[fuelle].raíz].padre := R.idents[destino].raíz
  R.elems[R.idents[fuelle].raíz].id := 0 {marca que no es raíz}
  {a continuación se actualizan los contadores de elementos}
  R.idents[destino].cnt := R.idents[destino].cnt + R.idents[fuelle].cnt
  R.idents[fuelle].cnt := 0 {marca de clase vacía}
devuelve R
funiverso

```

Fig. 6.12: implementación arborescente de las relaciones de equivalencia (cont.).

Esta idea presenta un problema: la raíz del árbol correspondiente a la clase no es accesible hasta que no finaliza la búsqueda del representante, de manera que los nodos de C no se puede subir durante la búsqueda misma. Por esto, se recorre C dos veces: en la primera vez, se localiza la raíz (se averigua así el resultado de la consulta) y, durante la segunda, realmente se cuelgan los nodos de C de la raíz. El algoritmo resultante se presenta en la fig. 6.13.

```

función clase (R es releq; v es elem) devuelve nat es
var temp, raíz son elem fvar
    {primero se busca la raíz del árbol}
    raíz := v
    mientras R.elems[raíz].id = 0 hacer raíz := R.elems[raíz].padre fmientras
    {luego se cuelgan de la raíz todos los nodos del camino entre v y la raíz}
    mientras R.elems[v].id = 0 hacer
        temp := v; v := R.elems[v].padre; R.elems[temp].padre := raíz
    fmientras
    devuelve R.elems[raíz].id

```

Fig. 6.13: codificación de clase con compresión de caminos.

Es difícil analizar el coste resultante del algoritmo. Si bien *fusiona* es $\Theta(1)$, es evidente que puede haber ejecuciones individuales de *clase* costosas (eso sí, nunca más ineficientes que $\Theta(\log n)$), pero es necesario destacar que, precisamente, cuanto peor es una ejecución individual, mejor organizado queda el árbol y su uso futuro se optimiza, porque acerca el máximo número posible de nodos a la raíz (v. fig. 6.14). Se puede demostrar que el coste de una secuencia de $n-1$ ejecuciones de *fusiona* y k de *clase*, $k \geq n$, es $\Theta(k\alpha(k, n))$, siendo α una función no decreciente casi inversa de la llamada *función de Ackerman*, que cumple que $1 \leq \alpha(k, n) \leq 4$ para todo k y n razonables (v. [CLR90, pp. 450-458] y [BrB97, pp. 203-204]); a efectos prácticos, el coste global de la secuencia de k operaciones es, pues, equivalente a $\Theta(k)$ (aunque aplicando estrictamente la definición de Θ no es así, no obstante el crecimiento lento de α). Este estudio global del coste se denomina *coste amortizado* (ing., *amortized complexity*) y, en nuestro caso, nos permite asegurar que la secuencia de $k+n-1$ operaciones se comporta como si cada una de ellas fuera $\Theta(1)$; este resultado es más preciso que el análisis habitual del caso peor, que comporta multiplicar el coste individual de las operaciones en el caso peor por la cota superior de su número de ejecuciones, lo cuál resultaría en $\Theta(k \log n)$ en nuestro contexto.

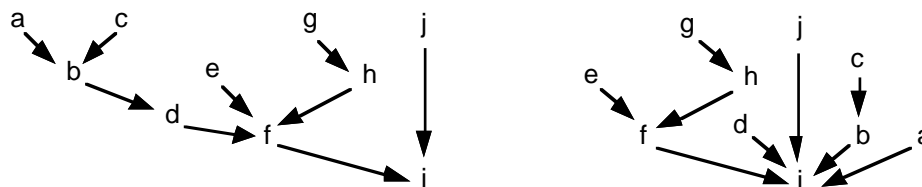


Fig. 6.14: a la derecha, compresión de caminos tras buscar *a* en el árbol de la izquierda.

6.3 Grafos

En la primera sección hemos introducido el TAD de las relaciones binarias sobre dos dominios cualesquiera de elementos; si los dos dominios son el mismo, la relación se denomina *grafo*⁴ (ing., *graph*). Por ello, se puede considerar un grafo como un conjunto de pares ordenados tales que los elementos de cada par cumplen el enunciado de la relación; los elementos del dominio de la relación se llaman *vértices* o *nodos* (ing., *vertex* o *node*) y el par que representa dos vértices relacionados se llama *arista* o *arco* (ing., *edge* o *arc*).

Un ejemplo típico de grafo es la configuración topológica de una red metropolitana de transportes, donde los vértices son las estaciones y las aristas los tramos que las conectan. A menudo los grafos representan una distribución geográfica de elementos dentro del espacio⁵ (ciudades, componentes dentro de un circuito electrónico, computadores pertenecientes a una red, etc.). La representación de la red metropolitana en forma de grafo permite formular algunas cuestiones interesantes, como por ejemplo averiguar el camino más rápido para ir de una estación a otra. El estudio de éste y otros algoritmos configura el núcleo central del resto del capítulo. Hay que destacar, no obstante, que no se introducen algunas versiones especialmente hábiles de estos algoritmos, puesto que su dificultad excede los objetivos de este libro, sobre todo en lo que se refiere al uso de estructuras de datos avanzadas; el lector que esté interesado puede consultar, por ejemplo, [BrB97] para una visión general y [CLR90] para un estudio en profundidad.

Del mismo modo que hay varios tipos de listas o de árboles, se dispone de diferentes modelos de grafos. Concretamente, definimos cuatro TAD que surgen a partir de las cuatro combinaciones posibles según los dos criterios siguientes:

- Un grafo es *dirigido* (ing., *directed*; también se abrevia por *digrafo*) si la relación no es simétrica; si lo es, se denomina *no dirigido*.
- Un grafo es *etiquetado* (ing., *labelled* o *weighted*) si la relación es valorada; si no lo es, se denomina *no etiquetado*. Los valores de la relación son sus *etiquetas* (ing., *label*).

En la fig. 6.15 se presenta un ejemplo de cada tipo. Para mostrar los grafos gráficamente, se encierran en un círculo los identificadores de los vértices y las aristas se representan mediante líneas que unen estos círculos; las líneas llevan una flecha si el grafo es dirigido y, si es etiquetado, el valor de la etiqueta aparece a su lado. En cada caso se dice cuál es el conjunto V de vértices y el enunciado R de la relación. En el resto de la sección estudiaremos detalladamente la especificación y la implementación de los grafos dirigidos y etiquetados, y comentaremos brevemente los otros modelos.

⁴ Hay una clase particular de grafos, llamados *grafos bipartitos* (ing., *bipartite graph*), que pueden definirse sobre dos dominios distintos, que no se estudian aquí; v., por ejemplo, [AHU83, pp.245-249].

⁵ De hecho, los grafos fueron definidos en el siglo XVIII por el matemático L. Euler para decidir si era posible atravesar todos los puentes de la ciudad de Königsberg, Prusia (actualmente Kaliningrado, Rusia, ciudad bañada por un río que rodea una isla) sin pasar más de una vez por ninguno de ellos.

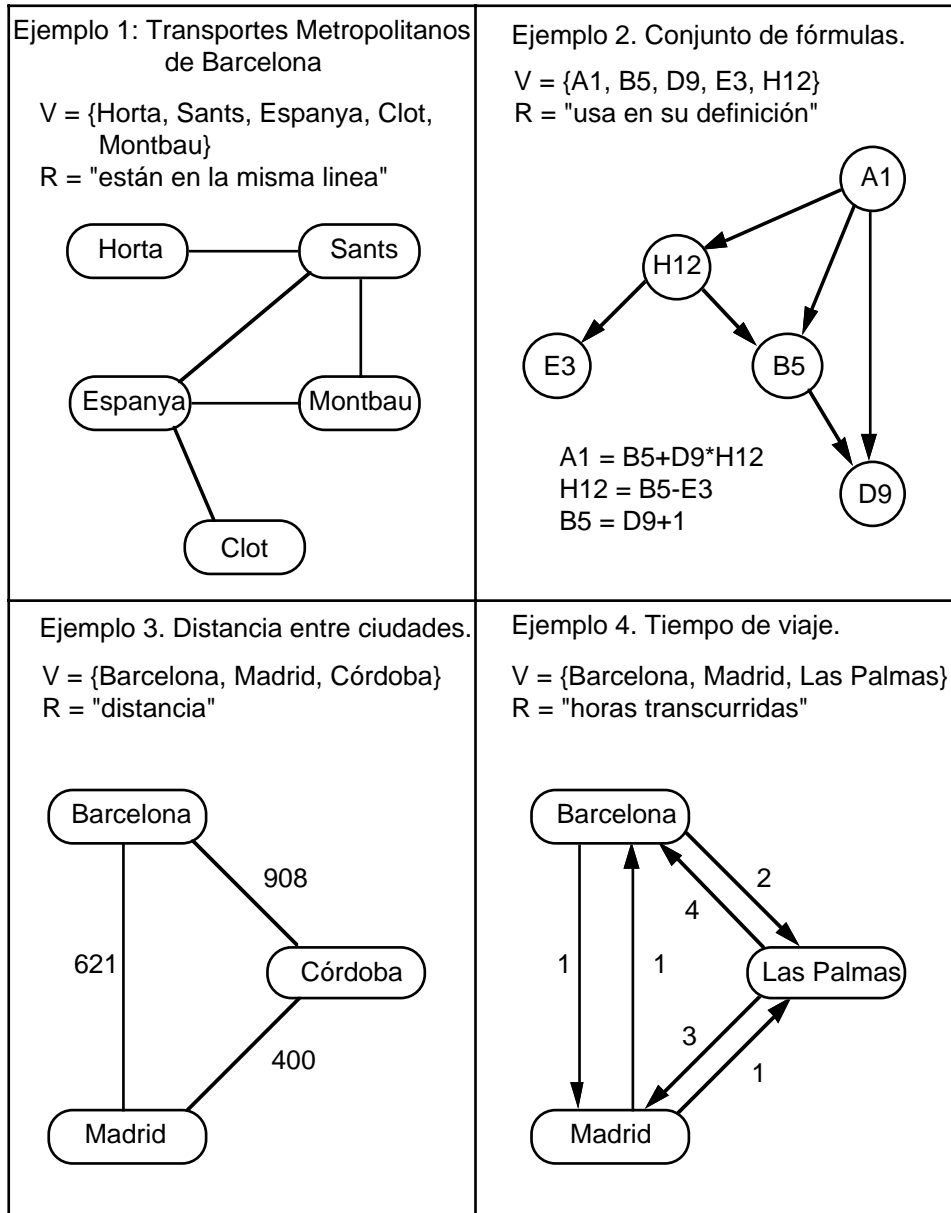


Fig. 6.15: diversos ejemplos de grafos.

6.3.1 Modelo y especificación

Dado un dominio V de vértices y un dominio E de etiquetas, definimos un grafo dirigido y etiquetado, g , como una función que asocia etiquetas a pares de vértices, $g \in \{f: V \times V \rightarrow E\}$ (igual que las relaciones binarias etiquetadas, de las que son un caso particular). A partir de ahora, denotaremos por n el número de vértices del grafo $g \in \{f: V \times V \rightarrow E\}$, $n = \|V\|$, y por a el número de aristas, $a = \|\text{dom}(g)\|$. Por lo que respecta al resto de modelos, en los grafos no dirigidos tenemos que, dados dos vértices cualesquiera u y v , se satisface la condición $\langle u, v \rangle \in \text{dom}(g) \Leftrightarrow \langle v, u \rangle \in \text{dom}(g) \wedge g(u, v) = g(v, u)$, mientras que en el caso de grafos g no etiquetados, el codominio de la función es el álgebra \mathcal{B} de valores cierto y falso, $g \in \{f: V \times V \rightarrow \mathcal{B}\}$, donde $g(u, v)$ vale cierto si u y v cumplen el enunciado de la relación⁶.

Cabe destacar que la mayoría de textos del ámbito de las estructuras de datos definen un grafo como un par ordenado de dos conjuntos, el conjunto de vértices y el conjunto de aristas. La especificación, la implementación y los diversos algoritmos sobre grafos que se presentan en este libro pueden adaptarse sin problemas a este modelo.

De ahora en adelante supondremos que V es finito (y, por consiguiente, $\text{dom}(g)$ también); si no, la mayoría de los algoritmos y las implementaciones que estudiaremos en las próximas secciones no funcionarán. Supondremos también que no hay aristas de un vértice a sí mismo (es decir, que la relación es antirreflexiva), ni más de una arista entre el mismo par de vértices si el grafo es no dirigido, ni más de una arista en el mismo sentido entre el mismo par de vértices si el grafo es dirigido; estas propiedades serán garantizadas por la especificación algebraica del tipo. En estas condiciones, el número máximo de aristas⁷ de un grafo dirigido de n nodos es $n^2 - n$, si de cada nodo sale una arista al resto (en los grafos no dirigidos, hay que dividir esta magnitud por dos); este grafo se denomina *completo* (ang., *complete* o *full*). Si el grafo no es completo, pero el número de aristas se acerca al máximo, diremos que es *denso*; por el contrario, si el grafo tiene del orden de n aristas o menos, diremos que es *disperso*. Supondremos, por último, que E presenta un valor especial \perp , que denota la inexistencia de arista; si el valor \perp no puede identificarse en E , se pueden hacer las mismas consideraciones que en el TAD de las relaciones etiquetadas.

Dado g un grafo dirigido y etiquetado, $g \in \{f: V \times V \rightarrow E\}$, dados $u, v \in V$ dos vértices y $e \in E$ una etiqueta, $e \neq \perp$, definimos las siguientes operaciones sobre el tipo:

- Crear el grafo vacío: *crea*, devuelve la función f_\emptyset tal que $\text{dom}(f_\emptyset) = \emptyset$. La función f_\emptyset representa el grafo con todos los vértices, pero sin ninguna arista.
- Añadir una nueva arista al grafo: *añade*(g, u, v, e) devuelve el grafo g' definido como:
 - $\diamond \text{dom}(g') = \text{dom}(g) \cup \{\langle u, v \rangle\} \wedge g'(u, v) = e$
 - $\diamond \forall u', v': \langle u', v' \rangle \in \text{dom}(g) - \{\langle u, v \rangle\}: g'(u', v') = g(u', v')$.

⁶ En este tipo de grafo se puede adoptar también el modelo $\mathcal{R}_{V \times V}$ de las relaciones no valoradas.

⁷ El número máximo de aristas interviene en el cálculo de la cota superior del coste de los algoritmos sobre grafos.

- Borrar una arista del grafo: $\text{borra}(g, u, v)$ devuelve el grafo g' definido como:
 - $\diamond \text{dom}(g') = \text{dom}(g) - \{ \langle u, v \rangle \}.$
 - $\diamond \forall u', v': \langle u', v' \rangle \in \text{dom}(g'): g'(u', v') = g(u', v').$
- Consultar la etiqueta asociada a una arista del grafo: $\text{etiqueta}(g, u, v)$, devuelve $g(u, v)$ si $\langle u, v \rangle \in \text{dom}(g)$, y \perp en caso contrario.
- Obtener el conjunto de vértices a los cuales llega una arista desde uno dado, llamados *sucesores* (ing., *successor*), juntamente con su etiqueta: $\text{suc}(g, u)$ devuelve el conjunto recorrible $s \in \mathcal{P}(V \times E)$, tal que $\forall v: v \in V: \langle u, v \rangle \in \text{dom}(g) \Leftrightarrow \langle v, g(u, v) \rangle \in s$.
- Obtener el conjunto de vértices de los cuales sale una arista a uno dado, llamados *predecesores* (ing., *predecessor*), juntamente con su etiqueta: $\text{pred}(g, v)$ devuelve el conjunto recorrible $s \in \mathcal{P}(V \times E)$, tal que $\forall v: v \in V: \langle u, v \rangle \in \text{dom}(g) \Leftrightarrow \langle u, g(u, v) \rangle \in s$.

Notemos, pues, que el modelo de los grafos dirigidos etiquetados es casi idéntico al de las relaciones binarias valoradas y esto se refleja en la especificación algebraica correspondiente, que consiste en una instancia de las relaciones, asociando el tipo de los vértices a los dos géneros que definen la relación y el tipo de las etiquetas como valor de la relación; los vértices serán un dominio con una relación de orden $<$ (v. fig. 6.16). Previamente se hacen diversos renombramientos para mejorar la legibilidad del universo y, después de la instancia, se renombran algunos símbolos para obtener la nueva signatura. Como la relación se define sobre un único dominio de datos, los tipos de los pares y de los conjuntos definidos en el universo de las relaciones són idénticos y, por ello, se les da el mismo nombre⁸. Por último, se prohíben las aristas de un nodo a sí mismo y la inserción de aristas indefinidas (esta última restricción posiblemente ya existiría en *RELACIÓN_ETIQUETADA*).

universo DIGRAFO_ETIQ(V es ELEM_<=, E es ELEM_ESP) es
renombra V.elem por vértice, E.elem por etiq, E.esp por indef
instancia RELACIÓN_ETIQUETADA(A, B son ELEM_<=, X es ELEM_ESP) donde
 A.elem, B.elem son vértice, A.=, B.= son V.=, A.<, B.< son V.<
 X.elem es etiq, X.esp es indef
renombra a_y_valor por vértice_y_etiq, cjt_a_y_valor por cjt_vértices_y_etiqs
 b_y_valor por vértice_y_etiq, cjt_b_y_valor por cjt_vértices_y_etiqs
 _a por _v, _b por _v, _val por _et
 relación por grafo, inserta por añade, consulta por etiqueta
 fila por suc, columna por pred
error $\forall g \in \text{grafo}; \forall v \in \text{vértice}; \forall e \in \text{etiq}: \text{añade}(g, v, v, e); \text{añade}(g, v, w, \text{indef})$
funiverso

Fig. 6.16: especificación del TAD de los grafos dirigidos y etiquetados.

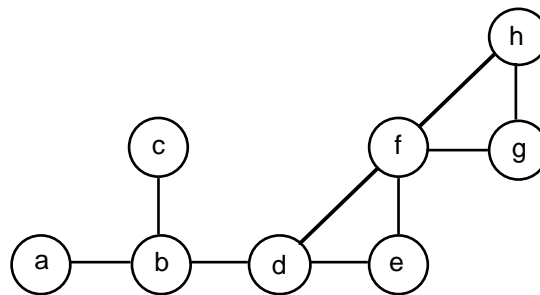
⁸ Formalmente este paso no es del todo correcto, porque los dos tipos son diferentes aunque se llamen igual, pero se podría corregir con poco esfuerzo añadiendo funciones de conversión del tipo; como el resultado complicaría el universo innecesariamente, se deja tal como está.

En cuanto al resto de modelos, hay pocos cambios en la especificación. Los comentarios sobre las variantes recorribles y abiertas de grafos son los mismos que dimos en el caso general de relaciones binarias. En los grafos no etiquetados desaparece el universo de caracterización de las etiquetas y, en consecuencia, la operación *etiqueta*, sustituida por la operación *existe?*: $\text{grafo } \text{vértice } \text{vértice} \rightarrow \text{bool}$, que responde si dos vértices están directamente unidos o no por una arista. Por lo que respecta a los grafos no dirigidos, hay que establecer una relación de conmutatividad de los dos vértices parámetros de *añade*. Además, la distinción entre sucesores y predecesores de un vértice pierde sentido y se sustituye por el concepto más general de *adyacencia* (ing., *adjacency*): dos vértices son adyacentes si hay una arista que los une.

Una característica común a todos estos modelos de grafo es que el conjunto de sus vértices es fijo; no obstante, ocasionalmente puede interesar que sea un subconjunto W , $W \subseteq V$, que pueda crecer y menguar durante la existencia del grafo. Esta variación introduce un cambio en el modelo porque a las operaciones del tipo habrá que añadir dos más para poner y sacar vértices del grafo, y otra para comprobar su presencia. La construcción precisa del modelo, así como su especificación e implementación, quedan como ejercicio para el lector.

Por último, antes de comenzar a estudiar implementaciones y algoritmos, definimos diversos conceptos a partir del modelo útiles en secciones posteriores (la especificación algebraica de los mismos se propone en el ejercicio 6.1). Un *camino* (ing., *path*) de longitud $s \geq 0$ dentro de un grafo $g \in \{f: V \times V \rightarrow E\}$ es una sucesión de vértices $v_0 \dots v_s$, $v_i \in V$, tales que hay una arista entre todo par consecutivo de nodos $\langle v_i, v_{i+1} \rangle$ de la secuencia. Dado el camino $v_0 \dots v_s$, diremos que sus *extremidades* son v_0 y v_s , y al resto de vértices los denominaremos *intermedios*. Igualmente, diremos que es *propio* (ing., *proper*) si $s > 0$; que es *abierto* si $v_0 \neq v_s$, o *cerrado* en caso contrario; que es *simple* si no pasa dos veces por el mismo vértice (exceptuando la posible igualdad entre extremidades) y que es *elemental* si no pasa dos veces por el mismo arco (un camino simple es forzosamente elemental). El camino $w_0 \dots w_r$ es *subcamino* (ing., *subpath*) del camino $v_0 \dots v_s$ si está incluido en él, es decir, si se cumple la propiedad: $\exists i: 0 \leq i \leq s - r - 1: v_0 \dots v_s = v_0 \dots v_i w_0 \dots w_r v_{i+r+2} \dots v_s$. Todas estas definiciones son independientes de si el grafo es o no dirigido y de si es o no etiquetado (v. fig. 6.17).

Dado un grafo $g \in \{f: V \times V \rightarrow E\}$, diremos que dos vértices $v, w \in V$ están *conectados* si existe algún camino dentro de g que tenga como extremidades v y w ; el sentido de la conexión es significativo si el grafo es dirigido. Si todo par de vértices de V está conectado diremos que g es *conexo* (ing., *connected*); en caso de grafos dirigidos, requeriremos la conexión de los vértices en los dos sentidos y hablaremos de un grafo *fuertemente conexo* (ing., *strongly connected*). Definimos un *componente conexo* (ing., *connected component*; *fuertemente conexo* en el caso dirigido) del grafo g o, simplemente, *componente*, como un subgrafo de g conexo maximal (es decir, que no forma parte de otro subgrafo conexo de g); un grafo $g' \in \{f: V \times V \rightarrow E\}$ es *subgrafo* de g si $\text{dom}(g') \subseteq \text{dom}(g)$ y además, en caso de ser etiquetado, se cumple que $\forall \langle u, v \rangle: \langle u, v \rangle \in \text{dom}(g'): g'(u, v) = g(u, v)$.

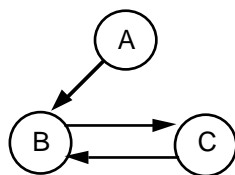


$a b c b d e$ es camino propio abierto no elemental
 $a b d e f g$ es camino propio abierto simple
 $a b d e f d$ es camino propio abierto elemental no simple
 $a b d g$ no es camino

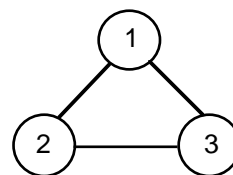
Fig. 6.17: estudio de la existencia de caminos en un grafo.

En un grafo dirigido, un *ciclo* (ing., *cycle*) es un camino cerrado propio; un grafo sin ciclos se llama *grafo acíclico*. Si, además, el camino no contiene ningún subcamino cerrado propio, entonces se denomina *ciclo elemental*. De forma simétrica, diremos que el ciclo A es un *subciclo* del ciclo B si A es un subcamino de B . En el caso de grafos no dirigidos, es necesario evitar ciclos de la forma $vw...wv$ (es decir, caminos que vuelven al principio pasando por la misma arista de salida); por tanto, haremos las siguientes precisiones:

- El camino $A = v_0 v_1 \dots v_{n-1} v_n$ es *simplificable* si $v_0 = v_n$ y $v_1 = v_{n-1}$, y su *simplificación* es el camino v_0 .
- La *simplificación extendida* de un camino es la sustitución de todo sus subcaminos simplificables por la simplificación correspondiente.
- Un *ciclo* es un camino cerrado propio A tal que su simplificación extendida sigue siendo un camino cerrado propio. Si, además, la simplificación extendida de A no contiene ningún subcamino cerrado propio, entonces el ciclo se denomina *ciclo elemental*. Diremos que el ciclo $A = v_0 \dots v_0$ es un *subciclo* del ciclo B si A es un subcamino de B .



A no es un ciclo
 $A B$ no es un ciclo
 $B C B$ es un ciclo elemental
 $B C B C B$ es un ciclo no elemental



1 no es un ciclo
 $1 2 1$ no es un ciclo
 $1 2 3 1$ es un ciclo elemental
 $1 2 3 2 1$ no es un ciclo

Fig. 6.18: estudio de la existencia de ciclos en un grafo.

6.3.2 Implementación

Estudiaremos tres tipos de implementaciones para grafos dirigidos y etiquetados; la extensión al resto de casos es inmediata y se comenta brevemente al final de la sección. Consideramos que los vértices se pueden usar directamente para indexar vectores⁹ y obtenerlos uno detrás de otro en un bucle "para todo" mediante las operaciones definidas en *ELEM_ORDENADO* (v. fig. 6.6) o las propias de los iteradores, según sea el caso.

a) Implementación por matriz de adyacencia

Dado g un grafo dirigido y etiquetado definido sobre un dominio V de vértices y con etiquetas de tipo E , $g \in \{f : V \times V \rightarrow E\}$, definimos su representación por *matriz de adyacencia* (ing., *adjacency matrix*) como una matriz bidimensional M , indexada por pares de vértices, que almacena en cada posición $M[u, v]$ la etiqueta de la arista que une u y v (v. fig. 6.19). Si entre los vértices u y v no hay ninguna arista, el valor de $M[u, v]$ es el valor indefinido de las aristas (de acuerdo con la especificación del tipo); si la especificación no considera la existencia de este valor especial, se requiere añadir a cada posición un campo booleano que marque la ausencia de arista.

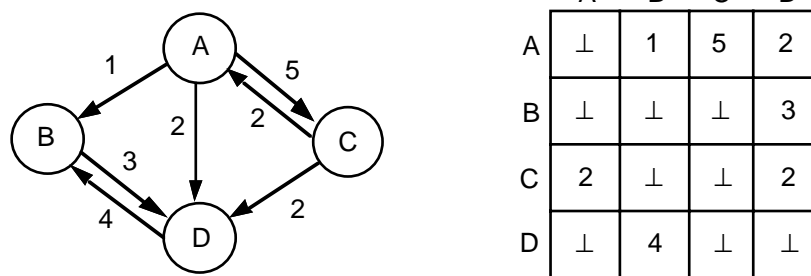


Fig. 6.19: un grafo y su implementación por matriz de adyacencia.

En la fig. 6.20 se muestra una posible implementación de esta estrategia. Notemos el uso implícito de las instancias de los pares y de los conjuntos recorribles, formuladas en la especificación de las relaciones y renombradas en la especificación de los grafos. Por su parte, el invariante simplemente confirma la inexistencia de aristas de un vértice a sí mismo, lo que obliga a que las posiciones de la diagonal principal valgan \perp . Como esta representación exige comparar etiquetas, el universo de caracterización correspondiente es *ELEM_ESP_*, similar a *ELEM_2_ESP_* pero con una única constante *esp*. La ordenación de los elementos en *suc* y *pred* coincide con el recorrido ordenado de las filas y columnas de la matriz.

⁹ Si no, sería necesario usar el TAD *tabla* como alternativa a los vectores; según la eficiencia exigida, la tabla se podría implementar posteriormente por dispersión, árboles de búsqueda o listas.

```

universo DIGRAFO_ETIQ_MATRIZ(V es ELEM_ORDENADO, E es ELEM_ESP_=) es
  implementa DIGRAFO_ETIQ(V es ELEM_<=, E es ELEM_ESP)
  renombra V.elem por vértice, E.elem por etiq, E.esp por indef
  usa BOOL
  tipo grafo es vector [vértice, vértice] de etiq ftipo
  invariante (g es grafo):  $\forall v: v \in \text{vértice}: g[v, v] = \text{indef}$ 
  función crea devuelve grafo es
  var g es grafo; v, w son vértice fvar
    para todo v dentro de vértice hacer
      para todo w dentro de vértice hacer g[v, w] := indef fpara todo
    devuelve g
  función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
    si (v = w)  $\vee$  (et = indef) entonces error si no g[v, w] := et fsi
    devuelve g
  función borra (g es grafo; v, w son vértice) devuelve grafo es
    g[v, w] := indef
    devuelve g
  función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
    devuelve g[v, w]
  función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
  var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer
      si g[v, w]  $\neq$  indef entonces s := añade(s, <w, g[v, w]>) fsi
    devuelve s
  función pred (g es grafo; v es vértice) ret cjt_vértices_y_etiqs es
  var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer
      si g[w, v]  $\neq$  indef entonces s := añade(s, <w, g[w, v]>) fsi
    devuelve s
universo

```

Fig. 6.20: implementación de los grafos dirigidos etiquetados por matriz de adyacencia.

El coste temporal de la representación, suponiendo una buena implementación de los conjuntos recorribles, queda: *crea* es $\Theta(n^2)$, las operaciones individuales sobre aristas son $\Theta(1)$ y tanto *suc* como *pred* quedan $\Theta(n)$, independientemente de cuántos sucesores o predecesores tenga el nodo en cuestión. Así, el coste de consultar todas las aristas del grafo es $\Theta(n^2)$. El coste espacial de la estructura es considerable, $\Theta(n^2)$.

b) Implementación por listas de adyacencia

Para un grafo g dirigido y etiquetado, definido sobre un dominio V de vértices y con etiquetas de tipo E , $g \in \{f: V \times V \rightarrow E\}$, se asocia a cada vértice una lista con sus sucesores; para acceder al inicio de estas listas, se usa un vector indexado por vértice; es la denominada representación por *listas de adyacencia* (ing., *adjacency lists*).

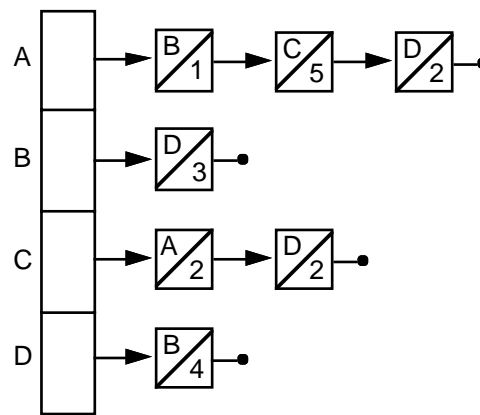


Fig. 6.21: implementación del grafo de la fig. 6.19 por listas de adyacencia.

Para que la implementación sea correcta respecto la especificación, deben ordenarse las listas de sucesores usando la operación $<$. Ésta es una diferencia fundamental con la implementación de grafos por listas de adyacencia propuestas en otros textos, donde el orden de los nodos es irrelevante. También se podría haber elegido una noción de corrección de las implementaciones menos restrictiva que la usada aquí, pero hemos preferido no salirnos del enfoque seguido en el resto del libro. En todo caso, la eficiencia asintótica no se ve afectada por el hecho de ordenar las listas o no ordenarlas.

En la fig. 6.22 se muestra una posible implementación de esta estrategia, que representa las citadas listas usando una instancia de las listas con punto de interés que incluya operaciones de insertar y borrar según el orden de los elementos; el resultado es una codificación simple, clara, modular y resistente a las modificaciones. Se exige una representación encadenada por punteros, por motivos de eficiencia. Notemos que la situación concreta del punto de interés no es significativa y, por ello, no necesita mantenerse durante la ejecución de las

diversas operaciones (si bien podría acarrear alguna inconsistencia al definir el modelo inicial). El invariante del grafo confirma de nuevo la ausencia de aristas de un vértice a sí mismo con una variante de la típica función *cadena*, que se define usando operaciones del TAD *lista*; las listas cumplirán su propio invariante (en concreto, el criterio de ordenación de los elementos). El universo introduce una función auxiliar, *busca*, que devuelve un booleano indicando si un elemento dado está dentro de una lista o no y, si está, coloca el punto de interés sobre él; gracias a esta función, evitamos recorridos redundantes de la lista ordenada.

```

universo DIGRAFO_ETIQ_LISTAS(V es ELEM_ORDENADO, E es ELEM_ESP) es
  implementa DIGRAFO_ETIQ(V es ELEM_<_, E es ELEM_ESP)
  renombra V.elem por vértice, E.elem por etiq, E.esp por indef
  usa BOOL
  instancia LISTA_INTERÉS_ORDENADA(ELEM_<) donde
    elem es vértice_y_etiq, < es V.<
    implementada con LISTA_INTERÉS_ORDENADA_ENC_PUNT(ELEM_<)
  renombra lista por lista_vértices_y_etiq
  tipo grafo es vector [vértice] de lista_vértices_y_etiqs ftipo
  invariante (g es grafo):  $\forall v: v \in \text{vértice}: v \notin \text{cadena}(\text{principio}(g[v]))$ 
    donde cadena:  $\text{lista\_vértices\_y\_etiqs} \rightarrow \mathcal{P}(\text{vértice})$  se define como:
      final?(l)  $\Rightarrow$  cadena(l) =  $\emptyset$ 
       $\neg \text{final?}(l) \Rightarrow \text{cadena}(l) = \{\text{actual}(l).v\} \cup \text{cadena}(\text{avanza}(l))$ 
  función crea devuelve grafo es
  var g es grafo; v es vértice fvar
    para todo v dentro de vértice hacer
      g[v] := LISTA_INTERÉS_ORDENADA.crea
    fpara todo
  devuelve g
  función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
  var está? es bool fvar
    si (v = w)  $\vee$  (et = indef) entonces error
    si no <g[v], está?> := busca(g[v], w)
      si está? entonces
        g[v] := LISTA_INTERÉS_ORDENADA.borra_actual(g[v])
      fsi
      g[v] := LISTA_INTERÉS_ORDENADA.inserta_actual(g[v], <w, et>)
    fsi
  devuelve g

```

Fig. 6.22: implementación de los grafos dirigidos etiquetados por listas de adyacencia.

```

función borra (g es grafo; v, w son vértice) devuelve grafo es
var está? es bool fvar
    <g[v], está?> := busca(g[v], w)
    si está? entonces g[v] := LISTA_INTERÉS_ORDENADA.borra_actual(g[v]) fsi
devuelve g

función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
var está? es bool; et es etiq fvar
    <g[v], está?> := busca(g[v], w)
    si está? entonces et := LISTA_INTERÉS_ORDENADA.actual(g[v]).et si no et := indef fsi
devuelve et

función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer s := añade(s, actual(g[w])) fpara todo
devuelve s

función pred (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
var w es vértice; está? es bool; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer
        <g[w], está?> := busca(g[w], v)
        si está? entonces s := añade(s, <w, actual(g[w]).et>) fsi
    fpara todo
devuelve s

{Función auxiliar busca(l, v) → <l', está?>: dada la lista de vértices l, busca el vértice v.
Devuelve un booleano con el resultado de la búsqueda; si vale cierto, el punto de
interés queda sobre v; si vale falso, queda sobre el elemento mayor de los mayores, o a
la derecha de todo, si v es mayor que el máximo de l}

función privada busca (l es lista_vértices_y_etiqs; v es vértice)
    devuelve <lista_vértices_y_etiqs, bool> es

var está, éxito es bool fvar
    l := principio(l); éxito := falso
    mientras ¬final?(l) ∧ ¬ éxito hacer
        si actual(l).v ≥ v entonces éxito := cierto si no l := avanza(l) fsi
    fmientras
    si éxito entonces está := (actual(l).v = v) si no está := falso fsi
devuelve <l, está?>

funiverso

```

Fig. 6.22: implementación de los grafos dirigidos etiquetados por listas de adyacencia (cont.).

Dado el coste constante de las operaciones sobre listas con el punto de interés como medio de referencia y el coste lineal de suprimir un elemento de una lista ordenada, y suponiendo de nuevo una buena representación para los conjuntos recorribles, la eficiencia temporal de la representación presenta las siguientes características:

- *crea* queda $\Theta(n)$.
- Las operaciones individuales sobre aristas dejan de ser constantes porque ahora es necesario buscar un elemento en la lista de sucesores de un nodo. El coste es, pues, $\Theta(k)$, siendo k el número estimado de aristas que saldrán de un nodo; este coste se convierte en $\Theta(n)$ en el caso peor.
- La operación *suc* queda $\Theta(k)$, que mejora en el caso medio la eficiencia de la implementación por matrices, porque los nodos sucesores están siempre calculados en la misma representación del grafo. No obstante, el caso peor sigue siendo $\Theta(n)$.
- Finalmente, *pred* queda $\Theta(a+n)$ en el caso peor; el factor a surge del examen de todas las aristas (si el vértice no tiene ningún predecesor), mientras que n se debe al hecho de que, si el grafo presenta menos aristas que nodos, el tiempo de examen del vector índice pasa a ser significativo. El caso mejor no bajará, pues, de $\Theta(n)$.

Es decir, excepto la creación (que normalmente no influye en el estudio de la eficiencia) y *suc*, el coste temporal parece favorecer la implementación por matriz. Ahora bien, como ya se comentó en el caso de las relaciones binarias, el buen comportamiento de *suc* hace que muchos de los algoritmos que veremos sean más eficientes con una implementación por listas, porque se basan en un recorrido de todas las aristas del grafo que se pueden obtener en $\Theta(a+n)$, frente al coste $\Theta(n^2)$ de las matrices, $0 \leq a \leq n^2 - n$, utilizando el bucle:

```

para todo v dentro de vértice hacer
    para todo w dentro de suc(g, v) hacer
        tratar arista (v, w)
    fpara todo
fpara todo
```

El coste espacial es claramente $\Theta(a+n)$ que, como mucho, es asintóticamente equivalente al espacio $\Theta(n^2)$ requerido por la representación matricial, pero que en grafos dispersos queda $\Theta(n)$. Ahora bien, es necesario tener en cuenta que el espacio para los encadenamientos puede no ser despreciable porque, generalmente, las etiquetas serán enteros y cada celda de las listas ocupará relativamente más que una posición de la matriz de adyacencia; en grafos densos, el coste real puede incluso contradecir los resultados asintóticos (v. ejercicio 6.5). Debe notarse también que, si los vértices no son un tipo por enumeración, no sólo debe sustituirse el vector índice por una tabla, sino que también es necesario sustituir el campo vértice por un apuntador a una entrada de la tabla en las listas, si se quiere evitar la redundancia de la información. Por ello, necesitaríamos una tabla parcialmente abierta. Estos comentarios también son válidos para la representación que estudiamos a continuación.

c) Implementación por multilistas de adyacencia

Ya se ha comentado la extrema ineficiencia de la función *pred* en el esquema de listas de adyacencia. Si se necesita que esta operación sea lo más rápida posible, se puede repetir la estrategia anterior, pero sustituyendo las listas de sucesores por listas de predecesores, lo que hace que *suc* sea ineficiente. Ahora bien, si se exige que *suc* sea también eficiente, deben asociarse dos listas a cada vértice, la de sus sucesores y la de sus predecesores. En realidad esta solución corresponde al concepto de multilista de grado dos usada para implementar las relaciones binarias entre dos conjuntos cualesquiera de elementos y por ello hablamos de *multilistas de adyacencia* (v. fig. 6.23). En el caso de los grafos, la relación binaria se establece entre elementos de un mismo conjunto, pero este hecho no afecta al esquema general.

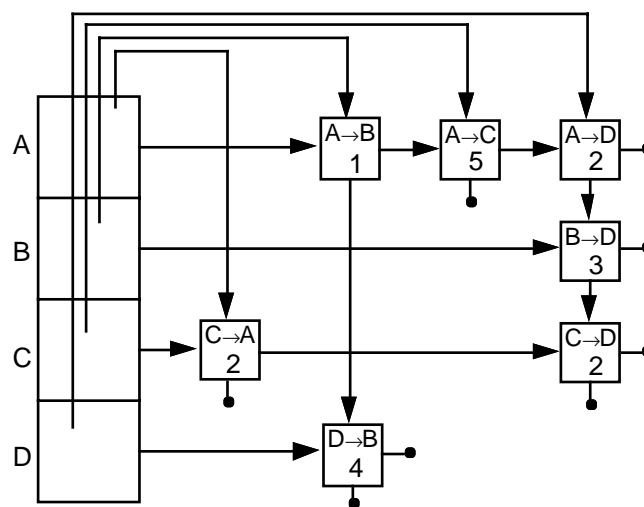


Fig. 6.23: implementación del grafo de la fig. 6.19 por multilistas de adyacencia.

En la fig. 6.24 se ofrece una implementación instanciando el TAD de las relaciones etiquetadas; notemos que, al contrario de la estrategia anterior, no se obliga a representar las multilistas de una manera concreta, porque las diversas implementaciones que de ellas existen ya se ocupan de optimizar espacio, y será el usuario del grafo quien escogerá la estrategia adecuada en su contexto (listas circulares, con identificadores en las celdas, etc.). El invariante establece, una vez más, la antirreflexividad de la relación. La eficiencia de las operaciones queda como en el caso anterior, excepto *pred*, que mejora por los mismos razonamientos que *suc*. Especialmente, el coste asintótico tampoco cambia, aunque debe considerarse que ahora el vector índice ocupa el doble y que también hay el doble de encadenamientos en las celdas (o más, según la representación de las multilistas).

```

universo DIGRAFO_ETIQ_MULTILISTAS(V es ELEM_ORDENADO, E es ELEM_ESP) es
  implementa DIGRAFO_ETIQ(V es ELEM_<_, E es ELEM_ESP)
  renombra V.elem por vértice, E.elem por etiq, E.esp por indef
  usa BOOL
  instancia RELACIÓN_ETIQUETADA(A, B son ELEM_<_, X es ELEM_ESP) donde
    A.elem, B.elem son vértice, X.esp es indef, ...
  tipo grafo es relación ftipo
  invariante (g es grafo):  $\forall v: v \in \text{vértice}: v \notin \text{fila}(g, v) \wedge v \notin \text{columna}(g, v)$ 
  función crea devuelve grafo es devuelve RELACIÓN_ETIQUETADA.crea
  función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
    si (v = w)  $\vee$  (et = indef) entonces error
    si no g := RELACIÓN_ETIQUETADA.inserta(g, v, w, et)
    fsi
  devuelve g
  función borra (g es grafo; v, w son vértice) devuelve grafo es
  devuelve RELACIÓN_ETIQUETADA.borra(g, v, w)
  función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
  devuelve RELACIÓN_ETIQUETADA.consulta(g, v, w)
  función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
  devuelve RELACIÓN_ETIQUETADA.fila(g, v)
  función pred (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
  devuelve RELACIÓN_ETIQUETADA.columna(g, v)
funiverso

```

Fig. 6.24: implementación de los digrafos etiquetados por multilistas de adyacencia.

d) Extensión al resto de modelos

Si el grafo es no dirigido, en la matriz de adyacencia no cambia nada en particular, tan sólo cabe destacar su simetría. En las listas y multilistas de adyacencia, puede optarse por repetir las aristas dos veces, o bien guardar la arista una vez solamente, en la lista de sucesores del nodo más pequeño de los dos según el orden establecido entre los vértices. Estas convenciones deben consignarse en el invariante de la representación.

Si el grafo es no etiquetado, el campo de etiqueta de la matriz se sustituye por un booleano que indica existencia o ausencia de arista, mientras que en las representaciones encadenadas simplemente desaparece y entonces el espacio relativo empleado en encadenamientos es todavía más considerable.

6.4 Recorridos de grafos

Como en todos los TAD que estudiamos, nos interesa generalmente tener versiones recorribles de los mismos. En el caso de los grafos, tenemos una particularidad ya comentada en la sección 6.1: nos puede interesar tanto recorrer los vértices que lo forman, como las aristas que contiene. En realidad, el segundo tipo de recorrido puede formularse en términos del primero: se obtienen todos los vértices y, para cada uno de ellos, se obtienen sus vértices sucesores, predecesores o adyacentes según sea el caso. Por lo tanto, nos centramos en el estudio del recorrido de los vértices de un grafo.

A veces no importa el orden de obtención de los nodos y es suficiente un bucle "para todo" sobre el tipo de los vértices que los obtenga según el orden definido en dicho tipo, a veces se necesita imponer una política de recorrido que depende de las aristas existentes, y entonces tenemos TAD recorribles ordenadamente. El segundo caso conduce al estudio de diversas estrategias de recorrido de grafos que se presentan en esta sección; como no dependen de si el grafo está etiquetado o no, consideramos el segundo caso. Por analogía con los árboles, distinguimos diferentes *recorridos en profundidad*, un *recorrido en anchura* y el llamado *recorrido por ordenación topológica*.

Para simplificar la signatura y como ya hemos hecho otras veces, definimos los recorridos como funciones *recorrido: grafo* → *lista_vértice*, siendo *lista_vértice* el resultado de instanciar las listas con punto de interés con el tipo de los vértices; la conversión de esta signatura a la propia de los TAD recorribles es inmediata. Una política diferente que podemos encontrar en diversas fuentes bibliográficas consiste en sustituir la lista por un vector indexado por vértices de modo que en cada posición residirá el ordinal de visita. Notemos que esta versión es un caso particular de devolver una lista y como tal se podría obtener efectuando las instancias oportunas. Ahora bien, debemos destacar que el algoritmo resultante permite eliminar el conjunto de vértices ya visitados, previa inicialización del vector a un valor especial; por otra parte, no es posible un recorrido posterior ordenado rápido del vector, porque es necesario ordenarlo antes.

Tanto en esta sección como en las siguientes nos centraremos en el estudio de los algoritmos y no en su encapsulamiento dentro de universos. Por ejemplo, escribiremos las pre y postcondiciones y los invariantes de los bucles especialmente interesantes, y supondremos que todas las instancias necesarias de conjuntos, colas, listas, etc., han sido formuladas previamente en alguna parte, explicitando las implementaciones que responden a los requerimientos de eficiencia pedidos. Por lo que respecta al encapsulamiento, tan sólo decir que hay tres opciones principales:

- Encerrar cada algoritmo individualmente dentro de un universo. El resultado sería un conjunto de módulos muy grande, seguramente difícil de gestionar y por ello poco útil.
- Encerrar todos los algoritmos referidos a un mismo modelo de grafo en un único

universo. Si bien facilita la gestión de una hipotética biblioteca de módulos reusables, hay que tener presente que diferentes algoritmos pueden exigir diferentes propiedades de partida sobre los grafos (que sean conexos, que no haya etiquetas negativas, etc.).

- Encerrar todos los algoritmos del mismo tipo y referidos a un mismo modelo de grafo en un único universo. Parece la opción más equilibrada, porque todos los algoritmos orientados a resolver una misma clase de problemas requieren propiedades muy similares o idénticas sobre el grafo de partida.

Todos los algoritmos usan conjuntos para almacenar los vértices visitados en cada momento porque, a diferencia de los árboles, puede accederse a un mismo nodo siguiendo diferentes caminos y deben evitarse las visitas reiteradas. Las operaciones sobre estos conjuntos serán la constante conjunto vacío, añadir un elemento y comprobar la pertenencia y, por ello, nos basaremos en la signatura presentada en la fig. 1.29, definiendo una operación más, \notin , como $\neg \in$. Por lo que respecta a la implementación, supondremos que las operaciones son de orden constante (por ejemplo, usando un vector de booleanos indexado por el tipo de los vértices).

6.4.1 Recorrido en profundidad

El *recorrido de búsqueda en profundidad* (ing., *depth-first search*; algunos autores añaden el calificativo "prioritaria") o, simplemente, *recorrido en profundidad*, popularizado por J.E. Hopcroft y R.E. Tarjan el año 1973 en "Efficient Algorithms for Graph Manipulation", *Communications ACM*, 16(6), es aplicable indistintamente a los grafos dirigidos y a los no dirigidos, considerando en el segundo caso cada arista no dirigida como un par de aristas dirigidas. El procedimiento es una generalización del recorrido preorden de un árbol: se comienza visitando un nodo cualquiera y, a continuación, se recorre en profundidad el componente conexo que "cuelga" de cada sucesor (es decir, se examinan los caminos hasta que se llega a nodos ya visitados o sin sucesores); si después de haber visitado todos los sucesores transitivos del primer nodo (es decir, él mismo, sus sucesores, los sucesores de sus sucesores, y así sucesivamente) todavía quedan nodos por visitar, se repite el proceso a partir de cualquiera de estos nodos no visitados. En el resto de la sección, a los sucesores transitivos de un nodo los llamaremos *descendientes* y, simétricamente, a los predecesores transitivos los llamaremos *antecesores*; la especificación de estos conceptos se propone en el ejercicio 6.1.

El recorrido en profundidad tiene diferentes aplicaciones. Por ejemplo, se puede usar para analizar la robustez de una red de computadores representada por un grafo no dirigido (v. [AHU83, pp. 243-245]). También se puede utilizar para examinar si un grafo dirigido presenta algún ciclo, antes de aplicar sobre él cualquier algoritmo que exija que sea acíclico.

La especificación del recorrido se estudia en el ejercicio 6.31. A la vista de la descripción dada queda claro que, en realidad, no hay un único recorrido en profundidad prioritaria de un grafo, sino un conjunto de recorridos, todos ellos igualmente válidos, y por ello no se especifica el procedimiento de construcción de una solución, sino un predicado que comprueba si una lista de vértices es un recorrido válido para un grafo; generalmente, esta indeterminación no causa ningún problema en la aplicación que necesita el recorrido. Al contrario que los árboles, el indeterminismo es una característica común a todas las estrategias de recorridos sobre grafos.

En la fig. 6.25 se presenta un algoritmo recursivo de recorrido en profundidad prioritaria, que usa un conjunto S para almacenar los vértices ya visitados, y un procedimiento auxiliar para recorrer recursivamente todos los descendientes de un nodo (incluido él mismo); su resultado es la lista de los vértices en orden de visita. Tal como establece el invariante, los elementos de la lista resultado y de S son siempre los mismos, pero el conjunto se mantiene para poder comprobar rápidamente si un vértice ya está en la solución. En el invariante se usa la operación *subgrafo*, cuya especificación se propone en el ejercicio 6.1. La transformación de este algoritmo en iterativo queda como ejercicio para el lector.

El recorrido en profundidad estándar se puede modificar adoptando justamente el enfoque simétrico, sin visitar un nodo hasta haber visitado todos sus descendientes, en lo que se puede considerar como la generalización del recorrido postorden de un árbol. Diversos textos denominan a este recorrido *recorrido en profundidad con numeración hacia atrás* o, simplemente, *recorrido en profundidad hacia atrás* (ing., *backward depth-first search*); por el mismo motivo, el recorrido en profundidad estudiado hasta ahora también se denomina *recorrido en profundidad hacia delante* (ing., *forward depth-first search*). En la fig. 6.26 se presenta un ejemplo para ilustrar la diferencia. Por lo que respecta al algoritmo de la fig. 6.25, simplemente sería necesario mover la inserción de un nodo a la lista tras el bucle.

El coste temporal del algoritmo depende exclusivamente de la implementación elegida para el grafo. Usando una representación por listas o multilistas de adyacencia obtenemos $\Theta(a+n)$, porque, dado un vértice, siempre se consultan las aristas que de él parten, aunque no se visite más que una vez; el factor n aparece por si el grafo tiene menos aristas que vértices. Si se elige la representación mediante matriz de adyacencia el coste es $\Theta(n^2)$, por culpa del tiempo lineal de la operación *suc*; es decir, si el grafo es disperso, la representación por listas es más eficiente por la rapidez en la obtención de todas las aristas del grafo. Por lo que respecta al espacio adicional empleado, es lineal sobre el número de nodos debido al conjunto.

```

{  $\mathcal{P} \equiv g$  es un grafo dirigido y no etiquetado }
función recorrido_en_profundidad (g es grafo) devuelve lista_vértices es
var S es cjt_vértices; v es vértice; l es lista_vértices fvar
  S :=  $\emptyset$ ; l := LISTA_INTERÉS.crea
  para todo v dentro de vértice hacer
    {  $I_0 \equiv$  es_recorrido_profundidad(l, subgrafo(g, S))  $\wedge$ 
       $\forall u: u \in \text{vértice}: (u \in l \Leftrightarrow u \in S) \wedge (u \in l \Rightarrow \forall w: w \in \text{descendientes}(g, u): w \in l)$  }
    si v  $\notin$  S entonces visita_componente(g, v, S, l) fsi
  fpara todo
devuelve l
{  $Q \equiv$  es_recorrido_profundidad(l, g) }      -- v. ejercicio 6.31

{  $\mathcal{P} \equiv v \notin S \wedge I_0$  }
acción privada visita_componente (ent g es grafo; ent v es vértice;
                                   ent/sal S es cjt_vértices; ent/sal l es lista_vértices) es
var w es vértice fvar
  S := S  $\cup$  {v}; l := LISTA_INTERÉS.inserta(l, v)
  {visita de todos los descendientes de v}
  para todo w dentro de suc(g, v) hacer
    {  $I \equiv$  es_recorrido_profundidad(l, subgrafo(g, S))  $\wedge$ 
       $\forall u: u \in \text{vértice}: (u \in l \Leftrightarrow u \in S) \wedge (u \in l \wedge u \neq v \Rightarrow \forall w: w \in \text{descendientes}(g, u): w \in l)$  }
    si w  $\notin$  S entonces visita_componente(g, w, S, l) fsi
  fpara todo
facción
{  $Q \equiv v \in S \wedge I_0$  }

```

Fig. 6.25: algoritmo del recorrido en profundidad prioritaria.

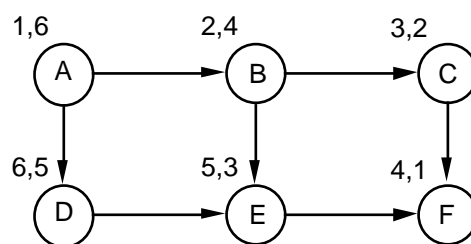


Fig. 6.26: numeración de los vértices de un grafo en un recorrido en profundidad iniciado en A (izquierda: hacia delante; derecha: hacia atrás) considerando el orden alfabético para resolver indeterminaciones.

6.4.2 Recorrido en anchura

El *recorrido de búsqueda en anchura* o *expansión* (ing., *breadth-first search*) o, simplemente, *recorrido en anchura* es otra estrategia aplicable indistintamente al caso de grafos dirigidos y no dirigidos, que generaliza el concepto de recorrido por niveles de un árbol: después de visitar un vértice se visitan los sucesores, después los sucesores de los sucesores, después los sucesores de los sucesores de los sucesores, y así reiteradamente. Si después de visitar todos los descendientes del primer nodo todavía quedan más nodos por visitar, se repite el proceso. La especificación de este recorrido se propone también en el ejercicio 6.31.

Una aplicación típica del recorrido en anchura es la resolución de problemas de planificación, donde se dispone de un escenario que se puede modelizar mediante un estado. El estado representa la configuración de un conjunto de elementos de manera que, si cambia su disposición o número, cambia el estado; los cambios de estado atómicos (es decir, que no se pueden considerar como la composición de otros cambios más simples) se denominan transiciones. En estas condiciones, una cuestión habitual consiste en determinar la secuencia más corta de transiciones que lleva de un estado inicial a un estado final, y la pregunta se puede contestar en términos de un recorrido en expansión de un grafo. De hecho, en 1959 E.F. Moore introdujo el recorrido en anchura en este contexto, como una ayuda para atravesar un laberinto ("The Shortest Path through a Maze", en *Proceedings of the International Symposium on the Theory of Switching*, Harvard University Press). En el apartado 7.1.3 se resuelve este problema para una situación concreta.

El algoritmo sobre grafos dirigidos (v. fig. 6.27) utiliza una cola para poder aplicar la estrategia expuesta. Notemos que el procedimiento principal es idéntico al del recorrido en profundidad prioritaria y que *visita_componente* es similar al anterior usando una cola de vértices¹⁰. Asimismo observemos que hay que marcar un vértice (es decir, insertarlo en el conjunto *S*) antes de meterlo en la cola, para no encolarlo más de una vez. El invariante del procedimiento auxiliar obliga a que todos los elementos entre *v* y los que están en la cola (pendientes de visita) ya hayan sido visitados, y a que todos los elementos ya visitados tengan sus sucesores, o bien visitados, o bien en la cola para ser visitados de inmediato.

Los resultados sobre la eficiencia temporal son idénticos al caso del recorrido en profundidad.

¹⁰ De hecho, la transformación en iterativo del procedimiento *visita_componente* del recorrido en profundidad resulta en el mismo algoritmo sustituyendo la cola por una pila.

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido no etiquetado}
función recorrido_en_anchura (g es grafo) devuelve lista_vértices es
var S es cjt_vértices; v es vértice; l es lista_vértices fvar
  S :=  $\emptyset$ ; l := LISTA_INTERÉS.crea
  para todo v dentro de vértice hacer
    { $I_0 \equiv$  es_recorrido_anchura(l, subgrafo(g, S))  $\wedge$ 
       $\forall u: u \in \text{vértice}: (u \in l \Leftrightarrow u \in S) \wedge (u \in l \Rightarrow \forall w: w \in \text{descendientes}(g, u): w \in l)$ }
    si v  $\notin$  S entonces visita_componente(g, v, S, l) fsi
  fpara todo
devuelve l
{ $Q \equiv$  es_recorrido_anchura(l, g)} -- v. ejercicio 6.31

{ $\mathcal{P} \equiv v \notin S \wedge I_0$ }
acción privada visita_componente (ent g es grafo; ent v es vértice;
  ent/sal S es cjt_vértices; ent/sal l es lista_vértices) es
var c es cola_vértices; u, w son vértice fvar
  S := S  $\cup$  {v}; c := COLA.encola(COLA.crea, v)
  mientras  $\neg$ COLA.vacía?(c) hacer
    { $I \equiv$  es_recorrido_anchura(l, subgrafo(g, S))  $\wedge$  ( $\forall u: u \in \text{vértice}: u \in l \Leftrightarrow u \in S$ )  $\wedge$ 
      ( $\forall u: u \in c: \forall w: w \in \text{ascendentes}(g, u) - \{u\}: w \in \text{descendientes}(g, v) \Rightarrow w \in l$ )  $\wedge$ 
      ( $\forall u: u \in l: \forall w: w \in \text{suc}(g, u): w \in l \vee w \in c$ ) }
    w := COLA.cabeza(c); c := COLA.desencola(c)
    l := LISTA_INTERÉS.inserta(l, w)
    para todo u dentro de suc(g, v) hacer
      si u  $\notin$  S entonces S := S  $\cup$  {u}; c := COLA.encola(c, u) fsi
    fpara todo
  fmientras
facció
{ $Q \equiv v \in S \wedge I_0$ }

```

Fig. 6.27: algoritmo de recorrido en anchura de un grafo dirigido.

6.4.3 Recorrido en ordenación topológica

La *ordenación topológica* (ing., *topological sort*) es un recorrido solamente aplicable a grafos dirigidos acíclicos, que cumple la propiedad de que un vértice sólo se visita si han sido visitados todos sus predecesores dentro del grafo. De esta manera, las aristas definen una restricción más fuerte sobre el orden de visita de los vértices.

La aplicación más habitual del recorrido en ordenación topológica aparece cuando los vértices del grafo representan tareas y las aristas relaciones temporales entre ellas. Por

ejemplo, en el plan de estudios de una carrera universitaria cualquiera, puede haber asignaturas con la restricción de que sólo se puedan cursar si previamente se han aprobado otras, llamadas *prerrequisitos*. Este plan de estudios forma un grafo dirigido, donde los nodos son las asignaturas y existe una arista de la asignatura *A* a la asignatura *B*, si y sólo si *A* es *prerrequisito* de *B* (v. fig. 6.28). Notemos, además, que el grafo ha de ser *acíclico*: si *A* es *prerrequisito* de *B* (directa o indirectamente) no debe existir ninguna secuencia de *prerrequisitos* que permitan deducir que *B* es *prerrequisito* de *A* (directa o indirectamente).

Cuando un alumno se matricula en este plan de estudios, ha de cursar diversas asignaturas (es decir, ha de recorrer parte del grafo del plan de estudios) respetando las reglas:

- Al comenzar, sólo puede cursar las asignaturas (es decir, visitar los nodos) que no tengan ningún *prerrequisito* (es decir, que no tengan ningún *predecesor* en el grafo).
- Sólo se puede cursar una asignatura si han sido cursados todos sus *prerrequisitos* previamente (es decir, si han sido visitados todos sus *predecesores*).

Así, pues, un estudiante se licencia cuando ha visitado un número suficiente de nodos respetando estas reglas, las cuales conducen a un recorrido en ordenación topológica.

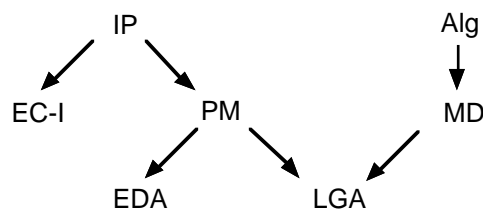


Fig. 6.28: parte del plan de estudios de la Ingeniería en Informática impartida en la Facultat d'Informàtica de Barcelona de la Universitat Politècnica de Catalunya.

La descripción del funcionamiento del algoritmo se presenta en la fig. 6.29: se van escogiendo los vértices sin *predecesores*, que son los que se pueden visitar y, a medida que se incorporan a la solución se borran las aristas que salen de ellos; la función *nodos* del invariante devuelve todos los elementos pertenecientes a una lista. Los recorridos obtenidos de esta forma responden a la especificación que se propone en el ejercicio 6.31.

Obviamente la implementación directa de esta descripción es costosa, ya que hay que buscar vértices sin *predecesores* repetidas veces. Una alternativa consiste en calcular previamente cuántos *predecesores* tiene cada vértice, almacenar el resultado en un vector y actualizarlo siempre que se incorpore un nuevo vértice a la solución. En la fig. 6.30 se presenta una implementación de esta última opción, que usa un vector de vértices *núm_pred*, que asocia el número de *predecesores* que todavía no están en la solución a

cada vértice, y un conjunto *ceros* donde se guardan todos los vértices que se pueden incorporar en el paso siguiente del algoritmo. La inicialización de la estructura, ciertamente rebuscada a simple vista, está diseñada para ser asintóticamente óptima para cualquier representación del grafo; con este objetivo se evita el uso de la operación *etiqueta*, que exige búsquedas lineales en caso de listas¹¹. Nótese el uso de una operación sobre conjuntos, *escoger_uno_cualquiera*, que devuelve un elemento cualquiera del conjunto y lo borra, y que debe ser ofrecida por la especificación del TAD de los conjuntos o bien por un enriquecimiento.

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido no etiquetado y acíclico}
función ordenación_topológica ( $g$  es grafo) devuelve lista_vértices es
var  $I$  es lista_vértices;  $v, w$  son vértices fvar
     $I := \text{LISTA\_INTERÉS.crea}$ 
    mientras queden nodos por visitar hacer
        { $I \equiv \text{es\_ordenación\_topológica}(I, \text{subgrafo}(g, \text{nodos}(I)))$  }
        escoger  $v \notin I$  tal que todos sus predecesores estén en  $I$ 
         $I := \text{LISTA\_INTERÉS.inserta}(I, v)$ 
    fmientras
devuelve  $I$ 
{ $Q \equiv \text{es\_ordenación\_topológica}(I, g)$ }      -- v. ejercicio 6.31

```

Fig. 6.29: presentación del algoritmo de ordenación topológica.

Para calcular el coste temporal del algoritmo se suman los costes de la inicialización de las estructuras y del bucle principal. En la inicialización, el primer bucle queda $\Theta(n)$, mientras que el segundo exige un examen de todas las aristas del grafo y queda, pues, $\Theta(a+n)$ en el caso de listas de adyacencia y $\Theta(n^2)$ en caso de matriz de adyacencia. En lo que respecta al bucle principal, un examen poco cuidadoso podría llevar a decir que es $\Theta(n^2)$, dado que el bucle se ejecuta n veces (en cada paso se añade un vértice a la solución) y en cada iteración se obtienen los sucesores del nodo escogido, operación que es $\Theta(n)$ en el caso peor (supongamos que *escoger_uno_cualquiera* es de orden constante). Ahora bien, notemos que la iteración interna exige, a lo largo del algoritmo, el examen de todas las aristas que componen el grafo con un tratamiento constante de cada una, lo que, en realidad, es $\Theta(a)$ y no $\Theta(n^2)$ si el grafo está implementado por listas de adyacencia. Como la operación *escoger_uno_cualquiera* se ejecuta un total de $n-1$ veces y es $\Theta(1)$, se puede concluir que la eficiencia temporal del algoritmo es, una vez más, $\Theta(a+n)$ con listas de adyacencia y $\Theta(n^2)$ con matriz de adyacencia y, por ello, la representación por listas de adyacencia es preferible si el grafo es disperso, mientras que si es denso la implementación no afecta a la eficiencia.

¹¹ Otra opción sería disponer de dos versiones del algoritmo en dos universos diferentes, una para implementación mediante matriz y otra para listas o multilistas. No obstante, preferimos esta opción para simplificar el uso del algoritmo.

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido no etiquetado y acíclico12}
función ordenación_topológica ( $g$  es grafo) devuelve lista_vértices es
var  $l$  es lista_vértices;  $v, w$  son vértice
    ceros es cjt_vértices; núm_pred es vector [vértice] de nat
fvar
    {inicialización de las estructuras en dos pasos}
    ceros :=  $\emptyset$ 
    para todo  $v$  dentro de vértice hacer
        núm_pred[ $v$ ] := 0; ceros := ceros  $\cup$  { $v$ }
    fpara todo
    para todo  $v$  dentro de vértice hacer
        para todo  $w$  dentro de suc( $g, v$ ) hacer
            núm_pred[ $w$ ] := núm_pred[ $w$ ] + 1; ceros := ceros - { $w$ }
        fpara todo
    fpara todo
    {a continuación se visitan los vértices del grafo}
     $l$  := LISTA_INTERÉS.crea
    mientras ceros  $\neq \emptyset$  hacer
        { $I \equiv$  es_ordenación_topológica( $l$ , subgrafo( $g$ , nodos( $l$ )))  $\wedge$ 
          $\forall w: w \in \text{vértice}: \text{núm\_pred}[w] = ||\{ \langle u, w \rangle: \langle u, w \rangle \in \text{dom}(g): u \notin l \} || \wedge$ 
          $w \in \text{ceros} \Rightarrow \text{núm\_pred}[w] = 0$ }
        <ceros, v> := escoger_uno_cualquiera(ceros)
         $l$  := LISTA_INTERÉS.inserta( $l$ ,  $v$ )
        para todo  $w$  dentro de suc( $g, v$ ) hacer
            núm_pred[ $w$ ] := núm_pred[ $w$ ] - 1
            si núm_pred[ $w$ ] = 0 entonces ceros := ceros  $\cup$  { $w$ } fsi
        fpara todo
    fmientras
devuelve  $l$ 
{ $Q \equiv$  es_ordenación_topológica( $l, g$ )}
```

Fig. 6.30: implementación del algoritmo de ordenación topológica.

¹² Como alternativa, se podría controlar la condición de aciclicidad del grafo en el propio algoritmo y entonces desaparecería de la precondition.

6.5 Búsqueda de caminos mínimos

En esta sección nos ocupamos de la búsqueda de caminos minimales dentro de grafos etiquetados no negativamente que sean lo más *cortos* posible, donde la longitud de un camino, también conocida como *coste*, se define como la suma de las etiquetas de las aristas que lo componen. Para abreviar, hablaremos de "suma de aristas" con significado "suma de las etiquetas de las aristas" y, por ello, hablaremos de "caminos mínimos" en vez de "caminos minimales". Concretamente, estudiamos un algoritmo para encontrar la distancia mínima de un vértice al resto y otro para encontrar la distancia mínima entre todo par de vértices.

Dada la funcionalidad requerida, es necesario formular algunos requisitos adicionales sobre el dominio de las etiquetas, que se traducen en nuevos parámetros formales de los universos que encapsulen estos algoritmos (v. fig. 6.31). Concretamente, consideramos E como un dominio ordenado, donde $<$ es la operación de ordenación y $=$ la de igualdad; para abreviar, usaremos también la operación \leq , definible a partir de éstas. También se precisa de una operación conmutativa $+$ para sumar dos etiquetas, que tiene una constante 0 , a la cual denominaremos *etiqueta de coste nulo*, como elemento neutro. Por último, nos interesa que el valor indefinido *esp* represente la etiqueta más grande posible, por motivos que quedarán claros a lo largo de la sección; para mayor claridad, en los algoritmos que vienen a continuación supondremos que *esp* se renombra por ∞ . Las etiquetas negativas no forman parte del género, tal como establece la última propiedad del universo.

universo ELEM_ESP_<_=_+_ caracteriza

usa BOOL

tipo elem

ops 0, esp: \rightarrow elem

$_<_, _<=_, _==_, _>=_, _>_:$ elem elem \rightarrow bool

$_+_:$ elem elem \rightarrow elem

ecns ...la igualdad y desigualdad se especifican de la manera habitual

$[x \neq \text{esp}] \Rightarrow x < \text{esp} = \text{cierto}$ {esp es cota superior}

$x < x = \text{falso}$ {estudio de la reflexividad...}

$((x < y) \Rightarrow (y < x)) = \text{falso}$ {...de la simetría...}

$((x < y \wedge y < z) \Rightarrow (x < z)) = \text{cierto}$ {...y de la transitividad de $<$ }

$x \leq y = (x < y) \vee (x = y)$

$x + y = y + x$ {conmutatividad}

$x + 0 = x; x + \text{esp} = \text{esp}$ {elementos neutro e idempotente}

$(x+y < x) = \text{falso}$ {etiquetas no negativas}

funiverso

Fig. 6.31: nuevos requerimientos sobre el dominio de las etiquetas.

6.5.1 Camino más corto de un nodo al resto

Dado un grafo $g \in \{f: V \times V \rightarrow E\}$ con etiquetas no negativas, se trata de calcular el coste del camino mínimo desde un vértice dado al resto (ing., *single-source shortest paths*)¹³. La utilidad de un procedimiento que solucione esta cuestión está clara; el caso más habitual es que el grafo represente una distribución geográfica, donde las aristas den el coste (en precio, en distancia o similares) de la conexión entre dos lugares y sea necesario averiguar el camino más corto para llegar a un punto partiendo de otro (es decir, determinar la secuencia de aristas para llegar a un nodo a partir del otro con un coste mínimo).

La solución más eficiente a este problema es el denominado *algoritmo de Dijkstra*, en honor a su creador, E.W. Dijkstra. Formulado en 1959 en "A note on two problems in connexion with graphs", *Numerical Mathematics*, 1, pp. 269-271, sobre grafos dirigidos (su extensión al caso no dirigido es inmediata y queda como ejercicio para el lector), el algoritmo de Dijkstra (v. fig. 6.22) es un algoritmo voraz¹⁴ que genera uno a uno los caminos de un nodo v al resto por orden creciente de longitud. Usa un conjunto S de vértices donde, a cada paso del algoritmo, se guardan los nodos para los que ya se sabe el camino mínimo y devuelve un vector indexado por vértices, de manera que en cada posición w se guarda el coste del camino mínimo que conecta v con w . Cada vez que se incorpora un nodo a la solución, se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él. Por convención, se supone que el camino mínimo de un nodo a sí mismo tiene coste nulo. Un valor infinito en la posición w del vector indica que no hay ningún camino entre v y w .

El invariante asegura que, en cada momento del algoritmo, el vector contiene caminos mínimos formados íntegramente por nodos de S (excepto el último nodo), y que los nodos de S corresponden a los caminos mínimos más cortos calculados hasta el momento. A tal efecto, utiliza diversas funciones: *caminos*, que da el conjunto de caminos que se pueden encontrar dentro del grafo entre dos nodos; *coste*, que da el coste de un camino; y *máximo* y *mínimo*, que dan los elementos mayor y menor de un conjunto, respectivamente. La especificación de la primera se propone en el ejercicio 6.1; la especificación de las otras es sencilla y queda como ejercicio para el lector. Abusando de la notación, se usa el operador de intersección entre un conjunto y un camino con el significado intuitivo.

La especificación del algoritmo se propone en el ejercicio 6.32, donde se generaliza el mismo, pues el resultado es una tabla de vértices y etiquetas; la versión de la fig. 6.32 es una particularización de este caso, donde la tabla se implementa con un vector porque los vértices son un tipo que permite acceder directamente al mismo; el caso general se propone en el mismo ejercicio. La fig. 6.33 muestra su funcionamiento.

¹³ Se podría pensar en el caso particular de encontrar el camino más corto entre dos vértices, pero el algoritmo es igual de costoso que éste más general.

¹⁴ Los algoritmos *voraces* (ing., *greedy*) son una familia de algoritmos que se estructuran como un bucle que, en cada paso, calcula una parte ya definitiva de la solución; para estudiar en profundidad sus características puede consultarse, por ejemplo, [BrB97].

$\{P \equiv g \text{ es un grafo dirigido etiquetado no negativamente}\}$
función Dijkstra (g es grafo; v es vértice) devuelve vector [vértice] de eti es
var S es cjt_vértices; D es vector [vértice] de eti fvar
 $\forall w: w \in \text{vértice}: D[w] := \text{etiqueta}(g, v, w)$
 $D[v] := 0; S := \{v\}$
mientras S no contenga todos los vértices hacer
 $\{I \equiv \forall w: w \in \text{vértice}: D[w] = \text{mínimo}\{\text{coste}(C) / C \in \text{caminos}(g, v, w) \wedge C \cap (S \cup \{w\}) = C\}$
 $\wedge \neg (\exists w: w \notin S: D[w] < \text{máximo}\{D[u] / u \in S\})\}$
 elegir $w \notin S$ tal que $D[w]$ es mínimo; $S := S \cup \{w\}$
 $\forall u: u \notin S$: actualizar distancia mínima comprobando si por w hay un atajo
fmientras
devuelve D
 $\{Q \equiv D = \text{camino_mínimo}(g, v)\}$ -- v. ejercicio 6.32

Fig. 6.32: descripción del algoritmo de Dijkstra.

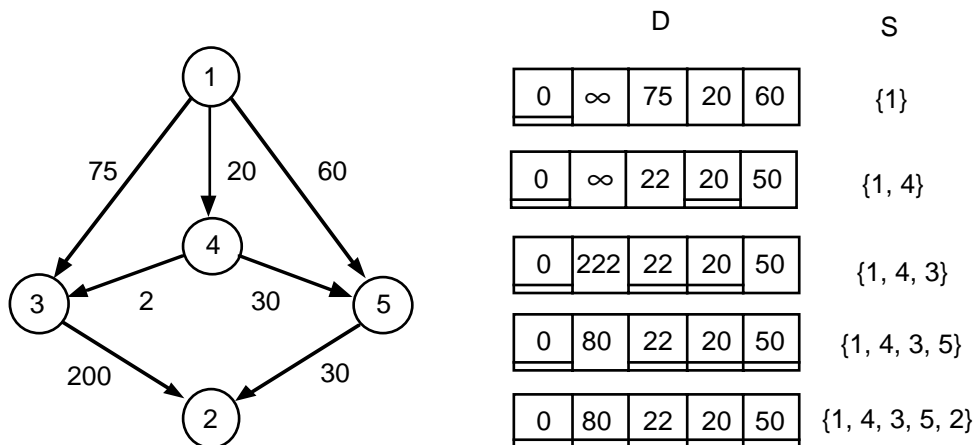


Fig. 6.33: ejemplo de funcionamiento del algoritmo de Dijkstra, donde 1 es el nodo inicial.

En la fig. 6.34 se detalla una posible codificación del algoritmo. Notemos que el uso del valor indefinido de las etiquetas como la etiqueta más grande posible permite denotar la inexistencia de caminos, porque cualquier camino entre dos nodos es más pequeño que ∞ . Asimismo, notemos que el algoritmo no trabaja con el conjunto S sino con su complementario, T , que debe presentar operaciones de recorrido para aplicarle un bucle "para todo". El número n de vértices es un parámetro formal más, definido en *ELEM_ORDENADO*. El bucle principal se ejecuta tan sólo $n-2$ veces, porque el último camino queda calculado después del último paso (no quedan vértices para hallar atajos).

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido etiquetado no negativamente}
función Dijkstra ( $g$  es grafo;  $v$  es vértice) devuelve vector [vértice] de etiq es
var  $T$  es cjt_recorrible_vértices;  $D$  es vector [vértice] de etiq;  $u, w$  son vértices fvar
 $T := \emptyset$ 
para todo  $w$  dentro de vértice hacer
     $D[w] := \text{etiqueta}(g, v, w); T := T \cup \{w\}$ 
fpara todo
 $D[v] := \text{ETIQUETA}.0; T := T - \{v\}$ 
hacer  $n-2$  veces {quedan  $n-1$  caminos por determinar}
    { $I \equiv \forall w: w \in \text{vértice}: D[w] = \text{mínimo}(\{\text{coste}(C) / C \in \text{caminos}(g, v, w) \wedge (C - \{w\}) \cap T = \emptyset\})$ 
     $\wedge \neg (\exists w: w \in T: D[w] < \text{máximo}(\{D[u] / u \notin T\})$ 
    {selección del mínimo  $w: w \in T \wedge (\forall u: u \in T: D[w] \leq D[u])$ }
     $\text{val} := \text{ETIQUETA}.\infty$ 
    para todo  $u$  dentro de  $T$  hacer
        si  $D[u] \leq \text{val}$  entonces  $w := u; \text{val} := D[u]$  fsi
        {como mínimo, siempre hay un nodo que cumple la condición}
    fpara todo
    {se marca  $w$  como vértice tractado}
     $T := T - \{w\}$ 
    {se recalculan las nuevas distancias mínimas}
    para todo  $u$  dentro de  $T$  hacer
        si  $D[w] + \text{etiqueta}(g, w, u) < D[u]$  entonces  $D[u] := D[w] + \text{etiqueta}(g, w, u)$  fsi
    fpara todo
fhacer
devuelve  $D$ 
{ $Q \equiv D = \text{caminos\_mínimos}(g, v)$ }

```

Fig. 6.34: una codificación posible del algoritmo de Dijkstra.

Se analiza a continuación el tiempo de ejecución, suponiendo que las operaciones sobre conjuntos están implementadas en tiempo constante, excepto la creación (por ejemplo, y aprovechando que los vértices son un tipo por enumeración, mediante un vector de booleanos). Distinguimos cuatro partes en el algoritmo: inicialización, selección del mínimo, marcaje de vértices y recálculo de las distancias mínimas. Si la representación es mediante matriz de adyacencia, la ejecución de *etiqueta* es constante y se obtiene:

- Inicialización: creación del conjunto y ejecución n veces de diversas operaciones constantes; queda $\Theta(n)$.
- Selección: las instrucciones del interior del bucle de selección son $\Theta(1)$. Por lo que respecta al número de ejecuciones del bucle, en la primera vuelta se consultan $n-1$ vértices, en la segunda $n-2$, etc., ya que la dimensión de T decrece en una unidad en

cada paso; es decir, a lo largo de las $n-2$ ejecuciones del bucle, sus instrucciones se ejecutarán $n(n-1)/2 - 1$ veces y, así, la eficiencia total del paso de selección será $\Theta(n^2)$.

- Marcaje: n supresiones a lo largo del algoritmo conducen a $\Theta(n)$.
- Recálculo de las distancias mínimas: queda $\Theta(n^2)$ por los mismos razonamientos que el paso de selección.

El coste total del algoritmo es la suma de las cuatro etapas, es decir, $\Theta(n^2)$.

En la representación por listas o multilista de adyacencias, lo único que parece cambiar es que la operación *etiqueta* deja de tener un coste constante. Ahora bien, un análisis más exhaustivo demuestra que en realidad la sentencia "si ... fsi" del paso de recálculo sólo se tiene que ejecutar a veces a lo largo del algoritmo, si se sustituye el bucle sobre los vértices de T por otro bucle sobre los vértices del conjunto de sucesores de w (evitando el uso reiterado de *etiq*). Como siempre se cumple que $a < n^2$, esta observación puede ser significativa, ya que, si también se consiguiera rebajar el coste del paso de selección, el algoritmo de Dijkstra para grafos no densos representados con estructuras encadenadas sería más eficiente que la representación mediante matriz.

Para alcanzar este objetivo basta con organizar astutamente el conjunto de vértices todavía no tratados, teniendo en cuenta que se selecciona el elemento mínimo. Por ejemplo, se puede sustituir el conjunto por una cola prioritaria, siendo sus elementos pares $\langle w, et \rangle$ de vértice (todavía no tratado) y etiqueta, que juega el papel de prioridad, con significado "la distancia mínima de v a w es et " (et sería el valor $D[w]$ de la fig. 6.32); las operaciones de obtención y supresión del mínimo encajan perfectamente dentro del algoritmo. Destacamos que necesitamos una versión que permitirá la existencia de elementos con idéntica prioridad. Ahora bien, el recálculo de las distancias del último paso puede exigir cambiar la prioridad de un elemento cualquiera de la cola. Por este motivo se define una nueva modalidad de cola con prioridad que permite el acceso directo a cualquier elemento y que, además de las operaciones habituales, incorpora tres más: la primera, para cambiar la etiqueta asociada a un vértice cualquiera por una más pequeña, reorganizando la cola si es necesario; otra, para obtener el valor asociado a un vértice (que valdrá ∞ si el vértice no está en la cola); y una tercera, para comprobar la presencia en la cola de un vértice dado. En la fig. 6.36 se muestra la signatura resultante de este tipo, aplicada al contexto del algoritmo, junto con su coste asintótico¹⁵, y en la fig. 6.35 se muestra un ejemplo de su funcionamiento suponiendo que, para mayor eficiencia, la cola se implementa con un montículo junto con un vector indexado por vértices, de forma que cada posición apunte al nodo del montículo que contenga la etiqueta asociada al vértice; la especificación e implementación del tipo quedan como ejercicio para el lector.

¹⁵ En esta signatura se observa la ausencia de la función *organiza* de transformación de un vector en una cola (v. apartado 4.4.2), más eficiente que la creación de la cola por inserciones individuales. Dicha ausencia se debe a que el algoritmo queda más claro sin que su coste asintótico se vea afectado. Si se considera conveniente, el algoritmo puede adaptarse fácilmente a esta modificación.

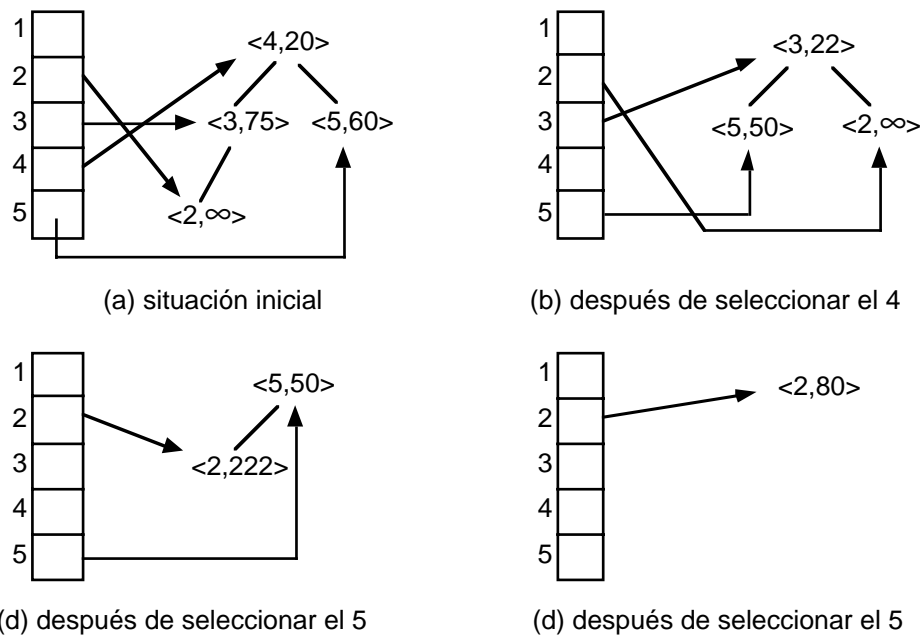


Fig. 6.35: evolución de la cola prioritaria aplicando el algoritmo sobre el grafo de la fig. 6.33.

crea: \rightarrow colapr_aristas; $\Theta(1)$
 inserta: colapr_aristas vértice etiq \rightarrow colapr_aristas; $\Theta(\log n)$
 menor: colapr_aristas \rightarrow vértice_y_etiq; $\Theta(1)$
 borra: colapr_aristas \rightarrow colapr_aristas; $\Theta(\log n)$
 sustituye: colapr_aristas vértice etiq \rightarrow colapr_aristas; $\Theta(\log n)$
 valor: colapr_aristas vértice \rightarrow etiq; $\Theta(1)$
 está?: colapr_aristas vértice \rightarrow bool; $\Theta(1)$
 vacía?: colapr_aristas \rightarrow bool; $\Theta(1)$

Fig. 6.36: signatura para las colas prioritarias extendidas con las operaciones necesarias.

El algoritmo resultante se muestra en la fig. 6.37. La inicialización requiere dos bucles para evitar el uso de la operación *etiqueta*, pues esta operación es ineficiente en la implementación con lista de adyacencia. Se podría modificar el algoritmo para que las etiquetas infinitas no ocupasen espacio en el montículo y mejorar así el tiempo de ejecución de las operaciones que lo actualizan. El análisis de la eficiencia temporal da como resultado:

- La inicialización es $\Theta(\sum k: 1 \leq k \leq n-1: \log k) + \Theta(n \log n) = \Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$, dado que el paso k -ésimo del primer bucle ejecuta una inserción en una cola de tamaño k y todos los pasos del segundo bucle ($n-1$ como máximo) modifican una cola de tamaño n .

- Las $n-1$ selecciones del mínimo son constantes, y su supresión, logarítmica sobre n , y queda, pues, $\Theta(n) + \Theta(\sum k: 1 \leq k \leq n-1: \log k) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$.
- El bucle interno examina todas las aristas del grafo a lo largo del algoritmo y, en el caso peor, efectúa una sustitución por arista, de coste $\Theta(\log n)$, y así queda $\Theta(a \log n)$.

En definitiva, el coste temporal es $\Theta((a+n) \log n)$, mejor que la versión anterior si el grafo es disperso; no obstante, si el grafo es denso, el algoritmo original es más eficiente. El espacio adicional continua siendo asintóticamente lineal sobre n , aunque la estructura es más voluminosa en términos no asintóticos.

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido etiquetado no negativamente}
función Dijkstra ( $g$  es grafo;  $v$  es vértice) devuelve vector [ $v$  vértice] de etiq es
var  $D$  es vector [ $v$  vértice] de etiq;  $A$  es colapr_aristas;  $u, w$  son vértices;  $et, val$  son etiq fvar
    {creación de la cola con todas las aristas  $\langle v, w \rangle$ }
     $A := COLA\_EXTENDIDA.crea$ 
    para todo  $w$  dentro de vértice hacer  $A := COLA\_EXTENDIDA.inserta(A, w, \infty)$  fpara todo
    para todo  $\langle w, et \rangle$  dentro de  $suc(g, v)$  hacer
         $A := COLA\_EXTENDIDA.modif(A, w, et)$ 
    fpara todo
    mientras  $\neg COLAPR\_EXTENDIDA.vacia?(A)$  hacer
        { $I \equiv \forall w: w \in \text{vértice}: x = \text{mínimo}(\{C: C \in \text{caminos}(g, v, w) \wedge$ 
             $\forall u: u \in C: u \neq w \Rightarrow \neg \text{está?}(A, u): \text{coste}(C)\})$ 
             $\wedge \text{menor}(A).et \geq \text{máximo}(\{u: u \in C \wedge \neg \text{está?}(A, u): D[u]\})$ 
            definiendo  $x: \text{está?}(A, w) \Rightarrow x = \text{valor}(A, w) \wedge \neg \text{está?}(A, w) \Rightarrow x = D[w]$  }
         $\langle w, val \rangle := COLAPR\_EXTENDIDA.menor(A)$  {selección de la arista mínima}
         $D[w] := val; A := COLAPR\_EXTENDIDA.borra(A)$  {marcaje de  $w$  como tratado}
        {recálculo de las nuevas distancias mínimas}
        para todo  $\langle u, et \rangle$  dentro de  $suc(g, w)$  hacer
            si  $COLAPR\_EXTENDIDA.está?(A, u)$  entonces
                si  $val+et < COLAPR\_EXTENDIDA.valor(A, u)$  entonces
                     $A := COLAPR\_EXTENDIDA.sustituye(A, u, val + et)$ 
                fsi
            fsi
        fpara todo
    fmientras
     $D[v] := ETIQ.0$ 
devuelve  $D$ 
{ $Q \equiv D = \text{caminos\_mínimos}(g, v)$ }

```

Fig. 6.37: el algoritmo de Dijkstra usando colas prioritarias.

La eficiencia temporal se puede mejorar aún más si estudiamos el funcionamiento del montículo. Destaquemos que el algoritmo efectúa exactamente $n-1$ hundimientos de elementos y, por otro lado, hasta un máximo de a flotamientos de manera que, en el caso general, hay más flotamientos que hundimientos. Estudiando estos algoritmos (v. apartado 4.4.2), se puede observar que un hundimiento requiere un número de comparaciones que depende tanto de la aridez del árbol como del número de niveles que tiene, mientras que un flotamiento depende tan sólo del número de niveles: cuantos menos haya, más rápido subirá el elemento hacia arriba. Por tanto, puede organizarse el montículo en un árbol k -ario y no binario, de manera que la altura del árbol se reduce. En [BrB87, p. 243] se propone demostrar que, con esta estrategia, el coste del algoritmo queda para grafos densos $\Theta(n^2)$ y, para otros tipos, $\Theta(a \log n)$, de manera que la versión resultante es la mejor para cualquier grafo representado por listas de adyacencia, independientemente de su número de aristas¹⁶.

Notemos que las diferentes versiones del algoritmo de Dijkstra presentadas aquí no devuelven la secuencia de nodos que forman el camino mínimo. La modificación es sencilla y queda como ejercicio para el lector: notemos que, si el camino mínimo entre v y w pasa por un vértice intermedio u , el camino mínimo entre v y u es un prefijo del camino mínimo entre v y w , de manera que basta con devolver un vector C , tal que $C[w]$ contenga el nodo anterior en el camino mínimo de v a w (que será v si está directamente unido al nodo de partida, o si no hay camino entre v y w). Este vector tendrá que ser actualizado al encontrarse un atajo en el camino (en realidad, el vector representa un árbol de expansión -v. sección 6.6- con apuntadores al padre, donde v es la raíz). Es necesario también diseñar una nueva acción para recuperar el camino a un nodo dado, que tendría como parámetro C .

6.4.2 Camino más corto entre todo par de nodos

Se trata ahora de determinar el coste del camino más corto entre todo par de vértices de un grafo etiquetado (ing., *all-pairs shortest paths*); de esta manera, se puede generalizar la situación estudiada en el apartado anterior, y queda plenamente justificada su utilidad. Una primera solución consiste en usar repetidamente el algoritmo de Dijkstra variando el nodo inicial; ahora bien, también se dispone del llamado *algoritmo de Floyd*, que proporciona una solución más compacta y elegante pensada especialmente para esta situación. El algoritmo de Floyd, definido por R.W. Floyd en 1962 sobre grafos dirigidos (su extensión al caso no dirigido es inmediata y queda como ejercicio para el lector) en "Algorithm 97: Shortest Path", *Communications ACM*, 5(6), p. 345, es un algoritmo dinámico¹⁷ que se estructura como un

¹⁶ Hay implementaciones asintóticamente más rápidas que usan una variante de montículo llamada *montículo de Fibonacci* (ing., *Fibonacci heap*), introducida por M. Fredman y R. Tarjan en 1987, que por su complejidad no se explica en este texto; v., por ejemplo, [HoS94, pp. 488-494].

¹⁷ La *programación dinámica* (ing., *dynamic programming*) es una técnica que estructura los algoritmos como bucles que, en cada paso, se acercan más a la solución, pero sin asegurar la obtención de ninguna parte definitiva antes de la finalización del algoritmo; para más detalles consultar, por ejemplo, [BrB97].

bucle, que trata un vértice denominado *pivote* en cada iteración, y que usa un vector bidimensional D indexado por pares de vértices para guardar la distancia mínima. Cuando u es el pivote, se cumple que $D[v, w]$ es la longitud del camino más corto entre v y w formado íntegramente por pivotes de pasos anteriores (excluyendo posiblemente las extremidades). La actualización de D consiste en comprobar, para todo par de nodos v y w , si la distancia entre ellos se puede reducir pasando por el pivote u mediante el cálculo de la fórmula $D[v, w] = \min(D[v, w], D[v, u] + D[u, w])$. Así pues, hay un par de diferencias en el funcionamiento de este algoritmo y el de Dijkstra: por un lado, los vértices se tratan en un orden independiente de las distancias y, por el otro, no se puede asegurar que ninguna distancia mínima sea definitiva hasta que acaba el algoritmo. Debe notarse que el algoritmo funciona incluso con aristas negativas, siempre que no haya ciclos negativos (es decir, que la suma de las etiquetas de las aristas que los componen no sea negativa), motivo por el que se podría relajar la restricción de aristas no negativas.

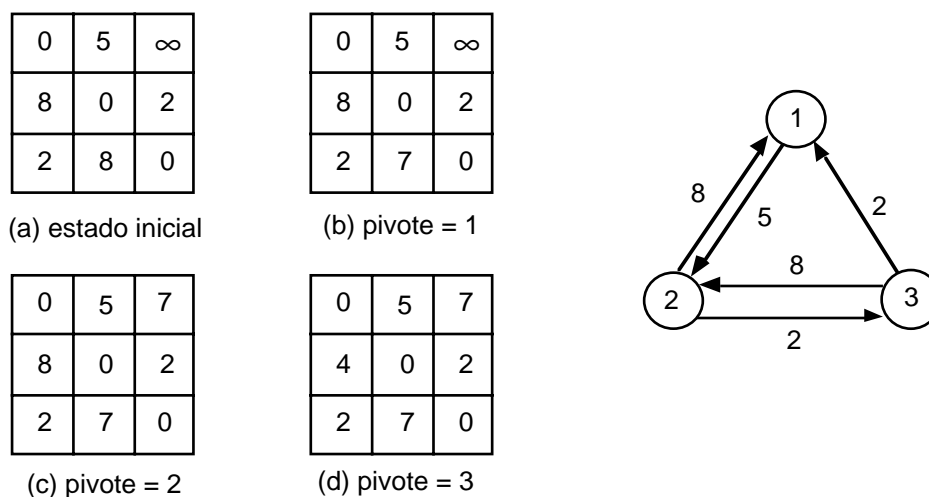


Fig. 6.38: ejemplo de funcionamiento del algoritmo de Floyd.

La especificación del problema se propone en el ejercicio 6.32. La implementación se presenta en la fig. 6.39. Notemos que la diagonal de D se inicializa a cero y nunca cambia. Por otra parte, observemos que tanto la fila como la columna del vértice u tratado en un paso del algoritmo permanecen invariables en este paso, por lo que se puede trabajar con una única matriz para actualizar las distancias mínimas. La inicialización de la matriz es ineficiente en caso de que el grafo esté implementado por (multi)listas de adyacencia, pero se deja así porque no afecta al coste total del algoritmo; si se considera conveniente, se puede modificar de manera similar al algoritmo de Dijkstra de la fig. 6.37, sustituyendo las n^2 llamadas a *etiqueta* por n bucles sobre los sucesores, de coste total a . El invariante usa una función $\text{pivotes}(u)$, que devuelve el conjunto de vértices que han sido pivotes en pasos anteriores

del algoritmo, lo que se puede determinar a partir del pivote del paso actual dado que los vértices son un tipo por enumeración (si no, el invariante necesitaría mantener un conjunto de los vértices que han sido pivotes).

```

{ $\mathcal{P} \equiv g$  es un grafo dirigido etiquetado que no contiene ciclos negativos}
función Floyd ( $g$  es grafo) devuelve vector [vértice,vértice] de etiq es
var D es vector [vértice, vértice] de etiq;  $u, v, w$  son vértice fvar
    {inicialmente la distancia entre dos vértices tiene el valor de la arista que
    los une; las diagonales se ponen a cero}
    para todo  $v$  dentro de vértice hacer
        para todo  $w$  dentro de vértice hacer
             $D[v, w] := \text{etiqueta}(g, v, w) \quad \{\infty \text{ si no hay arco}\}$ 
        fpara todo
             $D[v, v] := \text{ETIQUETA}.0$ 
        fpara todo
        para todo  $u$  dentro de vértice hacer
            { $I \equiv \forall v: v \in \text{vértice}: \forall w: w \in \text{vértice}: D[v, w] = \text{mínimo}(\{C: C \in \text{caminos}(g, v, w) \wedge$ 
                 $C \cap (\text{pivotes}(u) \cup \{v, w\}) = C: \text{coste}(C)\})$ 
            para todo  $v$  dentro de vértice hacer
                para todo  $w$  dentro de vértice hacer
                    si  $D[v, u] + D[u, w] < D[v, w]$  entonces  $D[v, w] := D[v, u] + D[u, w]$  fsi
                fpara todo
            fpara todo
        fpara todo
    devuelve D
    { $Q \equiv D = \text{todos\_caminos\_mínimos}(g, v)$ } -- v. ejercicio 6.32

```

Fig. 6.39: una codificación del algoritmo de Floyd.

La eficiencia temporal del algoritmo de Floyd es $\Theta(n^3)$, independientemente de la representación, igual que la aplicación reiterada de Dijkstra trabajando con una representación del grafo mediante matriz de adyacencia. Ahora bien, es previsible que Floyd sea más rápido (dentro del mismo orden asintótico) a causa de la simplicidad de cada paso del bucle y, como mínimo, siempre se puede argumentar que Floyd es más simple y elegante. Si el grafo está implementado por listas o multilistas de adyacencia y el algoritmo de Dijkstra se ayuda de colas prioritarias, el coste queda $\Theta((a+n)\log n)n$. La comparación entre este orden de magnitud y n^3 determina la solución más eficiente. Por último, debe notarse que el algoritmo de Floyd exige un espacio adicional $\Theta(1)$, mientras que cualquier versión de Dijkstra precisa una estructura $\Theta(n)$.

El algoritmo original de Floyd tampoco almacena los vértices que forman el camino; su modificación sigue la misma idea que en el caso del algoritmo de Dijkstra: si el camino mínimo de m a n pasa primero por p y después por q , la secuencia de vértices que forman el camino mínimo de p a q forma parte de la secuencia de vértices que forman el camino mínimo de m a n . Para implementar esta idea, se puede definir un vector bidimensional C indexado por vértices, de modo que $C[v, w]$ contiene un nodo u que forma parte del camino mínimo entre v y w . La concatenación de los caminos que se obtienen aplicando el mismo razonamiento sobre $C[v, u]$ y $C[u, w]$ da el camino mínimo entre v y w . La modificación del algoritmo queda como ejercicio.

6.6 Árboles de expansión minimales

Los algoritmos de la sección anterior permiten encontrar la conexión de coste mínimo entre dos vértices individuales de un grafo etiquetado. No obstante, a veces no interesa tanto minimizar conexiones entre nodos individuales como obtener un nuevo grafo que sólo contenga las aristas imprescindibles para una optimización global de las conexiones entre todos los nodos. Generalmente, esta optimización global implicará que diversos pares concretos de nodos no queden conectados entre ellos con el mínimo coste posible dado el grafo de partida.

Una aplicación inmediata es nuevamente la resolución de problemas que tienen que ver con distribuciones geográfica. Por ejemplo, supongamos que se dispone de un conjunto de ordenadores distribuidos geográficamente entre diferentes ciudades (posiblemente en diferentes países), a los que se quiere conectar para que puedan intercambiar datos, compartir recursos, etc. Hay muchas maneras de implementar esta conexión, pero supongamos que aquí se opta por pedir a las compañías telefónicas respectivas los precios del alquiler de la línea entre las ciudades implicadas. Una vez conocida esta información, para escoger las líneas que soportarán la red seguiremos una política austera, que asegure que todos los ordenadores se puedan comunicar entre ellos minimizando el precio total de la red. Pues bien, este problema puede plantearse en términos de grafos considerando que los ordenadores son los vértices y las líneas telefónicas son las aristas, formando un grafo no dirigido y etiquetado, del cual se quiere seleccionar el conjunto mínimo de aristas que permitan conectar todos los vértices con el mínimo coste posible. Para formular la resolución de este problema supondremos en el resto de esta sección que las etiquetas cumplen los mismos requerimientos que en la sección anterior.

Para plantear en términos generales los algoritmos que resuelven este tipo de enunciado, introduciremos primero unas definiciones (v. ejercicio 6.33 para su especificación):

- Un *árbol libre* (ing., *free tree*) es un grafo no dirigido conexo acíclico. Puede demostrarse por reducción al absurdo que todo árbol libre con n vértices presenta exactamente $n-1$ aristas; si se añade una arista cualquiera a un árbol libre se introduce un ciclo mientras que, si se borra, quedan vértices no conectados. También puede demostrarse que en un árbol libre cualquier par de vértices está unido por un único camino simple. Este tipo de árbol generaliza los tipos vistos en el capítulo 5, que se llaman por contraposición *árboles con raíz* (ing., *rooted tree*). De hecho, un árbol con raíz no es más que un árbol libre en el que un vértice se distingue de los otros y se identifica como raíz.
- Sea $g \in \{f: V \times V \rightarrow E\}$ un grafo no dirigido conexo y etiquetado no negativamente; entonces, un subgrafo g' de g , $g' \in \{f: V \times V \rightarrow E\}$, es un *árbol de expansión para g* (ing., *spanning tree*; también llamado *árbol de recubrimiento* o *árbol generador*) si es un árbol libre.
- Se define el conjunto de árboles de expansión para el grafo $g \in \{f: V \times V \rightarrow E\}$ no dirigido, conexo y etiquetado no negativamente, denotado por $ae(g)$, como:

$$ae(g) = \{g' \in \{f: V \times V \rightarrow E\} / g' \text{ es árbol de expansión para } g\}.$$
- Sea $g \in \{f: V \times V \rightarrow E\}$ un grafo no dirigido conexo y etiquetado no negativamente; entonces, el grafo no dirigido y etiquetado $g' \in \{f: V \times V \rightarrow E\}$ es un *árbol de expansión de coste mínimo para g* (ing., *minimum cost spanning tree*; para abreviar, los llamaremos *árboles de expansión minimales*), si g' está dentro de $ae(g)$ y no hay ningún otro g'' dentro de $ae(g)$, tal que la suma de las aristas de g'' sea menor que la suma de las aristas de g' .
- Se define el conjunto de árboles de expansión minimales para el grafo $g \in \{f: V \times V \rightarrow E\}$ no dirigido conexo y etiquetado no negativamente, $aem(g)$, como:

$$aem(g) = \{g' \in \{f: V \times V \rightarrow E\} / g' \text{ es árbol de expansión minimal para } g\}.$$

Dadas estas definiciones, se puede formular la situación del inicio del apartado como la búsqueda de un árbol de expansión minimal para un grafo no dirigido, conexo y etiquetado no negativamente. Hay varios algoritmos que resuelven este problema desde que, en el año 1926, O. Boruvka formuló el primero; en este texto, examinaremos dos que dan resultados satisfactorios, a pesar de que hay otros que pueden llegar a ser más eficientes y que, por su dificultad, no explicamos aquí (v., por ejemplo, [Tar83, cap. 6] y [CLR90, cap. 24]). Ambos algoritmos se basan en una misma propiedad que llamaremos *propiedad de los árboles de expansión minimales*: sea g un grafo no dirigido conexo y etiquetado no negativamente, $g \in \{f: V \times V \rightarrow E\}$, y sea U un conjunto de vértices tal que $U \subset V$, $U \neq \emptyset$; si $\langle u, v \rangle$ es la arista más pequeña de g tal que $u \in U$ y $v \in V-U$, existe algún $g' \in aem(g)$ que la contiene, es decir que cumple $g'(u, v) = g(u, v)$. La demostración por reducción al absurdo queda como ejercicio para el lector (v. [AHU83, pp. 234]).

6.5.1 Algoritmo de Prim

El algoritmo de Prim, definido por V. Jarník en el año 1930 y redescubierto por R.C. Prim y E.W. Dijkstra en 1957 (el primero en "Shortest connection networks and some generalizations", *Bell System Technical Journal*, 36, pp. 1389-1401, y el segundo en la misma referencia dada en el apartado 6.5.1), es un algoritmo voraz que aplica reiteradamente la propiedad de los árboles de expansión minimales, incorporando a cada paso una arista a la solución. El algoritmo (v. fig. 6.40 y 6.41) dispone de un conjunto U de vértices tratados (inicialmente, un vértice v cualquiera, porque todo vértice aparece en el árbol de expansión) y de su complementario, de manera que, de acuerdo con la propiedad de los árboles de expansión minimales, se selecciona la arista mínima que une un vértice de U con otro de su complementario, se incorpora esta arista a la solución y se añade su vértice no tratado dentro del conjunto de vértices tratados. La noción de arista se implementa como un trío $\langle \text{vértice}, \text{vértice}, \text{etiqueta} \rangle$. El invariante usa la operación *subgrafo* cuya especificación se propone en el ejercicio 6.1. Su coste es $\Theta(na)$, que puede llegar hasta $\Theta(n^3)$ si el grafo es denso.

```

{ $P \equiv g$  es un grafo no dirigido conexo etiquetado no negativamente}
función Prim ( $g$  es grafo) devuelve grafo18 es
var  $U$  es conj_vértices; gres es grafo;  $u, v$  son vértice;  $x$  es etiq fvar
    gres := crea;  $U := \{\text{un vértice cualquiera}\}$ 
    mientras  $U$  no contenga todos los vértices hacer
        { $I \equiv \text{subgrafo}(\text{gres}, U) \in \text{aem}(\text{subgrafo}(g, U))$ }
        seleccionar  $\langle u, v, x \rangle$  mínima tal que  $u \in U, v \notin U$ 
        gres := añade(gres,  $u, v, x$ );  $U := U \cup \{v\}$ 
    fmientras
devuelve gres
{ $Q \equiv \text{gres} \in \text{aem}(g)$ }

```

Fig. 6.40: presentación del algoritmo de Prim.

La versión preliminar se puede refinar y obtener el algoritmo de la fig. 6.42 de coste $\Theta(n^2)$, en general mejor que la anterior (nunca peor, porque la precondition garantiza que a vale como mínimo $n-1$). Esta nueva versión usa un vector *arista_mín*, indexado por vértices, que contiene información significativa para todos los vértices que todavía no forman parte de la solución. En concreto, si $v \notin U$, la posición *arista_mín*[v] contiene el par $\langle w, g(v, w) \rangle$, tal que $\langle v, w \rangle$ es la arista más pequeña que conecta v con un vértice $w \in U$. Para obviar U en la codificación, convenimos que si *arista_mín*[v] vale $\langle v, \infty \rangle$, entonces $v \in U$. La constante *un_vértice_cualquiera* puede ser un parámetro formal del universo que encapsule el algoritmo o bien un parámetro de la función.

¹⁸ También se puede devolver un árbol general o el conjunto de aristas del grafo resultante.

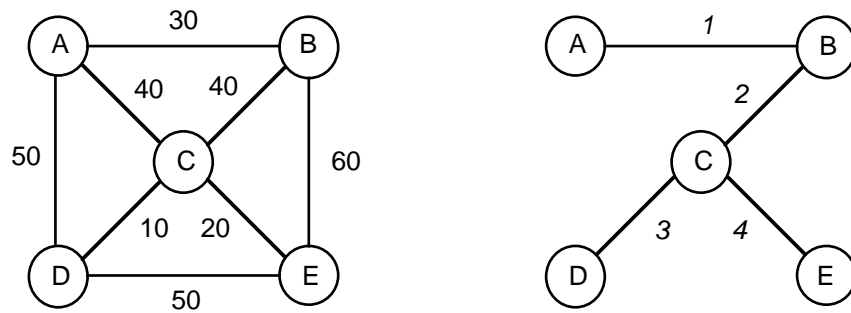


Fig. 6.41: aplicación del algoritmo de Prim: a la izquierda, un grafo no dirigido etiquetado acíclico; a la derecha, árbol resultante de aplicar Prim, tomando A como vértice inicial y numerando las aristas según el orden de inserción en la solución.

El cálculo de la eficiencia temporal de esta nueva versión queda:

- La inicialización es lineal en caso de matriz y cuadrática en caso de (multi)listas. Si se desea, se puede evitar el uso de *etiqueta* de manera similar a algunos algoritmos ya vistos, para reducir este último a lineal, aunque veremos que esta optimización no influye en el coste asintótico global del algoritmo.
- En el bucle principal hay algunas operaciones de orden constante, que no influyen en el coste total, el añadido de arista en el grafo y dos bucles internos. El añadido será constante usando una matriz y, en el caso peor, lineal usando (multi)listas, pero tampoco esto afectará al coste global. El bucle de selección es incondicionalmente $\Theta(n)$, ya que examina todas las posiciones del vector y, como se ejecuta $n-1$ veces, lleva a un coste $\Theta(n^2)$, mientras que el bucle de reorganización depende de la representación del grafo: en caso de matriz de adyacencia, calcular los adyacentes es $\Theta(n)$ y el coste total queda $\Theta(n^2)$, mientras que con listas, como siempre, su ejecución a lo largo del algoritmo lleva a un coste $\Theta(a+n)$, que en general es mejor, a pesar de que en este caso no afecta al orden global.

El coste asintótico total queda, en efecto, $\Theta(n^2)$, independientemente de la representación. Por lo que respecta al coste espacial, el espacio adicional empleado es $\Theta(n)$.

```

{ $P \equiv g$  es un grafo no dirigido conexo etiquetado no negativamente}
función Prim ( $g$  es grafo) devuelve grafo es
var arista_mín es vector [vértices] de vértice_y_etiq
    gres es grafo; primero, mín,  $v$ ,  $w$  son vértice;  $x$  es etiq
fvar
    {inicialización del resultado}
    primero := un_vértice_cualquiera
    para todo  $v$  dentro de vértice hacer
        arista_mín[ $v$ ] := <primero, etiqueta( $g$ , primero,  $v$ )>
    fpara todo
        {a continuación se aplica el método}
    gres := crea
    hacer  $n-1$  veces
        { $I \equiv \text{subgrafo}(gres, U) \in \text{aem}(\text{subgrafo}(g, U)) \wedge$ 
          $\forall v: v \notin U: \text{arista\_mín}[v] = \text{mínimo}\{u: u \in U: \text{etiqueta}(g, v, u)\},$ 
         donde  $U = \{u: u \in U \wedge \text{arista\_mín}[u] = \langle u, \infty \rangle: u\}$  }
        {primero se selecciona la arista mínima}
        mín := primero    {centinela, pues así  $\text{arista\_mín}[\text{mín}].\text{et}$  vale  $\infty$ }
        para todo  $v$  dentro de vértice hacer
            < $w$ ,  $x$ > := arista_mín[ $v$ ]
            si  $x < \text{arista\_mín}[\text{mín}].\text{et}$  entonces mín :=  $v$  fsi {como mínimo habrá uno}
        fpara todo
            {a continuación se añade a la solución}
            gres := añade(gres, mín, arista_mín[mín]. $v$ , arista_mín[mín]. $\text{et}$ )
            {se añade  $\text{mín}$  al conjunto de vértices tratados}
            arista_mín[mín] := <mín,  $\infty$ >
            {por último, se reorganiza el vector comprobando si la arista mínima de
             los vértices todavía no tratados los conecta a  $\text{mín}$ }
            para todo < $v$ ,  $x$ > dentro de adyacentes( $g$ , mín) hacer
                si (arista_mín[ $v$ ]. $v \neq v$ )  $\wedge$  ( $x < \text{arista\_mín}[v].\text{et}$ ) entonces
                    arista_mín[ $v$ ] := <mín,  $x$ >
                fsi
            fpara todo
        fhacer
    devuelve gres
    { $Q \equiv gres \in \text{aem}(g)$ }

```

Fig. 6.42: implementación eficiente del algoritmo de Prim.

6.5.2 Algoritmo de Kruskal

Como el algoritmo de Prim, el algoritmo ideado por J.B. Kruskal en el año 1956 ("On the shortest spanning subtree of a graph and the traveling salesman problem", en *Proceedings American Math. Society*, 7, pp. 48-50) se basa en la propiedad de los árboles de expansión minimales: partiendo del árbol vacío, se selecciona a cada paso la arista de menor etiqueta que no provoque ciclo sin requerir ninguna otra condición sobre sus extremos.

El algoritmo se presenta en las figs. 6.43 y 6.44; el invariante usa la operación *componentes*, que devuelve el conjunto de subgrafos que forman componentes conexos de un grafo dado. Su implementación directa es ineficiente dado que hay que encontrar la arista mínima y comprobar si se forma un ciclo. El estudio del invariante, no obstante, da una visión alternativa del algoritmo que permite implementarlo eficientemente: se considera que, en cada momento, los vértices que están dentro de un componente conexo en la solución forman una clase de equivalencia, y el algoritmo se puede considerar como la fusión continuada de clases hasta obtener un único componente con todos los vértices del grafo.

```

{ $\mathcal{P} \equiv g$  es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal ( $g$  es grafo) devuelve grafo es
var gres es grafo; u, v son vértice; x es eti; fvar
  gres := crea
  mientras gres no sea conexo hacer
    { $I \equiv \forall g': g' \in \{f: U \times U \rightarrow E\} \wedge g' \in \text{componentes}(\text{gres}): g' \in \text{aem}(\text{subgrafo}(g, U))\}$ }
    seleccionar  $\langle u, v, x \rangle$  mínima, todavía no examinada
    si no provoca ciclo entonces gres := añade(gres, u, v, x) fsi
  fmientras
  devuelve gres
{ $Q \equiv \text{gres} \in \text{aem}(g)$ }

```

Fig. 6.43: presentación del algoritmo de Kruskal.

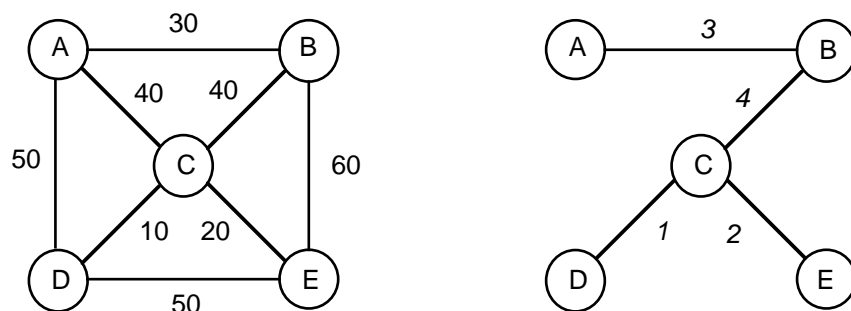


Fig. 6.44: aplicación del algoritmo de Kruskal sobre el grafo de la fig. 6.41.

En el grafo de la figura 6.44, la evolución de las clases es:

$$\{[A], [B], [C], [D], [E]\} \rightarrow \{[A], [B], [C], [D, E]\} \rightarrow \{[A], [B], [C, D, E]\} \rightarrow \{[A, B], [C, D, E]\} \rightarrow \{[A, B, C, D, E]\}$$

Así, puede construirse una implementación eficiente del algoritmo de Kruskal, que utiliza la especificación y la implementación arborescente del TAD de las relaciones de equivalencia. El algoritmo resultante (v. fig. 6.45) organiza, además, las aristas dentro de una cola de prioridades implementada con un montículo, de manera que se favorece el proceso de obtención de la arista mínima a cada paso del bucle. El invariante establece las diversas propiedades de las estructuras usando las operaciones de los TAD que intervienen y algunas operaciones de grafos cuya especificación se propone en el ejercicio 6.1.

```

{P ≡ g es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal (g es grafo) devuelve grafo es
var T es colapr_aristas; gres es grafo; u, v son vértice; x es etiq
    C es reseq_vértice; ucomp, vcomp son nat
fvar
    C := RELEQ.crea {inicialmente cada vértice forma una única clase}
    gres := GRAFO.crea; T := COLAPR.crea
    {se colocan todas las aristas en la cola}
    para todo v dentro de vértice hacer
        para todo <u, x> dentro de adyacentes(g, v) hacer T := inserta(T, v, u, x) fpara todo
    fpara todo
    mientras RELEQ.cuántos?(C) > 1 hacer
        {I ≡ ∀g': g' ∈ {f: U × U → E} ∧ g' ∈ componentes(gres): g' ∈ aem(subgrafo(g, U)) ∧
          ∀<u, v>: <u, v> ∈ dom(gres): etiqueta(g, u, v) ≤ mínimo(T).et ∧
          ∀u, v: u, v ∈ vértice: clase(C, u) = clase(C, v) ⇔ conectados(gres, u, v) }
        {se obtiene y se elimina la arista mínima de la cola}
        <u, v, x> := COLAPR.menor(T); T := COLAPR.borra(T)
        {a continuación, si la arista no provoca ciclo se añade a la solución y se
          fusionan las clases correspondientes}
        ucomp := RELEQ.clase(C, u); vcomp := RELEQ.clase(C, v)
        si ucomp ≠ vcomp entonces
            C := RELEQ.fusiona(C, ucomp, vcomp)
            gres := añade(gres, u, v, x)
        fsi
    fmientras
devuelve gres
{Q ≡ gres ∈ aem(g)}
```

Fig. 6.45: implementación eficiente del algoritmo de Kruskal.

El coste del algoritmo se puede calcular de la siguiente manera:

- La creación de la relación de equivalencia es $\Theta(n)$, y la del grafo resultado es $\Theta(n^2)$ usando una representación por matriz y $\Theta(n)$ usando listas de adyacencia.
- Las a inserciones consecutivas de aristas en la cola prioritaria quedan $\Theta(a \log a)$. Como a es un valor entre $n-1$ y $n(n-1)/2$, sustituyendo por estos valores en $\Theta(a \log a)$ obtenemos $\Theta(a \log n)$ en ambos casos, habida cuenta que $\Theta(a \log n^2) = \Theta(2a \log n)$. El coste se puede reducir hasta $\Theta(a)$ usando la función *organiza* de los pilones para convertir un vector con todas las aristas en la cola prioritaria correspondiente. También debe considerarse que la obtención de las aristas es $\Theta(n^2)$ con matriz y $\Theta(a)$ con listas.
- Como mucho, hay a consultas y supresiones de la arista mínima; el coste de la consulta es constante y el de la supresión logarítmico y, por ello, este paso queda $\Theta(a \log n)$ en el caso peor. Ahora bien, si la configuración del grafo conlleva un número pequeño de iteraciones (siempre $n-1$ como mínimo, eso sí) el coste es realmente menor, hasta $\Theta(n \log n)$; si la cola prioritaria se crea usando *organiza*, este hecho puede ser significativo. Notemos que el caso peor es aquél en que la arista de etiqueta mayor del grafo es necesaria en todo árbol de expansión minimal asociado al grafo, mientras que el caso mejor es aquél en que las $n-1$ aristas más pequeñas forman un grafo acíclico.
- El coste de las operaciones sobre las relaciones ha sido estudiado en la sección 6.2: averiguar cuántas clases hay es constante mientras que *fusiona* se ejecuta $n-1$ veces y *clase* entre $2(n-1)$ y $2a$ veces, por lo que el coste total al término del algoritmo es $\Theta(n)$ en el caso mejor y $\Theta(a)$ en el caso peor, usando la técnica de compresión de caminos.
- Las $n-1$ inserciones de aristas quedan $\Theta(n)$ con matriz y $\Theta(n^2)$ con listas. En este algoritmo, al contrario que en el anterior, este hecho es importante, por lo que debe evitarse el uso de una implementación encadenada del grafo resultado. Otra alternativa es eliminar la comprobación de existencia de la arista en la función *añade*, pues la mecánica del algoritmo de Kruskal garantiza que no habrán inserciones repetidas de aristas; como resultado, la función quedaría constante siempre y cuando las listas no se ordenaran a cada inserción sino en un paso posterior una vez obtenidas todas las aristas del resultado¹⁹.

Así, y teniendo en cuenta el razonamiento sobre la implementación del grafo resultado, el coste total del algoritmo es: en el caso peor (se examinan todas las aristas), $\Theta(a \log n)$ con listas de adyacencia y $\Theta(n^2 + a \log n)$ con matriz; en el caso mejor ($n-1$ aristas examinadas), $\Theta(a + n \log n)$ con listas y $\Theta(n^2)$ con matriz. La comparación de estas magnitudes con $\Theta(n^2)$ indica la conveniencia de usar Prim o Kruskal para obtener el árbol de expansión minimal (a tal efecto, v. ejercicio 6.23). Al tomar esta decisión, también puede influir que Kruskal necesita más memoria auxiliar a causa del montículo y de la relación, incluso asintóticamente hablando, porque el montículo exige un espacio $\Theta(a)$, por $\Theta(n)$ del vector auxiliar de Prim.

¹⁹ Recordemos que la ordenación de las listas de adyacencia es necesaria para que la implementación sea correcta con respecto a la especificación en el marco de la semántica inicial.

Ejercicios

Nota preliminar: en los ejercicios de este capítulo, debe calcularse el coste de todos los algoritmos que se pidan.

6.1 Escribir la especificación de los grafos: **a)** dirigidos no etiquetados; **b)** no dirigidos y etiquetados; **c)** no dirigidos y no etiquetados. Enriquecer estos universos y el de los grafos dirigidos y etiquetados con las operaciones:

caminos: $\text{grafo } \text{vértice } \text{vértice} \rightarrow \text{cjt_sec_vértices}$, devuelve el conjunto de caminos simples entre los dos vértices dados; cada camino se representa por una secuencia de vértices.

antecesores, descendientes: $\text{grafo } \text{vértice} \rightarrow \text{cjt_vértices}$, devuelve todos los vértices que son antecesores o descendientes del vértice dado, respectivamente.

ciclo: $\text{grafo } \text{vértice} \rightarrow \text{bool}$, responde si el vértice dado forma parte de algún ciclo.

conectados: $\text{grafo } \text{vértice } \text{vértice} \rightarrow \text{bool}$, indica si hay algún camino entre los vértices dados.

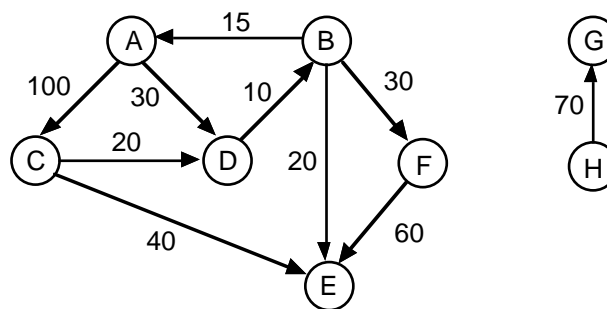
componentes: $\text{grafo} \rightarrow \text{cjt_grafos}$, devuelve un conjunto con todos los componentes conexos del grafo; cada componente se representa mediante un grafo.

conexo, acíclico: $\text{grafo} \rightarrow \text{bool}$, indican si el grafo es conexo o acíclico, respectivamente.

subgrafo: $\text{grafo } \text{cjt_vértices} \rightarrow \text{grafo}$, tal que $\text{subgrafo}(g, U)$ devuelve el grafo que tiene como nodos los elementos de U y como arcos aquéllos de g que conectan nodos de U .

6.2 Hacer una comparación exhaustiva de las diversas representaciones de grafos en lo que se refiere a espacio y tiempo de ejecución de las operaciones del TAD, estudiando para qué operaciones y características son más adecuadas.

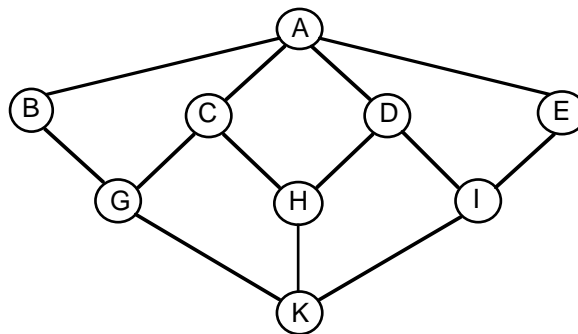
6.3 Representar de todas las formas posibles el grafo:



6.4 Diseñar un par de algoritmos para transformar un grafo representado mediante matriz de adyacencia en una representación mediante listas de adyacencia y viceversa.

6.5 En un ordenador cualquiera, los números enteros y los punteros se representan con 32 bits y los booleanos con un único bit. Calcular el número mínimo de aristas que debe tener un grafo dirigido no etiquetado de 100 vértices, suponiendo que los vértices se identifican mediante enteros, con el objetivo de que su representación mediante matriz de adyacencia ocupe menos bits que mediante listas de adyacencia. Repetir el ejercicio suponiendo la presencia de enteros como etiquetas.

6.6 Suponiendo que los sucesores de un nodo dado siempre se obtienen en orden alfabético, recorrer el grafo siguiente en profundidad y en anchura empezando por A:



6.7 Usar algún algoritmo conocido para decidir si un grafo no dirigido es conexo. Modificar el algoritmo para que, en caso de que no lo sea, encuentre todos los componentes conexos.

6.8 Escribir un algoritmo a partir de algún recorrido para calcular la *clausura transitiva* (ing., *transitive closure*) de un grafo no dirigido, es decir, que determine para todo par de nodos si están conectados o no. Comparar con el resultado del ejercicio 6.17.

6.9 Escribir un algoritmo que, dado un grafo dirigido y dos vértices de este grafo, escriba todos los caminos simples de un vértice al otro.

6.10 Escribir un algoritmo para enumerar todos los ciclos elementales dentro de un grafo no dirigido. ¿Cuál es el máximo número de ciclos elementales que se pueden encontrar?

6.11 Modificar el algoritmo de recorrido en anchura para obtener un programa que detecte si un grafo no dirigido presenta algún ciclo.

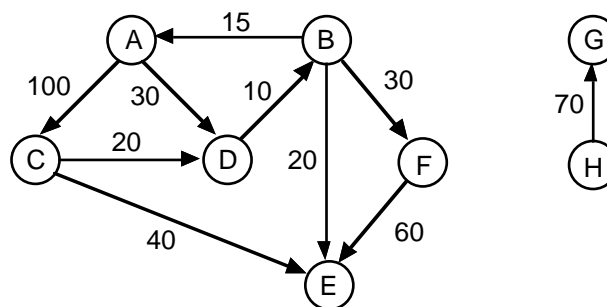
6.12 Formular el algoritmo de ordenación topológica a partir del algoritmo de recorrido en profundidad.

6.13 Una expresión aritmética puede representarse con un grafo dirigido acíclico. Indicar exactamente cómo, y qué algoritmo debe aplicarse para evaluarla. Escribir un algoritmo que transforme una expresión representada con un árbol en una expresión representada con un

grafo dirigido acíclico.

6.14 Los árboles de expansión no necesariamente minimales son un concepto útil para analizar redes eléctricas y obtener un conjunto independiente de ecuaciones. No obstante, en lugar de aplicar los algoritmos presentados en la sección 6.5, que tienen un coste elevado, se pueden adaptar los algoritmos de recorrido de grafos presentados en la sección 6.3, que permiten construir árboles de expansión con las aristas usadas para visitar los nodos del grafo, y entonces añadir consecutivamente el resto de aristas para formar el conjunto de ecuaciones (para más detalles, v. [HoS94, pp. 335-337] y [Knu68, pp. 393-398]). Modificar alguno de estos recorridos para utilizarlo en este contexto.

6.15 a) Encontrar el coste de los caminos mínimos entre todo par de nodos del grafo:



i) aplicando reiteradamente el algoritmo de Dijkstra, y ii) aplicando el algoritmo de Floyd.
b) Ahora, además, calcular cuáles son los nodos que forman los caminos mínimos entre toda pareja de nodos.

6.16 Razonar por qué el algoritmo de Dijkstra no siempre funciona para grafos con etiquetas no necesariamente positivas. Mostrar un grafo de tres nodos con alguna arista negativa para el cual no funcione el algoritmo.

6.17 Modificar el algoritmo de Floyd para que calcule la clausura transitiva de un grafo. El resultado es el denominado algoritmo de Warshall que, curiosamente, es anterior al de Floyd (v. "A theorem on boolean matrices", S. Warshall, *Journal of the ACM*, 9(1), pp.11-12, 1962). Comparar con el resultado del ejercicio 6.8.

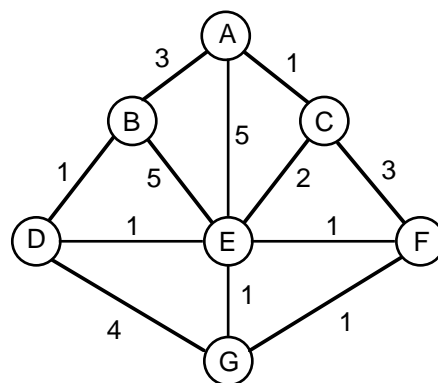
6.18 Modificar los algoritmos de Dijkstra y Floyd para que también devuelvan una estructura que almacene de forma compacta los caminos minimales. Diseñar e implementar algoritmos recursivos que recuperen los caminos minimales a partir de la información obtenida.

6.19 Para todas las situaciones que a continuación se enumeran, decir si afectan o no al funcionamiento normal del algoritmo de Dijkstra; en el caso de que lo hagan, indicar si existe alguna manera de solucionar el problema: **a)** el grafo presenta algún ciclo; **b)** el grafo es no

dirigido; **c)** las aristas pueden ser negativas; **d)** puede haber aristas de un nodo a sí mismo; **e)** pueden haber varias aristas de un nodo a otro; **f)** hay nodos inaccesibles desde el nodo inicial; **g)** hay varios nodos unidos con el nodo inicial por caminos igual de cortos.

6.20 Dado un grafo dirigido determinar, en función de su representación y de la relación entre el número de aristas y el número de nodos, si es mejor el algoritmo de Floyd o el algoritmo de Dijkstra reiterado para determinar la conexión entre toda pareja de nodos. En el segundo caso, especificar claramente cómo hay que modificar el algoritmo para adaptarlo al problema. Decir si hay alguna situación concreta que aconseje aplicar algún otro algoritmo conocido sobre el grafo.

6.21 Dado el grafo siguiente:



indicar, a ojo de buen cubero, todos los posibles árboles de expansión minimales asociados. A continuación, aplicarles los algoritmos de Kruskal y de Prim, estudiando todas las resoluciones posibles de las indeterminaciones que presente. Comparar los resultados.

6.22 El coste espacial adicional del algoritmo de Prim de la fig. 6.20 es $\Theta(n)$ asintóticamente y no se puede mejorar; ahora bien, las dos estructuras empleadas están fuertemente interrelacionadas, como muestra claramente el invariante, y se pueden fusionar, dado que dentro del conjunto sólo hay vértices de cero predecesores no tratados. Codificar esta nueva versión.

6.23 Estudiar la posibilidad de organizar el vector *arista_mín* del algoritmo de Prim como una cola prioritaria. ¿Qué coste queda? ¿Influye en este coste la configuración del grafo? (Esta implementación fue propuesta por D.B. Johnson el año 1975 en "Priority queues with update and finding minimum spanning trees", *Information Processing Letters*, 4, pp. 53-57.)

6.24 Razonar cómo afectan la implementación del grafo y la relación entre el número de nodos y aristas en la elección entre Prim y Kruskal.

6.25 Para todas las situaciones que se enumeran a continuación, decir si afectan o no al funcionamiento normal de los algoritmos de Prim y Kruskal; en caso de que lo hagan, indicar si hay alguna forma posible de solucionar el problema: **a)** el grafo presenta algún ciclo; **b)** el grafo es dirigido; **c)** las aristas pueden ser negativas; **d)** puede haber aristas reflexivas; **e)** puede haber varias aristas de un nodo a otro; **f)** hay nodos inaccesibles desde el nodo inicial; **g)** hay varios nodos unidos con el nodo inicial con una arista igual de corta.

6.26 Diseñar un algoritmo que decida si un grafo no dirigido dado contiene algún subgrafo completo de k vértices, usando la signatura que más convenga.

6.27 La región de Aicenev, al sur de Balquistán, es famosa en el mundo entero por su sistema de comunicaciones basado en canales que se ramifican. Todos los canales son navegables y todas las localidades de la región están situadas cerca de algún canal.

a) Una vez al año, los alcaldes de la región se reúnen en la localidad de San Macros desplazándose cada uno en su propio *vaporetto* oficial. Proponer a los alcaldes un algoritmo que les permita llegar a San Macros de forma que entre todos ellos hayan recorrido la mínima distancia.

b) A consecuencia de la crisis energética, los alcaldes han decidido compartir los *vaporettos* de manera que si, durante su trayecto, un *vaporetto* pasa por una ciudad donde hay uno o más alcaldes, los recoge, formándose entonces grupos que se dirigen a San Macros. Discutir la validez de la solución anterior. Si es necesario, proponer otra que reduzca la distancia recorrida por los *vaporettos* (pero no necesariamente por los alcaldes).

6.28 Una empresa dispone de un conjunto de ordenadores distribuidos en un ámbito geográfico. Varias parejas de estos ordenadores se podrían conectar físicamente mediante una línea cualquiera de comunicación (bidireccional). La empresa hace un estudio previo del coste de transmisión por unidad de información que tendría cada conexión en función de algunos parámetros (longitud de la línea, calidad de la señal, etc.) y, a partir de este resultado, quiere calcular qué conexiones tiene que usar para que se cumpla que: **a)** dos ordenadores predeterminados del conjunto estén comunicados entre sí con el mínimo coste de transmisión por unidad de información posible y **b)** todos los ordenadores estén comunicados, de manera que el coste total de la transmisión por unidad de información sea mínimo, cumpliendo la condición del punto anterior. Proponer un algoritmo que resuelva el problema.

6.29 Una multinacional tiene agencias en numerosas ciudades dispersas por todo el planeta. Suponiendo que se conoce el coste de un minuto de conversación telefónica entre todo par de ciudades, se quiere encontrar la forma de decidir a qué agencias debe llamarse desde toda agencia para que un mensaje originario de cualquiera de ellas llegue al destinatario con un coste total mínimo. Pensar cuál es la información mínima imprescindible que debe almacenarse en cada agencia.

6.30 a) El ayuntamiento de Villatortas de Arriba proyecta construir una red metropolitana. Previamente, el consistorio decide la ubicación de cada estación y calcula la distancia entre todo par de ellas, de manera que tan sólo queda determinar qué tramos (es decir, qué conexiones entre estaciones) se construyen. La política elegida consiste en reducir la longitud total de la red con la condición de que todas las estaciones queden conectadas entre sí (es decir, que desde cualquier estación se pueda llegar a cualquier otra). El resultado puede implicar la existencia de diferentes líneas de metro y, en este caso, existen estaciones que pertenecen a más de una línea y que permiten hacer transbordos (es decir, cambios de línea). Asociar esta situación a un modelo y a un algoritmo conocido y explicar cómo debería interpretarse la salida de dicho algoritmo para encontrar una posible configuración de la red (en el caso general, habrá diversas alternativas; devolver una cualquiera).

b) Al construir la red metropolitana, el ayuntamiento decide no seguir los resultados del estudio anterior sino que, movido por intereses oscuros, prefiere aceptar la propuesta de la coalición en el poder. A continuación, el consistorio quiere disponer de aquellas máquinas que, dadas dos estaciones, encuentran la manera de ir de una a otra reduciendo la distancia recorrida (suponer que un transbordo entre líneas tiene distancia 0). Justificar qué algoritmo debe implementarse en una máquina instalada en la estación *E* en los supuestos siguientes:

i) Que pueda pedirse el trayecto más corto entre cualquier par de estaciones.

ii) Que sólo pueda pedirse el trayecto más corto desde *E* a cualquier otra estación.

(En caso que haya más de una alternativa, devolver una cualquiera.)

c) Después de un estudio de los técnicos municipales, se considera que un criterio mejor para programar las máquinas consiste en reducir el número de estaciones de paso para ir de una estación a otra (suponer que un transbordo entre líneas no cuenta ningún paso). Explicar qué modelo y qué algoritmo se necesita para programar las máquinas. (En caso de que haya más de una alternativa, devolver una cualquiera.)

d) En las siguientes elecciones municipales, el principal partido de la oposición basa su campaña en sustituir estas máquinas por otras que, dadas dos estaciones, encuentran la manera de ir de una a otra con el número mínimo de transbordos. ¿Es posible? En caso afirmativo, decir qué modelo y qué algoritmo se necesita. ¿Qué debe modificarse en la solución si, en caso de haber diversas alternativas con el mínimo número de transbordos, se pidiera la que reduzca el número de estaciones de paso, siendo dicho cálculo lo más eficiente posible?

6.31 En este ejercicio nos centramos en la especificación de los diferentes recorridos sobre grafos estudiados en la sección 6.3. Considerando que hay muchos recorridos concretos asociados a una estrategia particular y que, además, la especificación de la obtención de uno (o todos) de estos recorridos, dentro del marco de la semántica inicial, resultaría prácticamente en una implementación por ecuaciones (debido al conocido problema de la sobreespecificación que presenta esta semántica), se ha optado por introducir diferentes predicados que comprueben si una lista con punto de interés es un recorrido válido para un grafo dirigido no etiquetado según cada estrategia.

a) Especificar la función *están_todos*: $lista_vértices \rightarrow bool$ que comprueba si una lista contiene todos los vértices del grafo, lo que es un requisito indispensable para que la lista se pueda considerar como un recorrido válido según cualquier estrategia. Definir los parámetros formales necesarios sobre los vértices que permitan obtener todos los valores del tipo. Si es necesario, enriquecer las listas con alguna operación.

b) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación *es_recorrido_profundidad*: $lista_vértices\ grafo \rightarrow bool$; l será un recorrido en profundidad de g si, para todo vértice v de la lista, el recorrido avanza por alguno de los caminos que salen de él y llega tan lejos como puede (es decir, hasta un vértice ya visitado o hasta un vértice sin sucesores). Los caminos se definen como una secuencia de vértices. Especificar las operaciones:

todos_caminos_no_cíclicos: $grafo\ lista_vértices\ cjt_vértices \rightarrow cjt_secs_vértices$, que genera el conjunto de caminos no cíclicos que salen de alguno de los vértices dados a la lista; el conjunto de vértices representa los vértices por los que ya ha pasado el camino y es necesario precisamente para controlar si es cíclico o no. Si es necesario, enriquecer las listas con alguna operación.

sigue_alguno: $cjt_secs_vértices\ lista_vértices \rightarrow bool$, que comprueba si hay algún camino de los generados anteriormente que se solape total o parcialmente con los vértices que hay en la lista, a partir de la posición actual; en este caso, se puede afirmar que efectivamente se ha seguido este camino. Si el solapamiento no es total, es indispensable que los vértices del camino que no se solapan ya hayan sido visitados previamente, porque este es el único motivo que justifica la interrupción en el recorrido de un camino.

prof_pro: $lista_vértices\ grafo \rightarrow bool$, que comprueba que todos los vértices de la lista cumplen la condición del recorrido en profundidad.

De esta manera, la especificación de la operación *es_recorrido_profundidad*? queda:

$$es_recorrido_profundidad(l, g) = están_todos(l) \wedge prof_pro(principio(l), g)$$

c) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación *es_recorrido_anchura*: $lista_vértices\ grafo \rightarrow bool$; l será un recorrido en anchura de g si, para todo vértice v de la lista, el recorrido explora inmediatamente todos sus sucesores todavía no visitados. Especificar las operaciones:

subsec: $sec_vértices\ lista_vértices \rightarrow sec_vértices$, que genera la subsecuencia del recorrido que tiene como primer elemento el primer no sucesor de v que aparece detrás de v . La lista de entrada representa los vértices sucesores y la secuencia contiene, inicialmente, los nodos que hay a partir del elemento actual, sin incluirlo.

no_hay_ninguno: $sec_vértices\ lista_vértices \rightarrow bool$, que comprueba que, en la subsecuencia obtenida con *subsec*, no hay ningún sucesor de v . Es necesario pasar la lista de vértices sucesores.

anchura: $lista_vértices\ grafo \rightarrow bool$, que comprueba que todos los vértices de la lista cumplen la condición de recorrido en anchura.

Así, la especificación de la operación *es_recorrido_anchura?* queda:

$$\text{es_recorrido_anchura}(l, g) = \text{están_todos}(l) \wedge \text{anchura}(\text{principio}(l), g)$$

d) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación *es_ordenación_topológica?*: *lista_vértices grafo* \rightarrow *bool*; l será un recorrido en ordenación topológica de g , si todo vértice v aparece en la lista después de todos sus antecesores. Especificar la operación *ord_top*: *lista_vértices grafo* \rightarrow *bool*, que comprueba que, para cada arista del grafo, se cumple la propiedad dada. Así, la especificación de la operación *es_ordenación_topológica?* queda:

$$\text{es_ordenación_topológica}(l, g) = \text{están_todos}(l) \wedge \text{ord_top}(\text{principio}(l), g)$$

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación.

6.32 En este ejercicio se quiere especificar la obtención de caminos mínimos en grafos.

a) Sea g un grafo dirigido y etiquetado no negativamente y sea v un vértice. Especificar la operación *camino_mínimo*: *grafo vértice* \rightarrow *tabla_vértices_y_etiqs*; *camino_mínimo*(g, v) devuelve una tabla T , tal que *consulta*(T, w) es igual al coste del camino mínimo entre v y w .

b) Sea g un grafo dirigido y etiquetado no negativamente. Especificar la operación *caminos_mínimos*: *grafo* \rightarrow *tabla_parejas_vértices_y_etiqs*; *caminos_mínimos*(g) devuelve una tabla T , tal que *consulta*($T, \langle v, w \rangle$) es igual al coste del camino mínimo entre v y w .

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación. Si es conveniente, usar alguna de las operaciones especificadas en el ejercicio 6.1. Implementar estas variantes generales de los algoritmos de cálculo de caminos mínimos.

6.33 En este ejercicio se quiere especificar el concepto de árbol de expansión minimal asociado a un grafo siguiendo varios pasos.

a) Sea g un grafo no dirigido y etiquetado no negativamente. Especificar la operación *no_cíclicos*: *grafo* \rightarrow *cjt_grafos*, tal que *no_cíclicos*(g) devuelve el conjunto de subgrafos de g que no presentan ningún ciclo.

b) Sea C un conjunto de grafos no dirigidos, acíclicos y etiquetados no negativamente. Especificar la operación *maximales*: *cjt_grafos* \rightarrow *cjt_grafos* tal que *maximales*(C) devuelve un conjunto D de grafos conexos, $D \subseteq C$. Notar que, por la propiedad de los árboles libres, o bien D es el conjunto vacío (si no había ningún grafo conexo en C), o bien todos sus elementos tienen exactamente $n - 1$ aristas, siendo n el número de vértices de los grafos.

c) Sea g un grafo no dirigido y etiquetado no negativamente. Especificar la operación *ae*: *grafo* \rightarrow *cjt_grafos* tal que *ae*(g) devuelve el conjunto de grafos que son árboles de expansión para g .

d) Sea C un conjunto de grafos, resultado de un cálculo de *ae*. Especificar la operación

minimales: $cjt_grafos \rightarrow cjt_cjt_grafos$ tal que *minimales*(*S*) devuelve un conjunto *D* de conjuntos de grafos que son minimales (respecto la suma de las etiquetas de sus aristas) dentro del conjunto *C*, $D \subseteq C$.

e) Finalmente, sea *g* un grafo no dirigido y etiquetado no negativamente. Especificar la operación *aem*: $grafo \rightarrow cjt_cjt_aristas$ que devuelve el conjunto de árboles de expansión minimales asociados al grafo.

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación.

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 destacar que, debido al conflicto entre eficiencia y modularidad expuesto en la sección 2.4, usaremos cuando convenga versiones recorribles o abiertas de los TAD. 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 si no se utilizan estas versiones 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 2.4.1). 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 módulos reusables 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 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. 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).

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 módulos reusables) 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 4.4 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

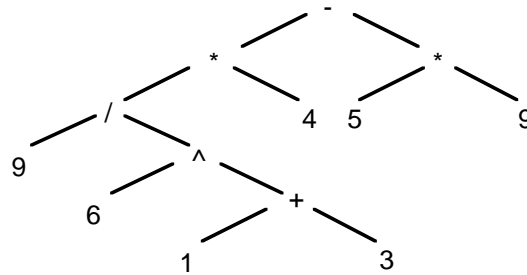


Fig. 7.1: representación jerárquica de la expresión $9/6 \wedge (1+3) * 4 - 5*9$.

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 (v. fig. 7.2). 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.

$2 + 3 * 5$	$2\ 3\ 5\ *\ +$
$(2 + 3) * 5$	$2\ 3 + 5\ *$
$9 / 6 \wedge (1 + 3) * 4 - 5 * 9$	$9\ 6\ 1\ 3 + \wedge / 4 * 5\ 9\ * -$

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 precondition, 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 aplicarse 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 módulos 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 empuja '*' 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 empuja, 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 \neg *vacía?*(*e*) hacer

{*I* \equiv 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 \neg *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*)) \geq *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 módulos; según el sistema gestor de la biblioteca, podrían residir los tres en un único módulo, 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: \rightarrow operador

aplica: valor operador valor \rightarrow valor

=, _≠_: operador operador \rightarrow bool

prio_ent, prio_pila: operador \rightarrow entero

ecns prio_ent(nulo) > prio_pila(nulo) = cierto

$[x \neq \text{nulo} \wedge x \neq \text{par_cerrar}] \Rightarrow \text{prio_ent}(x) > \text{prio_ent}(\text{par_cerrar}) = \text{cierto}$

... 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 \rightarrow símbolo

crea_op: operador \rightarrow símbolo

es_operador?: símbolo \rightarrow bool

valor: símbolo \rightarrow valor

operador: símbolo \rightarrow operador

errores $[\neg \text{es_operador?}(s)] \Rightarrow \text{operador}(s)$; $[\text{es_operador?}(s)] \Rightarrow \text{valor}(s)$

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 \rightarrow 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. Así, en la memoria pueden quedar agujeros (es decir, bloques libres), que han de tratarse en reservas posteriores (v. fig. 7.10).

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*).

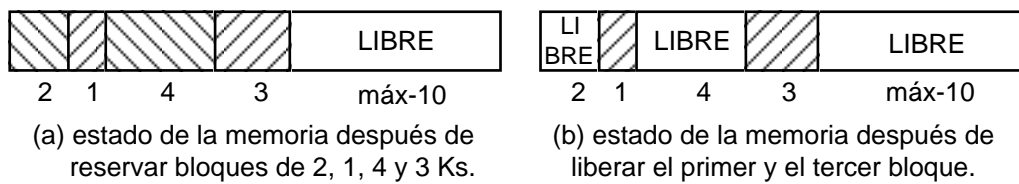


Fig. 7.10: reserva y liberación de bloques en la memoria.

- 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*¹.

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. Asimismo, 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.

¹ Hay otra estrategia similar, la del *peor ajuste* (ing., *worst fit*), que aquí no consideramos.

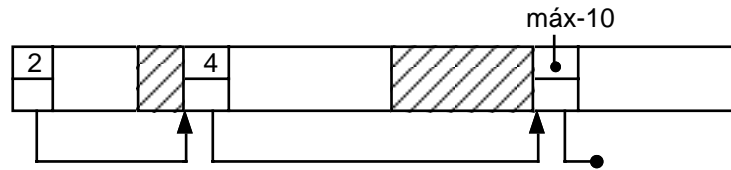


Fig. 7.11: implementación de la memoria de la figura 7.10(b).

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 exactamente² 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_espacio3 (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.

² En realidad, el algoritmo procura no dejar bloques libres menores de dos palabras para guardar el tamaño y el encadenamiento.

³ 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:

	s1	s2	s3	
--	----	----	----	--

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.

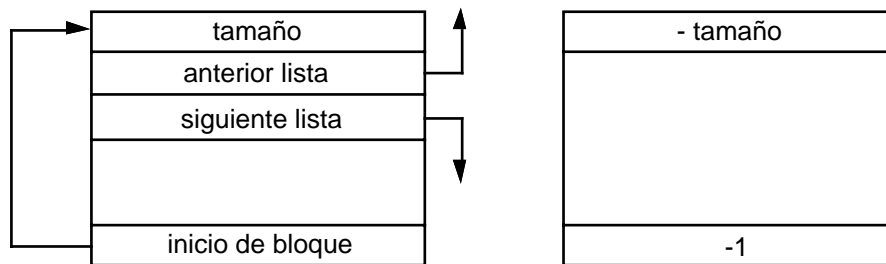


Fig. 7.13: información asociada a cada bloque de memoria (izq., libre; der., ocupado).

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 act4 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

```

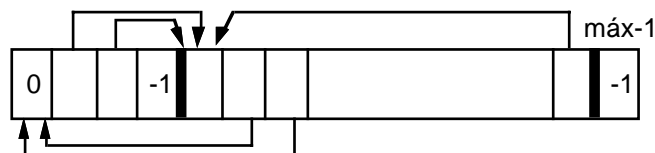


Fig. 7.14: creación de la memoria: algoritmo y esquema.

⁴ De hecho, *act* también podría residir en alguna palabra de la memoria *M* y así la estructura sería totalmente autocontenida.

El algoritmo de liberación de espacio de un bloque s_2 (v. fig 7.15) ha de considerar las cuatro combinaciones posibles de los estados de los bloques adyacentes al liberado (s_1 y s_3).

- s_1 y s_3 están ocupados: se inserta s_2 dentro de la lista de bloques libres y se actualiza su información de control.
- s_1 está libre y s_3 está ocupado: debe fusionarse el bloque liberado con s_1 , lo que implica modificar campos tanto de s_2 como de s_1 .
- s_1 está ocupado y s_3 está libre: debe fusionarse el bloque liberado con s_3 , lo que implica modificar campos tanto de s_2 como de s_3 .
- s_1 y s_3 están libres: deben fusionarse los tres bloques en uno, lo que implica modificar campos tanto de s_1 como de s_3 (pero no de s_2).

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 6.4.2: 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 6.4.2). 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.

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: \rightarrow lista_nat

funiverso

universo PLANIFICADOR_MODULAR implementa PLANIFICADOR

usa ESTADO, NAT, BOOL

instancia CJT_ \in (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

fuelle, 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 := $\emptyset \cup \{ \text{ESTADO.inicial} \}$; encontrado := falso

mientras \neg encontrado $\wedge \neg$ COLA.vacía?(C) hacer {cola vacía \Rightarrow 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 \leq 6) $\wedge \neg$ 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 \neg (destino \in S) entonces

S := $S \cup \{ \text{destino} \}$; C := encola(C, <destino, inserta(l, i)>)

fsi

fsi

fsi

i := i + 1

fmientras

fmientras

si \neg encontrado entonces l := LISTA_INTERÉS.crea fsi

devuelve l

funiverso

Fig. 7.17: primera implementación del planificador.

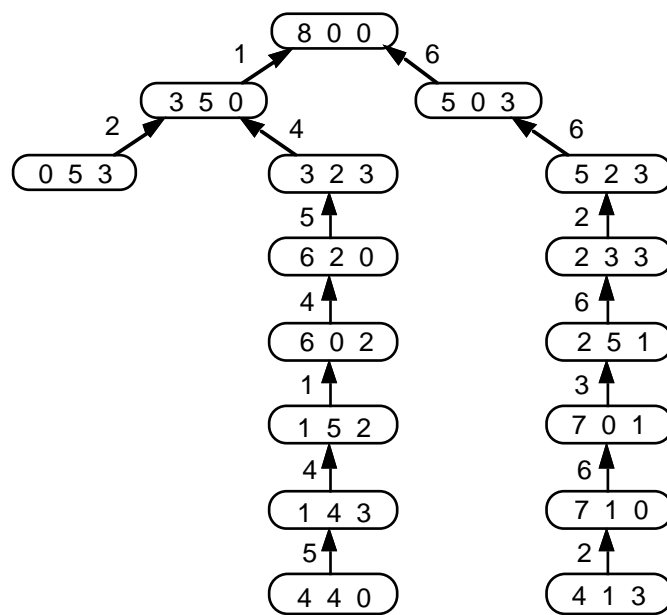


Fig. 7.18: árbol resultante de recorrer el grafo asociado al problema en amplitud.

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_ε (ELEM_=) implementado con CJT_ε_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 :=  $\emptyset \cup \{\text{ESTADO.inicial}\}$ ; encontrado := falso
    mientras  $\neg$  encontrado  $\wedge \neg$  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  $\leq$  6)  $\wedge \neg$  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  $\neg$  (destino  $\in$  S) entonces
              q := obtener_espacio
              si q = NULO entonces error
              si no q^ := <destino, i, p>; S := S  $\cup$  {destino}; C := encola(C, q)
            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  $\neq$  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 al 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: sin y con variantes abiertas de los TAD.

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 el hecho de 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.*

Estudieemos 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. 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 objetos⁵. En ese caso, no tiene sentido implementar la tabla por dispersión, porque la ganancia 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 supone que existe un universo en la biblioteca de componentes que implementa las listas

⁵ 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.

con punto de interés por punteros. El coste asintótico 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. También destacamos que esta operación destruye la pila; para evitar esta destrucción (y la consecuente duplicación previa) puede usarse la versión recorrible del TAD, ordenadamente según la política FIFO que caracteriza las pilas.

```

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 f tipo
  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(t) {cierra el bloque actual}
  función declara (t es ts; id es cadena; c es caracts) devuelve ts es
    si TABLA.consulta(PILA.cima(t), id)  $\neq$  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)  $\neq$  indef
  función consulta (t es ts; id es cadena) devuelve caracts es
  var v es caracts fsi
    v := indef
    mientras  $\neg$  PILA.vacía?(t)  $\wedge$  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*: $\rightarrow \text{cola}$, cola sin elementos; los n procesos no tienen acceso a ningún elemento
- inserta*: $\text{cola elem} \rightarrow \text{cola}$; 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*: $\text{cola nat} \rightarrow \text{elem}$, devuelve el valor de la cola accesible por un proceso dado; da error si el proceso no tiene acceso a ningún elemento
- avanza*: $\text{cola nat} \rightarrow \text{cola}$; 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*: $\text{cola nat} \rightarrow \text{cola}$, 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 módulos reusables; 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 función. 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
renombra tabla por cola_compartida
errores $\forall c \in \text{cola_compartida}; \forall k \in \text{nat}; \forall v \in \text{lista_elem}$
 $[k = 0 \vee k > n] \Rightarrow \text{asigna}(c, k, v), \text{consulta}(c, k, v), \text{borra}(c, k, v)$
ops
inserta: cola_compartida elem \rightarrow cola_compartida
primero: cola_compartida nat \rightarrow elem
avanza, borra: cola_compartida nat \rightarrow cola_compartida
privada inserta_todos: cola_compartida elem nat \rightarrow cola_compartida
privada borra_todos: cola_compartida nat nat \rightarrow cola_compartida
ecns $\forall c \in \text{cola_compartida}; \forall k \in \text{nat}; \forall v \in \text{lista_elem}$
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))
 $[k < n] \Rightarrow \text{inserta_todos}(c, v, k) =$
 $\text{inserta_todos}(\text{asigna}(c, k, \text{inserta_por_el_final}(\text{consulta}(c, k), v)), k+1)$
borra_todos(c, v, n) = asigna(c, n, borra_i_ésimo(consulta(c, n), v))
 $[k < n] \Rightarrow \text{borra_todos}(c, v, k) =$
 $\text{borra_todos}(\text{asigna}(c, k, \text{borra_i_ésimo}(\text{consulta}(c, k), v)), k+1)$
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, ni siquiera utilizando iteradores. 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):  $\forall k: 1 \leq k \leq n: 0 < \text{procesos}[k] \leq \text{long}(\text{cola})+1$ 
  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.col := LISTA_INTERÉS.crea
  devuelve c

  función inserta (c es cola_compartida; v es elem) devuelve cola_compartida es
  var k es nat fvar
    mientras  $\neg \text{final?}(c.col)$  hacer c.col := avanzar(c.col) fmientras
    c.col := LISTA_INTERÉS.inserta(c.col, v)
  devuelve c

  función primero (c es cola_compartida; k es nat) devuelve elem es
    c.col := principio(c.col)
    hacer c.procesos[k]-1 veces c.col := avanzar(c.col) fhacer
  devuelve actual(c.col) {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.col := principio(c.col)
    hacer c.procesos[k]-1 veces c.col := avanzar(c.col) fhacer
    c.col := borra(c.col) {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

universo

```

Fig. 7.23: implementación modular de la cola compartida.

Sin embargo, la supresión es todavía ineficiente. Primero, 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. Además, 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 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 (v. fig. 7.24).

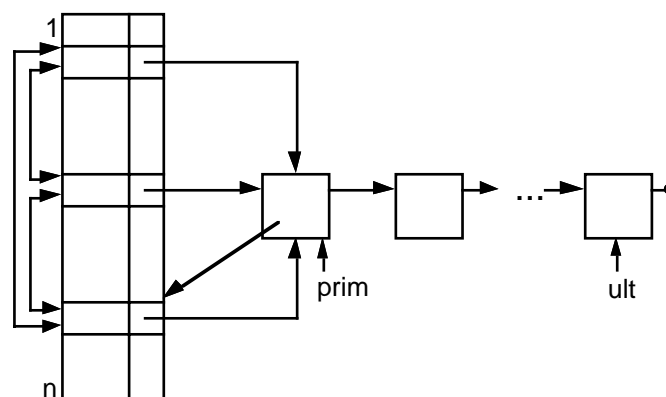


Fig. 7.24: esquema de una estructura de datos eficiente para la cola compartida.

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. 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.

Por último, comentar que podríamos haber usado una versión abierta del TAD de las colas, y los apuntadores se hubieran convertido en atajos. No lo hemos hecho porque en este caso igualmente había una parte de la estructura que debía implementarse directamente.

```

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.col.ult ∈ cadena(c.col.prim) ∧ c.col.ult^.enc = NULO ∧
    ∀p: p ∈ cadena(c.col.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.col.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.col.prim := obtener_espacio {para el fantasma}
    si c.col.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.col.prim> fpara todo
      c.procesos[1].ant := n; c.procesos[n].seg := 1
      {a continuación, se actualiza la cola}
      c.col.prim^.enc = NULO; c.col.prim^.pproc := 1; c.col.ult := c.col.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.col.ult^.enc := p; c.col.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(act); 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: → *tele*, emisora sin ítems

añade: *tele ítems* → *tele*, registra que la emisora compra un nuevo ítem; da error si ya existía algún ítem con el mismo nombre

cuál toca: *tele nat* → *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* → *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* → *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: \rightarrow tele

añade, privada emite_ítem: tele ítem \rightarrow tele

cuál_toca: tele nat \rightarrow ítem

emite: tele nat \rightarrow tele

da_todos: tele cadena \rightarrow lista_ítem_y_nat

privada está: tele ítem \rightarrow bool {comprueba si el ítem existe}

privada ninguno_emitido: tele nat \rightarrow bool {comprueba si se ha emitido algun ítem de la duración}

privada ninguno_sin_emitir: tele nat \rightarrow bool {comprueba si hay ítems sin emitir de la duración}

privada borra_ítem: tele ítem \rightarrow tele {elimina un ítem de la emisora}

privada lista_ítems: tele cadena \rightarrow lista_ítem {da todos los ítems de una compañía}

privada n_emisiones: tele ítem \rightarrow nat {cuenta el número de emisiones de un ítem}

privada cnt_emisiones: tele lista_ítem \rightarrow lista_ítem_y_nat {añade *n_emisiones* a cada ítem}

errores [está(t, v)] \Rightarrow añade(t, v); [\neg está(t, v)] \Rightarrow emite_ítem(t, v); cuál_toca(crea, k)

ecns

[duración(v) \neq duración(w)] \Rightarrow añade(añade(t, v), w) = añade(añade(t, w), v)

[duración(v) \neq duración(w)] \Rightarrow añade(emite_ítem(t, v), w) = emite_ítem(añade(t, w), v)

[duración(v) \neq duración(w)] \Rightarrow emite_ítem(emite_ítem(t, v), w) = emite_ítem(emite_ítem(t, w), v)

[duración(v) = k] \Rightarrow cuál_toca(añade(t, v), k) = v

[duración(v) \neq k] \Rightarrow cuál_toca(añade(t, v), k) = cuál_toca(t, k)

[duración(v) \neq k \vee n_emisiones(t, v) > 0] \Rightarrow cuál_toca(emite_ítem(t, v), k) = cuál_toca(t, k)

[duración(v) = k \wedge n_emisiones(t, v) = 0 \wedge \neg (ninguno_sin_emitir(borra_ítem(t, v), k) \wedge ninguno_emitido(t, k))] \Rightarrow cuál_toca(emite_ítem(t, v), k) = cuál_toca(borra_ítem(t, v), k)

[duración(v) = k \wedge n_emisiones(t, v) = 0 \wedge ninguno_sin_emitir(borra_ítem(t, v), k) \wedge ninguno_emitido(t, k)] \Rightarrow 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) \vee (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) \wedge (k \neq duración(v))

ninguno_sin_emitir(crea, k) = cierto

ninguno_sin_emitir(añade(t, v), k) = ninguno_sin_emitir(t, k) \wedge (k \neq 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)] \Rightarrow borra_ítem(añade(t, v), w) = t

[nombre(v) \neq nombre(w)] \Rightarrow borra_ítem(añade(t, v), w) = añade(borra_ítem(t, w), v)

[nombre(v) = nombre(w)] \Rightarrow borra_ítem(emite_ítem(t, v), w) = borra_ítem(t, w)

[nombre(v) \neq nombre(w)] \Rightarrow 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)]  $\Rightarrow$  lista_ítems(añade(t, v), c) = inserta(lista_ítems(t, c), v)
[c  $\neq$  compañía(v)]  $\Rightarrow$  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)]  $\Rightarrow$  n_emisiones(emite(t, k), v) = n_emisiones(t, v) + 1
[nombre(v)  $\neq$  cual_toca(t, k)]  $\Rightarrow$  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, exigen 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* demanda 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 (en cuanto a instancias, renombramientos, etc.) es intrínseca a la complejidad del problema y difícil de 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, UTILIDADES_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_ORD_POR_ÁRBOL_AVL_POR_PUNTEROS(ELEM_<_=
donde elem es cadena, < es CADENA.<, = es CADENA.=
renombra tabla por compañía
instancia TABLA(ELEM_=
implementada con LISTA_ORD_POR_PUNTEROS(B es ELEM_<_=
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 almacén 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; 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):

$$\forall v: v \in \text{ítem}: \text{está?}(t.H, \text{nombre}(v)) \Leftrightarrow$$

$$v \in t.D[\text{duración}(v)].E \vee v \in t.D[\text{duración}(v)].I \Leftrightarrow$$

$$\text{está?}(\text{consulta}(t.C, \text{compañía}(v)), \text{nombre}(v)) \wedge$$

$$\forall k: 0 \leq k \leq \text{MÁX.val}: \forall x: x \in t.D[k].I: \text{consulta}(t.H, x).nemisiones = 0 \wedge$$

$$\forall x: x \in t.D[k].E: \text{consulta}(t.H, x).nemisiones > 0 \wedge$$

$$\forall c: c \in \text{cadena} \wedge \text{está?}(t.C, c):$$

$$\forall x: x \in \text{cadena} \wedge \text{está?}(\text{consulta}(t.C, c), x): \text{compañía}(\text{consulta}(t.H, x).v) = c$$

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) 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 empyla como inédito en los ítems de su duración}
    t.D[duraciones(v)].inéditos := empyla(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 debido a la buena eficiencia asintótica de las operaciones. 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 adoptamos variantes abiertas de los TAD.

El uso de variantes abiertas tiene una primera consecuencia: la desaparición de la hemeroteca, es decir, del conjunto de pares <ítem, número de emisiones>. La razón de ser de este componente de la estructura era permitir acceso eficiente dado el identificador del ítem que residía en los componentes. Pero ahora es posible almacenar estos pares en cualquiera de estos otros componentes y ofrecer atajos hacia ellos, posibilitando acceso lo más rápido posible. Si guardamos los pares en los conjuntos ordenados de las compañías, las pilas y colas mantendrán atajos a los elementos de este conjunto. El espacio ahorrado es mucho: no sólo se cambian claves por atajos (que serán punteros en la implementación elegida), sino que dada la signatura del TAD (no hay supresiones de elementos) y la estructura de datos elegida para implementar los conjuntos ordenados (árboles AVL, que no efectúan movimientos físicos de los elementos), el problema de la obsolescencia de los atajos no existe y podemos elegir una implementación del TAD abierto más simple y que exige el mínimo espacio posible.

Sólo hay un punto a controlar. Dado que las operaciones sobre atajos necesitan pasar como parámetro el valor del TAD abierto sobre el que se accede, en las pilas y colas necesitamos identificar a qué conjunto ordenado debe aplicarse el atajo. Por ello, elegimos también una variante abierta de la tabla de compañías, y guardaremos pares de atajos en las pilas y colas: un atajo a la tabla de compañías para obtener el conjunto de ítems de esa compañía, y un atajo al ítem que se aplicará sobre el conjunto obtenido mediante el atajo anterior. Este pequeño incremento de espacio se compensa sobradamente con el ahorro mencionado.

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, que se simplifica incluso dada la desaparición del conjunto de ítems; por ejemplo, no es necesario instanciar la tabla de dispersión. El invariante se mantiene reducido, lo que también abunda en la simplicidad de la solución. En la tabla de compañías, se supone que la operación *último_atajo* devuelve el último atajo referenciado en una operación de asignación, independientemente de si la compañía existía o no. Se utilizan operaciones de modificar por atajo en los conjuntos y tablas de la estructura.

universo TELE_BRINCO_MODULAR_Y_EFICIENTE (MÁX es VAL_NAT)
implementa TELE_BRINCO(MÁX es VAL_NAT)
usa ÍTEM, CADENA, NAT, BOOL
instancia CONJUNTO_ORDENADO_ABIERTO(ELEM_CJT_<_=
implementado con CJT_ORD_ABIERTO_POR_ÁRBOL_AVL_PUNTEROS(ELEM_CJT_<_=
donde clave es cadena, elem es tupla v es ítem; nemisiones es nat ftupla
id es nombre(_v), <_< es nombre(_v) CADENA.< nombre(_v)
< es nombre(_v) CADENA.= nombre(_v), esp es <ÍTEM.esp, 0>
renombra tabla por compañía, atajo_conjunto por @ítem
instancia TABLA_ABIERTA(ELEM_<_<, ELEM_ESP)
implem con LISTA_ORD_ABIERTA_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_ABIERTO.crea
renombra tabla por compañías, atajo_tabla por @compañía
instancia PILA(ELEM) implementada con PILA_PUNTEROS(ELEM)
donde elem es tupla v es @ítem; c es @compañía ftupla
renombra pila por inéditos
instancia COLA(ELEM) implementada con COLA_PUNTEROS(ELEM)
donde elem es tupla v es @ítem; c es @compañía ftupla
renombra cola por emitidos
tipo privado duraciones es
vector [de 0 a MÁX.val] de tupla E es emitidos; I es inéditos ftupla
ftipo
tipo tele es tupla C es compañías; D es duraciones ftupla ftipo
invariante (t es tele):
 $\forall pc, pv: pc \in @compañía \wedge pv \in @ítem:$
 $\langle pc, pv \rangle \in t.D[duración(v)].E \vee \langle pc, pv \rangle \in t.D[duración(v)].I \Leftrightarrow$
 $definido?(t.C, pc) \wedge definido?(dato(t.C, pc), pv) \wedge$
 $\forall k: 0 \leq k \leq MÁX.val: \forall x: x \in t.D[k].I: dato(dato(t.C, x.c), x.v).nemisiones = 0 \wedge$
 $\forall x: x \in t.D[k].E: dato(dato(t.C, x.c), x.v).nemisiones > 0 \wedge$
 $\forall c: c \in cadena \wedge está?(t.C, c):$
 $\forall n: n \in cadena \wedge está?(consulta(t.C, c), n):$
 $compañía(consulta(consulta(t.C, c), n).v) = c$
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

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

```

función añade (t es tele; v es ítem) devuelve tele es
var caux es compañía; pc es @compañía; pv es @ítem fvar
    caux := consulta(t.C, compañía(v))
    si está?(caux, nombre(v)) entonces error {el ítem ya existe}
    si no
        {se añade a los ítems de la compañía y se obtienen los atajos}
        caux := añade(caux, <v, 0>)
        pv := último_atajo(caux)
        t.C := asigna(t.C, compañía(v), caux)
        pc := último_atajo(t.C)
        {a continuación se empila el apuntador como inédito en los ítems de su duración}
        t.D[duraciones(v)].inéditos := empila(t.D[duraciones(v)].inéditos, <pc, pv>)
    fsi
devuelve t

función cuál_toca (t es tele; k es nat) devuelve ítem es
var pc es @compañía; pv es @ítem 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 <pc, pv> := cabeza(t.D[k].emitidos) {el emitido hace más tiempo}
        fsi
        si no <pc, pv> := cima(t.D[k].inéditos) {el último añadido}
        fsi
devuelve dato(dato(t.C, pc), pv).v

función emite (t es tele; k es nat) devuelve tele es
var pc es @compañía; pv es @ítem; 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 <pc, pv> := cabeza(t.D[k].emitidos); t.D[k].emitidos := desencola(t.D[k].emitidos)
        fsi
        si no <pc, pv> := cima(t.D[k].inéditos); t.D[k].inéditos := desempila(t.D[k].inéditos)
        fsi
        t.D[k].emitidos := encola(t.D[k].emitidos, <pc, pv>)
        {a continuación, se registra la emisión del ítem}
        <v, n> := dato(dato(t.C, pc), pv)
        t.C := modifica_por_atajo(t.C, pc, modifica_por_atajo(dato(t.C, pc), pv, <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 ordenada 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 cuatro 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 cartas 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 → *pared*, crea una pared de una determinada longitud

añade: pared nat nat → *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 → *bool*, dice si un mueble de una anchura dada puede colocarse en una posición dada

nre: pared → *nat*, da el número de muebles en la pared

elem: pared nat → *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 → *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: $\text{pared nat} \rightarrow \text{pared}$, 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: $\rightarrow \text{progs}$, estructura vacía
nuevo_programa: $\text{progs programa} \rightarrow \text{prog}$, añade un nuevo programa; si ya hay uno con el mismo nombre, da error
borra: $\text{progs cadena} \rightarrow \text{progs}$, borra un programa dado su nombre; si no lo encuentra, da error
purga: $\text{progs fecha} \rightarrow \text{progs}$, borra todos los programas anteriores a una fecha dada
todos: $\text{progs} \rightarrow \text{lista_programas}$, da la lista de todos los programas en orden alfabético
obsoletos: $\text{progs fecha} \rightarrow \text{lista_programas}$, da la lista de programas anteriores a una fecha dada en cualquier orden
tema: $\text{progs cadena} \rightarrow \text{lista_programas}$, 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: $\rightarrow \text{meteorología}$, crea la estructura sin regiones
añadir: $\text{meteorología región clima} \rightarrow \text{meteorología}$, 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: → *central*, estructura sin estaciones, tramos ni trenes

añ_estación: *central estación* → *central*, añade una estación; da error si la estación ya existía

añ_vía: *central estación estación nat nat nat* → *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* → *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 → *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 → *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 algún 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 contruidos 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: → *agencia*, forma una estructura sin compañías ni viajes ni ciudades

ofrece: agencia cadena compañía nat → *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 → *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 → *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 favorezca 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* → *agencia*, añade un nuevo paquete; da error si ya había algún paquete con el mismo nombre

reserva: *agencia cadena nat* → *agencia*; la agencia hace una reserva de las plazas especificadas en el paquete dado; da error si no hay ningún paquete con este nombre o no hay suficientes sitios libres

no_llenos: *agencia* → *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 los 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 ningún 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 implementarlo, 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: tele, emisora sin vídeos

añadir: tele vídeo → *tele*, registra que la emisora compra un nuevo vídeo; no es necesario comprobar repeticiones

cual_toca?: tele nat → *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 → *tele*, toma nota de que el vídeo seleccionado por la operación anterior con la duración dada ya ha sido emitido

borra_malos: tele → *tele*, elimina los 10 vídeos que se han emitido menos veces

emitir_top_ten: tele nat → *<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: tele, inicializa la estructura, determinando cuáles son los períodos, de qué duración y su coste

comprar: tele nat anuncio → *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 → *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 → *lista_nat*, da la lista de períodos que se han comprado para ese anuncio
emitir: tele nat → *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 2011, 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 eurofuncionarios. 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 alcalde 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 usadas, *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 receta* → *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 operaciones, *ident*, *región* y *calidad*. Las operaciones son:

crea: $\rightarrow carta$, estructura vacía

ofrece: $carta\ plato\ vino\ nat \rightarrow carta$, 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: $carta\ cadena \rightarrow carta$, 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: $carta\ cadena \rightarrow lista_vinos$, 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: $carta \rightarrow vino$, 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, entusiasmado 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: \rightarrow *barroco*, estructura vacía

compra: *barroco obra* \rightarrow *barroco*, añade una nueva obra a la colección; si la obra ya existe, da error

pincha: *barroco cadena* \rightarrow *barroco*, registra la audición de una obra determinada, identificada por la cadena correspondiente; da error si la obra no existe

por_autor: *barroco cadena* \rightarrow *lista_obras*, devuelve la lista de todas las obras disponibles de un compositor determinado

menos_oídas: *barroco* \rightarrow *lista_obras*, devuelve la lista de las diez obras menos pinchadas de la colección, para su eventual eliminación con

elimina: *barroco cadena* \rightarrow *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* \rightarrow *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* \rightarrow *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 no tiene un máximo conocido ya que, al parecer 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.

Bibliografía

- [ADJ78] J.A. Goguen, J.W. Thatcher, E.G. Wagner. "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types". En *Current Trends in Programming Methodology*, Vol. IV, Prentice Hall, 1978.
- [AHU83] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [Bal93] J.L. Balcázar. *Programación Metódica*. McGraw-Hill, 1993.
- [BrB97] G. Brassard, P. Bratley. *Fundamentos de Algoritmia*. Ed. Prentice Hall, 1997.
- [CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [EhM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification*, Vol. 1. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.
- [EhM90] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification*, Vol. 2. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1990.
- [GoB91] G.H. Gonnet, R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 2ª edición, 1991.
- [HoS94] E. Horowitz, S. Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, 4ª edición, 1994.
- [Knu68] D.E. Knuth. *The Art of Computer Programming*, Vol. 1. Addison-Wesley, 1968.
- [Knu73] D.E. Knuth. *The Art of Computer Programming*, Vol. 3. Addison-Wesley, 1973.
- [LiG86] B.H. Liskov, J.V. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.
- [Mar86] J.J. Martin. *Data Types and Data Structures*. Prentice-Hall, 1986.
- [Meh84] K. Mehlhorn. *Data Structures and Algorithms*, vols. 1 y 2. Springer-Verlag, 1984.
- [Peñ98] R. Peña. *Diseño de Programas (2ª edición)*. Prentice Hall, 1998.
- [Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*. Regional Conference Series in Applied Mathematics (SIAM), Philadelphia, Pennsylvania, 1983.
- [TeA86] A.M. Tenenbaum, M.J. Augenstein. *Data Structures using PASCAL*. Prentice-Hall, 2ª edición, 1986.
- [vAP89] J.J. van Amstel, J.A.A.M. Poirters. *The Design of Data Structures and Algorithms*. Prentice Hall and Academic Service, 1989.
- [Wir86] N. Wirth. *Algorithms and Data Structures*. Prentice-Hall, 1986.

Índice temático

A

Abstracción	20, 23
Acción (en Merlí)	95
Acoplamiento de estructuras de datos	138
Adyacencia	350
Álgebra	
cociente de términos	40
de términos	31
inicial	40
objeto matemático	25, 30
respecto una signatura	30
Algoritmo	
Brent	288
compleción (Knuth-Bendix)	79
Dijkstra	370
Floyd	376
Kruskal	333, 384
ordenación por inserción	122
ordenación por montículo	238
Prim	381
voraz	381
Altura (de un árbol)	198
Antecesor (en un grafo)	361
Apiñamiento	282, 290
primario	283
Apuntador	98
de sitio libre	98
externo	139
Árbol	195

2-3, B, B*, B+	303
AVL	303
binario	196, 201
de búsqueda	221, 297
casi-completo	213, 233
cerrado por prefijo	196
compacto	196
completo	213
enhebrado	224
de Fibonacci	303
equilibrado	303
etiquetado	196
de expansión	379, 380
de expansión de coste mínimo	380
general	196
libre	380
<i>n</i> -ario	196
parcialmente ordenado	233
con punto de interés	196, 202
<i>quadtree</i>	246
con raíz	380
Arco	346
Aridad	
de un árbol	199
de un símbolo	28
Arista	v. <i>arco</i>
Ascendente (en un árbol)	199
Atajo	138
Axioma (especificación ecuacional)	34

B

Basura	178
Biblioteca de módulos reusables ...	24, 125
Bosque	200
Búsqueda	
auto-organizativa	255
de caminos mínimos	369
dicotómica (binaria)	256
por interpolación	256
lineal	256

con movimiento al inicio255
 por transposición.....255

C

Cadenav. *secuencia*
 predicado *cadena*.....171
 de reubicación282
 Camino
 en un árbol.....198
 de un clave (de dispersión)278
 en un grafo350
 mínimo369
 Campo (de tupla).....91
 Categoría74
 Ciclo351
 Clave249
 indefinida.....249
 invasora278
 Colisión258
 Cola
 circular159
 compartida423
 prioritaria231, 237
 TAD158
 Complejidad.....v. *eficiencia*
 Componente (fuertemente) conexo....350
 Compresión de caminos342
 Conexión (en un grafo).....350
 Congruencia inducida por las ecuaciones39
 Conjunto
 de base30
 TAD252
 Constante
 orden de magnitud116
 valor30
 Constructor de tipo
 por enumeración.....91
 puntero173
 tupla91
 vector91

Costev. *eficiencia*
 Cuadrado (método).....263
 Cubeta258
 Cursor98

D

Deducción ecuacional.....77
 Desbordamiento
 en el cálculo262
 en una tabla274
 Descendiente
 en un árbol.....199
 en un grafo361
 Desequilibrio305
 DD305, 308
 DI.....307, 310
 Digrafov. *grafo dirigido*
 Diseño
 descendente22
 de estructuras de datos397
 modular (con TAD).....22
 Dispersión
 concepto258
 función258, 259
 incremental.....274
 organizaciones.....270
 perfecta260
 valor de dispersión258
 División (método).....263

E

Ecuación34
 condicional49
 impurificadora.....46
 parte derecha/izquierda35
 de recurrencia119
 Eficiencia23, 108
 amortizada345

en el caso peor.....	110, 113
Elemento	
definido.....	249
distinguido.....	162
fantasma (centinela).....	170
Encadenamiento	168
Enlace	v. <i>encadenamiento</i>
Ensayo.....	278
Enriquecimiento	67
Especificación	20
algebraica (ecuacional)	25, 34
método de construcción.....	47
parametrizada (genérica).....	69
pre-post.....	97
Esquemas de programación	131
de búsqueda.....	131, 163
de recorrido.....	131, 163
Estructura de datos.....	25
funcional.....	249
lineal.....	151
de partición.....	332
Etiqueta.....	196, 346
de coste nulo.....	369
Evaluador de expresiones.....	399
Extremidad (de un camino)	350

F

Factor	
de carga (árbol).....	219
de carga (tabla).....	292
de equilibrio.....	304
Forma	
de un árbol.....	196
normal	78
Función	
de abstracción.....	100
de Ackerman.....	345
de asignación de variables	33
de evaluación de términos	33
de dispersión	258, 259

en Merlí	95
parcial	249, 250
de redispersión.....	278
de representación.....	100
TAD	249
total	249, 250
universal.....	263
Funtor	75

G

Género	v. <i>tipo</i>
Grado v. <i>aridad</i>	
Grafo.....	320
acíclico	351
bipartito	346
completo	348
denso.....	348
dirigido	346
disperso	348
etiquetado.....	346
no dirigido.....	346
no etiquetado	346
TAD.....	346

H

Hermano (en un árbol).....	199
Hijo (en un árbol).....	199
derecho, izquierdo.....	202
izquierdo, hermano derecho.....	214
Hoja	198

I

Identificador.....	249
Implementación	20
corrección.....	98
eficiencia	108

lenguaje de implementación	98	Morfismo	32
universo de implementación..89, 90		Multilista (de grado dos).....	324
Índice		Multilista de adyacencia	358
a tabla.....	249		
a vector.....	91	N	
Instancia	69	Nivel (en un árbol)	198
parcial	73	Nodo	195, 197, 346
Invariante		Notación asintótica	
de un bucle.....	93	O	112
de una representación.....	100	Ω	112
Invasor	v. <i>clave invasora</i>	Θ	113
Isomorfismo	32	Notación infix	400
Iterador.....	128	Notació postfix (polaca).....	399
		NULO.....	174
L			
<i>liberar_espacio</i>	174	O	
Lista		Obsolescencia (de atajos).....	142
de adyacencia.....	355	<i>obtener_espacio</i>	174
auto-organizativa.....	255	Ocultación de símbolos.....	67
circular	186	Operación	30
doblemente encadenada.....	187	auxiliar (oculta, privada)	50
encadenada.....	143	consultora.....	46
ordenada.....	161	constructora.....	46
con punto de interés	162	constructora generadora.....	46
		modificadora	46
M		Orden de magnitud	113
Marca	143		
Matriz		P	
de adyacencia.....	352	Padre (en un árbol).....	199
dispersa.....	324	Parámetro	
Memoria dinámica.....	174, 405	de entrada y/o de salida	95
<i>MFS</i> et.....	332	formal	69
Modelo		real.....	70
de un TAD	25, 43	Parametrización	69
inicial	37	Paso de parámetros	70
Módulo.....	23	Pila	
Montículo	212, 235		
de Fibonacci	370		

de sitios libres.....	146, 169
TAD	151
Plegamiento-desplegamiento.....	263
Posición	
de elemento	138
de vector	91
Postcondición	97
Precondición	97
Prioridad.....	231
Programación dinámica.....	376
Profundidad (en un árbol)	v. <i>nivel</i>
Puntero.....	173
Punto de acceso (en iterador).....	128

R

Raíz.....	195, 198
Rama.....	199
Recolección de basura.....	178
Recorrido	
en anchura.....	219, 228, 361, 364
inorden.....	220
por niveles	219, 228
en ordenación topológica.....	360, 365
postorden.....	220
preorden	220
en profundidad	360
en profundidad hacia atrás	362
Redispersión	278
Reescritura	78, 247
Referencia colgada	177
Regla de escritura	78
Relación	
binaria (TAD).....	319
binaria etiquetada.....	319
de equivalencia (TAD)	320, 332
etiquetada	320
de igualdad.....	100
m:n	v. <i>binaria</i>
Renombramiento de símbolos.....	68
Representante	

canónico.....	41, 46
de clase (de equivalencia).....	340
Representación de tipo	90, 91
encadenada.....	166, 168
circular.....	186
doblemente.....	187
secuencial.....	154, 166
Reubicación.....	143, 145
Reusabilidad.....	41, 46
Robin Hood (método).....	289
Rotación.....	306

S

S-aplicación.....	32
S-conjunto	27
Secuencia.....	151
Semántica de un TAD	43
de comportamiento	44
final	43
inicial	43
laxa.....	44
SIG-álgebra.....	30
Signatura.....	26
Símbolo	
de constante.....	28
de operación.....	28
Sinónimo.....	258
Sistema de reescritura.....	78
Canónico.....	79
Confluente	79
Noetheriano.....	79
Subárbol.....	195, 199
Subcamino.....	350
Subciclo.....	351
Subgrafo.....	350
Suma ponderada	261

T

Tabla	
de dispersión	258
de símbolos	60, 420
TAD	249
TAD	v. <i>Tipo Abstracto de Datos</i>
Teorema ecuacional	77
Teoría ecuacional	77
Término	28
sobre SIG	28
Tipo	
auxiliar (oculto, privado)	50
de datos	19, 25
de interés	66
Tipo Abstracto de Datos (TAD)	19
abierto	146
parcialmente abierto	139
recorrible	128
recorrible ordenadamente	128
totalmente abierto	v. <i>abierto</i>
Tupla	91
variante	91

U

<i>Union-find set</i>	332
Universo	26
de caracterización	69
de definición (de especificación)	35
genérico (parametrizado)	70
de implementación	89, 90
Uso	45, 66

V

Valor de dispersión	258
Variable	29
de control (bucle)	93
global	95

local	95
-------------	----

Vector	91
--------------	----

Vértice	346
---------------	-----

intermedio (en un camino)	350
---------------------------------	-----

Z

Zona de excedentes	274
--------------------------	-----

Zona principal	274
----------------------	-----

Índice de universos

ALFABETO.....	57	COORDENADAS	69
ARBOL_BINARIO.....	203	DIGRAFO_ETIQ	349
ARBOL_BINARIO_DE_BUSQUEDA.....	30	DIGRAFO_ETIQ_LISTAS.....	355
ARBOL_BINARIO_ENC_1_VECTOR.....	210	DIGRAFO_ETIQ_MATRIZ.....	353
ARBOL_BINARIO_ENC_PUNTEROS.....	205	DIGRAFO_ETIQ_MULTILISTAS.....	359
ARBOL_GENERAL.....	201	DIVISION.....	268
ARBOL_GENERAL_POR_BINARIO	216	DOS_ENTEROS	69, 72
BOOL	26, 35	ELEM	70
CADENA.....	57, 76	ELEM_ =	71
CJT.....	70	ELEM_ <	76
CJT_ \in	71	ELEM_ <_ =	130
CJT_ \in ACOTADO.....	127	ELEM_DISP_CONV.....	266
CJT_ \in ACOTADO_IND.....	137	ELEM_2_ESP_ =	280
CJT_ \in ACOT. IND_PARC_ABIERTO.....	141	ELEM_ESP	251
CJT_ \in ACOTADO_POR_VECT.....	99, 127	ELEM_ESP_ <_ =_ +	369
CJT_ \in ACOTADO_RECORRIBLE	129	ELEM_ORDENADO.....	332
CJT_ \in ACOT. REC. POR_VECT.....	133	ENTERO.....	42
CJT_ \in ACOTADO_REC. ORD.....	131	FUNCIONES_F	266
CJT_ \in ACOT. REC. ORD_VECT_1.....	134	FUNCIONES_G.....	267
CJT_ \in ACOT. REC. ORD_VECT_2.....	135	FUNCION_TOTAL	251
CJT_RECORRIBLE	253	FUNCION_TOTAL_RECO. ORD	253
CLAVE_DISPERSION	268	LISTA_INTERES	165
CLAVE_DISPERSION_LINEAL.....	286	LISTA_INTERES_ENC.....	172, 182
CLAVE_REDISPERSIÓN.....	280	LISTA_INTERES_ENC_PUNT.....	176, 184
COLA	159	LISTA_INTERES_SEC	166
COLA_CIRCULAR.....	161	MULTILISTA_TODO_CIRCULAR.....	328
COLA_PRIORITARIA.....	232	NAT	27, 45, 52
COLA_PRIOR. POR_MONTICULO.....	236	PAR	72
COMPOSICION_F_Y_G.....	268	PILA	153
CONJUNTO	252	PILA_SEC	155
		RACIONALES	69, 75
		REDISPERION_DOBLE.....	287
		REDISPERION_LINEAL.....	286
		REDISP_LIN. SUMA_POND_Y_DIV.....	286
		REDISP_DOB. SUMA_POND_Y_DIV.....	287
		RELACION.....	322

RELACION_DE_EQUIVALENCIA.....	335
RELACION_DE_EQUIV._ARB.....	343
RELACION_DE_EQUIV._LINEAL	338
RELACION_ETIQUETADA.....	323
SUMA_POND	267
SUMA_POND_Y_DIV.....	269
TABLA_DIRECTA	276
TABLA_IND_PUNTEROS.....	272
TABLA_ABIERTA.....	280
VAL_NAT	73
VECTOR.....	157