

GINÉS GARCÍA MATEOS
JOAQUÍN CERVERA LÓPEZ
NORBERTO MARÍN PÉREZ
DOMINGO GIMÉNEZ CÁNOVAS

**ALGORITMOS Y
ESTRUCTURAS DE DATOS
VOLUMEN I
ESTRUCTURAS DE DATOS**



ICE



Índice general

1. Problemas, programas, estructuras y algoritmos	1
1.1. Resolución de problemas	3
1.1.1. Análisis de requisitos del problema	3
1.1.2. Modelado del problema y algoritmos abstractos	4
1.1.3. Diseño de la solución	5
1.1.4. Implementación del diseño	6
1.1.5. Verificación y evaluación de la solución	7
1.2. Tipos de datos	7
1.2.1. Definición de tipo de datos, tipo abstracto y estructura	8
1.2.2. Tipos de tipos	10
1.2.3. Repaso de tipos y pseudolenguaje de definición	12
1.3. Algoritmos y algorítmica	14
1.3.1. Definición y propiedades de algoritmo	15
1.3.2. Análisis de algoritmos	17
1.3.3. Diseño de algoritmos	21
1.3.4. Descripción del pseudocódigo utilizado	25
1.4. Consejos para una buena programación	26
1.4.1. Importancia del análisis y diseño previos	26
1.4.2. Modularidad: encapsulación y ocultamiento	28
1.4.3. Otros consejos	29
Ejercicios propuestos	31
Cuestiones de autoevaluación	31
Referencias bibliográficas	32
2. Abstracciones y especificaciones	35
2.1. Las abstracciones en programación	37
2.1.1. Diseño mediante abstracciones	38
2.1.2. Mecanismos de abstracción: especificación y parametrización	39
2.1.3. Tipos de abstracciones: funcional, de datos e iteradores	40
2.1.4. Mecanismos de abstracción en los lenguajes de programación	42
2.2. Especificaciones informales	47
2.2.1. Especificación informal de abstracciones funcionales	47
2.2.2. Especificación informal de abstracciones de datos	48
2.2.3. Especificación informal de abstracciones de iteradores	51
2.3. Especificaciones formales algebraicas	52

2.3.1. Propiedades, notación y ventajas de las especificaciones formales	53
2.3.2. Especificaciones algebraicas o axiomáticas	54
2.3.3. Taxonomía de las operaciones de un TAD	56
2.3.4. Completitud y corrección de la especificación	57
2.3.5. Reducción de expresiones algebraicas	59
2.4. Especificaciones formales constructivas	61
2.4.1. Precondiciones y postcondiciones	61
2.4.2. Especificación como contrato de una operación	62
2.4.3. Necesidad de un modelo subyacente	64
2.4.4. Ejecución de especificaciones constructivas	67
Ejercicios resueltos	68
Ejercicios propuestos	75
Cuestiones de autoevaluación	77
Referencias bibliográficas	78
3. Conjuntos y diccionarios	79
3.1. Los tipos abstractos conjunto y diccionario	81
3.1.1. El concepto matemático de conjunto	82
3.1.2. El tipo abstracto diccionario	83
3.1.3. Representación mediante arrays de booleanos	85
3.1.4. Representación mediante listas de elementos	87
3.2. Tablas de dispersión	90
3.2.1. La dispersión y los sinónimos	91
3.2.2. Dispersión abierta	92
3.2.3. Dispersión cerrada	94
3.2.4. Funciones de dispersión	100
3.2.5. Estrategias de redispersión	102
3.3. Combinando estructuras de datos	104
3.3.1. Estructuras de datos múltiples	105
3.3.2. La relación muchos a muchos	109
3.3.3. Estructuras de listas múltiples	113
Ejercicios resueltos	116
Ejercicios propuestos	122
Cuestiones de autoevaluación	125
Referencias bibliográficas	126
4. Representación de conjuntos mediante árboles	127
4.1. Árboles trie	129
4.1.1. Árboles de prefijos	130
4.1.2. Representación de nodos trie	132
4.1.3. Búsqueda e inserción de palabras en un trie	136
4.1.4. Eficiencia y comparación entre estructuras	138
4.2. Relaciones de equivalencia	142
4.2.1. Representaciones sencillas de relaciones de equivalencia	145
4.2.2. Representación mediante punteros al padre	147

4.2.3. Equilibrado y compresión de caminos	149
4.3. Árboles de búsqueda balanceados	154
4.3.1. Árboles binarios de búsqueda	154
4.3.2. El peor caso de árbol AVL	157
4.3.3. Rotaciones simples y dobles sobre AVL	160
4.3.4. Inserción en un árbol AVL	162
4.3.5. Eliminación en un árbol AVL	165
4.4. Árboles B	168
4.4.1. Árboles de búsqueda no binarios	168
4.4.2. Inserción en un árbol B	172
4.4.3. Eliminación en un árbol B	172
4.4.4. Análisis de eficiencia de los árboles B	174
Ejercicios resueltos	177
Ejercicios propuestos	184
Cuestiones de autoevaluación	187
Referencias bibliográficas	189
5. Grafos	191
5.1. Definiciones y terminología de grafos	193
5.1.1. Definición y tipos de grafos	194
5.1.2. Terminología de la teoría de grafos	196
5.1.3. Árboles, grafos y multigrafos	198
5.1.4. Especificación del tipo grafo	201
5.2. Estructuras de representación de grafos	203
5.2.1. Representación con matrices de adyacencia	204
5.2.2. Representación con listas de adyacencia	206
5.2.3. Comparación entre estructuras de representación	208
5.3. Recorridos sobre grafos	211
5.3.1. Búsqueda primero en profundidad	211
5.3.2. Búsqueda primero en anchura	215
5.4. Árboles de expansión de coste mínimo	217
5.4.1. Algoritmo de Prim	218
5.4.2. Algoritmo de Kruskal	221
5.5. Problemas de caminos mínimos	223
5.5.1. Caminos mínimos empezando por un origen	224
5.5.2. Caminos más cortos entre todos los vértices	229
5.6. Algoritmos sobre grafos dirigidos	234
5.6.1. Componentes fuertemente conexos	235
5.6.2. Grafos dirigidos acíclicos	239
5.6.3. Flujo máximo en redes	243
5.7. Algoritmos sobre grafos no dirigidos	247
5.7.1. Puntos de articulación y componentes biconexos	248
5.7.2. Circuitos de Euler	251
5.8. Otros problemas con grafos	253
5.8.1. Ciclos hamiltonianos	254

5.8.2. Problema del viajante	255
5.8.3. Coloración de grafos	256
5.8.4. Isomorfismo de grafos	257
Ejercicios resueltos	259
Ejercicios propuestos	266
Cuestiones de autoevaluación	270
Referencias bibliográficas	271
A. Tutorial de ANSI C	275
A.1. Introducción	277
A.2. Primeros pasos	279
A.3. Tipos de datos elementales y variables	280
A.3.1. Tipo char	281
A.3.2. Tipo int	281
A.3.3. Tipos enumerados	282
A.3.4. Tipos float y double	282
A.3.5. Las conversiones de tipo	282
A.3.6. Modos de almacenamiento de las variables	283
A.4. Operadores	285
A.4.1. Aritméticos	285
A.4.2. Incrementales	285
A.4.3. Relacionales	286
A.4.4. Lógicos	286
A.4.5. De manejo de bits	287
A.4.6. De asignación	287
A.4.7. De dirección e indirección	287
A.4.8. Otros operadores	288
A.4.9. Expresiones: precedencia y asociatividad	288
A.5. Sentencias de control de flujo	290
A.5.1. Sentencias de selección	290
A.5.2. Sentencias de iteración	293
A.6. Tipos de datos agregados y punteros	295
A.6.1. Punteros	296
A.6.2. Arrays	297
A.6.3. Relación entre arrays y punteros	299
A.6.4. Estructuras y uniones	301
A.6.5. Definición de tipos	303
A.7. Funciones	303
A.7.1. Función main con argumentos	305
A.7.2. Funciones como parámetros	305
A.8. Funciones de E/S	306
A.8.1. printf	306
A.8.2. scanf	307
A.8.3. putchar y getchar	308
A.8.4. Manejo de ficheros	308

A.9. Gestión dinámica de la memoria	310
A.10. El preprocesador de C	312
A.11. Programas con varios ficheros	313
Ejercicios Propuestos	315
Referencias bibliográficas	317
B. Introducción básica a C++	319
B.1. Pequeños cambios respecto a C	320
B.1.1. Variables y tipos	320
B.1.2. Funciones	321
B.1.3. Operadores new y delete	323
B.1.4. Gestión de E/S	324
B.2. Introducción a la POO	324
B.3. Clases y objetos en C++, primeros pasos	325
B.4. Programa ejemplo: TAD cadena de caracteres	327
B.5. Plantillas	333
B.6. Excepciones	335
Ejercicios Propuestos	337
Referencias bibliográficas	338
C. Práctica 1: Análisis y diseño de estructuras de datos	339
C.1. Enunciado	339
C.2. Análisis de los requisitos de la aplicación	341
C.3. Diseño del programa	343
C.4. Implementación del diseño	348
C.5. Validación y verificación	349
C.6. Informe del desarrollo	350

Capítulo 1

Problemas, programas, estructuras y algoritmos

El proceso de resolución de problemas, en programación, está compuesto por una serie de pasos que se aplican de forma metódica. Empieza con el análisis de las características del problema, continúa con el diseño de una solución, la implementación del diseño y termina con la verificación y evaluación del programa resultante. Los dos ejes básicos en los que se articula el desarrollo de un programa son las estructuras de datos y los algoritmos. Las estructuras de datos determinan la forma de almacenar la información necesaria para resolver el problema. Los algoritmos manipulan esa información, para producir unos datos de salida a partir de unos datos de entrada. El objetivo último de la algorítmica es encontrar la mejor forma de resolver los problemas, en términos de eficiencia y de calidad del software.

Objetivos del capítulo:

- Entender el desarrollo de programas como un proceso metódico e ingenieril, formado por una serie de etapas con distintos niveles de abstracción, frente a la idea de la programación como arte.
- Tomar conciencia de la importancia de realizar siempre un análisis y diseño previos del problema, como pasos anteriores a la implementación en un lenguaje de programación.
- Motivar el estudio de los algoritmos y las estructuras de datos, como una disciplina fundamental de la informática.
- Repasar algunos de los principales conceptos de programación, que el alumno debe conocer de cursos previos como: tipos de datos, estructuras de datos, tipos abstractos, algoritmos, complejidad algorítmica y eficiencia.
- Entender la importancia del análisis de algoritmos, el objetivo del análisis, los factores que influyen y el tipo de notaciones utilizadas para su estudio.
- Distinguir entre algoritmos y esquemas algorítmicos, presentando brevemente los principales esquemas algorítmicos que serán estudiados.

Contenido del capítulo:

1.1.	Resolución de problemas	3
1.1.1.	Análisis de requisitos del problema	3
1.1.2.	Modelado del problema y algoritmos abstractos	4
1.1.3.	Diseño de la solución	5
1.1.4.	Implementación del diseño	6
1.1.5.	Verificación y evaluación de la solución	7
1.2.	Tipos de datos	7
1.2.1.	Definición de tipo de datos, tipo abstracto y estructura	8
1.2.2.	Tipos de tipos	10
1.2.3.	Repaso de tipos y pseudolenguaje de definición	12
1.3.	Algoritmos y algorítmica	14
1.3.1.	Definición y propiedades de algoritmo	15
1.3.2.	Ánalisis de algoritmos	17
1.3.3.	Diseño de algoritmos	21
1.3.4.	Descripción del pseudocódigo utilizado	25
1.4.	Consejos para una buena programación	26
1.4.1.	Importancia del análisis y diseño previos	26
1.4.2.	Modularidad: encapsulación y ocultamiento	28
1.4.3.	Otros consejos	29
	Ejercicios propuestos	31
	Cuestiones de autoevaluación	31
	Referencias bibliográficas	32

1.1. Resolución de problemas

La habilidad de programar ordenadores constituye una de las bases necesarias de todo informático. Pero, por encima del nivel de programación, la característica fundamental de un informático es su capacidad para **resolver problemas**. La resolución de problemas informáticos en la vida real implica otras muchas más habilidades que únicamente saber programar –habilidades que distinguen a un ingeniero de un simple programador. Implica comprender y analizar las necesidades de los problemas, saber modelarlos de forma abstracta diferenciando lo importante de lo irrelevante, disponer de una variada batería de herramientas conceptuales que se puedan aplicar en su resolución, trasladar un diseño abstracto a un lenguaje y entorno concretos y, finalmente, ser capaz de evaluar la corrección, prestaciones y posibles inconvenientes de la solución planteada.

Las herramientas que surgen en el desarrollo de programas son esencialmente de dos clases: **estructuras de datos** y **algoritmos**. Las estructuras de datos representan la parte estática de la solución al problema, el componente almacenado, donde se encuentran los datos de entrada, de salida y los necesarios para posibles cálculos intermedios. Los algoritmos representan la parte dinámica del sistema, el componente que manipula los datos para obtener la solución. Algoritmos y estructuras de datos se relacionan de forma muy estrecha: las estructuras de datos son manipuladas mediante algoritmos que añaden o modifican valores en las mismas; y cualquier algoritmo necesita manejar datos que estarán almacenados en cierta estructura. La idea se puede resumir en la fórmula magistral de Niklaus Wirth: **Algoritmos + Estructuras de datos = Programas**.

En ciertos ámbitos de aplicación predominará la componente algorítmica –como, por ejemplo, en problemas de optimización y cálculo numérico– y en otros la componente de estructuras –como en entornos de bases de datos y sistemas de información–, pero cualquier aplicación requerirá siempre de ambos. Esta dualidad aparece en el nivel abstracto: un tipo abstracto está formado por un dominio de valores (estructura abstracta) y un conjunto de operaciones (algoritmos abstractos). Y la dualidad se refleja también en el nivel de implementación: un módulo (en lenguajes estructurados) o una clase (en lenguajes orientados a objetos) están compuestos por una estructura de atributos o variables, y una serie de procedimientos de manipulación de los anteriores.

Antes de entrar de lleno en el estudio de los algoritmos y las estructuras de datos, vamos a analizar los pasos que constituyen el proceso de resolución de problemas.

1.1.1. Análisis de requisitos del problema

El proceso de resolución de problemas parte siempre de un **problema**, de un enunciado más o menos claro que alguien plantea porque le vendría bien que estuviera resuelto¹. El primer paso es la comprensión del problema, entender las características y peculiaridades de lo que se necesita. Este **análisis de los requisitos** del problema puede ser, de por sí, una de las grandes dificultades; sobre todo en grandes aplicaciones, donde los documentos de requisitos son ambiguos, incompletos y contradictorios.

¹ Esta afirmación puede parecer irrelevante por lo obvia. Pero no debemos perder de vista el objetivo último de la utilidad práctica, en el estudio de los algoritmos y las estructuras de datos.

El análisis debe producir como resultado un **modelo abstracto** del problema. El modelo abstracto es un modelo conceptual, una abstracción del problema, que reside exclusivamente en la mente del individuo y donde se desechan todas las cuestiones irrelevantes para la resolución del problema. Supongamos, por ejemplo, los dos siguientes problemas, con los cuales trabajaremos en el resto de los puntos.

Ejemplo 1.1 El problema de las cifras. Dado un conjunto de seis números enteros, que pueden ser del 1 al 10, ó 25, 50, 75 ó 100, encontrar la forma de conseguir otro entero dado entre 100 y 999, combinando los números de partida con las operaciones de suma, resta, producto y división entera. Cada uno de los seis números iniciales sólo se puede usar una vez.

Ejemplo 1.2 El problema de detección de caras humanas. Dada una imagen en color en un formato cualquiera (por ejemplo, bmp, jpeg o gif) encontrar el número de caras humanas presentes en la misma y la posición de cada una de ellas.

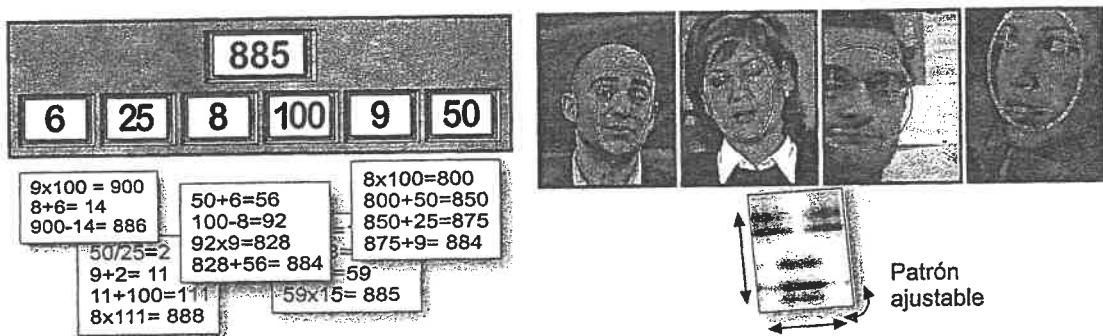


Figura 1.1: Ejemplos del problema de las cifras (izquierda) y detección de caras (derecha).

Ambos enunciados representan categorías de problemas muy distintas. El problema 1.1 tiene un enunciado más o menos claro, aunque cabrían algunas dudas. Es de tipo matemático, se puede modelar formalmente y es previsible que exista un algoritmo adecuado. El problema 1.2 tiene un enunciado más corto pero mucho más ambiguo, no está claro exactamente lo que se pide. Es más, incluso teniéndolo bien claro, el problema parece más adecuado para ser resuelto de cabeza por un humano, pero difícil de implementar en un ordenador. Aun así, ambos problemas tienen algo en común: son de interés, así que necesitamos resolverlos de la mejor forma que podamos.

1.1.2. Modelado del problema y algoritmos abstractos

El primer paso sería crear un **modelo abstracto**, en el que nos quedamos con lo esencial del problema. Normalmente, este modelo se crea a través de una **analogía** con algo conocido previamente. Por ejemplo, para enseñar a un alumno inexperto lo que es un algoritmo (concepto para él desconocido) se utiliza la analogía con una receta de cocina

(concepto conocido, esperemos), y las estructuras de datos se asimilan con la disposición de armarios, cajones y recipientes donde se guardan los ingredientes y el bizcocho resultante.

Ejemplo 1.1 (cont.) Modelo abstracto. Para el problema de las cifras, el modelo abstracto puede prescindir del hecho de que los números sean seis, de que estén en ciertos rangos y que se trabaje sólo con enteros. Una solución sería una expresión matemática –usando los números de partida– que produce el número buscado. Entonces, el problema se podría resolver generando todas las posibles expresiones que contengan los n números iniciales, o menos, y quedándonos la que obtenga un resultado más próximo al buscado.

Pero el modelo de un problema no tiene por qué ser único. Otra persona podría interpretarlo como un conjunto de cajas, que se combinan entre sí. Ahora tendríamos un modelo “temporal”: tenemos que buscar dos cajas que combinándolas nos den el objetivo o un valor próximo. Si no llegamos al resultado, buscar otra caja y así sucesivamente. La idea resultante sería similar a lo que aparece en la figura 1.1, abajo a la izquierda.

Ejemplo 1.2 (cont.) Modelo abstracto. En el problema de detección de caras, el modelo abstracto elimina el hecho de que las imágenes tengan distinto formato o incluso que los objetos buscados sean caras humanas. Un posible modelo sería un problema de patrones –patrones como los de un sastre– que admiten cierta variabilidad: se pueden mover, estirar, cambiar el tamaño, rotar, etc. El problema consistiría en buscar esos patrones en las imágenes, en todos los sitios que se pueda. Grosso modo, un algoritmo podría consistir en generar todas las posibles colocaciones, rotaciones y tamaños de los patrones y quedarnos con las que “encajen” con lo que hay en la imagen de partida.

1.1.3. Diseño de la solución

Una vez que tengamos un modelo completo y adecuado, significará que hemos entendido bien el problema. Como hemos visto en los ejemplos, el modelo conlleva también un algoritmo informal, que no es más que una vaga idea de cómo resolver el problema. El siguiente paso de refinamiento consistiría en diseñar una solución. Usando otra analogía, el **diseño de un programa** es equivalente a los planos de un arquitecto: el diseño describe el aspecto que tendrá el programa, los materiales que se necesitan y dónde y cómo colocarlos. El diseño es siempre un paso previo a la implementación, pero que se olvida muy a menudo por los programadores primerizos. Un ingeniero informático debería ser tan consciente del error de programar sin partir de un buen diseño previo, como lo es un arquitecto del fallo de hacer un puente sin tener antes los planos.

El diseño de un programa consta de dos clases de elementos: tipos de datos y algoritmos. Los tipos de datos usados a nivel de diseño son **tipos abstractos**, en los cuales lo importante son las operaciones que ofrecen y no la representación en memoria. Los algoritmos son una versión refinada de los algoritmos abstractos del paso anterior. No obstante, son aún **algoritmos en pseudocódigo**, donde se dan cosas por supuestas.

Definir los tipos de datos para almacenar la solución suele ser mucho más difícil que definirlos para los datos de entrada. Además, modificaciones en los algoritmos pueden requerir cambios en los tipos de datos y viceversa.

Ejemplo 1.1 (cont.) Diseño de la solución. En el problema de las cifras, los tipos de datos para la entrada podrían ser una tabla de 6 enteros y otro entero para el

número buscado. Pero ¿cómo representar la solución? En este caso, la solución es una serie de operaciones sobre los números del array. Cada operación podría ser una tupla ($indice_1, indice_2, operacion$), que significa: “operar el número de $indice_1$ en el array con el de $indice_2$ usando $operacion$ ”. La representación de la solución sería una lista de estas tuplas. Con esta representación, el algoritmo debería tener en cuenta que $indice_1$ e $indice_2$ ya están usados. Además aparece otro nuevo número, que puede usarse con posterioridad. Entonces, puede ser necesario definir otros tipos o modificar los existentes, por ejemplo haciendo que la tupla signifique: “operar el número ..., almacenando el resultado en $indice_1$ y colocando un 0 en $indice_2$ ”.

Ejemplo 1.2 (cont.) Diseño de la solución. En el problema de detección de caras, la entrada sería una imagen, que se puede representar con matrices de enteros, de cierto ancho y alto. Además, hay otro dato *oculto*, el modelo de lo que se considera una cara. ¿Cómo se puede modelar la forma de todas las posibles caras? ¿Qué información se debería almacenar? La representación no es nada trivial. Por ejemplo, una solución sencilla sería tener un conjunto variado de imágenes de caras de ejemplo. La representación de la solución debe indicar el número de caras y la posición de cada una; por ejemplo centro, (x_i, y_i) , escala, s_i , y rotación, α_i , de cada cara. Podríamos usar una lista de tuplas de tipo $(x_i, y_i, s_i, \alpha_i)$. Con todo esto, habría que diseñar un algoritmo en pseudolenguaje que pruebe todos los valores de los parámetros anteriores, y diga dónde se encuentra una cara.

1.1.4. Implementación del diseño

El siguiente paso en la resolución del problema es la implementación. La implementación parte del diseño previo, que indica qué cosas se deben programar, cómo se estructura la solución del problema, dónde colocar cada funcionalidad y qué se espera en concreto de cada parte en la que se ha descompuesto la solución. Esta descripción de lo que debe hacer cada parte es lo que se conoce como la **especificación**. Una implementación será correcta si cumple su especificación.

La dualidad tipos/algoritmos del diseño se traslada en la implementación a estructuras/algoritmos. Para cada uno de los tipos de datos del diseño, se debe elegir una **estructura de datos** adecuada. Por ejemplo, los dos ejemplos de problemas analizados necesitan listas. Una lista se puede representar mediante una variedad de estructuras: listas enlazadas con punteros o cursores, mediante celdas adyacentes en un array, enlazadas simple o doblemente, con nodos cabecera o no, etc. La elección debe seguir los criterios de eficiencia –en cuanto a tiempo y a uso de memoria– que se hayan especificado para el problema. En cada caso, la estructura más adecuada podrá ser una u otra.

Por otro lado, la **implementación de los algoritmos** también se puede ver como una serie de tomas de decisiones, para trasladar un algoritmo en pseudocódigo a un lenguaje concreto. Por el ejemplo, si se necesita repetir un cálculo varias veces, se puede usar un bucle *for*, un *while*, se puede usar recursividad e incluso se puede poner varias veces el mismo código si el número de ejecuciones es fijo².

Realmente, la implementación de estructuras de datos y algoritmos no va por separado. Ambas son simultáneas. De hecho, así ocurre con los mecanismos de los lenguajes

²Algo nada aconsejable, de todos modos.

de programación –módulos o clases– en los que se asocia cada estructura con los algoritmos que la manejan. La organización temporal de la implementación es, más bien, por funcionalidades: cuestiones de interface con el usuario, accesos a disco, funcionalidad matemática, etc.

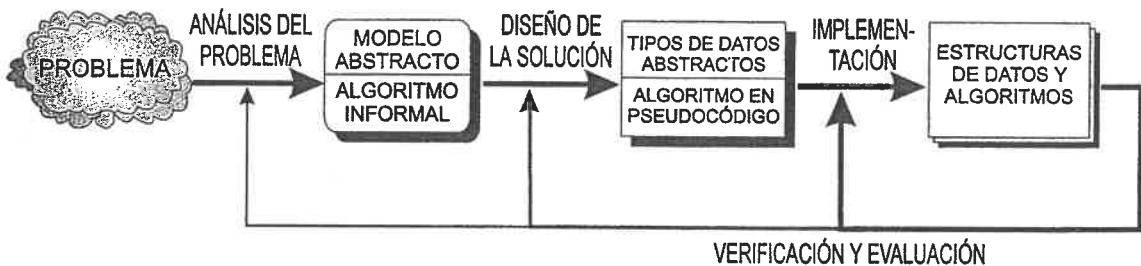


Figura 1.2: Esquema del proceso cíclico de desarrollo de software.

1.1.5. Verificación y evaluación de la solución

La resolución de un problema no acaba con la implementación de un programa³. Los programas deberían ser comprobados de forma exhaustiva, verificando que se ajustan a los objetivos deseados, obtienen los resultados correctos sin provocar fallos de ejecución. En la práctica, hablamos de un **proceso cíclico de desarrollo**: implementar, verificar, corregir la implementación, volver a verificar, volver a corregir, etcétera. En cierto momento, puede que tengamos que cambiar el diseño, e incluso puede que lo que esté fallando sea el modelo abstracto. Cuanto más atrás tengamos que volver, más tiempo habremos perdido haciendo cosas que luego resultan inútiles.

La evaluación de los programas incluye no sólo la corrección del resultado sino también la medición de las prestaciones, lo que normalmente se denomina el **análisis de eficiencia**. La eficiencia es una proporción entre los resultados obtenidos y los recursos consumidos. Fundamentalmente, un programa consume recursos de tiempo y de memoria. Aunque usualmente hablamos de “análisis de algoritmos”, tanto las estructuras de datos como los algoritmos influyen en el consumo de recursos de un programa.

Incluso antes de llegar a la implementación de un programa, es posible y adecuado hacer el análisis del algoritmo en pseudocódigo. Este análisis será una previsión de la eficiencia que obtendría la solución diseñada si la implementáramos. Se trata, por lo tanto, de un estudio teórico. Cuando el estudio se realiza sobre una implementación concreta, hablamos normalmente de estudios experimentales.

1.2. Tipos de datos

A modo de breve repaso, vamos a ver en esta sección las definiciones de tipo de datos, tipo abstracto, estructura de datos, las distintas clases de tipos y algunos de los principales

³O no debería. Desafortunadamente, estamos tan acostumbrados a ver programas que se cuelgan, como coches que fallan justo al salir del garaje, ¡exactamente por lo mismo: falta de pruebas!

tipos de datos usados en programación. Se supone que el alumno ya está familiarizado con todos estos conceptos. Es más, sería un interesante ejercicio pararse en este punto, cerrar el libro y tratar recordar todo lo que se pueda de los anteriores temas.

1.2.1. Definición de tipo de datos, tipo abstracto y estructura

Possiblemente, la primera duda del programador no ingeniero sería: ¿qué necesidad hay de tantos términos para referirse a lo mismo? Los conceptos de tipo abstracto de datos, tipo de datos y estructura de datos no son, en absoluto, redundantes ni incoherentes. Como ya hemos visto, aparecen en los distintos niveles de desarrollo. Un tipo abstracto es un concepto de diseño y por lo tanto previo e independiente de cualquier implementación. Un tipo de datos es un concepto de programación y se puede ver como la realización o materialización de un tipo abstracto. La estructura de datos es la organización o disposición en memoria de la información almacenada para cierto tipo de datos. Veamos, por partes, las definiciones.

Definición 1.1 Un Tipo Abstracto de Datos (TAD) está compuesto por un dominio abstracto de valores y un conjunto de operaciones definidas sobre ese dominio, con un comportamiento específico.

Por ejemplo, los booleanos constituyen un TAD, formado por el dominio de valores *{verdadero, falso}* y las operaciones sobre ese dominio: NO, Y, O, XOR, IMPLICA, etc. Decimos que el dominio de valores es *abstracto* porque no se le supone ninguna representación ni ningún significado. Es decir, el valor *verdadero* no tiene por qué ser un 1, ni debe almacenarse necesariamente en un bit. Pero, es más, ¿qué significa *verdadero*? Es un concepto hueco, vacío, sin ningún significado explícito. El único significado le viene dado implícitamente por su relación con las operaciones del tipo. Por ejemplo, *verdadero Y cualquier cosa* es siempre igual a *cualquier cosa*.

En el otro extremo de complejidad, otro ejemplo de TAD podrían ser las ventanas de un entorno Windows o XWindows. En este caso, el dominio de valores estaría compuesto por el conjunto –prácticamente infinito– de todas las ventanas en sus distintas formas, tamaños, colores y contenidos. Las operaciones sobre el dominio serían del tipo: crear ventana, mover, cambiar de tamaño, añadir botones, etc.

Las operaciones del TAD son abstractas, en el sentido de que sabemos lo que hacen pero no cómo lo hacen. A la hora de programar el TAD, esto da lugar a un principio conocido como **ocultación de la implementación**: lo importante es que se calcule el resultado esperado y no cómo se calcule. Volveremos a insistir en esto más adelante.

En cuanto que el TAD es un concepto abstracto, reside exclusivamente como un ente etéreo en la mente del programador. La realización del TAD en un lenguaje concreto, en cierto entorno de programación y por determinado programador, es lo que da lugar a un tipo de datos, ahora sin el apellido “abstracto”.

Definición 1.2 El tipo de datos de una variable, un parámetro o una expresión, determina el conjunto de valores que puede tomar esa variable, parámetro o expresión, y las operaciones que se pueden aplicar sobre el mismo.

El tipo de datos es un concepto de programación, de implementación, mientras que el TAD es un concepto matemático previo a la programación. Un tipo de datos debe ajustarse al comportamiento esperado de un TAD. Por ejemplo, el tipo `integer` de Pascal o el tipo `int` de C son dos tipos de datos, que corresponden al TAD de los enteros, \mathbb{Z} .

Además de los tipos de datos elementales existentes en un lenguaje de programación, los lenguajes suelen ofrecer mecanismos para que el usuario se defina sus propios tipos. Estos mecanismos pueden ser más o menos avanzados. En lenguajes como C o Pascal, la definición del tipo indica los atributos que se almacenan para las variables de ese tipo. Las operaciones se definen por separado. Por ejemplo, en C una pila de enteros sería:

```
struct
    PilaEnteros
{
    int datos[];
    int tope, maximo;
};
```

En lenguajes orientados a objetos, como C++ o Java, los tipos definidos por el usuario se llaman **clases**, y su definición incluye tanto los atributos almacenados como las operaciones sobre los mismos. Por ejemplo, en C++ la definición de las pilas podría ser la siguiente:

```
class
    PilaEnteros
{
    private:           // Atributos
        int datos[];
        int tope, maximo;
    public:            // Operaciones
        PilaEnteros (int max);
        Push (int valor);
        int Pop();
};
```

Volveremos a tratar los mecanismos de definición de tipos en el siguiente capítulo.

En la programación de un tipo de datos surgen dos cuestiones: cómo almacenar en memoria los valores del dominio y cómo programar las operaciones del tipo de datos. La primera cuestión implica diseñar una estructura de datos para el tipo. La segunda está relacionada con la implementación de las operaciones de manipulación y, evidentemente, estará en función de la primera.

Definición 1.3 La estructura de datos de un tipo de datos es la disposición en memoria de los datos necesarios para almacenar los valores de ese tipo.

Por ejemplo, para representar las pilas podemos usar una estructura de arrays como la que aparece antes, o una lista enlazada de enteros, o una lista de arrays de enteros, o un array de listas de enteros, y así sucesivamente. Por lo tanto, un tipo de datos puede ser implementado utilizando diferentes estructuras.

De forma análoga, una misma estructura de datos puede corresponder a diferentes tipos. Por ejemplo, una estructura de arrays puede usarse para implementar listas, pilas o colas de tamaño limitado. Será la forma de actuar de las operaciones la que determine si el array almacena una lista, una pila o una cola.

La utilización de estructuras de datos adecuadas para cada tipo de datos es una cuestión fundamental en el desarrollo de programas. La elección de una estructura tendrá efectos no sólo en la memoria utilizada por el programa, sino también en el tiempo de ejecución de los algoritmos. Por ejemplo, la representación de colas con arrays puede producir desperdicio de memoria, ya que el tamaño del array debe ser el tamaño de la máxima ocupación prevista de la cola. En una representación con punteros se podrá usar menos memoria, pero posiblemente el tiempo de ejecución será sensiblemente mayor.

1.2.2. Tipos de tipos

Dentro de este apartado vamos a hacer un repaso de terminología, clases y propiedades relacionadas con los tipos de datos.

Podemos hacer una primera clasificación de los tipos teniendo en cuenta si el tipo es un tipo estándar del lenguaje o no. Encontramos dos categorías:

- **Tipos primitivos o elementales.** Son los que están definidos en el lenguaje de programación. Por ejemplo, en C son tipos primitivos los tipos int, long, char y float, entre otros.
- **Tipos definidos por el usuario.** Son todos los tipos no primitivos, definidos por la misma persona que los usa, o bien por otro programador.

Idealmente, un buen lenguaje debería hacer transparente el uso de tipos de una u otra categoría. Es decir, al usar una variable de cierto tipo T debería ser lo mismo que T sea un tipo primitivo o definido por el usuario. Ambos deberían ser conceptualmente equivalentes. Pero esto no siempre ocurre. Por ejemplo, en C es posible hacer asignaciones entre tipos primitivos, pero no entre ciertos tipos definidos por el usuario. La asignación “`a = b;`” funcionará o no, dependiendo de que los tipos de a y b sean primitivos o no.

Por otro lado, podemos hacer otra clasificación –según el número de valores almacenados– en tipos simples y compuestos:

- **Tipo simple.** Una variable de un tipo simple contiene un único valor en cierto instante. Por ejemplo, los enteros, reales, caracteres y booleanos son tipos simples.
- **Tipo compuesto.** Un tipo compuesto es aquel que se forma por la unión de varios tipos simples o compuestos. Por lo tanto, una variable de tipo compuesto contiene varios valores de tipo simple o compuesto.

Los lenguajes de programación suelen ofrecer distintos mecanismos de composición para crear tipos compuestos. Los más habituales son los arrays y los registros. Un array contiene una serie de posiciones consecutivas de variables del mismo tipo, cada una identificada con un índice dentro del rango del array. Los registros se definen enumerando los campos que forman el tipo compuesto, cada uno de los cuales tiene un nombre y un

tipo (que, a su vez, puede ser simple o compuesto). Las **clases** de lenguajes orientados a objetos también se pueden ver como una manera de crear tipos compuestos, en la cual se pueden añadir atributos y operaciones al tipo.

Normalmente los tipos simples son tipos elementales y los tipos compuestos son definidos por el usuario. No obstante, también pueden existir tipos compuestos definidos en el lenguaje de programación, y tipos simples definidos por el usuario. Un mecanismo que permite al usuario definir nuevos tipos simples son los **enumerados**. Un enumerado se define listando de forma completa el dominio del tipo. Por ejemplo, el tipo **Sexo** se puede definir como un tipo enumerado con el dominio {masculino, femenino}.

Un **tipo contenedor**, o **colección**, es un tipo compuesto que puede almacenar un número indefinido de valores de otro tipo cualquiera. Por ejemplo, una lista de enteros o un array de **Sexo** son tipos contenedores. Un registro con dos enteros no sería un contenedor.

En general, interesa que los tipos contenedores puedan almacenar valores de otro tipo cualquiera; por ejemplo, que las listas puedan ser de enteros, de reales, de registros o de cualquier otra cosa. Un tipo se dice que es **genérico** o **parametrizado** si su significado depende de otro tipo que es pasado como parámetro. Por ejemplo, una lista genérica tendría la forma **Lista[T]**, donde **T** es un parámetro que indica el tipo de los objetos almacenados. Podríamos tener una variable **a** de tipo **Lista[entero]** y otra variable **b** de tipo **Lista[real]**; estos casos concretos se llaman **instanciaciones** del tipo genérico.

Realmente son muy pocos los lenguajes que permiten la definición de tipos parametrizados. Entre ellos se encuentra **C++**, que permite parametrizar un tipo mediante el uso de plantillas **template**. Por ejemplo, un tipo genérico **Pila[T]** sería:

```
template <class T>
class
    Pila
    {
        private:           // Atributos
            T datos[];
            int tope, maximo;
        public:            // Operaciones
            Pila (int max);
            Push (T valor);
            T Pop();
    };
```

En la parte de prácticas se profundizará más en el uso de plantillas como una manera de crear tipos genéricos.

Finalmente, podemos hacer una distinción entre tipos mutables e inmutables. Decimos que un tipo de datos es **inmutable** cuando los valores del tipo no cambian después de haber sido creados. El tipo será **mutable** si los valores pueden cambiar. ¿Qué implicaciones tiene que un tipo sea mutable o inmutable? Supongamos que queremos definir el tipo **Lista[T]**. Si el tipo se define con una representación mutable, podemos incluir operaciones que añaden o suprimen elementos de la lista.

operación Inserta (var lista: Lista[T]; elemento: T)
operación SuprimePrimero (var lista: Lista[T])

Donde *lista* es un parámetro que se puede modificar dentro de las operaciones. Pero si el tipo es inmutable no sería posible que un parámetro de entrada se modificara. En ese caso, las operaciones de inserción y eliminación deberían devolver una nueva lista, con el elemento correspondiente insertado o eliminado.

operación Inserta (*lista*: Lista[T]; *elemento*: T): Lista[T]

operación SuprimePrimero (*lista*: Lista[T]): Lista[T]

Según nos interese, definiremos los tipos como mutables o inmutables. Normalmente, la mayoría de los tipos serán mutables, aunque en algunos casos nos interesaría usar tipos inmutables: la suposición de que los valores del tipo no cambian puede simplificar, o hacer más eficiente, la implementación de la estructura de datos para el tipo.

1.2.3. Repaso de tipos y pseudolenguaje de definición

La mayoría de los lenguajes ofrecen al programador un conjunto similar de tipos de datos simples (enteros, reales, booleanos, etc.), con las operaciones necesarias para manejarlos. Estos tipos tienen una correspondencia con los tipos manejados por el procesador, por lo que su implementación es muy eficiente. Vamos a ver los que nos podemos encontrar más habitualmente. Para cada uno de ellos, indicaremos también la notación que se usará en el resto del libro para referirse a ese tipo, cuando usemos el pseudolenguaje⁴.

- **Enteros.** Con signo o sin signo, con precisión simple, doble o reducida (un byte). En nuestro pseudocódigo supondremos un único tipo entero, con las operaciones habituales.
- **Reales.** Con precisión simple o doble. Para los números reales se suelen usar representaciones mantisa-expONENTE. Cuando lo usemos, pondremos el tipo real.
- **Caracteres.** Lo denotaremos como carácter.
- **Cadenas.** En algunos lenguajes las cadenas de caracteres son tipos elementales. En otros, como en C, las cadenas no son un tipo elemental sino que son simplemente arrays de caracteres. En nuestro caso, suponemos que tenemos el tipo cadena con operaciones para concatenar o imprimir por pantalla.
- **Booleanos.** Igual que antes, no existen en C –donde se usan enteros en su lugar–, aunque sí en C++ y Pascal. Supondremos que tenemos el tipo booleano.

En cuanto a los tipos contenedores (listas, pilas, colas, árboles, etc.), normalmente no forman parte del lenguaje de programación, sino que se definen en librerías de utilidades que se pueden incluir en un programa o no. No obstante, sí que se suelen incluir mecanismos de composición de tipos como arrays, registros y tipos enumerados. En nuestro pseudolenguaje tenemos la posibilidad de definir nuevos tipos, mediante una cláusula **tipo**. Además, permitimos que se definan tipos parametrizados. Por ejemplo, podemos tener los siguientes tipos:

⁴Este pseudolenguaje de definición de tipos sólo indica la estructura de datos del tipo. Las operaciones no son descritas aparte. Al implementar los tipos –sobre todo usando un lenguaje orientado a objetos– no hay que olvidar que las estructuras y las operaciones deben ir juntas, en el mismo módulo o clase.

tipo

DiasMeses = array [1..12] de entero

Pila[T] = registro

datos: array [1..MAXIMO] de T

tope, maximo: entero

finregistro

Sexo = enumerado (masculino, femenino)

Para los arrays, se indica entre corchetes, $[min..max]$, el rango mínimo y máximo de los índices del array⁵. Para los registros, se listan sus atributos, cada uno con su nombre y tipo. Y, por otro lado, los enumerados se forman listando los valores del dominio.

En algunos sitios usaremos también registros con variantes. Un **registro con variantes** es un registro que contiene un campo *especial*, en función del cual una variable podrá tener unos atributos u otros (pero nunca ambos a la vez). Por ejemplo, si en una aplicación que almacena empleados de una empresa, queremos almacenar diferente información para los hombres y las mujeres podemos tener algo como lo siguiente:

tipo

Empleado = registro

nombre: cadena

edad: entero

según sexo: Sexo

masculino: (dirección: cadena; sueldo: entero)

femenino: (salario: entero; teléfono: entero)

finsegún

finregistro

En cuanto a los tipos colecciones, además de los arrays daremos por supuesto que disponemos de los siguientes:

- **Listas.** Tendremos listas parametrizadas, **Lista[T]**, y que guardan una posición actual. De esta forma, la inserción, eliminación, etc., se hacen siempre sobre la posición actual. Por conveniencia, en algunos sitios no se darán por supuestas las listas.
- **Pilas.** Una pila es una lista LIFO: last in, first out (último en entrar, primero en salir). Suponemos el tipo genérico **Pila[T]**, con las operaciones push, pop y tope.
- **Colas.** Una cola es una lista FIFO: first in, first out (primero en entrar, primero en salir). Suponemos el tipo genérico **Cola[T]**, con las operaciones meter, sacar y cabeza.

Por último, pero no menos importante, tenemos el tipo de datos **puntero**. Considerado como un tipo *non-grato* por algunos autores⁶ por tratarse de un concepto cercano al bajo nivel, lo cierto es que los punteros son esenciales en la definición de estructuras de datos. Los punteros son un tipo parametrizado. Una variable de tipo **Puntero[T]** contiene una referencia, o dirección de memoria, donde se almacena una variable de tipo **T**.

⁵Ojo, hay que tener en cuenta que en lenguajes como C/C++ el índice mínimo es siempre 0, así que sólo se indica el tamaño del array.

⁶Por ejemplo, según C.A.R. Hoare, "su introducción en los lenguajes de alto nivel fue un paso atrás del que puede que nunca nos recuperemos". En el lenguaje Java se decidió no permitir el uso de punteros, aunque dispone de referencias, que vienen a ser lo mismo.

En nuestro pseudolenguaje suponemos que tenemos el tipo Puntero[T] con las siguientes operaciones y valores especiales:

- **Operador de indirección.** Si a es de tipo Puntero[T], entonces la variable apuntada por a se obtiene con el operador de indirección flecha, \uparrow , esto es: $a\uparrow$ es de tipo T.
- **Obtención de la dirección.** Si t es una variable de tipo T, entonces podemos obtener la dirección de esa variable con la función PunteroA($t: T$), que devuelve un puntero de tipo Puntero[T].
- **Puntero nulo.** Existe un valor especial del tipo Puntero[T], el valor predefinido NULO, que significa que el puntero no tiene una referencia válida o no ha sido inicializado.
- **Creación de una nueva variable.** Para crear una nueva variable apuntada por un puntero, suponemos la función genérica Nuevo($T: \text{Tipo}$), que devuelve un Puntero[T] que referencia a una nueva variable creada. Esta variable se borrará con la función Borrar($a: \text{Puntero}[T]$).
- **Comparación.** Suponemos que hay definidos operadores de comparación de igualdad y desigualdad entre punteros ($=, \neq$).

No consideramos otras operaciones sobre punteros, al contrario de lo que ocurre con C/C++, donde los punteros son considerados como enteros, pudiendo aplicar sumas, restas, etc. Por esta razón, entre otras, decimos que lenguajes como C/C++ son lenguajes débilmente tipados: existen pocas restricciones en la asignación, mezcla y manipulación de tipos distintos. Como contraposición, los lenguajes fuertemente tipados, como Pascal, tienen reglas más estrictas en el sistema de compatibilidad de tipos. En la práctica, esta flexibilidad de C/C++ le proporciona una gran potencia pero es fuente de numerosos errores de programación.

1.3. Algoritmos y algorítmica

Como ya hemos visto, los algoritmos junto con las estructuras de datos constituyen los dos elementos imprescindibles en el proceso de resolución de problemas. Los primeros definen el componente manipulador y los segundos el componente almacenado, y se combinan estrechamente para crear soluciones a los problemas.

No está de más recordar que la historia de los algoritmos es mucho anterior a la aparición de los ordenadores. De hecho, podríamos datar la aparición de los algoritmos al primer momento en que los seres humanos se plantearon resolver problemas de forma genérica. Históricamente uno de los primeros algoritmos inventados, para la resolución de problemas de tipo matemático, es el algoritmo de **Euclides** de Alejandría, Egipto, propuesto alrededor del año 300 a.C. Euclides propone el siguiente método para calcular el máximo común divisor de dos números enteros.

Ejemplo 1.3 Algoritmo de Euclides para calcular el máximo común divisor de dos números enteros, a y b , siendo $a > b$.

1. Hacer $r := a$ módulo b
2. Si $r = 0$ entonces el máximo común divisor es b
3. En otro caso:
 - 3.1. Hacer $a := b$
 - 3.2. Hacer $b := r$
 - 3.3. Volver al paso 1

Por ejemplo, si aplicamos el algoritmo con $a = 20$ y $b = 15$, entonces $r = 5$. No se cumple la condición del paso 2 y aplicamos el paso 3. Obtenemos $a = 15$, $b = 5$. Volvemos al punto 1, llegando a $r = 0$. Al ejecutar el paso 2 se cumple la condición, con lo que el resultado es 5.

Pero el honor de dar nombre al término *algoritmo* lo recibe el matemático árabe del siglo IX **Muhammad ibn Musa al-Khwarizmi** (que significa algo así como Mohamed hijo de Moisés de Khorezm, en Oriente Medio). Sus tratados sobre la resolución de ecuaciones de primer y segundo grado ponen de relieve la idea de resolver cálculos matemáticos a través de una serie de pasos predefinidos. Por ejemplo, para resolver ecuaciones de segundo grado del tipo: $x^2 + bx = c$, define la siguiente serie de pasos⁷.

Ejemplo 1.4 Algoritmo de al-Khwarizmi para resolver la ecuación $x^2 + bx = c$. Se muestra el ejemplo $x^2 + 10x = 39$, el mismo que al-Khwarizmi utiliza en su libro “Hisab al-jabr w’al-muqabala” (“El cálculo de reducción y restauración”), sobre el año 825 d.C.

1. Tomar la mitad de b . (En el ejemplo, $10/2 = 5$)
2. Multiplicarlo por sí mismo. (En el ejemplo, $5 * 5 = 25$)
3. Sumarle c . (En el ejemplo, $25 + 39 = 64$)
4. Tomar la raíz cuadrada. (En el ejemplo, $\sqrt{64} = 8$)
5. Restarle la mitad de b . (En el ejemplo, $8 - 10/2 = 8 - 5 = 3$)

El resultado es: $\sqrt{(b/2)^2 - c - b/2}$, expresado algorítmicamente. El matemático bagdadí es también considerado como el introductor del sistema de numeración arábigo en occidente, y el primero en usar el número 0 en el sistema posicional de numeración.

1.3.1. Definición y propiedades de algoritmo

¿Qué es un algoritmo? En este punto, el lector no debería tener ningún problema para dar su propia definición de algoritmo. Desde los ejemplos de algoritmos más antiguos – como los de Euclides y al-Khwarizmi, mostrados antes– hasta los más modernos algoritmos – como los usados por el buscador Google o en compresión de vídeo con DivX– todos ellos se caracterizan por estar constituidos por una serie de pasos, cuya finalidad es producir unos datos de salida a partir de una entrada. Además, para que ese conjunto de reglas tenga sentido, las reglas deben ser precisas y terminar en algún momento. Así pues, podemos dar la siguiente definición informal de algoritmo.

Definición 1.4 Un algoritmo es una serie de reglas que dado un conjunto de datos de entrada (posiblemente vacío) produce unos datos de salida (por lo menos una) y cumple las siguientes propiedades:

⁷La notación es adaptada, ya que al-Khwarizmi no utiliza símbolos, expresiones ni variables, sino una descripción completamente textual. Por ejemplo, a b lo llama “las raíces” y a c “las unidades”.

- **Definibilidad.** El conjunto de reglas debe estar definido sin ambigüedad, es decir, no pueden existir dudas sobre su interpretación.
- **Finitud.** El algoritmo debe constar de un número finito de pasos, que se ejecuta en un tiempo finito y requiere un consumo finito de recursos.

En la figura 1.3 se muestra una interpretación de los algoritmos como cajas negras, donde se destaca la visión del algoritmo como algo que dado unos datos de entrada produce una salida. La propiedad de definibilidad se refiere a los algoritmos “en papel”, que también son algoritmos si están expresados sin ambigüedades. Un trozo de código implementado en un lenguaje de programación no tendrá ambigüedad, pero también deberá tener una duración finita para ser considerado un algoritmo. El problema general de demostrar la terminación de un conjunto de reglas es un problema complejo y no computable, es decir no puede existir ningún programa que lo resuelva.

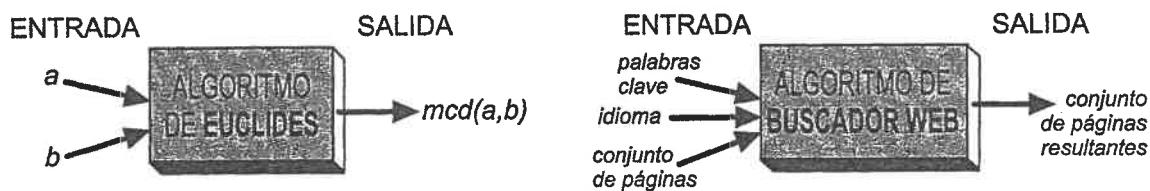


Figura 1.3: Interpretación de los algoritmos como cajas negras.

No todo lo que aparece en informática son algoritmos. Un ejemplo típico de algo que no es un algoritmo es un sistema operativo. El sistema operativo consta de una serie de instrucciones, pero no se le prevé una ejecución finita, sino que debería poder ejecutarse mientras no se requiera lo contrario.

Algoritmos deterministas y no deterministas

Según un algoritmo produzca o no siempre los mismos valores de salida para las mismas entradas, clasificamos los algoritmos en dos tipos:

- **Algoritmos deterministas.** Para los mismos datos de entrada producen siempre los mismos datos de salida.
- **Algoritmos no deterministas.** Para los mismos datos de entrada pueden producir diferentes salidas.

El no determinismo puede chocar un poco con la idea de definibilidad. Sin embargo, ambas propiedades no son contradictorias. Por ejemplo, el resultado de un algoritmo paralelo, que se ejecuta en distintos procesadores, puede depender de cuestiones de temporización que no forman parte de los datos del problema.

La existencia del no determinismo hace que un algoritmo no siempre se pueda ver como una función –en sentido matemático– de la forma $f: \text{ENTRADA} \rightarrow \text{SALIDA}$.

Definición formal de algoritmo

Donald E. Knuth, uno de los grandes pioneros en el estudio de los algoritmos y la programación, propone la siguiente definición formal de algoritmo.

Definición 1.5 Un **método de cálculo** es una cuaterna (Q, I, W, f) , en la cual:

- I es el conjunto de **estados de entrada**;
- W es el conjunto de **estados de salida**;
- Q es el conjunto de **estados de cálculo**, que contiene a I y W ; y
- f es una función: $f : Q \rightarrow Q$, con $f(w) = w; \forall w \in W$.

Definición 1.6 Una **secuencia de cálculo** es una serie de estados: $x_0, x_1, x_2, \dots, x_n, \dots$, donde $x_0 \in I$ y $\forall k \geq 0; f(x_k) = x_{k+1}$. Se dice que la secuencia de cálculo acaba en n pasos si n es el menor entero con $x_n \in W$.

Una definición formal como esta nos puede servir para comprobar, de manera rigurosa, si un algoritmo cumple las propiedades de finitud y definibilidad. Sin embargo, intentar resolver un problema definiendo la función f puede resultar una tarea bastante compleja. Y, lo que es peor, el resultado puede que nos aporte pocas ideas sobre cómo implementar el algoritmo en un lenguaje de programación. En lo sucesivo, intentaremos seguir una visión más práctica, pero formalizando en aquellos sitios donde sea conveniente.

La disciplina de la algorítmica

Si bien, como hemos visto, la historia de los algoritmos se remonta muy atrás, el estudio de los algoritmos no se concibió como una disciplina propia hasta bien entrada la mitad del pasado siglo XX. Actualmente, entendemos la **algorítmica** como la disciplina, dentro del ámbito de la informática, que estudia técnicas para construir algoritmos eficientes y técnicas para medir la eficiencia de los algoritmos. En consecuencia, la algorítmica consta de dos grandes áreas de estudio: el análisis y el diseño de algoritmos.

Pero, ¿cuál es el objetivo último de la algorítmica?, ¿qué es lo que motiva su estudio? El objetivo último es dado un problema concreto ser capaz de resolverlo de la mejor forma posible, de forma rápida, corta, elegante y fácil de programar. Y recordemos que en esta resolución entran también en juego las estructuras de datos, que son manejadas por los algoritmos. En definitiva, todos los ejemplos, técnicas, métodos y esquemas que vamos a estudiar tienen como finalidad última servir de herramientas útiles en el momento de afrontar la resolución de un problema completamente nuevo y desconocido.

1.3.2. Análisis de algoritmos

El análisis de algoritmos es la parte de la algorítmica que estudia la forma de medir la **eficiencia** de los algoritmos. Pero ¿cuándo decimos que un algoritmo es más o menos eficiente? Utilizando un punto de vista empresarial, podemos definir la eficiencia como la relación entre recursos consumidos y productos obtenidos. Se trata, por lo tanto, de una

nueva formulación de la bien conocida *ley del mínimo esfuerzo*: maximizar los resultados, minimizando el esfuerzo. Los resultados que ofrece un algoritmo pueden ser de distintos tipos:

- Un algoritmo puede resolver el problema de forma muy precisa, si es de tipo matemático, o garantizar el óptimo, en problemas de optimización, o encontrar siempre la solución, si es de satisfacción de restricciones.
- Otro algoritmo puede ser que sólo encuentre soluciones aproximadas, o más o menos cercanas al óptimo, pero siempre encuentre alguna.
- Finalmente, otro algoritmo puede que encuentre soluciones sólo en determinados casos, buenas o malas, y en otros casos acabe sin devolver ninguna respuesta.

Por otro lado, los **recursos** que consume un algoritmo pueden ser de distintos tipos. Entre los más importantes tenemos:

- Tiempo de ejecución, desde el inicio del programa hasta que acaba.
- Utilización de memoria principal.
- Número de accesos a disco, a la red o a otros periféricos externos.
- Número de procesadores usados, en el caso de los algoritmos paralelos, etc.

En aplicaciones distintas puede que el recurso crítico sea diferente. En la mayoría de los casos, el recurso clave es el tiempo de ejecución, por lo que muchas veces se asocia eficiencia con tiempo. Pero, recordemos, ni el tiempo es el único recurso, ni el tiempo por sí solo mide la eficiencia, si no que se debe contrastar con los resultados obtenidos.

En conclusión, un algoritmo será mejor cuanto más eficiente sea. No obstante, el sentido común nos dice que habrán otros muchos criterios para decidir lo bueno que es un algoritmo. Por ejemplo, será importante: que sea fácil de entender y de programar, que sea corto en su extensión, que sea robusto frente a casos extraños, que se pueda reutilizar en otros problemas, que se pueda adaptar, etc.

Factores en la medición de recursos

Supongamos el siguiente algoritmo sencillo, que realiza una búsqueda secuencial con centinela dentro de un array de enteros. La cuestión que se plantea es ¿cuántos recursos, de tiempo de ejecución y memoria, consume el algoritmo?

operación BusquedaSec (x, n: entero; a: array [1..n+1] de entero): entero

```

i := 0
a[n+1] := x
repetir
    i := i + 1
    hasta a[i] = x
    devolver i

```

Por muy perdido que ande el alumno, difícilmente habrá contestado algo como “tarda 3 segundos” o “requiere 453 Kbytes...”. Como mínimo, está claro que el tiempo y la

memoria dependen de n , de los datos de entrada, del procesador, de que estemos grabando un CD al mismo tiempo y de otras mil cosas más. Podemos clasificar todos los factores que intervienen en el consumo de recursos en tres categorías.

- **Factores externos.** No indican nada relevante sobre las características del algoritmo; son, más bien, cuestiones externas al algoritmo, relacionadas con el entorno en el que se implementa y ejecuta. Entre los principales factores externos tenemos: el ordenador donde se ejecuta el algoritmo, el uso de procesador por parte de otras tareas, el sistema operativo, el lenguaje de programación usado, el compilador (incluyendo las opciones de compilación usadas), la implementación concreta que se haga del algoritmo, etc.
- **Tamaño del problema.** El tamaño es el volumen de datos de entrada que debe manejar el algoritmo. Normalmente, el tamaño viene dado por uno o varios números enteros. La relación del tiempo con respecto al tamaño sí que resulta interesante sobre las características del algoritmo.
- **Contenido de los datos de entrada.** Para un tamaño de entrada fijo y bajo las mismas condiciones de ejecución, un algoritmo puede tardar distinto tiempo para diferentes entradas. Hablamos de diferentes casos: **mejor caso**, es el contenido de los datos de entrada que produce la ejecución más rápida del algoritmo (incluso para tamaños de entrada grandes); **peor caso**, es el que da lugar a la ejecución más lenta para un mismo tamaño; **caso promedio**, es el caso medio de todos los posibles valores de entrada.

Por ejemplo, en el anterior algoritmo de búsqueda secuencial con centinela, el tamaño del problema viene dado por n . No es lo mismo aplicar el algoritmo con $n = 5$ que con $n = 5.000.000$. Pero la ejecución con $n = 5.000.000$ puede tardar menos que con $n = 5$, si el x buscado se encuentra en la primera posición. Así pues, el mejor caso será que $a[1] = x$, independientemente de lo que valga n . El peor caso será que x no se encuentre en el array, con lo cual la búsqueda acabará al llegar a $a[n + 1]$. El caso promedio sería la media de todas las posibles entradas. Supongamos que x está en a con probabilidad p y que, en caso de estar, todas las posiciones tienen la misma probabilidad. Entonces se recorrerán $n/2$ posiciones con probabilidad p , y $n + 1$ posiciones con probabilidad $1 - p$.

Notaciones en el análisis de algoritmos

Nos interesa que el análisis de los recursos consumidos por un algoritmo sea independiente de los factores externos. En consecuencia, el objetivo es estudiar la variación del tiempo y la memoria utilizada, en función del tamaño de la entrada y, posiblemente, para los casos mejor, peor y promedio, en caso de ser distintos.

Esta dependencia se expresa como una o varias funciones matemáticas. El tiempo será normalmente una función $t : N \rightarrow R^+$, y de forma parecida la memoria $m : N \rightarrow R^+$. Pero si el algoritmo tarda distinto tiempo para la misma n , entonces no se puede decir que t sea una función de N en R^+ . En ese caso, definimos los tiempos en los casos mejor, peor y promedio; denotaremos estos tiempos como t_m , t_M y t_p , respectivamente.

En concreto, ¿qué se mide con $t(n)$? Pues lo que más nos interese.

- Podemos medir **tiempos de ejecución**, en unidades de tiempo, asignando constantes indefinidas a cada instrucción o línea de código. En el algoritmo de búsqueda secuencial, tendríamos los siguientes tiempos:
 $t_m(n) = c_1 + c_2 + c_4 + c_5 + c_6$; $t_M(n) = c_1 + c_2 + c_6 + (c_4 + c_5)(n + 1)$;
 $t_p(n) = c_1 + c_2 + c_6 + (c_4 + c_5)(1 + (1 - p)n + p n/2)$
Siendo cada c_i el tiempo de ejecución de la instrucción de la línea i -ésima.
- También podemos medir simplemente el **número de instrucciones**. Tendríamos:
 $t_m(n) = 5$; $t_M(n) = 5 + 2n$; $t_p(n) = 5 + 2((1 - p)n + p n/2)$
No todas las instrucciones tardan el mismo tiempo, pero sabemos que todas ellas tardan como máximo un tiempo que está acotado por una constante.
- Puede que lo que nos interese medir sea únicamente **algún tipo de instrucción**, que sabemos que es la más relevante. Por ejemplo, si contamos el número de comparaciones en el algoritmo de búsqueda secuencial tenemos:
 $t_m(n) = 1$; $t_M(n) = 1 + n$; $t_p(n) = 1 + (1 - p/2)n$

Las tres versiones de cada una de las funciones t_m , t_M y t_p anteriores, nos vienen a decir lo mismo: en el mejor caso el tiempo es constante, en el peor es proporcional a n , y en promedio depende de n y de p . Pero seguro que la última forma, contando sólo comparaciones, se puede interpretar más fácilmente, porque es la más sencilla.

Para simplificar las expresiones de $t(n)$ usamos las notaciones asintóticas. Las **notaciones asintóticas** son notaciones que sólo tienen en cuenta el crecimiento asintótico de una función –es decir, para valores tendiendo a infinito– y son independientes de las constantes multiplicativas. Tenemos las siguientes notaciones:

- **O-grande: orden de complejidad.** Establece una cota superior de la función, asintóticamente e independiente de constantes multiplicativas.
- **Ω : omega.** Establece una cota inferior, asintótica e independiente de constantes.
- **Θ : orden exacto.** Se usa cuando la cota inferior y superior de una función coinciden.
- **o -pequeña.** Es similar a la notación Θ , pero teniendo en cuenta el factor que multiplica al término de mayor grado.

Podemos encontrar diferentes utilidades a las notaciones asintóticas:

- **Simplificar la notación** de una fórmula con expresión larga y complicada. Por ejemplo, dado un $t(n) = 2n^2 + 7,2n \log_3 n - 21\pi n + 2\sqrt{n} + 3$, podemos quedarnos simplemente con que $t(n) \in \Theta(n^2)$, o también $t(n) \in o(2n^2)$.
- **Acotar una función** que no se puede calcular fácilmente. Esto puede ocurrir en algunos algoritmos cuyo comportamiento es difícil de predecir. Supongamos, por ejemplo, la función $t(n) = 8n^{5-\cos(n)}$, similar a la mostrada en la figura 1.4a). Se puede acotar superiormente con $t(n) \in O(n^6)$ e inferiormente con $t(n) \in \Omega(n^4)$.

- Delimitar el rango de variación del tiempo de un algoritmo, cuando no siempre tarda lo mismo. Por ejemplo, en el algoritmo de búsqueda secuencial tenemos distintos casos mejor y peor. Podemos decir que tiene un $O(n)$ y un $\Omega(1)$.

En la figura 1.4 se muestran dos ejemplos de aplicación de las notaciones asintóticas, para los dos últimos casos.

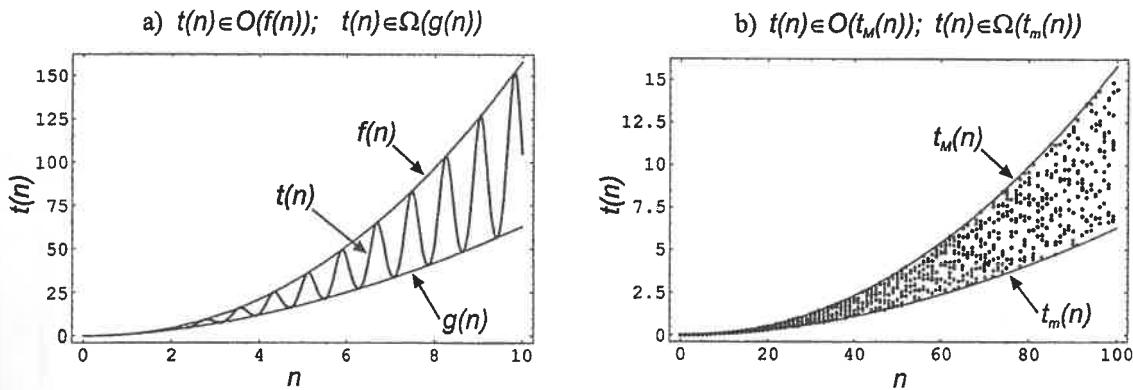


Figura 1.4: Distintas aplicaciones de las notaciones asintóticas. a) Para acotar una función oscilante o difícil de calcular con precisión. b) Para acotar un algoritmo que no siempre tarda lo mismo para el mismo n .

Las notaciones asintóticas pueden servir también para comparar fácilmente los tiempos de dos algoritmos. Pero hay que tener en cuenta sus limitaciones, derivadas de la independencia de las constantes y de los términos de menor grado. Por ejemplo, un algoritmo puede tener un tiempo $t_1(n) = 10n^{0,01} + 2$ y otro $t_2(n) = 2 \log_2 n + 30$. Las notaciones nos dicen que $O(\log n) < O(n^{0,01})$. Sin embargo, ¡ $t_1(n)$ sólo será mayor que $t_2(n)$ a partir de valores de n mayores que $4,5 * 10^{216}$!

1.3.3. Diseño de algoritmos

El diseño de algoritmos es la parte de la algorítmica que estudia **técnicas generales de construcción de algoritmos** eficientes. El objetivo no es, por lo tanto, estudiar unos cuantos algoritmos dados, sino desarrollar técnicas generales que se pueden aplicar sobre diversos tipos de problemas. Por ejemplo, divide y vencerás no es un algoritmo, sino una técnica que puede ser aplicada sobre una amplia variedad de problemas.

En primer lugar, cada técnica de diseño de algoritmos propone una **interpretación particular** del problema, es decir, una forma de ver o modelar el objetivo del problema. Esta interpretación consta de una serie de **elementos genéricos**, que deben ser concretados en cada caso. Por ejemplo, divide y vencerás interpreta que el problema trabaja con una serie de datos, que se pueden dividir para tener problemas más pequeños. Los elementos genéricos de la técnica serían: la manera de hacer la división de los datos, la forma de juntar las soluciones, una forma de resolver casos de tamaño pequeño, etc. Los componentes pueden ser tanto operaciones como estructuras de datos.

Habiendo resuelto los componentes genéricos del problema, la técnica propone un **esquema algorítmico**, que es una guía de cómo combinar los elementos para construir algoritmos basados en esa técnica. El esquema algorítmico puede ser más o menos detallado. Idealmente, el esquema algorítmico sería como un “algoritmo con huecos”, correspondientes a los elementos genéricos. El programador simplemente debería insertar el código y los tipos adecuados en los huecos que correspondan, y ya tendría un algoritmo que resuelve el problema. La idea se muestra en la figura 1.5.

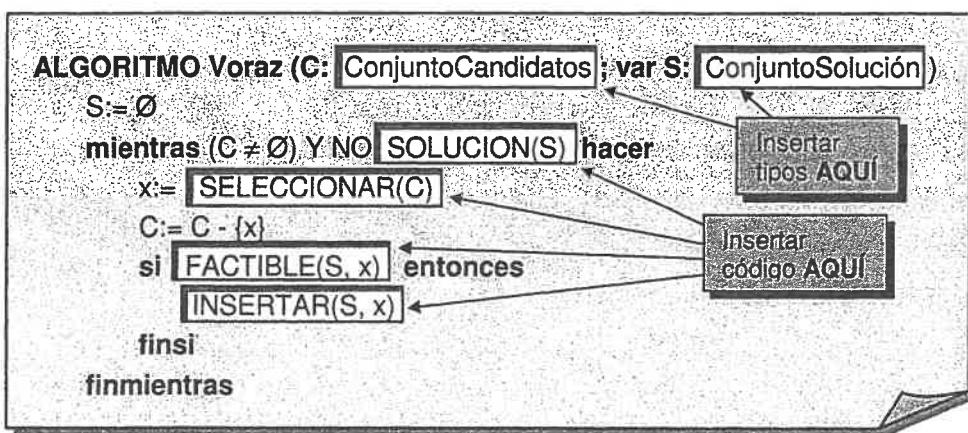


Figura 1.5: Interpretación de los esquemas algorítmicos como “algoritmos con huecos”. Esquema de algoritmo voraz.

En la práctica, esta idea de construir algoritmos “rellenando huecos de un esquema” no suele ser tan simple ni extendida, por varios motivos. En primer lugar, la variedad de situaciones que nos podemos encontrar en distintos problemas es, normalmente, mucho mayor que la prevista en los esquemas de algoritmos. Pero, por otro lado, en esquemas excesivamente flexibles el problema suele ser la ineficiencia; un algoritmo escrito desde cero para un problema particular puede incluir código optimizado para ese caso dado. De todas formas, los esquemas algorítmicos son una ayuda muy útil en la construcción de algoritmos, por lo menos para la obtención de una primera versión.

Otra cuestión fundamental en la resolución de problemas es decidir qué técnica se puede aplicar en cada caso. Aunque puede que ocurra en un examen, en un problema de la vida real nadie nos dirá “prógrámame esto utilizando backtracking” o “calcula aquello con programación dinámica”; el que lo resuelve debe decidir también cómo. Puede que se puedan aplicar varias técnicas, que alguna de ellas se pueda aplicar de diferentes modos, que distintas técnicas den diferentes resultados –por ejemplo, una aproxima la solución y otra encuentra el óptimo– o que no se pueda aplicar ninguna técnica. En el último caso, entrará en juego la habilidad de la persona para enfrentarse a situaciones nuevas.

Vamos a adelantar algunas de las principales técnicas generales de diseño de algoritmos, que veremos a lo largo de este libro. Se supone que el alumno está un poco familiarizado con alguna de ellas –como divide y vencerás, o backtracking–, aunque no resulta imprescindible.

Divide y vencerás

La técnica de **divide y vencerás** aporta una idea muy general: para resolver un problema primero divídelo en partes, luego resuélvelas por separado y finalmente junta las soluciones de cada parte. La idea es tan abierta que se podría aplicar casi en cualquier ámbito⁸. En programación, los subproblemas suelen ser del mismo tipo que el problema original, por lo que se resuelven aplicando recursividad.

Cualquier recursividad necesita un caso base (o varios), que actúe como punto de finalización. Los casos base en divide y vencerás serán problemas de tamaño pequeño, para los cuales se aplica una solución directa.

Por lo tanto, para que se pueda aplicar divide y vencerás: el problema se debe poder descomponer en partes, las soluciones de las partes deben ser independientes, tiene que haber algún método directo de resolución y debe haber una manera de combinar las soluciones parciales. Habrá muchos casos donde no se cumplan estas condiciones.

Algoritmos voraces

La interpretación de un problema bajo la perspectiva de los **algoritmos voraces** es básicamente la siguiente: tenemos un conjunto de candidatos y la solución se construye seleccionando algunos de ellos en cierto orden. Con este modelo del problema, el esquema voraz propone que la solución se construya paso a paso. Se empieza con una solución vacía. En cada paso se selecciona un candidato y se decide si se añade a la solución o no. Una vez tomada una decisión (añadir o descartar un candidato) nunca se puede deshacer. El algoritmo acaba cuando tengamos una solución o cuando no queden candidatos.

El esquema corresponde al viejo algoritmo de “comprar patatas en el mercado”: tenemos un montón de patatas candidatas y una bolsa vacía; lo que hacemos es ir llenando la bolsa con las mejores patatas, una a una, hasta completar 5 kilos. Los componentes genéricos que habría que estudiar en cada problema son: cuáles son los candidatos, qué criterio se usa para seleccionar un candidato del montón, el criterio para decir si se añade el candidato seleccionado, el criterio de finalización, etc.

Por la propia estructura del algoritmo, se puede intuir que los algoritmos voraces serán normalmente bastante rápidos. Como contrapartida, puede que no siempre encuentren solución óptima, aunque algunos de ellos sí que la garantizan.

Programación dinámica

La **programación dinámica** coincide con divide y vencerás en la idea de definir un problema recursivamente en base a subproblemas de menor tamaño. Sin embargo, en programación dinámica no se utiliza la recursividad de forma directa. En su lugar, se resuelven primero los problemas de tamaño pequeño, anotando la solución en una tabla. Usando las soluciones de la tabla, vamos avanzando hacia problemas cada vez más grandes, hasta llegar al problema original. La memoria ocupada por la tabla se convierte en uno de los factores críticos de los algoritmos de programación dinámica.

⁸De hecho, la expresión “divide y vencerás” se atribuye a Napoleón Bonaparte, que fue fiel a su aplicación en sus estrategias militares, hasta que en Waterloo, en 1815, se convirtió en “divide y perderás”.

Para obtener la descomposición recursiva, el problema se puede interpretar como una serie de toma de decisiones. Después de cada decisión, nos queda un problema de menor tamaño. Por ejemplo, el problema de comprar patatas podría interpretarse como: decidir para cada patata del montón si se toma o no (*true* o *false*). La decisión se hace resolviendo los subproblemas de usar el montón con una patata menos, suponiendo que la otra ya se ha tomado o no⁹. El resultado será la mejor de ambas posibilidades.

Un algoritmo de programación dinámica encontrará la solución óptima a un problema si cumple el principio de **optimalidad de Bellman**: una secuencia de decisiones óptimas se forma uniendo subsecuencias óptimas. No en todos los problemas se puede aplicar el principio, en muchos de ellos es difícil encontrar la definición recursiva y en otros demasiado ineficiente.

Backtracking

La técnica de **backtracking** se puede ver como la aplicación de una estrategia opuesta a los algoritmos voraces. Si en los algoritmos voraces nunca se deshacen las decisiones tomadas, en backtracking se pueden deshacer, quitar elementos que hemos añadido, probar otros. Y así hasta haber comprobado virtualmente todas las posibilidades.

En un algoritmo de backtracking, las soluciones al problema suelen representarse mediante tuplas (s_1, s_2, \dots, s_n) , donde cada s_i significará algo concreto y podrá tomar ciertos valores. El esquema genérico de backtracking consiste básicamente en generar todos los posibles valores consistentes de la tupla anterior –haciendo un recorrido exhaustivo y sistemático– y quedarse con la óptima o la que solucione el problema.

El orden de generación de los distintos valores de la tupla se puede representar mediante un árbol –el árbol implícito de backtracking, no necesariamente almacenado– donde la raíz es el comienzo del algoritmo y los hijos de un nodo son las tuplas que se generan a partir de una solución parcial. La técnica de backtracking es muy flexible y se puede aplicar sobre una gran variedad de problemas. Desafortunadamente, los algoritmos de backtracking suelen ser extremadamente costosos en tiempo de ejecución.

Ramificación y poda

En cierto sentido, la **ramificación y poda** se puede ver como una generalización y mejora de backtracking. Realiza la misma interpretación del problema, a través del uso de una tupla solución (s_1, s_2, \dots, s_n) , cuyos valores son generados mediante un recorrido implícito en un árbol. Pero difiere de backtracking en dos aspectos. ¿Cuáles?

Pues, obviamente, en la ramificación y en la poda. En cuanto a la ramificación, permite definir distintas estrategias para recorrer el árbol, es decir, determinar qué ramas explorar primero. En cuanto a la poda, define un proceso de eliminación de nodos del árbol por los cuales sabemos que no se encuentra la solución óptima.

La técnica se basa en ir haciendo estimaciones, para cada solución parcial, de la mejor solución que podemos encontrar a partir de la misma. Si la mejor solución que podemos obtener para un nodo no es mejor que lo que ya tengamos por otro lado, entonces será posible eliminar ese nodo.

⁹Notar que un algoritmo voraz decidiría si se usa cada patata de forma independiente.

Estrategia minimax y poda alfa-beta

Más que una técnica general de diseño de algoritmos, la **estrategia minimax** es una estrategia para la resolución de problemas de juegos. Consideramos exclusivamente los juegos de tablero, donde participan dos rivales –que llamamos *A* y *B*– de forma alternativa y no interviene el azar. Por ejemplo, dentro de esta categoría podemos encontrar las tres en raya, las damas, el ajedrez, el Nim, etc.

La evolución de todos los posibles estados del juego es representada mediante un árbol. La raíz representa el tablero inicial. Los hijos de la raíz son todos los posibles movimientos de *A*. Los hijos de estos son los movimientos de *B*, y así sucesivamente. Las hojas son situaciones donde acaba el juego y gana uno u otro jugador.

La estrategia minimax hace un recorrido del árbol de juego, intentando plasmar una idea bastante razonable: en los movimientos de *A* intenta ganar él y que pierda *B*, y en los de *B* intentará ganar él y que pierda *A*.

1.3.4. Descripción del pseudocódigo utilizado

Las técnicas de diseño de algoritmos son independientes del lenguaje e incluso del paradigma de programación que se utilice. Un algoritmo diseñado en papel puede ser implementado en Java, Pascal, Basic, C, Smalltalk o cualquier otro lenguaje. Por este motivo, en el estudio teórico de los algoritmos y las estructuras de datos utilizaremos un pseudocódigo independiente del lenguaje de programación¹⁰. El pseudocódigo usado es bastante intuitivo y su traducción a un lenguaje concreto resulta directa. A continuación vamos a comentar sus principales características.

Para los tipos de datos elementales, o definidos por el usuario, se utiliza el formato presentado en la sección 1.2.3. Las funciones y procedimientos se definen con la palabra **operación**, seguida por el nombre, y la lista de los parámetros formales entre paréntesis. Si un parámetro se puede modificar, se indicará colocando **var** delante de su nombre. Si la operación devuelve un valor, pondremos dos puntos, ‘:’, y a continuación el tipo del valor devuelto.

Por ejemplo, podemos tener las siguientes cabeceras de funciones y procedimientos:

operación SuprimeDos (**var** *C*: Conjunto[T]; *t*, *p*: T)

operación Miembro (*C*: Conjunto[T]; *t*: T): booleano

Normalmente, los nombres de las variables aparecerán en cursiva (*C*, *x*) y las palabras reservadas en negrita (**operación**, **var**).

Incluimos en nuestro pseudolenguaje las siguientes instrucciones:

- **Asignación.** Se expresa mediante ‘:=’. Por ejemplo, *C* := 4 + 3. Para la comparación se usa ‘=’ y ‘≠’.
- **Condicional.** La instrucción tendrá la forma: “**si** Condición **entonces** Acción **finsi**”. También puede haber una acción que se ejecute en caso de no cumplirse la condición. La notación sería:

```
  si Condición entonces
    AcciónSiCierto
```

¹⁰Aunque, como se podrá ver, la estructura general tiene ciertas similitudes con Pascal.

```

sino
    AcciónSiFalso
finsi

```

- **Valor devuelto.** En las funciones, se devolverá el resultado con “**devolver Valor**”.
- **Iteradores.** Utilizaremos los iteradores **para**, **mientras** y **repetir**. El formato será como el siguiente:

para <i>v</i> := <i>Rango</i> hacer	mientras <i>Condición</i> hacer	repetir
<i>Acción</i>	<i>Acción</i>	<i>Acción</i>
finpara	finmientras	hasta <i>Condición</i>

Dentro del código se podrán utilizar variables locales que no hayan sido definidas previamente. Normalmente estará claro el tipo de esas variables. En caso contrario, se expresará la definición de la variable en una sección **var**, después de la cabecera de la operación y antes del código. También podrán encontrarse operaciones que no estén definidas previamente, pero cuyo significado está bastante claro, como **error(Texto)** para indicar un error o **escribir(Texto)** para mostrar un resultado por pantalla. Para los comentarios dentro del código usaremos: // Comentario o (* Comentario *).

1.4. Consejos para una buena programación

Aun con la corta experiencia de programación que se le supone al alumno, seguro que habrá vivido una situación similar a la siguiente. Se empieza a escribir un programa, que previsiblemente no será muy largo, así que se coloca todo dentro del procedimiento principal. Poco a poco, se van añadiendo cosas, nuevos casos, más funcionalidad o un interfaz más completo. Cuando el programa principal es demasiado grande, se pasan algunos trozos de código como procedimientos. Pero no está claro dónde se hace cada cosa, mientras el programa continúa creciendo.

Al final el código se convierte en un *gigante con pies de barro*. Ocupa unas decenas de páginas pero carece de una estructura lógica clara. A posteriori se detectan cosas que se repiten y algunos trozos que deberían modificarse. Pero tratar de hacerlo bien requeriría tantos cambios que se va montando chapuza sobre chapuza. Pero eso no es nada. Cuando vuelves a retomar el programa, varios meses después, resulta imposible saber dónde, por qué y cómo se hacía cada cosa. Aun así, lo intentas modificar. Pero el programa ya no compila; ni volverá a hacerlo nunca más.

Por lo menos, la experiencia habrá servido para conocer qué se debe hacer y qué no, para construir un programa de tamaño mediano-grande. Vamos a ver, a continuación, una serie de consejos prácticos que debería observar todo buen programador.

1.4.1. Importancia del análisis y diseño previos

Usualmente, el gran error de los programadores inexpertos es empezar a teclear código cuando ni si quiera se ha acabado de entender el enunciado de un problema. La

programación, como una disciplina ingenieril –frente a la programación “como arte”¹¹– requiere plantearse de forma metódica la construcción de programas. El informático debe hacer un diseño previo de sus programas, de la misma forma que un arquitecto hace los planos antes de construir la casa.

Ya hemos visto, en el punto 1.1, los pasos que deben aplicarse sistemáticamente en la resolución de problemas. El estudio de los métodos, procesos, técnicas y herramientas para el análisis¹² y diseño de programas es lo que constituye el área de investigación de la **ingeniería del software**. Aun sin haber estudiado ingeniería del software, resulta siempre conveniente hacer un minucioso estudio previo del problema y de la solución propuesta, antes de empezar a programar. A nivel de análisis y diseño se trabajará con conceptos abstractos, olvidando los detalles menos relevantes.

Consejo práctico

Para un futuro ingeniero en informática, puede resultar un hábito interesante tomar medidas cuantitativas de su propio proceso de desarrollo de programas. Para ello, se puede utilizar una tabla como la mostrada en la figura 1.6. Dentro de esta tabla, para cada proyecto concreto se va anotando la dedicación temporal –en número de minutos, por ejemplo– que se emplea en cada fase de desarrollo, desde el análisis hasta la validación.

Proyecto: Algoritmo de búsqueda secuencial con centinela					Fecha de inicio: 3/3/2003
Programador: Ginés García Mateos					Fecha de fin: 4/3/2003
Día/Mes	Análisis	Diseño	Implement.	Validación	TOTAL
3/3	4	6			10
"		3	5	6	14
4/3			2	9	11
TOTAL (minutos)	11	10	7	15	35
MEDIAS (porcentaje)	31.4%	25.7%	20%	42.9%	100%

Figura 1.6: Tabla para contar la dedicación temporal de una persona a un proyecto software, en las diferentes fases de desarrollo. Las cantidades son minutos, medidos de forma aproximada.

Muchas veces la distinción entre una u otra fase no está muy clara. De modo orientativo, podemos diferenciarlas de la siguiente forma:

¹¹ “El proceso de escribir programas [...] puede llegar a ser una experiencia estética similar a componer música o escribir poesía”, Donald E. Knuth, *The Art of Computer Programming*, 1997.

¹² No confundir análisis de problemas con análisis de algoritmos. Analizar un problema es estudiar sus necesidades y características. Analizar un algoritmo es medir su consumo de recursos.

- **Análisis.** Cuenta desde que se empieza a leer el enunciado del problema. Incluye la comprensión del problema, búsqueda de información (libros de teoría, apuntes, manuales, código reutilizado, etc.) y modelado conceptual del problema.
- **Diseño.** Dentro de esta fase podemos contar todo el desarrollo “en papel”. Incluye: definir los bloques en los que se divide el problema, escribir un esquema de los algoritmos, simular mentalmente el esquema, etc.
- **Implementación.** Abarca desde que comenzamos a teclear en el ordenador hasta que se acaba el programa y se compila por primera vez, produzca fallos o no.
- **Validación.** Empieza cuando se hace la primera compilación. Incluye la corrección de errores de compilación, la realización de pruebas, detección de fallos en los resultados y su corrección. Acaba cuando se finaliza el proyecto.

Seguir un control detallado de este tipo puede tener varios objetivos. En primer lugar, sirve para ser conscientes del propio proceso de desarrollo de software, analizando a qué cosas se dedica proporcionalmente más y menos tiempo. En segundo lugar, puede ser útil para estudiar comparativamente la evolución del proceso a lo largo del tiempo. Esto podría utilizarse para detectar deficiencias o puntos débiles. Finalmente, es interesante para extraer conclusiones fundadas: ¿hasta qué punto ayuda dedicar más tiempo a análisis y diseño a reducir el tiempo de implementación y validación? ¿Es fácil distinguir entre las distintas fases? ¿Qué fase requiere más tiempo y por qué? ¿Cómo reducirlo?

1.4.2. Modularidad: encapsulación y ocultamiento

El resultado de un buen diseño de software debe ser la definición de una estructura general de la aplicación. Esta estructura consta de una serie de partes separadas, que se relacionan entre sí a través de los interfaces definidos. Por lo tanto, resulta conveniente utilizar los mecanismos que ofrece el entorno de programación para agrupar funcionalidades: **módulos, paquetes, clases, unidades**.

Un módulo, clase o paquete, agrupa un conjunto de funcionalidades relacionadas. Esto es lo que llamamos **encapsulación**: el módulo tiene un nombre descriptivo, bajo el cual se agrupan todas las utilidades relacionadas con el mismo. Por ejemplo, una clase **Pila** contiene tanto las estructuras de datos como las operaciones necesarias para manejar las pilas. La ventaja de encapsular es que sabemos dónde se encuentra cada cosa. En caso de haber fallos podemos encontrar quién es el responsable, y corregirlo. Si la funcionalidad se utiliza en otras aplicaciones, sólo tendremos que usar el mismo módulo.

Los paquetes son, normalmente, un nivel de encapsulación superior a los otros. Un paquete puede contener varias clases o módulos relacionados. Por ejemplo, un paquete **Colecciones** podría contener las clases **Pila, Lista, Cola, Conjunto**, etc.

Otro principio fundamental asociado a la modularidad es la **ocultación de la implementación**. Cuando se crea un módulo, una clase o un paquete, se definen una serie de operaciones para manejar la funcionalidad creada. Esto es lo que se llama la **interface** del módulo, clase o paquete. Los usuarios del módulo deben utilizar las operaciones del interface y sólo esas operaciones. Todo lo demás es inaccesible, está oculto. Es más, el

usuario sólo conoce cuál es el resultado de las operaciones, pero no cómo lo hacen. Por ejemplo, para el usuario de la clase Pila, el interface contiene las operaciones push, pop, etc., de las cuales conoce su significado, pero no cómo se han implementado.

Consejo práctico

El concepto de software se entiende normalmente como algo mucho más amplio que simplemente un programa; incluye tanto el programa como el documento de requisitos del problema, el diseño realizado y las especificaciones de cada parte. Sería adecuado que todos estos documentos se fueran completando en las mismas fases en las que deben hacerse, y no después de acabar la implementación del programa.

No estudiaremos aquí la forma de crear documentos de requisitos y diseños de software, que caen dentro de las asignaturas de ingeniería del software. El alumno puede utilizar la notación que le parezca más adecuada. En cuanto a la especificación, en el siguiente capítulo haremos un repaso de las especificaciones informales e introduciremos dos métodos de especificación formal. Es conveniente documentar siempre, con alguno de estos métodos, todas las abstracciones (tipos y operaciones) que se desarrollen.

1.4.3. Otros consejos

Las recomendaciones anteriores tienen más sentido cuando se trata de desarrollar aplicaciones de tamaño medio o grande; aunque siempre deberían aplicarse en mayor o menor medida. En este apartado vamos a ver algunos otros consejos más generales, aplicables tanto a problemas pequeños como de mayor tamaño.

- **Reutilizar programas existentes.** Un nuevo programa no debe partir desde cero, como si no existiera nada fuera del propio desarrollo. La reutilización se puede dar a distintos niveles. Se pueden reutilizar librerías, implementaciones de TAD, esquemas de programas, etc.
Reutilizar implica no sólo pensar qué cosas tenemos hechas que podamos volver a usar, sino también qué cosas que estamos haciendo podremos usar en el futuro. Una buena reutilización no se debe basar en “copiar y pegar” código, sino en compartir módulos, clases o paquetes, en distintos proyectos.
- **Resolver casos generales.** Si no supone un esfuerzo adicional, se deberá intentar resolver los problemas más generales en vez de casos particulares. Por ejemplo, en lugar de implementar un tipo pilas de enteros, sería más útil definir pilas que puedan almacenar cualquier cosa. De esta manera, se favorece la posibilidad de que las funcionalidades desarrolladas se puedan reutilizar en el futuro.
- **Repartir bien la funcionalidad.** A todos los niveles (a nivel de paquetes, de módulos y de procedimientos) se debe intentar repartir la complejidad del problema de manera uniforme. De esta forma, hay que evitar crear procedimientos muy largos y complejos. En su lugar, sería mejor detectar los componentes que lo forman y usar subrutinas. Un procedimiento corto, bien estructurado y tabulado, usando funciones y variables con nombres descriptivos adecuados, resulta más legible y, por lo tanto,

más fácil de comprender y mantener. De forma similar, si un módulo es muy grande, puede que estemos mezclando funcionalidades separadas y sea mejor crear dos o más módulos.

- **Simplificar.** No es mejor un programa cuanto más largo y complejo, sino más bien todo lo contrario. Una solución corta y directa es, a menudo, la solución más elegante y adecuada. Una solución enrevesada suele incluir código repetido, casos innecesarios, desperdicio de memoria y falla con más facilidad. Para simplificar se debe aplicar la lógica y el sentido común. Por ejemplo, suponiendo una función que devuelve un booleano, es muy típico encontrar cosas del siguiente estilo o similares:

```

si Condición = verdadero entonces
    devolver verdadero
sino
    devolver falso
finsi
```

Toda la construcción anterior no es más que un largo rodeo para decir simplemente:
devolver Condición

O, por ejemplo, cuando tenemos una variable *i* que debe ir oscilando entre dos valores, pongamos por caso los valores 1 y 2, también se suelen hacer análisis de casos *if-else*. El problema se soluciona con una simple asignación del tipo: *i* := 3 - *i*.

Consejo práctico

No perdamos de vista que el interés principal es que los programas desarrollados sean correctos y eficientes. Dentro de eso, resulta muy conveniente que se cuiden los otros criterios de una buena programación: legibilidad, simplicidad, reparto de la funcionalidad, reutilización. La mejor forma de evaluarse uno mismo en estos parámetros es tomar medidas cuantitativas y que sean significativas, al estilo de lo que se propone en el apartado 1.4.1.

En la figura 1.7 se propone un formato para el tipo de información que sería interesante recoger en cada proyecto realizado. El número de operaciones de la interface se refiere a las operaciones que son accesibles fuera del módulo o clase. Las privadas son las que no son accesibles desde fuera. Por simplicidad, la longitud del código se debe expresar en una unidad que sea fácil de medir, como por ejemplo líneas de código. Por otro lado, dentro del campo *reutilizado* se indica si el módulo ha sido reutilizado de un proyecto anterior, o si se han reutilizado partes con modificaciones.

Está claro que una tabla como la de la figura 1.7 sólo tendrá sentido cuando el programa esté completamente acabado y además sea correcto. Una vez con la tabla, cabría plantearse diferentes cuestiones. ¿Es uniforme el reparto de operaciones entre los módulos? ¿Hay módulos que destaque por su tamaño grande o pequeño? ¿Está justificado? ¿Es razonable la longitud media de las operaciones? ¿Hay alguna operación demasiado larga? ¿Se ha reutilizado mucho o poco? E, intentando ir un poco más allá, ¿existe algún otro parámetro de calidad que hayamos encontrado, pero que no está recogido en la tabla? En ese caso, ¿cómo se podría cuantificar?

Proyecto: Algoritmo ficticio					Fecha de inicio: 3/5/2003			
Programador: Ginés García Mateos					Fecha de fin: 7/5/2003			
Nombre del Módulo/Clase	Número de operaciones	Lineas de código	Reutilizada	Lineas por operación	Min	Max	Media	
Pilas	4	2	6	53	no	5	13	8,8
Principal	1	3	4	39	no	7	24	9,8
TOTAL	5	5	10	92	0	5	24	9,2
MEDIAS	2,5	2,5	5	46	0	6	18,5	9,2

Figura 1.7: Tabla para tomar diferentes medidas relacionadas con la calidad del software.

Ejercicios propuestos

Ejercicio 1.1 Busca uno o varios programas –preferiblemente de tamaño más o menos grande– que hayas escrito con anterioridad. Rellena una tabla como la de la figura 1.7. Si el programa no utiliza módulos, unidades o clases, intenta agrupar las operaciones por funcionalidad. Extrae conclusiones de los resultados obtenidos, respondiendo las cuestiones que se plantean al final del apartado 1.4.3. Haz una valoración crítica y comparativa (respecto a algún compañero o respecto a otros programas).

Consejo: Para facilitar el relleno de la tabla, es adecuado utilizar alguna herramienta como una hoja de cálculo. Normalmente, en este tipo de tablas las celdas claras las introduce el usuario y las oscuras se calculan automáticamente a partir de otras (aunque en el ejemplo no siempre es así).

Ejercicio 1.2 A partir de ahora, para los siguientes problemas de tamaño mediano que resuelvas, intenta hacer un seguimiento del tiempo que dedicas, utilizando una tabla como la de la figura 1.6. Igual que antes, se aconseja automatizarlo usando una hoja de cálculo. Estos documentos, junto con las especificaciones y las medidas de calidad de la tabla 1.7, deberían acompañar la documentación del programa final.

Ejercicio 1.3 Anota en una hoja todos los tipos de datos, estructuras y técnicas de diseño que conozcas. Para cada tipo indica las variantes que puede tener y las formas de implementarlos. Por ejemplo, se supone que debes conocer el tipo lista, que pueden ser circulares, enlazadas simple o doblemente, se pueden implementar con punteros o con arrays, etc.

Cuestiones de autoevaluación

Ejercicio 1.4 A lo largo de todo el capítulo, se hace énfasis en lo importante que resulta no empezar el desarrollo de programas directamente tecleando código. ¿Estás de acuerdo con esta afirmación? ¿Qué otras cosas deberían hacerse antes? ¿En qué fases?

Ejercicio 1.5 ¿Qué relación existe entre los algoritmos y las estructuras de datos? ¿Cuál va primero? ¿Cuál es más importante en la resolución de problemas? ¿Qué papel juega cada uno?

Ejercicio 1.6 ¿Cuáles son las diferencias entre tipo abstracto de datos, tipo de datos y estructura de datos? ¿Son conceptos redundantes? Por ejemplo, una lista ¿qué es?

Ejercicio 1.7 Enumera las propiedades fundamentales de un algoritmo. Dado un trozo de código, ¿se pueden demostrar formalmente las propiedades? ¿Y si es pseudocódigo? Un programa de edición de texto, al estilo *Word*, ¿se puede considerar un algoritmo? ¿En qué se diferencian un algoritmo, un programa, un problema y una aplicación?

Ejercicio 1.8 En el análisis de algoritmos, identificamos distintos factores que influyen en el consumo de recursos. Algunos resultan de interés y otros no. Distínguelos y justifícan por qué se consideran o no interesantes.

Ejercicio 1.9 Todas las notaciones asintóticas (excepto la Θ -pequeña) son independientes de constantes multiplicativas. Comprueba que los factores externos estudiados en el consumo de recursos de un algoritmo solo influyen en términos multiplicativos.

Ejercicio 1.10 ¿Para qué sirve un esquema algorítmico? ¿En qué se diferencia de un algoritmo? ¿Cumple las propiedades de definibilidad y finitud? ¿Por qué no? Enumera los distintos esquemas algorítmicos que conozcas.

Referencias bibliográficas

Antes de entrar de lleno en el contenido del libro, sería adecuado repasar los conceptos fundamentales que el alumno debe conocer de programación de primer curso. Para ello la mejor referencia es, sin duda, la que se usará entonces. Algunos libros de algoritmos y estructuras de datos incluyen capítulos de repaso de los tipos fundamentales –listas, pilas, colas y árboles– como los capítulos 2 y 3 de [Aho88], y los capítulos 3 y 4 de [Weis95], y el libro [Wirth80]; además de las definiciones relacionadas con estructuras y algoritmos, en los capítulos iniciales.

El proceso de resolución de problemas, planteado en el primer punto, es una versión resumida y simplificada del ciclo clásico de desarrollo de software. Pocos libros de la bibliografía inciden en este tema, que está más relacionado con el ámbito de la ingeniería del software. Una breve introducción a estos temas, así como a otros conceptos tratados en este libro, se pueden consultar en el capítulo 1 de [Aho88].

Algunos de los consejos de programación de la sección 1.4, han sido extraídos del apartado 1.6 de [Aho88]. Las tablas de las figuras 1.6 y 1.7 para medir el proceso personal de desarrollo de software, son una adaptación del proceso propuesto en [Humphrey01]. Son una simplificación, ya que este libro está más orientado hacia las cuestiones de ingeniería del software.

Además de la bibliografía en papel, que se detallará en cada uno de los capítulos sucesivos, Internet resulta una valiosa fuente de información adicional. Por ejemplo, un

completo compendio de todos los temas relacionados con los algoritmos y las estructuras de datos es el “Dictionary of Algorithms and Data Structures”:

<http://www.nist.gov/dads/>

Además de la documentación escrita, se pueden encontrar proyectos dirigidos a crear animaciones interactivas de algoritmos y estructuras de datos. Un portal interesante con numerosos enlaces es:

<http://www.cs.hope.edu/~alganim/ccaa/>



Capítulo 2

Abstracciones y especificaciones

Una abstracción es una simplificación de un objeto o fenómeno del mundo real. Las abstracciones son básicas en la construcción de aplicaciones de cierto tamaño, donde es necesario distinguir entre distintos niveles de abstracción, desde la visión global y genérica del sistema hasta, posiblemente, el nivel de código máquina. Una especificación, en este contexto, es una descripción del significado o comportamiento de la abstracción. Las especificaciones nos permiten comprender el efecto de un procedimiento o el funcionamiento de un tipo abstracto de datos, sin necesidad de conocer los detalles de implementación. En este capítulo se tratan los mecanismos de abstracción soportados por los lenguajes de programación, y cómo estas abstracciones son descritas mediante especificaciones más o menos rigurosas.

Objetivos del capítulo:

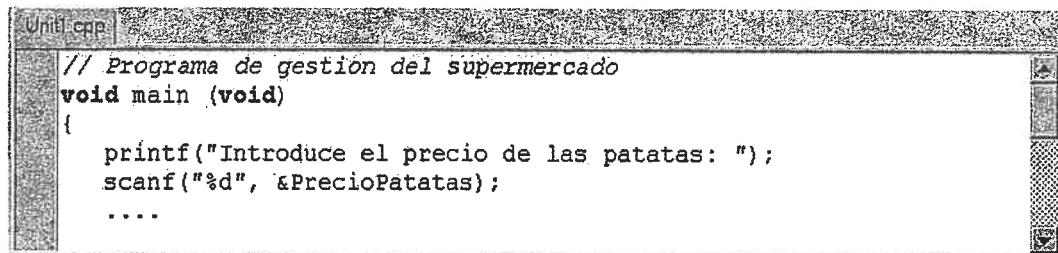
- Reconocer la importancia de la abstracción y conocer los tipos de abstracciones que aparecen en programación: funcional, de datos y de iteradores.
- Conocer los distintos mecanismos de los lenguajes de programación que dan soporte al uso de abstracciones, desde los módulos hasta las clases y objetos.
- Concienciarse de la utilidad de desarrollar especificaciones completas y precisas, entendiendo la especificación como un punto de acuerdo entre el usuario y el implementador de una abstracción.
- Estudiar una notación informal para la especificación de abstracciones.
- Comprender el método de especificación formal algebraico o axiomático (basado en una definición mediante axiomas) y el método constructivo u operacional (basado en el uso de precondiciones y postcondiciones).
- Aplicar las notaciones analizadas en la especificación de una amplia variedad de tipos abstractos de datos.

Contenido del capítulo:

2.1.	Las abstracciones en programación	37
2.1.1.	Diseño mediante abstracciones	38
2.1.2.	Mecanismos de abstracción: especificación y parametrización	39
2.1.3.	Tipos de abstracciones: funcional, de datos e iteradores	40
2.1.4.	Mecanismos de abstracción en los lenguajes de programación	42
2.2.	Especificaciones informales	47
2.2.1.	Especificación informal de abstracciones funcionales	47
2.2.2.	Especificación informal de abstracciones de datos	48
2.2.3.	Especificación informal de abstracciones de iteradores	51
2.3.	Especificaciones formales algebraicas	52
2.3.1.	Propiedades, notación y ventajas de las especificaciones formales	53
2.3.2.	Especificaciones algebraicas o axiomáticas	54
2.3.3.	Taxonomía de las operaciones de un TAD	56
2.3.4.	Compleitud y corrección de la especificación	57
2.3.5.	Reducción de expresiones algebraicas	59
2.4.	Especificaciones formales constructivas	61
2.4.1.	Precondiciones y postcondiciones	61
2.4.2.	Especificación como contrato de una operación	62
2.4.3.	Necesidad de un modelo subyacente	64
2.4.4.	Ejecución de especificaciones constructivas	67
	Ejercicios resueltos	68
	Ejercicios propuestos	75
	Cuestiones de autoevaluación	77
	Referencias bibliográficas	78

2.1. Las abstracciones en programación

Supongamos que recibimos el encargo de crear una aplicación para la gestión de una gran cadena de supermercados, que incluye las compras a los proveedores, el control de las cajas, la gestión de empleados, el inventario del almacén, la contabilidad de la empresa, las campañas publicitarias, la comunicación entre los distintos centros, etc. Los documentos de la aplicación ocupan unos 2.000 folios –aunque no están muy claros– y hay en torno a quince personas implicadas en el proyecto. Ni cortos ni perezosos, empezamos a escribir el programa:



```
Unit.cpp:1
// Programa de gestión del supermercado
void main (void)
{
    printf("Introduce el precio de las patatas: ");
    scanf("%d", &PrecioPatatas);
    ...
}
```

¡Imposible seguir por ese camino! ¡Debe de haber una manera más razonable de abordar la construcción de la aplicación! ¡Cuál es el problema? Uno no puede tener todo en la cabeza al mismo tiempo: los 2.000 folios de requisitos, los doce departamentos de la empresa –cada uno con sus necesidades e intereses–, la estructura final de la aplicación, los tipos de datos necesarios, el flujo de información, etc. Debe haber un método o un proceso, que nos permita ir poco a poco, desde el punto de vista más amplio y genérico del sistema hasta llegar a los pequeños subproblemas.

El estudio de estas técnicas para el desarrollo de grandes aplicaciones es parte del área conocida como *ingeniería del software*, como ya comentamos en el capítulo 1. Prácticamente todas ellas se basan en la aplicación del **concepto de abstracción**: dejar a un lado lo irrelevante y centrarse en lo importante, en la esencia de las cosas.

Pero, ¿qué es lo importante? Obviamente, en cada momento será una cosa distinta. En cierto estado inicial de desarrollo lo que importa será la estructura general de la aplicación. En otra fase, será la implementación de una operación concreta y, en una posible fase de optimización de código, lo importante puede ser el código máquina del programa. Así, tenemos distintos **niveles de abstracción**: según el nivel de detalle, decimos que estamos en un nivel de abstracción mayor o menor.

La abstracción no sólo aparece en el desarrollo de programas, sino que es usada en todos los órdenes de la vida, cuando intentamos abordar un problema complejo. Ser capaz de distinguir los aspectos superfluos de los que forman la esencia de un problema es una de las claves de la inteligencia humana. En programación, las abstracciones son aplicadas de forma continua, dando lugar a conceptos básicos como las rutinas, funciones, procedimientos y los tipos abstractos de datos. Pero la cosa no acaba ahí, y la aplicación de mecanismos de abstracción en los lenguajes de programación da lugar a conceptos y principios fundamentales, como los módulos, las clases, la genericidad, la ocultación de información, la encapsulación. Gracias a ellos, es posible construir aplicaciones cada vez más complejas y fiables, con un menor coste. El propio proceso de diseño de programas está basado en el uso de abstracciones.

2.1.1. Diseño mediante abstracciones

Consideremos la aplicación de gestión de la gran cadena de supermercados, planteada en la introducción. Vamos a ver cómo usando abstracciones se puede abordar su resolución de forma ordenada y sistemática.

Un ejemplo de diseño usando abstracciones

Dejando a un lado todos los detalles irrelevantes, podemos distinguir los grandes bloques en los que se divide el problema. Por un lado tenemos una base de datos, en la que estará almacenada toda la información que maneje la empresa. Por otro lado, tendremos una aplicación de administración y mantenimiento del sistema. Y finalmente tendremos una o varias aplicaciones de usuarios, que acceden a la parte que les corresponde de la base de datos.

De esta forma, se puede establecer una descomposición inicial del problema en tres grandes subproblemas: *Base de Datos*, *Administración* y *Usuarios*. Cada uno de los subproblemas es una abstracción de una parte del sistema. Existe una relación entre ellos, pero básicamente se pueden abordar de forma independiente. La estructura global del sistema se muestra en la figura 2.1.



Figura 2.1: Una visión muy abstracta de un sistema informático genérico.

¿Cómo resolver ahora los subproblemas? Pues volviendo a aplicar la abstracción, evidentemente. Para ello será necesario especificar, de forma abstracta, la funcionalidad que es responsabilidad de cada parte. Por ejemplo, si nos centramos en la Base de Datos, su misión es almacenar toda la información manejada en la empresa, permitiendo su consulta y modificación. Podemos distinguir, a su vez, cuatro grandes partes: *productos*, *proveedores*, *clientes* y *empleados*. De esta manera podemos seguir descomponiendo en partes cada vez más pequeñas, hasta llegar al nivel de programación. Luego se van combinando las soluciones de cada parte y se compone la solución para el problema original.

El proceso de diseño mediante abstracciones

Analizando el ejemplo anterior, se observa que existe un modo ordenado y sistemático de abordar el problema, que se va repitiendo en los distintos niveles de abstracción. Este proceso aparece siempre en la resolución de problemas complejos, y es lo

que se denomina **diseño mediante abstracciones**. El proceso está compuesto básicamente por los siguientes cuatro pasos.

Pasos en el proceso de diseño mediante abstracciones

1. Identificar los subproblemas en los que se divide el problema original.
2. Especificar cada subproblema de forma abstracta, usando alguna notación.
3. Resolver cada subproblema de forma separada.
4. Unir las soluciones y verificar la solución para el problema original.

En conclusión, se puede apreciar que se trata de un proceso cíclico de especificar / implementar / verificar. Tras el paso de verificación puede ser necesario volver al paso 3 (corregir la implementación), al paso 2 (cambiar la especificación) o incluso al 1 (hacer otra descomposición más adecuada de los subproblemas). Este ciclo constituye la idea básica del proceso clásico de desarrollo de programas, como vimos en el capítulo 1.

Por otro lado, existe una clara analogía con el **método científico** –aplicado en las ciencias experimentales– compuesto por los pasos de: observación, proposición de hipótesis, experimentación y verificación de la hipótesis. El proceso se puede ver, por lo tanto, como una aplicación del método científico a la disciplina de la programación. En nuestro caso, las abstracciones de objetos del mundo real juegan el papel de hipótesis que deben ser especificadas, implementadas y verificadas.

2.1.2. Mecanismos de abstracción: especificación y parametrización

Las abstracciones utilizadas en programación (procedimientos, funciones y tipos de datos) son descritas mediante especificaciones, que determinan su comportamiento esperado. Además, esas abstracciones pueden estar parametrizadas, de forma que su significado está dado en función de ciertos parámetros. La especificación y la parametrización son mecanismos de abstracción independientes, pueden aparecer juntos o no.

Abstracción por especificación

Como vimos en la introducción, la abstracción consiste en distinguir lo que es relevante de lo que no lo es. En la abstracción por especificación lo relevante es la descripción del efecto o resultado, y lo irrelevante es la forma de calcularlo.

La mayor parte de las abstracciones usadas en el desarrollo de programas son abstracciones por especificación. La especificación de un procedimiento, o de un tipo de datos, describe su comportamiento, pero no dice nada acerca de cómo funciona internamente, de cómo está implementado.

Consideremos, por ejemplo, las librerías del sistema operativo para crear procesos, abrir ventanas, pedir memoria, etc. La especificación describe el significado, el modo de uso y el efecto de las funciones existentes, pero las cuestiones de implementación quedan

completamente ocultas. Un programador que use esas librerías no sólo no necesita conocer la implementación, sino que conocerla supondría una complejidad excesiva e innecesaria.

Esto es lo que se conoce como el **principio de ocultación de la implementación**: el usuario de una abstracción no puede ni debe conocer la implementación de la misma. Este principio implica que el lenguaje tenga mecanismos que soporten la ocultación, y que el programador de la abstracción los use de forma adecuada para garantizar la ocultación.

La abstracción por especificación también da lugar al concepto de **encapsulación**: un conjunto de operaciones, o tipos, relacionados son agrupados en un mismo *módulo* o *paquete*. En consecuencia, el módulo se puede considerar como una abstracción de nivel superior, compuesta por varias abstracciones que proporcionan un conjunto de funcionalidades relacionadas. Por ejemplo, las librerías del sistema operativo podrían ser separadas en distintos módulos: uno para la parte gráfica, otro para el manejo de ficheros, otro para los procesos, etc.

Abstracción por parametrización

Las abstracciones pueden estar parametrizadas, de manera que el significado de la abstracción no es fijo, sino que depende del valor de uno o más parámetros. Una clase muy importante es la *parametrización de tipo*, en la cual el parámetro es un tipo de datos.

Supongamos, por ejemplo, que queremos implementar un conjunto de funciones para ordenar los elementos de un array, sean del tipo que sean. Deberíamos definir una función por cada posible tipo de entrada:

```
operación OrdenaEntero (A : array [min..max] de entero)
operación OrdenaReal (A : array [min..max] de real)
operación OrdenaCaracter (A : array [min..max] de carácter)
```

....

Resultaría mucho más interesante crear una abstracción más genérica, que se pueda aplicar sobre cualquier tipo de datos del array. Para ello, *factorizamos* todas las operaciones anteriores creando una nueva operación, donde el tipo de datos de los elementos del array es un parámetro más de la función:

```
operación Ordena[T: tipo_comparable] (A : array [min..max] de T)
```

La operación Ordena[T] es una abstracción de nivel superior, que permite ordenar un array de cosas de cualquier tipo. Se dice que Ordena[T] es una abstracción genérica o parametrizada.

La abstracción por parametrización se aplica también sobre abstracciones de datos, permitiendo definir tipos de datos genéricos. Por ejemplo, un tipo Conjunto[T: tipo] podría almacenar un conjunto de elementos de tipo T, mientras que Pareja[K: tipo] almacenaría un par de valores de cualquier tipo K.

2.1.3. Tipos de abstracciones: funcional, de datos e iteradores

De forma consciente o inconsciente, los mecanismos de abstracción son usados constantemente en programación. Podemos distinguir entre distintos tipos o categorías de abstracciones, en función de qué es lo que se abstrae. Los dos tipos fundamentales son la *abstracción de datos* y la *funcional*. Adicionalmente, encontramos las *abstracciones de iteración*.

iteradores, que permiten definir un recorrido abstracto sobre los elementos de una colección.

Abstracciones funcionales

En la abstracción funcional abstraemos un trozo de código, más o menos largo, al cual le asignamos un nombre. El concepto resultante es la idea de **rutina, procedimiento o función**. La especificación de un procedimiento determina cuáles son los parámetros de entrada, los de salida y el efecto que tendrá su ejecución, pero no se indica nada sobre la implementación. Para el usuario del procedimiento es indiferente cómo se haya implementado, siempre que se garantice el efecto descrito en la especificación.

Por ejemplo, el procedimiento de ordenación `OrdenaEntero(A: array [min..max] de entero)` es una abstracción funcional, de la cual sabemos que su resultado es que los elementos del array de enteros *A* son ordenados en la salida de menor a mayor. Pero, respetando el *principio de ocultación de la implementación*, no sabemos nada acerca de cómo lo hace. Los algoritmos de ordenación por selección, inserción, burbuja, mezcla, etc., son posibles implementaciones de la abstracción `OrdenaEntero`.

Los procedimientos y funciones son *generalizaciones* del concepto de operador de un lenguaje de programación. En lugar de usar los operadores existentes en el lenguaje, un programador puede definir su propio conjunto de operadores, con comportamientos más complejos y adaptados a sus necesidades.

Abstracciones de datos

En la abstracción de datos se abstrae un dominio de valores, junto con un conjunto de operaciones sobre ese dominio, que poseen un significado particular. Esta abstracción da lugar al concepto de **tipo abstracto de datos (TAD)**. La especificación del TAD determina el comportamiento de las operaciones definidas sobre el tipo, pero no dice nada sobre cómo son almacenados los valores del tipo o cómo están implementadas las operaciones.

Por ejemplo, los números naturales forman un TAD, al que le asignamos el nombre **Natural**. Cada natural puede tomar valores dentro de un dominio $\{0, 1, 2, 3, \dots\}$ y puede ser manipulado a través de un conjunto de operaciones definido (suma, multiplicación, resta, etc.) con un significado conocido. En este caso el principio de ocultación de la implementación es aplicado tanto sobre las operaciones como sobre la estructura que se usa para almacenar los datos del dominio.

Igual que con los procedimientos, los TAD se pueden considerar como generalizaciones del concepto de tipo de datos primitivo. En lugar de usar los tipos primitivos definidos en un lenguaje (enteros, reales, caracteres, etc.) el usuario se puede definir sus propios tipos, combinando los tipos primitivos. O puede usar los TAD implementados por otro programador y que sean de utilidad para su problema.

Tipos contenedores y parametrizados

Una clase importante de TAD son los tipos **contenedores o colecciones**. Un tipo contenedor es un TAD que está compuesto por un número no fijo de valores de otro tipo.

Por ejemplo, un array, una cola, una lista o un árbol son tipos contenedores. Las implementaciones de listas mediante cursores, punteros o tablas, son ejemplos de *estructuras de datos* usadas para almacenar los valores del tipo. Se puede decir que la estructura de datos es la *concretización* del tipo *abstracto*.

Para conseguir una definición lo más general posible, se debe permitir que el tipo de valores almacenados en el contenedor sea un TAD cualquiera, es decir, tener listas, arrays o colas de cualquier cosa. En consecuencia, los TAD colecciones deben estar *parametrizados*, incluyendo un parámetro que indica el tipo de objetos almacenados. Por ejemplo, el TAD parametrizado Lista [T: tipo] define las listas de valores de tipo T. Dos posibles instanciaciones del tipo serían: Lista [Natural], o Lista [Lista [Natural]].

Abstracciones de iteradores

Un tipo interesante de abstracción –aunque menos frecuente– es la abstracción de iteradores. Un **iterador** permite realizar un recorrido sobre los elementos de una colección de forma abstracta. En un iterador lo relevante es que se recorren todos los elementos de una colección, independientemente de cuál sea la forma de almacenar los valores del tipo.

Los iteradores son una generalización de los iteradores elementales de los lenguajes de programación. Por ejemplo, un iterador **para**, o **for**, tiene la forma:

para *i* := 1, ..., *n* **hacer** Acción sobre *i*

Esta instrucción permite recorrer todos los enteros entre 1 y *n*, aplicando determinada operación sobre cada valor. Si en lugar de enteros tenemos listas, pilas, árboles o cualquier otra colección, sería muy útil disponer de una operación similar a la anterior, que nos permita recorrer todos los elementos de forma abstracta. La operación podría tener la forma:

para cada elemento *i* **de la lista** *L* **hacer** Acción sobre *i*

para cada elemento *i* **del árbol** *A* **hacer** Acción sobre *i*

Los dos ejemplos anteriores son abstracciones de iteradores: permiten procesar todos los elementos de una colección, independientemente de los detalles de representación del tipo de datos. Otros ejemplos de iteradores pueden incluir una condición sobre los elementos a iterar, devolver otra colección con los elementos procesados o fijar un orden concreto de recorrido. El formato podría ser del siguiente tipo:

para cada elemento *i* **de la colección** *C* **que cumpla** *P(i)* **hacer** Acción sobre *i*

res:= **Seleccionar** los elementos *i* **de la colección** *C* **que cumplan** *P(i)*

Eliminar los elementos *i* **de la colección** *C* **que cumplan** *P(i)*

para cada elemento *i* **del árbol** *A* **en pre-orden** **hacer** Acción sobre *i*

2.1.4. Mecanismos de abstracción en los lenguajes de programación

La evolución de los lenguajes de programación se puede considerar, en gran medida, como un intento por incluir y dar soporte a mecanismos de abstracción cada vez más avanzados. En particular, la forma de proveer un mecanismo de definición y uso de Tipos Abstractos de Datos ha sido una de las principales motivaciones para la aparición de nuevos conceptos. La evolución va casi siempre en la dirección de ofrecer características

más próximas al concepto teórico de TAD: posibilidad de usar un TAD como un tipo cualquiera, principio de ocultación de la implementación, encapsulación y modularidad, genericidad, reutilización de TAD, posibilidad de usar iteradores.

Pero, ¿por qué es tan importante el concepto de Tipo Abstracto de Datos? Un lenguaje que no permite usar TAD –o un entorno de programación para el que no disponemos de TAD– es como una casa sin muebles y sin decoración. Es posible vivir en ella y realizar tareas rutinarias, aunque tendríamos que hacerlo de manera muy rudimentaria. Por ejemplo, deberíamos dormir en el suelo o beber agua chupando del grifo. Los TAD son como los muebles de la casa, no son estrictamente necesarios para vivir pero facilitan mucho las cosas. Los muebles, que hacen el papel de tipos abstractos:

- Aportan una funcionalidad extra fundamental, un valor añadido a la casa, permitiendo realizar las tareas comunes de manera más sencilla y cómoda.
- Pueden cambiarse por otros, moverse de sitio, arreglarlos en caso de rotura, etc., lo cual no es posible hacerlo –o resulta mucho más costoso– con la casa en sí.
- Los muebles y la decoración están adaptados a las necesidades, gustos y caprichos de cada uno, mientras que la casa es la misma para todo el bloque de apartamentos.
- Hay una separación clara entre la persona que fabrica el mueble y la que lo utiliza. Para usar un mueble no es necesario conocer nada de carpintería.

En definitiva, los TAD son uno de los conceptos más importantes en programación. Su uso permite construir aplicaciones cada vez más complejas, simplificando la utilización de funcionalidades adicionales. Los TAD pueden ser, por ejemplo, desde los enteros y los booleanos, hasta las ventanas de Windows, las conexiones a Internet o el propio sistema operativo.

Lenguajes de programación primitivos

Los lenguajes de programación primitivos (como Fortran o BASIC) ofrecían un conjunto predefinido de tipos elementales: entero, real, carácter, cadena, array, etc. Pero no existía la posibilidad de definir tipos nuevos. Si, por ejemplo, un usuario necesitaba usar una pila, debía definir y manejar las variables adecuadas de forma “manual”. Por ejemplo, el programa en notación Pascal sería algo como lo siguiente.

```

var
  PilaUno, PilaDos: array [1..100] of integer;
  TopePilaUno, TopePilaDos: integer;

begin
  PilaUno[TopePilaUno]:= 52;
  Write(PilaDos[TopePilaDos]);
  TopePilaUno:= TopePilaUno + 1;
  ...

```

Utilizar las pilas de esta manera resulta difícil y engorroso, y el código generado no se puede reutilizar en otras aplicaciones. Además, el programador debería tener en cuenta que PilaUno se corresponde con TopePilaUno y PilaDos con TopePilaDos. En estas circunstancias, es muy fácil equivocarse o acceder a posiciones no válidas del array.

Tipos definidos por el usuario

Los *tipos definidos por el usuario* son un primer paso hacia los TAD. Aparecen en lenguajes como C y Pascal, y permiten agrupar un conjunto de variables bajo un nombre del tipo. Estos tipos se pueden usar en el mismo contexto que un tipo elemental, es decir, para definir variables, parámetros o expresiones de ese tipo nuevo. En el ejemplo anterior, podríamos definir las pilas de enteros como un nuevo tipo definido por el usuario.

```
type
  Pila= record
    Tope: integer;
    Datos: array [1..100] of integer;
  end;

var
  PilaUno, PilaDos: Pila;

procedure Push (valor: integer; var pila: Pila);
procedure Pop (var pila: Pila);
...
```

El gran inconveniente de este mecanismo es que no permite garantizar el ocultamiento de la implementación. En teoría, las pilas sólo deberían ser usadas con las operaciones Push, Pop, etc., definidas sobre ellas. Pero, en su lugar, un programador podría hacer sin problemas: PilaUno.Datos[72]:=36; o PilaDos.Tope:=-12;. El lenguaje debería impedir las anteriores instrucciones, que acceden directamente a la representación del tipo.

Módulos y encapsulación

Los módulos o paquetes son un mecanismo de encapsulación; permiten tener agrupados bajo un mismo nombre una serie de tipos y operaciones relacionados. Y lo que es más interesante, en algunos casos, como en el lenguaje Modula-2, los módulos ofrecen ocultación de la implementación: los tipos definidos dentro de un módulo sólo se pueden usar a través de las operaciones de ese mismo módulo. De esta forma se evitarían los problemas vistos en el apartado anterior. El programa sería algo del siguiente tipo.

```
MODULE ModuloPilasEnt;
EXPORT PilaEntero, Push, Pop, Top, Nueva;

TYPE
  PilaEntero= RECORD
```

```
Tope: INTEGER;  
Datos: ARRAY [1..100] OF INTEGER;  
END;  
PROCEDURE Push (valor: integer; VAR pila: PilaEntero);  
PROCEDURE Pop (var pila: PilaEntero);  
...  
  
USE ModuloPilasEnt;  
VAR PilaUno, PilaDos: PilaEntero;  
...  
Push(23, PilaUno);  
Pop(PilaDos);  
...  
PilaUno.Datos[72]:= 36; ----> ERROR EN TIEMPO DE COMPILEACIÓN  
PilaDos.Tope:= -12; ----> ERROR EN TIEMPO DE COMPILEACIÓN  
...
```

El compilador impide las dos últimas instrucciones, indicando que los atributos Datos y Tope no son accesibles.

Clases y objetos

Hoy por hoy, las clases son el mecanismo más completo y extendido para definir y utilizar TAD. De hecho, su utilización implica tantos conceptos nuevos que hablamos de un paradigma de programación propio: la *programación orientada a objetos*. Una clase es, al mismo tiempo, un tipo definido por el usuario y un módulo donde se encapsulan los datos y operaciones de ese tipo. Un objeto es una instancia particular de una clase, un valor concreto.

Las clases permiten usar ocultamiento de la implementación. Para ello, en la definición de la clase es posible decir qué partes son visibles desde fuera (sección *public*) y cuáles no (sección *private*). En el ejemplo de las pilas, los atributos Datos y Tope deben ser privados, y Push, Pop, etc., públicos. La definición del tipo, usando el lenguaje ObjectPascal¹, sería como la siguiente.

```
type  
  PilaEntero= class  
    private:  
      Tope: integer;  
      Datos: array [1..100] of integer;  
    public:  
      procedure Inicializar;  
      procedure Push (valor: integer);  
      procedure Pop;  
      function Top: integer;  
  end;
```

¹Utilizamos aquí ObjectPascal para facilitar la comparación con las definiciones de los apartados anteriores. En las prácticas se usará el lenguaje C++.

```

...
var PilaUno, PilaDos: PilaEntero;
...
PilaUno.Push(23);
PilaDos.Pop;
Write(PilaUno.Top);
...
PilaUno.Datos[72]:= 36; ----> ERROR EN TIEMPO DE COMPILACIÓN
PilaDos.Tope:= -12;       ----> ERROR EN TIEMPO DE COMPILACIÓN
...

```

Las pilas, y en general cualquier objeto de una clase, sólo pueden ser manipuladas a través de las operaciones públicas de su definición. De esta forma, es posible conseguir la ocultación de la implementación.

Hay que notar que en la definición de las operaciones con pilas, la propia pila no aparece como un parámetro. Por ejemplo, donde antes teníamos: Pop(PilaUno) ahora tenemos: PilaUno.Pop. Esto es lo que se conoce como el **principio de acceso uniforme**: los atributos y las operaciones de una clase están conceptualmente al mismo nivel, y se accede a ellos usando la notación punto “.”. En la llamada PilaUno.Pop, el valor PilaUno (llamado “objeto receptor”) es un parámetro implícito de la operación Pop.

Algunos lenguajes orientados a objetos permiten aplicar *parametrización de tipo*, dando lugar a la definición de TAD genéricos o parametrizados. Por ejemplo, usando el lenguaje C++ es posible definir el tipo genérico Pila<T>, de pilas de cualquier tipo T. La definición se hace a través de la sentencia **template**.

```

template <class T>
class Pila {
private:
    T Datos [ ];
    int Maximo, Tope;
public:
    Pila (int max);      // Operación de inicialización de una pila
    ~Pila ();            // Operación de eliminación de una pila
    Push (T valor);
    Pop ();
    T Tope ();
};

...
Pila<int> PilaUno(100); // Instanciación a pila de enteros, con tamaño 100
Pila<char> PilaDos(200); // Instanciación a pila de caracteres, con tamaño 200
...
PilaUno.Push(5);
PilaDos.Push('w');
...

```

Dentro de las prácticas profundizaremos en el uso de las clases como un mecanismo de definición de tipos abstractos, y en la utilización de patrones para conseguir clases parametrizadas.

2.2. Especificaciones informales

En una especificación informal, normalmente la descripción de una abstracción se realiza textualmente utilizando el lenguaje natural. En consecuencia, las especificaciones informales pueden resultar ambiguas e imprecisas. Lo que una persona entiende de una forma, otra lo puede entender de manera distinta. Es más, puede que la especificación no sea completa, quedando parte del comportamiento sin especificar.

Una notación de especificación, formal o informal, define las partes que debe tener la especificación, las partes que pueden ser opcionales, y el significado y tipo de descripción de cada parte. Existe una gran variedad de notaciones de especificación informal, muchas de ellas adaptadas a lenguajes de programación y aplicaciones concretos. En este punto se estudia un formato no ligado a ningún lenguaje particular.

2.2.1. Especificación informal de abstracciones funcionales

La especificación informal de una abstracción funcional –es decir, de una rutina, función o un procedimiento– debe contener la siguiente información:

- Nombre de la operación.
- Lista de parámetros, tipo de cada parámetro y del valor devuelto, en su caso.
- Requisitos necesarios para ejecutar la operación.
- Descripción del resultado de la operación.

La notación propuesta para la especificación informal de abstracciones funcionales tiene el siguiente formato:

Operación <nombre> (ent** <id>:<tipo>; <id>:<tipo>;... ; **sal** <tipo_resultado>)**

Requiere: Descripción textual, donde se establecen los requisitos y restricciones de uso de la operación.

Modifica: Lista de parámetros de entrada que se modifican o pueden modificarse.

Calcula: Descripción textual del resultado de la operación.

Las cláusulas **Requiere** y **Modifica** sonopcionales, al igual que es posible que no existan parámetros de entrada o de salida. Si la función es parametrizada, entonces el formato sería del siguiente tipo:

Operación <nombre>[<id>:<tipo>;...] (ent** <id>:<tipo>; ... ; **sal** <tipo_result>)**

...

Aunque no resulta muy frecuente, la función puede estar parametrizada por más de un tipo genérico. Además, la cláusula **Requiere** puede contener restricciones sobre estos tipos genéricos. A continuación se muestran algunos ejemplos de especificaciones informales de abstracciones funcionales, usando el formato definido.

Ejemplo 2.1 Especificación informal de una operación **QuitarDuplicados**, para eliminar los elementos repetidos de un array de enteros.

Operación QuitarDuplicados (ent A: array [] de entero; sal entero)

Requiere: El tamaño de A debe ser mayor que cero.

Modifica: A

Calcula: Elimina los elementos duplicados del array A , manteniendo el orden de los elementos que no se repiten. Devuelve el número de los elementos no repetidos existentes.

Ejemplo 2.2 Especificación informal de una operación parametrizada RecorridoPreorden, para hacer el recorrido en preorden de un árbol binario de elementos de cualquier tipo.

Operación RecorridoPreorden [T: tipo] (ent A: ArbolBinario[T]; sal Lista[T])

Calcula: Devuelve como resultado una lista con los elementos del árbol A recorridos en preorden.

Ejemplo 2.3 Especificación informal de una operación parametrizada BuscarEnArray, para buscar un valor dentro de un array de cualquier tipo T .

Operación BuscarEnArray [T: tipo] (ent A: array [] de T; valor: T; sal entero)

Requiere: El tipo T debe tener definida una operación de comparación *Igual*(ent T, T ; sal booleano).

Calcula: Devuelve el menor valor de índice i , tal que *Igual(valor, A[i]) = verdadero*. Si no existe tal valor, entonces devuelve un número fuera del rango del array A .

Normalmente, las características del formato usado (partes de la especificación, nivel de detalle de la explicación, cosas que se describen más o menos) suelen variar mucho de una aplicación a otra. En la figura 2.2 se muestran dos ejemplos concretos de especificaciones informales en aplicaciones reales. En ambos casos, se puede decir que la descripción está *centrada en los parámetros*; se hace especial énfasis en el efecto de cada uno de los parámetros sobre el resultado final de la operación.

2.2.2. Especificación informal de abstracciones de datos

La especificación informal de un TAD describe el significado y uso del tipo, enumerando las operaciones que se pueden aplicar. La especificación debería contener al menos la siguiente información:

- Nombre del TAD.
- Si el TAD es un tipo parametrizado, número de parámetros. En ese caso, se pueden imponer restricciones sobre los tipos que se usen como parámetros en la instanciación.
- Lista de operaciones definidas sobre los valores del TAD.
- Especificación de cada una de las operaciones de la lista anterior.

La notación propuesta para la especificación informal de abstracciones de datos tiene la siguiente forma.

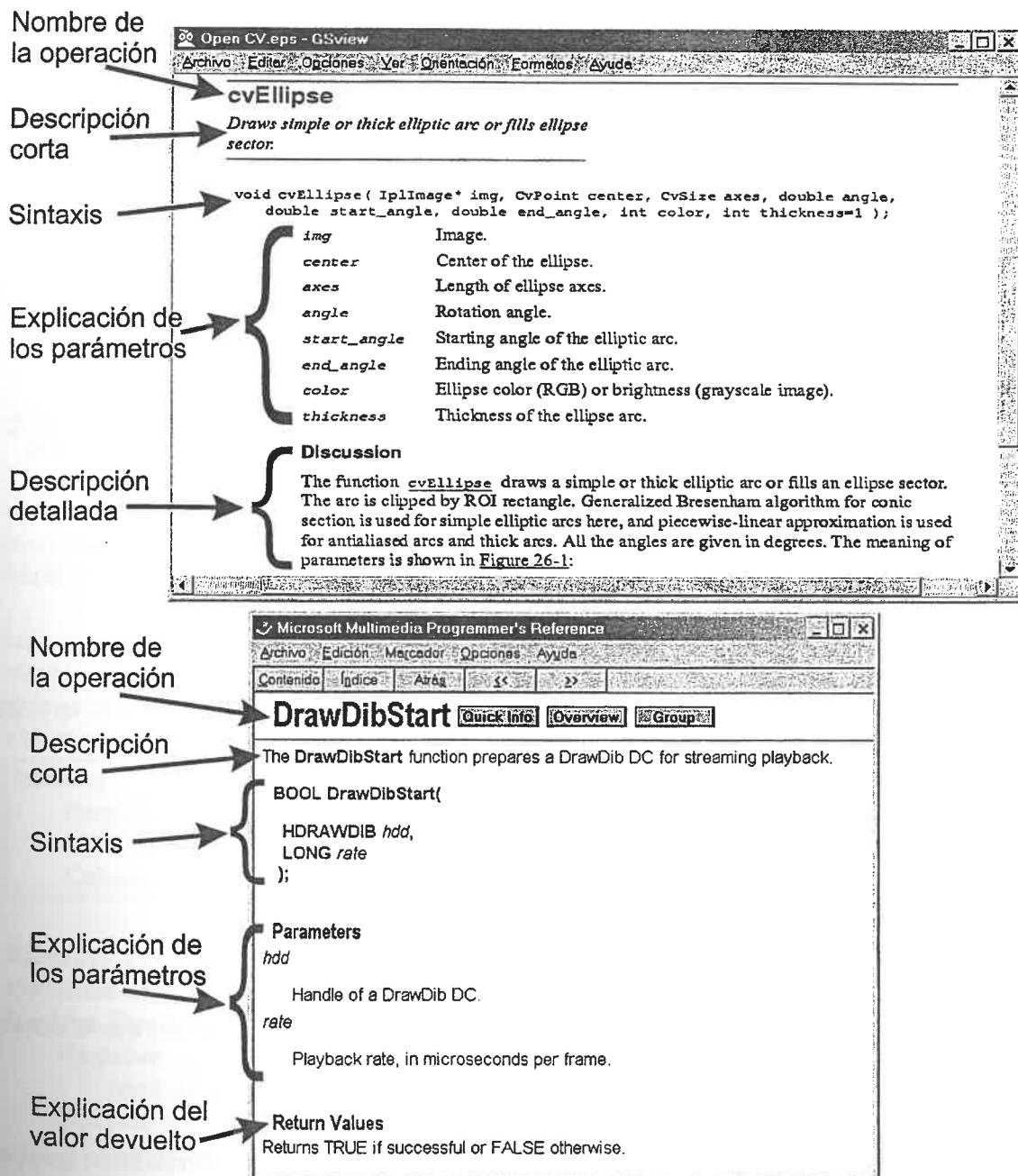


Figura 2.2: Especificaciones informales como documentación del código. Arriba: procedimiento "cvEllipse" de la librería Intel OpenCV. Abajo: función "DrawDibStart" de las librerías Microsoft Multimedia API.

TAD <nombre> es <lista de operaciones>

Descripción

Descripción textual del significado, comportamiento y modo de uso del tipo abstracto.

Operaciones

Especificación informal de cada una de las operaciones de <lista de operaciones>.

Fin <nombre>.

Vamos a ver algunas especificaciones de ejemplo, sobre tipos parametrizados y no parametrizados, con representaciones mutables o inmutables. Esta última distinción se puede realizar teniendo en cuenta la forma de las operaciones: en la representación inmutable los parámetros de entrada nunca se modifican.

Ejemplo 2.4 Especificación informal del TAD Polinomio.

TAD Polinomio es Crear, Grado, Coef, Sumar, Prod, Igual

Descripción

Los valores de tipo Polinomio son polinomios de coeficientes enteros no modificables.

Operaciones

Operación Crear (ent c, n: entero; sal Polinomio)

Requiere: $n \geq 0$.

Calcula: Devuelve el polinomio cx^n .

Operación Grado (ent P: Polinomio; sal entero)

Calcula: Devuelve el grado del polinomio P .

Operación Coef (ent P: Polinomio; n: entero; sal entero)

Calcula: Devuelve el coeficiente del polinomio P asociado al término x^n . Si no existe ese término, devuelve 0.

Operación Sumar (ent P, Q: Polinomio; sal booleano)

Calcula: Devuelve la suma de los polinomios $P+Q$.

Operación Prod (ent P, Q: Polinomio; sal booleano)

Calcula: Devuelve el producto de los polinomios $P \times Q$.

Operación Igual (ent P, Q: Polinomio; sal booleano)

Calcula: Devuelve verdadero si $P = Q$ y falso en caso contrario.

Fin Polinomio.

Ejemplo 2.5 Especificación informal del TAD parametrizado Asociacion, que almacena un par de valores (*clave, valor*).

TAD Asociacion[tclave, tvalor: tipo] es Crear, Clave, Valor, PonValor, Igual

Requiere

Los tipos tclave y tvalor deben tener definidas sendas operaciones de comparación *Igual*(ent *tclave, tclave; sal booleano*) e *Igual*(ent *tvalor, tvalor; sal booleano*).

Descripción

Los valores de tipo *Asociacion[tclave, tvalor]* son tuplas modificables, compuestas por el par (*clave: tclave, valor: tvalor*), que asocia una clave con un valor correspondiente.

Operaciones

Operación Crear (ent c: tclave; v: tvalor; sal Asociacion[tclave,tvalor])

Calcula: Devuelve la asociación compuesta por el par (*c, v*).

Operación Clave (ent A: Asociacion[tclave,tvalor]; sal tclave)

Calcula: Devuelve el primer elemento de la tupla A , es decir si $A = (c, v)$, devuelve c .

Operación Valor (ent A : Asociacion[tclave,tvalor]; sal tvalor)

Calcula: Devuelve el segundo elemento de la tupla A , es decir si $A = (c, v)$, devuelve v .

Operación PonValor (ent A : Asociacion[tclave,tvalor]; valor: tvalor)

Requiere: El nuevo valor debe ser distinto del antiguo, es decir $Igual(Valor(A), valor) = \text{falso}$.

Modifica: A

Calcula: Cambia el segundo elemento de la tupla A , poniendo $valor$.

Operación Igual (ent $A1, A2$: Asociacion[tclave,tvalor]; sal booleano)

Calcula: Devuelve *verdadero* si $A1$ y $A2$ contienen la misma clave y valor.

Fin Asociacion.

2.2.3. Especificación informal de abstracciones de iteradores

A diferencia de lo que ocurre con las abstracciones funcionales y de datos, no existe un claro consenso sobre el formato de las abstracciones de iteradores y la forma de implementarlas en los lenguajes de programación. Recordemos que la abstracción de iteradores implica un procesamiento repetido sobre los elementos de una colección.

Si utilizamos un lenguaje de programación que permita pasar funciones como parámetros, entonces la abstracción de iteradores se puede expresar como una abstracción funcional, donde un parámetro es la colección sobre la que iterar y el otro es la acción a aplicar. Por lo tanto, la especificación tendrá un formato similar a la especificación de abstracciones funcionales.

Iterador <nombre> (ent <id>:<tipo>; <id>:<tipo>; ... ; sal <tipo_resultado>)

Requiere: Descripción textual de los requisitos del iterador.

Modifica: Indicación de si se modifica la colección de entrada o no.

Calcula: Descripción textual del resultado del iterador.

Ejemplo 2.6 Especificación informal de un iterador ParaTodoHacer, para iterar sobre los elementos de un conjunto de cualquier tipo.

Iterador ParaTodoHacer [T: tipo] (ent C: Conjunto[T]; accion: Operacion)

Requiere: *accion* debe ser una operación que recibe un parámetro de tipo T y no devuelve nada, *accion*(*ent T*).

Calcula: Recorre todos los elementos c del conjunto C , aplicando sobre ellos la operación *accion(c)*.

Ejemplo 2.7 Especificación informal de un iterador Seleccionar, que devuelve todos los elementos de una colección que cumplen cierta condición.

Iterador Seleccionar [T: tipo; Coleccion: tipo_coleccion] (ent C: Coleccion[T]; condicion: Operacion; sal Coleccion[T])

Requiere: *condicion* debe ser una operación que recibe un parámetro de tipo T y devuelve un booleano, *condicion* (*ent T; sal booleano*). *tipo_coleccion* debe ser un TAD parametrizado, que incluya un iterador *ParaTodoHacer*.

Calcula: Devuelve una colección que contiene todos los elementos c de C tal que $condición(c)$ es verdadero.

Si el lenguaje no permite pasar procedimientos como parámetros, la abstracción de iteradores puede ser realizada a través de un TAD que proporcione operaciones para iniciar la iteración, obtener el elemento actual, avanzar al siguiente y comprobar si hemos procesado todos los elementos. En este caso, la notación toma una forma similar a la especificación de abstracciones de datos.

Tipoterador <nombre> es <lista de operaciones>

Requiere

Requisitos del iterador.

Descripción

Descripción textual del significado y modo de uso del iterador.

Operaciones

Especificación informal de cada una de las operaciones de <lista de operaciones>.

Fin <nombre>.

Ejemplo 2.8 Especificación informal de un iterador IteradorPreorden, que permite recorrer todos los elementos de un árbol binario en preorden.

Tipoterador IteradorPreorden [T: tipo] es Iniciar, Actual, Avanzar, EsUltimo

Descripción

Los valores de tipo *IteradorPreorden[T]* son iteradores definidos sobre árboles binarios de cualquier tipo T . Los elementos del árbol son devueltos en preorden. El iterador se debe inicializar con *Iniciar*.

Operaciones

Operación Iniciar (ent A: ArbolBinario[T]; sal IteradorPreorden)

Calcula: Devuelve un iterador nuevo, colocado sobre la raíz de A .

Operación Actual (ent iter: IteradorPreorden; sal T)

Calcula: Devuelve el elemento actual en la iteración.

Operación Avanzar (ent iter: IteradorPreorden)

Requiere: *EsUltimo(iter)* debe ser *falso*.

Modifica: *iter*

Calcula: Avanza el iterador *iter* al siguiente elemento en preorden, dentro del árbol binario sobre el que fue creado.

Operación EsUltimo (ent iter: IteradorPreorden; sal booleano)

Calcula: Devuelve *verdadero* si y sólo si el elemento actual de la iteración es el último.

Fin IteradorPreorden.

2.3. Especificaciones formales algebraicas

En una especificación formal, el comportamiento de una operación o de un TAD se describe de forma rigurosa utilizando una notación matemática no ambigua. En las

especificaciones algebraicas, o axiomáticas, la descripción se realiza a través de la definición de una serie de axiomas que relacionan unas operaciones con otras.

2.3.1. Propiedades, notación y ventajas de las especificaciones formales

Como hemos visto, las especificaciones informales presentan el grave inconveniente de la ambigüedad; la descripción textual puede ser poco precisa, incompleta e inexacta. En ciertas aplicaciones, conocer de forma rigurosa los requisitos y los resultados de una función puede resultar un factor crítico. En esos casos se hace necesario usar especificaciones formales, en las cuales las abstracciones son descritas mediante una notación matemática precisa, y por lo tanto no hay lugar a la ambigüedad.

Ventajas de las especificaciones formales

Además de la importante mejora que supone tener una especificación precisa de los procedimientos y tipos de datos, podemos señalar las siguientes ventajas de usar una notación formal de especificación:

- **Prototipado.** Al usar una descripción rigurosa, las especificaciones formales pueden ser procesadas por un ordenador, para simular la ejecución de un procedimiento o el uso de un TAD. De esta manera podríamos tener un prototipo de programa, antes de haberlo implementado, generado automáticamente a partir de las especificaciones.
- **Corrección de programas.** La especificación formal se puede usar para comprobar que la ejecución de un programa cumple el comportamiento deseado. Dado un programa y una especificación del mismo, sería posible introducir comprobaciones que garanticen que cada procedimiento verifica lo que determina su especificación.
- **Reutilización.** Una especificación formal escrita de forma genérica se puede reutilizar en distintos ámbitos, de manera que en cada problema no debemos partir de cero. Esto también es posible con la notación informal, aunque suele ser menos frecuente, debido a que las especificaciones informales normalmente están orientadas a aplicaciones concretas.

No obstante, en la práctica, la mayoría de las especificaciones utilizadas en la documentación de programas son de tipo informal, debido a la dificultad que supone especificar formalmente problemas de complejidad elevada. A pesar de ello, conviene conocer las notaciones formales e intentar crear especificaciones con el mayor grado de rigor y formalismo posible.

Formato de notación

Las notaciones de especificación formal que vamos a ver constan de cuatro partes: *NOMBRE*, *Conjuntos*, *Sintaxis* y *Semántica*. El formato de la especificación y el significado de cada parte es el siguiente.

NOMBRE

Nombre del TAD. Puede estar parametrizado, en cuyo caso se indicarán los parámetros entre paréntesis.

CONJUNTOS

Conjuntos que intervienen en la definición del TAD. Además del conjunto formado por los elementos del propio TAD definido, pueden aparecer otros conjuntos (por ejemplo, el conjunto B de booleanos, el conjunto Z de los enteros, los reales R , etc.).

SINTAXIS

Signatura de las operaciones definidas sobre el TAD, es decir, especificación del nombre, tipo de los valores de entrada y de salida. Para describir la sintaxis se utilizará la notación matemática para funciones, usando el formato:

$$<\text{nombre_operación}> : <\text{conjunto_dominio}> \rightarrow <\text{conjunto_resultado}>$$

Por ejemplo, la función *coseno*, que recibe un real y produce un real entre -1 y 1 tendrá la sintaxis: $\text{coseno} : R \rightarrow [-1, 1]$. Y una función *producto*, para multiplicar dos naturales será: $\text{producto} : N \times N \rightarrow N$.

SEMÁNTICA

Descripción del significado de cada operación.

En los dos tipos de especificaciones que vamos a ver, el formato de las tres primeras secciones es común; la diferencia se encuentra en la forma de describir la semántica de las operaciones. Como vamos a ver, en la especificación algebraica la semántica es descrita usando los axiomas, que relacionan unas operaciones con otras, mientras que en la operacional la descripción se realiza a través de una precondición y una postcondición.

2.3.2. Especificaciones algebraicas o axiomáticas

De acuerdo con el diccionario de la R.A.E., un **axioma** es una “proposición tan clara y evidente que se admite sin necesidad de demostración”. En una especificación algebraica, la semántica está constituida por un conjunto de axiomas, cada uno de los cuales relaciona el resultado de una operación con el resultado de otra. Un axioma tiene la forma:

$$<\text{Expresión 1}> = <\text{Expresión 2}>$$

Donde $<\text{Expresión 1}>$ y $<\text{Expresión 2}>$ son expresiones cualesquiera que usan operaciones de los tipos incluidos en la parte Conjuntos. El anterior axioma indica que ambas expresiones son equivalentes, es decir, en cualquier sitio donde aparezca $<\text{Expresión 1}>$ se puede sustituir por $<\text{Expresión 2}>$, y viceversa. Por ejemplo, si estamos definiendo las operaciones Suma y Producto del TAD Natural podríamos tener el axioma:

$$\text{Suma (dos, dos)} = \text{Producto (dos, dos)}$$

El anterior axioma relaciona la suma y el producto para un caso muy particular. Normalmente interesa definir axiomas más genéricos. Por ejemplo, suponiendo que a y b son naturales cualesquiera, tenemos los axiomas:

$$\text{Suma (a, b)} = \text{Suma (b, a)}$$

$$\text{Producto (a, b)} = \text{Producto (b, a)}$$

Veamos, como ejemplo, la especificación formal del TAD Booleano, compuesto por los valores lógicos *verdadero*, *falso*. Además de las operaciones lógicas usuales, se incluye un condicional de la forma *SI condición ENTONCES v1 SI NO v2*. Los axiomas son numerados para permitir su posterior referencia.

Ejemplo 2.9 Especificación formal algebraica del TAD Booleano.

NOMBRE

Booleano

CONJUNTOS

B Conjunto de valores booleanos

U Conjunto universal (compuesto por elementos de cualquier tipo)

SINTAXIS

verdadero : $\rightarrow B$

falso : $\rightarrow B$

Not : $B \rightarrow B$

And : $B \times B \rightarrow B$

Or : $B \times B \rightarrow B$

Si : $B \times U \times U \rightarrow U$

SEMÁNTICA

$\forall b \in B ; \forall u_1, u_2 \in U$

1. Not (verdadero) = falso

2. Not (falso) = verdadero

3. And (verdadero, b) = b

4. And (falso, b) = falso

5. Or (verdadero, b) = verdadero

6. Or (falso, b) = b

7. Si (verdadero, u1, u2) = u1

8. Si (falso, u1, u2) = u2

Consideraciones importantes

Del anterior ejemplo podemos extraer algunas conclusiones generales, y hacer algunas consideraciones a tener en cuenta en la construcción de especificaciones formales algebraicas.

- En la introducción vimos que un TAD está compuesto por un dominio y un conjunto de operaciones sobre ese dominio. Sin embargo, la especificación formal sólo describe las operaciones, de forma que los valores del dominio son implícitos. Para devolver valores particulares del dominio tenemos que usar operaciones sin parámetros (también llamadas operaciones *constantes*), como por ejemplo: falso : $\rightarrow B$; verdadero : $\rightarrow B$.
- Para especificar la sintaxis de una función con varios parámetros usamos el producto cartesiano \times . Por ejemplo, la función Or toma dos valores booleanos y devuelve un booleano: $Or : B \times B \rightarrow B$.

- Si una operación puede devolver valores de varios tipos distintos, entonces usamos la unión \cup de conjuntos. Por ejemplo, una operación de división entre enteros puede devolver un real, infinito o indefinido: Divide : $Z \times Z \rightarrow R \cup \{+\infty, -\infty, NoNat\}$
- En los axiomas sólo pueden aparecer variables cuantificadas² en la cláusula Semántica (en este caso b , $u1$ y $u2$) y operaciones que estén definidas formalmente en los tipos incluidos en la parte Conjuntos. Ninguna otra variable u operación se puede dar por supuesta. Por ejemplo, en la especificación de los booleanos no tendría sentido suponer que conocemos las operaciones lógicas \neg , \vee y \wedge .
- Aunque las funciones han sido definidas usando una notación prefija, en los siguientes ejemplos supondremos que se pueden aplicar también con sintaxis infija. En particular, para el condicional usaremos la sintaxis: Si $<\text{condicion}> \Rightarrow <\text{valor_si_cierto}> | <\text{valor_si_falso}>$
- Hay una diferencia muy importante entre el concepto matemático de función y el concepto informático. Una función matemática es una simple relación de un dominio de entrada con uno de salida, y por lo tanto *nunca jamás* puede modificar el valor de sus parámetros. La propia idea de modificar un parámetro de entrada carece de sentido en una función matemática. En las especificaciones formales trabajamos con el concepto matemático de función; en consecuencia, los tipos definidos con la especificación formal tendrán implícitamente una representación *immutable*.

2.3.3. Taxonomía de las operaciones de un TAD

En principio, todas las operaciones definidas en un TAD son igual de importantes y juegan el mismo papel dentro de la especificación. En la práctica es posible –y resulta de gran utilidad– distinguir entre distintos tipos de operaciones, en función del uso de las mismas en el manejo de valores del tipo. Podemos clasificar las operaciones de un TAD en tres grupos:

- **Operaciones de construcción o constructores.** Son el conjunto mínimo de operaciones a partir del cual se puede obtener cualquier valor del dominio del TAD definido. Por ejemplo, el par de operaciones: verdadero, falso, son dos constructores³ del TAD Booleano. La elección de los constructores no tiene por qué ser única. Por ejemplo, en este caso también se podrían tomar como constructores el par verdadero, Not; o el par falso, Not.
- **Operaciones de modificación.** Son todas las operaciones que devuelven un valor del TAD definido, pero que no son constructores. Por ejemplo, And y Or son operaciones de modificación dentro del TAD Booleano.

²Es decir, de las que aparecen en algún “para todo”.

³No confundir con el concepto de constructor en orientación a objetos. Un constructor OO es una operación que reserva memoria para una nueva instancia de una clase e inicializa sus campos con ciertos valores.

- **Operaciones de consulta.** Llamamos así a todas las operaciones que no devuelven un valor del TAD definido, sino de otro tipo cualquiera. Por ejemplo, el condicional Si devuelve un valor de tipo U, por lo que es una operación de consulta.

Consideremos la especificación formal algebraica del TAD Natural. Obviamente, no podemos suponer que conocemos nada de los números naturales –ni siquiera la igualdad entre naturales–, puesto que es precisamente lo que se está definiendo. Empezaremos viendo las tres primeras partes de la especificación.

NOMBRE

Natural

CONJUNTOS

N Conjunto de los naturales

B Conjunto de valores booleanos

SINTAXIS

cero : $\rightarrow N$

sucesor : $N \rightarrow N$

EsCero : $N \rightarrow B$

Suma : $N \times N \rightarrow N$

Igual : $N \times N \rightarrow B$

Doble : $N \rightarrow N$

Antes de estudiar la semántica de los naturales, vamos a clasificar las operaciones del TAD según la taxonomía realizada. En el siguiente apartado veremos los axiomas que hay que definir para conseguir una especificación válida. En este caso, tenemos los siguientes tipos de operaciones:

- **Constructores:** cero y sucesor. A partir de ellos es posible obtener cualquier número natural. Por ejemplo, el valor 1 sería sucesor(cero), el 2 sería sucesor(sucesor(cero)), el 3 sería sucesor(sucesor(sucesor(cero))) y así sucesivamente.
- **Modificación:** Suma y Doble. Dado uno o dos naturales devuelven un natural, y no son constructores.
- **Consulta:** EsCero e Igual. Ambas operaciones devuelven un valor de tipo booleano.

2.3.4. Completitud y corrección de la especificación

La cuestión clave en la creación de una especificación formal algebraica es determinar cuántos axiomas y cuáles deben ser incluidos en la sección de semántica para conseguir que la especificación formal sea válida. Para que una especificación algebraica sea válida, los axiomas deben ser los justos y suficientes para garantizar que se cumplen las dos siguientes propiedades:

- **Completitud.** Cualquier expresión del TAD se debe poder resolver, es decir encontrar una expresión equivalente más sencilla. Normalmente, este resultado será una expresión en la que sólo aparezcan operaciones de construcción. Por ejemplo, el resultado final de una expresión booleana será verdadero o falso, y si la expresión es de tipo Natural el resultado debe ser cero, sucesor(cero), ..., sucesor(...(cero)...).

- **Corrección.** Dada una expresión, sólo se debe poder obtener un resultado final. En otro caso, habría alguna contradicción en los axiomas. Por ejemplo, si a partir de una expresión booleana podemos obtener el resultado verdadero, aplicando ciertos axiomas, y por otro lado el resultado falso, aplicando otros axiomas, entonces habría un error de corrección. Algunos de los axiomas introducidos sería contradictorios.

Por lo tanto, no se trata de incluir muchos axiomas en la semántica, y de forma desordenada, sino de encontrar los necesarios para conseguir las dos propiedades anteriores. En concreto, la propiedad de completitud implica que deben haber axiomas para resolver todas las posibles expresiones usando las operaciones del TAD. Luego todas las operaciones deben estar definidas con los posibles valores de entrada. Todas excepto los constructores, puesto que si la expresión sólo usa constructores, entonces ya tendremos el resultado final.

Aunque el problema no se puede resolver automáticamente, existe un método que ayuda a determinar de forma más o menos sistemática los axiomas que se deben incluir. El método consta de dos pasos.

1. Distinguir cuáles son los constructores del TAD y cuáles son las operaciones de modificación y consulta.
2. Introducir los axiomas necesarios y suficientes para relacionar, de la forma más general posible, cada operación de modificación y consulta con cada uno de los constructores.

Aplicando el método sobre el TAD Natural, tenemos que los constructores son cero y sucesor y el resto de operaciones son de modificación o consulta. Por lo tanto, los axiomas deben relacionar todas esas operaciones con ambos constructores. El resultado debe ser una expresión más sencilla, esto es, una expresión usando los constructores. Por ejemplo, siendo n y m enteros cualesquiera, podemos definir la operación EsCero mediante los dos siguientes axiomas:

$$\text{EsCero}(\text{cero}) = \text{verdadero}$$

$$\text{EsCero}(\text{sucesor}(n)) = \text{falso}$$

Para la operación Igual necesitamos cuatro axiomas:

$$\text{Igual}(\text{cero}, \text{cero}) = \dots; \quad \text{Igual}(\text{cero}, \text{sucesor}(n)) = \dots$$

$$\text{Igual}(\text{sucesor}(n), \text{cero}) = \dots; \quad \text{Igual}(\text{sucesor}(n), \text{sucesor}(m)) = \dots$$

Como se puede observar, el proceso no es completamente automático. Para la operación Igual se pueden simplificar los axiomas en sólo tres casos. Por otro lado, los dos axiomas de la operación Doble también pueden ser simplificados, quedándonos con un sólo axioma. En definitiva una posible semántica para la especificación algebraica del TAD Natural podría ser la siguiente.

SEMÁNTICA

$$\forall n, m \in \mathbb{N}$$

1. $\text{EsCero}(\text{cero}) = \text{verdadero}$
2. $\text{EsCero}(\text{sucesor}(n)) = \text{falso}$
3. $\text{Suma}(\text{cero}, m) = m$
4. $\text{Suma}(\text{sucesor}(n), m) = \text{sucesor}(\text{Suma}(n, m))$

5. Igual (cero, n) = EsCero (n)
6. Igual (sucesor (n), cero) = falso
7. Igual (sucesor (n), sucesor (m)) = Igual (n, m)
8. Doble (n) = Suma (n, n)

Las especificaciones algebraicas tienen un marcado carácter recursivo: la mayoría de los axiomas son definidos de forma recurrente, utilizando un razonamiento inductivo. Por ejemplo, el axioma 7 del ejemplo anterior expresa el resultado de la operación Igual en función del ella misma, para enteros de menor valor. Los axiomas 5 y 6 serían casos base de esta recursividad. Para la Suma, el axioma 3 sería un caso base, mientras que el 4 tiene el efecto de “sacar fuera” una operación sucesor dentro del primer parámetro, y por lo tanto nos acerca al caso base anterior.

2.3.5. Reducción de expresiones algebraicas

La reducción de una expresión algebraica es el proceso mediante el cual se van aplicando los axiomas de la semántica, hasta obtener un resultado final para esa expresión. En principio, todos los axiomas tienen la misma importancia y además se pueden aplicar en ambos sentidos de la igualdad⁴. En la práctica, se puede imponer un orden, de manera que si se pueden aplicar varios axiomas se aplique primero el de menor número. Esto eliminaría la ambigüedad en los casos donde varios axiomas puedan ser aplicados. Además, tal y como han sido definidos, el sentido de aplicación suele ser sustituir la parte izquierda por la parte derecha.

El resultado final de una expresión, siempre que esté bien construida, será una expresión usando los constructores. No obstante, si en la expresión aparecen variables de valor no determinado, el resultado puede no estar sólo en función de los constructores.

Ejemplo 2.10 Reducción de la expresión $\text{Suma}(\text{sucesor}(\text{sucesor}(\text{cero})), \text{sucesor}(\text{cero}))$.

$\text{Suma}(\text{sucesor}(\text{sucesor}(\text{cero})), \text{sucesor}(\text{cero})) = [\text{Axioma 4, con } n = \text{sucesor}(\text{cero}), m = \text{sucesor}(\text{cero})]$

$\text{sucesor}(\text{Suma}(\text{sucesor}(\text{cero}), \text{sucesor}(\text{cero}))) = [\text{Axioma 4, con } n = \text{cero}, m = \text{sucesor}(\text{cero})]$

$\text{sucesor}(\text{sucesor}(\text{Suma}(\text{cero}, \text{sucesor}(\text{cero})))) = [\text{Axioma 3, con } m = \text{sucesor}(\text{cero})]$

$\text{sucesor}(\text{sucesor}(\text{sucesor}(\text{cero})))$

Vamos a ver ahora la especificación algebraica del TAD contenedor genérico $\text{Pila}[T]$ y la reducción de algunas expresiones de ejemplo. En este caso, los constructores del tipo son las operaciones `crearPila` y `push`; la operación de modificación es `pop` y las operaciones `top` (elemento en el tope de la pila), `esVacía` e `igual` son de consulta.

En caso de aplicar la operación `top` sobre una pila vacía ocurre un error. Para indicarlo añadimos el conjunto M de mensajes de error, de forma que la sintaxis de `top` es: $S \rightarrow T \cup M$. Para la operación `pop` decidimos que al aplicarla sobre una pila vacía devuelva la pila vacía.

Ejemplo 2.11 Especificación formal algebraica del TAD genérico o parametrizado $\text{Pila}[T]$, de pilas de elementos de un tipo cualquiera T .

⁴Recordemos que, en las especificaciones algebraicas, el significado de los axiomas no es “valor devuelto” o “asignación”, sino “equivalencia entre expresiones”.

NOMBRE

Pila[T]

CONJUNTOS

S Conjunto de pilas de tipo T

T Conjunto de elementos que pueden ser almacenados

B conjunto de valores booleanos

M Conjunto de mensajes de error { "La lista está vacía" }

SINTAXIScrearPila : \rightarrow Spush : $T \times S \rightarrow S$ top : $S \rightarrow T \cup M$ pop : $S \rightarrow S$ esVacia: $S \rightarrow B$ igual: $S \times S \rightarrow B$ **SEMÁNTICA** $\forall t, t_2 \in T; \forall s, s_2 \in S$

1. top(crearPila) = "La pila está vacía"

2. top(push(t, s)) = t

3. pop(crearPila) = crearPila

4. pop(push(t, s)) = s

5. esVacia(crearPila) = verdadero

6. esVacia(push(t, s)) = falso

7. igual(crearPila, crearPila) = verdadero

8. igual(crearPila, push(t, s)) = falso

9. igual(push(t, s), crearPila) = falso

10. igual(push(t, s), push(t₂, s₂)) = (t=t₂) AND igual(s, s₂)**Ejemplo 2.12** Reducción de la expresión top (pop (push (t1, push (t2, push (t3, a))))).

top (pop (push (t1, push (t2, push (t3, a))))) = [Axioma 4, con t= t1, s= push(t2,push(t3,a))]

top (push (t2, push(t3, a))) = [Axioma 2, con t= t2, s= push(t3,a)]

t2

Ejemplo 2.13 Reducción de la expresión push(top(pop(crearPila)), crearPila).

push(top(pop(crearPila)), crearPila) = [Axioma 3]

push(top(crearPila), crearPila) = [Axioma 1]

push("La pila está vacía", crearPila)

ERROR en la expresión. La operación push espera un parámetro de tipo T.

En este segundo ejemplo, la expresión es errónea y no se puede seguir simplificando.

Es decir, el resultado no es un mensaje de error sino que la expresión push(top(pop(crearPila))) es equivalente a push("La pila está vacía", crearPila), la cual es una expresión errónea. Un resultado de expresión errónea lo podríamos obtener también si intentáramos reducir una expresión como pop(cero), que no respeta la sintaxis de la operación.

Otra posibilidad es añadir a la especificación formal una facilidad sintáctica, que diga algo del tipo: "Si aparece un mensaje a la entrada de una operación, el resultado será el mismo mensaje". Las cláusulas de este tipo, aplicables en cualquier punto de la especificación, se conocen normalmente como **assertos invariantes**. De esta forma,

incluyendo el anterior comportamiento como aserto invariante del tipo Pila, el resultado final de la expresión sería: "La pila está vacía".

2.4. Especificaciones formales constructivas

En una especificación formal constructiva, u operacional, el significado de las operaciones se establece mediante las cláusulas de **precondición** y **postcondición**. Al contrario de las especificaciones algebraicas –donde las operaciones se definen a través de las relaciones entre ellas– en las especificaciones constructivas cada operación es descrita independientemente de las demás. Por este motivo, se dice que en el método algebraico el significado de las operaciones es *implícito*, mientras que en el constructivo el significado es *explícito*.

El formato de notación utilizado en las especificaciones constructivas coincide con el ya visto para las algebraicas en las partes Nombre, Conjuntos y Sintaxis. La diferencia está en la parte Semántica, que contendrá la pre- y postcondición de cada operación.

2.4.1. Precondiciones y postcondiciones

La precondición y la postcondición son condiciones lógicas que indican, respectivamente, los requisitos de una operación y su efecto.

- **Precondición.** Relación lógica que deben de cumplir los valores de entrada para que la operación se pueda aplicar con garantías de producir una ejecución correcta.
- **Postcondición.** Relación lógica que se cumple con los valores de salida después de ejecutar la operación, siempre que se cumpliera la precondición.

Las pre- y postcondiciones son también llamadas **asertos**: afirmaciones que deben ser ciertas antes y después de ejecutar una operación, respectivamente. Para cada operación **<nombre>** definida en la parte Sintaxis, dentro de la parte Semántica deben aparecer las dos siguientes cláusulas:

```
pre-<nombre>(<param_entrada>):= <condición_lógica>
post-<nombre>(<param_entrada>;<param_salida>):= <condición_lógica>
```

Ejemplo 2.14 Especificación constructiva de una operación **maximo**, que tiene como entrada dos números reales y devuelve como resultado el mayor de los dos.

maximo: R × R → R

```
pre-maximo(x,y):= true
post-maximo(x,y;r):= (r ≥ x) ∧ (r ≥ y) ∧ (r = x ∨ r = y)
```

Ejemplo 2.15 Especificación constructiva de una operación **max_restring**, máximo restringido, que calcula el máximo de dos números, pero sólo se puede aplicar a enteros positivos.

max_restring: Z × Z → Z

```
pre-max_restring(x,y):= (x > 0) ∧ (y > 0)
post-max_restring(x,y;r):= (r ≥ x) ∧ (r ≥ y) ∧ (r = x ∨ r = y)
```

Como se ve en los ejemplos anteriores, la postcondición no debe indicar necesariamente un valor concreto para el resultado. Simplemente expresa una condición lógica que es cierta. Igual que en las especificaciones algebraicas, es posible usar condicionales de la forma *SI* <condición> \Rightarrow <valor_si_cierto> | <valor_si_falso>. Además, también se pueden usar los cuantificadores \forall y \exists .

Ejemplo 2.16 Especificación constructiva de una operación **BusquedaBinaria**, que realiza una búsqueda binaria dentro de un array de elementos de un tipo parametrizado **T**.

BusquedaBinaria: $\text{Array}[T] \times T \rightarrow \mathbb{N}$

pre-BusquedaBinaria(*a, t*) ::= $\forall i = 1, \dots, \text{Tamaño}(a) - 1; a[i] \leq a[i + 1]$
post-BusquedaBinaria(*a, t; n*) ::= *SI* $t < a[1] \Rightarrow n = 0$ | *SI* $t > a[\text{Tamaño}(a)] \Rightarrow n = \text{Tamaño}(a) + 1$ | ($a[n] = t$) $\vee (a[n] < t < a[n + 1])$

Se supone que la operación **Tamaño**, definida sobre arrays, devuelve el número de elementos del array, y que el tipo parametrizado **T** tiene definidas operaciones de comparación entre valores del tipo.

En este caso, la precondition expresa el requisito de que el array de entrada debe estar ordenado de menor a mayor. En tal caso, la operación devuelve la posición donde se encuentra, o se debería encontrar, el valor buscado. Este resultado es lo que se indica en la postcondición. Pero se puede ver que la postcondición no dice nada de cómo se busca el elemento. Es un detalle de implementación que no debe aparecer en la especificación.

Error frecuente 2.1 Es muy importante no confundir *especificación* e *implementación*: la especificación describe el resultado esperado de una operación; la implementación es un trozo de código que calcula ese resultado. Por lo tanto, la especificación debe ser independiente de la implementación. Sin embargo, el método constructivo permite definir especificaciones que son muy próximas a implementaciones. Por ejemplo, una posible postcondición para la operación **máximo** podría ser:

post-maximo(*x, y; r*) ::= *SI* $x > y \Rightarrow r = x$ | $r = y$

En este ejemplo sencillo, se ve más claramente el significado de la operación. Sin embargo, en general, hay que evitar este tipo de especificaciones porque mezclan especificación con implementación, ofreciendo al usuario de la abstracción detalles innecesarios de programación.

2.4.2. Especificación como contrato de una operación

El significado de una especificación constructiva es: si se cumple la precondition y se ejecuta la operación, entonces se puede asegurar que se cumple la postcondición. Una buena analogía para una especificación de este tipo es un **contrato** —que conlleva ciertos derechos y obligaciones— que se establece entre el que implementa la operación y el que la usa. La idea de la especificación como un contrato es mostrada en la figura 2.3.



Figura 2.3: La especificación de un producto software como contrato entre el que lo programa y el que lo usa.

En concreto, el contrato en la especificación constructiva estipula lo siguiente:

IMPLEMENTADOR	
Derechos Supone que se cumple la precondición, sin preocuparse de qué hacer en caso contrario. Esto le evita tener que comprobarla.	Obligaciones Debe hacer los cálculos necesarios para conseguir que se cumpla la postcondición al acabar la ejecución de la operación.
USUARIO	
Obligaciones Es el responsable exclusivo de garantizar que se cumpla la precondición, antes de ejecutar la operación.	Derechos Obtiene los resultados que se indican en la postcondición, siempre que haya cumplido sus obligaciones.

Así que el responsable de comprobar la precondición es el usuario de la operación, y el responsable de garantizar la postcondición es el implementador de la operación. Este reparto de responsabilidades ayuda a simplificar la construcción de programas, puesto que se evitan comprobaciones redundantes e innecesarias.

Pero, ¿qué ocurre si no se cumple una precondición? Simplemente no se garantiza que se cumpla la postcondición. En tales casos, el creador de la especificación puede elegir básicamente entre dos opciones:

- No especificar nada. En caso de error en los parámetros de entrada, el efecto será imprevisible; puede que en algunas situaciones se cumpla la postcondición, en otras no, o incluso puede que el programa se cuelgue.
- Especificar un comportamiento estándar en caso de fallo de la precondición. Este comportamiento puede ser, por ejemplo, interrumpir la ejecución del programa, seguir como si no se hubiera ejecutado la operación, indicar la situación en una variable de error o provocar una excepción.

En la práctica, el uso de excepciones suele ser la solución más utilizada. La ejecución de una excepción implica que la operación interrumpe su ejecución. El control pasa al procedimiento que ha ejecutado la llamada, que tendrá definido un código para el tratamiento de excepciones. En caso contrario, la excepción pasará al que lo ha llamado y así sucesivamente.

Una tercera opción sería incluir el tratamiento del error dentro de la operación que la implementa y también dentro de la propia especificación formal. Igual que en la especificación algebraica, se debería indicar en la sintaxis que la operación puede devolver un mensaje de error. Por ejemplo, vamos a suponer que en la especificación constructiva de la operación máximo restringido incluimos el tratamiento de errores, en caso de que alguno de los parámetros no sea positivo. La especificación sería:

max_restring2: $Z \times Z \rightarrow Z \cup \{\text{"Error de precondición"}\}$

```

pre-max_restring2( $x, y$ ):= verdadero
post-max_restring2( $x, y; r$ ):= SI ( $x > 0 \wedge y > 0$ )
 $\Rightarrow (r \geq y) \wedge (r \geq y) \wedge (r = x \vee r = y)$ 
| "Error de precondición"

```

Está claro que esta elección rompe la filosofía de los contratos: el implementador comprueba y trata las condiciones de error en los parámetros de entrada, lo cual no debería ser su responsabilidad. Al definirlo de esta manera, se puede decir que la especificación “oculta” los verdaderos requisitos de la operación, puesto que la precondición siempre es cierta. En cierto sentido, el cliente es “engañoado”: cree que puede ejecutar la operación sin problemas, pero se encuentra con mensajes de error inesperados.

2.4.3. Necesidad de un modelo subyacente

Si el TAD que se está definiendo es un tipo complejo, expresar las precondiciones y las postcondiciones usando únicamente operaciones lógicas puede ser difícil o inviable. En esos casos es posible utilizar otro tipo de datos como **tipo subyacente**, en el cual se basa la especificación del tipo que está siendo definido. Este tipo o modelo subyacente debería estar especificado formalmente, ya sea por el método algebraico o por el constructivo.

Un ejemplo de TAD donde resulta necesario un modelo subyacente son los tipos colección o contenedores. Para definir de forma constructiva cualquier contenedor es necesario usar como modelo subyacente algún otro tipo contenedor. En última instancia necesitaremos algún TAD contenedor cuya definición no esté basada en otro contenedor; este tipo subyacente será especificado de forma algebraica.

Vamos a ver como ejemplo la especificación formal constructiva de los TAD genéricos **Pila[T]** y **Cola[T]**. La definición de estos tipos se basará en el modelo subyacente **Lista[T]**, el cual será definido de forma algebraica.

Ejemplo 2.17 Especificación formal algebraica del TAD parametrizado **Lista[T]**, que define las listas de un tipo cualquiera **T**. Las listas se crean con las operaciones: **crearLista**, que devuelve una lista vacía; **formarLista**, que devuelve una lista con un sólo elemento; y **concatenar** que une dos listas en una.

NOMBRE

Lista[T]

CONJUNTOS

- L Conjunto de listas de tipo T
- T Conjunto de elementos que pueden ser almacenados
- B Conjunto de valores booleanos
- N Conjunto de naturales
- M Conjunto de mensajes { "La lista está vacía" }

SINTAXIS

- crearLista : $\rightarrow L$
- formarLista : $T \rightarrow L$
- concatenar : $L \times L \rightarrow L$
- primero : $L \rightarrow T \cup M$
- último : $L \rightarrow T \cup M$
- cabecera : $L \rightarrow L$
- cola : $L \rightarrow L$
- longitud : $L \rightarrow N$
- esVacía : $L \rightarrow B$

SEMÁNTICA

- $\forall t \in T ; \forall a, b \in L$
- 1. $\text{primero}(\text{crearLista}) = \text{"La lista está vacía"}$
- 2. $\text{primero}(\text{formarLista}(t)) = t$
- 3. $\text{primero}(\text{concatenar}(a, b)) = \text{SI } \text{esVacía}(a) \Rightarrow \text{primero}(b) \mid \text{primero}(a)$
- 4. $\text{último}(\text{crearLista}) = \text{"La lista está vacía"}$
- 5. $\text{último}(\text{formarLista}(t)) = t$
- 6. $\text{último}(\text{concatenar}(a, b)) = \text{SI } \text{esVacía}(b) \Rightarrow \text{último}(a) \mid \text{último}(b)$
- 7. $\text{cabecera}(\text{crearLista}) = \text{crearLista}$
- 8. $\text{cabecera}(\text{formarLista}(t)) = \text{crearLista}$
- 9. $\text{cabecera}(\text{concatenar}(a, b)) = \text{SI } \text{esVacía}(b) \Rightarrow \text{cabecera}(a) \mid \text{concatenar}(a, \text{cabecera}(b))$
- 10. $\text{cola}(\text{crearLista}) = \text{crearLista}$
- 11. $\text{cola}(\text{formarLista}(t)) = \text{crearLista}$
- 12. $\text{cola}(\text{concatenar}(a, b)) = \text{SI } \text{esVacía}(a) \Rightarrow \text{cola}(b) \mid \text{concatenar}(\text{cola}(a), b)$
- 13. $\text{longitud}(\text{crearLista}) = \text{cero}$
- 14. $\text{longitud}(\text{formarLista}(t)) = \text{sucesor } (\text{cero})$
- 15. $\text{longitud}(\text{concatenar}(a, b)) = \text{suma}(\text{longitud}(a), \text{longitud}(b))$
- 16. $\text{esVacía}(\text{crearLista}) = \text{verdadero}$
- 17. $\text{esVacía}(\text{formarLista}(t)) = \text{falso}$
- 18. $\text{esVacía}(\text{concatenar}(a, b)) = \text{esVacía}(a) \text{ AND } \text{esVacía}(b)$

Ejemplo 2.18 Especificación formal constructiva del TAD parametrizado Cola[T], usando como modelo subyacente el tipo Lista[T].

NOMBRE

Cola[T]

CONJUNTOS

- Q Conjunto de colas de tipo T
- L Conjunto de listas de tipo T
- T Conjunto de elementos que pueden ser almacenados

B Conjunto de valores booleanos

SINTAXIS

crearCola : $\rightarrow Q$
 frente : $Q \rightarrow T$
 inserta : $T \times Q \rightarrow Q$
 resto : $Q \rightarrow Q$
 esVacíaCola: $Q \rightarrow B$

SEMÁNTICA

$\forall t \in T; \forall p, q \in Q; \forall b \in B$
 pre-crearCola() ::= verdadero
 post-crearCola(); q ::= q=crearLista
 pre-frente(q) ::= not esVacía(q)
 post-frente(q; t) ::= t=primero(q)
 pre-inserta(t, q) ::= verdadero
 post-inserta(t, q; p) ::= p=concatenar(q, formarLista(t))
 pre-resto(q) ::= not esVacía(q)
 post-resto(q; p) ::= p=cola(q)
 pre-esVacíaCola(q) ::= verdadero
 post-esVacíaCola(q; b) ::= b=esVacía(q)

Ejemplo 2.19 Especificación formal constructiva del TAD parametrizado Pila[T], usando como modelo subyacente el tipo Lista[T]. Se puede ver que la única diferencia con la especificación del TAD Cola[T] es el orden de inserción de los elementos en la operación push.

NOMBRE

Pila[T]

CONJUNTOS

S Conjunto de pilas de tipo T
 L Conjunto de listas de tipo T
 T Conjunto de elementos que pueden ser almacenados
 B Conjunto de valores booleanos

SINTAXIS

crearPila : $\rightarrow S$
 tope : $S \rightarrow T$
 push : $T \times S \rightarrow S$
 pop : $S \rightarrow S$
 esVacíaPila : $S \rightarrow B$

SEMÁNTICA

$\forall t \in T; \forall s, r \in S; \forall b \in B$
 pre-crearPila() ::= verdadero
 post-crearPila(); s ::= s=crearLista
 pre-tope(s) ::= not esVacía(s)
 post-tope(s; t) ::= t=primero(s)
 pre-push(t, s) ::= verdadero
 post-push(t, s; r) ::= r=concatenar(formarLista(t), s)

pre-pop(s)::= not esVacía(s)
post-pop(s; r)::= r==cola(s)
pre-esVacíaPila(s)::= verdadero
post-esVacíaPila(s; b)::= b==esVacía(s)

2.4.4. Ejecución de especificaciones constructivas

Como vimos en la introducción del apartado 2.3.2, una de las ventajas de las especificaciones formales, respecto a las informales, es la posibilidad de ejecutar o, más exactamente, simular las especificaciones. La ejecución de una expresión constructiva consiste en comprobar las precondiciones y postcondiciones. Para cada llamada, se comprueba la precondición; si es cierta, entonces se puede dar por válida la postcondición; si no se cumple la precondición, el mecanismo de ejecución de la especificación indicaría el error y el punto donde se ha producido.

Ejemplo 2.20 Vamos a comprobar el resultado obtenido con la especificación constructiva del tipo Pila[Natural] para la siguiente expresión:

tope(push(4, pop(push(2, crearPila))))

Operación	Precondición	Postcondición
1. crearPila	verdadero	$o1 = \text{crearLista}$
2. push(2, o1)	verdadero	$o2 = \text{concatenar}(\text{formarLista}(2), o1)$
3. pop(o2)	$\text{not esVacía}(o2) = [\text{Ax. 18}] \text{ not } (\text{esVacía}(\text{formarLista}(2)) \text{ AND } \text{esVacía}(\text{crearLista})) = [\text{Ax. 16,17}] \text{ not } (\text{falso AND verdadero}) = \text{not falso} = \text{verdadero}$	$o3 = \text{cola}(o2) = [\text{Ax. 12}] \text{ Si esVacía}(\text{formarLista}(2)) \Rightarrow \text{cola}(\text{crearLista}) \mid \text{concatenar}(\text{cola}(\text{formarLista}(2)), \text{crearLista}) = [\text{Ax. 17}] \text{ concatenar}(\text{cola}(\text{formarLista}(2)), \text{crearLista}) = [\text{Ax. 11}] \text{ concatenar}(\text{crearLista}, \text{crearLista})$
4. push(4, o3)	verdadero	$o4 = \text{concatenar}(\text{formarLista}(4), o3)$
5. tope(o4)	$\text{not esVacía}(o4) = [\text{Ax. 18}] \text{ not } (\text{esVacía}(\text{formarLista}(4)) \text{ AND } \text{esVacía}(o3)) = [\text{Ax. 17}] \text{ not } (\text{falso AND esVacía}(o3)) = \text{not falso} = \text{verdadero}$	$o5 = \text{primero}(o4) = [\text{Ax. 3}] \text{ Si esVacía}(\text{formarLista}(4)) \Rightarrow \text{primero}(o3) \mid \text{primero}(\text{formarLista}(4)) = [\text{Ax. 17}] \text{ primero}(\text{formarLista}(4)) = [\text{Ax. 2}] 4$

Ejemplo 2.21 Vamos a comprobar ahora el resultado obtenido usando la especificación del TAD Cola[Natural] para la expresión:

esVacíaCola(inserta(1, resto(createCola)))

Operación	Precondición	Postcondición
1. crearCola	verdadero	$o1 = \text{crearLista}$
2. resto(o1)	$\text{not esVacía}(o1) = \text{not esVacía}(\text{crearLista}) = [\text{Ax. 16}] \text{ not verdadero} = \text{falso}$	ERROR en la expresión resto(createCola). No se cumple la precondición. Ejecución interrumpida.

Otra interesante aplicación de las especificaciones constructivas es comprobar la corrección de programas. El concepto de **corrección de un programa** es un concepto relativo: un programa se dice que es correcto si cumple su especificación. Usando especificaciones constructivas es posible comprobar la ejecución correcta de un programa. Para ello, la precondición y la postcondición son añadidas como comprobaciones al principio

y al final de cada procedimiento, respectivamente. Un fallo de la precondición indica un error en el procedimiento que hace la llamada; un fallo en la postcondición indica un error de programación en la función correspondiente.

Por ejemplo, utilizando C++ las pre- y postcondiciones pueden ser comprobadas usando la función `assert(condición)`. Si en el momento de ejecutarse la función `assert` la condición es cierta, se sigue la ejecución normalmente. Si la condición es falsa, se genera una excepción, indicando el punto donde se ha producido.

Ejemplo 2.22 Implementación de precondiciones y postcondiciones en C++. Especificación formal constructiva de una operación `RaizCuarta` que, dado un número real positivo x , calcula $\sqrt[4]{x}$.

RaizCuarta : R → R

- **pre-RaizCuarta(x)::=** $x \geq 0$
- **post-RaizCuarta(x; r)::=** $r^4 = x$

Implementación de la operación en C++, incluyendo la comprobación de precondición y postcondición.

```
float RaizCuarta (float x)
{
    assert (x>=0);           // PRECONDICIÓN
    float r;
    r= sqrt(x);
    if (r!=0.0) r= sqrt(r);
    assert (r*r*r*r==x);    // POSTCONDICIÓN
    return r;
}
```

En este caso concreto, hay que llevar cuidado con los redondeos de los números en punto flotante. Con toda probabilidad, multiplicar cuatro veces la raíz cuarta de x no dará exactamente x . Una postcondición más adecuada podría ser algo como:

```
assert (fabs(r*r*r*r-x)<= 0.0001); // POSTCONDICIÓN
```

Ejercicios resueltos

Ejercicio 2.1 Desarrollar una especificación formal para el TAD genérico `Conjunto[T]`, por el método axiomático o algebraico. La especificación debe ser completa, es decir, todas las operaciones deben estar definidas en la semántica. Incluye las operaciones: Vacío, EsVacío, Inserta, Suprime, Miembro, Unión, Intersección, Diferencia y Cardinalidad (número de elementos del conjunto).

Solución.

Los dos constructores del tipo son las operaciones Vacío e Inserta. Así pues, debemos relacionar las restantes operaciones con ambos constructores de la forma adecuada, es decir, de manera que podamos obtener el resultado de cualquier expresión en función de los constructores.

NOMBRE

Conjunto[T]

CONJUNTOS

- C Conjunto de conjuntos de elementos de tipo T
- T Conjunto de elementos que pueden ser almacenados
- N Conjunto de los números naturales
- B Conjunto de valores booleanos

SINTAXIS

- Vacio : $\rightarrow C$
- Inserta : $C \times T \rightarrow C$
- EsVacío : $C \rightarrow B$
- Suprime : $C \times T \rightarrow C$
- Miembro : $C \times T \rightarrow B$
- Unión : $C \times C \rightarrow C$
- Intersección : $C \times C \rightarrow C$
- Diferencia : $C \times C \rightarrow C$
- Cardinalidad : $C \rightarrow N$

SEMÁNTICA

- $\forall c, d \in C; \forall t, p \in T$
- 1. EsVacío(Vacio) = verdadero
- 2. EsVacío(Inserta(c, t)) = falso
- 3. Suprime(Vacio, t) = Vacío
- 4. Suprime(Inserta(c, t), p) = SI ($t = p$) \Rightarrow Suprime(c) | Inserta(Suprime(c, p), t)
- 5. Miembro(Vacio, t) = falso
- 6. Miembro(Inserta(c, t), p) = SI ($t = p$) \Rightarrow verdadero | Miembro(c, p)
- 7. Unión(Vacio, c) = c
- 8. Unión(Inserta(c, t), d) = SI Miembro(d, t) \Rightarrow Unión(c, d) | Inserta(Unión(c, d), t)
- 9. Intersección(Vacio, c) = Vacío
- 10. Intersección(Inserta(c, t), d) = SI Miembro(d, t) \Rightarrow Inserta(Intersección(c, d), t)
| Intersección(c, d)
- 11. Diferencia(Vacio, c) = Vacío
- 12. Diferencia(Inserta(c, t), d) = SI Miembro(d, t) \Rightarrow Diferencia(c, d)
| Inserta(Diferencia(c, d), t)
- 13. Cardinalidad(Vacio) = cero
- 14. Cardinalidad(Inserta(c, t)) = SI Miembro(c, t) \Rightarrow Cardinalidad(c)
| Sucesor(Cardinalidad(c))

Ejercicio 2.2 A la especificación del ejercicio 2.1 añade las operaciones $\text{máximo}(c)$ y $\text{mínimo}(c)$ (para calcular el mayor y el menor elemento del conjunto c) y $\text{sucesor}(c, n)$ y $\text{predecesor}(c, n)$ (para calcular el elemento de c más próximo a n, por arriba o por abajo, respectivamente). Se supondrá que existen las operaciones adecuadas de comparación entre valores de tipo T.

Solución.

Los axiomas deben relacionar cada una de las operaciones anteriores con los constructores del tipo, Vacío e Inserta. Vamos a suponer que el TAD T tiene definidas las operaciones $\text{min}, \text{max} : T \times T \rightarrow T$, y $\text{mayorQue} : T \times T \rightarrow B$. Además, suponemos que

el máximo y el mínimo valor representable del tipo T son `masInfinitoTipoT` y `menosInfinitoTipoT`, respectivamente.

Las partes que se ven modificadas en la sintaxis y en la semántica son las siguientes.

SINTAXIS

`máximo: C → T`

`mínimo: C → T`

`sucesor: C × T → T`

`predec: C × T → T`

SEMÁNTICA

$\forall c \in C; \forall t, p \in T$

15. `máximo(Vacío) = menosInfinitoTipoT`

16. `máximo(Inserta(c, t)) = max(t, máximo(c))`

17. `mínimo(Vacío) = masInfinitoTipoT`

18. `mínimo(Inserta(c, t)) = min(t, mínimo(c))`

19. `sucesor(Vacío, t) = masInfinitoTipoT`

20. `sucesor(Inserta(c, t), p) = SI mayorQue(t, p) ⇒ min(t, sucesor(c, p)) | sucesor(c, p)`

21. `predec(Vacío, t) = menosInfinitoTipoT`

22. `predec(Inserta(c, t), p) = SI mayorQue(p, t) ⇒ max(t, predec(c, p)) | predec(c, p)`

Otra posibilidad sería decidir que se devuelva un mensaje de error en caso de aplicar las operaciones `máximo` y `mínimo` sobre un conjunto vacío, o en caso de que no exista ningún predecesor o sucesor en el conjunto. De esta forma, habría que añadir un conjunto M de mensajes de error, y cambiar la sintaxis y la semántica de las operaciones.

CONJUNTOS

...
M Conjunto de mensajes de error { "El conjunto está vacío", "No existe sucesor", "No existe predecesor" }

SINTAXIS

`máximo: C → T ∪ M`

`mínimo: C → T ∪ M`

`sucesor: C × T → T ∪ M`

`predec: C × T → T ∪ M`

SEMÁNTICA

$\forall c \in C; \forall t, p \in T$

15'. `máximo(Vacío) = "El conjunto está vacío"`

16'. `máximo(Inserta(c, t)) = SI EsVacío(c) ⇒ t | max(t, máximo(c))`

17'. `mínimo(Vacío) = "El conjunto está vacío"`

18'. `mínimo(Inserta(c, t)) = SI EsVacío(c) ⇒ t | min(t, mínimo(c))`

19'. `sucesor(c, t) = SI EsVacío(c) ⇒ "No existe sucesor"`

| `SI mayorQue(mínimo(c), t) ⇒ mínimo(c) | sucesor(Suprime(c, mínimo(c)), t)`

20'. `predec(c, t) = SI EsVacío(c) ⇒ "No existe predecesor"`

| `SI mayorQue(t, máximo(c)) ⇒ máximo(c) | predec(Suprime(c, máximo(c)), t)`

Ejercicio 2.3 Partiendo de la especificación formal para el TAD Natural vista en el apartado 2.3.3, añade a la especificación las operaciones: predecesor, producto y poten-

cia. Ten en cuenta que pueden ocurrir casos donde el resultado no sea un número natural.
Solución.

Puesto que el resultado de estas operaciones puede ser un número no natural, debemos añadir un valor NoNatural a la especificación. Obviamente, no tiene sentido añadir ese valor como una operación constante del tipo NoNatural : $\rightarrow N$, ya que entonces estaríamos diciendo que ¡no natural es un natural! La solución es añadirlo como un nuevo conjunto en la especificación formal:

CONJUNTOS

...
NoN Conjunto formado por el elemento NoNatural

SINTAXIS

predecesor : $N \rightarrow N \cup \text{NoN}$

producto : $N \times N \rightarrow N$

potencia : $N \times N \rightarrow N \cup \text{NoN}$

Los axiomas de la semántica deben ser los necesarios y suficientes para definir de forma completa las tres operaciones. Para ello, debemos relacionarlos con los dos constructores del tipo: cero y sucesor.

SEMÁNTICA

$\forall n, m \in N$

9. predecesor(cero) = NoNatural

10. predecesor(sucesor(n)) = n

11. producto(cero, n) = cero

12. producto(sucesor(n), m) = suma(m, producto(n, m))

13. potencia(cero, cero) = NoNatural

14. potencia(cero, sucesor(n)) = cero

15. potencia(sucesor(n), cero) = sucesor(cero)

16. potencia(sucesor(n), sucesor(m)) = producto(sucesor(n), potencia(sucesor(n), m))

Para la operación potencia son necesarios cuatro axiomas para especificarla completamente. La regla 16, escrita con la notación habitual dice que $(n+1)^{m+1} = (n+1)(n+1)^m$, y la regla 12 diría que $(n+1)m = m + n m$. De esta forma, la operación potencia se define a través de producto, producto se define a través de suma y, finalmente, como ya vimos, suma se define a través del constructor sucesor.

Ejercicio 2.4 Con la especificación del ejercicio 2.3, comprueba el resultado que se obtiene para las siguientes expresiones:

a) igual(predecesor(sucesor(cero)), sucesor(predecesor(cero)))

b) producto(sucesor(sucesor(cero)), sucesor(sucesor(cero)))

Solución.

a) igual(predecesor(sucesor(cero)), sucesor(predecesor(cero)))

igual(predecesor(sucesor(cero)), sucesor(predecesor(cero))) = [Axioma 10, con $n=cero$]

igual(cero, sucesor(predecesor(cero))) = [Axioma 9]

igual(cero, sucesor(NoNatural))

ERROR en la expresión. La operación sucesor espera un parámetro de tipo Natural.

b) producto(sucesor(sucesor(cero)), sucesor(sucesor(cero)))

Para simplificar las expresiones, cambiaremos sucesor por s y cero por c.

$\text{producto}(\text{s}(\text{s}(c)), \text{s}(\text{s}(c))) = [\text{Axioma } 12, \text{ con } n = \text{s}(c), m = \text{s}(\text{s}(c))]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{producto}(\text{s}(c), \text{s}(\text{s}(c)))) = [\text{Axioma } 12, \text{ con } n = c, m = \text{s}(\text{s}(c))]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{suma}(\text{s}(\text{s}(c)), \text{producto}(c, \text{s}(\text{s}(c))))) = [\text{Axioma } 11, \text{ con } n = \text{s}(\text{s}(c))]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{suma}(\text{s}(\text{s}(c)), c)) = [\text{Axioma } 4, \text{ con } n = \text{s}(c), m = c]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{s}(\text{suma}(\text{s}(c), c))) = [\text{Axioma } 4, \text{ con } n = c, m = c]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{s}(\text{s}(\text{suma}(c, c)))) = [\text{Axioma } 3, \text{ con } m = c]$
 $\text{suma}(\text{s}(\text{s}(c)), \text{s}(\text{s}(c))) = [\text{Axioma } 4, \text{ con } n = \text{s}(c), m = \text{s}(\text{s}(c))]$
 $\text{s}(\text{suma}(\text{s}(c), \text{s}(\text{s}(c)))) = [\text{Axioma } 4, \text{ con } n = c, m = \text{s}(\text{s}(c))]$
 $\text{s}(\text{s}(\text{suma}(c, \text{s}(\text{s}(c))))) = [\text{Axioma } 3, \text{ con } m = \text{s}(\text{s}(c))]$
 $\text{s}(\text{s}(\text{s}(c))))$

Ejercicio 2.5 Evaluar el resultado de las siguientes expresiones, usando la especificación formal por el método axiomático del TAD Pila[T], estudiada en el apartado 2.3.5.

- a) $\text{esVacía}(\text{pop}(\text{crearPila}))$
- b) $\text{top}(\text{pop}(\text{pop}(\text{push}(a, \text{push}(b, \text{push}(c, \text{crearPila})))))))$
- c) $\text{esVacía}(\text{pop}(\text{push}(a, \text{push}(x, \text{crearPila}))))$

Solución.

- a) $\text{esVacía}(\text{pop}(\text{crearPila}))$
 $\text{esVacía}(\text{pop}(\text{crearPila})) = [\text{Axioma } 3]$
 $\text{esVacía}(\text{crearPila}) = [\text{Axioma } 5]$
veradero
b) $\text{top}(\text{pop}(\text{pop}(\text{push}(a, \text{push}(b, \text{push}(c, \text{crearPila})))))))$
 $\text{top}(\text{pop}(\text{pop}(\text{push}(a, \text{push}(b, \text{push}(c, \text{crearPila}))))))) = [\text{Axioma } 4, \text{ con } t = a, s = \text{push}(b, \text{push}(c, \text{crearPila}))]$
 $\text{top}(\text{pop}(\text{push}(b, \text{push}(c, \text{crearPila})))) = [\text{Axioma } 4, \text{ con } t = b, s = \text{push}(c, \text{crearPila})]$
 $\text{top}(\text{push}(c, \text{crearPila})) = [\text{Axioma } 2, \text{ con } t = c, s = \text{crearPila}]$
c
c) $\text{esVacía}(\text{pop}(\text{push}(a, \text{push}(x, \text{crearPila}))))$
 $\text{esVacía}(\text{pop}(\text{push}(a, \text{push}(x, \text{crearPila})))) = [\text{Axioma } 4, \text{ con } t = a, s = \text{push}(x, \text{crearPila})]$
 $\text{esVacía}(\text{push}(x, \text{crearPila})) = [\text{Axioma } 6, \text{ con } t = x, s = \text{crearPila}]$
falso

Ejercicio 2.6 Proponer algún modelo subyacente adecuado para escribir la especificación formal por el método constructivo del TAD Conjunto[T]. Escribir la parte de semántica para las operaciones: Vacío, EsVacío, Inserta, Suprime, Miembro, Unión.

Solución.

Un posible modelo subyacente podría ser el TAD Lista[T], definido por el método axiomático en el apartado 2.4.3. Este tipo tiene las operaciones: crearLista, esVacía, formarLista, concatenar, cabeza, cola, primero y último. La especificación por el método constructivo podría ser la siguiente.

NOMBRE

Conjunto[T]

CONJUNTOS

C Conjunto de conjuntos de elementos de tipo T

T Conjunto de elementos que pueden ser almacenados

N Conjunto de los números naturales

B Conjunto de valores booleanos

SINTAXISVacío : $\rightarrow C$ Inserta : $C \times T \rightarrow C$ EsVacío : $C \rightarrow B$ Suprime : $C \times T \rightarrow C$ Miembro : $C \times T \rightarrow B$ Unión : $C \times C \rightarrow C$ **SEMÁNTICA** $\forall c, d \in C; \forall t, p \in T$ **pre-Vacío()** ::= verdadero**post-Vacío();** ::= $c = \text{crearLista}$ **pre-Inserta(c, t)** ::= verdadero**post-Inserta($c, t; d$)** ::= $d = (\text{SI Miembro}(c, t) \Rightarrow c \mid \text{concatenar}(c, \text{formarLista}(t)))$ **pre-Suprime(c, t)** ::= verdadero**post-Suprime($c, t; d$)** ::= $d = (\text{SI esVacía}(c) \Rightarrow c$ | $\text{SI } (\text{primero}(c)=t) \Rightarrow \text{cola}(c) \mid \text{Inserta}(\text{Suprime}(\text{cola}(c), t), \text{primero}(t))$ **pre-Miembro(c, t)** ::= verdadero**post-Miembro($c, t; b$)** ::= $b = (\text{SI esVacía}(c) \Rightarrow \text{falso}$ | $(\text{primero}(c)=t) \text{ OR } \text{Miembro}(\text{cola}(c), t)$ **pre-Unión(c, d)** ::= verdadero**post-Union ($c, d; e$)** ::= $e = (\text{SI esVacía}(d) \Rightarrow c$ | $\text{SI Miembro}(c, \text{primero}(d)) \Rightarrow \text{Union}(c, \text{cola}(d))$ | $\text{concatenar}(\text{formarLista}(\text{primero}(d)), \text{Union}(c, \text{cola}(d)))$

Aun siendo correcta, el inconveniente de la especificación anterior se encuentra en que es muy próxima a lo que sería una implementación del tipo Conjunto, sugiriendo una estructura de representación mediante listas. No obstante, cualquier implementación válida de los conjuntos será coherente con la anterior especificación, use listas o no.

Ejercicio 2.7 A la definición formal axiomática del TAD Conjunto[T], del ejercicio 2.1, queremos añadir las operaciones: EsSubcjo, EsSubPropio y Esgual, que dados dos conjuntos comprueban si uno es subconjunto del otro, si uno es subconjunto propio del otro o si ambos son iguales, respectivamente. Escribe la sintaxis y la semántica de las operaciones.

Solución.

Igual que en algunos de los ejemplos anteriores, es posible simplificar la semántica usando operaciones que ya estén definidas. De esta forma, bastaría con definir EsSubcjo en base a los constructores. Las operaciones EsSubPropio y Esgual se pueden definir con un sólo axioma, usando la anterior operación.

...

SINTAXISEsSubcjo : $C \times C \rightarrow B$ EsSubPropio : $C \times C \rightarrow B$ EsVacío : $C \times C \rightarrow B$ **SEMÁNTICA** $\forall c1, c2 \in C; \forall t \in T$

$\text{EsSubcjo}(\text{Vacío}, c1) = \text{verdadero}$
 $\text{EsSubcjo}(\text{Inserta}(t, c1), c2) = \text{Miembro}(t, c2) \text{ AND } \text{EsSubcjo}(c1, c2)$
 $\text{Igual}(c1, c2) = \text{EsSubcjo}(c1, c2) \text{ AND } \text{EsSubcjo}(c2, c1)$
 $\text{EsSubPropio}(c1, c2) = \text{EsSubcjo}(c1, c2) \text{ AND NOT Igual}(c1, c2)$

Ejercicio 2.8 Comprueba qué resultado se obtiene para las siguientes expresiones, utilizando la especificación por el método constructivo del TAD Cola[T], estudiada en el apartado 2.4.3.

- a) $\text{frente}(\text{inserta}(4, \text{inserta}(5, \text{crearCola})))$
- b) $\text{esVacíaCola}(\text{resto}(\text{inserta}(a, \text{crearCola})))$
- c) $\text{inserta}(\text{frente}(\text{crearCola}), \text{crearCola}))$

Solución.

- a) $\text{frente}(\text{inserta}(4, \text{inserta}(5, \text{crearCola})))$

Operación	Precondición	Postcondición
1. crearCola	verdadero	$o1 = \text{crearLista}$
2. $\text{inserta}(5, o1)$	verdadero	$o2 = \text{concatenar}(o1, \text{formarLista}(5))$
3. $\text{inserta}(4, o2)$	verdadero	$o3 = \text{concatenar}(o2, \text{formarLista}(4))$
4. $\text{frente}(o3)$	$\text{not esVacía}(o3) = [\text{Ax. 18}]$ $\text{not } (\text{esVacía}(o2) \text{ AND esVacía}(\text{formarLista}(4))) = [\text{Ax. 17}]$ $\text{not } (\text{esVacía}(o2) \text{ AND falso}) =$ $\text{not falso} = \text{verdadero}$	$o4 = \text{primero}(o3) = \text{primero}(\text{concatenar}(o2, \text{formarLista}(4))) = [\text{Ax. 3}] \text{ primero}(o2) = \text{primero}(\text{concatenar}(\text{crearLista}, \text{formarLista}(5))) = [\text{Ax. 3}] \text{ primero}(\text{formarLista}(5)) = [\text{Ax. 2}] 5$

- b) $\text{esVacíaCola}(\text{resto}(\text{inserta}(a, \text{crearCola})))$

Operación	Precondición	Postcondición
1. crearCola	verdadero	$o1 = \text{crearLista}$
2. $\text{inserta}(a, o1)$	verdadero	$o2 = \text{concatenar}(o1, \text{formarLista}(a))$
3. $\text{resto}(o2)$	$\text{not esVacía}(o2) = [\text{Ax. 18}]$ $\text{not } (\text{esVacía}(o1) \text{ AND esVacía}(\text{formarLista}(a))) =$ $[\text{Ax. 16,17}] \text{ not } (\text{verdadero AND falso}) = \text{not falso} = \text{verdadero}$	$o3 = \text{cola}(o2) = \text{cola}(\text{concatenar}(o1, \text{formarLista}(a))) = [\text{Ax. 12}] \text{cola}(\text{formarLista}(a)) = [\text{Ax. 11}] \text{crearLista}$
4. $\text{esVacíaCola}(o3)$	verdadero	$o4 = \text{esVacía}(o3) = \text{esVacía}(\text{crearLista}) = [\text{Ax. 12}] \text{verdadero}$

- c) $\text{inserta}(\text{frente}(\text{crearCola}), \text{crearCola}))$

Operación	Precondición	Postcondición
1. crearCola	verdadero	$o1 = \text{crearLista}$
2. $\text{frente}(o1)$	$\text{not esVacía}(o1) = \text{not esVacía}(\text{crearLista}) = [\text{Ax. 16}] \text{not verdadero} = \text{falso}$	ERROR en la expresión $\text{frente}(\text{crearCola})$. No se cumple la precondición. Ejecución interrumpida.

Ejercicios propuestos

Ejercicio 2.9 Suponiendo la especificación formal algebraica vista en el apartado 2.3.3 para el TAD Natural (con las operaciones: cero, sucesor, esCero, igual y suma), escribe la sintaxis y la semántica correspondientes a las operaciones de comparación entre naturales: esMenor, esMenorIgual, esMayor, esMayoriGual; y las operaciones Máximo y Mínimo.

Ejercicio 2.10 Se define el TAD Bolsa[T] como un conjunto de elementos de tipo T donde un elemento puede ser insertado más de una vez. Escribe una especificación algebraica y otra constructiva para el tipo Bolsa[T], con las operaciones: bolsaVacía, inserta, suprime, esVacía, numVeces (devuelve el número de veces que un elemento ha sido insertado en la bolsa) e igual (comprueba si dos bolsas son iguales).

Ejercicio 2.11 Considera la siguiente especificación formal algebraica del TAD CMT (contador módulo tres). Las operaciones definidas son: Nuevo : \rightarrow CMT; Inc : CMT \rightarrow CMT; Dec : CMT \rightarrow CMT. Los axiomas de la parte Semántica son los siguientes:

- $\forall c \in \text{CMT}$
- 1. $\text{Inc}(\text{Dec}(c)) = c$
- 2. $\text{Dec}(\text{Inc}(c)) = c$
- 3. $\text{Inc}(\text{Inc}(\text{Inc}(c))) = c$
- 4. $\text{Dec}(\text{Dec}(\text{Dec}(c))) = c$

Demuestra, usando los axiomas de la especificación formal (e indicando el número de los que se usen), que se cumplen las dos siguientes propiedades:

- $\text{Dec}(\text{Nuevo}) = \text{Inc}(\text{Inc}(\text{Nuevo}))$
- $\text{Inc}(c) = \text{Dec}(\text{Dec}(c)), \forall c \in \text{CMT}$

Ejercicio 2.12 Usando la especificación de los ejercicios 2.1 y 2.2, aplicada sobre el tipo Conjunto[Natural], escribe la expresión correspondiente a crear un conjunto con los elementos $\{5, 2\}$. Sobre ese conjunto, aplica las operaciones: máximo, mínimo, sucesor(C, 22), predec(C, 6) y deduce el resultado usando los axiomas.

Ejercicio 2.13 Construye una especificación formal algebraica para el TAD genérico Lista[T] en la cual los constructores sean las operaciones: listaVacía (crea una lista vacía) e inserta (añade un elemento al principio de la lista). Añade las operaciones: primero, último, cola, concatenar y longitud.

Ejercicio 2.14 Queremos añadir a la especificación algebraica del tipo Lista[T] las operaciones: invertirLista(l) y elementoEn(l, n) (devuelve el elemento de la lista l insertado en la posición n-ésima). Escribe la sintaxis y la semántica de las operaciones anteriores, tomando como base las especificaciones algebraicas de listas del apartado 2.4.3 y la del ejercicio 2.13.

Ejercicio 2.15 Utilizando las especificaciones de los ejercicios 2.13 y 2.14, obtén el resultado de las siguientes expresiones. Se supone que l es una lista cualquiera.

- elementoEn(cola(inserta(4, inserta(2, l))), 3)
- primero(cola(invertirLista(inserta(7, listaVacía))))
- invertirLista(invertirLista(inserta(a, inserta(b, listaVacía)))))

Ejercicio 2.16 Construye la especificación formal algebraica del TAD Persona, que está formada por un registro con tres campos: nombre (de tipo cadena), dirección (de tipo cadena) y teléfono (de tipo entero). Incluye operaciones de consulta y modificación de cada uno de ellos. Muestra varias expresiones de ejemplo, donde se establezcan varios valores y luego se consulten algunos de ellos. ¿Cuáles son los constructores del tipo?

Ejercicio 2.17 En una especificación formal constructiva del tipo $\text{Array}[T]$, tenemos una operación $\text{Ordena} : A \rightarrow A$, que dado un array devuelve sus elementos ordenados de menor a mayor. Escribe la semántica para esta operación, indicando las precondiciones y postcondiciones adecuadas. Se supone que existen otras operaciones del tipo: $\text{Tamaño} : \text{Array} \rightarrow \text{Entero}$; $\text{ObtenValor} : \text{Array} \times \text{Entero} \rightarrow T$; $\text{PonValor} : \text{Array} \times \text{Entero} \times T \rightarrow \text{Array}$; $\text{MenorQue} : T \times T \rightarrow B$.

Ejercicio 2.18 Construye una especificación formal axiomática para el TAD $\text{Array}[T]$. Los índices del array son enteros y el rango de índices del array no está limitado. Las operaciones sobre el TAD son: Nuevo, PonValor, ObtenValor (para escribir o leer un valor en una posición del array, respectivamente) y Máximo (para obtener el índice máximo de array). Adopta las decisiones que creas adecuadas para los posibles casos de error. ¿Cuáles son los constructores del tipo y las operaciones de modificación y consulta?

Ejercicio 2.19 A la definición formal constructiva del TAD $\text{Conjunto}[T]$, del ejercicio 2.6, queremos añadir las operaciones: EsSubconjunto, EsSubPropio y EsIgual, que dados dos conjuntos comprueban si uno es subconjunto del otro, si uno es subconjunto propio del otro o si ambos son iguales, respectivamente. Escribe la sintaxis y la semántica de las operaciones, utilizando las pre- y postcondiciones adecuadas.

Ejercicio 2.20 Tenemos definido el TAD Pantano, por el método axiomático, con las operaciones:

$\text{Nuevo} : N \rightarrow \text{Pantano}$. Devuelve un pantano con capacidad máxima N y cantidad actual de agua 0.

$\text{Llenar} : \text{Pantano} \rightarrow \text{Pantano}$. Pone la cantidad actual de agua del pantano al valor de capacidad máxima.

$\text{Cantidad} : \text{Pantano} \rightarrow N$. Devuelve la cantidad actual de agua del pantano

$\text{Transvasar} : \text{Pantano} \times N \rightarrow \text{Pantano} \cup \text{Error}$. Decrementa la cantidad actual en N , siempre que sea posible.

$\text{Ocupación} : \text{Pantano} \rightarrow R$. Devuelve el porcentaje actual de ocupación del pantano.

Escribe una especificación formal para este tipo abstracto. Se puede utilizar el método algebraico o el constructivo. Si lo crees conveniente puedes añadir otras operaciones.

Ejercicio 2.21 Evalúa el resultado de las siguientes expresiones sobre pilas, usando la especificaciones formales constructivas del apartado 2.4.3 y la especificación algebraica del apartado 2.3.5.

- $\text{esVacía}(\text{pop}(\text{push}(t1, \text{pop}(\text{push}(t2, \text{crearPila}))))))$
- $\text{tope}(\text{pop}(\text{pop}(\text{push}(t3, \text{crearPila}))))$

Ejercicio 2.22 Construye una especificación formal axiomática para el TAD ArbolBinario[T]. El tipo debe contener las operaciones: CrearArbol, Construir (dados dos árboles I y D, y un elemento x, devuelve un nuevo árbol donde x es la raíz y los subárboles izquierdo y derecho son I y D, respectivamente), Raíz (devuelve la raíz), Hijolq (devuelve el hijo izquierdo), HijoDer y EsVacío.

Ejercicio 2.23 Partimos de las especificaciones formales algebraicas del TAD Lista[T], del apartado 2.4.3, y ArbolBinario[T], del ejercicio 2.22. Añade a este último las operaciones OrdenPrevio, OrdenSimétrico y OrdenPosterior que, dado un árbol como argumento, devuelven una lista que contiene todos los elementos del árbol ordenados según su recorrido en orden previo, simétrico y posterior, respectivamente.

Ejercicio 2.24 Definimos el TAD ColaCíclica[T] como una cola FIFO usual (primero en entrar, primero en salir), pero en la cual al sacar un elemento de la cabeza, se vuelve a meter en la cola en la última posición. Las operaciones del tipo son: Crear, Meter (mete un elemento en la última posición de la cola), Cabeza (devuelve el elemento que está en la cabeza de la cola) y Avanzar (el elemento de la cabeza pasa al final de la cola). Escribe una especificación formal axiomática del tipo ColaCíclica[T]. Si lo crees conveniente puedes incluir otras operaciones (en cuyo caso deberás especificarlas también).

Ejercicio 2.25 Queremos definir el TAD genérico GrafoDirigido[T] por el método axiomático, con las operaciones: GrafoVacío (crea un nuevo grafo sin vértices), InsArista (añade una arista al grafo, con los dos vértices pasados como parámetros), ExisteArista (comprueba si existe una arista en el grafo) y GradoEntrada (devuelve el grado de entrada de un vértice dado). Escribe la especificación formal axiomática del tipo. Tener en cuenta lo siguiente:

- Los vértices son del tipo genérico T.
- No existe una operación para insertar vértices, ya que se supone que son añadidos implícitamente en la operación de insertar aristas (si no existen ya).
- Si se inserta una arista que ya existe, no ocurre nada ya que sólo puede existir una arista entre un par de vértices.

Cuestiones de autoevaluación

Ejercicio 2.26 Uno de los resultados fundamentales de utilizar abstracciones es la distinción entre especificación e implementación. ¿Qué diferencias existen entre una especificación y una implementación? ¿Qué relaciones de dependencia existen entre ambas, es decir cuál va antes y por qué? Indica las ventajas de tener una buena especificación de un tipo o procedimiento, según se use una notación formal o informal.

Ejercicio 2.27 Para una aplicación que utiliza colas de enteros, representadas mediante listas enlazadas, definimos la especificación formal del TAD ColaEnteros por el método axiomático. ¿Qué partes de la especificación formal deberíamos cambiar si en otra aplicación queremos representar las colas mediante arrays? Responde la misma cuestión suponiendo que se usa el método constructivo de especificación.

Ejercicio 2.28 ¿Qué diferencias existen entre un TAD mutable y uno no mutable? ¿Es posible desarrollar una especificación formal algebraica de un TAD mutable? ¿Por qué? ¿Y si la especificación formal es constructiva u operacional?

Ejercicio 2.29 ¿Qué ocurre en una especificación formal constructiva si en una expresión no se cumple la precondición? ¿Puede ocurrir algo parecido en una especificación por el método algebraico? ¿Qué ocurre si no se cumple la postcondición? ¿De quién es la responsabilidad en cada caso?

Ejercicio 2.30 Sea un TAD cualquiera A, definido mediante una especificación formal algebraica. ¿Qué propiedad cumplen las operaciones que son los constructores del tipo? ¿Qué otros tipos de operaciones pueden existir? ¿Por qué es importante esta distinción?

Ejercicio 2.31 Supón que desarrollamos la especificación algebraica del TAD genérico Array[T], incluyendo una operación para ordenar los elementos de un array. ¿Cómo se podría expresar con los axiomas el hecho de que el resultado debe ser un array ordenado?

Ejercicio 2.32 La abstracción de iteradores aparece como un tipo de abstracción diferente a las abstracciones de datos y las funcionales. ¿Podrían considerarse como un caso especial de abstracción funcional? ¿Cómo se puede implementar abstracciones de iteradores usando lenguajes como Pascal o C?

Ejercicio 2.33 En el formato de notación para la especificación informal de abstracciones de datos, del apartado 2.2.2, la cabecera tiene la forma: **TAD nombre es lista_operaciones**. El uso de **es**, en lugar de **tiene** u **operaciones**, intenta resaltar el hecho de que el tipo *sólo* debe ser usado mediante las operaciones definidas con el mismo. Por ejemplo, si el tipo es un registro no se puede acceder a sus campos directamente. ¿Qué implicaciones tiene esto en la implementación de un TAD utilizando clases? ¿Qué partes del tipo deberían ir en la parte **public** y cuáles en la parte **private**?

Ejercicio 2.34 ¿Cómo podemos asegurar que una especificación formal algebraica es válida, es decir no contiene contradicciones y es completa? ¿Y en el caso de la especificación constructiva?

Referencias bibliográficas

El concepto de abstracción y los mecanismos y tipos de abstracciones se encuentran en la mayoría de los libros de algoritmos y estructuras de datos, como por ejemplo el capítulo 1 de [Aho88]. Una descripción más extensa puede consultarse en el capítulo 2 de [Collado87], donde aparece también el formato de especificación informal de abstracciones funcionales y de datos, y el método de especificación axiomático.

Para profundizar en las especificaciones algebraicas se puede utilizar [Franch94], donde se utiliza una aproximación bastante formal al estudio de los TAD. En cuanto a las especificaciones constructivas, la idea de las pre- y postcondiciones como contratos surge de [Meyer99], y es descrita en los capítulos 11 y 12. En este libro se encuentran las bases y los principios de la programación orientada a objetos, aunque en general resulta algo excesivo para el nivel aquí tratado.

Capítulo 3

Conjuntos y diccionarios

Almacenar y recuperar información es una de las necesidades básicas de cualquier programa o sistema informático. Existe una gran variedad de estructuras de datos que responden a estos requerimientos, pero sólo un reducido número de tipos de datos abstractos. Entre estos, los dos tipos fundamentales son los conjuntos y los diccionarios. Los conjuntos permiten almacenar un número indeterminado de elementos distintos, de cierto tipo, siendo indiferente el orden de inserción de los mismos. Los diccionarios almacenan conjuntos de asociaciones de claves con valores; a cada clave se le asocia determinada información, que después se podrá recuperar, borrar o modificar. En este capítulo analizaremos algunas estructuras lineales –listas y tablas– para representar conjuntos y diccionarios.

Objetivos del capítulo:

- Familiarizarse con la notación y terminología de los tipos de datos abstractos conjunto y diccionario.
- Conocer la importancia y ubicuidad de los tipos conjunto y diccionario en el desarrollo de programas, independientemente de la estructura que se use para implementarlos.
- Ser capaz de diseñar, implementar y analizar la eficiencia de las principales estructuras de representación no arbóreas para los tipos conjunto y diccionario, adaptando el diseño a las necesidades específicas de cada aplicación.
- Conocer la estructura de datos de tablas de dispersión, sus distintas variantes y los factores que influyen en su eficiencia y uso de memoria.
- Concienciarse de la importancia del factor eficiencia, en cuanto a tiempo y uso de memoria, en el diseño e implementación de estructuras de datos.
- Comprender la posibilidad de tener diferentes estructuras de acceso para un mismo conjunto de datos almacenados, a través del estudio de estructuras duales o múltiples.

Contenido del capítulo:

3.1.	Los tipos abstractos conjunto y diccionario	81
3.1.1.	El concepto matemático de conjunto	82
3.1.2.	El tipo abstracto diccionario	83
3.1.3.	Representación mediante arrays de booleanos	85
3.1.4.	Representación mediante listas de elementos	87
3.2.	Tablas de dispersión	90
3.2.1.	La dispersión y los sinónimos	91
3.2.2.	Dispersión abierta	92
3.2.3.	Dispersión cerrada	94
3.2.4.	Funciones de dispersión	100
3.2.5.	Estrategias de redispersión	102
3.3.	Combinando estructuras de datos	104
3.3.1.	Estructuras de datos múltiples	105
3.3.2.	La relación muchos a muchos	109
3.3.3.	Estructuras de listas múltiples	113
	Ejercicios resueltos	116
	Ejercicios propuestos	122
	Cuestiones de autoevaluación	125
	Referencias bibliográficas	126

3.1. Los tipos abstractos conjunto y diccionario

El volumen de información que deben manejar los sistemas informáticos se ve claramente reflejado en las dimensiones de la red de redes, Internet. Varios cientos de millones de usuarios consultan diariamente unos cuantos miles de millones de páginas, que ocupan en torno a algunas decenas de billones de bytes. Lógicamente, todos estos datos están distribuidos entre millones de servidores, que se reparten la tarea. Pero existe una aplicación que virtualmente requiere que todas las páginas sean almacenadas y procesadas en un sólo ordenador: los buscadores de Internet.

Teniendo en cuenta el volumen de información manejado, resulta realmente sorprendente que un buscador sea capaz de buscar una palabra o secuencia de palabras cualesquiera, entre más de tres mil millones de páginas, produciendo un resultado fiable en unas pocas décimas de segundo. Pero, aparte de usar estructuras de datos optimizadas al máximo, la primera cuestión a resolver no son las estructuras sino las necesidades de información, es decir, qué datos se almacenan y qué tipo de operaciones necesitamos hacer sobre los mismos; en definitiva, qué tipos abstractos de datos son necesarios.

Hagamos un poco de *ingeniería inversa* sobre los tipos abstractos que previsiblemente son usados en un buscador de Internet. Podemos encontrar diferentes necesidades de información y, para cada una de ellas, tenemos un TAD que la provee.

- En primer lugar, necesitamos almacenar las direcciones de todas las páginas existentes. Cada dirección podría ser una cadena de caracteres, de un tamaño máximo dado. **Direcciones:** Conjunto de cadenas de caracteres.
- Representar las páginas mediante la cadena de la dirección resulta un poco engorroso; sería más adecuado asignar un código entero a cada página. Un código de 4 bytes sólo permite unos 4.000 millones de páginas, así que usaremos enteros de 5 bytes. **Claves:** Conjunto de pares (Entero de 5 bytes, Cadena de caracteres).
- Para cada página debemos almacenar una serie de información adicional (título de la página, fecha, contenido, etc.). **Páginas:** Conjunto de pares (Entero de 5 bytes, Descripción de la página).
- Sin duda, la capacidad más notable de un buscador es la de obtener los resultados de forma casi instantánea. ¿Cómo es posible conseguirlo? Pues teniendo precalculadas las búsquedas. En lugar de tener para cada página las palabras que aparecen, tenemos para cada palabra el conjunto de las páginas donde aparece. **Palabras:** Conjunto de pares (Cadena de caracteres, Conjunto de enteros de 5 bytes).

En definitiva, la aplicación requiere almacenar mucha información y de naturaleza muy variada. Y, sin embargo, resulta curioso que en todos los casos sólo aparece un tipo abstracto: el tipo **conjunto**. Realmente, podemos distinguir otro tipo abstracto, el tipo **conjunto de pares**, donde el primer elemento del par actúa como clave y el segundo como valor asociado a la clave. Esto es básicamente lo que se conoce como el TAD **diccionario**.

3.1.1. El concepto matemático de conjunto

El ejemplo anterior es una pequeña muestra de la profusión y ubicuidad de los conjuntos en el ámbito de la programación. Los conjuntos son uno de los tipos más elementales en la mayoría de las aplicaciones, actuando en ocasiones como una base sobre la que se construyen otros tipos más complejos. Los conjuntos son también básicos en la mayoría de las ramas de las matemáticas. Vamos a repasar el concepto matemático y las propiedades de los conjuntos, contrastándolos con el concepto informático.

Definición de conjunto y propiedades

Definición 3.1 Un conjunto es una colección no ordenada de **elementos** (o miembros) distintos.

La anterior definición no impone ninguna restricción sobre los elementos contenidos en los conjuntos. En programación, normalmente, interesa restringir el tipo de los elementos almacenados; de esta forma, podemos tener conjuntos de enteros, de cadenas, de árboles, de conjuntos, etc. En general, hablamos del tipo genérico **Conjunto[T]**, siendo T el tipo de los miembros.

Por otro lado, un conjunto es un tipo colección o contenedor. La particularidad de este tipo –respecto a otros contenedores como listas, tablas o árboles– es que los elementos *no están ordenados*. Por lo tanto, no tienen sentido operaciones del tipo: obtener el primer elemento, obtener el n-ésimo, el último, la raíz, etc., que aparecen en los tipos lista, tabla o árbol. Puede existir una relación de orden entre los elementos del conjunto, lo cual no implica que los elementos deban ser almacenados o procesados en ese orden. Más bien, la propiedad de un conjunto de ser no ordenado se refiere a que el orden de inserción de los elementos es indiferente.

Definición 3.2 Una relación de orden en un conjunto *C* es una relación “*<*” que cumple la *propiedad transitiva*:

$$\forall a, b, c \in C; (a < b) \wedge (b < c) \Rightarrow (a < c)$$

Definición 3.3 Se dice que una relación de orden es un **orden total** si se cumple que, para cualquier par de elementos, o son iguales o uno es menor que el otro:

$$\forall a, b \in C; (a < b) \vee (b < a) \vee (a = b)$$

Según la definición, todos los elementos de un conjunto deben ser distintos. En ciertas aplicaciones puede ser interesante permitir la repetición de elementos. En esos casos hablamos más propiamente del tipo abstracto **bolsa**: una bolsa es una colección no ordenada de elementos con repetición. Por lo tanto, la sintaxis de una operación de consulta tiene la forma: **numeroVeces: C × T → N**.

Notación y terminología de conjuntos

En este apartado se hará un repaso de las principales operaciones usadas en teoría de conjuntos. La definición de un conjunto se puede hacer por extensión, indicando todos sus elementos, o mediante proposiciones. Por ejemplo:

Declaración por extensión. $C = \{a, b, c, \dots, z\}$; $D = \{1, 4, 7\} = \{4, 7, 1\}$

Declaración mediante proposiciones. $C = \{x \in T \mid P(x) = \text{verdadero}\}$

En la tabla 3.1 se repasan las principales operaciones sobre conjuntos. La especificación formal de algunas de estas operaciones aparece en el ejercicio 2.1.

Nombre	Signatura	Notación matemática
Conjunto vacío	Vacio : $\rightarrow C$	\emptyset
Conjunto universal	T	\mathcal{U}
Inserción	Inserta : $C \times T \rightarrow C$	$c1 \cup \{c\} = \{a, b, c, d\}$
Eliminación	Suprime : $C \times T \rightarrow C$	$c2 - \{a\} = c2$
Pertenencia	Miembro : $C \times T \rightarrow B$	$a \in c1$
No pertenencia	NoMiembro : $C \times T \rightarrow B$	$b \notin c3$
Unión	Union : $C \times C \rightarrow C$	$c1 \cup c2 = \{a, b, c, d, e\}$
Intersección	Interseccion : $C \times C \rightarrow C$	$c1 \cap c2 = \{b, d\}$
Diferencia	Diferencia : $C \times C \rightarrow C$	$c1 - c2 = \{a\}$
Inclusión	Subconj : $C \times C \rightarrow B$	$c1 \subseteq c2 = \text{falso}$
Inclusión propia	SubconjPropio : $C \times C \rightarrow B$	$c3 \subset c2 = \text{verdadero}$
Igualdad	Igual : $C \times C \rightarrow B$	$(c1 = c2) = \text{falso}$
Cardinalidad	Cardinalidad : $C \rightarrow N$	$ c1 = 3; c2 = 4$

Tabla 3.1: Operaciones del TAD genérico Conjunto[T] y su equivalente usando la notación matemática de conjuntos. Se define C como el conjunto de todos los Conjunto[T], y suponemos las variables: $a, b, c, d, e \in T$ y $c1 = \{a, b, d\}$, $c2 = \{b, c, d, e\}$, $c3 = \{c, d\}$.

Hay que notar que, en sentido matemático, la eliminación de un elemento que no pertenece a un conjunto –o la inserción de un elemento que ya pertenece a un conjunto– no producen ningún error. En ambos casos el conjunto resultante es el mismo que el de partida. No obstante, es posible que en algunas aplicaciones sea de utilidad que la implementación sobre conjuntos detecte los casos anteriores como situaciones de error.

3.1.2. El tipo abstracto diccionario

Los conjuntos permiten almacenar datos, que después son consultados mediante la operación Miembro. Por lo tanto, un conjunto es como un saco donde se meten y se sacan cosas. Pero, en muchas ocasiones, suele resultar más interesante pegarle una etiqueta a las cosas almacenadas que actúe como código, clave o identificador de ese objeto. De esta forma, la manipulación de un elemento siempre se hace a través de su clave. Retomando la aplicación del buscador de Internet, observamos un ejemplo típico: la mayoría de los datos almacenados son pares de elementos donde el primer elemento del par actúa como clave y el segundo como valor asociado a la misma.

- **Claves:** Conjunto de pares (Entero de 5 bytes, Cadena de caracteres). En este caso, la operación de consulta recibe un entero de 5 bytes (la clave de la página) y devuelve como resultado una cadena (la dirección de la página correspondiente).

- **Páginas:** Conjunto de pares (Entero de 5 bytes, Descripción de la página). Dada una clave de una página, devuelve un registro que contiene la información almacenada sobre esa página.
- **Palabras:** Conjunto de pares (Cadena de caracteres, Conjunto de enteros de 5 bytes). Dada una palabra (sobre la cual se hace una consulta), devuelve un conjunto con todas las páginas donde aparece esa palabra. En este caso las claves son las palabras y los valores conjuntos de claves.

Estos pares son lo que se conoce como el TAD **asociación**: una asociación es un par (clave: tipo_clave; valor: tipo_valor). El primer elemento del par desempeña la función de clave mediante la cual se identifica un determinado objeto, mientras que el segundo es el valor asociado a esa clave. Ambos tipos están parametrizados. La especificación informal dè este TAD se vio en el capítulo anterior, en el ejemplo 2.5.

Un **diccionario** es, básicamente, un conjunto de asociaciones. Las estructuras de datos que veremos en este capítulo y en el siguiente se pueden usar para almacenar tanto conjuntos como diccionarios. No obstante, existen algunas diferencias en la sintaxis y la semántica de sus operaciones, por lo que podemos decir que son dos TAD claramente diferenciados.

- En los conjuntos, la operación Miembro devuelve un valor booleano que indica la pertenencia o no de un elemento. En los diccionarios, la operación de consulta recibe una clave y devuelve el valor asociado a la misma, en caso de que exista.
- De forma similar, en los diccionarios la operación de eliminación de un elemento sólo necesita como parámetro la clave del objeto a eliminar.
- A diferencia de los conjuntos, normalmente los diccionarios se usan como un almacén único, por lo que no suelen ser necesarias operaciones del tipo: unión, intersección, diferencia, etc.
- En la operación de inserción, si ya existe un elemento con la misma clave, entonces se modifica el valor asociado. Esta cuestión no se plantea en los conjuntos.

Los diccionarios se pueden ver como una generalización del TAD **array** o **tabla**. En una tabla se accede a los elementos –para hacer inserciones, modificaciones o consultas– a través de un índice entero. Un diccionario, visto de forma abstracta, es como una tabla donde los índices son de un tipo cualquiera, tipo_clave, y las celdas son de tipo tipo_valor. Por este motivo, los diccionarios son también llamados **tablas asociativas**.

A continuación se desarrolla una posible especificación informal para el TAD genérico **Diccionario[tclave, tvalor]**, incluyendo un conjunto mínimo de operaciones.

TAD Diccionario[tclave, tvalor: tipo] es Crear, Inserta, Consulta, Suprime

Requiere

El tipo **tclave** debe tener definida una operación de comparación *Igual* (*ent tclave, tclave; sal boolean*). Además el tipo **tvalor** debe tener definida una operación constante *NULO_TVALOR*.

Descripción

Los valores de tipo *Diccionario[tclave, tvalor]* son diccionarios modificables, que almacenan pares (*clave: tclave, valor: tvalor*), que asocian una clave con un valor correspondiente. La operación Crear se usa para tener un diccionario vacío. La consulta, inserción y modificación se hacen por valores de clave.

Operaciones

Operación Crear (sal Diccionario[tclave,tvalor])

Calcula: Devuelve un diccionario nuevo, que no contiene asociaciones.

Operación Inserta (ent D: Diccionario[tclave,tvalor]; c: tclave; v: tvalor)

Modifica: D

Calcula: Inserta el par (*c, v*) en el diccionario *D*. Si ya existía en *D* un elemento con la misma clave (*c, v'*), entonces elimina la asociación (*c, v'*) antes de insertar (*c, v*).

Operación Consulta (ent D: Diccionario[tclave,tvalor]; c: tclave; sal tvalor)

Calcula: Busca si en el diccionario *D* existe alguna asociación con la clave *c*. Si existe esa asociación (*c, v*), entonces devuelve *v*. En caso contrario devuelve *NULO_TVALOR*.

Operación Suprime (ent D: Diccionario[tclave,tvalor]; c: tclave)

Modifica: D

Requiere: Busca en el diccionario *D* un par (*c, v*) con la misma clave pasada como parámetro. Si lo encuentra, lo elimina del diccionario. En otro caso no produce ninguna modificación.

Fin Diccionario.

3.1.3. Representación mediante arrays de booleanos

Antes de estudiar representaciones avanzadas de conjuntos, vamos a plantear los dos tipos más básicos de estructuras que podemos encontrar: **arrays de booleanos** y **listas de elementos**. La primera corresponde a una idea bastante intuitiva: usamos una tabla de booleanos, donde para cada elemento *t* del conjunto universal se indica si pertenece o no a un conjunto dado. En la representación mediante listas, simplemente utilizamos una lista para almacenar los elementos que sí pertenecen.

Aunque estas estructuras no son viables cuando se necesite almacenar muchos datos, en problemas pequeños y con tamaño restringido suelen ser las más adecuadas. En la figura 3.1 se muestran un par de ejemplos sencillos de estas estructuras.

La representación de conjuntos mediante arrays de booleanos implica que a cada elemento del conjunto universal se le asocia un índice único en el array. Esta indexación puede ser inmediata, por ejemplo si tenemos conjuntos de números de 1 a 100, o conjuntos de caracteres alfabéticos. Las definiciones del tipo serían:

tipo

ConjuntoNum = array [1, ..., 100] de booleano

ConjuntoCar = array ['a', ..., 'z'] de booleano

Para otro tipo de elementos almacenados, debe existir alguna forma de asociar un índice único a cada elemento. Obviamente, el tamaño del array debe ser igual al rango de posibles elementos –o lo que es lo mismo, el tamaño del conjunto universal– que llamaremos *Rango(tclave)*. En el caso de los diccionarios, el array debería ser de celdas de tipo

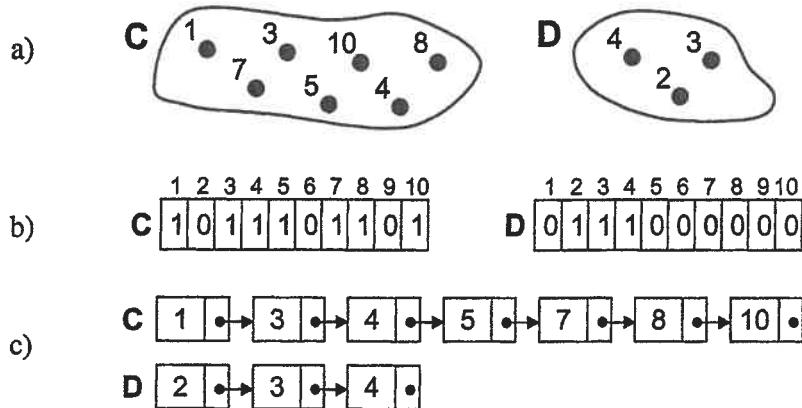


Figura 3.1: Representaciones y estructuras de conjuntos. a) Representación mediante diagramas de Venn o de patata. b) Estructura de array de booleanos. c) Estructura de listas enlazadas.

tvalor, en lugar de ser de booleanos.

tipo

Conjunto[T] = array [Rango(T)] de booleano

Diccionario[tclave, tvalor] = array [Rango(tclave)] de tvalor

Implementación de las operaciones

La implementación de las operaciones –tanto del tipo conjunto como diccionario– consiste en manejar de forma adecuada los valores de la tabla. Para las operaciones que trabajan con un sólo elemento (insertar, miembro y eliminar) conocemos exactamente la posición que ocupa dentro del array, así que se pueden realizar en un tiempo constante con un simple acceso al array.

operación Miembro (C: Conjunto[T]; t: T): booleano

devolver $C[t]$

operación Inserta (var C: Conjunto[T]; t: T)

$C[t]:= \text{verdadero}$

operación Suprime (var C: Conjunto[T]; t: T)

$C[t]:= \text{falso}$

operación Inserta (var D: Diccionario[tclave,tvalor]; c: tclave; v: tvalor)

$D[c]:= v$

operación Suprime (var D: Diccionario[tclave,tvalor]; c: tclave)

$D[c]:= \text{NULO_TVALOR}$

operación Consulta (D: Diccionario[tclave,tvalor]; c: tclave): tvalor

devolver $D[c]$

Por otro lado, las operaciones sobre conjuntos: unión, intersección, diferencia, subconjunto, etc., se transforman en las operaciones booleanas correspondientes.

operación Intersección (C, D: Conjunto[T]; var R: Conjunto[T])

para cada t en $\text{Rango}(T)$ hacer

```

 $R[t]:= C[t] \text{ Y } D[t]$ 
operación Unión ( $C, D: \text{Conjunto}[T]; \text{var } R: \text{Conjunto}[T]$ )
  para cada  $t$  en  $\text{Rango}(T)$  hacer
     $R[t]:= C[t] \text{ O } D[t]$ 
operación Diferencia ( $C, D: \text{Conjunto}[T]; \text{var } R: \text{Conjunto}[T]$ )
  para cada  $t$  en  $\text{Rango}(T)$  hacer
     $R[t]:= C[t] \text{ Y NO } D[t]$ 
operación Subconjunto ( $C, D: \text{Conjunto}[T]$ ): booleano
   $\text{esSubcjt}:= \text{verdadero}$ 
  para cada  $t$  en  $\text{Rango}(T)$  hacer
     $\text{esSubcjt}:= \text{esSubcjt} \text{ Y } (C[t] \text{ O NO } D[t])$ 
devolver  $\text{esSubcjt}$ 

```

Evaluación de la eficiencia

La eficiencia de la estructura de arrays de booleanos está estrechamente relacionada con el hecho de que se represente cada posición del conjunto universal mediante un booleano. Como ya hemos visto, esto implica que las operaciones para acceder o modificar un elemento se pueden hacer en un tiempo constante, $O(1)$.

Por contra, para las operaciones que implican todos los elementos la eficiencia depende del tamaño del conjunto universal, independientemente del tamaño de los conjuntos representados. Si N es el tamaño del conjunto universal, el tiempo de las operaciones sería un $O(N)$. Lo mismo ocurre con la memoria requerida.

Supongamos, por ejemplo, que usamos conjuntos de números de 1 a 64; representando cada booleano con 1 bit, necesitamos 8 bytes para representar cada conjunto. Las operaciones de unión, intersección, etc., se pueden realizar con tan solo 2 operaciones lógicas, si el tamaño de palabra de la máquina es 32 bits. Pero ¿qué ocurriría en el ejemplo de los buscadores de Internet, donde representamos conjuntos de enteros de 5 bytes? El tamaño del conjunto universal sería 2^{5*8} , lo cual requeriría unos 128 Gbytes por cada conjunto representado. Es más, si quisiéramos representar las direcciones, por ejemplo mediante cadenas de hasta 80 caracteres, necesitaríamos unos $256^{80}/8 \approx 5,7 * 10^{191}$ bytes, ¡muchas veces más que el número de partículas subatómicas del universo conocido¹! ¡Y eso independientemente de que el conjunto representado contenga 14 ó 14.000 millones de páginas!

3.1.4. Representación mediante listas de elementos

En la representación de conjuntos y diccionarios mediante listas, usamos una estructura de listas para almacenar sólo los elementos que sí pertenecen a un conjunto o diccionario dado. De esta forma, la eficiencia depende del tamaño de los conjuntos representados, y no del conjunto universal. A su vez, la implementación de las listas puede utilizar listas enlazadas (simple o doblemente) con punteros o cursores, o representación de listas mediante tablas.

¹Estimado entre 10^{72} y 10^{87} . Y por encima también de un *googol*, que equivale a 10^{100} .

En cualquiera de los casos, supondremos un tipo `Lista[T]` con las operaciones: `ListaVacia`, crea una lista vacía; `Primero`, se coloca en la primera posición de la lista; `Avanzar`, avanza a la siguiente posición; `Actual`, devuelve el elemento de la posición actual; `EsUltimo`, comprueba si hemos pasado de la última posición de la lista; `InsLista`, inserta un valor en la posición actual; `ElimLista`, elimina el elemento de la posición actual; y `Copiar`, copia una lista en otra.

Las definiciones de los tipos conjunto y diccionario serían las siguientes:

tipo

`Conjunto[T] = Lista[T]`

`Diccionario[tclave, tvalor] = Lista[Asociacion[tclave, tvalor]]`

Implementación de las operaciones

La implementación debe tener en cuenta que en las listas es importante el orden de inserción de los elementos y que puede haber elementos repetidos. Ninguna de esas dos cosas ocurren en los conjuntos. La implementación consistirá en manejar de forma adecuada las listas correspondientes.

operación Miembro (`C: Conjunto[T]; t: T`): booleano

`Primero(C)`

mientras (`Actual(C) ≠ t`) Y NO `EsUltimo(C)` **hacer**

`Avanzar(C)`

finmientras

devolver `Actual(C)=t`

operación Inserta (`var C: Conjunto[T]; t: T`)

si NO Miembro(`C, t`) **entonces**

`InsLista(C, t)` // Se inserta en la última posición

finsi

operación Suprime (`var C: Conjunto[T]; t: T`)

si Miembro(`C, t`) **entonces**

`ElimLista(C)` // La posición actual es ya el elemento a eliminar

finsi

En el caso de los diccionarios, debemos recordar que los accesos se hacen por la clave. Por ejemplo, la operación de consulta sería:

operación Consulta (`D: Diccionario[tclave, tvalor]; c: tclave`): `tvalor`

`Primero(D)`

mientras (`Clave(Actual(D)) ≠ c`) Y NO `EsUltimo(D)` **hacer**

`Avanzar(D)`

finmientras

si Clave(`Actual(D)`) = `c` **entonces**

devolver `Valor(Actual(D))`

sino

devolver `NULO_TVALOR`

finsi

Por otro lado, las operaciones: unión, intersección, etc., requieren recorrer completamente una de las listas y, para cada elemento, buscarlo en la otra lista.

operación Intersección ($C, D: \text{Conjunto}[T]; \text{var } R: \text{Conjunto}[T]$)

```

 $R := \text{ListaVacía}$ 
Primero( $C$ )
mientras NO EsUltimo( $C$ ) hacer
    si Miembro( $D$ , Actual( $C$ )) entonces
        InsLista( $R$ , Actual( $C$ ))
    finsi
    Avanzar( $C$ )
finmientras
```

operación Unión ($C, D: \text{Conjunto}[T]; \text{var } R: \text{Conjunto}[T]$)

```

 $R := \text{Copiar}(D)$ 
Primero( $C$ )
mientras NO EsUltimo( $C$ ) hacer
    si NO Miembro( $D$ , Actual( $C$ )) entonces
        InsLista( $R$ , Actual( $C$ ))
    finsi
    Avanzar( $C$ )
finmientras
```

Evaluación de la eficiencia

Sea n el tamaño, en término promedio, de los conjuntos representados y N el tamaño del conjunto universal; está claro que $n \leq N$. La representación de arrays de booleanos utiliza 1 bit para cada elemento, mientras que con listas se requiere un puntero más un valor del tipo T . Sin embargo, mientras que la primera representa los N posibles valores, la segunda está en función de n . En concreto, si k_1 es el número de bytes de un puntero y k_2 el número de bytes un valor de tipo T , una representación de listas usará $(k_1 + k_2)n$ bytes, lo cual está en $O(n)$. En la mayoría de los casos, ocurrirá que $n \ll N$.

Las operaciones que trabajan con un elemento, recorren toda la lista hasta encontrar la posición del elemento buscado. En el peor caso –si el elemento buscado no se encuentra– se recorrerá toda la lista y el tiempo de ejecución será un $O(n)$. Si la lista estuviera ordenada, por ejemplo de menor a mayor, sólo necesitaríamos recorrerla hasta encontrar un valor igual o mayor. En término medio reducimos el tiempo a la mitad, pero seguimos teniendo un $O(n)$. En cualquier caso, la eficiencia es mucho peor que con arrays, donde teníamos un $O(1)$.

Respecto a las operaciones unión, intersección, diferencia, etc., que trabajan con dos conjuntos, hemos visto que la implementación consiste en recorrer una lista y buscar cada elemento en el otro conjunto. Suponiendo ambos conjuntos de tamaño n , el orden de complejidad sería un $O(n^2)$. En este caso sí que podemos obtener una mejora usando listas ordenadas, como vamos a ver.

Mejora de la eficiencia con listas ordenadas

Supongamos que usamos listas ordenadas para representar los conjuntos. En la operación Miembro deberíamos cambiar la condición del bucle **mientras** a $\text{Actual}(C) < t$, y en Inserta deberíamos introducir el elemento en la posición adecuada.

Para conseguir la unión de dos conjuntos, utilizando listas ordenadas, podemos usar un procedimiento parecido al procedimiento de mezcla en la ordenación por mezcla. Recorremos las dos listas de forma simultánea, avanzando en cada caso con la que tenga el menor elemento.

operación Intersección ($C, D: \text{Conjunto}[T]; \text{var } R: \text{Conjunto}[T]$)

```

 $R := \text{ListaVacía}$ 
Primero( $C$ )
Primero( $D$ )
mientras NO EsUltimo( $C$ ) Y NO EsUltimo( $D$ ) hacer
    si Actual( $C$ ) = Actual( $D$ ) entonces
        InsLista( $R$ , Actual( $C$ ))
        Avanzar( $C$ )
        Avanzar( $D$ )
    sino si Actual( $D$ ) < Actual( $C$ ) entonces Avanzar( $D$ )
    sino Avanzar( $C$ )
    finsi
finmientras
```

En este caso la operación sólo recorre una vez cada elemento de cada una de las dos listas, tardando un tiempo constante para cada elemento. En el peor caso el procedimiento hace $2n$ pasos, con lo que el tiempo es un $O(n)$; la mejora es sustancial respecto al $O(n^2)$ que se necesita con listas no ordenadas.

En cualquier caso, las estructuras de representación de conjuntos mediante tablas y listas son adecuadas cuando los conjuntos utilizados son de tamaño reducido y conocido. Cuando no se cumplen esas condiciones, es necesario utilizar estructuras más avanzadas, como las tablas de dispersión o las que veremos en el capítulo siguiente.

3.2. Tablas de dispersión

Entre todos los tipos de operaciones existentes sobre conjuntos y diccionarios, el objetivo en muchas aplicaciones se encuentra en las operaciones que trabajan con un solo elemento: inserta, suprime y consulta; y en conseguir una implementación muy rápida. Ya hemos visto que la representación mediante listas requiere para estas operaciones un $O(n)$, resultante de tener que recorrer toda la lista de tamaño n .

Por ejemplo, en el buscador de Internet el diccionario **Claves** contendría unos 3.000 millones de asociaciones (*clave, dirección*). ¡Es imposible conseguir una implementación rápida si tenemos que recorrerlas todas secuencialmente! Y eso no es nada comparado con el tiempo constante, $O(1)$, necesario en la representación mediante arrays. Pero, desafortunadamente, lo inviable en ese caso es la memoria, ya que la tabla debería tener una posición para cada uno de los 2^{40} posibles valores de clave. Contando direcciones de 80 caracteres de longitud, necesitaríamos unos $2^{40} * 80 = 81.920$ Gbytes. Con la representación de listas, suponiendo punteros de 8 bytes, sólo usaríamos unos $3 * 10^9 * (80 + 8) = 246$ Gbytes, ¡300 veces menos! Así que, ¿cómo quedarnos con lo mejor de una y otra estructura?

3.2.1. La dispersión y los sinónimos

La clave de la representación mediante arrays de booleanos es que para cada elemento i sabemos que ocupa exactamente la posición i -ésima en el array, lo cual permite un acceso inmediato al mismo. El inconveniente es el tamaño excesivo de la tabla. Así que la solución es sencilla: usar una tabla de tamaño reducido y definir una función arbitraria que asocie a cada elemento una posición predefinida dentro de esa tabla. Esta función es lo que se conoce como **función de dispersión**²: dada una clave en $\text{Rango}(\text{tclave})$, devuelve un valor en el intervalo de posiciones de la tabla. Si la tabla es de tamaño B , el rango de posiciones será $[0, \dots, B - 1]$, con lo que la función de dispersión tiene la forma:

$$h : \text{Rango}(\text{tclave}) \rightarrow [0, \dots, B - 1]$$

Por ejemplo, una posible función de dispersión con claves enteras podría ser:

$$h(x) := \lfloor (275x^2)/77 + \sqrt{17 + 25x} + 33x + B|\cos(x)| \rfloor \bmod B$$

En la figura 3.2 se muestra gráficamente la idea de la dispersión. El objetivo de la función de dispersión, $h(x)$, es repartir los elementos de la forma más aleatoria y uniforme posible dentro del rango de posiciones de la tabla. La función estará definida sobre enteros, cadenas, reales, etc., según el tipo `tclave` de los datos almacenados.

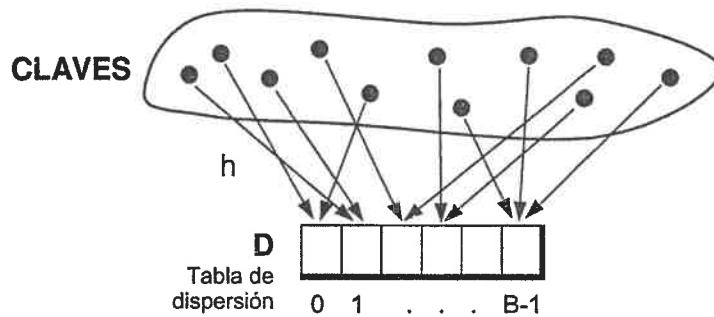


Figura 3.2: Representación gráfica de las funciones de dispersión. Dada una clave x la función de dispersión produce una posición $h(x)$ en el intervalo $[0, \dots, B - 1]$.

Básicamente, el funcionamiento ideal de una tabla de dispersión, supuesta una tabla D de tamaño $[0..B - 1]$ y un par $(\text{clave}, \text{valor})$, sería el siguiente:

- **Inserción.** Calcular $h(\text{clave})$, e insertar valor en esa posición de la tabla, es decir: $D[h(\text{clave})] := \text{valor}$.
- **Consulta.** Recordemos que, en un diccionario, se consulta una clave y el resultado es un valor. Por lo tanto, aplicamos $h(\text{clave})$ y devolvemos el elemento en esa posición, esto es: *devolver $D[h(\text{clave})]$* .
- **Eliminación.** La eliminación se realiza también por clave, así que consistiría en algo del tipo $D[h(\text{clave})] := \text{NULO}$.

²En inglés se pueden encontrar los términos *hash function*, *hash tables* y *hashing methods*.

Si la función de dispersión se puede calcular en un tiempo constante, las tres operaciones anteriores se podrían llevar a cabo en un tiempo constante, $O(1)$, en el mejor caso. Pero vamos a ver cómo surge un problema que hace que la implementación no sea tan sencilla, lo cual redonda también en los tiempos de ejecución.

Ejemplo 3.1 Utilizamos una tabla de dispersión para almacenar un conjunto de números enteros entre 0 y 1000. La tabla tiene tamaño $B = 10$, y la función de dispersión es: $h(x) = x \bmod 10$. Suponiendo que almacenamos los valores $\{263, 198, 782, 810, 362, 444, 753, 522, 107\}$, vamos a analizar la distribución de los elementos en la tabla.

0	1	2	3	4	5	6	7	8	9
810		782	263	444			107	198	
		362	753						
		522							

La función de dispersión $h(x) = x \bmod 10$ simplemente devuelve la última cifra decimal de x . Aunque es muy sencilla, tiene el inconveniente de no ser muy “aleatoria”. La aplicación de la función *módulo* 10, o en general *módulo* B , implica que el resultado siempre estará en el rango $[0, \dots, B - 1]$. Por este motivo suele aparecer en muchas funciones de dispersión.

El tamaño de la tabla de dispersión, B , se debe ajustar según el tamaño esperado de los conjuntos representados, y no según el rango de claves. De hecho, como ya hemos visto, en la mayoría de las aplicaciones ocurrirá que el rango de claves será mucho mayor de lo que nos podemos permitir. Esto tiene una implicación directa: si no conocemos a priori los elementos que serán almacenados, podrá ocurrir que para dos elementos distintos obtengamos el mismo valor de dispersión. Estos son los llamados **sinónimos**: dados dos elementos x e y distintos y una función de dispersión h , se dice que x e y son sinónimos si $h(x) = h(y)$. Por ejemplo, en la tabla del ejemplo 3.1 los valores $\{782, 362, 522\}$ son sinónimos; de hecho serán sinónimos todos los números que acaben en la misma cifra.

El problema de los sinónimos es que en una posición de la tabla sólo cabe un valor, así que ¿qué hacemos cuando dos elementos tienen el mismo valor de dispersión? Existen unas cuantas formas de solucionar el problema de los sinónimos; las distintas técnicas de dispersión difieren en la forma de tratarlos y, en principio, ninguna de ellas es mejor que otra en todos los casos. Los dos tipos principales de dispersión, que vamos a estudiar en este capítulo, son:

- **Dispersión abierta.** Las posiciones de la tabla no son elementos sino listas de elementos, así que todos los sinónimos irán dentro de la misma lista.
- **Dispersión cerrada.** Si al insertar un elemento la posición que le corresponde ya está ocupada, buscamos otra posición libre donde insertarlo.

3.2.2. Dispersión abierta

En la dispersión abierta –también conocida como dispersión externa o encadenamiento– se adopta una estrategia sencilla para solucionar el problema de los sinónimos:

las posiciones de la tabla de dispersión son listas de valores del tipo almacenado (es decir, elementos o asociaciones) a las cuales irán a parar todos los sinónimos que tengan ese valor de h . Estas listas de la tabla son también llamadas **cubetas**, en analogía con un cubo donde cabe un número indefinido de cosas.

La definición del tipo de datos podría tener la siguiente forma:

tipo

Conjunto[T] = array [0..B-1] de Lista[T]

Diccionario[tclave, tvalor] = array [0..B-1] de Lista[Asociacion[tclave, tvalor]]

Es cuestión del implementador decidir qué tipo de listas utilizar: enlazadas, uni o bidireccionales, circulares, ordenadas o no, etc. En general, podríamos usar también cualquier otra estructura que permita añadir dinámicamente un número indeterminado de datos, como por ejemplo árboles. La implementación de las operaciones insertar, consultar y suprimir, se convierte simplemente en aplicar $h(x)$ y hacer la inserción, consulta o eliminación sobre la lista correspondiente.

Ejemplo 3.2 Utilizamos una estructura de dispersión abierta para representar los datos del ejemplo 3.1. La tabla tiene $B = 10$ cubetas y la función de dispersión es $h(x) = x \bmod 10$. Los valores almacenados son $\{263, 198, 782, 810, 362, 444, 753, 522, 107\}$. Suponiendo que usamos listas enlazadas y no ordenadas, la tabla resultante sería la mostrada en la figura 3.3.

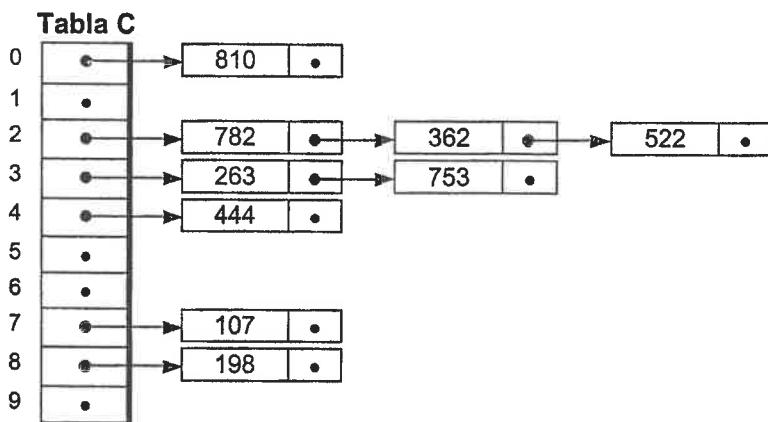


Figura 3.3: Ejemplo de tabla de dispersión abierta, usando listas enlazadas y no ordenadas.

Por ejemplo, la implementación de la operación **Miembro(C, t)** usando conjuntos representados con dispersión abierta sería como la siguiente. Suponiendo el tipo de datos **Lista[T]** visto en el punto 3.1.4, con las operaciones **Primero**, **Actual**, **Avanzar** y **EsUltimo**.

operación Miembro ($C: \text{Conjunto}[T]; t: T$): booleano

var cubeta: Lista[T]

cubeta:= $C[h(t)]$

Primero(cubeta)

mientras NO EsUltimo(cubeta) Y (Actual(cubeta) ≠ t) hacer

```

Avanzar(cubeta)
finmientras
devolver Actual(cubeta) = t

```

Evaluación de la eficiencia

Si utilizamos listas enlazadas para representar las cubetas de la tabla, el tiempo de ejecución de las operaciones inserta, suprime y consulta será proporcional al tamaño de esas listas. Suponiendo que el conjunto contiene n elementos, en el mejor caso –si la función de dispersión los reparte de forma equitativa entre las B cubetas– cada lista tendrá un tamaño promedio de n/B elementos. El tiempo de ejecución de las operaciones será³ un $O(1 + n/B)$. Si $B \geq n$ entonces tenemos en promedio un tiempo constante.

Sin embargo, en el peor caso –si h es tal que todos los elementos son sinónimos– sólo se utilizará una de las cubetas, cuya lista tendrá una longitud n . Así que el tiempo de las operaciones será un $O(n)$ en ese caso. En consecuencia, elegir una buena función de dispersión es imprescindible para conseguir una implementación eficiente de las operaciones. La función de dispersión será buena si reparte los elementos de forma uniforme entre las cubetas.

En cuanto a la memoria utilizada en la tabla, suponiendo que cada puntero ocupa k_1 bytes y cada asociación k_2 bytes, tenemos por un lado lo correspondiente a la tabla en sí, $B k_1$, y por otro lo correspondiente a las listas, $n(k_1 + k_2)$. En total necesitamos $B k_1 + n(k_1 + k_2)$ bytes.

3.2.3. Dispersión cerrada

En la dispersión cerrada –también conocida como dispersión interna– las posiciones de la tabla de dispersión no son listas sino valores concretos del tipo a almacenar, es decir del tipo elemento o asociación. La definición del tipo sería:

```

tipo
Conjunto[T] = array [0..B-1] de T
Diccionario[tclave, tvalor] = array [0..B-1] de Asociacion[tclave, tvalor]

```

Por lo tanto, a diferencia de la dispersión abierta, sólo se ocupa la memoria requerida para la tabla, sin necesitar almacenamiento adicional en listas. Pero esto también implica que en la tabla sólo caben B elementos, frente al número ilimitado en la dispersión abierta.

Es más, en cada posición de la tabla sólo cabe un elemento. ¿Qué pasa entonces con los sinónimos? Cuando intentamos insertar un elemento x y nos encontramos que la posición $h(x)$ está ocupada por otro elemento, decimos que ocurre una **colisión**. En caso de colisión se debe buscar una nueva posición donde colocar ese elemento. Este proceso de “buscar una nueva posición” es lo que se conoce como **redispersión**.

La nueva posición donde meter el elemento vendrá dada por otra función $h_1(x) : Rango(clave) \rightarrow [0, \dots, B - 1]$. Si la celda $h_1(x)$ de la tabla está libre, el elemento se mete en ese lugar. En otro caso, si también está ocupada, entonces volvemos a redispersar:

³Añadimos un 1 para tener en cuenta las operaciones que se realizan siempre, de forma fija, como aplicar $h(x)$ y acceder a la primera posición. Es necesario en este caso, porque aunque $B \gg n$ el tiempo no tiende a 0 sino que será constante.

aplicamos otra función $h_2(x)$ y repetimos el proceso. Así debemos repetir hasta encontrar una posición libre donde meter x , hasta encontrar x (si ya estaba en la tabla) o hasta comprobar que no quedan huecos libres en la tabla de dispersión (es decir, después de haber hecho $B - 1$ intentos). La familia de funciones $h_i(x)$, para $i = 1, 2, \dots, B - 1$, es conocida como la **función de redispersión**.

Un ejemplo sencillo de función de redispersión es la conocida como **redispersión lineal**, que realiza una búsqueda lineal y circular en la tabla, a partir de la posición $h(x)$:

$$h_i(x) = (h(x) + i) \bmod B$$

O bien:

$$h_i(x) = (h(x) - i) \bmod B$$

Ejemplo 3.3 Supongamos que la tabla del ejemplo 3.1 se representa usando dispersión cerrada. La función de dispersión es $h(x) = x \bmod 10$, y se aplica redispersión lineal, es decir, $h_i(x) = (h(x) + i) \bmod 10$. Vamos a estudiar cómo se insertarían los elementos en la tabla, suponiendo el orden de inserción: 263, 198, 782, 810, 362, 444, 753. El resultado aparece en la figura 3.4.

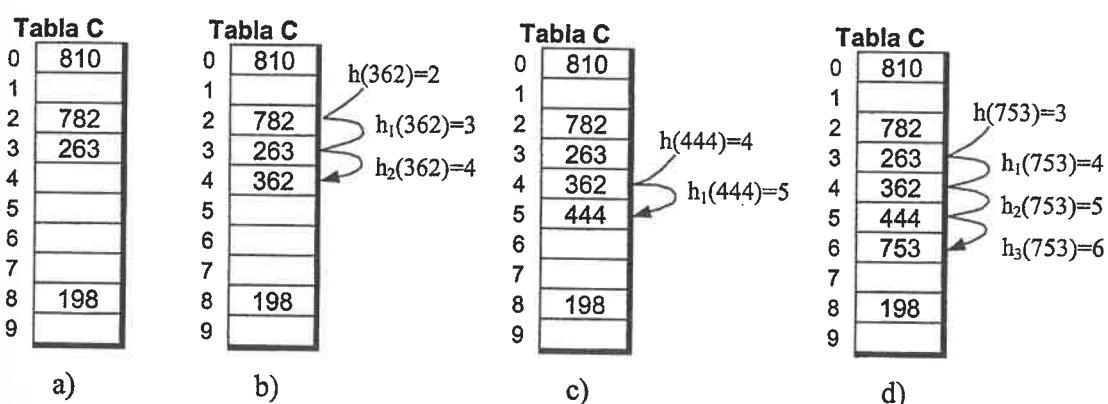


Figura 3.4: Ejemplo de tabla de dispersión cerrada, usando redispersión lineal. a) Despues de insertar 263, 198, 782 y 810. b) Despues de insertar 362. c) Despues de insertar 444. d) Despues de insertar 753.

La lista de posiciones que son comprobadas al insertar un elemento en la tabla es conocida como su **cadena o secuencia de búsqueda**. Por ejemplo, en el caso anterior la secuencia de búsqueda del elemento 753 es la cadena $\{3, 4, 5, 6\}$. Por un lado, está claro que la secuencia que se recorre debe ser la misma cuando se inserta un elemento, que posteriormente cuando se elimina, consulta o modifica. Por otro lado, la eficiencia de las operaciones será inversamente proporcional a la longitud de las secuencias de búsqueda. En este factor juegan un papel fundamental las funciones de dispersión y redispersión.

Implementación de las operaciones

Las operaciones de inserción, consulta y eliminación tienen una estructura similar: dado un elemento x , comprobar su secuencia de búsqueda –es decir, $h(x)$, $h_1(x)$, $h_2(x)$,

$h_3(x)$, ... – hasta encontrar ese elemento, una posición vacía o recorrer toda la tabla. Después, hacemos lo que corresponda en cada caso. De cara a la implementación, podemos definir una función auxiliar que realice este proceso de búsqueda. Vamos a suponer que trabajamos con el TAD Diccionario con la definición dada al principio de este punto.

operación Buscar (*D*: Diccionario[tclave,tvalor]; *c*: tclave): entero

```

i:= 1
k:= h(c)
mientras (i < B) Y (D[k].clave ≠ NULO) Y (D[k].clave ≠ c) hacer
    k:= hi(c)
    i:= i + 1
finmientras
devolver k

```

Usando esta función, la implementación de las operaciones Inserta y Consulta son inmediatas. Recordemos que en la operación de inserción, si ya está esa clave, se hace una modificación del valor asociado a la misma.

operación Inserta (**var** *D*: Diccionario[tclave,tvalor]; *c*: tclave; *v*: tvalor)

```

k:= Buscar(D, c)
si (D[k].clave = NULO) O (D[k].clave = c) entonces
    D[k].clave:= c
    D[k].valor:= v
sino
    error("La tabla está llena.")
finsi

```

operación Consulta (*D*: Diccionario[tclave,tvalor]; *c*: tclave): tvalor

```

k:= Buscar(D, c)
si (D[k].clave = c) entonces
    devolver D[k].valor
sino
    devolver NULO
finsi

```

Un valor NULO de clave significa que la posición correspondiente de la tabla está vacía. Para la operación Suprime, la idea directa sería buscar primero la clave, y colocar NULO en caso de encontrarla. Pero, ¡cuidado, podemos estar rompiendo alguna secuencia de búsqueda con resultados catastróficos! Si algún elemento *y* produjo una colisión con el elemento eliminado *x*, entonces la secuencia de búsqueda de *y* pasa por la posición a eliminar. Pero si colocamos un NULO en esa posición, la búsqueda de *y* pararía en esa posición, creyendo que *y* no está en la tabla. Por ejemplo, en el caso del ejemplo 3.3 esto ocurriría si eliminamos 362 y luego buscamos 444 ó 753.

Una solución sencilla para este problema es definir un valor especial de clave: SUPRIMIDO, que indique la posición donde había un elemento que ha sido eliminado.

operación Suprime (**var** *D*: Diccionario[tclave,tvalor]; *c*: tclave)

```

k:= Buscar(D, c)
si (D[k].clave = c) entonces
    D[k].clave:= SUPRIMIDO
    D[k].valor:= NULO

```

finsi

En la operación de búsqueda, se sigue buscando si encontramos un SUPRIMIDO, con lo cual no rompemos ninguna secuencia de búsqueda. Sin embargo, en la inserción esa posición sí puede ser utilizada como un espacio libre donde colocar un elemento.

Si las eliminaciones e inserciones en la tabla son frecuentes, la utilización de marcas SUPRIMIDO acarrea una pérdida de eficiencia. Una opción mejor –aunque más compleja de implementar– es no utilizar estas marcas especiales, sino reorganizar los elementos cuya secuencia de búsqueda pasa por la posición eliminada, de forma que la tabla quede en un estado consistente. Por ejemplo, si suponemos que se usa redispersión lineal, entonces sabemos que los elementos que han podido colisionar con el eliminado se encontrarán en posiciones consecutivas.

operación Suprime (**var** D: Diccionario[tclave,tvalor]; c: tclave)

```

k:= Buscar(D, c)
si (D[k].clave = c) entonces
    sup:= k
    k:= (k + 1) mod B
    mientras (D[k].clave ≠ NULO) Y (sup ≠ k) hacer
        si ColisionaCon (k, sup) entonces
            D[sup].clave:= D[k].clave
            D[sup].valor:= D[k].valor
            sup:= k
        finsi
        k:= (k + 1) mod B
    finmientras
    D[sup].clave:= NULO
    D[sup].valor:= NULO
finsi

```

La operación ColisionaCon(k, sup) comprueba si la secuencia de búsqueda del elemento colocado en la posición k pasa por la posición sup . La secuencia de búsqueda de ese elemento empieza en $h(D[k].clave)$, sigue en $h(D[k].clave) + 1, \dots$, y así circularmente hasta llegar a k . La implementación sería la siguiente:

operación ColisionaCon (k, sup : entero): booleano

```

inic:= h(D[k].clave)
devolver (sup - inic) mod B < (k-inic) mod B

```

Evaluación de la memoria necesaria

La memoria utilizada en una tabla de dispersión cerrada viene dada por el tamaño de la tabla; no hay un uso adicional de memoria. Si la tabla es de tamaño B y cada elemento o asociación requiere k_2 bytes, la tabla ocupa $B k_2$ bytes. ¿Cómo es este valor en comparación con la dispersión abierta, que requiere $B k_1 + n(k_1 + k_2)$ bytes (siendo k_1 el tamaño de un puntero)? Si $k_1 \geq k_2$, entonces la dispersión cerrada siempre ocupará menos memoria.

Pero lo normal es que ocurra lo contrario y hasta que $k_2 \gg k_1$. Por ejemplo, en el diccionario Páginas del buscador, las asociaciones contienen enteros de 5 bytes y descrip-

ciones. Una descripción completa de una página podría requerir, digamos, un tamaño k_2 de varios kilobytes. En ese caso, el tamaño de un puntero k_1 es despreciable frente a k_2 . Mientras la dispersión abierta requiere $n k_2$ bytes, la cerrada utilizaría $B k_2$, independientemente de lo que valga n pero siendo siempre con $n \leq B$. Es decir, en la dispersión cerrada debemos reservar espacio para el mayor tamaño posible del conjunto o diccionario, usando siempre una cantidad fija, aunque la tabla esté llena o vacía.

Una solución al problema anterior en dispersión cerrada es definir la tabla como un array de punteros a asociaciones, inicialmente todos con valor NULO.

tipo

Conjunto[T] = array [0..B-1] de Puntero[T]

Diccionario[tclave, tvalor] = array [0..B-1] de Puntero[Asociacion[tclave,tvalor]]

Sólo las cubetas que estén ocupadas tendrán en su puntero un valor no nulo. Ahora, la memoria necesaria sería la correspondiente a la tabla, $B k_1$, más la usada por cada elemento existente, $n k_2$. Aunque este valor $B k_1 + n k_2$ es menor que el de la dispersión abierta, hay que recordar que en dispersión abierta es posible que $n \geq B$, cosa que no ocurre en dispersión cerrada.

Otra posibilidad es utilizar la técnica conocida como **reestructuración** de tablas de dispersión. La idea consiste en usar dispersión cerrada, pero variando dinámicamente el tamaño B de la tabla conforme aumenta el número de elementos n . De esta forma, podemos empezar con un tamaño inicial, por ejemplo $B = 100$. Cuando se llene la tabla, creamos otra, por ejemplo de tamaño $B = 200$, y pasamos todos los elementos de la tabla antigua a la nueva. La operación de inserción debería tener en cuenta este hecho y la función de dispersión también debería adaptarse según el valor de B .

Evaluación de la eficiencia computacional

En dispersión cerrada, el tiempo de ejecución de las operaciones depende del tamaño de las secuencias de búsqueda. Podemos establecer una similitud entre la longitud de las secuencias de búsqueda en dispersión cerrada y el número de elementos en cada lista en dispersión abierta. A medida que tengamos más elementos n , ambas serán mayores y la eficiencia de las operaciones será menor. Sin embargo, hay algunas diferencias en cuanto a la forma en que crece el tiempo en ambos casos.

- En dispersión cerrada caben como máximo B elementos, mientras que en dispersión abierta cabe un número indeterminado.
- En dispersión abierta las listas (correspondientes a cada cubeta) contienen todos los sinónimos. En dispersión cerrada la secuencia de búsqueda de un valor puede incluir sinónimos y valores que no sean sinónimos; depende de la función de redispersión.

Del segundo punto se concluye que si los sinónimos tienen la misma secuencia de búsqueda, entonces la eficiencia de la dispersión cerrada es siempre peor. Esto ocurre, por ejemplo, con la redispersión lineal: si $h(x) = h(y)$ entonces $h_i(x) = (h(x) + i) \bmod B = (h(y) + i) \bmod B = h_i(y)$.

Supongamos que la tabla de dispersión cerrada contiene n elementos –repartidos de forma uniforme por la tabla– y que queremos insertar un nuevo elemento x . Entonces, la

probabilidad de que ocurra una colisión en $h(x)$ será n/B , que es el porcentaje de cubetas ocupadas. En caso de colisión se aplicará redispersión, buscando entre $B - 1$ posiciones de las cuales $n - 1$ están ocupadas. La probabilidad de una nueva colisión será $(n - 1)/(B - 1)$, por lo que la probabilidad de al menos dos colisiones será $\frac{n(n-1)}{B(B-1)}$. De la misma forma, la probabilidad de al menos tres colisiones será $\frac{n(n-1)(n-2)}{B(B-1)(B-2)}$ y así sucesivamente.

Si n y B son suficientemente grandes, las anteriores probabilidades se pueden aproximar a n/B , $(n/B)^2$, $(n/B)^3$, etc. La longitud promedio de las secuencias de búsqueda será la suma de cada posible longitud por la probabilidad de esa longitud. Como mínimo –si no hay colisiones– tenemos longitud 1; luego tendremos longitud 2 o más con probabilidad n/B , 3 o más con $(n/B)^2$ y así sucesivamente. Por lo tanto, el tamaño promedio aproximado de las secuencias de búsqueda será:

$$1 + \sum_{i=1}^n \left(\frac{n}{B}\right)^i \approx \sum_{i=0}^{\infty} \left(\frac{n}{B}\right)^i = \frac{1}{1 - n/B} \quad (3.1)$$

La longitud de las secuencias de búsqueda está en función de n/B , es decir del porcentaje de ocupación de la tabla. Recordemos que el tiempo de ejecución es proporcional a esa longitud, esto es, será un $O(1/(1 - n/B))$. Cuando n/B es próximo a 0, el tiempo es próximo a un tiempo constante. Sin embargo, cuando el porcentaje tiende a 1 el tiempo crece rápidamente. El crecimiento de esta función –comparado con el crecimiento del tiempo en dispersión abierta– es mostrado en la figura 3.5.

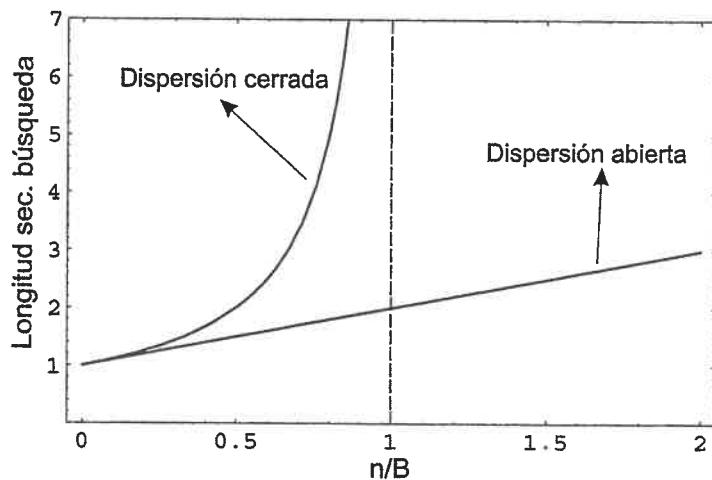


Figura 3.5: Comparación de la eficiencia en dispersión abierta y cerrada. Longitud promedio de las secuencias de búsqueda (en inserción o búsqueda sin éxito) en función del porcentaje de llenado de la tabla, n/B .

Una estrategia habitual al implementar estructuras de dispersión es establecer una máxima capacidad de llenado de las cubetas. Si se supera esa capacidad, se aplica una reestructuración de la tabla. Por ejemplo, si fijamos que las longitudes de las secuencias de búsqueda no superen 3 pasos en promedio, entonces debemos aplicar reestructuración

en dispersión cerrada cuando $n \approx 0,75B$ y en dispersión abierta cuando $n \approx 2B$. En ese caso, creamos una nueva tabla usando un B mayor, típicamente el doble.

3.2.4. Funciones de dispersión

Ya hemos visto que una buena función de dispersión es fundamental para conseguir eficiencia en una tabla de dispersión. Pero, ¿qué características debe tener una buena función de dispersión? Básicamente dos:

- La función debe minimizar la probabilidad de que aparezcan sinónimos. Para ello, debería repartir los elementos aleatoriamente y de manera uniforme entre el rango de posiciones de la tabla.
- Puesto que buscamos eficiencia, la función de dispersión en sí debería ser rápida de calcular. Normalmente se utilizan funciones que se pueden calcular en un tiempo constante. En caso de usar funciones más costosas, habría que considerar su tiempo en el estudio de la eficiencia.

Por otro lado, está claro que una función de dispersión, con la forma $h(x)$, sólo depende de x , es decir siempre devuelve el mismo valor para la misma x . Esto no es sólo una propiedad de una buena función de dispersión, sino de cualquiera. En consecuencia, la *aleatoriedad* antes mencionada es, más propiamente, lo que se denomina *pseudoaleatoriedad*. Para conseguirla debemos intentar que el valor de dispersión dependa de la clave de una manera “no inmediata”. Por ejemplo, si las claves son enteros de 5 bytes y $h(x) = x \bmod 256$, entonces la función de dispersión sólo depende del último byte.

Si se conocen a priori los elementos almacenados –o los que serán almacenados con más probabilidad– podremos hacer un estudio riguroso de qué función de dispersión produce menos colisiones. En otro caso, se suelen aplicar funciones de dispersión genéricas, como las expuestas a continuación. En las siguientes funciones suponemos que las claves son números enteros. Si tenemos cadenas, caracteres, reales u otro tipo, siempre será posible hacer una conversión a enteros. Por ejemplo, en el caso de las cadenas de caracteres podemos sumar los códigos ASCII de cada carácter, o bien considerar una cadena de n caracteres como un entero de n bytes.

Métodos de multiplicación y división

Los métodos basados en multiplicación y división son los tipos más sencillos de función de dispersión existentes. Tienen la siguiente forma:

- **Multiplicación.** $h(x) = (x C) \bmod B$
- **División.** $h(x) = (x \bmod P) \bmod B$

En el primero, es necesario que C y B sean primos entre sí. En otro caso, $h(x)$ no produciría todos los posibles valores del intervalo $[0, \dots, B - 1]$. En el segundo, debe ser $P > B$, y también se aconseja que ambos números sean primos entre sí, aunque no resulta esencial. Estos métodos suelen ser los preferidos cuando no tenemos mucha información sobre las claves.

Métodos de plegado o folding

En la técnica de plegado (o en inglés *folding*) se supone que las claves están formadas por una cadena de números, de la forma $x = x_1x_2x_3 \dots x_k$. Las cadenas son agrupadas de n en n , y luego todas las partes son combinadas con una suma o un XOR. Por ejemplo, suponiendo que las claves son DNI podemos agrupar los dígitos decimales de 3 en 3. Si tenemos el número 73.483.112, los grupos serían 73, 483 y 112, y el valor de dispersión sería: $h(73.483.112) = (73 + 483 + 112) \bmod B$. Matemáticamente, la función se puede expresar de la siguiente manera:

$$h(x) = (\lfloor x/10^6 \rfloor + \lfloor x/10^3 \rfloor \bmod 10^3 + x \bmod 10^3) \bmod B$$

También sería posible trabajar a nivel de bytes, en cuyo caso el plegado se haría utilizando desplazamientos lógicos. Además, de esta forma la función de dispersión se podría aplicar más fácilmente a cualquier tipo de claves, sean enteros o no.

Otra variante del plegado consiste en invertir el orden de alguna de las subcadenas. Por ejemplo, suponiendo la clave $x_1x_2x_3x_4x_5x_6x_7x_8x_9$ y tomando partes de 3 en 3, podríamos invertir la subcadena central: $(x_1x_2x_3) + (x_6x_5x_4) + (x_7x_8x_9)$. En principio, nada garantiza que vaya a funcionar mejor de esta forma.

Método del centro del cuadrado

Ya hemos comentado que para conseguir una buena función de dispersión es interesante que el valor obtenido dependa de todas las partes de la clave. Una forma de conseguirlo es multiplicar la clave por sí misma y tomar los dígitos centrales. Por ejemplo, si la clave es $x_1x_2x_3$, el cuadrado será $(x_1*x_1)(2x_1*x_2)(x_2*x_2 + 2x_1*x_3)(2x_2*x_3)(x_3*x_3)$. Se puede ver que el tercer dígito del cuadrado depende de todos los dígitos del número.

La forma de aplicar esta técnica resulta bastante sencilla. Siendo las claves enteros de 4 cifras decimales, la función de dispersión podría tener la forma:

$$h(x) = \lfloor x^2/1000 \rfloor \bmod B$$

O en general:

$$h(x) = \lfloor x^2/C \rfloor \bmod B$$

Para que este método funcione adecuadamente se recomienda usar un valor de C tal que $BC^2 \approx K^2$, siendo el rango de las claves $x = 0, 1, \dots, K$.

Método de extracción

En los métodos anteriores se buscaba una función de dispersión que dependiera de todas las partes de la clave. Si la clave es de tamaño fijo y reducido la función se puede calcular en un tiempo constante. Pero, ¿qué pasaría si la aplicamos sobre cadenas de caracteres de tamaño arbitrario? Por ejemplo, en la aplicación del buscador podríamos tener una tabla de dispersión indexada por cadenas, que dada una dirección de una página devuelva su clave. Una función de plegado requeriría recorrer todos los caracteres, tardando un $O(n)$ para cadenas de longitud n .

La extracción tiene por objetivo limitar el tiempo de ejecución a una constante. Para claves excesivamente largas, simplemente se extrae una parte de las mismas y se aplica cualquiera de los métodos anteriores sobre la parte extraída. Por ejemplo, en el caso de las cadenas de caracteres podemos extraer los caracteres del tercero al sexto y convertirlo

a un entero de 4 bytes.

Método de base o radix

En el intento de conseguir una función de valor aleatorio o impredecible, es posible hacer un cambio de base de la clave y considerarlo luego en la base original. Por ejemplo, el número 1999 en base decimal, lo podemos convertir a base 9, dando lugar a 2661; ahora podemos considerar 2661 como si estuviera en base 10 y aplicar $2661 \bmod B$.

Está claro que si aplicamos base 9, nunca se obtendrán valores que contengan algún 9, lo cual produciría una mala función de dispersión. No obstante, al aplicar módulo B sí que se podrán obtener todas las posibles posiciones de la tabla.

El mayor inconveniente de esta técnica, respecto a las anteriores, es que el cambio de base puede resultar un cálculo excesivamente costoso. En general, debemos buscar un equilibrio entre la aleatoriedad de una función y el coste de calcularla.

3.2.5. Estrategias de redispersión

Por muy buena que sea una función de dispersión, siempre podrán aparecer sinónimos que den lugar a colisiones⁴. En la dispersión abierta no hay colisiones (propiamente dichas), puesto que todos los sinónimos son almacenados en una lista. En dispersión cerrada las colisiones se resuelven volviendo a dispersar el elemento, es decir buscando una nueva posición donde meterlo. Ya hemos visto que esto da lugar a la idea de *secuencia de búsqueda* para un elemento.

Es posible asociar el concepto de secuencia de búsqueda, de dispersión cerrada, con las listas asociadas a cada cubeta, en dispersión abierta. Pero, mientras que las listas contienen todos los elementos sinónimos, las secuencias en dispersión cerrada pueden contener sinónimos o no. Por ejemplo, si usamos redispersión lineal, es evidente que dos elementos sinónimos producirán las mismas cadenas de búsqueda. Es más, las cadenas contendrán todos los sinónimos y podrán contener alguno que no sea sinónimo. En consecuencia el tiempo de las operaciones será siempre mayor o igual que en dispersión abierta.

A parte de lo anterior, la redispersión lineal tiene otro inconveniente conocido como el **problema de agrupamiento**, que se puede apreciar en el ejemplo de la figura 3.4. El problema ocurre cuando se llena un grupo de cubetas consecutivo. Cualquier inserción que caiga en una posición dentro de ese grupo dará lugar a que se tenga que recorrer todo el grupo de forma secuencial. El elemento se colocará al final del grupo, incrementando así el problema para las inserciones posteriores.

En consecuencia, una buena función de redispersión, $h_i(x)$, debería tener las siguientes propiedades:

- La función debería recorrer todos los valores posibles en la tabla, para los distintos valores de i .
- Una buena función de redispersión debe evitar el problema del agrupamiento. Idealmente, los sinónimos deberían tener secuencias de búsqueda distintas.

⁴A menos que se conozcan exactamente los valores a almacenar, y se diseñe una función de dispersión específica para el caso. Esto es lo que se conoce como *función de dispersión perfecta*.

- Igual que en la función de dispersión, el tiempo de calcular la función de redispersión debería ser rápido, con el objetivo último de conseguir eficiencia.

Redispersión con saltos de tamaño C

Con el fin de evitar la formación de grupos consecutivos, podemos pensar en modificar la redispersión lineal, usando ahora saltos de cierto tamaño constante C .

$$h_i(x) = (h(x) + Ci) \bmod B$$

Para que se recorran todas las posiciones de la tabla, es necesario que C y B sean primos entre sí. Por ejemplo, si $B = 8$, $C = 3$ y $h(x) = 0$, los sucesivos valores de $h_i(x)$ serían: 3, 6, 1, 4, 7, 2, 5. Si B es un número primo, podemos utilizar cualquier valor de C . Esta estrategia de redispersión es muy sencilla pero no evita el problema del agrupamiento; simplemente se producirán grupos a saltos de tamaño C .

Redispersión cuadrática

La idea de la redispersión cuadrática es parecida a la de la redispersión lineal, pero en lugar de usar un recorrido lineal, utilizamos saltos que se incrementen de forma cuadrática. Además, para que la técnica funcione correctamente los saltos deben ir alternando entre $+i^2$ y $-i^2$. La función de redispersión tiene la siguiente forma:

$$h_i(x) = (h(x) + (-1)^{i-1} \lfloor (i+1)/2 \rfloor^2) \bmod B$$

Esta función no garantiza que se recorran todas las posiciones de la tabla para cualquier valor de B . Se ha comprobado que para que se recorran todas las posiciones, B debe ser un número primo de la forma $B = 4k + 3$, para algún entero k . Por ejemplo, si $h(x) = 0$ y $k = 2$, entonces $B = 11$ y los sucesivos valores de redispersión serían: 1, 10, 4, 7, 9, 2, 5, 6, 3, 8.

La redispersión cuadrática evita el problema del agrupamiento, pero sólo parcialmente. Los sinónimos tienen secuencias de búsqueda idénticas, por lo que los grupos estarán compuestos por los grupos de sinónimos existentes. Aunque el problema es menor, aún sigue siendo un inconveniente.

Redispersión con permutación prefijada

Un grupo amplio de funciones de dispersión son las formadas a partir de cierta permutación de los valores $\{1, 2, \dots, B - 1\}$, que es prefijada de antemano. Sea esta permutación la serie de valores: D_1, D_2, \dots, D_{B-1} , la función de dispersión sería:

$$h_i(x) = (h(x) + D_i) \bmod B$$

Esta técnica se puede considerar como una generalización de las anteriores. Por ejemplo, si elegimos la permutación $(1, 2, 3, 4, 5, \dots)$ tendremos redispersión lineal, y con $(1, -1, 4, -4, 9, -9, 16, \dots)$ tendremos una redispersión cuadrática. Si B es potencia de 2, se puede conseguir una permutación mediante el siguiente algoritmo.

- Tomamos una constante *adecuada* K en el intervalo $(1, 2, \dots, B - 1)$.
- Empezamos con un valor inicial D_1 cualquiera, también en el intervalo $(1, 2, \dots, B - 1)$.

- Suponiendo que el valor D_{i-1} de la permutación tiene una representación binaria (d_1, d_2, \dots, d_n) , el siguiente valor D_i viene dado por: si $d_1 = 0$ entonces $D_i = (d_2, d_3, \dots, d_n, 0)$; en otro caso $D_i = (d_2, d_3, \dots, d_n, 0) \text{ XOR } K$.

Por ejemplo, para $B = 8$ podemos tomar $K = 5$ y empezando con $D_1 = 1$ tenemos la siguiente permutación.

D_1	D_2	D_3	D_4	D_5	D_6	D_7
001	010	100	000 $\oplus 101$ =101	010 $\oplus 101$ =111	110 $\oplus 101$ =011	110
1	2	4	5	7	3	6

De nuevo, esta función de redispersión tiene el inconveniente de producir las mismas secuencias de búsqueda para los elementos sinónimos, ya que la permutación es prefijada de antemano. Una alternativa podría ser utilizar distintos valores iniciales para los sinónimos. De esta forma, tenemos que el valor inicial D_1 debería depender de la clave, por ejemplo $D_1(x) = x \bmod (B - 1) + 1$. Los siguientes valores serían calculados según el algoritmo anterior.

Redispersión con saltos de tamaño variable

Basándonos en la idea de la redispersión con saltos de tamaño C , podemos definir una función similar pero utilizando tamaños de paso distintos para valores de clave distintas. Los tamaños de paso válidos deberían ser primos respecto al tamaño de la tabla B . Si B es un número primo, cualquier C entre 1 y $B - 1$ será válido. Con esto, la función de redispersión sería:

$$h_i(x) = (h(x) + iC(x)) \bmod B$$

Donde $C(x)$ es una función cualquiera –al estilo de la función de dispersión– que devuelve un valor en el intervalo: $(1, \dots, B - 1)$. Por este motivo, esta técnica se suele conocer también como **dispersión doble**. Por ejemplo, una función válida podría ser $C(x) = 1 + (x^2/D + Ex) \bmod (B - 1)$.

Eligiendo $C(x)$ de forma adecuada, podemos conseguir que los sinónimos produzcan secuencias de búsqueda distintas, resolviendo así el problema del agrupamiento.

3.3. Combinando estructuras de datos

Hasta ahora hemos estudiado las estructuras de datos como realizaciones en memoria de un tipo abstracto de datos. En la práctica, al diseñar una estructura, tenemos en cuenta cuestiones de eficiencia que son obviadas en el estudio del tipo abstracto. En una aplicación real necesitaremos, posiblemente, hacer adaptaciones de las estructuras estudiadas e incluso combinar varias de ellas en lo que podríamos llamar una *estructura múltiple*. Pero, ¿por qué iba a ser necesario combinar dos estructuras para los mismos datos? Y ¿cómo hacer la combinación?

La eficiencia de una estructura de datos es un concepto relativo: la eficiencia se mide en función de las operaciones que se vayan a utilizar sobre esa estructura. Una estructura

puede resultar eficiente para cierto tipo de operaciones pero muy ineficiente para otro tipo. Por ejemplo, hemos visto que las tablas de dispersión son muy eficientes para acceder a cierto elemento de forma directa según su clave. Pero ¿y si queremos acceder de forma ordenada, por ejemplo buscar el máximo o el mínimo? O ¿cómo podríamos implementar una operación que acceda por un atributo que no sea la clave? Deberíamos acceder a todas las cubetas de la tabla, ver si están ocupadas o no, y en caso de estarlo comprobar el elemento y actuar de forma adecuada. En definitiva, resultarían muy ineficientes para ese tipo de operaciones.

3.3.1. Estructuras de datos múltiples

Si sobre unos mismos datos necesitamos distintos tipos de operaciones, entonces posiblemente no exista una estructura que nos proporcione eficiencia para todas ellas. En ese caso, puede ser adecuado combinar varias estructuras –que refieran a los mismos datos– cada una de las cuales nos proporciona un acceso rápido para cierto tipo de operaciones.

Supongamos, por ejemplo, que en una aplicación de gestión empresarial almacenamos información sobre los empleados. En concreto, tenemos el nombre de cada empleado, su dirección, sueldo, DNI y teléfono (o teléfonos, si tiene más de uno). Las operaciones que más se van a utilizar serán las siguientes:

- Dado un DNI (A1), un nombre (A2) o un teléfono (A3), devolver todos los datos del empleado correspondiente.
- Insertar (B1), eliminar (B2) o modificar (B3) un empleado, accediendo por número de DNI.
- Listar los empleados por sueldo (C1) o por número de DNI (C2).

Para almacenar los datos relativos a un empleado podemos utilizar un tipo de datos **Empleado**, definido del siguiente modo:

tipo

Empleado = registro
Nombre: cadena
Direccion: cadena
DNI: entero
Telefonos: Lista[entero]
Sueldo: entero

finregistro

Pero, realmente el problema no es cómo representar un empleado, sino cómo almacenar el conjunto de todos los empleados de forma que todas las operaciones anteriores se puedan realizar de forma eficiente; es decir, cómo representar los datos.

Distintas estructuras de representación

Analicemos distintas posibles estructuras de representación para el problema.

- Para facilitar las operaciones de tipo (C1) sería interesante almacenar los registros de empleados en una lista *ListaSueldos: Lista[Empleado]*, ordenada por valor del campo Sueldo. El listado de los empleados por orden de sueldo se puede hacer fácilmente

recorriendo esta lista. Pero ahora, ¿qué pasa con las operaciones (A), (B) y (C2), de consulta por DNI, nombre o número de teléfono? Por ejemplo, para la (A2) tendríamos que recorrer toda la lista hasta encontrar un elemento cuyo valor del campo Nombre coincida con el buscado. Si tenemos n empleados, necesitaríamos un $O(n)$ en el caso promedio. Pero, es más, para la operación de consulta por número de teléfono, (A3), además de recorrer toda la lista deberíamos consultar la lista Telefonos dentro de cada empleado. El tiempo de ejecución sería un $O(n r)$, siendo r el número promedio de teléfonos por empleado. La representación es, por lo tanto, adecuada para un tipo de operación pero ineficiente para los demás.

- Para facilitar las operaciones de consulta por nombre, sería más conveniente tener una tabla de dispersión donde las claves fueran los nombres y los valores fueran de tipo Empleado, por ejemplo *HashNombre*: TablaHash[cadena, Empleado]. La consulta por nombre sería muy rápida; usando un número de cubetas $B \approx n$ podríamos alcanzar un $O(1)$ para esas operaciones. Pero, ¿qué ocurre ahora para la consulta por teléfono, por DNI o listar según el sueldo? Los teléfonos y los sueldos de los empleados están almacenados en la tabla, pero intentar acceder por ellos resultaría muy costoso. Por ejemplo, si usamos dispersión cerrada, deberíamos recorrer todas las cubetas, comprobando las que no están vacías, y recorrer sus listas de teléfonos. El orden de complejidad sería un $O(B + n r)$, con $B > n$.
- Para solucionar el problema anterior, podríamos usar una tabla de dispersión abierta *HashTelefonos*: TablaHash[entero, Empleado] donde las claves fueran números de teléfono y los valores de tipo Empleado. Usando ahora una tabla de tamaño $B \approx n$ conseguiríamos implementar la operación de tipo (A3) de forma muy eficiente, en un $O(r)$ en promedio. Pero para todas las demás operaciones deberíamos recorrer la tabla completamente. Por ejemplo, para la operación de consulta por DNI tendríamos que recorrer toda la tabla y buscar un registro con ese DNI. El tiempo de ejecución sería un $O(B + n r)$. Además, si un empleado tiene más de un teléfono, se estaría repitiendo información en la tabla.
- Por otro lado, si consideramos las operaciones (A1) y (B), de consulta por DNI, lo más conveniente sería usar una estructura de datos que nos permita acceder rápidamente por DNI, como una tabla de dispersión. El inconveniente ahora es la operación (C2) para listar ordenadamente los DNI. Ya hemos visto que las tablas de dispersión resultan muy ineficientes para este tipo de operaciones. Una buena solución sería utilizar un árbol binario de búsqueda, ArbolDNI, que dado un conjunto de n datos nos proporciona un tiempo $O(\log n)$ para la búsqueda directa y $O(n)$ para el listado ordenado (con un recorrido en in-orden). Pero, nuevamente, optimizar las operaciones que acceden por DNI implica hacer más costosas las operaciones que acceden por otros campos.

Estructura de datos combinada

Está claro que ninguna estructura por separado nos proporciona una buena solución para todos los tipos de operaciones. Cada una está orientada a optimizar cierta clase

de operaciones. La cuestión es ¿sería posible aprovechar lo mejor de cada una de ellas, consiguiendo eficiencia en todas las operaciones? La respuesta es que sí; la solución consiste básicamente en utilizar al mismo tiempo todas las estructuras antes propuestas, haciendo que referencien al mismo conjunto de datos. Es decir, tenemos un mismo conjunto de datos almacenados, pero diferentes estructuras de acceso a los mismos. Esta idea da lugar a lo que se conoce como **estructuras de datos múltiples**. En nuestro caso, la estructura sería como la mostrada en la figura 3.6.

Para no duplicar información, tanto la lista como las tablas de dispersión y el árbol deberían contener punteros o referencias a registros de tipo Empleado. De esta forma, estos datos no están repetidos, sino que cada empleado se almacena en memoria una sola vez. La definición de los tipos sería la siguiente:

tipo

Empleados = **registro**

ListaSueldos: Lista[Puntero[Empleado]]

HashNombres: TablaHash[cadena, Puntero[Empleado]]

HashTelefonos: TablaHash[entero, Puntero[Empleado]]

ArbolDNI: ArbolBinarioBusqueda[entero, Puntero[Empleado]]

finregistro

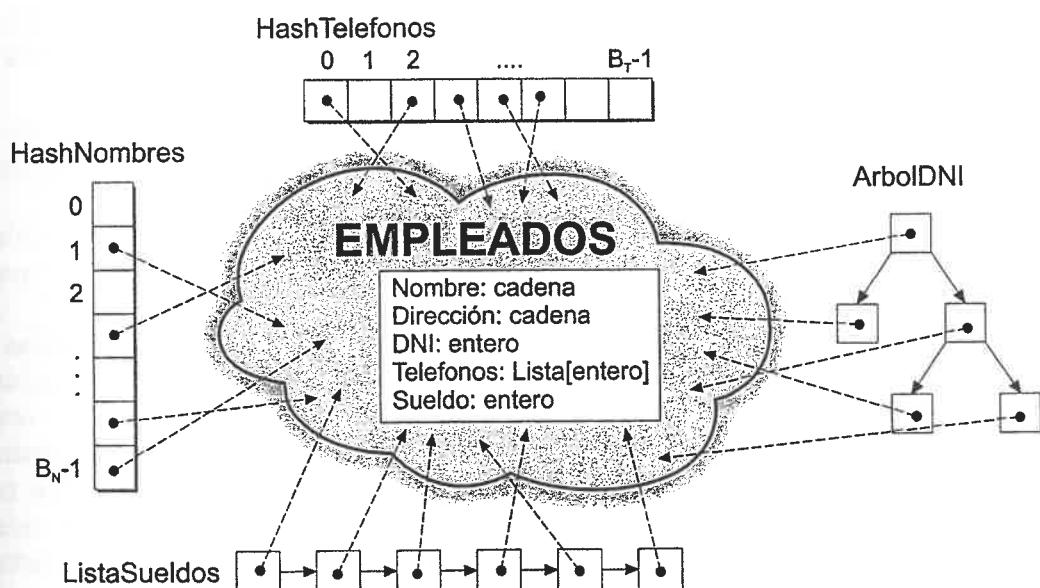


Figura 3.6: Ejemplo de estructura de datos múltiple. Varias estructuras referencian al mismo conjunto de datos almacenados.

A continuación analizamos cómo sería la implementación de las operaciones enumeradas anteriormente. Suponemos que la estructura almacena n empleados, y cada uno tiene en promedio r números de teléfono.

(A1) **Consultar por DNI.** Buscar el número de DNI en el árbol ArbolDNI, y devolver el puntero del empleado correspondiente. El tiempo de ejecución sería un $O(\log n)$.

(A2) Consultar por nombre. Buscar el nombre en la tabla HashNombres, y devolver el puntero del empleado correspondiente. El tiempo promedio sería un $O(1)$.

(A3) Consultar por teléfono. Buscar el teléfono en la tabla HashTelefonos, y devolver el puntero del empleado correspondiente. El tiempo de ejecución sería un $O(r)$, suponiendo que $B_T \approx n$.

(B1) Insertar un empleado por DNI. En este caso, la inserción implica hacer cuatro inserciones independientes en las cuatro estructuras, con lo que el tiempo viene dado por el tiempo de la más costosa. Deberíamos hacer 1 inserción en HashNombres, con un $O(1)$, r inserciones en HashTelefonos, con $O(r)$ cada una, 1 inserción en ArbolDNI, con $O(\log n)$, y 1 inserción en ListaSueldos, con $O(n)$.

(B2) Eliminar un empleado por DNI. Sería necesario eliminar en las cuatro estructuras, de forma independiente. Para ello, en primer lugar deberíamos acceder a ArbolDNI y obtener los datos del empleado a eliminar. Igual que antes, el tiempo lo determina el tiempo de la eliminación más costosa, que sería la eliminación en ListaSueldos, con un $O(n)$.

(B3) Modificar un empleado por DNI. La modificación sería equivalente a hacer una eliminación y una inserción, con la ventaja de que si un campo no se modifica, nos podemos ahorrar las operaciones correspondientes. Como en el caso anterior, lo primero sería acceder a ArbolDNI, obtener los datos del empleado y comprobar los que se modifican. Si se modifica el nombre tendríamos un $O(\log n)$, si se modifican los teléfonos un $O(\log n + r^2)$, y si es el sueldo (o todos los campos) un $O(n)$.

(C1) Listar ordenadamente por sueldo. El listado se podría hacer fácilmente recorriendo la lista ListaSueldos, y devolviendo los datos de los empleados apuntados. El tiempo de ejecución sería un $O(n)$.

(C2) Listar ordenadamente por DNI. Como ya hemos mencionado, la ordenación se puede conseguir con un recorrido en in-orden del árbol binario ArbolDNI. El tiempo de ejecución sería un $O(n)$.

En definitiva, este tipo de estructuras son lo que se conoce como **estructuras de datos duales** (cuando se combinan dos estructuras) o **múltiples** (cuando se combinan más de dos). La ventaja de este tipo de representaciones –si están bien diseñadas– es que se incrementa la eficiencia de las operaciones en las aplicaciones que requieran distintos modos de acceso. En el mejor caso, la estructura combinada ofrece lo mejor de cada estructura individual. No obstante, es necesario cuidar el diseño ya que podría ocurrir también todo lo contrario, que la estructura combinada tenga lo peor de cada estructura, resultando muy ineficiente.

En cualquier caso, la estructura múltiple implica una duplicación de información, por lo que siempre ocupará más memoria que cada estructura por separado. Además, es necesario mantener la coherencia de los datos almacenados. Como hemos visto, esto afecta particularmente a las operaciones de inserción, modificación o eliminación, que deben trabajar a la vez sobre todas las estructuras, actualizándolas de manera adecuada.

En conclusión, del concepto de estructura de datos se pueden diferenciar dos ideas distintas, aunque normalmente aparecen asociadas: la estructura de los datos almacenados y las estructuras de acceso a esos datos. Sobre un mismo conjunto de datos almacenados podemos tener distintas estructuras de acceso, cada una de las cuales ofrece un acceso adecuado a cierto tipo de operaciones.

3.3.2. La relación muchos a muchos

Ya hemos visto el interés de definir estructuras de datos duales o múltiples, en ciertas aplicaciones. Un tipo especialmente útil son las estructuras de listas múltiples, como por ejemplo las que surgen para representar **relaciones muchos a muchos**.

En el entorno de las bases de datos y los sistemas de información, al estudiar los datos manejados por un sistema se suele distinguir entre *objetos*, o *entidades*, y *relaciones* entre los anteriores. Por ejemplo, en la base de datos de la universidad podemos tener los objetos: estudiante, profesor y asignatura; y las relaciones: estudiante con asignatura, profesor con asignatura, asignatura con asignatura, etc. Las relaciones son siempre entre dos tipos de objetos. A su vez, dentro de estas relaciones distinguimos varias clases según el número de elementos que pueden estar relacionados. Una relación puede ser: uno a uno, uno a muchos o muchos a muchos. Supongamos que los tipos relacionados son *A* y *B*.

- En la **relación uno a uno**, un objeto de tipo *A* está relacionado con un y sólo un objeto de tipo *B*, y viceversa. Por ejemplo, un profesor (o un estudiante) tiene un DNI, y un DNI es exclusivo de un profesor (o un estudiante).
- En la **relación uno a muchos**, un objeto de tipo *A* puede estar relacionado con muchos objetos de tipo *B*, pero el *B* sólo está relacionado con uno de tipo *A*. Por ejemplo, la relación entre profesor y asignatura (que podemos llamar “imparte”) es una relación uno a muchos; una asignatura la imparte un sólo profesor (supongamos), pero un profesor puede impartir muchas asignaturas.
- En la **relación muchos a muchos**, un objeto *A* puede estar relacionado con muchos objetos *B*, y viceversa. Por ejemplo, las *matrículas* se pueden considerar como relaciones entre estudiantes y asignaturas. Un estudiante está matriculado en muchas asignaturas y una asignatura recibe las matrículas de muchos estudiantes. Por lo tanto, las matrículas son relaciones muchos a muchos.

En la figura 3.7 se muestra un ejemplo concreto de la relación *matrícula*, entre estudiantes y asignaturas. Nuestro interés es diseñar una estructura de datos adecuada a este problema. Buscamos una representación que nos proporcione un acceso eficiente a las matrículas, y al mismo tiempo haciendo un uso razonable de la memoria.

Estudiantes	Matrículas / Notas		Asignaturas					
	Id.	Nombre	1	Algebra	2	Física	3	AAED
1	Agapito			5				
2	Lucas			7		8		
3	Jhonny					3		4
4	Pepita					7		

Figura 3.7: Ejemplo de la relación muchos a muchos *matrícula*. Un estudiante está matriculado en muchas asignaturas y una asignatura recibe matrículas de muchos estudiantes.

Utilizando las mismas ideas básicas vistas en representación de conjuntos, llegamos a las dos soluciones directas al problema de las relaciones muchos a muchos: utilizar un

array bidimensional, o matriz, o bien una estructura de listas. En cierto sentido, ambas ideas corresponden a dos estrategias de representación opuestas: representación estática y contigua en memoria, o representación dinámica y enlazada.

Representación mediante matrices

Supongamos que de cada asignatura y estudiante almacenamos el nombre y un identificador, y para cada matrícula tenemos la nota, almacenada como un entero. La definición de los tipos sería la siguiente:

tipo

Estudiante = **registro**

 nombre: cadena

 id: entero

finregistro

Asignatura = **registro**

 nombre: cadena

 id: entero

finregistro

Nota = entero

Para simplificar, consideramos que los identificadores de los estudiantes van de 1 a n , y las asignaturas de 1 a m , de manera que se pueden almacenar ambos en sendos arrays. En caso contrario (por ejemplo, si los identificadores de los estudiantes son números de DNI), se podrían usar tablas de dispersión, utilizando los identificadores como claves.

Estudiantes: array [1..n] de Estudiante

Asignaturas: array [1..m] de Asignatura

La forma más sencilla de representar una relación muchos a muchos es mediante un simple array bidimensional de matrícululas. El tamaño de cada dimensión corresponde al número de objetos de cada uno de los dos tipos relacionados. En el caso de la relación matrícula tendremos.

tipo

Matriculas = array [1..n, 1..m] de Nota

Si M es de tipo Matriculas, cada posición $M[i, j]$ de la matriz indica la nota del alumno i en la asignatura j , para cada $i = 1..n$ y $j = 1..m$. Podemos usar el valor 0, o algún otro valor especial, en caso de no estar matriculado. En la figura 3.8 se muestra gráficamente la disposición en memoria de los datos usando esta representación.

Estudiantes		Asignaturas		Matrículas		
nombre	id	nombre	id	1	2	3
1 Agapito	1	1 Algebra	1			
2 Lucas	2	2 Física	2			
3 Jhonny	3	3 AAED	3			
4 Pepita	4			5	8	4
				7		

Figura 3.8: Representación de la relación muchos a muchos *matrícula*, usando matrices.

Las operaciones de inserción, eliminación o consulta de una matrícula, dado un identificador de asignatura y de estudiante, son inmediatas. Las tres se consiguen fácilmente en un $O(1)$. Las operaciones de listar los estudiantes matriculados en una asignatura j , o las asignaturas en las que está matriculado un estudiante i , se realizan mediante un simple recorrido de la columna o fila correspondiente. Por ejemplo, en el primer caso comprobaríamos todos los $M[s, j]$, para $s = 1..n$, listando los que tengan valor no nulo. El orden de complejidad sería claramente un $O(n)$. De forma similar, para el segundo caso el orden sería un $O(m)$.

En cuanto al uso de memoria, nos encontramos con una limitación propia de las representaciones mediante arrays: es necesario conocer el tamaño que tendrán los datos almacenados. Si no lo conocemos a priori, debemos usar un tamaño suficientemente grande para permitir que se añadan nuevos estudiantes o asignaturas. Pero si el tamaño es grande, estaremos desperdiando memoria.

Es más, puede que estemos desperdiando memoria aun conociendo el número de asignaturas y estudiantes. Está claro que si cada entero ocupa k_1 bytes, la memoria ocupada será $n \cdot m \cdot k_1$ bytes. ¿Es mucho o poco? Supongamos la Universidad de Murcia, que tiene unos 30.000 estudiantes y, digamos, unas 2.000 asignaturas. Además, como queremos tener información de la nota de cada estudiante, la convocatoria en la que está matriculado, etc., los valores del array serán registros que ocupan 4 bytes. La memoria necesaria sería: $30.000 \cdot 2.000 \cdot 4$ bytes = 240 Mbytes. Realmente no es mucho para toda la universidad, pero ¿cuál es el problema? Normalmente un estudiante no estará matriculado en más de 60 asignaturas. Si representamos sólo las asignaturas en las que un estudiante está matriculado usariamos: $30.000 \cdot 60 \cdot 4$ bytes = 7,2 Mbytes. Eso quiere decir que ¡el 97% de la matriz estará sin utilizar! De toda la memoria reservada sólo usamos el 3%. Esto es lo que se conoce como una matriz *dispersa* o *escasa*, donde la mayoría de las posiciones tienen valor nulo (o cero, en aplicaciones matemáticas).

Representación mediante listas

Para evitar el desperdicio de memoria, se puede utilizar una representación dinámica mediante listas, en las cuales sólo se almacenarán las matrículas que realmente existan. En principio, podemos elegir entre tres posibilidades distintas: una sola lista con todas las matrículas, una lista para cada estudiante con sus matrículas o una lista para cada asignatura con sus matrículas.

tipo

Mat₁ = **registro**

id_est: entero

id_asign: entero

nota: Nota

finregistro

Matriculas₁ = **Lista[Mat₁]** // Una lista con todas las matrículas

Mat₂ = **registro**

id_asign: entero

nota: Nota

finregistro

```
Matriculas2 = array [1..n] de Lista[Mat2] // Una lista por estudiante
```

```
Mat3 = registro
```

```
    id_est: entero
```

```
    nota: Nota
```

```
finregistro
```

```
Matriculas3 = array [1..m] de Lista[Mat3] // Una lista por asignatura
```

En la figura 3.9 se muestra la estructura correspondiente a estas tres posibles representaciones, para el ejemplo del apartado anterior.

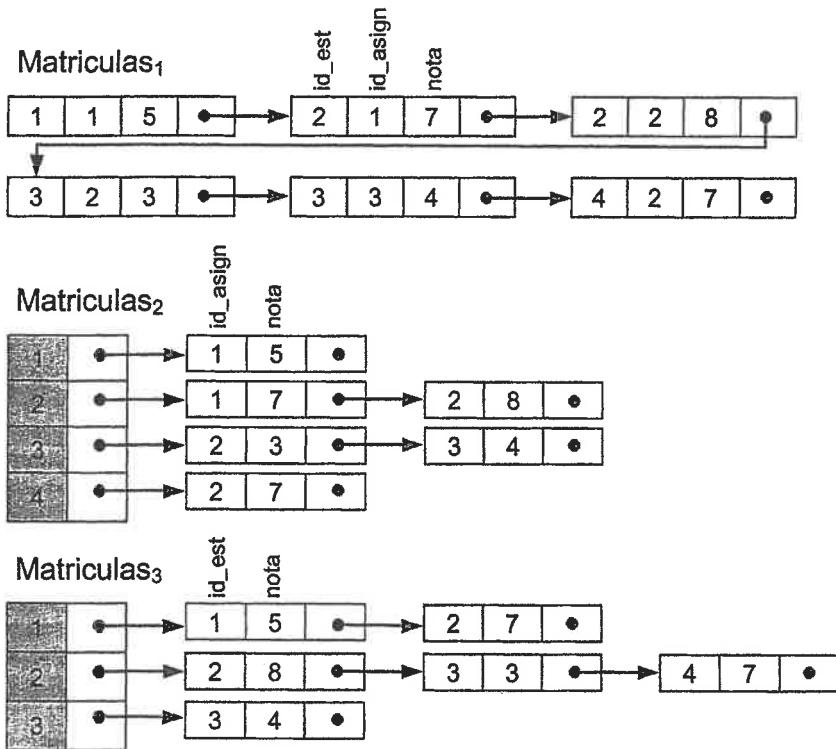


Figura 3.9: Representación de la relación muchos a muchos *matrícula*, usando listas simples. Arriba: usando una sola lista de matrículas. Centro: usando una lista de matrículas por estudiante. Abajo: usando una lista de matrículas por asignatura.

Respecto a la representación con matrices tenemos que, por un lado, se almacenan menos matrículas –únicamente las existentes–, pero por otro lado se utiliza más memoria en las listas, en los punteros y los identificadores adecuados. Si suponemos n estudiantes, m asignaturas, t matrículas, k_1 bytes por entero y k_2 bytes por puntero, en la primera estructura cada registro Mat_1 ocupa $3k_1$ bytes. Cada celda de la lista contiene, además, un puntero por lo que ocuparía $3k_1 + k_2$ bytes. En total, la lista Matriculas_1 ocuparía $t(3k_1 + k_2)$ bytes. Sólo depende del número de matrículas, no del número de estudiantes o asignaturas.

En las otras dos estructuras, los registros Mat_2 y Mat_3 ocupan $2k_1$ bytes. La memoria utilizada por Matriculas_2 y Matriculas_3 sería $t(2k_1 + k_2)$ bytes en las listas, más lo

correspondiente a los arrays: $n k_2$ y $m k_2$ bytes, respectivamente. Se puede razonar sobre la obtención de estos valores observando la representación de la figura 3.9.

Por ejemplo, considerando los valores del anterior apartado, y enteros y punteros de 4 bytes, la memoria ocupada de las tres representaciones de listas sería 28,8 Mbytes, 21,72 Mbytes y 21,6 Mbytes respectivamente, unas 10 veces menos que con matrices.

El problema es ahora el alto tiempo de ejecución de las operaciones. Con la primera posibilidad, cualquier simple operación de consulta o eliminación de una matrícula implica recorrer toda la lista, dando lugar a un $O(t)$. Con la segunda, cada lista tendrá de media t/n elementos. Consultar una matrícula concreta, dado un identificador de estudiante y de asignatura, requiere un tiempo $O(t/n)$, el mismo que para listar todas las matrículas de un estudiante. Sin embargo, si queremos consultar los alumnos matriculados en una asignatura tendremos que recorrer todas las listas, consumiendo un $O(t + n)$.

Algo parecido pasa con la tercera estructura. Consultar los estudiantes de una asignatura es rápido, pero para saber las asignaturas de un estudiante dado tenemos que recorrer todas las listas.

3.3.3. Estructuras de listas múltiples

Ya hemos visto que la utilización de listas es la única opción viable –en cuanto a uso de memoria– para la representación de matrices escasas, como las que aparecen en las relaciones muchos a muchos. Pero hemos visto también que el uso de listas implica una reducción de la eficiencia de las operaciones.

En el ejemplo de las matrículas de estudiantes en asignaturas, las operaciones necesarias son del tipo: insertar, eliminar o consultar una matrícula, listar las matrículas de un estudiante y listar las matrículas de una asignatura. Si usamos listas de matrículas para cada estudiante, la consulta de matrículas para una asignatura se hace ineficiente, y lo contrario ocurre si las listas son por asignatura. ¿Cuál es la solución? Usando la idea de las estructuras duales, podemos combinar los dos tipos de listas anteriores en una estructura de listas múltiples donde cada matrícula pertenezca a la vez a dos listas: la lista de las demás matrículas de ese estudiante y la lista de las restantes matrículas de esa asignatura.

En general, una estructura de listas múltiples es una colección de celdas en la que algunas de ellas pueden pertenecer a más de una lista a la vez. En nuestro caso tenemos:

- Los elementos de la estructura pueden ser de tres tipos distintos: estudiante, asignatura o matrícula.
- Los registros de estudiantes forman una lista y los de asignaturas otra. Además, cada registro de estudiante apunta al primer elemento de la lista de matrículas de ese estudiante, y cada asignatura apunta al primer elemento de la lista de matrículas en esa asignatura.
- Cada elemento de matrícula, que relaciona un estudiante e con una asignatura a , pertenecen al mismo tiempo a dos listas: lista de matrículas del estudiante e y lista de matrículas de la asignatura a . Estas listas son circulares, es decir la última matrícula de un estudiante (o asignatura) apunta al registro de estudiante (o asignatura) correspondiente.

Para definir el tipo de datos podemos utilizar registros con variantes. El tipo enumerado `clase_registro` indica el tipo de registro en la estructura de listas múltiples. Según esa clase, las variables de tipo `tipo_registro` tendrán unos u otros atributos.

tipo

```
clase_registro = enumerado (estudiante, asignatura, matricula)
tipo_registro = registro
    según clase: clase_registro
        estudiante: (id: entero; nombre: cadena; pri_mat, sig_est: Puntero[tipo_registro])
        asignatura: (id: entero; nombre: cadena; pri_mat, sig_asi: Puntero[tipo_registro])
        matricula: (nota: Nota; sig_est, sig_asi: Puntero[tipo_registro])
    finsegún
```

finregistro

En la figura 3.10 se muestra gráficamente la disposición en memoria de los registros en la estructura de listas múltiples.

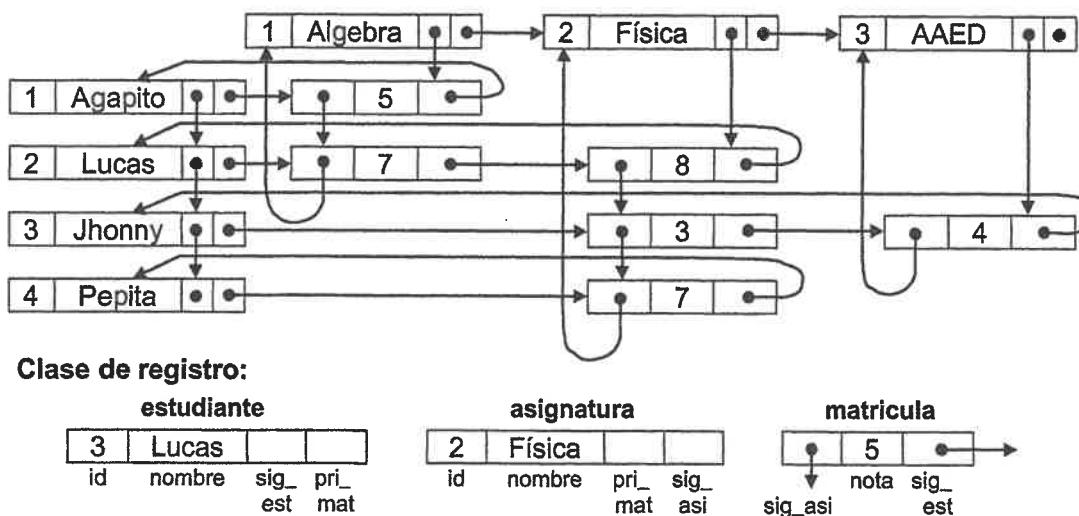


Figura 3.10: Representación de la relación muchos a muchos *matrícula*, usando listas múltiples. Se muestra abajo el significado de cada campo para cada uno de los tres tipos de registro.

Implementación de las operaciones

Supongamos que las operaciones de consulta o manipulación de matrículas reciben como parámetros punteros a los registros de estudiante o asignatura correspondientes⁵. Para buscar la matrícula asociada a un estudiante *e* en una asignatura *a* deberíamos recorrer las matrículas del estudiante *e*, y para cada una comprobar si está en la asignatura buscada. Para esto hacemos uso del hecho de que las listas son circulares.

⁵En caso contrario, se deberían recorrer las listas de estudiantes o de asignaturas; o bien podríamos usar sendas tablas de dispersión donde las claves sean identificadores o nombres, y los valores punteros a los registros correspondientes. La elección depende del tipo de las operaciones requeridas.

```
operación CalculaNota (e, a: Puntero[tipo_registro]): Nota
    m := e↑.pri_mat
    mientras m ≠ e hacer
        tmp := m↑.sig_asi
        mientras tmp↑.clase ≠ asignatura hacer
            tmp := tmp↑.sig_asi
        finmientras
        si tmp = a entonces
            devolver m↑.nota
        finsi
        m := m↑.sig_est
    finmientras
    error("Matrícula no existente")
```

Si nos fijamos en la figura 3.10, vemos que el anterior algoritmo utiliza la variable *m* para recorrer las listas horizontalmente, empezando por *e*, mientras que con *tmp* las recorreremos en sentido vertical. De forma similar, para listar los nombre de los estudiantes matriculados en una asignatura *a* deberíamos recorrer las matrículas de esa asignatura y encontrar el nombre del estudiante asociado a cada registro de matrícula.

```
operación ListarEstudiantes (a: Puntero[tipo_registro])
    m := a↑.pri_mat
    mientras m ≠ a hacer
        tmp := m↑.sig_est
        mientras tmp↑.clase ≠ estudiante hacer
            tmp := tmp↑.sig_est
        finmientras
        escribir("Nombre del estudiante: ", tmp↑.nombre)
        m := m↑.sig_asi
    finmientras
```

Evaluación de la eficiencia

En cuanto a la utilización de memoria, si consideramos sólo la utilizada en las matrículas (para comparar, en igualdad de condiciones, con las estructuras anteriores) tenemos que cada registro de matrícula contiene un entero y dos punteros, es decir $k_1 + 2k_2$ bytes. En total, necesitamos $t(k_1 + 2k_2)$ bytes para las matrículas. Para los punteros de los registros de estudiantes y asignaturas necesitaríamos adicionalmente $2k_2(n + m)$ bytes.

La utilización de memoria está dentro de los mismos valores que aparecían en las estructuras de listas simples. Para el ejemplo concreto manejado en los anteriores apartados, la memoria necesaria por los registros de matrículas sería de 21,6 Mbytes.

Por otro lado, el tiempo de ejecución de las operaciones mejora en algunos casos y empeora en otros. Por ejemplo, la operación **CalculaNota** debe recorrer una lista de matrículas horizontalmente. Para cada elemento recorre la lista correspondiente en vertical. Si tenemos *t* matrículas, entonces las listas horizontales serán, en promedio, de tamaño t/n y las verticales t/m . El tiempo de la operación **CalculaNota** en caso de no encontrarse la matrícula buscada estaría en $O(t^2/(nm))$. Si llamamos *p* = *t/(nm)*, al porcentaje de

matrículas existentes en relación al máximo posible (en el ejemplo anterior el 3%), el orden de complejidad se puede expresar como $O(t p)$.

Si nos fijamos en la operación ListarEstudiantes, su funcionamiento es básicamente el mismo, cambiando únicamente el tratamiento que se hace después del bucle interno. En este caso para una lista vertical, de tamaño promedio t/m , se recorre una lista horizontal, de tamaño promedio t/n . Nuevamente, el tiempo sería un $O(t^2/(nm)) = O(t p)$. Lo mismo tendríamos para una operación ListarAsignaturas, para mostrar las asignaturas en las que está matriculado un estudiante, o para una operación de eliminar, insertar o modificar una matrícula dada.

Si cada alumno tiene al menos una matrícula, entonces $n < t/n < t p < t$, y $O(t) = O(t + n)$; si cada asignatura tiene como mínimo un estudiante, entonces $m < t/m < t p < t$, y $O(t) = O(t + m)$. De acuerdo con esto, la estructura de listas múltiples, con un $O(t p)$, supone un término medio entre las estructuras de listas, que oscilan entre $O(t/n)$ ó $O(t/m)$ y $O(t + n)$ ó $O(t + m)$.

Es más, es posible mejorar la eficiencia de la estructura de listas múltiples, a costa de usar más memoria. Para ello, en la definición del tipo `tipo_registro` cambiamos los atributos de `matricula` para añadir también el identificador del estudiante y de la asignatura correspondientes. La definición sería:

tipo

```
tipo_registro = registro
```

```
...
```

```
matricula: (nota: Nota; id_est, id_asi: entero; sig_est, sig_asi: Puntero[tipo_registro])
```

finregistro

Con esta definición estamos añadiendo información redundante. Pero, en compensación, esa información nos puede permitir ahorrar muchos pasos de recorrido. Por ejemplo, para buscar una matrícula concreta del estudiante *e* en la asignatura *a*, sólo tendríamos que recorrer la lista horizontal del estudiante *e* y para cada matrícula comprobar el campo `id_asi`. El tiempo sería ahora $O(t/n)$, en lugar de $O(t p)$. Lo mismo pasaría con las operaciones para listar asignaturas o estudiantes.

La memoria que necesitaríamos con la versión redundante de las listas múltiples sería $t(3k_1 + 2k_2)$ bytes. En el ejemplo concreto, pasamos de 21,6 Mbytes a 36 Mbytes. No obstante, el ahorro de tiempo puede merecer la pena.

En la tabla 3.2 se muestra comparativamente la memoria y el tiempo de ejecución de algunas operaciones, para las distintas estructuras de representación de matrices escasa vistas en esta sección. Se puede concluir que las listas múltiples ofrecen unos valores de eficiencia razonables, con un uso de memoria reducido.

Ejercicios resueltos

Ejercicio 3.1 Para almacenar conjuntos de números enteros usamos tablas de dispersión. Como se espera que los conjuntos representados sean pequeños, definimos el tamaño de tabla $B = 15$. La función de dispersión es $h(x) = \lfloor x^2/10 \rfloor \bmod 15$. Mostrar las tablas de dispersión resultantes de insertar (en este orden) los elementos: 72, 23, 4, 5, 12, 25, 6, 2, 33, 24, 49, 74, usando las siguientes técnicas:

Matrices	Listas simples			Listas múltiples	
	Una lista	Por estud.	Por asign.	Normal	Redundante
Memoria (bytes)	nmk_1	$t(3k_1 + k_2)$ $+nk_2$	$t(2k_1 + k_2)$ $+mk_2$	$t(2k_1 + k_2)$ $+mk_2$	$t(k_1 + 2k_2)$ $+2k_2(n + m)$
Ejemplo (Mbytes)	240	28,8	21,72	21,61	21,87 36,26
Consultar matrícula	$O(1)$	$O(t)$	$O(t/n)$	$O(t/m)$	$O(tp)$
Ejemplo (pasos)	1	1.800.000	60	900	54.000 60
Listar asignaturas	$O(m)$	$O(t)$	$O(t/n)$	$O(t + m)$	$O(tp)$
Ejemplo (pasos)	2.000	1.800.000	60	1.820.000	54.000 60
Listar estudiantes	$O(n)$	$O(t)$	$O(t + n)$	$O(t/m)$	$O(tp)$
Ejemplo (pasos)	30.000	1.800.000	1.830.000	900	54.000 900

Tabla 3.2: Tabla comparativa de la memoria y el tiempo de ejecución, para distintas implementaciones de la relación muchos a muchos. Se muestran los valores concretos para el ejemplo desarrollado, donde n es el número de estudiantes; m el número de asignaturas; t el número de matrículas; $p = t/(nm)$ el porcentaje de matrículas en función del máximo; k_1 los bytes por entero; y k_2 los bytes por puntero.

- a) Dispersión abierta.
- b) Dispersión cerrada con redispersión lineal: $h_i(x) = (h(x) + i) \bmod 15$.
- c) Dispersión cerrada con dispersión doble, es decir: $h_i(x) = (h(x) + iC(x)) \bmod 15$, siendo $C(x) = 2(x \bmod 4 + 1)$.

Comparar la longitud media de las secuencias de búsqueda en cada caso y la memoria utilizada, suponiendo que tanto los punteros como los enteros ocupan 4 bytes.

Solución.

En primer lugar, calculamos el valor de $h(x)$ y $C(x)$ para los elementos a insertar.

x	72	23	4	5	12	25	6	2	33	24	49	74
$h(x)$	8	7	1	2	14	2	3	0	3	12	0	7
$C(x)$	2	8	2	4	2	4	6	6	4	2	4	6

Las tablas de dispersión resultantes son mostradas en la figura 3.11. En la tabla 3.3 se muestran las longitudes (total y media) de las secuencias de búsqueda y la memoria ocupada en cada uno de los tres casos.

Recordemos que la eficiencia de las tablas de dispersión está relacionada con las longitudes de las secuencias de búsqueda. En este ejemplo, la dispersión abierta sería la más eficiente, puesto que las secuencias son más cortas. Sin embargo, esta estructura necesita un 160 % más de memoria. Con dispersión cerrada y redispersión usando $C(x)$ se consigue –para este ejemplo concreto– la mejor relación entre eficiencia y uso de memoria.

Dispersión	Redispersión	Long. sec. búsqueda		Memoria (bytes)
		Total	Média	
Abierta		16	$16/12 = 1,33$	$15*4 + 12*4*2 = 156$
Cerrada	Lineal	24	$24/12 = 2,0$	$15*4 = 60$
Cerrada	Disp. doble	17	$17/12 = 1,42$	$15*4 = 60$

Tabla 3.3: Longitud de las secuencias de búsqueda y memoria ocupada en tres estructuras de dispersión, con los datos del ejercicio 3.1

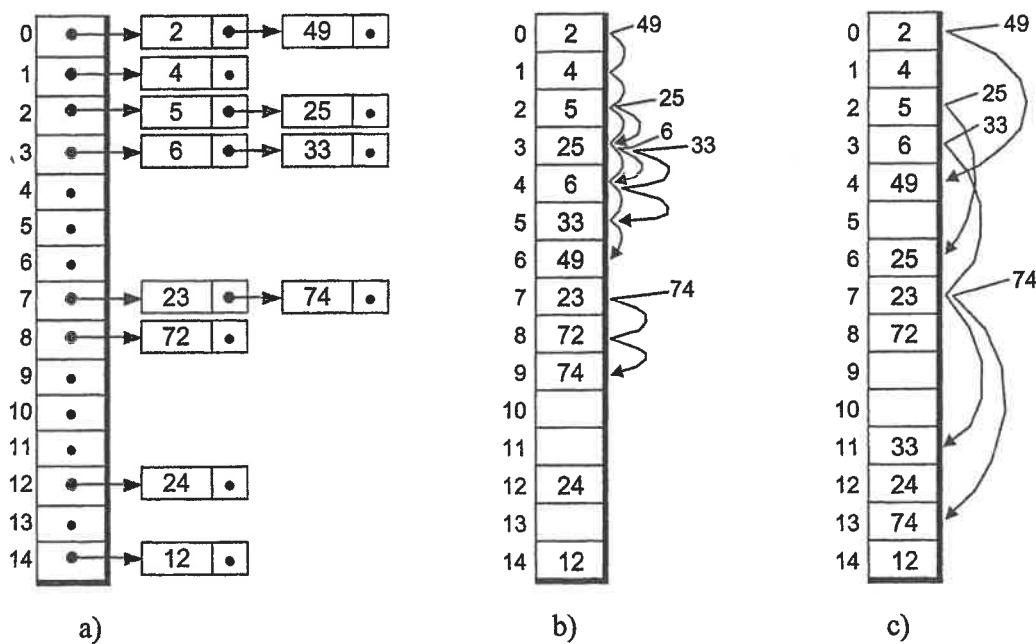


Figura 3.11: Tablas resultantes del ejercicio 3.1. a) Con dispersión abierta. b) Con dispersión cerrada y redispersión lineal. c) Con dispersión doble.

Ejercicio 3.2 La reestructuración de una tabla de dispersión consiste en sustituirla por otra con más cubetas, cuando la tabla está muy llena. Por ejemplo, suponiendo dispersión cerrada, si partimos de una tabla D_1 de tamaño B_1 , la nueva tabla es D_2 de tamaño B_2 , siendo $B_2 \approx 2B_1$. Deberíamos meter todos los elementos de D_1 en D_2 , con una nueva función de dispersión que dé valores en el intervalo $0, \dots, B_2 - 1$. Escribe un algoritmo en pseudocódigo para realizar la reestructuración de la tabla. ¿Cuál sería el orden de complejidad del algoritmo en un caso promedio?

Solución.

Supongamos que los valores almacenados son de tipo T , y las tablas D_1 y D_2 están definidas como arrays de tipo T . Además, damos por supuestas las funciones `Crear` e `Inserta`, para crear una nueva tabla de dispersión cerrada e insertar un elemento, respectivamente.

El algoritmo debe recorrer todas las cubetas de $D1$ y, si contienen un elemento, aplicar una inserción en $D2$. El algoritmo sería como el siguiente.

operación Reestructurar ($D1$: array [0.. B_1-1] de T; var $D2$: array [0.. B_2-1] de T)

```

    Crear( $D2$ )
    para  $i := 0, \dots, B_1-1$  hacer
        si  $D1[i] \neq$  NULO entonces
            Inserta( $D2, D1[i]$ )
        finsi
    finpara

```

Analizando el tiempo de ejecución, vemos que hay un bucle que se repite B_1 veces. No obstante, el tiempo de cada paso no es necesariamente constante, ya que la operación `Inserta` no siempre lo es.

En el mejor caso, en la inserción no se producirán nunca colisiones. En esta situación el tiempo de `Inserta` es un $O(1)$, por lo que el tiempo total sería un $O(B_1)$.

En el peor caso, si todos los elementos colisionan con todos los anteriores, entonces la longitud de la secuencia de búsqueda del primer elemento sería 1, la del segundo 2, y así sucesivamente. Si tenemos n elementos, la suma de las secuencias de búsqueda sería: $1 + 2 + \dots + n - 1 + n = n(n + 1)/2$. El tiempo de ejecución sería un $O(B_1 + n^2)$.

En el caso promedio, la secuencia de búsqueda del primer elemento será de longitud 1; la del segundo sería $1/(1 - 1/B_2)$; la del tercero sería $1/(1 - 2/B_2)$; y así hasta la del último, que sería $1/(1 - (n - 1)/B_2)$. Sumándolas todas tenemos:

$$\sum_{i=0}^{n-1} \frac{1}{1 - i/B_2} \quad (3.2)$$

Calcular este sumatorio de forma exacta puede ser un poco complejo. Pero podemos comprobar que todos los términos son menores que 2, y obtener una cota de forma sencilla. En primer lugar, todos los términos son menores que $1/(1 - n/B_2)$, y como $B_2 > 2n$, cada término será siempre menor que 2.

$$\sum_{i=0}^{n-1} \frac{1}{1 - i/B_2} < \sum_{i=0}^{n-1} \frac{1}{1 - n/B_2} = n \frac{1}{1 - n/B_2} < n2 \quad (3.3)$$

El tiempo total debe incluir también las operaciones del bucle **para**, que se repiten B_1 veces. El orden de complejidad sería un $O(B_1 + n)$. Además, como estamos en dispersión cerrada, se cumplirá que $n \leq B_1$. En conclusión, en el caso promedio la reestructuración tiene un orden de complejidad de $O(B_1)$, igual que en el mejor caso.

Ejercicio 3.3 En una tabla de dispersión cerrada, las claves son enteros compuestos por n bytes: $K = k_1, k_2, \dots, k_n$. Tenemos definidas dos funciones de dispersión (h^1 y h^2) y de redispersión (h_i^1 y h_i^2):

$$h^1(K) = (k_1 k_2) \bmod B \quad h_i^1(K) = (h^1(K) + i) \bmod B$$

$$h^2(K) = \lfloor \sum_{i=1}^n \pi 10^{k_i+10^i} \rfloor \bmod B \quad h_i^2(K) = (h^2(K) + k_{(i \bmod n+1)}^2 / 8) \bmod B$$

Indica comparativamente las principales ventajas e inconvenientes de cada una de las anteriores estrategias de dispersión y de redispersión.

Solución.

Vamos a comparar en primer lugar las funciones de dispersión.

- $h^1(K)$. Es una función muy fácil de aplicar. Se puede calcular en un tiempo constante, con un producto y una división. El inconveniente es que sólo depende de los dos primeros bytes de K , por lo que no resulta muy aleatoria.
- $h^2(K)$. El efecto de esta función es sumar dígitos decimales de π , según las posiciones dadas por las k_i . Además, depende de todas las partes de la clave K . Por lo tanto, resulta bastante aleatoria y, en principio, debe repartir los elementos de manera uniforme. El problema es que es muy costosa de calcular. El tiempo estaría en un $O(n)$, suponiendo que tenemos precalculados y almacenados los dígitos decimales de π .

En cuanto a las funciones de redispersión, podemos decir lo siguiente:

- $h_i^1(K)$. Es una simple redispersión lineal, con las ventajas y problemas que ello implica: es muy fácil y rápida de calcular y se garantiza que recorre todas las posiciones de la tabla. Como contrapartida, produce problemas de agrupamiento para los sinónimos.
- $h_i^2(K)$. Esta estrategia utiliza un tamaño de salto que depende de k_1 en la primera colisión, de k_2 en la segunda, y así sucesivamente. En principio, es una estrategia mejor que la redispersión lineal y podría solucionar el problema del agrupamiento. El problema es que no garantiza que se recorran todas las posiciones de la tabla. Es más, como máximo devolverá n posiciones distintas. Por lo tanto, no sería una buena estrategia de redispersión.

Ejercicio 3.4 Las tablas de dispersión, en sus distintas variantes, son una técnica muy utilizada para tener un rápido acceso directo a los datos según una clave. Pero, ¿qué ocurre si queremos acceder de forma ordenada, por ejemplo, buscar el máximo, el mínimo o recorrer todos los datos de menor a mayor? Explica cómo se podría hacer el recorrido secuencial por orden de clave en una tabla de dispersión abierta o cerrada. ¿Cuál sería el orden de complejidad? ¿Es adecuada la estructura a ese tipo de acceso?

Solución.

Está claro que las ventajas de las tablas de dispersión se pierden cuando se necesita un acceso ordenado, en lugar de un acceso directo por clave. Esto es debido a que la estructura de los elementos en la tabla no guarda ninguna relación con el orden definido entre las claves. Para conseguir la ordenación –ya sea en dispersión abierta o cerrada– habría que hacer algo como lo siguiente:

1. Recorrer todas las cubetas desde la 0 hasta la $B - 1$, extrayendo los elementos a otra estructura auxiliar, por ejemplo una lista o un array.
2. Ordenar la estructura auxiliar y listar los elementos en el orden obtenido.

El primer paso recorre toda la tabla, independientemente de que esté completamente vacía o llena. El tiempo de ejecución sería $O(B + n)$. El segundo paso, la ordenación propiamente dicha, se puede conseguir en un $O(n \log n)$, por ejemplo con ordenación por mezcla. En total, el orden de complejidad sería $O(B + n \log n)$. Como veremos en el siguiente capítulo, cuando se trata de optimizar al mismo tiempo las operaciones de acceso directo y ordenado, resultan más adecuadas las estructuras arbóreas.

Ejercicio 3.5 Para solucionar el problema de los sinónimos en las tablas de dispersión, definimos la siguiente estrategia. Aparte de la tabla de dispersión, tenemos una estructura de lista enlazada llamada “lista de desbordamiento”. Cuando se produce una colisión, el elemento que colisiona es añadido directamente a la lista de desbordamiento en la última posición (después de comprobar que no se encuentra ya en la misma).

Escribe un algoritmo en pseudocódigo para eliminar un elemento x de la estructura de dispersión anterior, sin usar marcas SUPRIMIDO. Supón que la tabla de dispersión es T , la lista de desbordamiento L y la función de dispersión h ; por lo tanto, la llamada tendrá la forma: Eliminar(x, T, L).

Solución.

En primer lugar, debemos buscar la clave x en el lugar que le corresponde en la tabla, es decir en $h(x)$. Si se encuentra, entonces se puede eliminar. Pero, para no romper ninguna secuencia de búsqueda, tenemos que buscar si en la lista de desbordamiento hay algún sinónimo de x , en cuyo caso pasará a la posición eliminada.

Si x no está en $T[h(x)]$, puede que esté en la lista de desbordamiento, aunque sólo si esa posición de la tabla no está vacía. Si la posición de la tabla está vacía, podemos asegurar que x no está en la lista de desbordamiento, ya que la operación de inserción lo habría colocado antes en la tabla.

El algoritmo en pseudocódigo podría ser como el siguiente.

```

operación Eliminar (x: clave; var T: array [0..B-1] de clave; var L: Lista[clave])
    p := h(x)
    si T[p] = x entonces
        Buscar en L si hay algún y tal que h(y) = h(x)
        si existe el elemento y anterior entonces
            Eliminar y de L
            T[p] := y
        sino
            T[p] := NULO
        finsi
    sino si T[p] ≠ NULO entonces
        Buscar x en L
        si se encuentra entonces
            Eliminar x de L
        finsi
    finsi

```

Ejercicio 3.6 En cierta aplicación decidimos usar tablas de dispersión para representar los números de teléfono de una agenda personal, como por ejemplo {968121314, 968236677, 968990012, 606172614, 968367722}. Tenemos disponible una librería que implementa la

dispersión abierta y la cerrada, y que permite definir todos los parámetros a usar (tipo de dispersión, tamaño de la tabla, tipo de datos almacenado en la tabla, función de redispersión, etc.), excepto la función de dispersión que es fija y vale: $h(x) = (x \text{ div } 10^6) \text{ mod } B$. Elige los parámetros configurables de la implementación más adecuados para este caso, de manera que se eviten los problemas de la función de dispersión (que, para esta aplicación, es previsible que produzca muchos sinónimos).

Solución.

Si utilizáramos dispersión abierta, todos los sinónimos irían a parar a la misma cubeta o a unas pocas cubetas. El efecto sería equivalente a trabajar con listas simples, con lo que si tenemos n elementos las operaciones tardarían un $O(n)$ en promedio.

El problema se puede resolver usando dispersión cerrada y eligiendo una función de redispersión tal que dos elementos con el mismo valor de dispersión tengan distintos valores de redispersión. Por ejemplo, la función de redispersión podría ser $h_i(x) = (x - i) \text{ mod } B$. El tamaño de la tabla, B , debería ser mayor que el número de elementos almacenados, por ejemplo $B \geq 1,5n$. Para el ejemplo concreto, si tomamos $B = 8$, tenemos la tabla siguiente:

0	1	2	3	4	5	6	7
968121314 $h(\cdot) = 0$	968367722 $h(\cdot) = 0$		968990012 $h(\cdot) = 0$	968236677 $h(\cdot) = 0$	606172614 $h(\cdot) = 6$		
		$h_1(\cdot) = 2$		$h_1(\cdot) = 4$	$h_1(\cdot) = 5$		

El resultado es que aunque la función de dispersión sea muy mala, con la función de redispersión conseguimos arreglar en parte el problema. En este ejemplo concreto, sólo se produce una colisión para los sinónimos.

Hay que resaltar el hecho de que aplicando redispersión lineal ($h_i(x) = (h(x) + i) \text{ mod } B$) no se solucionaría el problema, y tampoco usando cualquier otra función de redispersión que produzca las mismas secuencias de búsqueda para los sinónimos. También hay que recordar que la función de redispersión $h_i(x)$ no sólo depende de x , sino también de i , el número de colisiones para esa clave.

Ejercicios propuestos

Ejercicio 3.7 Sean $A = \{1, 2, 3\}$ y $B = \{3, 4, 5\}$, mostrar la ejecución y los resultados de las siguientes operaciones, suponiendo una representación con vectores de bits y con listas enlazadas (ordenadas y no ordenadas).

- a) Union(A, B, R)
- b) Intersección(A, B, R)
- c) Diferencia(A, B, R)
- d) Miembro($1, A$)
- e) Inserta($1, A$)
- f) Suprime($1, A$)
- g) Min(A)

Ejercicio 3.8 La realización de conjuntos mediante vectores de bits se puede usar siempre que el conjunto universal se pueda “traducir” a los enteros de 1 a N . Describir cómo se

haría esta traducción (en caso de ser posible) y cuánto valdría N , si el conjunto universal fuera:

- a) los enteros de n a m , para cualquier $n \leq m$
- b) los enteros $n, n+2, n+4, \dots, n+2k$, para cualesquiera n y k
- c) los cuadrados perfectos $1, 4, 9, 16, 25, 36, 49, \dots$
- d) los caracteres 'a', 'b', ..., 'z'
- e) arrays de dos caracteres, cada uno de ellos entre 'a' y 'z'
- f) los números reales de 0 a 10
- g) las cadenas de hasta 20 caracteres

Ejercicio 3.9 Implementa en pseudocódigo las operaciones sobre conjuntos $\text{EsSubconjunto}(A, B)$, $\text{Cardinalidad}(A)$, $\text{Máximo}(A)$ y $\text{Mínimo}(A)$. Utiliza las representaciones mediante arrays de booleanos, listas de elementos y tablas de dispersión. Hacer una estimación del tiempo de ejecución necesario para cada operación, según el tipo de representación.

Ejercicio 3.10 Escribe en pseudocódigo la implementación de las operaciones `Inserta`, `Suprime` y `Consulta`, para la representación del tipo `Diccionario[tclave, tvalor]` del apartado 3.2.2, suponiendo B posiciones de dispersión y una función $h(x)$ predefinida.

Ejercicio 3.11 Supón que estamos insertando enteros en una tabla de dispersión de $B=17$ cubetas, utilizando la función de dispersión $h(x) = \lfloor x^2/13 \rfloor \bmod 17$.

- a) Mostrar la tabla de dispersión abierta si se insertan los valores 23, 44, 82, 11, 1, 8, 55, 27, 125, 216, 343, 88.
- b) Repetir el anterior apartado usando una tabla de dispersión cerrada con resolución lineal de colisiones.
- c) Repetir usando una tabla de dispersión cerrada de tamaño 33, con $h(x) = \lfloor x^2/13 \rfloor \bmod 33$ y resolución lineal de colisiones.
- c) ¿En cuál de los casos anteriores se deben realizar menos operaciones? ¿En cuál se utiliza menos memoria? Discute qué representación conseguiría una mejor relación tiempo/memoria.

Ejercicio 3.12 En una aplicación que usa dispersión cerrada tenemos $B = 20$ cubetas, y una función de dispersión con la forma $h(x) = x \bmod 20$. Además, en esta aplicación se conocen a priori los elementos que se van a almacenar, que son: 72, 51, 132, 32, 631, 391, 211. ¿Por qué la función de dispersión elegida no es buena para esta aplicación concreta? Define una buena función de redispersión para este caso, que evite el problema que aparece con redispersión lineal.

Ejercicio 3.13 Un sistema operativo necesita controlar la memoria libre y la que está siendo usada por algún proceso. La memoria se puede considerar como un conjunto de posiciones, dentro de cierto rango. ¿Cómo sería la implementación mediante vectores de bits y mediante listas enlazadas? Discute la implementación de las operaciones: obtener cantidad total de memoria libre, pedir una cierta cantidad de memoria y liberar un bloque de memoria. ¿Cuál sería su complejidad en cada una de las implementaciones?

Ejercicio 3.14 Considerar el método de hashing en una tabla cerrada con B cubetas enumeradas de 0 a $B - 1$, siendo las claves x enteros en el intervalo $0, \dots, 3B - 1$. La función de dispersión es $h(x) = x \bmod B$, y las colisiones se resuelven con la función de redispersión $h_i(x) = (h(x) + i h'(x)) \bmod B$ (intervalos constantes, dependientes del valor de x). Define una función $h'(x)$ de modo que claves distintas con igual valor de $h(x)$ tengan secuencias de búsqueda distintas (valores de h' distintos). ¿En qué casos se recorre toda la tabla con la función obtenida?

Ejercicio 3.15 Supón una tabla de dispersión con 20 cubetas, donde las claves están en el intervalo 0..80.000. Definir una estrategia de dispersión que use la técnica de plegado (*o folding*) para este caso. Aplicarla a la introducción de los siguientes elementos: 40734, 71263, 01371, 41, 28497, 10101, 02747, 10230, 1, 12890, 50902, 0, 65126, 54879, 3, 11111, 21. Utiliza una tabla de dispersión abierta y una tabla de dispersión cerrada, con la función de redispersión que creas más conveniente.

Comenta brevemente la bondad de la función diseñada para este ejemplo, comparando las longitudes de las secuencias de búsqueda obtenidas en los dos tipos de dispersión.

Ejercicio 3.16 Un anagrama es una palabra que se obtiene permutando las letras de otra palabra. Por ejemplo, las palabras “cerdo”, “credo” y “cedro” son anagramas. Supongamos que tenemos un conjunto formado por todas las palabras reales de cinco letras. Queremos organizarlas en grupos, de forma que cada grupo contenga todas aquellas palabras formadas por permutaciones de las mismas cinco letras; es decir, cada palabra de un grupo será un anagrama del resto de palabras de su grupo. Para conseguirlo vamos a emplear una tabla de dispersión. Describir el problema, la función de dispersión que se debe utilizar, la estructura de la tabla empleada, el método de resolución de colisiones y escribir un algoritmo para resolver este problema. ¿Cómo podríamos aplicar el algoritmo propuesto a todas las palabras del diccionario de la R.A.E.L.?

Ejercicio 3.17 Considerar una tabla de dispersión cerrada con $B = 10$ cubetas, en la que queremos insertar los elementos: 422, 12, 72, 392, 763, 842, 652, 723. Define una función de dispersión y una estrategia de redispersión adecuadas para este caso, que evite el problema del agrupamiento. Mostrar la tabla cerrada resultante tras la inserción de los elementos.

Ejercicio 3.18 En una tabla de dispersión cerrada con $B = 10$ y la función de dispersión: $h(x) = x \bmod B$ insertamos los siguientes elementos, en el orden dado: 102, 59, 89, 90, 12, 2, 45, 13 y 24. Para resolver las colisiones se utiliza redispersión lineal. Mostrar la estructura de la tabla.

Después de las inserciones, se eliminan los elementos 12 y 59. Mostrar la ejecución de las operaciones de eliminación, usando la implementación con marcas SUPRIMIDO y la implementación sin marcas SUPRIMIDO. ¿Cuál requiere menos tiempo?

Ejercicio 3.19 Utilizando la estructura de listas múltiples para la representación de matrículas, vista en el apartado 3.3.3, resolver las siguientes cuestiones.

- Implementar una operación `MediaEstudiante(e)` para calcular la nota media del estudiante e en las asignaturas en las que está matriculado.

- b) Implementar una operación `MediaAsignatura(a)` para calcular la nota media obtenida por los estudiantes de la asignatura a .
- c) Implementar una operación para encontrar el estudiante que tenga más matrículas en las distintas asignaturas.
- d) Hacer una estimación del orden de complejidad de las operaciones anteriores, suponiendo los parámetros de la tabla 3.2.

Ejercicio 3.20 En una aplicación matemática se utilizan matrices *escasas* de tamaño grande, por ejemplo de tamaño 1.000×1.000 , pero sólo el 2% de ellas son distintas de 0. Para representar el tipo `MatrizEscasa` utilizamos una estructura de listas múltiples, similar a la estudiada en el apartado 3.3.3. Definir los tipos adecuados para implementar matrices escasas. Usando esa definición, implementar las operaciones para sumar dos matrices y para multiplicarlas. Estudia el uso de memoria y el orden de complejidad de las operaciones, suponiendo que las matrices son de tamaño $n \times n$ y el $p\%$ de las posiciones son distintas de 0.

Ejercicio 3.21 Un fichero de *logs* es un fichero de texto donde se almacenan fechas de acceso a un servidor web. Un analizador lee ficheros de *logs* y obtiene estadísticas sobre los tiempos de uso. Una parte de los ficheros de texto es el nombre del mes ("Enero", "Febrero", "Marzo", etc.), que se lee como una cadena c . El objetivo es encontrar la forma más rápida posible de convertir la cadena c en el número de mes correspondiente.

Una solución directa podría ser comparar c con los nombres de los 12 meses, hasta encontrar una equivalencia; pero, esta solución sería excesivamente lenta. Resolver el problema utilizando tablas de dispersión. Hacer una estimación del tiempo de ejecución de la operación de conversión. Sugerencia: Definir una función de dispersión adecuada, que no provoque sinónimos y se pueda calcular rápidamente (por ejemplo, extrayendo el primer y el tercer carácter). Definir una tabla con tamaño reducido, donde en cada posición i se almacene el número del mes cuyo valor de dispersión sea i .

Cuestiones de autoevaluación

Ejercicio 3.22 Un compilador necesita almacenar información de las variables del programa que está compilando. Para cada una de ellas se almacenarán una serie de datos (tipo de la variable, nombre, línea donde está declarada, etc.). El compilador sólo necesita funciones para guardar información de una variable y recuperar información de una variable. ¿A qué tipo abstracto de datos corresponde esta necesidad? Si disponemos de bastante memoria, ¿qué estructura de representación será más eficiente en cuanto a tiempo?, ¿por qué?

Ejercicio 3.23 ¿Puede tener sentido, en algún caso, utilizar una tabla de dispersión con un número de cubetas B mucho menor que el tamaño de los conjuntos usados, por ejemplo 10 veces menor? ¿En qué situación, y qué beneficios se obtendrían respecto a una representación con listas ordenadas o no ordenadas?

Ejercicio 3.24 En una tabla de dispersión abierta hacemos que las listas almacenadas en cada cubeta sean listas ordenadas. ¿Qué mejora se obtendría para las operaciones sobre el tipo de datos? ¿En qué circunstancias la mejora sería más notable?

Ejercicio 3.25 ¿Cuál es el problema de la redispersión lineal? ¿Se soluciona el problema con redispersión cuadrática? ¿Y utilizando una permutación fijada de antemano? Indica las condiciones que deben cumplirse para que una estrategia de redispersión solucione el problema.

Ejercicio 3.26 En una tabla de dispersión, reservamos de forma fija espacio en cada cubeta para q elementos. Esto puede verse como una estrategia mixta entre la dispersión cerrada, donde $q = 1$, y la abierta, donde q no está limitado. Con esta estructura, ¿se soluciona el problema de la redispersión? Señala las principales ventajas e inconvenientes de esta estrategia sobre la dispersión abierta y cerrada.

Ejercicio 3.27 En una aplicación donde queremos usar dispersión, disponemos de relativamente poca memoria. En este caso, ¿qué estrategia de dispersión es más adecuada, suponiendo que se conoce a priori el tamaño de los conjuntos que se van a usar? ¿Por qué? ¿Y si no se conoce el tamaño a priori?

Referencias bibliográficas

La notación y terminología de conjuntos y diccionarios se pueden encontrar en muchos libros de estructuras de datos, si bien no suelen aparecer de forma explícita. Las tablas de dispersión son una técnica de representación muy ampliamente utilizada y, por lo tanto, aparecen en muchos libros. El nivel de detalle con el que se estudian aquí es similar al tratado en [Aho88] en el capítulo 4, o al de [Weis95], en el capítulo 5. En esta última referencia se puede encontrar otra técnica de redispersión, conocida como *dispersión extensible*, que se suele usar en aplicaciones de bases de datos.

Para profundizar en el estudio de las tablas de dispersión se puede consultar el capítulo 11 de [Cormen90], o el capítulo 10 de [Drozdek01], entre otros.

Las ideas de las estructuras de datos duales y las listas múltiples están sacadas de [Aho88], y son analizadas en el apartado 4.12.

Capítulo 4

Representación de conjuntos mediante árboles

La representación eficiente de conjuntos y diccionarios requiere un estudio de las características particulares de cada aplicación. Cuando se necesitan operaciones de búsqueda y consulta secuencial, de acuerdo con cierto orden, las estructuras de árboles son una buena elección. Los árboles son básicamente estructuras jerárquicas, con nodos que actúan de padres, hijos, hojas y un nodo destacado que juega el papel de raíz. No obstante, conseguir eficiencia en las estructuras arbóreas requiere introducir alguna restricción que garantice que los árboles quedan equilibrados, puesto que en el peor caso el árbol podría tomar la forma de lista.

Objetivos del capítulo:

- Conocer y comprender una variedad de técnicas eficientes de representación de conjuntos y diccionarios mediante estructuras arbóreas.
- Adquirir la capacidad de evaluar las necesidades de representación de una aplicación específica, tomando decisiones justificadas sobre las estructuras de representación más adecuadas.
- Ser capaz de diseñar e implementar las estructuras estudiadas, realizando las adaptaciones que cierta aplicación específica requiera.
- Comprender la necesidad de usar mecanismos de equilibrado o balanceo para conseguir eficiencia en las representaciones arbóreas.

Contenido del capítulo:

4.1.	Árboles trie	129
4.1.1.	Árboles de prefijos	130
4.1.2.	Representación de nodos trie	132
4.1.3.	Búsqueda e inserción de palabras en un trie	136
4.1.4.	Eficiencia y comparación entre estructuras	138
4.2.	Relaciones de equivalencia	142
4.2.1.	Representaciones sencillas de relaciones de equivalencia	145
4.2.2.	Representación mediante punteros al padre	147
4.2.3.	Equilibrado y compresión de caminos	149
4.3.	Árboles de búsqueda balanceados	154
4.3.1.	Árboles binarios de búsqueda	154
4.3.2.	El peor caso de árbol AVL	157
4.3.3.	Rotaciones simples y dobles sobre AVL	160
4.3.4.	Inserción en un árbol AVL	162
4.3.5.	Eliminación en un árbol AVL	165
4.4.	Árboles B	168
4.4.1.	Árboles de búsqueda no binarios	168
4.4.2.	Inserción en un árbol B	172
4.4.3.	Eliminación en un árbol B	172
4.4.4.	Ánalisis de eficiencia de los árboles B	174
	Ejercicios resueltos	177
	Ejercicios propuestos	184
	Cuestiones de autoevaluación	187
	Referencias bibliográficas	189

4.1. Árboles trie

Cuando hablamos de eficiencia –y particularmente en la implementación de un tipo de datos– nos referimos tanto al uso de memoria como al tiempo de ejecución de las operaciones. En general, las estructuras de dispersión consiguen una alta eficiencia para las operaciones de acceso inmediato según cierta clave, con un uso razonable de memoria. Pero, ¿qué ocurre con las operaciones que necesitan un acceso ordenado? La representación es muy poco adecuada para esos casos. En tales situaciones resulta mejor utilizar estructuras de listas. Sin embargo, como contrapartida, son las operaciones de acceso directo las que se vuelven ineficientes usando listas. En última instancia, el problema es la utilización de estructuras *lineales*, ya sean tablas o listas.

Las representaciones arbóreas son una buena alternativa, que ofrece normalmente interesantes relaciones entre uso de memoria y tiempo de las operaciones de acceso directo y secuencial. Un árbol es básicamente una estructura jerárquica. En la figura 4.1 se muestran dos ejemplos de árboles usados para representar distinto tipo de información. Vamos a hacer un breve repaso de algunos conceptos fundamentales sobre árboles; se da por hecho que el lector está más o menos familiarizado con la terminología de árboles.

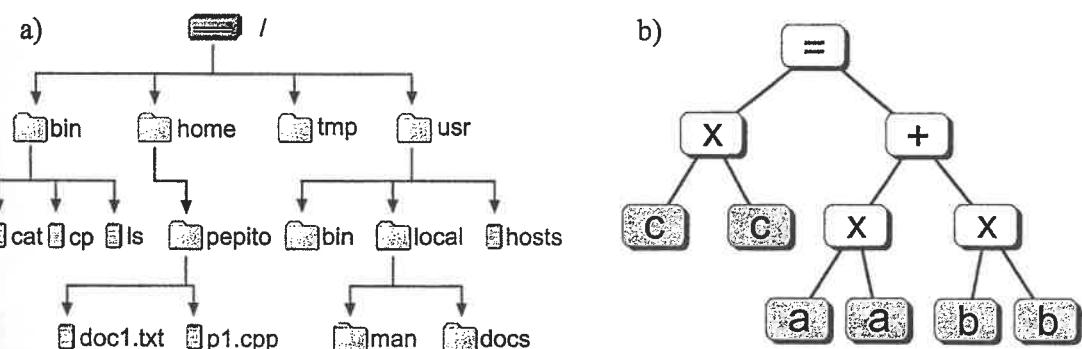


Figura 4.1: Dos ejemplos de árboles en aplicaciones distintas. a) Representación de un árbol de directorios. b) Representación de la expresión aritmética: $c^2 = a^2 + b^2$.

Un árbol contiene un conjunto de elementos o **nodos** conectados a través de una relación no simétrica: la relación **padre/hijo** (obviamente, no es lo mismo ser padre de alguien que ser su hijo). Existe un nodo especial dentro de un árbol, llamado la **raíz**, que se distingue por no tener ningún parente. Un árbol tiene una y solo una raíz. Los nodos que no tienen hijos –de estos puede haber varios– se llaman **hojas**. Los nodos no hoja se llaman también **nodos internos**. Los **descendientes** de un nodo son sus hijos y los descendientes de sus hijos. De forma similar, los **antecesores** son el parente del nodo y los antecesores de su parente. Los **hermanos** de un nodo son los hijos de su parente. El **nivel** de un nodo dentro del árbol es el **número de antecesores** que tiene. La **profundidad** o **altura** de un árbol es el **máximo valor de nivel de sus nodos**. Por ejemplo, los dos árboles de la figura 4.1 son de altura 3.

Existen muchas variedades de árboles y formas de representarlos. Según las operaciones definidas y su comportamiento, tendremos un tipo abstracto de árbol particular.

Por ejemplo, podemos definir el tipo abstracto parametrizado `ArbolBinario[T]` de árboles binarios de cierto tipo genérico `T`.

No obstante, en este capítulo no estamos interesados en el TAD árbol en sí mismo, sino en la implementación de los tipos abstractos conjunto y diccionario, utilizando estructuras arbóreas. ¿Cuál es la diferencia? Nuestro objetivo último será implementar las operaciones `Inserta`, `Suprime`, `Consulta`, etc., definidas sobre conjuntos y diccionarios, con el mismo significado y comportamiento que en el capítulo anterior. De forma abstracta, es decir, de cara al usuario, es independiente que se implementen con árboles, con listas o con tablas de dispersión¹.

4.1.1. Árboles de prefijos

Muchas estructuras de datos –al igual que otros avances científicos– surgen del desarrollo de una solución particular para cierto problema específico. Después, aplicando un paso de generalización, extendemos la idea para incluir una variedad más amplia de problemas. Este es el caso de los árboles `trie`², creados para ser utilizados en la representación de diccionarios de palabras en los correctores ortográficos. Analicemos las características particulares de este tipo de aplicaciones.

Un editor de texto interactivo

Consideremos un editor de texto, con comprobación interactiva de la sintaxis. El editor comprueba la ortografía a medida que el usuario escribe palabras. Cada vez que termina una palabra, el programa comprueba si existe esa palabra o se ha cometido un error. Si hay un error, el usuario puede recibir alguna sugerencia o añadir la palabra al diccionario, si piensa que es correcta.

El problema es, en esencia, una aplicación del tipo conjunto: buscar palabra en un conjunto, añadir palabra, borrar palabra, etc. Un lenguaje como el español contiene unas 80.000 palabras, sin incluir plurales, conjugaciones o nombres propios. Si los incluimos, podemos llegar a tener en torno a 3 millones de palabras. Y, nuevamente, el requisito de eficiencia es ineludible; la búsqueda en el diccionario no puede tardar más de unos pocos milisegundos. Además, se debe reducir al máximo el uso de memoria, pues seguro que el usuario querrá hacer otras cosas al mismo tiempo.

¿Cuál es la característica particular y definitoria de esta aplicación? Las palabras no aparecen de forma aleatoria, sino que existen relaciones entre ellas. Muchas palabras comparten prefijos o sufijos comunes, que se repiten una y otra vez a lo largo del diccionario. Consideremos, por ejemplo, las palabras que aparecen en el *Diccionario de la Real Academia Española*, a partir de la palabra “esparto”.

esparto esparvar esparvel esparver espasmar espasmo espasmódica espata
espatarrada espatarrarse espática espático espato espátula espatulomancia espatulomancia
espaviento espavorecida espavorecido espavorida espavorido espay especería especia especial

¹De nuevo, hacemos hincapié en la diferencia entre TAD y estructura de datos: una estructura de árboles se puede usar para implementar un TAD conjunto.

²El nombre proviene del inglés *retrieve* (recuperar), es similar a *tree* (árbol), pero pronunciado /trai/.

especialidad especialista especialización especializar especialmente especie especiera
especiería especiero específica especificación especificadamente ...

Todas ellas comparten las tres primeras letras, "esp", y la mayoría comparten muchas más. Por ejemplo, una amplia familia surge del prefijo "especial": especial -idad, -ista, -ización, -izar, -mente. Esta representación mediante guiones nos sugiere una forma de representar las palabras: factorizar los prefijos comunes y después ramificar hacia las distintas formas en las que puede seguir una palabra. Aplicando la idea sobre algunas de las anteriores palabras, obtendríamos la representación mostrada en la figura 4.2.

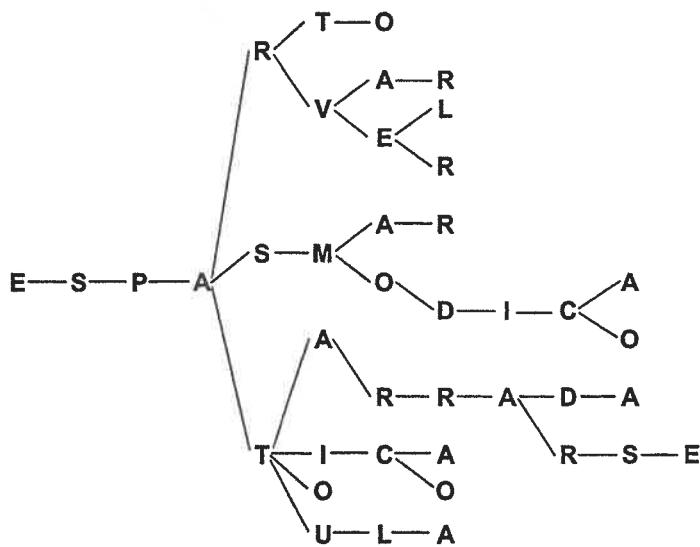


Figura 4.2: Árbol de palabras del *Diccionario de la Real Academia Española*, desde la palabra "esparto" hasta "espatula" (por el momento, obviamos las tildes).

De forma natural, hemos llegado a una representación arbórea: existe una raíz, nodos descendientes de la misma, y distintas ramas por las que se bifurcan las distintas palabras del diccionario a partir de un prefijo común. Esto es básicamente un árbol trie.

Definición de árbol trie

Una vez con la idea de la estructura de **árbol de prefijos**, vamos a analizar su implementación. El objetivo es definir los tipos y algoritmos necesarios para representar árboles como el mostrado en la figura 4.2. Debemos tener en cuenta algunas cuestiones.

- Los árboles almacenados son árboles n -arios, es decir, un nodo puede tener cero, uno o varios hijos. En principio, no se puede predecir cuántos hijos tendrá un nodo. Como máximo podrá tener tantos como letras del alfabeto, aunque lo normal es que tenga muchos menos.
- Las palabras del diccionario son las hojas del árbol. Sin embargo, también pueden existir palabras que no sean hojas sino nodos intermedios. Por ejemplo, esto ocurre

con la palabra “espasmo”. ¿Cómo distinguimos que “espasmo” es una palabra pero “espatar” no? Para solucionarlo, añadiremos al alfabeto una marca de fin \$, que indicará una palabra completa.

- Realmente las palabras no están almacenadas en los nodos, sino que se forman a través de los caminos de un nodo hasta la raíz, es decir los antecesores de ese nodo.
- La idea de los árboles de prefijos se puede extender a otros tipos de datos. Por ejemplo, en una aplicación de listín telefónico el alfabeto estaría formado por los dígitos decimales.

En definitiva, podemos definir los árboles trie de la siguiente manera.

Definición 4.1 Dado un alfabeto cualquiera (a partir del cual se construye un conjunto de palabras o expresiones), un **árbol trie** es una estructura de árbol en la que:

- La raíz del árbol representa la cadena vacía.
- Un nodo puede tener tantos hijos como caracteres del alfabeto más uno. Cada hijo está etiquetado con un carácter o una marca de fin \$.
- La sucesión de etiquetas desde la raíz hasta un nodo hoja, etiquetado con la marca de fin \$, representa una palabra.
- A todos los nodos, excepto a la raíz y a las hojas etiquetadas con \$, se les denomina **prefijos** del árbol.

En la figura 4.3 se muestra un ejemplo de representación de árbol trie, usando la anterior definición. Se puede apreciar que hemos hecho una sutil modificación respecto a la idea mostrada en la figura 4.2. Mientras que antes poníamos una etiqueta en los nodos, ahora la etiqueta está asociada a la relación de un nodo hacia sus descendientes.

4.1.2. Representación de nodos trie

Abordando ahora la representación en memoria de árboles trie, vemos que la cuestión fundamental es decidir cómo representar los nodos del árbol. Un nodo de árbol trie debe contener un conjunto (de tamaño variable) de punteros a hijos, cada uno asociado a un carácter del alfabeto o \$. Esto es, precisamente, lo que conocemos como un diccionario, donde los caracteres son las claves y los valores son punteros a nodos trie. Por lo tanto, podemos hacer las siguientes equivalencias. Los tipos están parametrizados en función de A, de tipo Alfabeto, que indica el tipo de los elementos que forman las palabras. Como ejemplo, una posible instanciación sería: `NodoTrie[A: Alfabeto]`.

tipo

$$\begin{aligned} \text{NodoTrie}[A: \text{Alfabeto}] &= \text{Diccionario}[A, \text{Puntero}[\text{NodoTrie}[A]]] \\ \text{ArbolTrie}[A: \text{Alfabeto}] &= \text{Puntero}[\text{NodoTrie}[A]] \end{aligned}$$

En consecuencia, la representación de nodos trie podría utilizar cualquiera de las estructuras de diccionarios estudiadas en este capítulo o en el anterior. También es posible

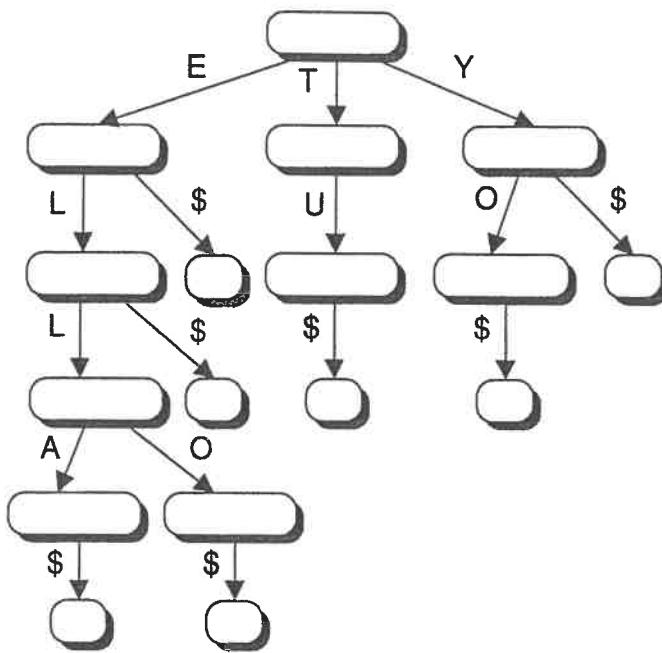


Figura 4.3: Árbol trie compuesto por el conjunto de palabras: {YO, TU, EL, ELLA, ELLLO, Y, E}.

definir los *NodoTrie* como un TAD, con las operaciones en concreto que debe ofrecer una implementación del tipo. La definición podría ser como la siguiente.

TAD NodoTrie[A: Alfabeto] es Crear, Asigna, ValorDe, TomaNuevo

Requiere

El tipo *A* es un alfabeto que debe tener definida una operación de comparación *Igual* (*ent A, A; sal boolean*), y que incluye un carácter especial *\$* que se interpreta como final de palabra.

Descripción

Los valores de tipo *NodoTrie[A]* representan nodos de un árbol trie. Cada nodo almacena un diccionario con representación mutable, donde las claves son caracteres y los valores son punteros a *NodoTrie[A]*. La asignación de un carácter se hace con la operación *Asigna* o *TomaNuevo* y la consulta con *ValorDe*.

Operaciones

Operación Crear (sal Puntero[NodoTrie[A]])

Calcula: Devuelve un puntero a un nodo trie nuevo, que no contiene asociaciones.

Operación Asigna (ent n: NodoTrie[A]; c: A; v: Puntero[NodoTrie[A]])

Modifica: n

Calcula: Inserta el par *(c, v)* en el diccionario del nodo *n*. Es decir, en el nodo *n* el carácter *c* apunta a *v*. Si ya existía la clave *c*, modifica el valor de *v*.

Operación ValorDe (ent n: NodoTrie[A]; c: A): Puntero[NodoTrie[A]]

Calcula: Busca si en el diccionario asociado a *n* existe alguna asociación para el carácter *c*. Si existe esa asociación *(c, v)*, entonces devuelve *v*. En caso contrario

devuelve el puntero *NULO*.

Operación TomaNuevo (ent *n*: NodoTrie[A]; *c*: A)

Modifica: *n*

Requiere: Crea un nuevo nodo trie, cuyo puntero es asociado al carácter *c* dentro del nodo *n*. Esta operación es equivalente a aplicar la operación Crear(*tmp*), y luego Asigna(*n*, *c*, PunteroA(*tmp*)).

Fin NodoTrie.

Normalmente, el número de caracteres que forman el alfabeto de un árbol trie será de tamaño conocido y reducido. Por ejemplo, en el caso de las palabras en español el alfabeto contiene 27 letras más la marca \$. Si incluimos los acentos tendríamos 33 elementos en total. En estas circunstancias, es posible usar las dos técnicas básicas de representación de diccionarios: mediante arrays y mediante listas enlazadas. Estas representaciones aplicadas sobre nodos trie son mostradas en la figura 4.4.

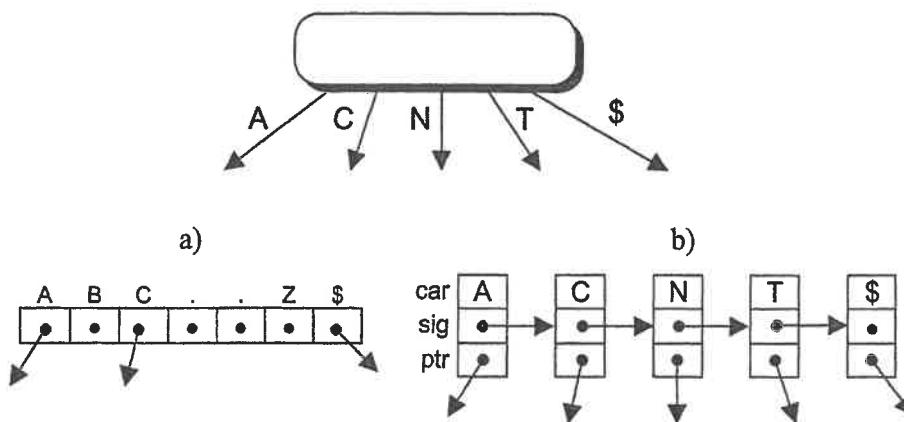


Figura 4.4: El tipo *NodoTrie["A", ..., "Z", \$]* y las dos representaciones básicas. a) Representación mediante arrays. b) Representación mediante listas.

Representación de nodos mediante arrays

En la primera representación básica, la definición del tipo *NodoTrie* es un array de punteros indexado por los caracteres del alfabeto. Suponemos que estos caracteres son numerados de manera secuencial y que la función *Rango* devuelve un rango enumerado con los valores del alfabeto.

tipo

NodoTrie[A: Alfabeto] = array [Rango(A)] de Puntero[NodoTrie[A]]

La implementación de las operaciones es sencilla y eficiente.

operación Asigna (var *n*: NodoTrie[A]; *c*: A; *v*: Puntero[NodoTrie[A]])

n[*c*] := *v*

operación ValorDe (*n*: NodoTrie[A]; *c*: A): Puntero[NodoTrie[A]]

devolver *n*[*c*]

operación TomaNuevo (var *n*: NodoTrie[A]; *c*: A)

```

tmp:= CrearNodo( )
para cada i en Rango(A) hacer
    tmp[i]:= NULO
finpara
n[c]:= tmp

```

Claramente, todas las operaciones tardan un $O(1)$, excepto TomaNuevo que tarda un $O(d)$ en la inicialización, siendo d el número de caracteres del alfabeto. Sin embargo, nuevamente el problema de la representación con arrays es el uso de memoria. Se reserva una cantidad fija, que después puede no ser usada. En el caso de los árboles Trie, los arrays estarán muy ocupados en los niveles próximos a la raíz, pero es previsible que en los niveles más profundos la mayoría de las posiciones estén sin ocupar.

Representación de nodos mediante listas

Para solucionar el problema del uso de memoria utilizamos una representación mediante listas enlazadas, en la que sólo reservamos memoria para las claves existentes. Pero, ya podemos prever que este ahorro de memoria repercutirá en el tiempo de ejecución. La definición del tipo podría ser como la siguiente.

tipo

NodoTrie[A: Alfabeto] = Lista[Asociación[A, Puntero[NodoTrie[A]]]]

Es decir, un nodo trie es una lista de asociaciones, de caracteres con punteros a nodos descendientes³. En la figura 4.4b) se concreta más aun la representación del tipo. La definición sería la siguiente.

tipo

NodoTrie[A: Alfabeto] = registro

car: A // Carácter (clave)

ptr: Puntero[NodoTrie[A]] // Nodo descendiente (valor)

sig: Puntero[NodoTrie[A]] // Siguiente nodo en la lista

finregistro

Este tipo de representación de árboles n-arios, donde los nodos son listas de punteros a nodos descendientes, es lo que se conoce normalmente como **representaciones hijo izquierdo-hermano derecho**; el nodo apuntado por ptr es el hijo izquierdo y el apuntado por sig el hermano derecho.

Las operaciones Crea, Asigna, ValorDe y TomaNuevo se convierten en simples operaciones de manejo de las listas de manera adecuada. Por ejemplo, suponiendo que usamos listas ordenadas según la clave, la operación ValorDe con la segunda definición podría ser la siguiente.

operación ValorDe (n: NodoTrie[A]; c: A): Puntero[NodoTrie[A]]

tmp:= PunteroA(n)

mientras (tmp ≠ NULO) Y (tmp↑.car < c) hacer

tmp:= tmp↑.sig

finmientras

si (tmp = NULO) O (tmp↑.car ≠ c) entonces

³Recordemos que, en última instancia, un nodo trie no es ni más ni menos que una instanciación concreta del TAD Diccionario.

```

devolver NULO
sino
    devolver tmp $\uparrow$ .ptr
finsi

```

El tiempo de ejecución de la operación será proporcional al tamaño de las listas. En el peor caso, la lista será de tamaño d , el tamaño del alfabeto. Por lo tanto, el tiempo está en un $O(d)$. En el caso promedio, el tiempo dependerá del número de descendientes existentes en cada nodo. Como ya hemos visto, es previsible que este número sea próximo a d en la raíz y próximo a 1 en las hojas.

Algo parecido pasaría con la operación **Asigna**, que debería buscar en primer lugar la posición de la lista donde colocar la clave en orden. La operación tendría también un $O(d)$, aunque en promedio el tiempo es bastante menor.

4.1.3. Búsqueda e inserción de palabras en un trie

Recordemos que el tipo **ArbolTrie** ha sido definido como un puntero al tipo **NodoTrie**. A su vez, hemos dado dos posibles implementaciones para **NodoTrie**. En la figura 4.5 se muestra gráficamente la disposición en memoria de estas dos estructuras, para el ejemplo de la figura 4.3.

Respetando el principio de ocultamiento, las operaciones sobre árboles trie deberían ser independientes de la representación que se use en los nodos. Es decir, las operaciones **Inserta**, **Miembro** y **Suprime** sobre árboles trie están basadas en las operaciones **Crear**, **Asigna**, **ValorDe** y **TomaNuevo** sobre nodos trie, y deben funcionar igual para los dos tipos de estructuras de **NodoTrie** estudiadas sin necesidad de hacer cambios en el código.

Supongamos definido el tipo **cadena** para representar palabras, con posibilidad de indexación para acceder a los caracteres individuales que forman la cadena. Por ejemplo, si **c** es de tipo **cadena** y contiene la palabra "hola", entonces **c[1]**= "h", **c[2]**= "o", **c[3]**= "l", **c[4]**= "a" y **c[5]**= \$. El método para encontrar una palabra en el árbol consistiría en moverse a través del mismo hacia las ramas correspondientes, según la palabra buscada, empezando en la raíz. La implementación de **Miembro** podría ser como la siguiente.

operación **Miembro** (*a*: **ArbolTrie[A]**; *cad*: **cadena**): booleano

```

var tmp: Puntero[NodoTrie[A]]
i:= 1
tmp:= a
mientras (cad[i]  $\neq$  $) Y (tmp  $\neq$  NULO) hacer
    tmp:= ValorDe(tmp $\uparrow$ , cad[i])
    i:= i + 1
finmientras
devolver (tmp  $\neq$  NULO) Y (ValorDe(tmp $\uparrow$ , $)  $\neq$  NULO)

```

Para indicar en el árbol que el prefijo actual es una palabra, el puntero asociado a la marca de fin \$ toma un valor no nulo. Si no queremos utilizar memoria adicional, podemos hacer simplemente que el puntero referencia al mismo nodo, como aparece en la figura 4.5. El procedimiento **Inserta** es parecido a **Miembro**, en el recorrido simultáneo que realiza en el árbol y en la cadena. Pero, en este caso, si la rama por la que se mueve no existe, entonces la debe ir creando.

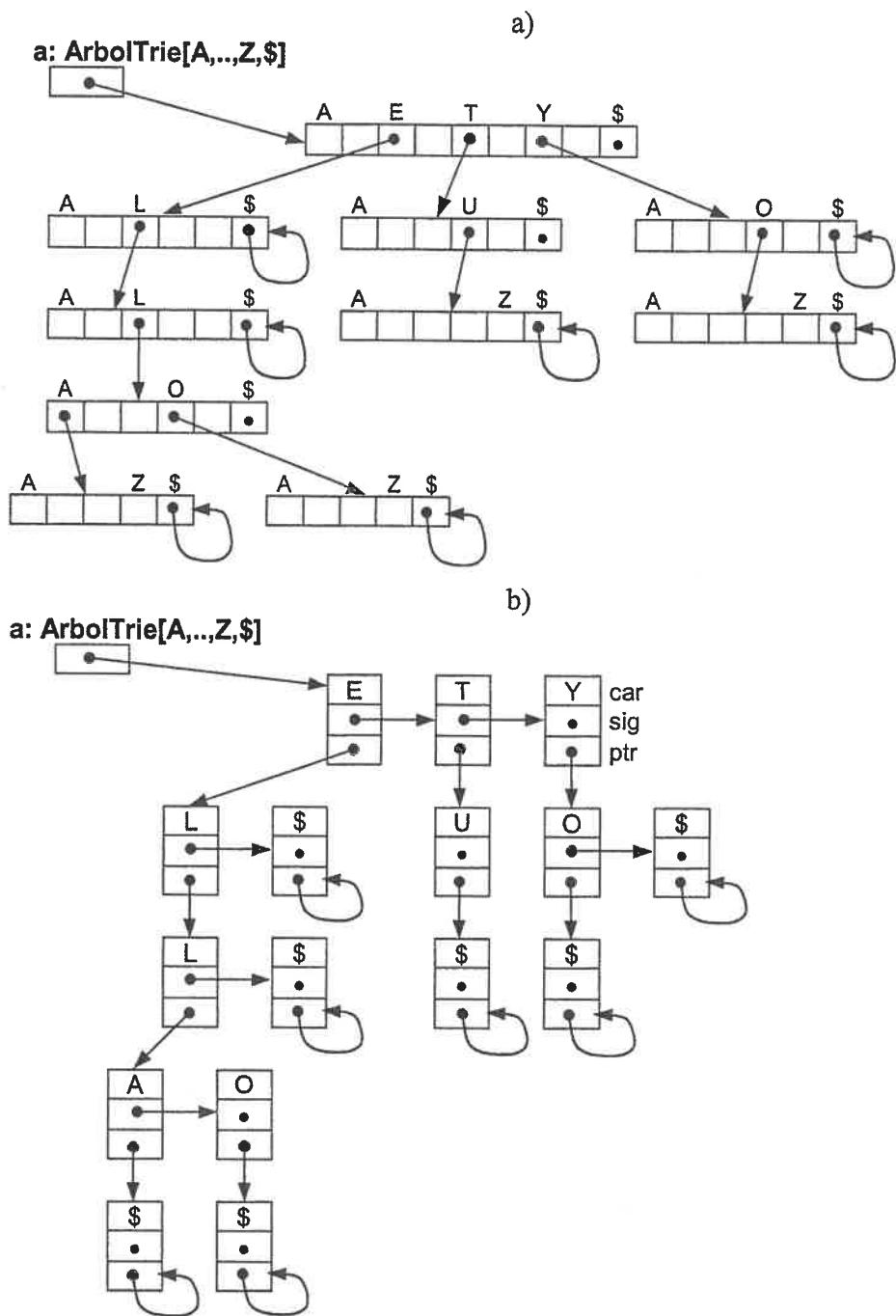


Figura 4.5: Estructuras de árbol trie con el conjunto de palabras: {YO, TU, EL, ELLA, ELLA, Y, E}. a) Representando los nodos mediante arrays. b) Representando los nodos mediante listas.

```

operación Inserta (var a: ArbolTrie[A]; cad: cadena)
var tmp: Puntero[NodoTrie[A]]
    si a = NULO entonces
        a:= CrearNodo
    finsi
    i:= 1
    tmp:= a
    mientras (cad[i] ≠ $) hacer
        si ValorDe(tmp↑, cad[i]) = NULO entonces
            TomaNuevo (tmp↑, cad[i])
        finsi
        tmp:= ValorDe(tmp↑, cad[i])
        i:= i + 1
    finmientras
    Asigna(tmp↑, cad[i], tmp)

```

Teniendo en cuenta el funcionamiento de las anteriores operaciones, el árbol trie almacena un conjunto de palabras. Es decir, para cada posible palabra simplemente sabemos si existe o no. Supongamos que queremos usar los trie para representar diccionarios, esto es, asociamos a cada palabra un valor de cierto tipo, por ejemplo la definición de esa palabra o su traducción al esperanto. ¿Qué partes habría que modificar en la estructura y en las operaciones?

La respuesta es relativamente simple: los punteros asociados a \$, en lugar de referenciar al mismo nodo deberían apuntar a una variable del tipo adecuado. Cada puntero etiquetado con \$ indica una palabra completa, de manera que el valor al que señala el puntero sería el valor correspondiente a esa palabra.

Por otro lado, la implementación de la operación Suprime sería parecida a las dos anteriores. En caso de existir la palabra, el puntero final asociado a \$ debería ponerse a valor nulo. Además, habría que eliminar memoria de forma adecuada. Si en un nodo todos los punteros toman valor nulo, se podría eliminar ese nodo, eliminando también el puntero del nodo padre.

4.1.4. Eficiencia y comparación entre estructuras

Tal y como han sido diseñados, los árboles trie serán adecuados cuando el conjunto de cadenas representadas comparten muchos prefijos comunes. Para conocer el “grado de partición” de los prefijos, deberíamos comparar el número total de prefijos con la suma de las longitudes de las cadenas. Por ejemplo, en el árbol de la figura 4.3 existen 9 prefijos⁴, mientras que la longitud total de las cadenas contenidas es 16. Eso quiere decir que cada prefijo se usa unas 1,78 veces en promedio. En el ejemplo más realista de la figura 4.2, existen 41 prefijos⁵ y la longitud de las cadenas es 127; cada prefijo se usa en promedio en 3,1 palabras distintas. Este número podría aumentar bastante más –posiblemente hasta 6 ó 7– si consideramos las conjugaciones verbales; y cuanto mayor sea más adecuada será la representación de árboles trie. Vamos a analizarlo cuantitativamente, comparándolo con

⁴Recordemos que los prefijos son todos los nodos excepto la raíz y las hojas.

⁵En este caso, los prefijos serían todos los nodos del árbol.

las estructuras de dispersión.

Utilización de memoria

Supongamos que en un árbol trie almacenamos n palabras, la suma de las longitudes de las palabras es l , el número total de prefijos es p y el alfabeto consta de d caracteres (incluyendo la marca de fin). Está claro que $n \leq p \leq l$, aunque normalmente $n \ll p \ll l$. Consideremos también k_1 bytes el tamaño de un puntero y k_2 bytes lo que ocupa un carácter.

La relación l/n indica la longitud, en término promedio, de las palabras almacenadas. Por otro lado, como ya hemos visto, la relación l/p indica el grado de compartición de cada prefijo por diferentes palabras. Este último factor es el que determina el ahorro de memoria, respecto a una representación donde no se factoricen los prefijos.

- Si representamos los nodos trie mediante arrays, está claro que cada nodo ocupará en total $d k_1$ bytes. Existirá un nodo por cada prefijo y uno más para la raíz. En total, la memoria necesaria será $d k_1(p + 1)$ bytes.
- En la representación de nodos mediante listas enlazadas –usando la segunda definición– cada celda de una lista contiene un carácter y dos punteros, lo cual hace $2k_1 + k_2$ bytes. En este caso, habrá tantas celdas como prefijos más una celda con la etiqueta $\$$ por cada una de las n palabras. Sumando, el espacio total ocupado será $(2k_1 + k_2)(n + p)$ bytes.

Vamos a comparar la memoria necesaria con la que se utilizaría usando dispersión abierta y cerrada. Para estos dos casos, hay que tener en cuenta que las cadenas pueden tener una longitud variable. Supondremos que una cadena es representada mediante un puntero a una zona de memoria, que contiene un array de caracteres acabado con la marca de fin $\$$, como se muestra en la figura 4.6. De esta forma no se limita el tamaño máximo de las cadenas (como ocurriría si reserváramos un array de longitud fija). Una cadena de longitud t ocupará $k_1 + (t + 1)k_2$ bytes.

Con la anterior representación y usando los resultados del capítulo 3, la memoria requerida en dispersión cerrada sería $B_1 k_1 + (l + n)k_2$ bytes, donde $(l + n)k_2$ es el tamaño total de un array que almacena consecutivamente todas las cadenas (ver figura 4.6a)). Finalmente, la memoria en dispersión abierta sería $B_2 k_1 + 2n k_1 + (l + n)k_2$ bytes (ver figura 4.6b)). Diferenciamos el número de cubetas en dispersión cerrada, B_1 , y abierta, B_2 , porque no necesariamente deben coincidir.

Con estos resultados ¿qué estructura ocupa menos memoria? ¿Cuánta menos? Es difícil hacerse una idea exacta a partir de las fórmulas, puesto que dependen de muchos factores. En su lugar, vamos a asignar valores concretos a las variables para dos ejemplos y analizar el comportamiento en esos casos. El primer ejemplo son todas las palabras listadas en la sección 4.1.1, donde tenemos que $n = 38$ palabras, $l = 378$ letras en total, $p = 118$ prefijos, $d = 28$ letras del alfabeto. En el segundo ejemplo contamos las conjugaciones del verbo “hablar”, con los valores $n = 37$ palabras, $l = 283$ letras en total, $p = 52$ prefijos, y el mismo $d = 28$. En este caso $l/p = 5,44$. En ambos ejemplos, consideraremos que un puntero ocupa $k_1 = 2$ bytes (puesto que los conjuntos son de tamaño reducido) y un carácter $k_2 = 1$ byte. Los resultados son mostrados en la tabla 4.1.

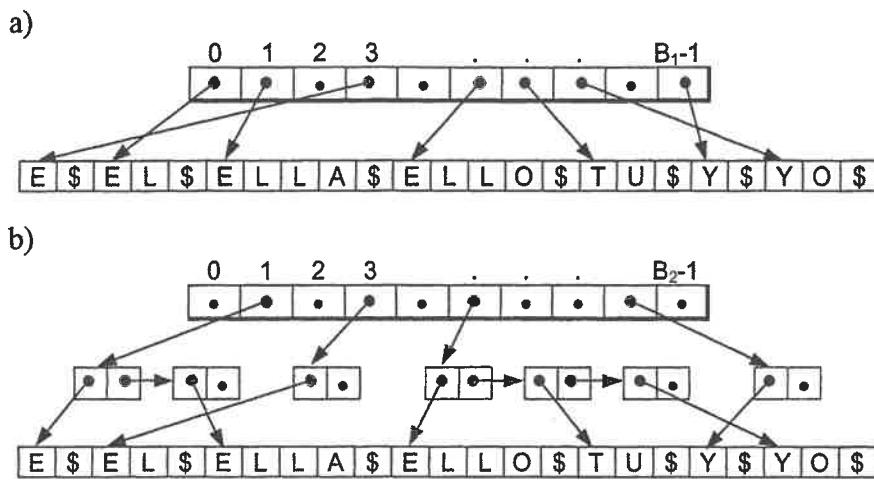


Figura 4.6: Representación mediante tablas de dispersión del conjunto de palabras: {YO, TU, EL, ELLA, ELLO, Y, E}. a) Con dispersión cerrada. b) Con dispersión abierta.

Estructura	Memoria necesaria	Ejemplo 1 (bytes)	Ejemplo 2 (bytes)
Árbol trie Nodos con arrays	$d k_1(p + 1)$	6.664	2.968
Árbol trie Nodos con listas	$(2k_1 + k_2)(n + p)$	780	445
Dispersión cerrada (con $B_1 = 1, 5n$)	$B_1 k_1 + (l + n)k_2$	530	431
Dispersión abierta (con $B_2 = n$)	$B_2 k_1 + 2n k_1 + (l + n)k_2$	644	542

Tabla 4.1: Tabla comparativa de la memoria usada en representación de conjuntos, con árboles trie y tablas de dispersión. El ejemplo 1 corresponde a las 38 palabras listadas en la sección 4.1.1. El ejemplo 2 son 37 palabras de la conjugación del verbo “hablar”.

Como era previsible, la representación de tries con arrays ocupa casi un orden de magnitud (10 veces) más que las restantes. Esto es debido a que la mayoría de las posiciones de los arrays estarán con valor NULO, dando lugar a un gran desperdicio de memoria. En un conjunto de tamaño grande esto podría ser un gran problema. Es más, si tuviéramos que usar punteros de 4 bytes o más, el grado de desperdicio de memoria sería aun mucho mayor.

Por contra, el uso de memoria en la representación con listas se encuentra dentro de los mismos órdenes de magnitud que usando tablas de dispersión, mejorándolo en algunos casos. En particular, será mejor cuantos más prefijos se compartan. En el ejemplo 1, donde $l/p = 3,20$, los nodos trie con listas ocupan un 21,1% más que con dispersión abierta; sin embargo en el ejemplo 2, donde $l/p = 5,44$, el árbol trie usa un 21,8% menos de memoria. En todos los casos, la dispersión cerrada requiere sensiblemente menos memoria.

Tiempos de ejecución

Los árboles trie tienen una propiedad muy interesante: el tiempo de ejecución no depende del número n de elementos almacenados, sino de la longitud de la palabra sobre la que se hace la consulta. En efecto, si analizamos las operaciones **Miembro** e **Inserta**, vemos que ambas constan de un bucle que se repite tantas veces como la longitud de la palabra más uno. Dentro de los bucles se ejecutan algunas instrucciones con tiempo constante y la operación **ValorDe**.

- En la representación de tries con arrays el tiempo de **ValorDe** es un $O(1)$. Por lo tanto, el tiempo de buscar un elemento sería proporcional a la longitud de la palabra. En promedio, la longitud de las palabras será l/n y el tiempo $O(l/n)$.
- En la representación con listas, el tiempo de la operación **ValorDe** depende de la longitud de las listas en los nodos del árbol. Sabemos que estas listas serán como máximo de tamaño d , pero normalmente serán mucho menores. En promedio, se puede comprobar⁶ que las listas serán de tamaño $(p+n)/(p+1)$, lo cual –teniendo en cuenta que $p \geq n$ – es siempre menor que 2. Por lo tanto, el tiempo de **ValorDe** en promedio está limitado por una constante y el tiempo de la búsqueda es un $O(l/n)$.

En el capítulo anterior vimos que la eficiencia de las tablas de dispersión es proporcional al tamaño de las secuencias de búsqueda que, a su vez, están en función de la proporción de llenado n/B . El tiempo de las operaciones en dispersión abierta era de $O(1 + n/B)$, mientras que en dispersión cerrada teníamos un $O(1/(1 - n/B))$. Pero, realmente, deberíamos incluir también el tiempo de calcular la función de dispersión, que fue obviado en el capítulo anterior. Una buena función tendría que procesar todos los caracteres de la cadena, por lo que su tiempo estaría en $O(l/n)$. En definitiva, los tiempos de las distintas representaciones son mostrados en la tabla 4.2.

Árbol trie Nodos con arrays	Árbol trie Nodos con listas	Dispersión Cerrada	Dispersión Abierta
$O(l/n)$	$O(l/n)$	$O(n/B + l/n)$	$O(1/(1 - n/B) + l/n)$

Tabla 4.2: Tabla comparativa del tiempo de ejecución, para la operación de consulta, usando árboles trie y tablas de dispersión.

La eficiencia de los árboles trie, para las operaciones de consulta, es claramente mejor que con tablas de dispersión. Además, los árboles trie permiten una rápida implementación de otras operaciones como calcular el mínimo de un conjunto, el máximo, listar por orden alfabético, enumerar las palabras que incluyan un prefijo, etc. Usando tablas de dispersión, todas esas operaciones requerirían un costoso recorrido completo de toda la tabla.

⁶Existen en total $(p+n)$ elementos de listas, repartidos entre $(p+1)$ nodos.

Es más, en la aplicación del corrector ortográfico interactivo los tries resultan especialmente interesantes. A medida que el usuario escribe, el programa iría moviéndose por el árbol, de manera que cada pulsación de una tecla correspondería a bajar un nivel en el árbol. Al acabar una palabra, se comprobaría si existe un valor para la etiqueta $\$$. Todas las operaciones tardarían un tiempo constante y muy reducido. El corrector podría incluir otras utilidades, como por ejemplo completar una palabra a medio escribir si a partir de ese prefijo sólo existe una palabra correcta.

4.2. Relaciones de equivalencia

Las relaciones de equivalencia son un concepto matemático definido sobre un conjunto dado cualquiera. Como tantos otros conceptos matemáticos, está basado en una idea intuitiva, la representación de relaciones del tipo: ciudades en una misma región, alumnos de la misma clase, instrucciones dentro del mismo bloque de código, enteros con el mismo valor de módulo P , etc.

Definición 4.2 Una relación de equivalencia sobre un conjunto C es una relación R que cumple las siguientes propiedades⁷:

- Reflexiva. $\forall a \in C; a R a$
- Simétrica. $\forall a, b \in C; a R b \Leftrightarrow b R a$
- Transitiva. $\forall a, b, c \in C; (a R b) \wedge (b R c) \Rightarrow (a R c)$

Es fácil comprobar estas propiedades para los ejemplos anteriores. Por ejemplo, la propiedad reflexiva significa que una ciudad está en la misma región que ella misma, ¡obviamente!; la simétrica diría que si la ciudad a está en la misma región que b , entonces b está en la misma región que a ; y la transitiva, que si a está en misma región que b , y esta en la misma que c , entonces a y c están en la misma región. Las tres se cumplen de manera trivial. En la figura 4.7 se muestra un ejemplo particular de relación de equivalencia, definida sobre un conjunto de nueve elementos.

Las relaciones de equivalencia dan lugar al concepto de **clase de equivalencia**. La clase de equivalencia de un elemento a , según una relación R sobre un conjunto C , es el subconjunto de todos los elementos c de C tales que $c R a$. Por ejemplo, en los casos anteriores tendríamos: la clase de las ciudades de la Región de Murcia, la clase de los alumnos de segundo curso, etc. El conjunto de todas las clases de equivalencia definidas sobre C según la relación R , forma una **partición** de ese conjunto, es decir, son un conjunto de subconjuntos disjuntos y la unión de todos ellos es el conjunto de partida C .

Nuestro interés, como programadores, es estudiar la implementación eficiente de relaciones de equivalencia definidas sobre conjuntos finitos. Además, vamos a considerar especialmente relaciones que no pueden ser calculadas mediante una fórmula (como ocurre, por ejemplo, con la igualdad de enteros módulo P) y que pueden variar en tiempo de ejecución. En otro caso, la implementación sería inmediata. Pero, en primer lugar, vamos a definir un tipo abstracto de datos para el uso de relaciones de equivalencia.

⁷La expresión “ $a R b$ ” se lee “ a está relacionado con b ”.

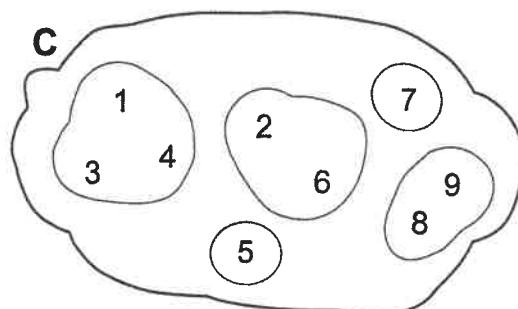


Figura 4.7: Ejemplo de relación de equivalencia y clases de equivalencia, definidos sobre el conjunto de los enteros del 1 a 9. Los elementos que están en la misma clase de equivalencia aparecen dibujados dentro del mismo subconjunto del conjunto C .

El TAD relación de equivalencia

La definición del tipo abstracto debe enumerar las operaciones que se pueden aplicar sobre relaciones de equivalencia y su significado. Existen varias maneras de plantear la utilización del tipo. Una primera forma sería empezar con una relación mínima, vacía, al crearla y tener una operación para unir varias clases de equivalencia en una. Otra posibilidad sería empezar una sola clase de equivalencia al crear e incluir operaciones para partir una clase en dos o más. A lo largo de toda esta sección, trabajaremos exclusivamente con la primera posibilidad.

Además de los constructores del tipo –que, en este caso, serían la operación Crear para formar una nueva relación vacía y Union para juntar dos clases– añadimos una operación Encuentra para consultar la clase de equivalencia de un elemento de C cualquiera. La especificación informal del tipo abstracto es la siguiente.

TAD RelacionEquiv[T: tipo] es Crear, Union, Encuentra

Requiere

El tipo T debe tener definida una operación de comparación *Igual* (*ent* T, T ; *sal* boolean).

Descripción

Los valores de tipo RelacionEquiv[T] son relaciones de equivalencia modificables, definidas sobre conjuntos de elementos del tipo T . En particular, el conjunto de elementos que pueden participar en la relación de equivalencia es indicado en la operación Crear. Para añadir relaciones entre elementos se debe utilizar la operación Union, y para consulta la clase de un elemento se debe usar Encuentra.

Operaciones

Operación Crear (ent C : Conjunto[T]; sal RelacionEquiv[T])

Requiere: El conjunto C debe ser no vacío.

Calcula: Devuelve una relación de equivalencia mínima sobre el conjunto C , donde las clases de equivalencia son los elementos en sí mismos, es decir, para todo a y b de C , a está relacionado con b sí y sólo si $a=b$.

Operación Union (ent R : RelacionEquiv[T]; a, b : T)

Requiere: Los elementos a y b deben pertenecer al conjunto C sobre el que se ha definido la relación de equivalencia R .

Modifica: R

Calcula: En la nueva relación de equivalencia R , los elementos a y b pertenecen a la misma clase de equivalencia. Además, se mantienen las demás relaciones definidas previamente sobre R .

Operación Encuentra (**ent** R : RelacionEquiv[T]; a : T; **sal** T)

Requiere: El elemento a deben pertenecer al conjunto C sobre el que se ha definido la relación de equivalencia R .

Calcula: Devuelve el nombre de la clase de equivalencia a la que pertenece el elemento a dentro de la relación R . Este nombre es un elemento representativo dentro de dicha clase.

Fin RelacionEquiv.

Para saber si dos elementos x e y están en la misma clase, simplemente comprobamos si $\text{Encuentra}(x) = \text{Encuentra}(y)$. Como se documenta en la especificación de Encuentra, el resultado es un elemento de los que pertenecen a la clase de x . Este elemento puede ser seleccionado arbitrariamente entre los de dicha clase. Solamente necesitamos que al aplicar Encuentra para dos elementos relacionados, devuelva siempre el mismo valor.

Por ejemplo, en la relación de la figura 4.7, a la clase formada por $\{1, 3, 4\}$ se le puede asociar como nombre representativo cualquiera de esos tres valores; pero una vez asociado uno, no puede variar. En definitiva, lo importante no es el valor concreto, sino que $\text{Encuentra}(1) = \text{Encuentra}(3) = \text{Encuentra}(4)$. Obviamente, el nombre sí que puede cambiar después de una operación Union.

Ejemplo de uso de relaciones de equivalencia

Un ejemplo de utilización intensiva de las relaciones de equivalencia lo podemos encontrar en el procesamiento de imágenes digitales. Supongamos una imagen en escala de grises, como la de la figura 4.8, que está compuesta por una matriz de píxeles. En este caso, las clases de equivalencia son las regiones contiguas y del mismo color. Estas clases surgen de la siguiente relación de equivalencia: dos píxeles a y b están relacionados si tienen el mismo color (o nivel de gris) y además son adyacentes en la imagen⁸.

En esta aplicación, el cálculo y la modificación de las clases de equivalencia puede resultar interesante para diversos propósitos. Por ejemplo, si queremos aplicar la operación “rellenar una región de color”, tendremos que buscar la clase de equivalencia del punto sobre el que se aplica el relleno. Tras aplicar la operación, pueden ocurrir cambios en las clases de equivalencia, que se deberán tener en cuenta posteriormente.

También podrían resultar interesantes operaciones para contar el número de regiones existentes en la imagen, o saber cuál contiene más píxeles. Por ejemplo, un posible algoritmo de detección de caras humanas en una imagen podría estar basado en buscar una región contigua de color de piel, y que contenga por lo menos dos regiones oscuras correspondientes a los ojos.

En todas estas aplicaciones, la relación está definida sobre un conjunto finito, y puede variar dinámicamente a lo largo del tiempo. Si consideremos imágenes con una resolución de 800×600 píxeles, entonces el conjunto C sobre el que está definida la relación tendrá 480.000 elementos. Por lo tanto, conseguir una implementación muy eficiente para

⁸Es decir, uno se encuentra inmediatamente encima, debajo, a la izquierda o a la derecha del otro.

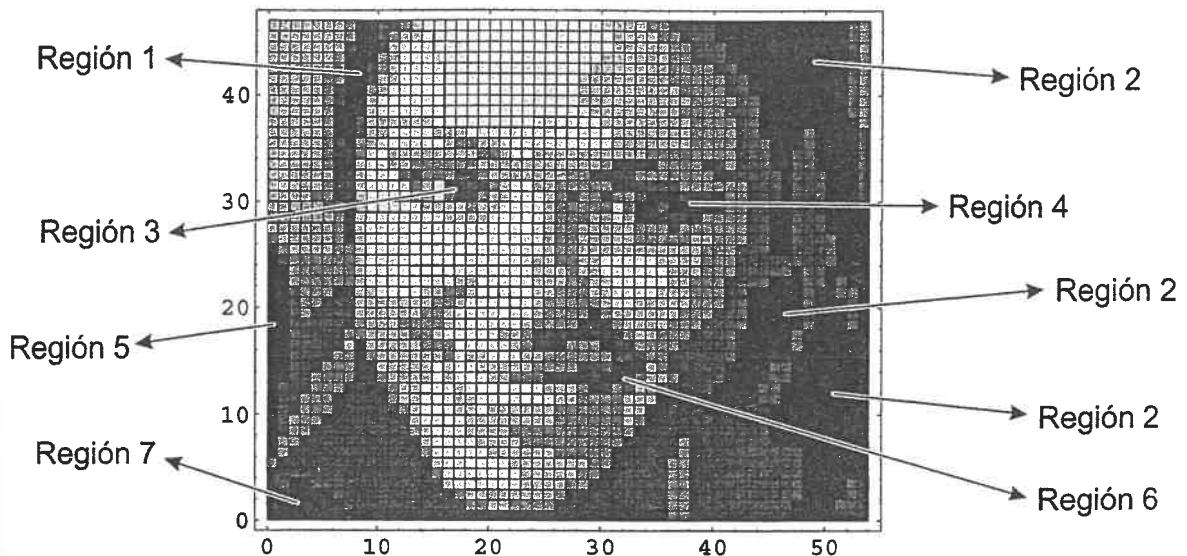


Figura 4.8: Ejemplo de aplicación de relaciones y clases de equivalencia, en procesamiento de imágenes. Aparecen señaladas algunas clases (regiones) cuyos píxeles tienen color negro.

conocer y manejar las clases de equivalencia resulta fundamental en este caso.

4.2.1. Representaciones sencillas de relaciones de equivalencia

La representación de relaciones de equivalencia es un ejemplo típico de problema donde se intuye fácilmente una solución simple y directa. Pero un pequeño e irritante contratiempo hace que esta solución se comporte peor de lo que se esperaba, lo cual obliga a tomar un largo rodeo para alcanzar una representación eficiente.

Representación mediante arrays

Supongamos, para simplificar el problema, que las relaciones de equivalencia están definidas sobre un tipo enumerado. Es más, supongamos que los elementos del conjunto C sobre el que se definen están numerados desde 1 hasta cierto límite N . Por ejemplo, en el caso de las imágenes, podemos asignar un número consecutivo a cada píxel, de arriba abajo y de izquierda a derecha. Para imágenes de resolución 800×600 , $N = 480.000$.

La representación más sencilla podría consistir en un array de tamaño N , donde en cada posición se almacena el nombre de la clase de equivalencia del elemento correspondiente. La definición del tipo sería la siguiente.

tipo

RelacionEquiv[N] = array [1..N] de entero

En la figura 4.9a) se muestra un ejemplo de la disposición en memoria de la estructura, para la relación de equivalencia definida en la figura 4.7.

La implementación de las operaciones **Crear** y **Encuentra** es inmediata.

operación Crear (var R: RelacionEquiv[N])

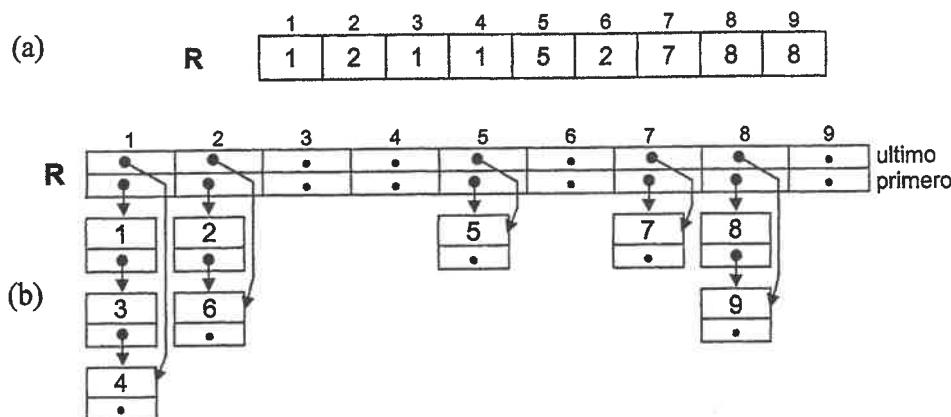


Figura 4.9: Ejemplo de representaciones sencillas de relaciones de equivalencia, para la relación de la figura 4.7. a) Representación con arrays. b) Representación con listas.

```

para i := 1, ..., N hacer
    R[i]:= i
finpara
operación Encuentra (R: RelacionEquiv[N]; a: entero): entero
devolver R[a]

```

La operación Encuentra está claramente en un $O(1)$, lo cual resulta muy interesante. Por otro lado, Crear tarda un $O(N)$, lo cual no es muy preocupante puesto que sólo se aplicará una vez. El problema es, sin embargo, la operación Union que es previsible que aparezca con más frecuencia. ¿Cómo hacer la unión de dos clases? Simplemente cambiando en el array el nombre de una de las clases por la otra.

Supongamos que los parámetros que recibe la operación son nombres de clases (y no valores de elementos particulares). La implementación podría ser la siguiente.

operación Union (var R: RelacionEquiv[N]; a, b: entero)

```

para i := 1, ..., N hacer
    si R[i] = b entonces
        R[i]:= a
    finsi
finpara

```

Es fácil ver que la operación Union tarda un $O(N)$, en todos los casos. Este coste puede resultar excesivo para muchas aplicaciones. En el ejemplo del procesamiento de imágenes, se aplicará la unión siempre que un píxel tenga a un lado algún otro píxel del mismo color –lo cual suele ser muy frecuente-. Si tenemos que aplicar Union para cada píxel de una imagen de 800×600 , el número total de comparaciones dentro del bucle **para** sería: $(800 * 600)^2 = 1230$ mil millones de comparaciones!

Representación mediante listas

Para solucionar el problema anterior podemos intentar una representación dinámica mediante listas, al estilo de las representaciones básicas de conjuntos estudiadas en el

capítulo anterior. Para cada clase de equivalencia usamos una lista de elementos, en la que estarán los elementos que pertenecen a la misma. La definición del tipo sería.

tipo

RelacionEquiv[N] = array [1..N] de Lista[entero]

Siendo suficientemente cuidadosos, deberíamos usar una estructura que permita concatenar dos listas de forma rápida. Esto se puede conseguir, por ejemplo, usando listas circulares y doblemente enlazadas; o simplemente almacenando en la cabecera de las listas un puntero al primer y al último elemento, como se muestra en la figura 4.9b).

Usando la segunda opción, la operación Union se puede conseguir en un tiempo constante, $O(1)$, de la siguiente forma. Se supone que el tipo Lista incluye los atributos primero y ultimo, y los nodos de lista tienen siguiente, para enlazar los elementos.

operación Union (var R: RelacionEquiv[N]; a, b: entero)

$R[a].ultimo.siguiente := R[b].primero$

$R[a].ultimo := R[b].ultimo$

$R[b] := \text{NULO}$

El problema surge ahora con la operación Encuentra. ¿Cómo podemos saber la clase a la que pertenece un elemento concreto? Deberíamos recorrer todas las listas hasta encontrarlo en alguna de ellas.

operación Encuentra (R: RelacionEquiv[N]; a: entero): entero

para $i := 1, \dots, N$ **hacer**

si BuscaLista ($R[i]$, a) **entonces**

devolver i

finsi

finpara

error ("El elemento", a, "no está en ninguna clase")

Damos por supuesta la operación BuscaLista, que realiza un simple recorrido secuencial en una lista. No siempre tardará el mismo tiempo, pero sabemos que en el peor caso se recorrerán en total todos los elementos de todas las listas, es decir N . En promedio, la operación Encuentra recorrerá la mitad de las listas, tardando un $O(N)$.

Si resulta malo que la operación Union esté en un $O(N)$, como ocurría usando arrays, el hecho de que Encuentra tarde un $O(N)$ puede ser aun más desastroso en aplicaciones que requieren un uso intensivo de las relaciones de equivalencia.

4.2.2. Representación mediante punteros al padre

El inconveniente último de las dos anteriores representaciones es que utilizan estructuras lineales –ya sean enlazadas (listas) o contiguas en memoria (arrays)– lo cual ocasiona tiempos lineales, bien para la búsqueda o para la unión de clases de equivalencia. En su lugar vamos a estudiar el uso de estructuras arbóreas. Vamos a disponer de un árbol para cada clase, donde la raíz del árbol será por definición el nombre de esa clase.

Básicamente, la unión de dos árboles se puede conseguir en un tiempo constante, colocando un árbol como subárbol del otro. En cuanto a la búsqueda, está claro que necesitamos una operación que dado un elemento del árbol encuentre cuál es la raíz. Por esta razón, usamos una representación de árboles con punteros al nodo padre.

Árboles de punteros al padre

La clave de la representación de árboles con punteros al padre es que para cada nodo del árbol sólo necesitamos conocer un valor: un puntero o referencia a su padre dentro del árbol. Para la raíz del árbol ese valor será nulo o algún otro valor especial.

Suponiendo que el número de elementos del árbol es fijo, o limitado por un tamaño reducido, la estructura de punteros al padre se puede almacenar en simple array.

tipo

ArbolEnteros[N] = array [1..N] de entero

Si R es un árbol de enteros que usa esta estructura, entonces la posición $R[i]$ indica el padre del nodo i . Un ejemplo de representación de árboles con esta definición se puede ver en la figura 4.10. En este caso, se usa el valor 0 para indicar que el nodo es la raíz de un árbol.

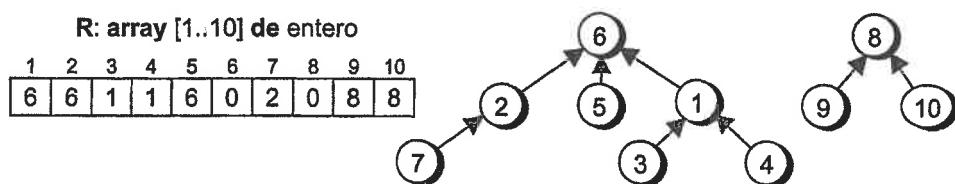


Figura 4.10: Ejemplo de representación de árboles mediante punteros al padre.

Bosques de relaciones de equivalencia

La anterior representación de árboles con punteros al padre se ajusta perfectamente a nuestras necesidades: el número de elementos de la relación de equivalencia es fijo, se define al crear la relación. Realmente, la estructura no almacena un solo árbol sino un conjunto de árboles, lo que normalmente se conoce como un bosque. La definición del tipo sería la siguiente⁹.

tipo

RelacionEquiv[N] = array [1..N] de entero

Como ya hemos avanzado, cada árbol del bosque representa una clase de equivalencia, en la cual la raíz es el nombre de esa clase. La implementación de las operaciones es sencilla. La operación *Crear* crea una relación de equivalencia mínima, donde cada elemento sólo está relacionado consigo mismo. Equivalentemente, cada elemento del conjunto será la raíz de un árbol. Usando el valor 0 para representar las raíces, tenemos.

operación Crear (var R: RelacionEquiv[N])

para $i := 1, \dots, N$ **hacer**

$R[i] := 0$

⁹El lector podrá apreciar un curioso hecho, ¡la definición es exactamente la misma que en la representación mediante arrays! La diferencia es la forma en la que se maneja el array. Mientras que antes lo usábamos para almacenar las clases de cada elemento, ahora lo usamos para representar árboles. Esto demuestra, una vez más, la estrecha relación existente entre algoritmos y estructuras de datos. La estructura de datos no determina únicamente un tipo de datos, sino que se deben tener en cuenta también los algoritmos que manipulan esa estructura.

finpara

Aunque su tiempo es claramente un $O(N)$, ya hemos visto que en este caso no supone un inconveniente, ya que sólo se hará al principio. Por otro lado, la operación Union simplemente coloca un árbol como subárbol del otro. Igual que antes, suponemos que los parámetros no son elementos sino nombres de clases de equivalencia¹⁰.

operación Union (**var** R: RelacionEquiv[N]; a, b: entero)

R[a]:= b // a se coloca como hijo de b

Así pues, la operación tarda un interesante $O(1)$. Finalmente, la operación Encuentra debe subir por el árbol, hacia sus antecesores, hasta llegar a la raíz de su árbol. Una posible implementación recursiva sería la siguiente.

operación Encuentra (R: RelacionEquiv[N]; a: entero): entero

si R[a] = 0 **entonces**

devolver a

sino

devolver Encuentra(R[a])

finsi

El tiempo de la operación Encuentra depende de la profundidad del nodo buscado dentro del árbol. En principio, es de esperar que un árbol de N nodos tenga una profundidad de un orden logarítmico en función de N , con lo cual el tiempo estaría en un $O(\log n)$. Sin embargo, en el peor caso el árbol puede adoptar una forma lineal: cualquier nodo sólo tiene un descendiente como máximo. En tal caso, un árbol de N nodos tendrá profundidad N y las operaciones de búsqueda estarán en promedio en $O(N)$. Vamos a ver un ejemplo ilustrativo de una situación de este tipo.

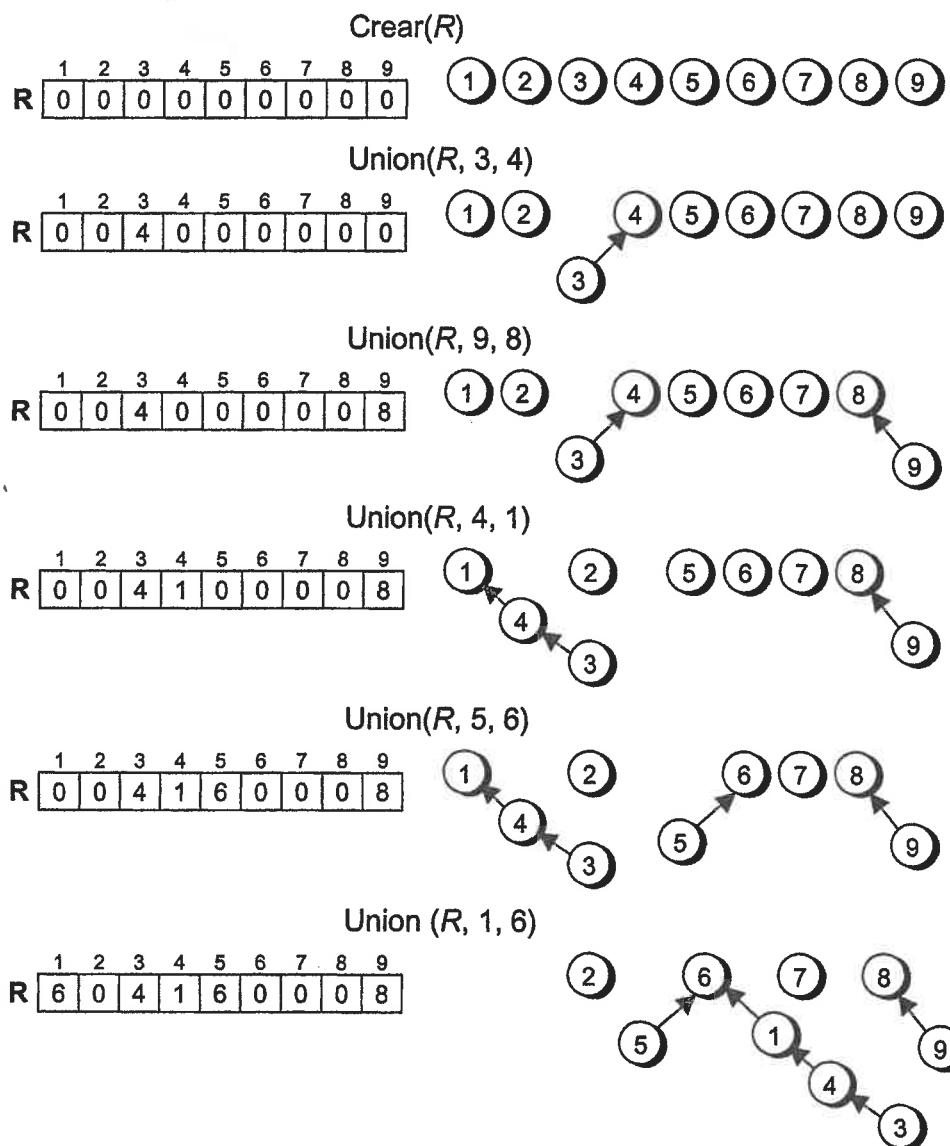
Ejemplo 4.1 Utilizando la representación de relaciones de equivalencia con punteros al padre, definimos una variable R de tipo RelacionEquiv[9], para almacenar relaciones en el conjunto de los naturales de 1 a 9. Sobre esta estructura aplicamos las siguientes operaciones: Crear(R), Union(R, 3, 4), Union(R, 9, 8), Union(R, 4, 1), Union(R, 5, 6), Union(R, 1, 6) y Encuentra(R, 3). El resultado se muestra en la figura 4.11.

4.2.3. Equilibrado y compresión de caminos

Podemos extraer una conclusión práctica del anterior ejemplo: si queremos conseguir eficiencia en una representación de árboles debemos garantizar que el árbol crecerá a lo ancho, más que a lo largo. Un árbol poco profundo significa pocos pasos en la búsqueda, lo cual se traduce en operaciones más rápidas. Entonces, ¿cómo podemos hacer que el árbol crezca a lo ancho?

Supongamos, en primer lugar, que tenemos un árbol a de profundidad 40 y otro árbol b de profundidad 2. Tenemos dos opciones –en principio igual de válidas–, colocar a como hijo de b o viceversa. Si hacemos que a sea hijo de b , entonces el nuevo árbol tendrá profundidad 41. En cambio, si colocamos b como hijo de a , la profundidad del árbol no varía, seguiría siendo 40. El resultado es bastante claro: si queremos que el árbol no crezca en profundidad, debemos colocar el árbol menos profundo como subárbol

¹⁰En caso contrario, habría que aplicar las operaciones de búsqueda correspondientes.



$\text{Encuentra}(R, 3) = \text{Encuentra}(R, 4) = \text{Encuentra}(R, 1) = \text{Encuentra}(R, 6) = 6$

Figura 4.11: Evolución de la estructura de relaciones de equivalencia, para las operaciones del ejemplo 4.1.

del más profundo. Si ambos son igual de profundos, entonces necesariamente el árbol resultante crecerá en profundidad.

Calcular la profundidad de los árboles puede resultar de por sí muy costoso, y especialmente con el tipo de representación usado. Pero si se almacena la profundidad y se va actualizando a medida que crece el árbol, podemos lograrlo eficientemente.

Para no incurrir en un uso adicional de memoria, vamos a usar la misma definición del tipo que en el anterior apartado, pero cambiando la forma de manejar las operaciones.

En particular, si antes almacenábamos un 0 en el array para las raíces, ahora podemos tener 0 o un número negativo indicando la profundidad de ese árbol. De esta manera, en la operación Union podemos comprobar fácilmente cuál es el árbol más profundo. En caso de que sean iguales, da lo mismo cuál se coloque como hijo y además sabemos que el árbol resultante aumentará de nivel.

```
operación Union (var R: RelacionEquiv[N]; a, b: entero)
    si R[b] < R[a] entonces // El árbol b es más profundo
        R[a]:= b
    sino // El árbol a es más o igual de profundo
        si R[a] = R[b] entonces
            R[a]:= R[a]-1
        finsi
        R[b]:= a
    finsi
```

Por otro lado, en la operación Encuentra es posible hacer otra modificación interesante conocida como **compresión de caminos**. Supongamos que sobre una relación de equivalencia aplicamos la operación Encuentra(*x*), obteniendo como resultado una clase *y* cualquiera. La operación habrá necesitado un número de pasos que depende de su profundidad en el árbol. Ahora bien, puesto que después de ejecutar la operación sabemos que la clase de *x* es *y*, es posible hacer que *x* apunte directamente a *y*, esto es *R[x]:=y*. De esta forma, la próxima vez que se ejecute Encuentra(*x*) sólo se necesitará un paso para llegar a la raíz del árbol.

Es más, la idea de apuntar directamente a la raíz del árbol se puede aplicar a todos los antecesores del nodo *x*, que se recorren al aplicar la operación de búsqueda. En definitiva, la operación Encuentra con compresión del árbol sería la siguiente.

```
operación Encuentra (var R: RelacionEquiv[N]; a: entero): entero
    si R[a] ≤ 0 entonces
        devolver a
    sino
        R[a]:= Encuentra(R, R[a])
        devolver R[a]
    finsi
```

El objetivo del proceso de compresión de caminos es reducir la profundidad de los nodos sobre los que se aplica la búsqueda. En algún caso, esta reducción puede dar lugar a una disminución en la profundidad total del árbol.

Sin embargo, se puede ver que la implementación de la operación Encuentra no actualiza esa profundidad, que es almacenada en las raíces con valores negativos. Esta omisión no es casual, recalcular la profundidad del árbol supondría un coste computacional de $O(N)$ que echaría a perder la eficiencia de tipo sin producir una mejora sustancial posterior. Así pues, podemos decir que la profundidad almacenada en los nodos raíces es realmente un valor de profundidad máxima o profundidad suponiendo que no se aplican búsquedas.

Ejemplo 4.2 Vamos a analizar la ejecución de las operaciones del ejemplo 4.1, con las modificaciones propuestas para Union y Encuentra. Las operaciones aplicadas son :

$\text{Crear}(R)$, $\text{Union}(R, 3, 4)$, $\text{Union}(R, 9, 8)$, $\text{Union}(R, 4, 1)$, $\text{Union}(R, 5, 6)$, $\text{Union}(R, 4, 6)$ y $\text{Encuentra}(R, 3)$. El resultado se muestra en la figura 4.12.

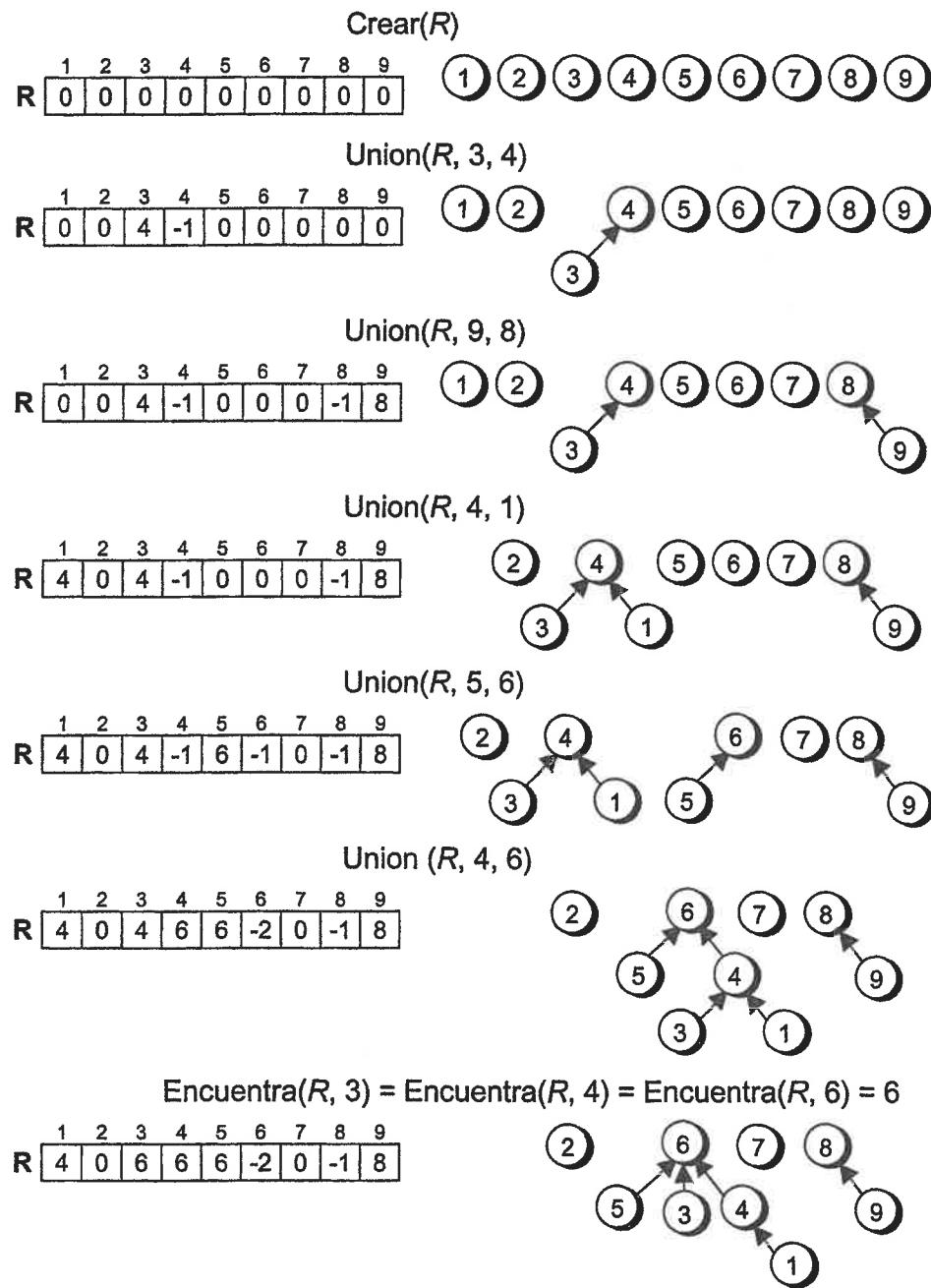


Figura 4.12: Evolución de la estructura de relaciones de equivalencia, para las operaciones del ejemplo 4.1, usando la representación de punteros al padre, con equilibrado y compresión de caminos.

Como cabía esperar, la profundidad de los árboles obtenidos es menor aplicando

equilibrado y compresión de camino. Esto significa un menor tiempo de ejecución para las operaciones del tipo.

Se puede ver que, en el ejemplo, la operación $\text{Encuentra}(R, 1, 6)$ ha sido sustituida por $\text{Encuentra}(R, 4, 6)$. Recordemos que la operación Encuentra requiere que se aplique sobre raíces de los árboles. Mientras que el nodo 1 es raíz en el ejemplo 4.1, en el ejemplo 4.2 no lo es. En su lugar, usamos la clase del nodo 1, es decir 4.

Evaluación de la eficiencia

Es evidente que la estructura utilizada para representar relaciones de equivalencia necesita una cantidad de memoria proporcional al tamaño N del conjunto sobre el que se aplica la relación. En el caso de las imágenes, por ejemplo, la memoria podría ser tanta como la utilizada por la imagen en sí. No obstante, este uso es razonable teniendo en cuenta que la relación no puede ser calculada mediante una fórmula.

En lo que se refiere al tiempo de ejecución, tenemos por un lado la operación Union que ejecuta siempre como máximo un número fijo de instrucciones, con lo que su tiempo es un $O(1)$. Por otro lado, el tiempo de la operación Encuentra es proporcional al nivel del nodo buscado en el árbol, como ya hemos visto. El tiempo necesario dependerá del caso.

Mejor caso. Si se aplican las operaciones Union de forma adecuada, el árbol no crece en profundidad más que una cantidad fija y reducida. En esta situación el tiempo de Encuentra sería un $O(1)$.

Peor caso. El peor caso se dará cuando el árbol crezca en profundidad lo más rápidamente posible. Teniendo en cuenta la estructura del procedimiento Union , el árbol sólo crecerá de nivel cuando los dos árboles unidos tengan la misma altura. Se puede demostrar por inducción¹¹ que un árbol de profundidad h tiene como mínimo 2^h nodos. Por lo tanto, para N nodos la profundidad máxima sería $\log_2 N$, y el tiempo de ejecución de Encuentra en el peor caso es un $O(\log N)$.

Caso promedio. El tiempo promedio de Encuentra , incluyendo equilibrado y compresión de caminos, resulta difícil de calcular. Se puede ver intuitivamente que el tiempo será mucho menor que un tiempo logarítmico. En término medio, será muy difícil que el árbol alcance profundidad 4. Teniendo en cuenta que el árbol de profundidad 5 se forma uniendo dos árboles de profundidad 4, será mucho más difícil obtener el árbol de profundidad 5, y así sucesivamente.

En concreto, el tiempo de ejecución de Encuentra en el caso promedio es un $O(\alpha(N))$; donde $\alpha(N)$ es una función de crecimiento muy lento, situado entre un $O(1)$ y un $O(\log n)$. La función $\alpha(N)$ se define como la pseudo-inversa de una función $A(x)$ de crecimiento muy rápido, conocida como **función de Ackerman**. Por ejemplo, los valores iniciales de esta función son: $A(1) = 2$, $A(2) = 2^2$, $A(3) = 2^{2^2}$, $A(4) = 2^{2^{2^2}}$ donde los puntos significan repetir la operación de elevación 65.536 veces. $A(5)$ es un valor tan grande que no existe una notación estándar para expresarlo de forma expandida. A efectos prácticos, el tiempo promedio de Encuentra se puede considerar casi un $O(1)$.

¹¹Base de la inducción: un árbol con altura 1 tiene como mínimo 2 nodos. Paso de inducción: suponiendo que los árboles de altura $h - 1$ tienen como mínimo 2^{h-1} nodos, un árbol de altura h tendrá como mínimo 2^h nodos. La demostración se deja como ejercicio. Tener en cuenta que un árbol de altura h se forma uniendo dos árboles de altura $h - 1$.

4.3. Árboles de búsqueda balanceados

Un clase muy importante de árboles son los llamados **árboles de búsqueda**. En un árbol de búsqueda, la relación de orden definida entre las claves determina el lugar donde deben colocarse los nodos en el árbol. Son, por lo tanto, estructuras de datos adecuadas para la representación de conjuntos y diccionarios, que permiten localizar una clave en un tiempo logarítmico. El acceso secuencial –para operaciones como listar en orden, buscar el mínimo, el máximo, etc.– también es posible y eficiente.

Existen varios tipos de árboles de búsqueda. En esta sección haremos un repaso de los árboles binarios de búsqueda, viendo la necesidad de incluir condiciones de balanceo. En la siguiente sección veremos árboles de búsqueda no binarios.

4.3.1. Árboles binarios de búsqueda

Los **árboles binarios de búsqueda** (ABB) son el tipo más sencillo de árbol de búsqueda. Se supone que el lector está más o menos familiarizado con los ABB, así que nos limitaremos a hacer un rápido repaso.

Definición 4.3 Un **árbol binario de búsqueda** es un árbol en el cual:

- Cada nodo contiene una clave. Entre las claves existe una relación de orden (más exactamente, una relación de orden completo). Normalmente hablamos de “el valor de un nodo” refiriéndonos a “el valor de la clave almacenada en un nodo”.
- Cada nodo tiene como mínimo cero hijos (en cuyo caso lo denominamos una **hoja**) y como máximo dos (que denominaremos **hijo izquierdo** e **hijo derecho**).
- Para todo nodo x del árbol, el valor del hijo izquierdo de x (si existe) es menor que el valor de x , y el del hijo derecho de x (si existe) es mayor que el valor de x .

La tercera condición es la que define la colocación de los elementos en el árbol, y es la que permite localizar una clave en un tiempo logarítmico, en promedio. La definición del tipo de datos ABB podría ser como las siguientes, suponiendo que sólo almacenamos claves o que almacenamos también valores asociados a claves.

tipo

```

ArbolBB[T] = registro
    clave: T
    izq, der: Puntero[ArbolBB[T]]
finregistro
ArbolBB2[tclave, tvalor] = registro
    clave: tclave
    valor: tvalor
    izq, der: Puntero[ArbolBB2[T]]
finregistro

```

En la figura 4.13 se muestran dos ejemplos de árboles binarios de búsqueda del tipo **ArbolBB[entero]**, usando la primera definición.

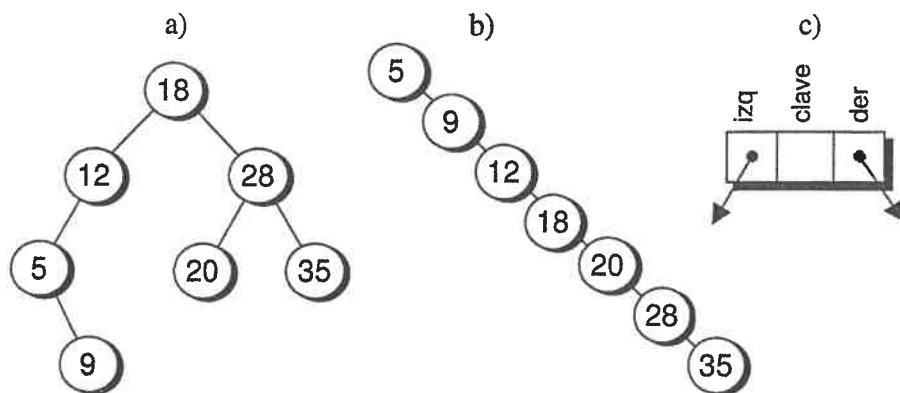


Figura 4.13: Árboles binarios de búsqueda sin condición de balanceo.

El recorrido secuencial de todas las claves se puede realizar en un $O(n)$, siendo n el número de nodos del árbol. Para ello, bastaría con un simple recorrido en inorder del árbol. Por otro lado, la localización de una clave concreta se consigue mediante una sencilla búsqueda que empieza en la raíz y desciende recursivamente hacia la rama correspondiente, haciendo uso de la relación de orden. Usando la segunda definición del tipo, la implementación sería como la siguiente.

operación Consulta (*a*: Puntero[ArbolBB2[tclave,tvalor]]; *c*: tclave): tvalor
si *a* = NULO **entonces**
 devolver NULO
sino si *a*.clave = *c* **entonces**
 devolver *a*.valor
sino si *a*.clave < *c* **entonces**
 devolver Consulta(*a*.der, *c*)
sino
 devolver Consulta(*a*.izq, *c*)
finsi

Considerando el funcionamiento del algoritmo, está claro que el tiempo de ejecución es proporcional al nivel de la clave buscada dentro del árbol. En el mejor caso, un árbol binario de altura h tiene $2^{h+1} - 1$ nodos. Inversamente¹², un árbol con n nodos tendrá en el mejor caso una altura $\lceil \log_2(n+1) - 1 \rceil \in O(\log n)$. Este es el mejor caso, pero ¿qué ocurre en el peor caso?

El árbol de la figura 4.13b) es un ejemplo de peor caso de ABB. En este árbol cada nodo sólo tiene un descendiente, por lo que la profundidad del árbol es $n - 1$. El tiempo de ejecución del procedimiento Consulta será un $O(n)$ en ese caso, igual que si usáramos una lista enlazada.

¹²Es decir, despejando h de la ecuación: $2^{h+1} - 1 = n$.

Árboles binarios de búsqueda perfectamente balanceados

Aunque el ejemplo anterior de peor caso de ABB parece una situación un tanto extrema, es evidente la degradación de la eficiencia que ocurrirá cuando se den situaciones parecidas, donde el árbol presenta una estructura desequilibrada en algunas de las ramas. Casos como el anterior podrán ocurrir con frecuencia, siempre que al insertar las claves estas vengan en orden, ya sea creciente o decreciente.

Este problema implica la necesidad de garantizar, de alguna manera, que el árbol tendrá una estructura balanceada. Los ABB balanceados surgen de imponer restricciones de balanceo sobre los nodos del árbol. Si al hacer una inserción, o una eliminación, no se cumplen las condiciones impuestas, será necesario reestructurar el árbol hasta que se cumplan.

Una primera posibilidad es definir una condición de balanceo basada en el número de nodos de los subárboles. Esta condición da lugar a los ABB perfectamente balanceados.

Definición 4.4 Un ABB se dice que está **perfectamente balanceado** si, para todos los nodos de árbol, el número de nodos de su subárbol izquierdo difiere como máximo en uno del número de nodos de su subárbol derecho.

Por ejemplo, en el árbol de la figura 4.13a) todos los nodos cumplen la condición, excepto el 12. Para ese nodo, el subárbol izquierdo tiene 2 nodos y el derecho 0. Por lo tanto, el árbol no está perfectamente balanceado. En la figura 4.14 se muestran algunos ejemplos de ABB perfectamente balanceados.

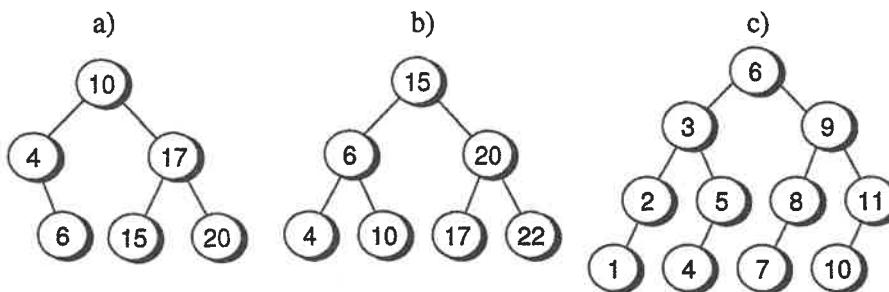


Figura 4.14: Árboles binarios de búsqueda perfectamente balanceados.

Se puede comprobar fácilmente que un ABB perfectamente balanceado con n nodos siempre tendrá altura $\lceil \log_2(n + 1) - 1 \rceil$. En cualquier caso, el tiempo de ejecución del procedimiento Consulta, de búsqueda de una clave, será $O(\log n)$. Pero mantener esta condición de balanceo también implica un coste, que afectará fundamentalmente a la operación de inserción en el árbol.

Supongamos que en el ABB perfectamente balanceado de la figura 4.14a) insertamos la clave 22. El nodo debería ir a la derecha del 20, por lo que la raíz del árbol se desbalancea y habría que reequilibrar el árbol. ¿Cuál debería ser el árbol resultante? Sólo existe un ABB perfectamente balanceado y que contenga las claves del árbol de la figura 4.14a)

más la 22; el árbol resultante es el de la figura 4.14b). Se puede ver que en el nuevo árbol todos los elementos han cambiado de posición. En términos de implementación, todos los punteros de los nodos cambian de valor dando como resultado una estructura completamente distinta. En consecuencia, el procedimiento debería ejecutar como mínimo n pasos, y de hecho se puede encontrar un algoritmo para reestructurar el árbol en $O(n)$.

A efectos prácticos, el funcionamiento de un ABB perfectamente balanceado sería equivalente a usar un array ordenado, donde se puede hacer una búsqueda binaria en $O(\log n)$ y una inserción en $O(n)$, buscando la posición que debe ocupar la clave y después desplazando a la derecha en el array los valores mayores.

Árboles binarios de búsqueda balanceados o AVL

La conclusión, tras el estudio de los ABB perfectamente balanceados, es que la condición de balanceo impuesta da lugar a árboles muy bien equilibrados; pero mantener esa condición supone un coste tan elevado que se echan a perder las ventajas de la representación. La solución, como casi siempre, está en el término medio: imponer una condición de balanceo pero menos restrictiva que la de balanceo perfecto.

Esta condición de balanceo más débil se basa en la diferencia de alturas de los subárboles, en lugar de en la diferencia del número de nodos. Los árboles resultantes se conocen como **árboles AVL**, en honor a sus creadores, los matemáticos rusos G. M. Adelson-Velskii e Y. M. Landis, en 1962.

Definición 4.5 Un ABB se dice que está **balanceado**, o que es un **AVL**, si para cualquier nodo del árbol la altura de su subárbol izquierdo difiere como máximo en uno de la altura de su subárbol derecho.

Por definición, decimos que un árbol vacío (sin nodos) tiene altura -1, mientras que un árbol con un solo nodo tiene altura 0. Podemos decir, por ejemplo, que una hoja es un nodo que tiene dos subárboles de altura -1.

Se puede comprobar que la condición de balanceo AVL es más débil que la de balanceo perfecto, en el sentido de que cualquier árbol perfectamente balanceado es también un árbol AVL. Sin embargo, la implicación contraria no ocurre. Por ejemplo, los árboles de las figuras 4.15b) y 4.15c) son AVL pero no cumplen la condición de balanceo perfecto.

Error frecuente. La definición de árbol AVL requiere que la condición de balanceo se cumpla en **todos los nodos de árbol**, y no sólo en la raíz. Por ejemplo, en el árbol de la figura 4.15d) se cumple la condición para la raíz, pero no para el nodo 1, que tiene un subárbol de altura -1 y otro de altura 1. Por lo tanto, ese árbol no es un AVL.

El funcionamiento de los árboles AVL es exactamente el mismo que el de los ABB, con el añadido de tener que garantizar la condición de balanceo en las operaciones de modificación, es decir la inserción y la eliminación. Si al aplicar alguna de estas operaciones la diferencia de alturas de algún nodo se hace mayor que uno, entonces será necesario rebalancear el árbol.

4.3.2. El peor caso de árbol AVL

Antes de estudiar la implementación de las operaciones de inserción y eliminación, vamos a analizar el *peor caso* de árbol AVL. Entendemos por peor caso la situación de

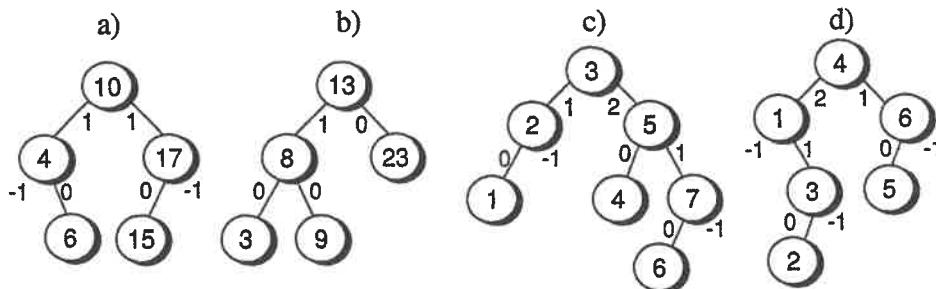


Figura 4.15: Árboles binarios de búsqueda y AVL. Debajo de cada nodo (excepto en las hojas) se muestran las alturas de los subárboles. a) Árbol perfectamente balanceado y AVL. b) Árboles AVL pero no perfectamente balanceados. d) Árbol no AVL ni perfectamente balanceado.

máximo desequilibrio permitido para un AVL en cuanto a alturas, es decir cuando para todos los nodos no hoja sus subárboles tienen alturas distintas.

El interés de estudiar este peor caso reside en que nos sirve para determinar cuánto puede crecer como máximo la altura de un árbol AVL para un número de nodos determinado. De esta forma, podemos establecer una cota superior para el tiempo de ejecución de las operaciones de búsqueda; recordemos que el tiempo es proporcional al nivel de la clave buscada y, a su vez, la altura indica el máximo nivel de un nodo del árbol.

El análisis del caso más desfavorable de AVL se puede hacer por inducción, estudiando lo que ocurre con altura $h = 0, 1, 2, 3, \dots$. Sea $N(h)$ una función que indica el **número de nodos de un árbol AVL de altura h** en el peor caso. Esta $N(h)$ es, al mismo tiempo, el mínimo número de nodos del AVL para altura h . Empezando con valores de h pequeños tenemos:

- Para altura $h = 0$, el AVL en el peor caso contiene un sólo nodo, luego $N(0) = 1$.
- Para altura $h = 1$, el árbol sería como el mostrado en la figura 4.16b), $N(1) = 2$.
- Para altura $h = 2$, el árbol sería como el mostrado en la figura 4.16c), $N(2) = 4$.

De esta forma, vemos que en el caso general el peor AVL de altura h sería un árbol donde la raíz tiene un subárbol de altura $h - 1$ y otro subárbol de altura $h - 2$. Ambos subárboles serían, recursivamente, los peores casos de AVL para las alturas correspondientes. Por lo tanto, el número de nodos sería: $N(h) = N(h - 1) + N(h - 2) + 1$, donde el “+1” corresponde a la raíz del árbol. Incluyendo los casos base, tenemos la siguiente fórmula recursiva:

$$N(h) = \begin{cases} 1 & \text{Si } h = 0 \\ 2 & \text{Si } h = 1 \\ N(h - 1) + N(h - 2) + 1 & \text{Si } h > 1 \end{cases} \quad (4.1)$$

Debido a la similitud de esta fórmula con los números de Fibonacci ($f(x) = f(x -$

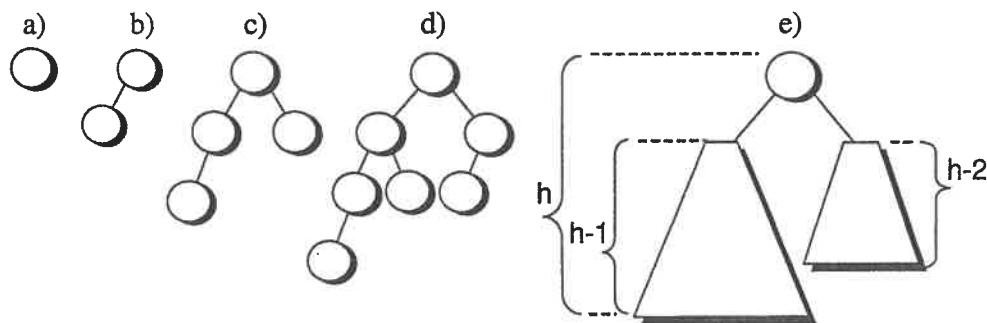


Figura 4.16: Peor caso de árbol AVL, según la altura h . a) $h = 0$. b) $h = 1$. c) $h = 2$. d) $h = 3$. e) Caso general, para un $h > 3$ cualquiera.

$1) + f(x - 2)$) los árboles AVL son también conocidos como **árboles de Fibonacci**¹³. En el capítulo 8 se estudia un conjunto de técnicas matemáticas para resolver ecuaciones recurrentes de este tipo. Por el momento, nos limitaremos a dar el resultado. Despejando la recurrencia anterior tenemos:

$$N(h) = \left(1 + \frac{2}{\sqrt{5}}\right) \left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right) \left(\frac{1-\sqrt{5}}{2}\right)^h - 1$$

$$N(h) \approx 1,89 * 1,62^h + 0,11 * (-0,62)^h - 1 \quad (4.2)$$

Realmente, lo que nos interesa sería la función “inversa”, es decir una función que nos diga cuánto vale la altura h para un número de nodos dado n . Para ello, tendríamos que despejar h de la fórmula anterior. Vamos a hacerlo de forma aproximada.

Para valores suficientemente grandes, el término $0,11 * (-0,62)^h$ tiende a 0; por ejemplo, para $h = 14$ vale 0,0001. Así que tendríamos: $n \approx 1,89 * 1,62^h - 1$. Por lo tanto, podemos despejar h de forma aproximada:

$$h \approx \log_{1,62} \left(\frac{n+1}{1,89} \right) \quad (4.3)$$

Por lo tanto, desechando las constantes multiplicativas tenemos que el crecimiento máximo de la altura h pertenece a un $O(\log n)$, siendo n el número de nodos del árbol. Es decir, en el peor caso la altura del árbol crece logarítmicamente y, en consecuencia, el tiempo de ejecución de las operaciones de búsqueda está también en $O(\log n)$.

En las secciones siguientes veremos cómo podemos conseguir también implementaciones para las operaciones de inserción y eliminación con una complejidad logarítmica.

¹³Aunque, particularmente, preferimos el nombre AVL que atribuye el mérito a sus verdaderos inventores, y no al célebre matemático italiano del siglo XIII.

4.3.3. Rotaciones simples y dobles sobre AVL

Los algoritmos para la inserción y eliminación en AVL hacen uso de un tipo de operaciones elementales conocidas como **rotaciones**. Una rotación en un ABB es una operación que modifica en parte la estructura del árbol, pero sigue manteniendo la propiedad fundamental del ABB: para cada nodo x , el subárbol izquierdo contiene valores menores que x y el derecho valores mayores que x .

Rotaciones simples

Existen cuatro tipos de rotaciones estándar, que se distinguen en simples y dobles, y a la derecha o a la izquierda. En una **rotación simple a la derecha** de un nodo A ($RSD(A)$): el hijo derecho de A pasa a ser la nueva raíz B (del árbol o subárbol), el nodo A “baja” como nuevo hijo izquierdo de B , y A toma como hijo izquierdo el anterior hijo derecho de B . En la figura 4.17 se muestra gráficamente la operación de $RSD(A)$.

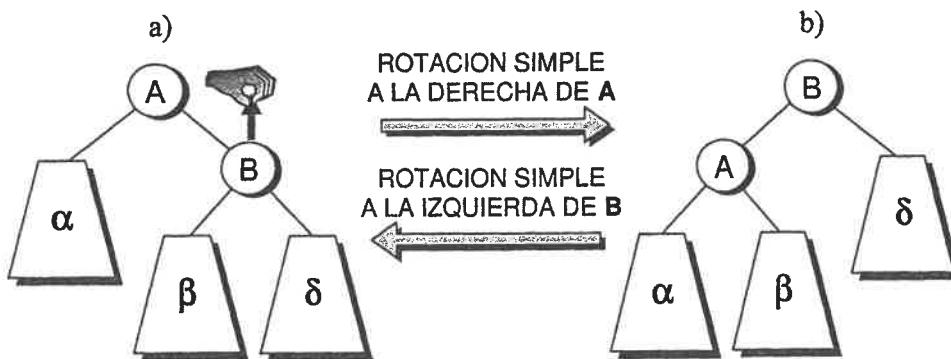


Figura 4.17: Rotación simple en un ABB. Rotación simple a la derecha de A : de a) a b). Rotación simple a la izquierda de B : de b) a a).

La **rotación simple a la izquierda** de un nodo A ($RSI(A)$) es la operación simétrica a la $RSD(A)$. En este caso, es el hijo izquierdo de A el que “sube” como la nueva raíz. Si nos fijamos en la figura 4.17, el árbol antes de aplicar la rotación sería 4.17b) y después de aplicar una $RSI(B)$ el resultado sería 4.17a).

La implementación de la operación consistiría en una simple serie de asignaciones de punteros, que se realizan de forma fija. Usaremos la siguiente definición del tipo `ArbolAVL[T]`, que almacena en cada nodo la altura del subárbol correspondiente.

tipo

`ArbolAVL[T] = Puntero[NodoAVL[T]]`

`NodoAVL[T] = registro`

 clave: T

 altura: entero

 izq, der: ArbolAVL[T]

`finregistro`

Las operaciones de rotación serían las siguientes. Hacemos uso de una función `Altura` que devuelve la altura del árbol o -1 si el árbol es vacío.

```

operación Altura ( $A$ : ArbolAVL[T]): entero
  si  $A = \text{NULO}$  entonces
    devolver -1
  sino
    devolver  $A^{\uparrow}.\text{altura}$ 
  finsi
operación RSD (var  $A$ : ArbolAVL[T])
   $B := A^{\uparrow}.\text{der}$ 
   $A^{\uparrow}.\text{der} := B^{\uparrow}.\text{izq}$ 
   $B^{\uparrow}.\text{izq} := A$ 
   $A^{\uparrow}.\text{altura} := 1 + \max(\text{Altura}(A^{\uparrow}.\text{izq}), \text{Altura}(A^{\uparrow}.\text{der}))$ 
   $B^{\uparrow}.\text{altura} := 1 + \max(A^{\uparrow}.\text{altura}, \text{Altura}(B^{\uparrow}.\text{der}))$ 
   $A := B$ 
operación RSI (var  $B$ : ArbolAVL[T])
   $A := B^{\uparrow}.\text{izq}$ 
   $B^{\uparrow}.\text{izq} := A^{\uparrow}.\text{der}$ 
   $A^{\uparrow}.\text{der} := B$ 
   $B^{\uparrow}.\text{altura} := 1 + \max(\text{Altura}(B^{\uparrow}.\text{izq}), \text{Altura}(B^{\uparrow}.\text{der}))$ 
   $A^{\uparrow}.\text{altura} := 1 + \max(B^{\uparrow}.\text{altura}, \text{Altura}(A^{\uparrow}.\text{izq}))$ 
   $B := A$ 

```

Hay que hacer las siguientes consideraciones importantes.

- Obviamente, el tiempo de ejecución de las rotaciones es constante, $O(1)$.
- Después de aplicar una rotación sobre un ABB, el árbol resultante sigue siendo un ABB. Esto se puede comprobar fácilmente, viendo que las relaciones implicadas en el árbol 4.17a) y 4.17b) son exactamente las mismas.
- El resultado anterior no implica que al aplicar la rotación sobre un AVL se obtenga un AVL. Las rotaciones se usan para equilibrar árboles que están desbalanceados, pero si se aplica sobre uno balanceado puede desbalancearlo.
- Las rotaciones se pueden aplicar sobre la raíz y sobre otros nodos que no sean la raíz.

Rotaciones dobles

En algunos casos de desbalanceo no basta con una rotación simple y es necesario aplicar una **rotación doble**. Una rotación doble consiste en dos rotaciones simples. En concreto, la **rotación doble a la derecha** de un nodo A ($RDD(A)$) se obtiene aplicando una RSI del subárbol derecho de A , seguida de una $RSD(A)$. El resultado se muestra gráficamente en la figura 4.18. Se puede interpretar el efecto de la rotación doble como “tirar del nodo C hacia arriba”. Ese nodo pasa a ser la nueva raíz y los nodos A y B “caen” como hijos de C .

De forma simétrica, la **rotación doble a la izquierda** de A se consigue aplicando una $RSD(A.\text{izq})$ y a continuación una $RSI(A)$. La implementación de ambas rotaciones dobles es inmediata.

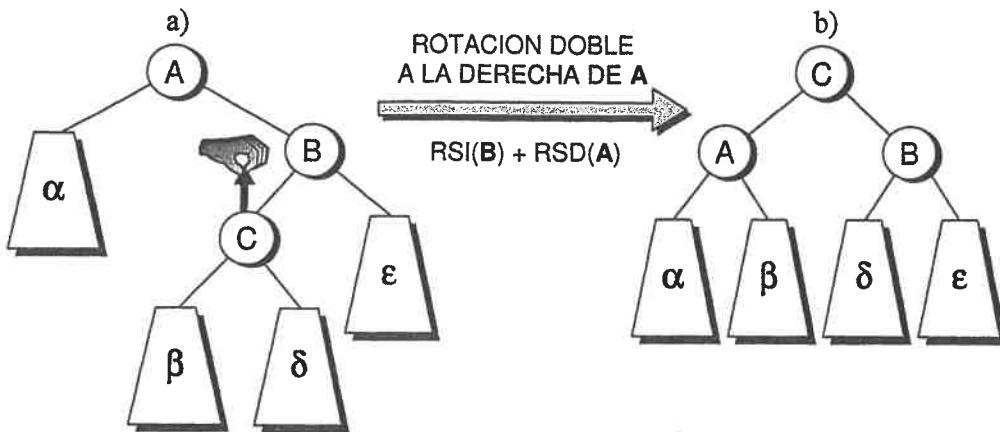


Figura 4.18: Rotación doble a la derecha en un ABB. a) Antes de la rotación. b) Despues de $RDD(A)$.

operación RDD (var A: ArbolAVL[T])

 RSI($A \uparrow .der$)

 RSD(A)

operación RDI (var A: ArbolAVL[T])

 RSD($A \uparrow .izq$)

 RSI(A)

Podemos hacer las mismas consideraciones que antes: el tiempo de aplicar las rotaciones dobles es constante, ambas mantienen la estructura de ABB, pero no necesariamente producen un AVL partiendo de un AVL.

4.3.4. Inserción en un árbol AVL

Básicamente, la inserción de un nuevo elemento en un AVL es equivalente a una inserción en un ABB normal, seguida de una comprobación de la condición de balanceo y equilibrado del árbol en caso de no cumplirse. La condición de balanceo no debe comprobarse en todos los nodos del árbol –lo cual requeriría como mínimo un $O(n)$ para un árbol con n nodos– sino sólo en los antecesores del nodo donde se ha hecho la inserción. Si se utiliza un procedimiento recursivo para la inserción, es fácil hacer estas comprobaciones en los antecesores a la vuelta de la recursividad.

Entrando un poco más en detalle, el algoritmo para la inserción de una clave x en un árbol AVL se puede resumir en los siguientes pasos.

1. Utilizar un procedimiento recursivo para insertar x . Primero se busca la hoja donde debe ir x y después se inserta. Si ya existe la clave x , no se hace nada.
2. A la vuelta del procedimiento recursivo, por cada nodo recorrido calcular las alturas de sus dos subárboles y comprobar si difieren en más de uno. En caso contrario, recalcular la altura del subárbol actual y volver de la recursividad.

3. Si la diferencia de alturas es mayor que uno, será necesario rebalancear el árbol aplicando las rotaciones adecuadas. Después se recalcula la altura del nodo actual y se finaliza la llamada recursiva.

Casos de desbalanceo en la inserción

La clave se encuentra ahora en el paso 3, donde se debe decidir qué rotación aplicar para rebalancear el árbol. ¿Cuántas rotaciones deben hacerse, de qué tipo y sobre qué nodos? Afortunadamente, se puede comprobar mediante un simple análisis de casos que todas las posibles situaciones de balanceo se pueden resumir en un número reducido de posibilidades. Cada una de ellas tiene asociada una rotación –a modo de regla fija o receta mágica– que las soluciona sin necesidad de comprobar lo que haya por debajo.

¿Cuáles son estas posibilidades de balanceo? Supongamos que el nodo que ha detectado el desbalanceo es A y que la inserción que lo ha provocado ha ocurrido en el subárbol izquierdo de A , que llamaremos B . Podemos distinguir, a su vez, dos casos: que la inserción sea en el subárbol izquierdo de B o en el subárbol derecho de B . El primer caso es llamado también **Caso II** (la clave siguió la ruta izquierda-izquierda), y el segundo **Caso ID** (la ruta fue izquierda-derecha).

En la figura 4.19 se muestran estos dos casos de desbalanceo y las rotaciones que hay que aplicar para reequilibrar el árbol. El caso II, detectado en un nodo A , se soluciona con una $RSI(A)$; y el caso ID, detectado en A , se rebalancea con una $RDI(A)$. Existirían dos casos simétricos, que surgen cuando el desbalanceo aparece en el subárbol derecho de A : el **Caso DD**, que se arregla con una $RSD(A)$; y el **Caso DI**, para el cual se debe aplicar una $RDD(A)$.

La demostración de que estas rotaciones *efectivamente* arreglan el desbalanceo se hace mediante lo que podemos denominar el **cálculo de alturas**, como aparece en la figura 4.19. El proceso de cálculo de alturas consiste en dos pasos: primero se analiza qué altura debe tener cada subárbol (de forma genérica) en el árbol inicial, suponiendo que la altura total es un h cualquiera; después se trasladan esas alturas al árbol resultante de hacer las rotaciones, y se comprueba que para todos los nodos se cumple la condición de AVL. Este proceso verifica que se cumple la condición de balanceo para los nodos que aparecen de forma explícita (A , B , C , etc.) y puesto que los subárboles (α , β , δ , etc.) no se modifican, también se cumplirá la condición en sus nodos. En definitiva, el árbol resultante es un AVL, siempre que el árbol previo a la inserción fuera un AVL.

Se puede observar un curioso hecho: después de aplicar cualquiera de los casos de rotación, la altura final del árbol resultante es h , la misma que antes de hacer la inserción. Esto implica que, en una inserción, sólo será necesario hacer como máximo un rebalanceo del árbol. Cuando se haga el primer rebalanceo, la altura del árbol no se modificará, por lo que no aparecerán más desequilibrios.

Implementación de la inserción en AVL

Supongamos la definición de los tipos del apartado 4.3.3. Como ya hemos estudiado, la operación de inserción en un AVL debe realizar primero una inserción recursiva de la clave en el árbol. Después, a la vuelta de la recursividad, se comprueban las condiciones de balanceo y los casos de rotación. La implementación podría ser como la siguiente.

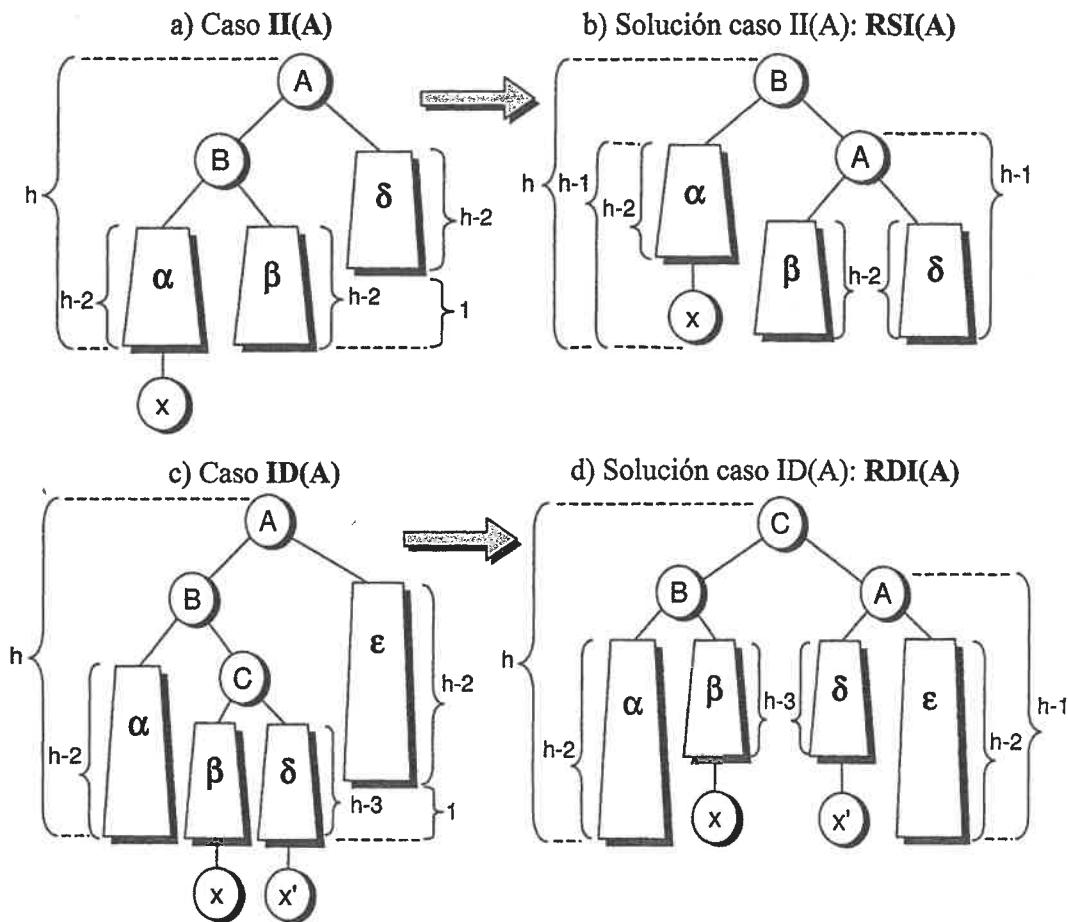


Figura 4.19: Casos de desbalanceo en un AVL, tras la inserción de una clave x , y las rotaciones aplicadas para solucionarlo. En el caso ID, la clave insertada puede ser x o x' . El caso DD sería simétrico a II, y DI simétrico a ID.

```

operación Inserta (var A: ArbolAVL[T]; x: T)
    si A = NULO entonces
        A:= Nuevo(NodoAVL[T])
        A↑.clave:= x
        A↑.izq:= NULO
        A↑.der:= NULO
        A↑.altura:= 0
    sino
        si x < A↑.clave entonces      // Inserción en subárbol Izquierdo
            Inserta(A↑.izq, x)
            si Altura(A↑.izq)-Altura(A↑.der) > 1 entonces
                si x < A↑.izq↑.clave entonces
                    RSI(A)      // Caso II

```

```

sino
    RDI( $A$ ) // Caso ID
finsi
sino
     $A^{\uparrow}.\text{altura} := 1 + \max(\text{Altura}(A^{\uparrow}.\text{izq}), \text{Altura}(A^{\uparrow}.\text{der}))$ 
finsi
sino si  $x > A^{\uparrow}.\text{clave entonces}$  // Inserción en subárbol Derecho
    Inserta( $A^{\uparrow}.\text{der}, x$ )
    .... // Caso simétrico al anterior, para el subárbol derecho
finsi // Si  $x = A^{\uparrow}.\text{clave}$ , el elemento ya está en el árbol, no hacer nada

```

En lo que se refiere al tiempo de ejecución, el algoritmo anterior realizará tantas llamadas recursivas como el nivel en el que se deba insertar la clave dentro del árbol. Ya hemos visto que la profundidad máxima del árbol está en un $O(\log n)$, siendo n el número de nodos del AVL. Además, está claro que en cada llamada recursiva se ejecutará un número de operaciones limitado por una constante (recordemos que el tiempo de Altura, RSD y RDD es un $O(1)$). En conclusión, el tiempo de ejecución de Inserta es un $O(\log n)$.

4.3.5. Eliminación en un árbol AVL

Conceptualmente, el funcionamiento de la operación de eliminación en un AVL es muy parecido a la inserción. En primer lugar hacemos la eliminación como en un ABB normal, usando un procedimiento recursivo, y después vamos comprobando las condiciones de balanceo a la vuelta de la recursividad, en los antecesores de la posición eliminada. Igual que antes, podemos hacer un análisis de casos para las situaciones de desbalanceo en la eliminación; y cada una de ellas tendrá asociada una rotación para solucionarla.

Al eliminar un elemento de un ABB hay que tener en cuenta si se encuentra en una hoja o no. En el primer caso, se puede suprimir la entrada de forma directa. En el segundo caso, hay que tener cuidado de no desconectar los hijos del nodo eliminado. La estructura del algoritmo de eliminación de una clave x en un AVL sería la siguiente:

1. Buscar la clave x en el AVL, utilizando un procedimiento recursivo. Si no se encuentra, acabar sin modificar el árbol.
2. Si x se encuentra en una hoja, suprimir la hoja y continuar en el paso 5 a partir del parente de x en el árbol.
3. Si x no es una hoja, entonces si tiene un solo hijo conectar el parente de x con el hijo de x , como se muestra en la figura 4.20a). Continuar en el paso 5 a partir del parente de x .
4. En otro caso, si x tiene dos hijos, colocar en la posición de x el siguiente elemento en orden (el mayor de su subárbol izquierdo, o el menor de su subárbol derecho), como aparece en la figura 4.20b). Suprimir el nodo de la posición intercambiada con x y seguir en el paso 5 como si la eliminación se hubiera producido en esa posición.
5. A la vuelta de la recursividad, por cada nodo recorrido calcular las alturas de sus dos subárboles y comprobar si difieren en más de uno. En caso contrario, recalcular

la altura del subárbol actual y volver de la recursividad.

6. Si la diferencia de alturas es mayor que uno, será necesario rebalancear el árbol. Comprobar el caso que ocurre y aplicar la rotación correspondiente. Después recalcular la altura del nodo actual y finalizar la llamada recursiva.

A diferencia de la inserción, los casos de desbalanceo en la eliminación no dependen de los recorridos (II, ID, DI, DD) sino de las alturas en el subárbol opuesto al de la eliminación. Supongamos que se suprime la clave x de un AVL y en el punto 6 del algoritmo se detecta desbalanceo en un nodo A . Si x se encontraba en el subárbol izquierdo de A , entonces para saber qué rotación aplicar debemos fijarnos en el subárbol derecho de A . Más concretamente, los casos dependen de las alturas h_1 y h_2 de los subárboles izquierdo y derecho, respectivamente, del subárbol derecho de A . En la figura 4.20c) se muestran estas alturas en el árbol.

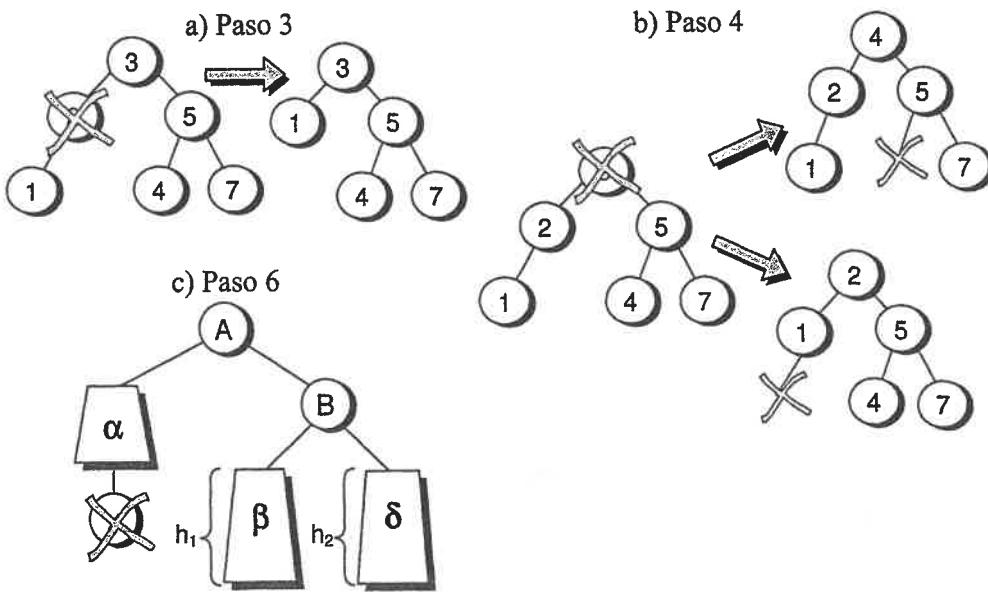


Figura 4.20: Eliminación de una clave en un AVL. a) Paso 3: eliminación de un nodo que sólo tiene un hijo. b) Paso 4: eliminación de un nodo con dos hijos. c) Paso 6: eliminación con desbalanceo.

Según la relación entre h_1 y h_2 podemos encontrar tres casos de desbalanceo en la eliminación.

- Si $h_1 = h_2$, entonces el desbalanceo se soluciona aplicando una $RSD(A)$. Además, se puede comprobar que la altura del árbol no cambia, respecto a antes de suprimir el elemento.
- Si $h_1 < h_2$, se aplica también una $RSD(A)$. No obstante, en este caso la altura del árbol sí que disminuye en uno. Esto implica que puede ser necesaria más de una rotación a distintos niveles, al contrario de lo que ocurría con la inserción.

- Si $h_1 > h_2$, entonces hay que aplicar una $RDD(A)$. Como en el caso anterior, la altura del árbol disminuye en uno.

En la figura 4.21 se muestra el cálculo de alturas para demostrar que el anterior análisis de casos resuelve todas las posibles situaciones de desbalanceo en la supresión. Es decir, las rotaciones producen árboles AVL suponiendo que el árbol de partida (el árbol antes de hacer la eliminación) era también un AVL.

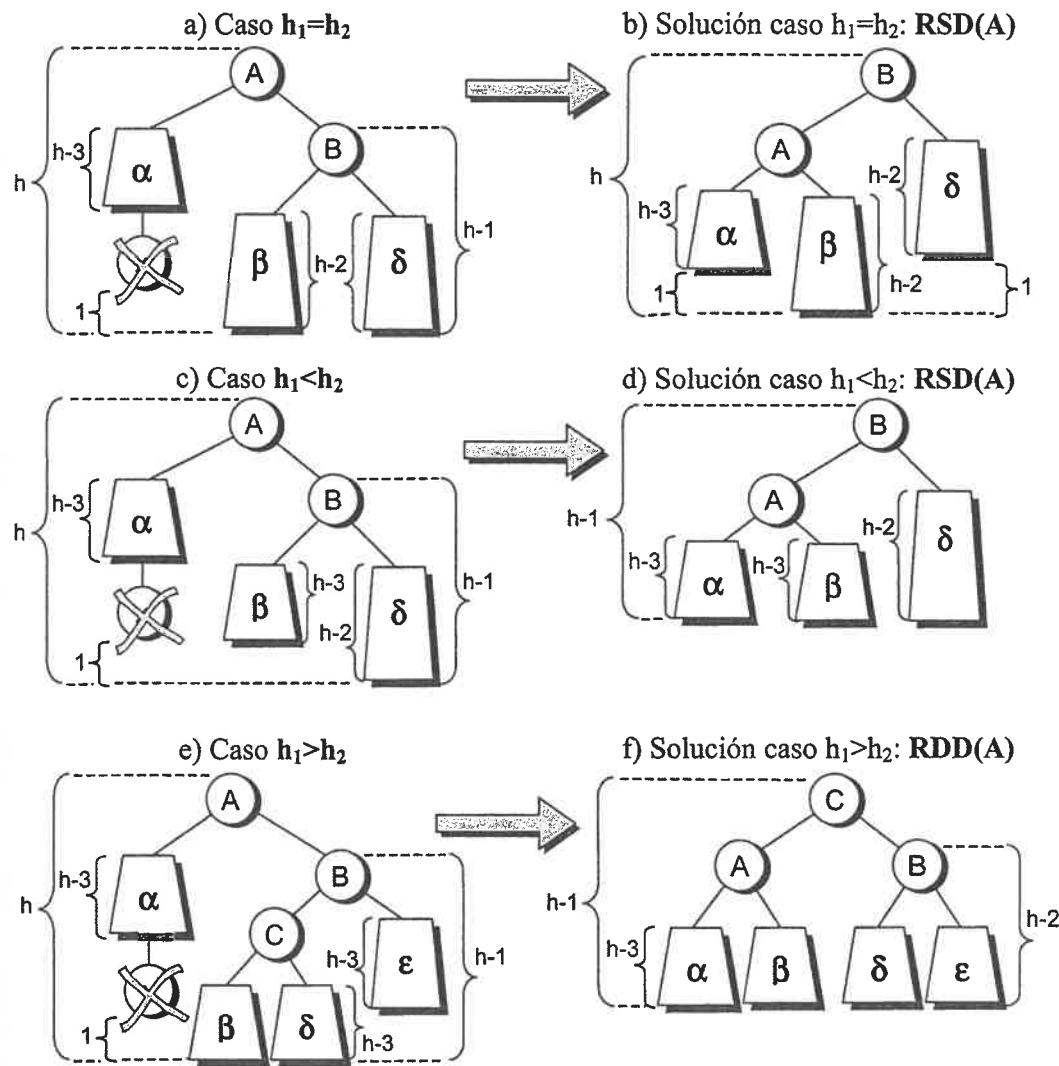


Figura 4.21: Casos de desbalanceo en un AVL, tras la eliminación de una clave x , y las rotaciones aplicadas para solucionarlo. Los casos de eliminación en el subárbol derecho serían simétricos a los anteriores.

La implementación de la operación de eliminación en un AVL sería parecida a la de inserción. El procedimiento sería un poco más largo, ya que hay que comprobar si el nodo es una hoja o no, así que se deja como ejercicio.

En cuanto al tiempo de ejecución, básicamente podemos aplicar el mismo análisis que en la inserción. El tiempo total para una supresión será proporcional al nivel de la clave buscada en el AVL, puesto que el tiempo en cada nivel es constante (o limitado por una constante). Como el nivel máximo está en un $O(\log n)$, el tiempo de ejecución será también un $O(\log n)$.

Ejemplo 4.3 En un árbol AVL inicialmente vacío almacenamos números enteros. Insertamos los siguientes elementos: 7, 14, 32, 9, 40, 8, 3, 4. Después eliminamos del árbol los valores: 9, 32, 3. Vamos a ver la estructura del árbol después de aplicar cada operación, indicando los casos en los que ocurre desbalanceo y la rotación que se aplica.

El resultado aparece en la figura 4.22. Se puede ver que en la inserción se requieren en total 3 rotaciones, para 8 inserciones. Por otro lado, de las 3 eliminaciones ocurren desbalanceos en 2 de ellas.

4.4. Árboles B

Los **árboles B**¹⁴ constituyen una categoría muy importante de estructuras de datos, que permiten una implementación eficiente de conjuntos y diccionarios, para operaciones de consulta y acceso secuencial. Existe una gran variedad de árboles B: los árboles B, B+ y B*; pero todas ellas están basadas en la misma idea, la utilización de árboles de búsqueda no binarios y con condición de balanceo.

En concreto, los árboles B+ son ampliamente utilizados en la representación de índices en bases de datos. De hecho, este tipo de árboles están diseñados específicamente para aplicaciones de bases de datos, donde la característica fundamental es la predominancia del tiempo de las operaciones de entrada/salida de disco en el tiempo de ejecución total. En consecuencia, se busca minimizar el número de operaciones de lectura o escritura de bloques de datos del disco duro o soporte físico.

4.4.1. Árboles de búsqueda no binarios

En cierto sentido, los árboles B se pueden ver como una generalización de la idea de árbol binario de búsqueda a árboles de búsqueda no binarios. Consideremos la representación de árboles de búsqueda binarios y n -arios de la figura 4.23.

En un ABB (figura 4.23a) si un nodo x tiene dos hijos, los nodos del subárbol izquierdo contienen claves menores que x y los del derecho contienen claves mayores que x . En un árbol de búsqueda no binario (figura 4.23b), cada nodo interno puede contener q claves x_1, x_2, \dots, x_q , y $q + 1$ punteros a hijos que están “situados” entre cada par de claves y en los dos extremos. Las claves estarán ordenadas de menor a mayor: $x_1 < x_2 < \dots < x_q$. Además, el hijo más a la izquierda contiene claves menores que x_1 ; el hijo entre x_1 y x_2 contiene claves mayores que x_1 y menores que x_2 ; y así sucesivamente hasta el hijo más a la derecha que contiene claves mayores que x_q .

¹⁴La B viene de *balanceados*.

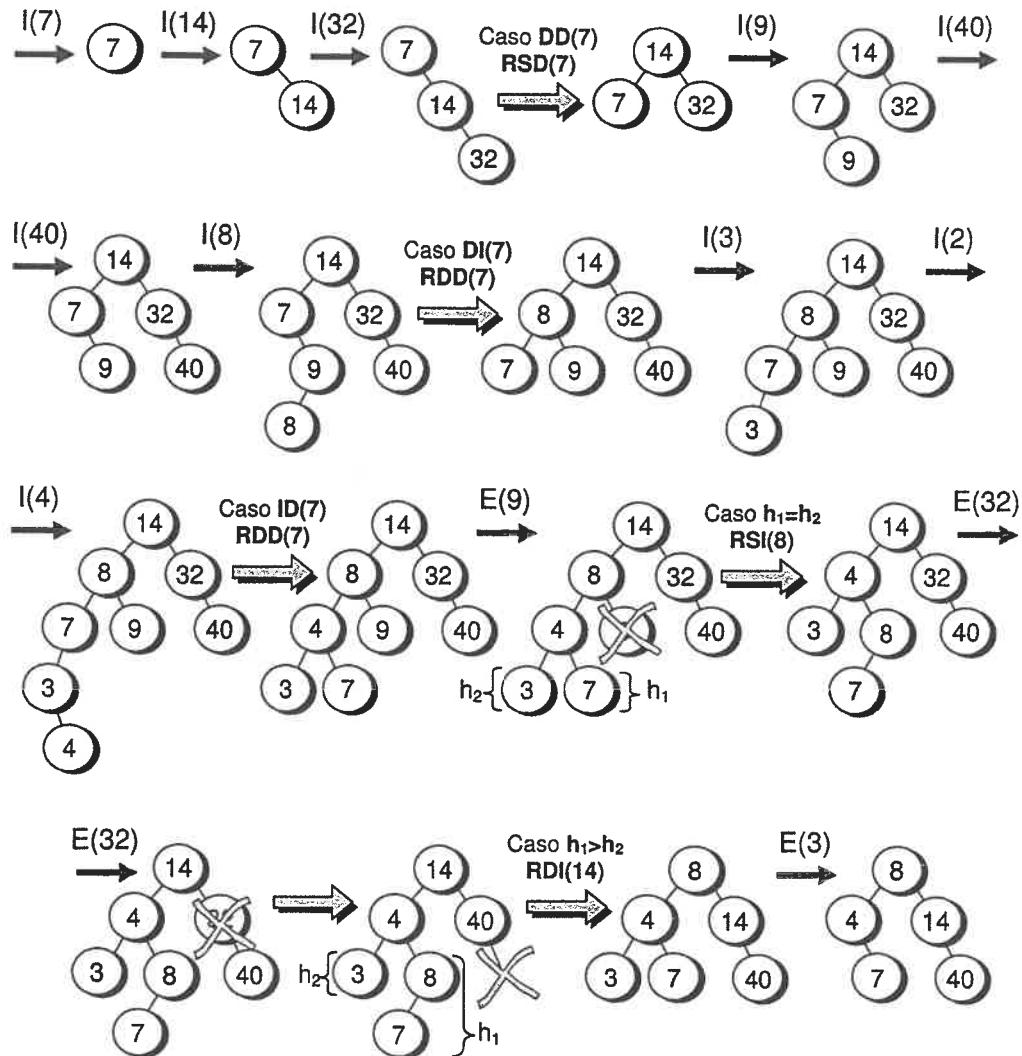


Figura 4.22: Inserción y eliminación de elementos en un AVL, según las operaciones del ejemplo 4.3.

Definición de árbol B

Un **árbol B de orden p** es básicamente un árbol de búsqueda n -ario donde los nodos tienen p hijos como máximo, y en el cual se añade la condición de balanceo de que todas las hojas estén al mismo nivel. La definición formal de árbol B es la siguiente.

Definición 4.6 Un **árbol B de orden p** , siendo p un entero mayor que 2, es un árbol con las siguientes características:

1. Los nodos internos son de la forma $(p_1, x_1, p_2, x_2, \dots, x_{q-1}, p_q)$, siendo p_i punteros a nodos hijos y x_i claves, o pares $(clave, valor)$ en caso de representar diccionarios.
2. Si un nodo tiene q punteros a hijos, entonces tiene $q - 1$ claves.

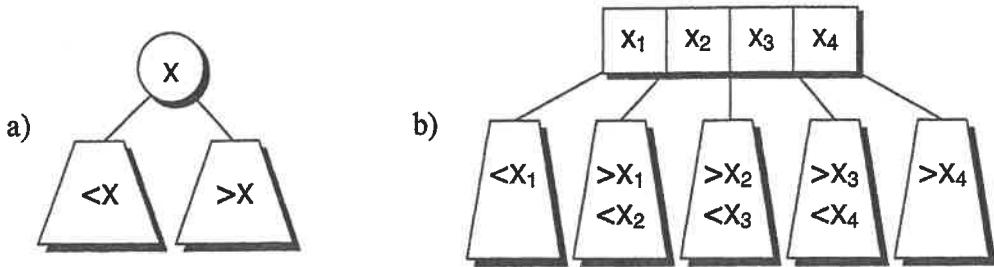


Figura 4.23: Estructura de los árboles de búsqueda. a) Árbol binario de búsqueda. b) Árbol de búsqueda n -ario.

3. Para todo nodo interno, excepto para la raíz, $\lceil p/2 \rceil \leq q \leq p$. Es decir, un nodo puede tener como mínimo $\lceil p/2 \rceil$ hijos y como máximo p . El valor $\lceil p/2 \rceil - 1$ indicaría el mínimo número de claves en un nodo interno y se suele denotar por d .
4. La raíz puede tener 0 hijos (si es el único nodo del árbol) o entre 2 y p .
5. Los nodos hoja tienen la misma estructura, pero todos sus punteros son nulos.
6. En todos los nodos se cumple: $x_1 < x_2 < \dots < x_{q-1}$.
7. Para todas las claves k apuntadas por un puntero p_i de un nodo se cumple:
 - Si $i = 1$ entonces $k < x_1$.
 - Si $i = q$ entonces $x_{q-1} < k$.
 - En otro caso, $x_{i-1} < k < x_i$.
8. Todas los nodos hoja están al mismo nivel en el árbol.

Los **árboles B+** son una variante de los árboles B, que se utiliza en representación de diccionarios. La estructura de los nodos hoja es distinta de la de los nodos internos. En esencia, la modificación consiste en que en los nodos internos sólo aparecen claves, mientras en los nodos hoja aparecen las asociaciones (*clave, valor*).

Por otro lado, tenemos la variante de los **árboles B***. Su principal característica es que si en los árboles B –teniendo en cuenta la anterior definición– los nodos deben estar ocupados como mínimo hasta la mitad, en los B* tienen que estar ocupados más de dos tercios del máximo. En adelante nos centraremos en el estudio de los árboles B. En la figura 4.24 se muestra un ejemplo de árbol B de orden $p = 5$.

Representación del tipo árbol B

Pasando ahora a la implementación de árboles B de cierto orden p , lo normal es reservar espacio de forma fija para p hijos y $p - 1$ elementos. La definición, suponiendo que el tipo almacenado es T, podría ser como la siguiente.

tipo

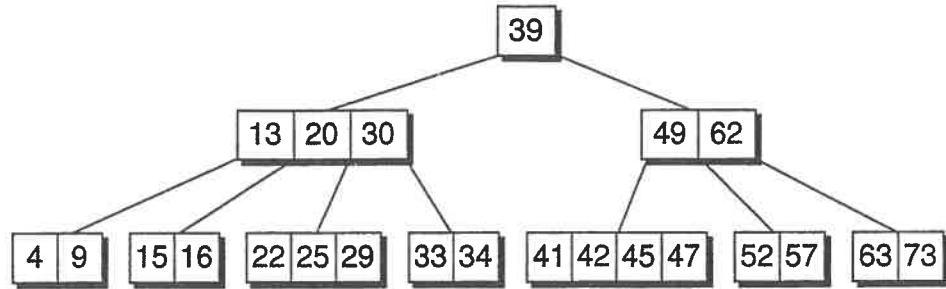


Figura 4.24: Ejemplo de árbol B de orden $p = 5$. Cada nodo interno excepto la raíz, debe tener entre 3 y 5 hijos o, equivalentemente, entre 2 y 4 entradas.

ArbolB[T,p] = Puntero[NodoArbolB[T,p]]

NodoArbolB[T,p] = registro

q: entero // Número de punteros no nulos del nodo

x: array [1..p-1] de T

ptr: array [1..p] de ArbolB[T,p]

finregistro

En aplicaciones reales de bases de datos, el valor de p se ajusta de forma que un nodo ocupe exactamente un bloque de disco. Por ejemplo, si suponemos que los enteros, punteros y valores de tipo T ocupan 4 bytes, un nodo sería de tamaño $2 * 4p$ bytes; si los bloques de disco son de 4.096 bytes, entonces usaríamos un $p = 512$.

La implementación de la operación de búsqueda consistiría simplemente en empezar por la raíz y descender hacia la rama correspondiente, según el valor de la clave buscada. Por ejemplo, la operación Miembro sobre conjuntos podría ser como la siguiente.

operación Miembro (b: ArbolB[T,p]; c: T): booleano

si $b = \text{NULO}$ entonces

devolver false

sino

para $i := 1, \dots, b.q-1$ hacer

si $c = b.x[i]$ entonces

devolver verdadero

sino si $c < b.x[i]$ entonces

devolver Miembro($b.ptr[i]$, c)

finsi

finpara

devolver Miembro($b.ptr[b.q]$, c)

finsi

Claramente, el tiempo de ejecución depende de la altura del árbol que, como veremos adelante, crece logarítmicamente con el número de nodos. En concreto, por cada nivel de la clave buscada habrá una llamada recursiva a Miembro.

4.4.2. Inserción en un árbol B

Además de mantener la estructura de árbol de búsqueda, el procedimiento de inserción en un árbol B debe asegurar la propiedad que impone que todas las hojas estén al mismo nivel. La nueva entrada debe insertarse siempre en un nodo hoja¹⁵. Si hay sitio en la hoja que le corresponde, el elemento se puede insertar directamente.

En otro caso aplicamos un proceso de **partición de nodos**, que es mostrado en la figura 4.25. El proceso consiste en lo siguiente: con las $p - 1$ entradas de la hoja donde se hace la inserción más la nueva entrada, se crean dos nuevas hojas con $\lceil(p - 1)/2\rceil$ y $\lfloor(p - 1)/2\rfloor$ entradas. La entrada m que está en la mediana aparece como una nueva clave de nivel superior, y tiene como hijas las dos hojas recién creadas.

El proceso se repite recursivamente en el nivel superior, en el que habrá que insertar m . Si m cabe en el nodo interno correspondiente, se inserta. En otro caso, se parte el nodo interno y se repite el proceso hacia arriba. Si se produce la partición a nivel de la raíz entonces tenemos el caso donde la profundidad del árbol aumenta en uno.

En definitiva, el esquema del algoritmo de inserción de una clave x en un árbol B de orden p sería el siguiente.

1. Buscar el nodo hoja donde se debería colocar x , usando un procedimiento parecido a la operación Miembro. Si el elemento ya está en el árbol, no se vuelve a insertar.
2. Si la hoja contiene menos de $p - 1$ entradas, entonces quedan sitios libres. Insertar x en esa hoja, en la posición correspondiente.
3. Si no quedan sitios libres, cogemos las $p - 1$ entradas de la hoja y x . La mediana m pasa al nodo padre, así como los punteros a sus nuevos hijos: los valores menores que m forman el nodo hijo de la izquierda y los valores mayores que m forman el nodo hijo de la derecha.
4. Si el nodo padre está completo, se dividirá a su vez en dos nodos, propagándose el proceso de partición hasta la raíz.

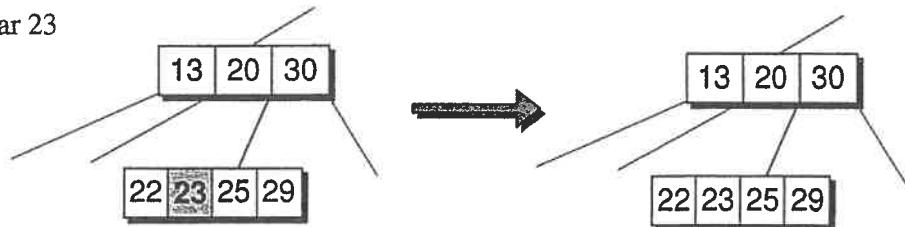
El algoritmo garantiza las propiedades de árbol B: todas las hojas están al mismo nivel y los nodos internos (excepto, posiblemente, la raíz) están llenos como mínimo hasta la mitad. Por otro lado, el tiempo de ejecución depende de la altura del árbol. Pero, además, el tiempo en cada nivel no es constante; el proceso de partición tardará un $O(p)$ en el peor caso. No obstante, ya hemos visto que realmente el factor a considerar es el número de nodos tratados, más que las operaciones que se realicen dentro de cada nodo.

4.4.3. Eliminación en un árbol B

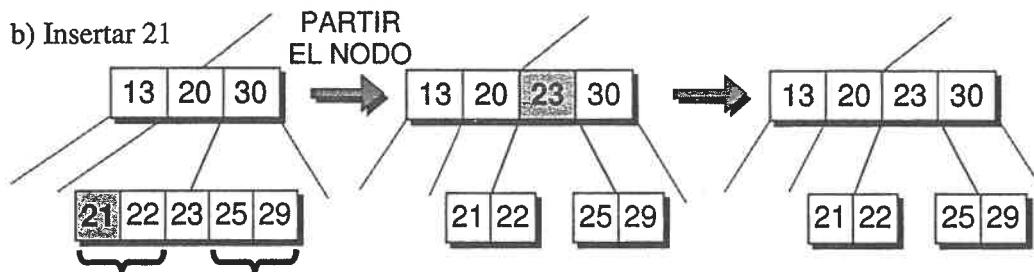
Igual que la inserción en un árbol B hace uso de la partición de un nodo en dos, la eliminación se caracteriza por el proceso de **unión de dos nodos** en uno nuevo, en caso de que el nodo se vacíe hasta menos de la mitad. No obstante, hay que tener en cuenta todas las situaciones que pueden ocurrir en la supresión.

¹⁵Ya que recordemos que los nodos internos deben tener un puntero más que claves.

a) Insertar 23



b) Insertar 21



c) Insertar 26, 27, 28

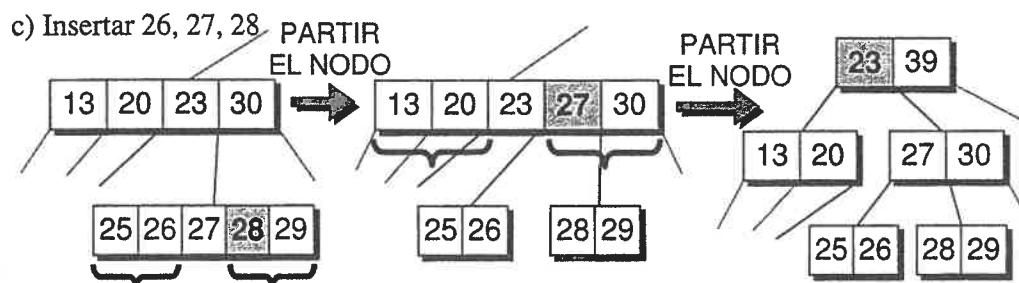


Figura 4.25: Inserción de claves en el árbol B de orden $p = 5$ de la figura 4.24. a) Inserción sin necesidad de partición. b) La inserción de 21 obliga a partir una hoja. c) Despues de insertar 28, hay que partir la hoja, y después otra vez a nivel superior.

Dada una clave x a eliminar de un árbol B, en primer lugar debemos buscar el nodo donde se encuentra x . Si se encuentra dentro de un nodo interno, no se puede suprimir de forma directa. En ese caso, habrá que sustituir x en el árbol por la siguiente clave en orden –es decir, la mayor del subárbol izquierdo o la menor del subárbol derecho de x – y se continúa con la eliminación como si se hubiera producido en la posición sustituida. Por ejemplo, si en el árbol de la figura 4.24 eliminamos la clave 39, deberíamos colocar en su lugar 34 o bien 41, y seguir con el proceso de supresión a partir de la hoja correspondiente. De esta forma, el grueso del proceso de eliminación siempre parte de un nodo hoja.

Si la clave a eliminar está en una hoja –a la que llamamos la **hoja de supresión**– o se ha aplicado la sustitución explicada antes, entonces podemos encontrarnos varios casos. Recordemos que un nodo debe contener como mínimo $d = \lceil p/2 \rceil - 1$ entradas. Los casos dependen del número de entradas de la hoja de supresión, en relación con d .

- Si la hoja de supresión tiene más de d entradas, se puede eliminar la clave directamente. Acabaría la operación sin más modificaciones.

- Si la hoja contiene exactamente d entradas, entonces al eliminar la clave se queda con $d - 1$. Será necesario “añadirle” alguna entrada, pero ¿cuál? Algún nodo hermano podría “prestarle” una entrada, si tiene alguna de sobra. Pueden ocurrir dos casos.
 - Si existe algún nodo hermano, adyacente a la hoja de supresión y que tenga más de d entradas, entonces le hace un **préstamo**: la entrada del padre común pasa a la hoja de supresión y la entrada adecuada del nodo hermano¹⁶ pasa a la posición del padre. Este proceso se muestra en la figura 4.26b).
 - Si todos los hermanos adyacentes a la hoja de supresión tienen d entradas, entonces no se puede producir el préstamo. En ese caso, la solución es **unir dos nodos en uno**. Con las $d - 1$ entradas del nodo de supresión, más las d entradas de un hermano y la entrada del padre común, se forma un nuevo nodo con $2d$ entradas. En la figura 4.26c) aparece un ejemplo de supresión que acarrea una unión de nodos.

Hay que tener en cuenta que el último caso, la unión de dos nodos, da lugar a la eliminación de una entrada a nivel superior. Por lo tanto, el proceso de eliminación debería repetirse en ese nivel superior. Es decir, si el nodo interno –el padre de la hoja de supresión– contiene más de d entradas, se puede eliminar directamente. Si tiene d entradas, entonces ocurrirá uno de los dos casos anteriores: si algún hermano tiene más de d entradas le presta una; y en otro caso se juntan dos nodos en uno. En definitiva, el proceso se iría repitiendo sucesivamente desde las hojas hasta la raíz.

Está claro que el nodo raíz no tiene hermanos, por lo que nunca podrá recibir préstamos o unirse con un hermano. Básicamente, esta es la razón por la que en los árboles B se permite que la raíz tenga menos de d entradas.

Si en el proceso de eliminación se suprime una entrada de la raíz (al juntarse dos hijos suyos) y la raíz sólo tenía esa entrada, entonces tenemos un caso donde la altura del árbol disminuye en uno. Por ejemplo, si en el árbol de la figura 4.26d) eliminamos el valor 63, habría que unir nodos, formando una hoja con 47, 52, 62 y 73. La unión se repetiría a nivel superior, dando lugar a un nuevo nodo interno con 20, 30, 39 y 45. Este nodo sería la nueva raíz, de manera que el árbol tendría ahora altura uno.

4.4.4. Análisis de eficiencia de los árboles B

En el análisis de eficiencia de los árbol B, el recurso crítico es el número de lecturas/escrituras de bloques del disco duro. Típicamente, las operaciones de acceso a disco son varios órdenes de magnitud más lentas que las que se realizan en memoria. En las implementaciones reales y eficientes de árboles B, se hace que cada nodo del árbol ocupe exactamente un bloque de disco, lo cual se consigue ajustando el parámetro p . Por lo tanto, hasta cierto límite, el tiempo de las instrucciones que se ejecutan dentro de cada nodo es despreciable y el factor importante es el número de nodos tratados¹⁷.

¹⁶Es decir, la mayor o la menor, según el nodo hermano esté a la izquierda o a la derecha del nodo de supresión.

¹⁷Podríamos decir algo parecido de las demás estructuras de datos estudiadas en este y en otros capítulos. La diferencia es que esas otras estructuras suelen encontrarse en aplicaciones que hacen uso de disco o no, mientras que los árboles B son típicos de aplicaciones de BB.DD.

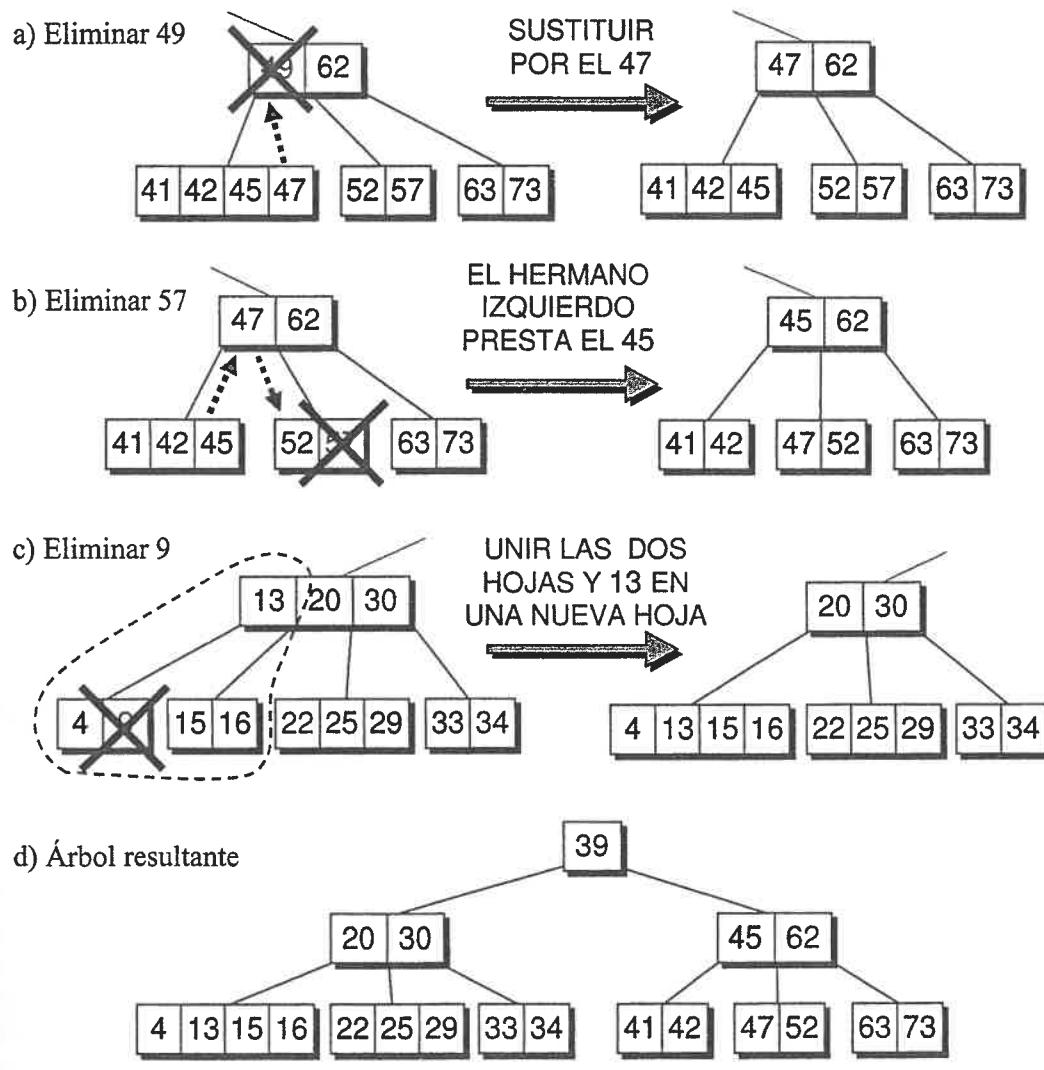


Figura 4.26: Eliminación de claves en el árbol B de orden $p = 5$ de la figura 4.24. a) Eliminación en un nodo interno. b) La eliminación de 57 requiere un préstamo del hermano. c) La eliminación de 9 produce una unión de hojas. d) Resultado final.

Análisis de tiempos de ejecución

Igual que con las restantes estructuras arbóreas, el número de nodos recorridos en las distintas operaciones es proporcional a la altura del árbol. En concreto, si la altura del árbol es h , la operación de búsqueda de una clave accederá a $h + 1$ nodos; la inserción en el peor caso tratará $2h + 1$ nodos, si se debe hacer la partición a todos los niveles; y, de forma similar, la eliminación visitará $3h + 1$ nodos en el peor caso.

Para obtener el número de nodos recorridos en función del número n de claves almacenadas en el árbol B, vamos a calcular primero la función inversa, es decir el número n de claves que caben para una cierta altura h . Podemos distinguir dos casos: en el mejor,

n_{mejor} , cada nodo interno tendrá p hijos (el máximo); y en el peor, n_{peor} , tendrá 2 ó $d+1$ si es la raíz o no (el mínimo), respectivamente. Contando el número de claves almacenadas en cada nivel, para una altura total h , tenemos lo siguiente.

Nivel	n_{mejor}	n_{peor}
0	$p - 1$	1
1	$(p - 1)p$	$2d$
2	$(p - 1)p^2$	$2d(d + 1)$
3	$(p - 1)p^3$	$2d(d + 1)^2$
...
h	$(p - 1)p^h$	$2d(d + 1)^{h-1}$

Sumando todas las claves en las distintas alturas obtenemos n_{mejor} y n_{peor} .

$$n_{mejor}(h) = (p - 1) \sum_{i=0}^h p^i = p^{h+1} - 1 \quad (4.4)$$

$$n_{peor}(h) = 1 + 2d \sum_{i=0}^{h-1} (d + 1)^i = 2(d + 1)^h - 1 \quad (4.5)$$

Ahora podemos despejar la altura h y expresarla en función de n , que es lo que realmente nos interesa. Tenemos.

$$h_{mejor}(n) = \lceil \log_p(n + 1) \rceil - 1 \quad (4.6)$$

$$h_{peor}(n) = \left\lceil \log_{d+1} \frac{n + 1}{2} \right\rceil = \left\lceil \log_{\lceil p/2 \rceil} \frac{n + 1}{2} \right\rceil \quad (4.7)$$

Como se puede ver, en todos los casos la altura está en un $O(\log n)$ y lo mismo ocurrirá con el tiempo de las operaciones sobre el árbol B, si contamos sólo la entrada/salida de disco. Pero lo que resulta realmente interesante es la base del logaritmo. No es lo mismo¹⁸ un logaritmo en base 2 que en base 512. En la tabla 4.3 se muestran comparativamente las alturas según el número n de claves, utilizando árboles AVL y B. Se muestran también unos valores de ejemplo, suponiendo que queremos almacenar el conjunto de los DNI de todos los españoles (alrededor de 40 millones) y que $p=512$.

Estructura	Altura mejor caso (ejemplo)	Altura peor caso (ejemplo)		
Árbol AVL	$\lceil \log_2(n + 1) \rceil - 1$	25	$\lceil \log_{1,62}((n + 1)/1,89) \rceil$	35
Árbol B	$\lceil \log_p(n + 1) \rceil - 1$	2	$\lceil \log_{\lceil p/2 \rceil}((n + 1)/2) \rceil$	4

Tabla 4.3: Alturas en el mejor y peor caso de un árbol AVL y un árbol B con n nodos. En el ejemplo, $n = 40.000.000$ y $p = 512$.

Realmente, la comparativa debería tener en cuenta que de los veinte o treinta nodos leídos en el AVL, puede que muchos de ellos estén en el mismo bloque de disco. El

¹⁸Aunque sí que son iguales en cuanto a órdenes de complejidad. Es decir $O(\log_2 x) = O(\log_{512} x)$.

número de E/S sería algo menor. Sin embargo, el árbol B nos garantiza que la búsqueda no requerirá nunca más de cinco E/S (una más que la altura).

Por otro lado, si suponemos que tanto el árbol B como el AVL están en memoria, deberíamos multiplicar el número de nodos recorridos por el tiempo en cada nodo. El tiempo por nodo en el AVL es constante, mientras que en el árbol B depende del número de entradas del nodo. Haciendo una búsqueda binaria en cada nodo, tendríamos $\lceil \log_2(q+1) \rceil$ comparaciones en el peor caso, si el nodo tiene q entradas. Si consideramos el mejor caso de altura, $q = p - 1$ en todos los nodos y el número de nodos tratados sería $\lceil \log_p(n+1) \rceil$. Multiplicando los dos términos tenemos:

$$\lceil \log_2 p \rceil \lceil \log_p(n+1) \rceil \approx \lceil \log_2(n+1) \rceil$$

En conclusión, en el árbol B el número de comparaciones es un logaritmo en base 2, ¡exactamente el mismo logaritmo que con árboles AVL!¹⁹! La ventaja de los árboles B se encuentra, por lo tanto, cuando consideramos las E/S de disco.

Utilización de memoria

La implementación de árboles B usando un bloque de disco –de tamaño fijo– por cada nodo, implica una reserva de memoria que después puede ser utilizada o no. Esta situación es comparable a lo que puede ocurrir con tablas de dispersión, donde se reservan muchas cubetas que después pueden usarse o no. Algo parecido ocurre en los árboles AVL. Por cada clave existen dos punteros, pero todos los punteros de los hijos tendrán siempre valor nulo. En un árbol B, con un p suficientemente grande, la proporción está en torno a un puntero por clave. Por contra, en el peor caso los nodos estarán medio vacíos.

Consideremos que una entrada (clave y valor) ocupa k_1 bytes y un puntero k_2 bytes. Un árbol AVL necesitará siempre $n(k_1 + 2k_2)$ bytes, para almacenar n entradas. Por otro lado, de forma aproximada, el árbol B suponiendo que sólo se ocupan los nodos hasta la mitad, ocuparía $2n(k_1 + k_2)$. Si los nodos se llenan con más proporción, el tamaño tendería a $n(k_1 + k_2)$. En definitiva, según el porcentaje de llenado de los nodos la utilización de memoria será más o menos eficiente.

Ejemplo 4.4 En un árbol B de orden $p=4$ inicialmente vacío almacenamos números enteros. Insertamos los siguientes elementos: 37, 14, 60, 9, 22, 51, 10, 5, 55, 70, 1. La estructura del árbol después de cada operación se muestra en la figura 4.27.

Ejercicios resueltos

Ejercicio 4.1 En cierta aplicación, utilizamos un árbol trie para representar palabras en dos o más idiomas, por ejemplo, español e inglés. Queremos añadir a cada palabra una definición de su significado y la traducción al otro idioma. Describir la estructura de datos, mostrando las definiciones de los tipos necesarios. Hay que tener en cuenta que algunas palabras pueden tener significado en los dos idiomas, por ejemplo, can, conductor, mete, sin.

¹⁹Se puede comprobar que tomando el peor caso también obtenemos un logaritmo en base 2.

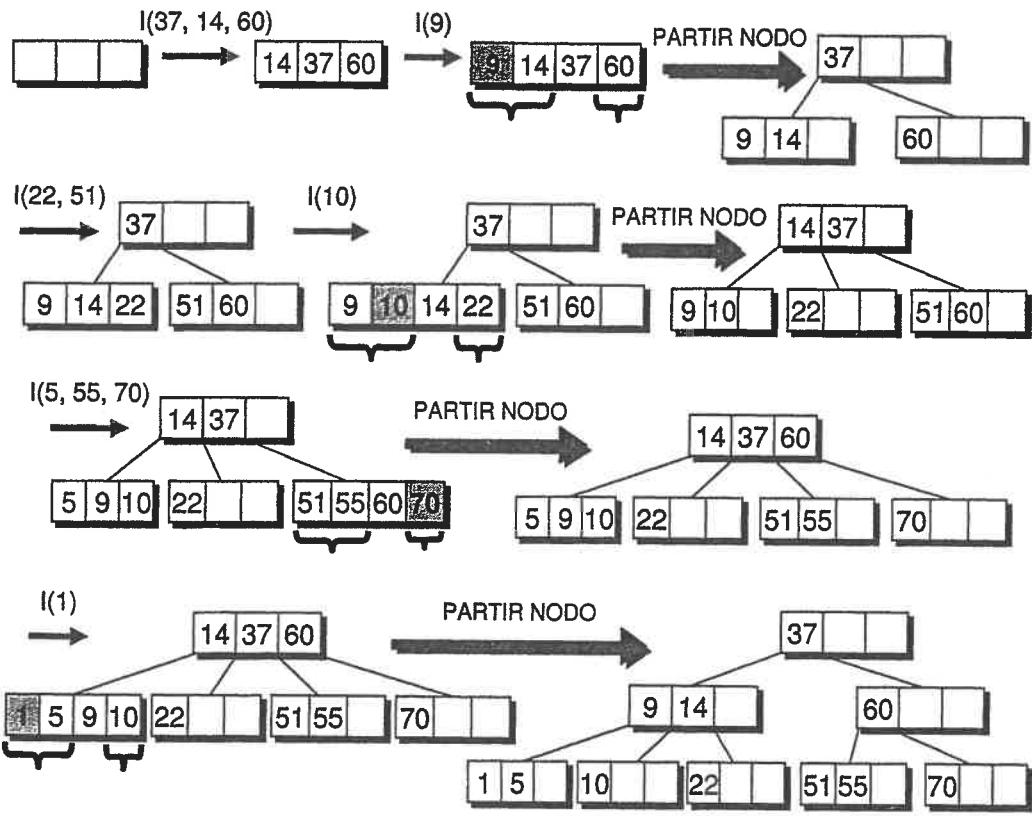


Figura 4.27: Inserción en un árbol B de orden $p=4$, según las operaciones del ejemplo 4.4.

Solución.

Está claro que la información añadida corresponde a la posición donde es colocada la marca de fin \$. Ahora la marca puede ser de dos tipos: \$I- palabra en inglés, \$E- palabra en español, y así sucesivamente si tuviéramos más idiomas. Realmente, estas marcas deberían corresponder con códigos ASCII de caracteres no utilizados.

La definición del tipo ArbolTrie podría ser como la siguiente.

tipo

ArbolTrie[A: Alfabeto] = Puntero[NodoTrie[A]]

Para representar los nodos del trie utilizamos listas enlazadas, es decir una representación hijo izquierdo/hermano derecho. Además, dentro de los elementos de las listas podemos encontrarnos también con definiciones de palabras completas, que serán apuntadas por los nodos con marca de fin. Por lo tanto, usamos registros con variantes para definir el tipo.

tipo

ClaseNodo = enumerado (prefijo, descripción)

NodoTrie[A: Alfabeto] = registro

según clase: ClaseNodo

prefijo: registro

car: A // Carácter o \$E o \$I

```

ptr: Puntero[NodoTrie[A]] // Nodo descendiente o descripción
sig: Puntero[NodoTrie[A]] // Siguiente nodo en la lista
finregistro
descripción: registro
definición: cadena // Definición de la palabra
traducción: cadena // Traducción al otro idioma
finregistro
finsegún
finregistro

```

Si una palabra existe en los dos idiomas, entonces aparecerá un nodo con \$E y otro con \$I, apuntando a las descripciones correspondientes.

Ejercicio 4.2 Escribe un algoritmo para resolver el siguiente problema con árboles trie. Dada una cadena *prefijo*, listar todas las palabras del árbol que empiecen por *prefijo*. Por ejemplo, si *prefijo* = "pane", podrían aparecer por pantalla: "panecillo, panegírica, panegírico, panegirista, panel, ...". Suponer que tenemos la función *Longitud(cadena)* para conocer la longitud de una cadena y accedemos a las letras con *cadena[1]*, *cadena[2]*, *cadena[3]*, etc. Utilizar la definición de árboles trie estudiada en la sección 4.1.2, con las operaciones: Crear, Asigna, ValorDe y TomaNuevo.

¿Qué ocurre si en lugar de buscar por prefijos queremos buscar por sufijos? Dar una idea de cómo resolver el problema en ese caso.

Solución.

En primer lugar, deberíamos colocarnos en el nodo del árbol correspondiente al prefijo *prefijo* pasado como parámetros. El resultado serían todos los descendientes de ese nodo etiquetados con la marca \$. Por lo tanto, el algoritmo tiene dos partes: un procedimiento para moverse hasta *prefijo* y otro procedimiento ListarTodas recursivo, para listar todas las palabras completas a partir de un nodo del árbol. La implementación sería como la siguiente.

```

operación PalabrasConPrefijo (prefijo: cadena; arbol: ArbolTrie[A])
var ptr, aux: Puntero[NodoTrie[A]]
i := 1
ptr := arbol
mientras (ptr ≠ NULO) Y (i ≤ Longitud(prefijo)) hacer
    aux := ValorDe(ptr↑, prefijo[i])
    si aux = NULO entonces
        escribir("No existe el prefijo", prefijo)
        acabar
    finsi
    ptr := aux
    i := i + 1
finmientras
ListarTodas(prefijo, ptr)

```

```

operación ListarTodas (prefijo: cadena; ptr: Puntero[NodoTrie[A]])
```

var

```

c: A
aux: Puntero[NodoTrie[A]]

si ValorDe (ptr↑, $) ≠ NULO entonces
    escribir (prefijo)
finsi
para c en Rango(A) hacer
    aux:= ValorDe (ptr↑, c)
    si aux ≠ NULO entonces
        ListarTodas(prefijo + c, aux)
    finsi
finpara

```

Si en lugar de buscar por prefijos buscamos por sufijos, tenemos el inconveniente de que sólo podemos saber si se cumple la condición al llegar a una hoja del trie. Por lo tanto, deberíamos buscar todas las palabras del árbol y para cada una comprobar si acaba con el sufijo requerido. Se puede conseguir, por ejemplo, usando el procedimiento ListarTodas, modificando la acción asociada a:

```

si ValorDe (ptr↑, $) ≠ NULO entonces
    ejecutando la operación escribir sólo si la palabra acaba con el sufijo que se pide.

```

Ejercicio 4.3 A la estructura de representación de relaciones de equivalencia con equilibrado y compresión de caminos, queremos añadir una operación *Quitar(R, a)* que elimina todas las relaciones de *a* con los demás nodos. Es decir, la clase de *a* pasa a ser ella misma, pero las demás clases existentes no se modifican. Implementar la operación *Quitar*.

Solución.

Habrá que colocar *a* como raíz de su propio árbol, haciendo algo como: $R[a]:= 0$. Además, hay que tener en cuenta que *a* puede tener otros hijos, y la relación entre ellos y los padres de *a* no debe perderse. Es más, si *a* ya era una raíz de un árbol, entonces habrá que elegir una nueva raíz entre los hijos de *a*. Por lo tanto, siempre será necesario recorrer toda la tabla, cambiando los nodos cuyo padre fuera *a*. La implementación podría ser como la siguiente.

```

operación Quitar (var R: RelacionEquiv[N]; a: entero)
    si R[a] < 0 entonces // a era una raíz
        nuevaRaiz:= 0
    sino
        nuevaRaiz:= R[a]
    finsi
    para i:= 1, ..., N hacer
        si R[i] = a entonces
            si nuevaRaiz = 0 entonces
                nuevaRaiz:= i // i pasa a ser la nueva raíz del árbol
                R[i]:= R[a] // Su profundidad es la misma que la de a
            sino
                R[i]:= nuevaRaiz
            finsi

```

finsi

finpara

Claramente, el tiempo de ejecución es un $O(N)$, independientemente de que a tuviera hijos o no. La razón es que en la estructura de punteros al padre no se conocen de forma explícita los hijos de un nodo.

Se puede ver cómo en el paso de que a fuera una raíz, a la raíz buscada le asignamos como profundidad la misma que la que tenía a . Realmente la profundidad puede haber cambiado. Pero, como vimos en la sección 4.2.3, recalcular la profundidad sería un proceso excesivamente costoso.

Ejercicio 4.4 En un árbol AVL inicialmente vacío insertamos, por este orden, los siguientes elementos: 8, 13, 10, 5, 11, 6, 7; y después eliminamos: 13, 6, 7, 5. Mostrar la estructura del árbol antes y después de cada operación que requiera un rebalanceo, indicando el caso que ocurre y cuál es el tipo de rebalanceo aplicado.

Solución.

Se trata simplemente de aplicar las ideas vistas en teoría para la inserción y eliminación en un AVL, sobre el ejemplo concreto. El resultado aparece en la figura 4.28.

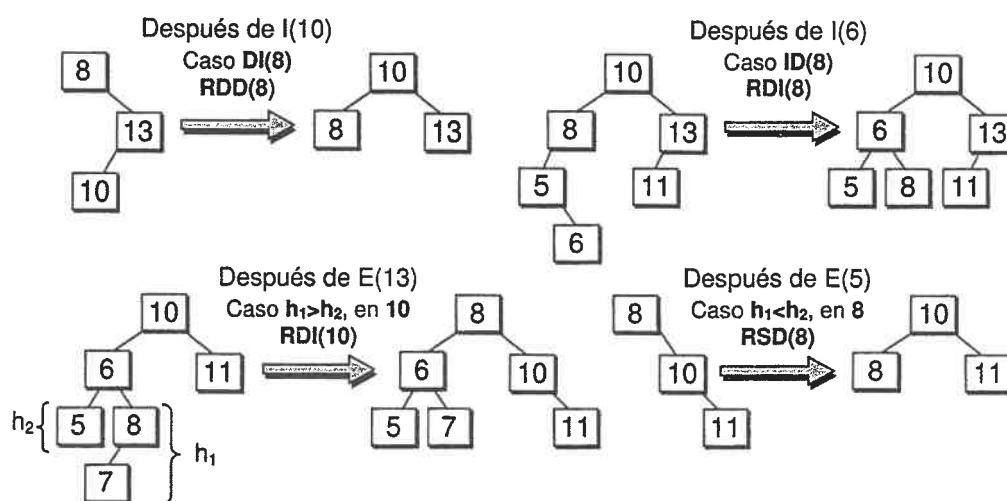


Figura 4.28: Operaciones en el AVL resultantes del ejercicio 4.4.

Ejercicio 4.5 Queremos construir una variante intermedia entre los árboles AVL y los ABB no balanceados. Para ello, definimos los árboles BWM igual que los árboles AVL, pero la condición de balanceo es que la altura de los subárboles de cada nodo puede diferir como máximo en 2.

Expón las principales ventajas e inconvenientes de los árboles BWM respecto a los AVL. Mostrar los peores casos, en cuanto a número mínimo de nodos, de árbol BWM para altura 1, 2, 3, 4, y 5. ¿Cuál es el número mínimo de nodos para una altura h ?

Solución.

Como se puede ver, la definición es muy parecida a la de árbol AVL, pero aumentando la máxima diferencia de alturas permitida. Por lo tanto, tenemos una condición de balanceo más débil que en los árboles AVL. La principal ventaja es que, en principio, será previsible que hayan menos rotaciones en la inserción y la eliminación. El inconveniente es que al estar el árbol menos balanceado, crecerá más en altura para un número de nodos dado. Además, habría que estudiar los casos de desbalanceo en la inserción y la eliminación; y decidir qué rotaciones habría que aplicar para solucionarlo.

En cuanto al peor caso de BWM, podemos aplicar un razonamiento similar al de los árboles AVL. El peor caso de BWM de altura h , para $h > 3$, se formaría uniendo dos subárboles con diferencia de altura 2, es decir uno de altura $h - 1$ y otro de altura $h - 3$. En la figura 4.29 se muestran los peores casos de BWM para alturas desde 0 hasta 5.

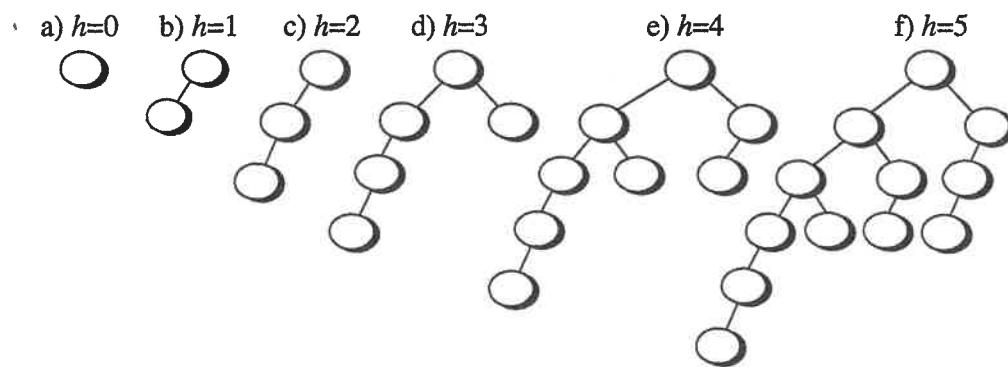


Figura 4.29: Peores casos de árbol BWM, según la definición del ejercicio 4.5.

El número mínimo de nodos para una altura h vendría dado por la fórmula recursiva:

$$N(h) = \begin{cases} 1 & \text{Si } h = 0 \\ 2 & \text{Si } h = 1 \\ 3 & \text{Si } h = 2 \\ N(h - 1) + N(h - 3) + 1 & \text{Si } h > 2 \end{cases} \quad (4.8)$$

Se puede comprobar que el resultado es de la forma: $N(h) = cl.466^h + \dots$, donde c es una constante y los puntos significan términos de menor grado. Por lo tanto, la altura para un cierto número de nodos sería un logaritmo en base 1,466 y el tiempo de ejecución de las búsquedas estaría en $O(\log n)$.

Ejercicio 4.6 Considerar el árbol B de orden $p = 4$ resultante del ejemplo 4.4. Sobre el mismo se aplican las siguientes operaciones: Inserta(15), Inserta(6), Insertar(8), Elimina(70), Elimina(60), Elimina(51). Mostrar la estructura del árbol después de la inserción de 8 y después de cada eliminación.

Solución.

Los árboles resultantes se muestran en la figura 4.30.

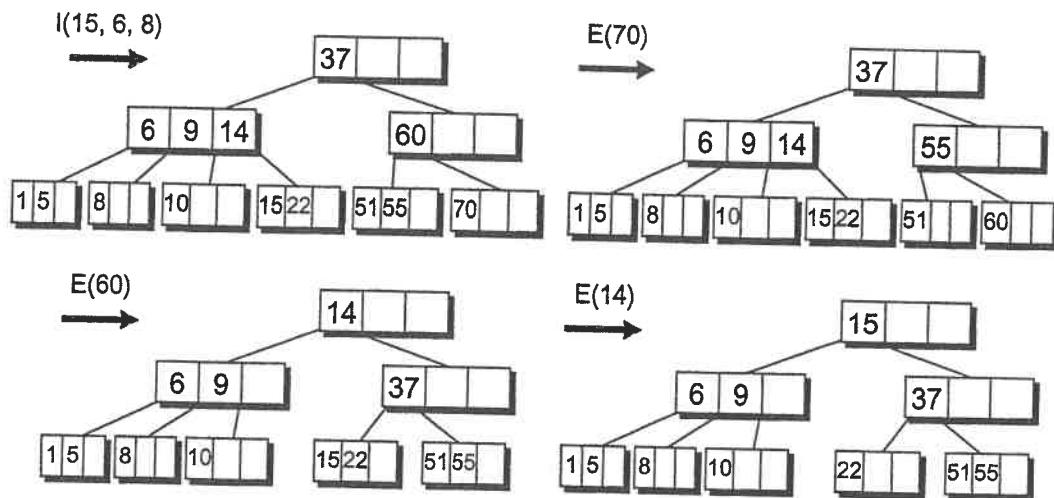


Figura 4.30: Árboles B resultantes del ejercicio 4.6.

Ejercicio 4.7 Un árbol B de orden $p = n + 1$ se usa para almacenar números enteros. La definición del tipo de datos es la siguiente.

tipo

```

NodoArbolB = registro
    valores: array [1..n] de entero
    punteros: array [1..n+1] de Puntero[NodoArbolB]
finregistro
```

ArbolB = Puntero[NodoArbolB]

Si una posición está vacía, entonces el valor correspondiente vale -1 y el puntero vale NULO. Para los nodos hoja, todos los punteros tienen valor NULO. Escribe un procedimiento para recorrer de forma ordenada (de menor a mayor) todos los elementos almacenados en un árbol B dado, usando esta definición.

Solución.

El procedimiento es básicamente un recorrido recursivo del árbol B en orden simétrico o in-orden. En un árbol binario, el recorrido en in-orden se consigue recorriendo primero el subárbol izquierdo, luego la raíz y después el subárbol derecho. Ahora debemos tener en cuenta que un nodo puede tener n entradas y $n+1$ hijos como máximo. La implementación podría ser la siguiente.

```

operación RecorrerArbolB (B: ArbolB)
    si B↑.punteros[1] ≠ NULO entonces
        RecorrerArbolB(B↑.punteros[1])
    para i := 1, ..., n hacer
        si B↑.valores[i] ≠ -1 entonces
            escribir(B↑.valores[i])
        si B↑.punteros[i+1] ≠ NULO entonces
            RecorrerArbolB(B↑.punteros[i+1])
finpara
```

Ejercicio 4.8 El dibujo de la figura 4.31 representa un árbol B de orden $p = 5$. ¿Es correcto el estado del árbol o hay algún error? En caso de haber errores, indica cuáles y por qué.

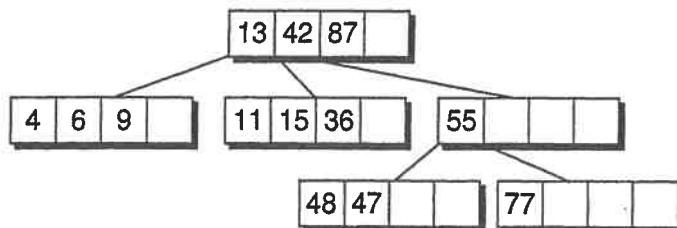


Figura 4.31: Árboles B erróneo, del ejercicio 4.6.

Solución.

El árbol B de la figura 4.31 presenta las siguientes inconsistencias:

- Todos los nodos hoja deberían estar al mismo nivel. En este caso hay dos hojas a nivel 1, y otras dos a nivel 2.
- El elemento 11 está mal colocado, porque es menor que 13 pero está a su derecha.
- La raíz tiene tres entradas y tres hijos. Es erróneo, porque cualquier nodo interno debe contener una entrada menos que hijos. El 87 debería ser colocado en otro sitio.
- El nodo interno que contiene al 55 es erróneo porque sólo contiene una entrada. En un árbol B de orden $p = 5$ los nodos internos deben tener como mínimo 2 entradas, es decir, deben estar llenos como mínimo hasta la mitad.
- Normalmente, la condición de “llenado hasta la mitad” también se impone para los nodos hoja, aunque según la definición estudiada no es obligatorio. Si la tenemos en cuenta, la hoja que contiene sólo el 77 sería errónea.
- La hoja que contiene los elementos 48 y 47 es errónea, porque los elementos dentro de un nodo deben estar ordenados de menor a mayor. Primero debería ir 47 y luego 48.

Ejercicios propuestos

Ejercicio 4.9 Muestra el resultado de insertar los siguientes elementos en un árbol trie: patria, patriada, patriar, patriarca, patriarcago, patriarcado. Elige un tipo de representación para los nodos del árbol y haz una estimación aproximada de la memoria ocupada para este ejemplo concreto.

Ejercicio 4.10 Un sistema multiusuario debe llevar el control de las personas que acceden al mismo. Para ello, se debe guardar para cada persona su nombre, apellidos y una clave de acceso. La operación básica consiste en dado un nombre y apellidos, obtener su clave de acceso. Puesto que se espera que el sistema tenga una cantidad muy grande de usuarios –y muchos de ellos tendrán el nombre o algún apellido comunes– se busca una representación eficiente en cuanto a tiempo y tamaño. ¿Cómo sería la representación mediante árboles Trie? ¿Qué ventajas e inconvenientes tiene? ¿Qué otras estructuras pueden ser adecuadas para este problema?

Ejercicio 4.11 Implementar las siguientes operaciones sobre árboles trie, suponiendo los tipos definidos en el apartado 4.1.2. Hacer una estimación del orden de complejidad de las operaciones.

- a) Encontrar la menor palabra, en orden alfabético, contenida en un trie.
- b) Comprobar si a partir de cierto nodo sólo existe una palabra válida.
- c) Encontrar la palabra más larga de un árbol trie.
- d) Eliminar una palabra de un árbol trie.

Ejercicio 4.12 Cierto algoritmo de compresión se basa en la repetición de secuencias que suelen existir en los ficheros. El programa lee un fichero y devuelve un fichero de salida comprimido. Si se encuentra una secuencia larga que apareció con anterioridad, se sustituye la secuencia por una referencia $COPIA(X, Y)$, que indica que en ese lugar se deben colocar X caracteres empezando por los que aparecieron Y posiciones antes en ese mismo fichero. Por ejemplo, la cadena: "Modulador-Demodulador..." se comprimiría como: "Modulador-Dem $COPIA(8, 12)$...".

Utilizando tries se facilitaría la búsqueda de secuencias que han aparecido con anterioridad. Describe las principales características de la estructura de tries en esta aplicación y como sería manejado el trie en el proceso de compresión.

Ejercicio 4.13 Define el TAD genérico RelacionEquivalencia[T], mediante alguno de los métodos de especificación formal estudiados en el capítulo 2. Añade operaciones para unir dos clases, quitar elementos de una clase de equivalencia, y ponerlos en una clase nueva o en una clase distinta.

Ejercicio 4.14 Utilizando la estructura de representación de relaciones de equivalencia, con equilibrado de nodos y compresión de caminos, muestra los árboles resultantes tras aplicar las siguientes operaciones. Se supone que el conjunto universal contiene 9 elementos, numerados del 1 al 9.

Unión(2, 3), Unión(4, 5), Unión(7, 6), Unión(4, 2), Unión(9, 4), Búsqueda(3), Unión(7, 4), Unión(1, 5)

Compara el resultado con la estructura obtenida suponiendo que se usa la implementación sin equilibrado ni compresión de caminos. ¿Con cuál se obtienen mejores resultados, en cuanto al tiempo de ejecución? ¿Por qué?

Ejercicio 4.15 En el peor caso de la estructura de relaciones de equivalencia con equilibrado de nodos (es decir, el mínimo número de nodos para cierta altura), un árbol con altura h tendrá 2^h nodos. Mostrar cuál sería el peor caso del árbol de altura 4. ¿Cómo se forma este árbol? ¿Cuál sería el tiempo de ejecución en ese caso? Ojo: aunque tengamos un 2^h , el árbol en este caso no es un árbol binario.

Ejercicio 4.16 En una aplicación de procesamiento de imágenes, representamos las imágenes como matrices mediante un tipo: `Imagen = array [1..800, 1..600] de entero`. Nos interesa encontrar las regiones contiguas del mismo color, como en la figura 4.8. Para ello utilizamos la estructura de relaciones de equivalencia, cuya implementación damos por supuesta. Escribe un algoritmo que dada una imagen del tipo `Imagen` devuelva una relación de equivalencia que contenga las regiones a las que pertenece cada píxel. Sugerencia: recorrer la matriz de arriba abajo, de izquierda a derecha. Para cada punto, comprobar si hay que unirlo a una clase existente, a una nueva clase o hay que unir dos clases.

Ejercicio 4.17 Mostrar los pasos de ejecución y la estructura de árbol resultante al aplicar las siguientes operaciones sobre un árbol AVL inicialmente vacío:

`Inserta(8), Inserta(20), Inserta(12), Inserta(5), Inserta(35), Inserta(40),
Inserta(7), Elimina(35), Inserta(10), Elimina(20), Elimina(5)`

Ejercicio 4.18 Muestra un ejemplo de árbol AVL donde al eliminar un elemento dado se requiera más de un rebalanceo para reequilibrar el árbol. Muestra el árbol antes y después de la eliminación. Ojo, no se pide un ejemplo de rotación doble sino de más de un rebalanceo a distintos niveles.

Ejercicio 4.19 Para el árbol AVL mostrado en la figura 4.32, aplicar las siguientes inserciones de elementos: 27, 22, 25, 12, 11. Después eliminar: 7, 20, 27, 25. En caso de desbalanceo decir de qué tipo es y cuál es la operación que se aplica para solucionarlo. ¿La solución obtenida es única?

Repetir las mismas operaciones suponiendo que se utiliza un ABB perfectamente balanceado. Compara los resultados obtenidos con las dos estructuras.

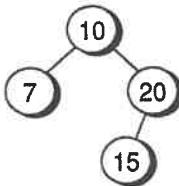


Figura 4.32: Árbol AVL del ejercicio 4.19.

Ejercicio 4.20 Dada la definición de árboles BWM del ejercicio 4.5, comprobar que se puede aplicar el mismo análisis de casos que para los árboles AVL –pero ahora con la condición de diferencia de altura 2– en el caso de desbalanceo en la inserción. Es decir, verificar mediante cálculo de alturas que los casos que aparecen son II, ID, DI y DD, y las rotaciones que los solucionan son las mismas que con los AVL.

Mostrar la estructura de árbol BWM tras la inserción de los siguientes elementos: 11, 10, 4, 5, 2, 6, 7, 9, 8, 12, 13.

Mostrar el árbol AVL resultante de insertar los mismos elementos. Compara el número de rebalanceos necesarios en el AVL con el requerido para el caso de los árboles BWM. ¿Era previsible el resultado obtenido? ¿Por qué?

Ejercicio 4.21 Usando el esquema de algoritmo del apartado 4.3.5, escribe en pseudocódigo un algoritmo para la eliminación de un elemento en un árbol AVL.

Ejercicio 4.22 Suponiendo una estructura de árboles B de orden $p = 5$, muestra cómo sería el resultado de aplicar las siguientes inserciones sobre un árbol inicialmente vacío: 9, 4, 15, 1, 3, 20, 24, 35, 17, 18, 19, 40, 41, 50, 36, 22, 21, 10, 0. ¿Cuál es la altura del árbol?

A partir del árbol B resultante, mostrar la aplicación de las siguientes eliminaciones: 19, 40, 21, 4.

Ejercicio 4.23 Suponiendo la definición del tipo ArbolB del ejercicio 4.7, escribe procedimientos en pseudocódigo para resolver los siguientes problemas:

- Encontrar el mayor y el menor elemento del árbol.
- Encontrar el siguiente elemento (en orden) a uno dado.
- Listar todos los elementos entre un par de valores, *min* y *max*.
- Eliminar un elemento de un árbol B, teniendo en cuenta todos los casos.

Ejercicio 4.24 En una base de datos de personas, necesitamos acceder a los datos de una persona por su nombre o por su número de DNI. Para ello creamos dos estructuras de representación que referencian a los mismos datos: un árbol B para los nombres y una tabla hash donde las claves son los números de DNI. Define los tipos necesarios para tener esta estructura dual y describe las operaciones necesarias para manejarla. Señala las principales ventajas e inconvenientes de esta forma de representación.

Ejercicio 4.25 Partiendo del árbol B de la figura 4.26d), mostrar la aplicación de las operaciones: Inserta(18), Elimina(30), Elimina(52), Elimina(33), Elimina(41), Elimina(45).

Ejercicio 4.26 En un árbol B de orden $p = 3$ inicialmente vacío, insertamos los elementos: 10, 8, 24, 12, 17, 15 y 13. Mostrar el árbol B después de cada inserción que modifique la estructura del árbol y el resultado final. El resultado obtenido ¿depende del orden en que han sido insertados los elementos? Mostrar también un ABB perfectamente balanceado que contenga los mismos elementos.

Cuestiones de autoevaluación

Ejercicio 4.27 Explica por qué es necesario, en la representación de conjuntos mediante árboles trie, utilizar una marca de fin de palabra \$, puesto que podríamos hacer que las palabras del conjunto se correspondieran con las hojas del árbol, sin necesidad de utilizar marcas de fin.

Ejercicio 4.28 En un árbol trie queremos representar palabras acentuadas y distinguir entre mayúsculas y minúsculas. Por lo tanto, se propone añadir el conjunto de caracteres del alfabeto las vocales acentuadas, mayúsculas y minúsculas. Explica cómo afectaría a la memoria y al tiempo de ejecución, con las representaciones de nodos con arrays y con listas enlazadas. ¿Qué otras posibilidades existen, modificando la estructura de árboles trie? Sugerencia: La información del acento podría ir asociada a la marca de fin.

Ejercicio 4.29 Tal y como hemos visto, los árboles trie se utilizan para representar conjuntos o diccionarios de palabras. Teniéndolo en cuenta, ¿es correcto definir un trie como un tipo de datos abstracto? En otro caso, ¿qué es? Justifica la respuesta, en función de las propiedades de un TAD. Resuelve las mismas cuestiones para las tablas de dispersión, los AVL y los árboles B.

Ejercicio 4.30 Los nodos de un trie se pueden ver como diccionarios, donde las claves son letras y los valores son punteros a nodos. ¿Qué otras estructuras podrían usarse para representar los nodos de un trie, de forma que se eliminen los problemas de representar con arrays (desperdicio de memoria) y con listas (búsquedas lentas dentro de la lista)? Justifica la respuesta y muestra cuál sería la eficiencia de la operación Miembro aplicada sobre el trie.

Ejercicio 4.31 En el estudio de las relaciones de equivalencia, partimos del hecho de que las relaciones no se pueden calcular mediante fórmula y pueden cambiar en tiempo de ejecución. En caso de no cumplirse alguna de estas condiciones, ¿por qué no tendría sentido usar la estructura de datos mediante punteros al padre? ¿Cuál sería la forma óptima de implementar las operaciones del tipo en esos casos?

Ejercicio 4.32 Los árboles AVL son utilizados como un método para representar conjuntos ordenados. Propón un esquema en pseudocódigo para implementar las operaciones Unión, Intersección y Diferencia. Compara la eficiencia de estas operaciones con la conseguida en otras implementaciones de conjuntos ordenados.

Ejercicio 4.33 Suponiendo que sobre un AVL se aplica una rotación simple o doble, ¿el árbol resultante conserva siempre la propiedad de ser un árbol AVL? Compruébalo haciendo un cálculo de las alturas de los subárboles.

Ejercicio 4.34 En la operación Inserta, para añadir una clave nueva a un árbol AVL, sabemos que como máximo se realizará un rebalanceo a una altura determinada. Aprovechando esta propiedad, modificamos el procedimiento de inserción para que no se compruebe la condición de equilibrio después de haber realizado una reestructuración. ¿Qué mejora se obtiene en el tiempo de ejecución y en el orden de complejidad con esta modificación?

Ejercicio 4.35 Comentar razonadamente cómo puede ser utilizado un árbol AVL para ordenar una secuencia de n elementos en un tiempo $O(n \log n)$.

Ejercicio 4.36 Si contamos el número de comparaciones necesarias para encontrar un elemento, los árboles AVL y los árbol B están dentro de valores muy próximos. ¿En qué tipo de aplicaciones los árboles B ofrecen una mejora importante sobre los AVL? ¿Por qué? ¿Cuánto es exactamente la mejora y en función de qué está dada?

Ejercicio 4.37 ¿Cuántos casos pueden aparecer en la eliminación de una entrada en un árbol B? ¿Cómo se soluciona cada uno de ellos? ¿Pueden existir varios resultados válidos para una eliminación?

Ejercicio 4.38 Tanto las tablas de dispersión, como los árboles trie, AVL y B se utilizan para representar conjuntos y diccionarios. Compara de forma crítica las ventajas e inconvenientes de cada uno respecto de los demás. Indica el tipo de aplicaciones en las que resulta más adecuado cada tipo de estructura.

Referencias bibliográficas

Muchas de las estructuras estudiadas en este capítulo se pueden encontrar en los capítulos 5 y 11 de [Aho88]. Por ejemplo, en el apartado 5.3 se describe la estructura de árboles trie. En el capítulo 7 de [Drozdek01] se desarrolla una implementación de tries en Java, incluyendo un completo estudio de su aplicación a un corrector ortográfico interactivo.

La estructura de relaciones de equivalencia, mediante árboles con punteros al padre, se puede encontrar en varias referencias con distintos nombres. En [Aho88], en el apartado 5.5, aparece como conjuntos con Combina y Encuentra; en [Weis95], en el capítulo 8, se habla del TAD conjunto ajeno (disjunto); y en [Baase00], en el apartado 6.6, se denomina el problema como relaciones de equivalencia dinámicas. En estas dos últimas referencias se puede consultar un análisis más completo del caso promedio de la estructura y de la función de Ackerman.

En cuanto a los ABB balanceados y árboles AVL, las referencias más interesantes son [Weis95], en el apartado 4.4; [Wirth80], en el capítulo 4; y el capítulo 5 de [Hernández01]. En la siguiente página web se puede encontrar una animación interactiva de los árboles AVL:

<http://www.seanet.com/users/arsen/avltree.html>

Los árboles B son analizados en muchos libros de estructuras de datos y algoritmos, así como en referencias de bases de datos. Destacamos las referencias [Aho88], apartado 11.4; [Hernández01], apartado 6.2; [Wirth80], apartado 4.5; [Cormen90], capítulo 18. Finalmente, en el capítulo 7 de [Drozdek01] se analizan los árboles B así como las variantes B+ y B*. Hay que advertir que el concepto de “orden de un árbol B” puede cambiar de unas referencias a otras. Lo que nosotros hemos denominado orden 5, puede encontrarse en algunos sitios como orden 4 y en otros como orden 2. Por este motivo, hablamos de “orden p ”, refiriéndonos al número máximo de punteros de un nodo.

Capítulo 5

Grafos

Un grafo es una abstracción compuesta por dos tipos de elementos: objetos y relaciones entre los objetos. Los objetos son también llamados nodos o vértices. Las relaciones son siempre entre dos vértices y se denominan aristas. Existe una amplia variedad de problemas que se pueden modelar utilizando grafos, desde una red de ordenadores hasta un circuito eléctrico, con componentes y conexiones entre los mismos. En diferentes aplicaciones surgen distintas clases de grafos: dirigidos, no dirigidos, etiquetados, con pesos, acíclicos, etc. Afortunadamente, existen numerosos algoritmos sobre grafos, que resuelven los problemas que aparecen con más frecuencia. Estos algoritmos sirven como herramientas en las que apoyarnos para resolver problemas de la vida real que puedan ser descritos mediante grafos.

Objetivos del capítulo:

- Conocer y comprender la terminología usada en teoría de grafos, incluyendo: tipos de grafos, propiedades, conceptos y problemas típicos sobre grafos.
- Ser capaz de diseñar e implementar una estructura de datos para el tipo grafo –en sus distintas variantes (dirigidos, no dirigidos, etiquetados)– usando listas y matrices de adyacencia.
- Valorar críticamente las ventajas e inconvenientes de las representaciones de grafos mediante listas y matrices de adyacencia, y su influencia en la eficiencia de los algoritmos sobre grafos.
- Conocer y comprender el funcionamiento de una variedad de algoritmos clásicos sobre grafos (tales como los algoritmos de Prim, Kruskal, Dijkstra, Floyd y Warshall), razonando sobre las ideas subyacentes que aportan y analizando su complejidad computacional.
- Ser capaz de usar los algoritmos estudiados como herramientas prácticas para la resolución de problemas en un contexto genérico, a través de la transformación de un problema de interés en un problema sobre grafos.

Contenido del capítulo:

5.1.	Definiciones y terminología de grafos	193
5.1.1.	Definición y tipos de grafos	194
5.1.2.	Terminología de la teoría de grafos	196
5.1.3.	Árboles, grafos y multigrafos	198
5.1.4.	Especificación del tipo grafo	201
5.2.	Estructuras de representación de grafos	203
5.2.1.	Representación con matrices de adyacencia	204
5.2.2.	Representación con listas de adyacencia	206
5.2.3.	Comparación entre estructuras de representación	208
5.3.	Recorridos sobre grafos	211
5.3.1.	Búsqueda primero en profundidad	211
5.3.2.	Búsqueda primero en anchura	215
5.4.	Árboles de expansión de coste mínimo	217
5.4.1.	Algoritmo de Prim	218
5.4.2.	Algoritmo de Kruskal	221
5.5.	Problemas de caminos mínimos	223
5.5.1.	Caminos mínimos empezando por un origen	224
5.5.2.	Caminos más cortos entre todos los vértices	229
5.6.	Algoritmos sobre grafos dirigidos	234
5.6.1.	Componentes fuertemente conexos	235
5.6.2.	Grafos dirigidos acíclicos	239
5.6.3.	Flujo máximo en redes	243
5.7.	Algoritmos sobre grafos no dirigidos	247
5.7.1.	Puntos de articulación y componentes biconexos	248
5.7.2.	Circuitos de Euler	251
5.8.	Otros problemas con grafos	253
5.8.1.	Ciclos hamiltonianos	254
5.8.2.	Problema del viajante	255
5.8.3.	Coloración de grafos	256
5.8.4.	Isomorfismo de grafos	257
	Ejercicios resueltos	259
	Ejercicios propuestos	266
	Cuestiones de autoevaluación	270
	Referencias bibliográficas	271

5.1. Definiciones y terminología de grafos

En los dos capítulos anteriores hemos analizado estructuras de representación de conjuntos y los diccionarios. Un conjunto, o un diccionario, contienen una colección de elementos independientes entre sí. Pero en muchas aplicaciones existe cierta relación entre los objetos que interesa modelar; y no sólo eso, sino que la relación resulta más relevante para el problema que los objetos por separado. En esos casos, surge de forma natural el tipo de datos abstracto grafo. Vamos a verlo con algunos ejemplos de problemas.

- a) **Mapa de carreteras.** Una agencia de transportes dispone de una flota de camiones, que distribuyen pimientos a nivel nacional. Se necesita almacenar un plano de carreteras con rutas y distancias entre ciudades, para conocer los caminos más cortos.
- b) **Planificación de tareas.** En el proyecto de construir una pirámide hay 12.000 tareas involucradas. Existen precedencias: algunas tareas deben hacerse antes que otras. Necesitamos planificar las tareas para tardar el menor tiempo total posible.
- c) **Equipo de fútbol.** Un entrenador quiere ganar la liga a toda costa. Para ello, estudiamos los rivales analizando la circulación del balón entre los jugadores. El objetivo es encontrar cuál es el jugador clave del equipo rival, para intentar anularlo.
- d) **Autómata finito.** Representamos expresiones regulares mediante autómatas finitos deterministas. El autómata tiene un conjunto de estados y transiciones entre estados. El objetivo es, dada una entrada, ver si es una expresión válida del autómata.

¿Cuál es la relación entre las cuatro aplicaciones anteriores? En principio, ninguna. Pero veamos las representaciones gráficas de los problemas en las figuras 5.1 y 5.2.

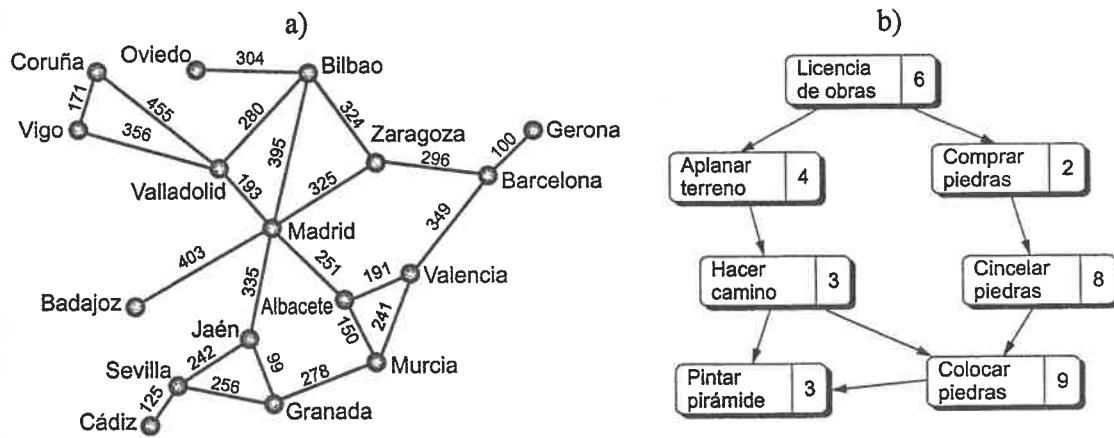


Figura 5.1: Representación de problemas mediante grafos. a) Mapa de carreteras. ¿Cuál es la distancia mínima de Cádiz a Gerona? b) Grafo de planificación de tareas. El número indica los meses que requiere la tarea. ¿En cuánto tiempo se puede construir la pirámide?

Si analizamos detenidamente la información manejada por los cuatro problemas, encontramos las dos similitudes clave: en todos ellos tenemos un conjunto de elementos –que pueden ser ciudades, tareas, jugadores o estados– y relaciones entre esos elementos –caminos entre ciudades, precedencias entre tareas, pases entre jugadores o transiciones entre estados–. Así que, moviéndonos hacia un nivel de abstracción superior, tenemos

un nuevo tipo abstracto de datos, formado por dos clases de componentes: objetos y relaciones entre los mismos. A este tipo lo llamamos **grafo**.

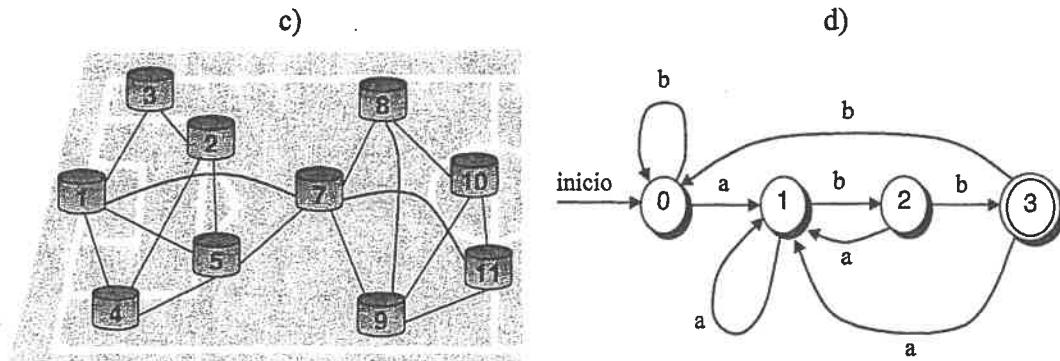


Figura 5.2: Representación de problemas mediante grafos. c) Estrategias de pase del balón de un equipo. ¿Qué jugador hay que eliminar para descolocar al equipo? d) Autómata finito determinista. La secuencia: abbaabba, ¿es una expresión válida?

El estudio teórico de problemas sobre grafos, de forma genérica, nos servirá para desarrollar una serie de algoritmos que después se podrán aplicar sobre problemas de la vida real. Por ejemplo, de los enunciados anteriores, el problema a) es un típico caso de caminos mínimos en grafos, para el cual existen los algoritmos clásicos de Dijkstra y Floyd; el problema b) es un caso de utilización de grafos dirigidos acíclicos, donde lo interesante es el cálculo de los caminos más largos; el problema c) está relacionado con la conectividad en grafos, y se puede resolver buscando puntos de articulación del grafo; finalmente, el problema d) es un caso particular de recorridos y caminos en grafos.

5.1.1. Definición y tipos de grafos

Una vez con la idea intuitiva de lo que es un grafo, vamos a ver la definición formal.

Definición 5.1 Un **grafo** $G = (V, A)$ es un par formado por un conjunto de **vértices**, **nodos** o **puntos**, V , y un conjunto de **arcos** o **aristas**, A . Cada arista de A es un par (v, w) , donde v y w pertenecen a V .

Por lo tanto, cuando trabajamos sobre grafos debemos definir dos cosas: 1) qué representan los nodos, y 2) qué representan las aristas. Por ejemplo, un uso clásico de los grafos es la representación de algoritmos mediante diagramas de flujo, como el mostrado en la figura 5.3. ¿Cómo se define el grafo? Pues definiendo los nodos y los arcos. Primero: los nodos representan instrucciones, o bloques de instrucciones que se ejecutan secuencialmente. Segundo: las aristas representan el flujo de control desde un bloque hasta otro.

En algunos problemas, los nodos que componen una arista deben ser distintos, es decir, (v, v) no es considerada una arista válida. Por ejemplo, no es posible que un jugador juegue consigo mismo, o que una tarea sea precedente de ella misma¹. En otros casos, la

¹En cuyo caso, nunca se podría ejecutar, ya que la precedencia implica que una tarea sólo puede empezar cuando haya acabado la predecesora.

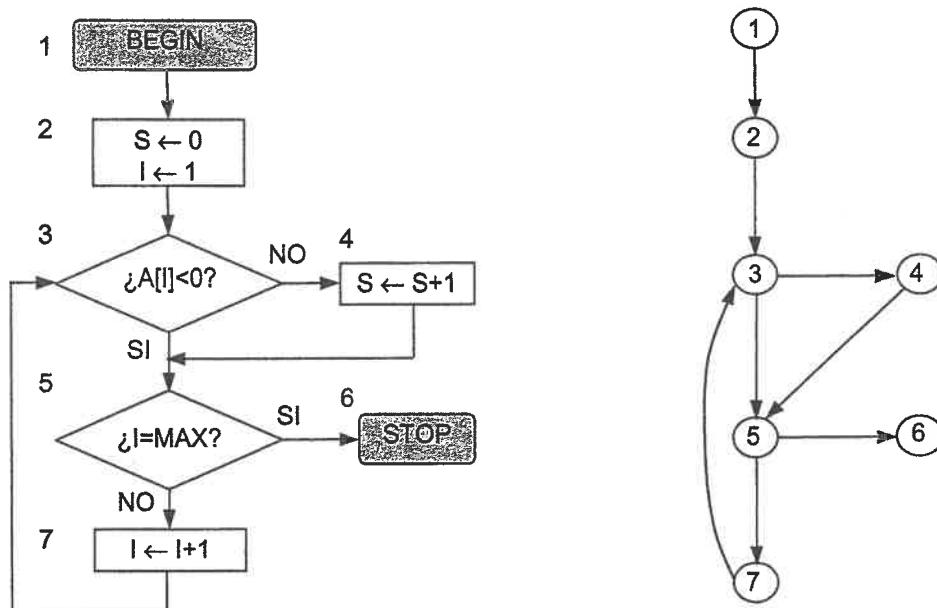


Figura 5.3: Diagrama de flujos de un algoritmo (izquierda) y el grafo correspondiente (derecha).

existencia de aristas del tipo (v, v) puede estar permitida. Por ejemplo, en el autómata es perfectamente posible una transición de un estado hacia sí mismo.

Grafos dirigidos y no dirigidos

Según cómo sean las aristas, distinguimos dos tipos de grafos:

- **Grafos no dirigidos.** Las aristas son pares no ordenados, es decir $(v, w) = (w, v)$.
- **Grafos dirigidos o digrafos.** Las aristas son pares ordenados, es decir $(v, w) \neq (w, v)$.

Para distinguir las aristas, se suele usar la notación $\langle v, w \rangle$ en grafos dirigidos y (v, w) en grafos no dirigidos. Los grafos no dirigidos se utilizan para representar relaciones *simétricas*. Por ejemplo, las relaciones “existe carretera entre” y “juega con” son relaciones simétricas entre ciudades y jugadores, respectivamente. Siempre que se dé en un sentido, se dará también en el otro. Por lo tanto, los grafos de las figuras 5.1a) y 5.2c) son grafos no dirigidos. Gráficamente, la arista se representa mediante una línea.



Los grafos no dirigidos representan relaciones no simétricas, en la cual los nodos de una arista $\langle v, w \rangle$ desempeñan papeles distintos. Por ejemplo, las relaciones “ v precede a

w " y "transición de v a w " son no simétricas. En el primer caso, v es el nodo antecesor y w el sucesor. En el segundo caso, v es el estado inicial y w el estado final. En consecuencia, los grafos de las figuras 5.1b), 5.2d) y 5.3 son grafos dirigidos.

Si $\langle v, w \rangle$ es una arista de un grafo dirigido, entonces decimos que v es la **cola** de la arista y w es la **cabeza**. Gráficamente, la arista es representada mediante una flecha que va desde v hasta w .



Grafos etiquetados, no etiquetados y grafos con pesos

Normalmente, cada vértice de un grafo lleva asociada una etiqueta; por ejemplo, en la figura 5.1b) el nombre de la tarea y el tiempo que se tarda en ejecutarla. Si también las aristas pueden estar etiquetadas, entonces hablamos de grafos etiquetados.

Definición 5.2 Un **grafo etiquetado** $G = (V, A, W)$ es una tripla en la que (V, A) es un grafo, y W es una función que a cada arco de A le asigna una etiqueta de cierto tipo T , es decir, $W : A \rightarrow T$. Si $a \in A$, $W(a)$ será la etiqueta del arco.

Si el tipo T de las etiquetas es un tipo numérico (naturales, enteros, reales, etc.) entonces decimos que se trata de un **grafo con pesos** o una **red**. Por ejemplo, el grafo de la figura 5.2d) es un grafo etiquetado con caracteres, y el de la figura 5.1a) es un grafo con pesos.

5.1.2. Terminología de la teoría de grafos

La terminología utilizada en teoría de grafos es bastante extensa, debido a las numerosas aplicaciones de los grafos. Vamos a introducir algunos de los conceptos más importantes. Afortunadamente, la mayoría de ellos representan ideas bastante intuitivas y fáciles de comprender.

Definición 5.3 Dado un grafo no dirigido $G = (V, A)$, si la arista $(v, w) \in A$, diremos que los vértices v y w son **adyacentes**, y que el arco (v, w) es **incidente** o **incide** en los vértices v y w . Si el grafo $G = (V, A)$ es dirigido y $\langle v, w \rangle \in A$, diremos que v es **adyacente a w** , y w es **adyacente de v** , y el arco $\langle v, w \rangle$ **incide** en w .

Por ejemplo, en el grafo de la figura 5.3 los nodos adyacentes a 3 son 4 y 5. Los nodos adyacente de 3 son 2 y 7. Normalmente, el concepto de "nodo adyacente a" es más común que el de "nodo adyacente de". Es más, al desarrollar los algoritmos sobre grafos supondremos que tenemos un iterador, que nos permite recorrer todos los nodos adyacentes a uno dado.

para cada nodo w adyacente a v hacer

Acción sobre w

finpara

Otro concepto muy importante sobre grafos son los caminos. La definición de camino es también intuitiva.

Definición 5.4 Un **camino** de un vértice u a v en un grafo $G = (V, A)$, es una secuencia de vértices $u, w_1, w_2, w_3, \dots, w_p, v$, tal que $(u, w_1), (w_1, w_2), \dots, (w_{p-1}, w_p), (w_p, v) \in A$.

Es decir, el camino es un recorrido por los nodos, moviéndose a través de las aristas del grafo. La definición es igualmente válida para grafos dirigidos y no dirigidos.

Definición 5.5 La **longitud** de un camino es el número de arcos que lo forman, es decir, el número de nodos que lo forman menos 1.

Por ejemplo, en el camino de la definición 5.4 la longitud del camino sería $p + 1$. De forma parecida, podemos definir el coste de un camino en los grafos con pesos.

Definición 5.6 El **coste** de un camino $u, w_1, w_2, w_3, \dots, w_p, v$, en un grafo con pesos $G = (V, A, W)$ es la suma de los costes de los arcos que lo forman, es decir, $W(u, w_1) + W(w_1, w_2) + \dots + W(w_{p-1}, w_p) + W(w_p, v)$.

Muchos de los conceptos fundamentales sobre grafos están relacionados con los caminos, sus tipos, su existencia o no, etc. Por ejemplo, un camino se dice que es un **camino simple** si todos los vértices que lo constituyen, excepto quizás el primero y el último, son diferentes.

Definición 5.7 Un **ciclo** en un grafo dirigido es un camino en el cual el primer vértice y el último son iguales. En el caso de los grafos no dirigidos, se impone además la condición de que las aristas del camino sean distintas². El ciclo se dice que es un **ciclo simple** si, además de ser un ciclo, es un camino simple.

Por ejemplo, el grafo de la figura 5.3 posee varios ciclos; como por ejemplo las secuencias 3,5,7,3, y 3,4,5,7,3. El grafo de la figura 5.1b) no posee ningún ciclo, ¿por qué³? Un grafo sin ciclos se dice que es **acíclico**.

Definición 5.8 En un grafo no dirigido $G = (V, A)$, en el que $v, w \in V$, se dice que v y w están **conectados** si existe algún camino en el grafo G que vaya desde v a w . Esto implicaría que también existe un camino de w a v . Un grafo no dirigido se dice que es un **grafo conectado** o **conexo** si para todo par de vértices $v, w \in V$, existe un camino de v a w en G .

Por ejemplo, el grafo no dirigido de la figura 5.4a) es un grafo no conexo, compuesto por 8 nodos.

En los grafos dirigidos, al concepto de conexo se le añade la etiqueta “fuertemente”, ya que requiere la existencia de caminos en ambos sentidos.

Definición 5.9 Un grafo dirigido $G = (V, A)$, se dice que es un **grafo fuertemente conexo** si para todo par de vértices $v, w \in V$, existe un camino de v a w , y de w a v en G .

²Esto se hace para evitar, por ejemplo, que en un grafo no dirigido con solo una arista (v, w) , el camino: v, w, v sea considerado como un ciclo. No lo es.

³Resolveremos esta cuestión más adelante. Pero sería interesante plantearse qué consecuencias supondría la existencia de un ciclo en este grafo.

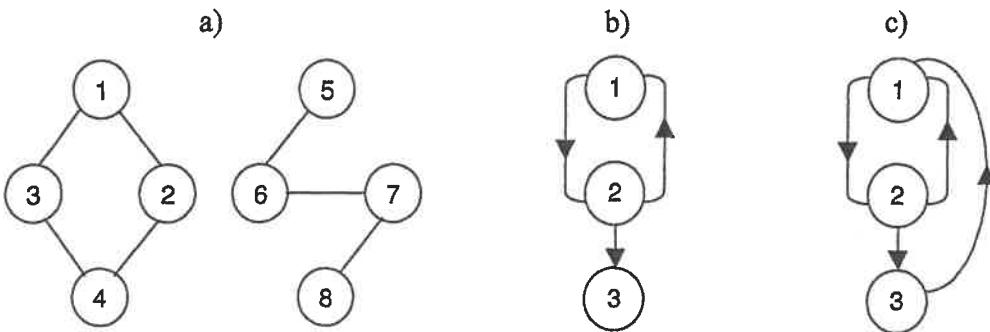


Figura 5.4: Grafos conexos e inconexos. a) Grafo no dirigido no conexo. b) Grafo dirigido no fuertemente conexo. c) Grafo dirigido fuertemente conexo.

El grafo de la figura 5.4b) no es fuertemente conexo, y el de la figura 5.4c) sí lo es. Al trabajar con varios grafos, por ejemplo G_1, G_2, G_3, \dots , se puede utilizar la notación $V(G_i)$, $A(G_i)$, para referirse a los vértices y a las aristas del grafo G_i , respectivamente. Por ejemplo, podemos definir el concepto de subgrafo de la siguiente forma.

Definición 5.10 Un subgrafo de un grafo G es un grafo G' tal que $V(G') \subseteq V(G)$ y $A(G') \subseteq A(G)$.

En la figura 5.5 se muestran dos grafos, G_1 y G_2 y algunos subgrafos suyos.

Dado un grafo no conexo, dirigido o no dirigido, puede ser interesante en algunas aplicaciones conocer sus subgrafos que son conexos.

Definición 5.11 Dado un grafo G , los **componentes conexos** de G (o **fuertemente conexos**, en grafos dirigidos) son todos los subgrafos conexos y maximales de G . La condición de **maximal** significa que es el mayor subgrafo que cumple la condición.

Por ejemplo, el grafo de la figura 5.4a) tiene dos componentes conexos, uno formado por los nodos $\{1, 2, 3, 4\}$ y otro por $\{5, 6, 7, 8\}$. El grafo G_2 de la figura 5.5 tiene dos componentes fuertemente conexos, uno el formado por $\{1, 2\}$ y el otro por $\{3\}$.

Definición 5.12 Se llama **grado** de un vértice al número de arcos que inciden en él. En un grafo dirigido se distinguen: **grado de entrada** de v , es el número de arcos de los que v es la cabeza; **grado de salida** de v , es el número de arcos de los que v es la cola.

Por ejemplo, en el grafo G_1 de la figura 5.5, todos los nodos tienen grado 3, mientras que el nodo 2 del grafo G_2 tiene grado de entrada 1 y de salida 2. Veremos algunos conceptos más sobre grafos, que definiremos según los vayamos necesitando.

5.1.3. Árboles, grafos y multigrafos

Los árboles se pueden considerar como un caso especial de grafos. Una misma estructura arbórea, como la de la figura 5.6b), se puede ver al mismo tiempo como un árbol

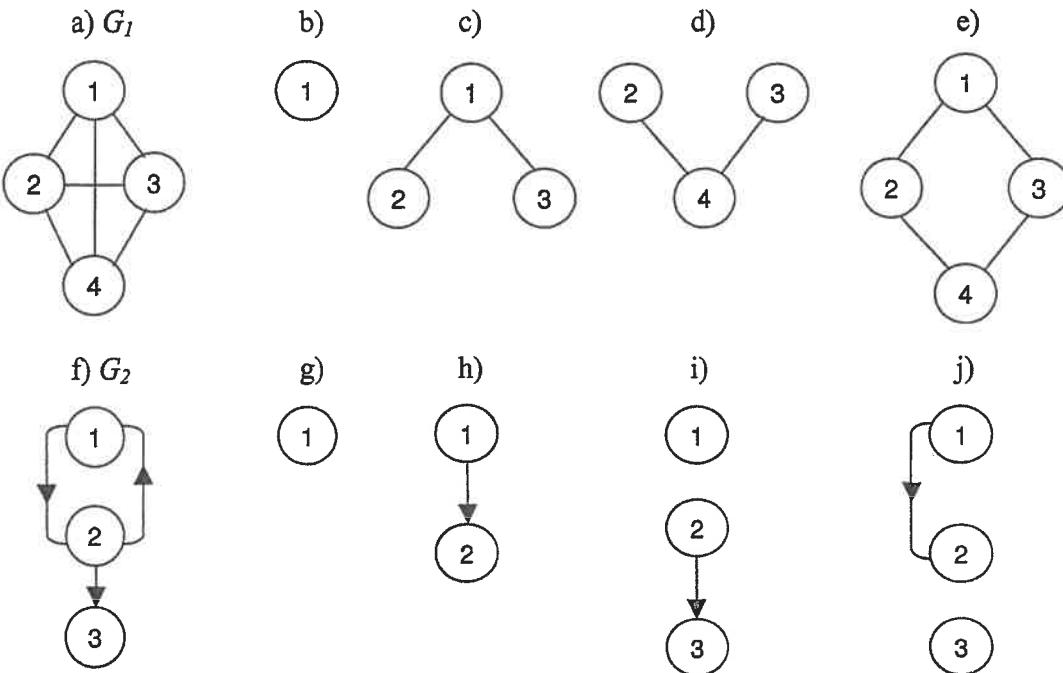


Figura 5.5: Grafos y subgrafos. a) Grafo no dirigido G_1 . b)-e) Algunos subgrafos de G_1 . f) Grafo dirigido G_2 . g)-j) Algunos subgrafos de G_2 .

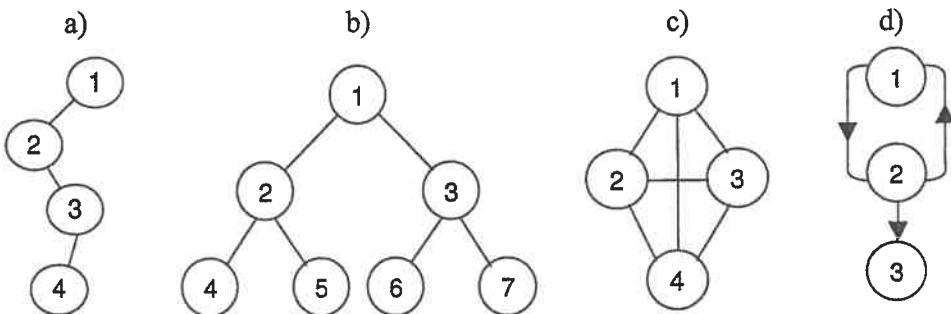


Figura 5.6: Listas, árboles y grafos. a) Una lista se puede interpretar como un árbol y como un grafo. b) Un árbol se puede ver como un grafo. c),d) La relación contraria no siempre es posible, si existen ciclos.

y como un grafo. En particular, muchos autores consideran que un árbol no es más que un grafo no dirigido y sin ciclos. Sin embargo, ambas cosas no son exactamente iguales. ¿En qué se diferencian?

La diferencia está en la raíz. En un árbol existe un nodo destacado que desempeña el papel de raíz del árbol. En un grafo todos los nodos tienen el mismo papel. En principio, no hay ningún nodo destacado, a menos que se indique explícitamente. Por ejemplo, en el grafo de la figura 5.6b) cualquier nodo podría hacer de raíz, no sólo el 1.

Teniendo en cuenta este detalle, podemos establecer una relación entre árboles y grafos, en la cual incluimos también las listas. Cualquier lista se puede considerar como un árbol, en el que cada nodo sólo tiene un hijo. A su vez, cualquier árbol se puede interpretar como un grafo no dirigido y sin ciclos.

A esta relación podemos añadir, como estructura más general, los multigrafos. Un multigrafo está compuesto por nodos y aristas, pero puede existir más de una arista entre dos nodos. En los grafos normales y corrientes, esto no es posible ya que A es definido como un *conjunto* de aristas y, por lo tanto, no se permite la repetición. En la figura 5.7 se muestra un ejemplo de multigrafo.

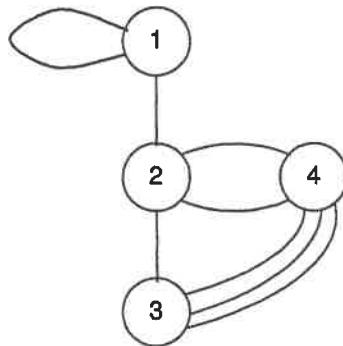


Figura 5.7: Un ejemplo de multigrafo no dirigido.

Si L es el conjunto infinito de todas las listas posibles, A el de los árboles, G el de los grafos y M el de los multigrafos, podemos decir que $L \subset A \subset G \subset M$. Todos los conceptos vistos sobre grafos pueden extenderse a los multigrafos. No obstante, en la práctica, la utilización de multigrafos no suele ser muy frecuente, ya que aportan poco interés respecto a los grafos.

Grafos con nombre propio

Vamos a ver algunos casos concretos de grafos que tienen nombre propio. Por ejemplo, un grafo se dice que es un **grafo completo** si tiene el máximo número de aristas posibles. En la figura 5.8 se muestran dos ejemplos de grafos completos.

El número máximo de aristas, que llamaremos a_{comp} , dependerá del número de nodos del grafo, de si es dirigido o no, y de si permitimos aristas del tipo (v, v) o no. Supongamos que el grafo G tiene n nodos. Entonces, si es no dirigido y se permiten aristas de un nodo consigo mismo: $a_{comp} = n(n + 1)/2$, y si no se permiten: $a_{comp} = n(n - 1)/2$. En el caso de los grafos dirigidos, si se permite la arista $\langle v, v \rangle$ entonces: $a_{comp} = n^2$ y en caso contrario: $a_{comp} = n(n - 1)$.

Según el número de aristas de un grafo, a , en relación al máximo posible, a_{comp} , decimos informalmente que el grafo está más o menos conectado. Por ejemplo, en un grafo completo $a = a_{comp}$ y decimos que es el más conectado posible. Un grafo donde $a \ll a_{comp}$, decimos que está poco conectado. Este tipo de grafos con pocas aristas también se conoce como **grafo escaso o disperso**.

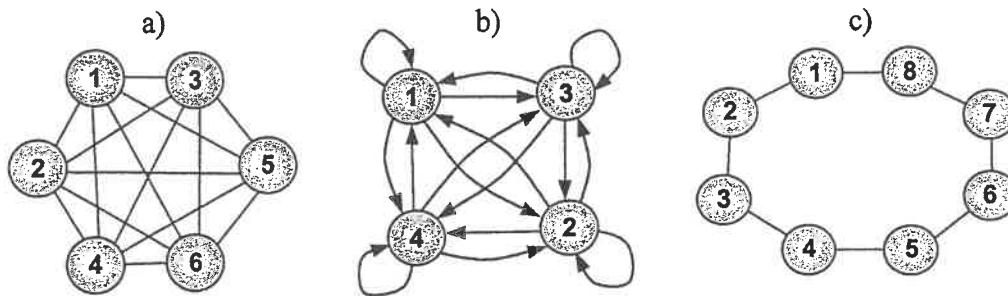


Figura 5.8: Grafos completos y anillo. a) Grafo no dirigido completo de 6 nodos, suponiendo que no se permiten las aristas (v, v) . b) Grafo dirigido completo de 4 nodos, si permitimos (v, v) . c) Grafo no dirigido con estructura de anillo.

Un grafo se llama un **anillo** si existe un ciclo simple que recorre todos sus nodos, y no existen más aristas que las de ese ciclo. La idea es bastante intuitiva. Se puede ver un ejemplo en la figura 5.8c). Un anillo es un ejemplo de grafo escaso.

Un grafo G se dice que es un **grafo multietapa** si el conjunto de nodos se puede partir en p subconjuntos disjuntos $V(G) = V_1 \cup V_2 \cup \dots \cup V_p$, de forma que todas las aristas $(v, w) \in A(G)$ son tales que si $v \in V_i$ entonces $w \in V_{i+1}$. Igual que antes, se puede comprender mejor la definición a través de un ejemplo, el de la figura 5.9a). El concepto se puede aplicar tanto a grafos dirigidos como no dirigidos. El valor de p es llamado el número de etapas.

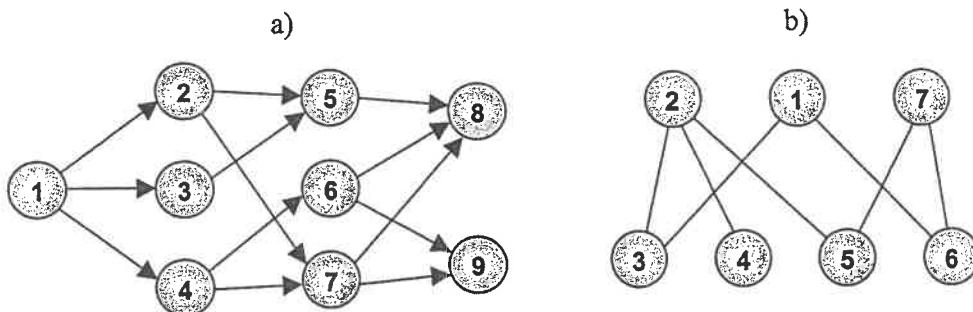


Figura 5.9: Grafos multietapa. a) Grafo dirigido multietapa, con 4 etapas: $\{1\}$, $\{2,3,4\}$, $\{5,6,7\}$, $\{8,9\}$. b) Grafo bipartito, con 2 etapas.

Finalmente, decimos que un grafo no dirigido es **bipartito** si es un grafo multietapa con $p = 2$ etapas. Es decir, el conjunto de vértices se puede descomponer en dos subconjuntos, de manera que todas las aristas unen nodos de subconjuntos distintos.

5.1.4. Especificación del tipo grafo

¿Existe el TAD grafo? Realmente no existe un único TAD grafo, sino una variedad de tipos abstractos. El hecho de que el grafo sea dirigido o no, etiquetado o no, etc.,

no son cuestiones de implementación, sino que dan lugar a diferentes abstracciones. De esta forma, podemos distinguir los siguientes TAD: grafo no dirigido no etiquetado, grafo no dirigido etiquetado, grafo no dirigido con pesos, grafo dirigido no etiquetado, grafo dirigido etiquetado, grafo dirigido con pesos, etc. En algunos casos, puede ser también interesante el tipo grafo dirigido acíclico, con sus variantes etiquetadas o no.

Según la categoría de grafo, tendrá sentido una clase de operaciones u otras. Por ejemplo, las operaciones para insertar, eliminar o consultar vértices y aristas, existirán en todos los tipos de grafos. Otras operaciones, como las de cálculo de caminos más cortos, sólo aparecerán en grafos con pesos; y los problemas de árboles de expansión de mínimo coste sólo aparecerán en grafos no dirigidos con pesos.

Ejemplo 5.1 Vamos a ver como ejemplo un trozo de la especificación informal del TAD grafo no dirigido con pesos. En los ejercicios veremos algunos ejemplos de especificación formal algebraica del tipo grafo.

TAD GrafoNDP[T: TipoNumerico] es Crear, InsertaNodo, ExisteNodo, SuprimeNodo, InsertaArista, ExisteArista, PesoArista, SuprimeArista, AdyacentesA, AdyacentesDe, CaminoMinimo, ExistenCiclos, ...

Requiere

El tipo T es un tipo numérico, que tiene operadores de suma, comparación, cero y valor NULO.

Descripción

Los valores de tipo $GrafoNDP[T]$ son grafos no dirigidos con pesos de tipo T , y con representación mutable. Los grafos se crean con la operación Crear. Después de esto, se pueden añadir vértices y aristas, sin restricciones de tamaño. Existen operaciones para conocer los caminos mínimos entre dos nodos, comprobar si existen ciclos, etc. Los nodos se referencian mediante un número entero: 1, 2, 3, etc.

Operaciones

Operación Crear (sal GrafoNDP[T])

Calcula: Devuelve un grafo nuevo, que no contiene vértices ni aristas.

Operación InsertaNodo (ent G: GrafoNDP[T]; nodo: entero)

Modifica: G

Calcula: Inserta el nodo $nodo$ en el grafo G . Si ya existía no se modifica el grafo.

Operación ExisteNodo (ent G: GrafoNDP[T]; nodo: entero; sal booleano)

Calcula: Devuelve *verdadero* si $nodo$ es un nodo del grafo G y *falso* en caso contrario.

Operación SuprimeNodo (ent G: GrafoNDP[T]; nodo: entero)

Requiere: ExisteNodo($G, nodo$) debe ser *verdadero*.

Modifica: G

Calcula: Elimina $nodo$ del grafo G .

Operación InsertaArista (ent G: GrafoNDP[T]; v, w: entero; peso: T)

Requiere: ExisteNodo(G, v) y ExisteNodo(G, w) deben ser *verdadero*.

Modifica: G

Calcula: Inserta en el grafo G la arista (v,w) , con coste $peso$. Si ya existía la arista, se actualiza el peso.

Operación ExisteArista (ent G: GrafoNDP[T]; v, w: entero; sal booleano)

Calcula: Devuelve verdadero si la arista (v,w) pertenece al grafo G .

Operación PesoArista (ent G: GrafoNDP[T]; v, w: entero; sal T)

Calcula: Devuelve el coste de la arista (v,w) , o NULO si no existe la arista.

.....
Fin GrafoNDP.

5.2. Estructuras de representación de grafos

Después de estudiar la notación utilizada en teoría de grafos, vamos a centrarnos en los dos problemas que nos incumben como programadores: la representación de grafos y la resolución de problemas sobre grafos. Consideremos los grafos de la figura 5.10.

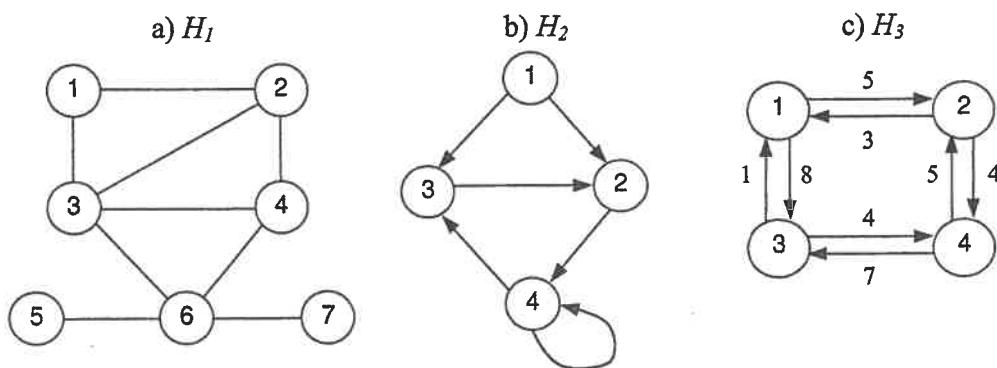


Figura 5.10: Ejemplos de grafos. a) No dirigido. b) Dirigido. c) Dirigido y etiquetado.

Los conjuntos de vértices y de arcos de los grafos H_1 , H_2 y H_3 serían:

$$V(H_1) = \{1, 2, 3, 4, 5, 6, 7\}; A(H_1) = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 6), (5, 6), (6, 7)\}$$

$$V(H_2) = \{1, 2, 3, 4\}; A(H_2) = \{<1, 3>, <1, 2>, <3, 2>, <2, 4>, <4, 3>, <4, 2>\}$$

$$V(H_3) = \{1, 2, 3, 4\}; A(H_3) = \{<1, 2>, <2, 1>, <2, 4>, <4, 2>, <3, 4>, <4, 3>, <3, 1>, <1, 3>\}$$

En consecuencia, puesto que los grafos no son más que pares de conjuntos –uno de vértices y otro de aristas– podríamos representarlos usando cualquiera de las estructuras de conjuntos vistas en los dos capítulos anteriores. Si los nodos o las aristas pueden tener etiquetas, entonces tendríamos diccionarios en lugar de conjuntos.

Sin embargo, es poco probable que una representación que almacene todas las aristas en el mismo conjunto funcione eficientemente. En la práctica, el diseño de la estructura de datos debe tener en cuenta el tipo de operaciones que se necesitan. De esta forma, la estructura es construida para conseguir eficiencia en las operaciones que más interesan. En particular, las operaciones que aparecerán típicamente serán del estilo: saber si existe una arista entre dos nodos dados, recorrer todos los nodos adyacentes a uno dado, los nodos adyacentes de uno dado, etc.

Vamos a analizar las dos formas más utilizadas para representar el conjunto A de aristas: mediante matrices y listas de adyacencia. La discusión es muy similar al problema de representar *relaciones muchos a muchos*, que analizamos en el capítulo 3, en la sección

3.3.2. Las posibilidades que se plantean son esencialmente las mismas y la mejor estructura depende de las características de cada aplicación particular.

5.2.1. Representación con matrices de adyacencia

Como ya discutimos en la representación de conjuntos, básicamente las dos estrategias opuestas en el diseño de una estructura de datos son: usar representaciones estáticas o dinámicas. En el caso de los grafos, la representación estática da lugar a la estructura de **matrices de adyacencia**. Sea un grafo $G = (V, A)$ con n vértices. La matriz de adyacencia es una matriz A –es decir, un array bidimensional– de tamaño $n \times n$, en la que cada posición $A[i, j]$ corresponde a la arista (i, j) . Si el grafo es no etiquetado, entonces la matriz será de booleanos. Tendremos $A[i, j] = \text{verdadero}$ si $(i, j) \in A(G)$, y $A[i, j] = \text{falso}$ en caso contrario. La definición del tipo sería inmediata.

tipo

Nodo = entero

GrafoNoEtiq = array [1..n, 1..n] de booleano

En la tabla 5.1 se muestran las matrices de adyacencia de los grafos H_1 y H_2 , que aparecen en la figura 5.10.

A_1	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	1	1	0	0	0
3	1	1	0	1	0	1	0
4	0	1	1	0	0	1	0
5	0	0	0	0	0	1	0
6	0	0	1	1	1	0	1
7	0	0	0	0	0	1	0

A_2	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	1	0	0
4	0	0	1	1

Tabla 5.1: Matrices de adyacencia de los grafos H_1 y H_2 de la figura 5.10. Para simplificar, se usa 1 y 0, en lugar de *verdadero* y *falso*.

Está claro que en los grafos no dirigidos la matriz de adyacencia será siempre simétrica, con el desperdicio de memoria que ello supone. Si fuera necesario ahorrar memoria, se podría almacenar sólo la parte de la matriz por encima de la diagonal principal.

¿Cómo modificar la representación si tenemos grafos etiquetados? En lugar de tener matrices de booleanos, usaríamos matrices del tipo de las etiquetas de las aristas. Si el grafo es de la forma $G = (V, A, W)$, con $W : A \rightarrow T$, la matriz de adyacencia sería de tipo T . En la posición $A[i, j]$ se almacenaría el valor de la etiqueta correspondiente.

tipo

Nodo = entero

GrafoEtiq[T] = array [1..n, 1..n] de T

El tipo T debería tener un valor especial, *NULO*, de manera que si no existe una arista (v, w) entonces $A[v, w] = \text{NULO}$. En los grafos con pesos, la matriz de adyacencia suele conocerse también como **matriz de costes**. En muchas aplicaciones que usan matrices de costes, el valor especial para una arista que no existe es un $+\infty$. Por ejemplo, si

el grafo representa caminos entre ciudades, el valor $+\infty$ indica que no existe esa carretera, por lo que el coste de pasar por ella sería infinito.

Como ejemplo, en la tabla 5.2 se muestra la representación del grafo H_3 de la figura 5.10c). En este caso no se supone que *NULO* sea $+\infty$.

A_3	1	2	3	4
1	-	5	8	-
2	3	-	-	4
3	1	-	-	4
4	-	5	7	-

Tabla 5.2: Matriz de adyacencia, o de costes, del grafo H_3 de la figura 5.10c). Para simplificar, se usa “-”, en lugar de *NULO*.

Implementación de la operaciones

La representación con matrices de adyacencia es útil en problemas sobre grafos en los cuales sea necesario saber si dos vértices están conectados o no. Conocer si existe una arista entre dos nodos dados se puede comprobar de forma inmediata.

operación ExisteArista (G : GrafoNoEtiq; v, w : nodo): booleano

devolver $G[v, w]$

Para recorrer los nodos adyacentes a un nodo v dado, simplemente habría que recorrer la fila correspondiente de la matriz. La implementación del iterador: **para cada** nodo w adyacente a v **hacer** Acción sobre w , sería la siguiente:

para $w := 1, \dots, n$ hacer

si $G[v, w] \neq \text{NULO}$ entonces

Acción sobre w

finsi

finpara

De forma similar, en el caso de los grafos dirigidos, para conocer las aristas que llegan a un nodo habría que recorrer la columna correspondiente a v . Por otro lado, si necesitamos conocer el número total de arcos que hay en el grafo –o la suma de los costes de todas las aristas, en los grafos con pesos– el algoritmo debería recorrer completamente la matriz, tardando un $O(n^2)$.

operación SumarCostesAristas (G : GrafoEtiq[T]): T

acum := 0

para $v := 1, \dots, n$ hacer

para $w := 1, \dots, n$ hacer

si $G[v, w] \neq \text{NULO}$ entonces

acum := acum + $G[v, w]$

finsi

finpara

finpara

devolver acum

Si la matriz que representa el grafo es dispersa o escasa, entonces se desperdicia mucha memoria con este tipo de representación. Y no sólo eso, sino que el tiempo de ejecución sólo depende del número de nodos, independientemente de que hayan más o menos aristas.

5.2.2. Representación con listas de adyacencia

Frente a las matrices de adyacencia, en la representación de grafos con listas de adyacencia sólo se almacenan las aristas que pertenecen al grafo. Pero, en lugar de almacenar todas las aristas en una lista única, se utiliza una lista para cada nodo con las aristas que salen del mismo. Esto es lo que se conoce como la **lista de adyacencia** de ese nodo.

Sea un grafo $G = (V, A)$ con n vértices. Para cada vértice del grafo, $v \in V(G)$, tenemos una lista de vértices adyacentes a v . Suponiendo que cada lista tiene un nodo 'cabecera', todos los nodos cabecera de todas las listas constituirán un array. La definición del tipo de datos podría ser como la siguiente:

tipo

Nodo = entero

GrafoNoEtiq = array [1..n] de Lista[Nodo]

Supongamos que el grafo tiene n nodos y a aristas. Si el grafo G es dirigido, necesitaremos n nodos cabecera y a celdas en las listas de adyacencia. Si es no dirigido, entonces el número de celdas en las listas será el doble, $2a$, ya que cada arista (v, w) es representada dos veces: en la lista de v y en la w . En la figura 5.11 se muestra la representación del grafo no dirigido H_1 de la figura 5.10, utilizando listas de adyacencia.

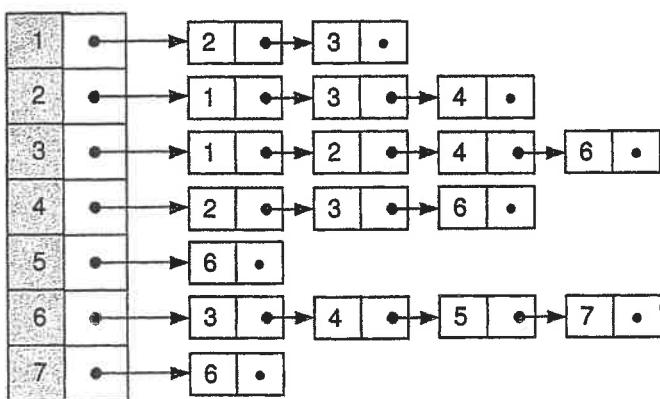


Figura 5.11: Listas de adyacencia del grafo H_1 de la figura 5.10a).

Si utilizamos grafos etiquetados, las listas deben incluir también la etiqueta asociada a cada arista. Supongamos que las etiquetas son de tipo T. La definición del tipo de datos sería la siguiente:

tipo

Nodo = entero

Arista[T] = registro

vert: Nodo

```

etiq: T
finregistro
GrafoEtiq[T] = array [1..n] de Lista[Arista[T]]

```

En la figura 5.12 se muestra un ejemplo de representación del grafo dirigido y con pesos, H_3 , de la figura 5.10c).

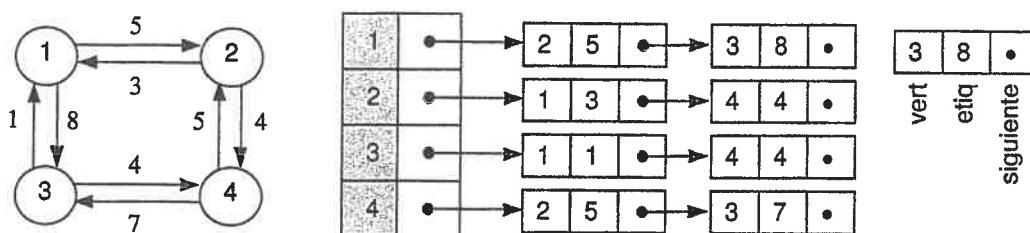


Figura 5.12: Grafo dirigido con pesos y lista de adyacencia correspondiente.

Implementación de la operaciones

Utilizando listas de adyacencia, las operaciones sobre el grafo se convierten en recorridos sobre las listas correspondientes. Por ejemplo, para consultar si existe o no una arista entre dos nodos dados (v, w), tendremos que buscar en la lista de adyacencia del nodo v . Supongamos que sobre el tipo `Lista[T]` tenemos definidas las operaciones: `Valor`, que devuelve el valor en una posición de la lista, y `Siguiente`, que devuelve el siguiente nodo de la lista. La implementación de `ExisteArista` podría ser como la siguiente:

operación ExisteArista (G : GrafoNoEtiq; v, w : Nodo): booleano

```

var /: Lista[Nodo]
/:= G[v]
mientras / ≠ NULO hacer
  si Valor(/) = w entonces
    devolver verdadero
  finsi
  /:= Siguiente(/)
finmientras
devolver falso

```

De forma similar, el iterador: **para cada** nodo w adyacente a v **hacer** Acción sobre w , se implementaría recorriendo la lista de adyacencia de v :

```

/:= G[v]
mientras / ≠ NULO hacer
  w:= Valor(/)
  Acción sobre w
  /:= Siguiente(/)
finmientras

```

Sin embargo, el recorrido de las aristas que llegan a un nodo sería mucho más complejo. Sería necesario recorrer todas las listas de adyacencia de todos los nodos, y

encontrar las aristas que llegan a ese nodo. Por ejemplo, suponiendo grafos dirigidos, la operación para calcular el grado de entrada de un nodo podría ser la siguiente:

operación GradoEntrada (G : GrafoNoEtiq; v : nodo): entero

```

cuenta:= 0
para w:= 1, ..., n hacer
    si ExisteArista( $G$ ,  $w$ ,  $v$ ) entonces
        cuenta:= cuenta + 1
    finsi
finmientras
devolver cuenta

```

Similarmente, una operación para sumar los costes de todas las aristas del grafo debería recorrer todas las listas, acumulando los costes.

5.2.3. Comparación entre estructuras de representación

Como en la mayoría de los casos, según el tipo de aplicación una estructura resultará mejor en ciertas situaciones y peor en otras. En general, las matrices de adyacencia serán mejores cuando los grafos sean de tamaño conocido y reducido. Las listas de adyacencia serán más adecuadas cuando los grafos puedan tener tamaño variable, y cuando tengan pocas aristas respecto al máximo posible, es decir si son grafos *escasos*.

Utilización de memoria

Consideremos grafos etiquetados, con n nodos y a aristas. Para analizar el uso de memoria, suponemos que el tamaño de un puntero es k_1 bytes y el de una etiqueta k_2 bytes. Para no introducir más variables, tomaremos que el tamaño de un entero es también k_1 bytes.

Con matrices de adyacencia la memoria total ocupada sería: k_2n^2 bytes, independientemente de lo que valga a . Con listas de adyacencia, cada celda de la lista ocuparía $2k_1 + k_2$ bytes. En los grafos dirigidos, existe una celda por cada una de las a aristas. En total tendríamos: $k_1n + (2k_1 + k_2)a$ bytes. En los no dirigidos las aristas están duplicadas, excepto para las que son de la forma (v, v) , si se permiten. En ese caso, la memoria sería aproximadamente $k_1n + 2(2k_1 + k_2)a$.

Por lo tanto, la comparación entre ambas estructuras depende de la relación entre a y n^2 . Si no tenemos en cuenta el término k_1n –correspondiente a los nodos cabecera en las listas de adyacencia– la proporción de a/n^2 para la cual ambas estructuras ocupan lo mismo sería: $k_2/(2k_1 + k_2)$. Por ejemplo, si las etiquetas ocupan lo mismo que los punteros, entonces la relación sería $1/3$. Es decir, si el número de aristas del grafo es mayor que $1/3$ del máximo, la matriz de adyacencia ocuparía menos espacio; y si es menor sería mejor usar listas de adyacencia.

En grafos poco conectados o escasos, donde $a \ll n^2$, la representación con listas de adyacencia siempre sería más adecuada. En la práctica, este tipo de grafos suelen ser más comunes. Supongamos, por ejemplo, que utilizamos grafos para representar una red mundial de ordenadores. Digamos que el conjunto de nodos contiene unos 10.000. Sin embargo, cada ordenador sólo está conectado directamente, por término medio, con otros

10 ordenadores. Entonces, $a = 100.000$, $n = 10.000$ y la proporción a/n^2 sería $1/1.000$. Es más, en una aplicación realista de este tipo se deben permitir operaciones para añadir o eliminar nodos. Mientras que esto resulta fácil con listas de adyacencia, con matrices no cabrían más nodos que el tamaño que tengamos reservado.

Eficiencia de las operaciones

Vamos a estudiar comparativamente la eficiencia de las operaciones con listas y matrices de adyacencia. Para algunas operaciones siempre será mejor usar listas, para otras será mejor con matrices y para otras dependerá del grafo sobre el que se aplica.

- **Consultar arista.** Las matrices de adyacencia permiten un acceso muy rápido, en un $O(1)$, a una arista particular. En las listas de adyacencia es necesario recorrer toda la lista; en el peor caso la lista será de tamaño n y el tiempo un $O(n)$. En el caso promedio, la lista sería de tamaño a/n y el tiempo un $O(1 + a/n)$.
- **Recorrer adyacentes.** Para recorrer los nodos adyacentes a uno dado, con matrices de adyacencia recorríamos la fila correspondiente, tardando siempre un $O(n)$. Con listas, el tamaño de la lista es precisamente el número de elementos que tenemos que recorrer. Igual que antes, en el peor caso será un $O(n)$ y en promedio un $O(1 + a/n)$.
- **Grado de entrada.** Cuando se trata de conocer las aristas que llegan a un nodo, con matrices simplemente cambiamos columnas por filas. El tiempo sería un $O(n)$. Pero con listas de adyacencia tenemos que recorrer todas las listas de todos los nodos. En total, deberíamos comprobar n nodos cabecera y a nodos de las listas, con lo que el tiempo sería un $O(n + a)$.
- **Contar todas las aristas.** El recorrido de todas las aristas supone un recorrido completo de la matriz de adyacencia, que necesitaría un $O(n^2)$. Con listas de adyacencia, ocurre igual que con la operación anterior: tendríamos que recorrer todas las listas de todos los nodos, tardando un $O(n + a)$.

En la tabla 5.3 se resume el tiempo de ejecución y el uso de memoria para los dos tipos de representaciones de grafos.

Por cuestiones de uso de memoria, la representación de listas de adyacencia se convierte en la única alternativa viable en muchas aplicaciones. El tiempo que proporciona esta representación es bastante “razonable” en todas las operaciones, excepto las que necesitan conocer las aristas que llegan hasta un nodo. ¿Cómo solucionarlo? Pues, igual que tenemos en las listas de adyacencia las aristas que salen de un nodo, podemos tener otras listas con las aristas que llegan a ese nodo.

En consecuencia, tenemos una estructura de listas múltiples, similar a las estudiadas en la sección 3.3.2. En la figura 5.13 se muestra un ejemplo de esta estructura de representación para un grafo dirigido. Cada celda asociada a una arista (v, w) pertenece a la vez a dos listas: la lista de salida de v y la lista de entrada de w .

Las listas utilizadas son circulares. La definición del tipo sería la siguiente:

tipo

clase_registro = enumerado (nodo, arista)

	Matrices de adyacencia	Listas de adyacencia
Memoria (bytes)	$k_2 n^2$	$k_1 n + (2k_1 + k_2)a$
Consultar arista	$O(1)$	$O(1 + a/n)$
Recorrer adyacentes	$O(n)$	$O(1 + a/n)$
Grado de entrada	$O(n)$	$O(n + a)$
Contar las aristas	$O(n^2)$	$O(n + a)$

Tabla 5.3: Comparación de memoria y tiempo de ejecución de algunas operaciones, para la representación de grafos con matrices y listas de adyacencia.

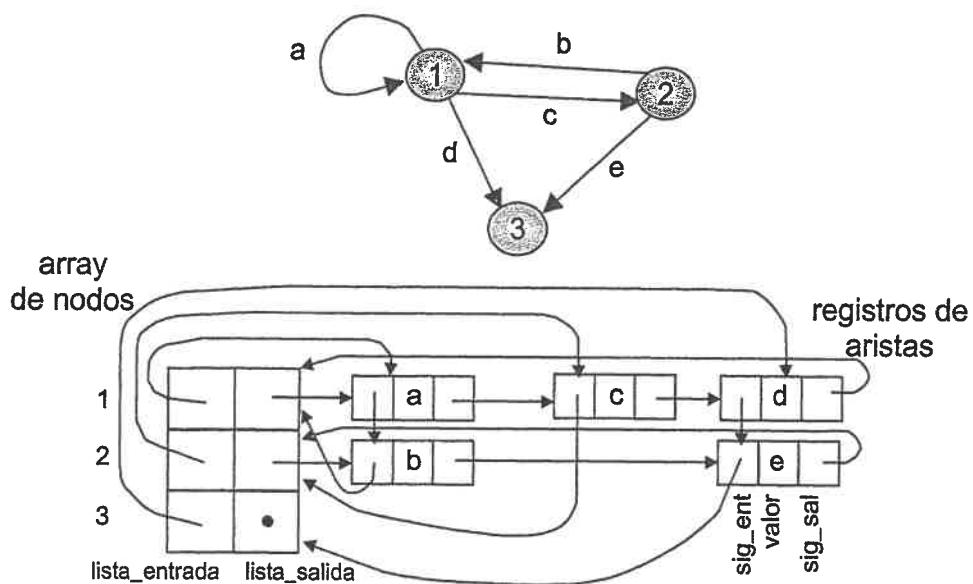


Figura 5.13: Grafo dirigido y etiquetado (arriba) y estructura de listas múltiples de adyacencia asociada (abajo).

```

tipo_registro[T] = registro
según clase: clase_registro
    nodo: (lista_entrada, lista_salida: Puntero[tipo_registro])
    arista: (valor: T; sig_ent, sig_sal: Puntero[tipo_registro])
finsegún
finregistro
GrafoEtiq[T] = array [1..n] de tipo_registro[T]
  
```

Utilizando listas de adyacencia múltiples, el tiempo de la operación para conocer el grado de entrada de un nodo sería, en promedio un $O(1 + a/n)$. Como contrapartida, se necesita más memoria para los punteros de las listas.

5.3. Recorridos sobre grafos

El recorrido de un grafo, ya sea dirigido o no dirigido, es un proceso mediante el cual se visitan sus vértices ordenadamente, según un orden dado por las aristas. Básicamente, la idea es muy parecida al recorrido sobre árboles; se parte de un vértice dado y sirven para visitar los vértices y los arcos de manera sistemática, moviéndose a través de las aristas del grafo.

Fundamentalmente, podemos encontrar dos tipos de recorridos sobre grafos:

- **Búsqueda primero en profundidad.** En un árbol, sería el equivalente a un recorrido en preorden. Se elige un nodo v de partida. Se marca como visitado y se recorren los nodos no visitados adyacentes a v , usando recursivamente la búsqueda primero en profundidad.
- **Búsqueda primero en amplitud o anchura.** Aplicado sobre un árbol, sería similar a recorrerlo por niveles: primero la raíz, luego los hijos, después los nietos, etc. En un grafo, empezando por un nodo v , se visitan primero todos los nodos adyacentes a v , luego todos los adyacentes a los anteriores (es decir los que están a distancia 2) y no visitados, luego los que están a distancia 3, y así sucesivamente hasta recorrer todos los nodos.

En sí mismas, las operaciones de recorrido no tienen mucho interés. Si simplemente quisiéramos procesar todos los nodos, sin importar el orden, podríamos hacerlo empezando por el 1, 2, 3, etc. La utilidad de los recorridos es que pueden ser utilizados para resolver una amplia variedad de problemas sobre grafos. El orden de recorrido define un orden en el que se deben procesar los nodos para resolver problemas como la búsqueda de ciclos, de puntos de articulación, de componentes fuertemente conexos, etc.

Para no pasar por un mismo vértice del grafo más de una vez, se usa un array de marcas, en las que se indica si un vértice está visitado o no. Supondremos que este array marca es una variable global.

```

var
    marca : array [1..n] de enumerado (visitado, noVisitado)
. operación BorraMarcas
    para v:= 1, ..., n hacer
        marca[v]:= noVisitado
    finpara
```

5.3.1. Búsqueda primero en profundidad

La búsqueda primero en profundidad, como ya hemos comentado, es un proceso recursivo equivalente al recorrido en orden previo de un árbol. Se selecciona un vértice v del

grafo y, para cada vértice adyacente a v no visitado, se recorre invocando recursivamente la búsqueda primero en profundidad. Los nodos visitados se van marcando en el orden de ejecución del procedimiento recursivo. Este procedimiento es válido tanto para grafos dirigidos como no dirigidos. La diferencia se encuentra en lo que se considere como nodo adyacente a uno dado. La implementación de la búsqueda en profundidad sería como la siguiente:

```
operación bpp ( $v$ : Nodo)
    marca[ $v$ ]:= visitado
    para cada nodo  $w$  adyacente a  $v$  hacer
        si marca[ $w$ ] = noVisitado entonces
            bpp( $w$ )
        finsi
    finpara
```

En la figura 5.14 se muestra un ejemplo de la aplicación de bpp(a) sobre un grafo no dirigido. El orden de recorrido de unos nodos a partir de otros se puede ver como un árbol. Es lo que se conoce como el **árbol de expansión en profundidad**. La raíz del árbol es el nodo, a , para el cual se hace la llamada inicial. Los descendientes de un nodo v son todos los nodos w para los cuales v ejecuta la llamada recursiva bpp(w).

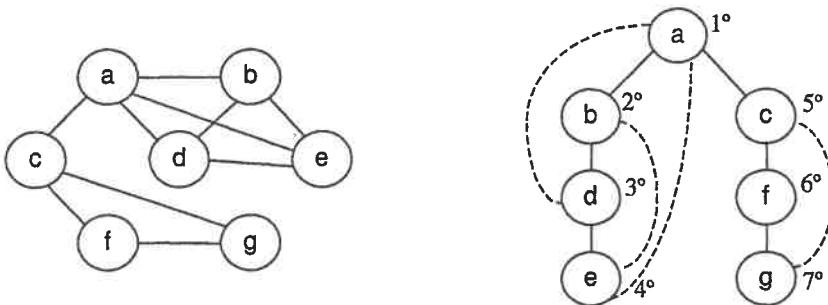


Figura 5.14: Grafo no dirigido (izquierda) y árbol de expansión en profundidad (derecha). Los números indican el orden en que son visitados los nodos.

Las aristas dibujadas con línea continua en la figura 5.14 son llamadas **arcos del árbol**. Se puede ver que aparecen también algunas aristas marcadas con línea discontinua. Estas aristas son los arcos que no son del árbol. Es decir, son aristas del grafo pero que no pertenecen al árbol de expansión. La existencia de estas aristas se comprobaría cuando la condición “**si** marca[w] = noVisitado ...” es falsa.

Ya que podría ocurrir que después de realizar la llamada a bpp haya vértices que aún no hayan sido visitados, el anterior procedimiento suele aparecer dentro de otro más general, que inicia las llamadas recursivas. En definitiva, el procedimiento de búsqueda primero en profundidad sería el siguiente:

```
operación BusquedaPrimeroEnProfundidad
    BorraMarcas
    para  $v := 1, \dots, n$  hacer
        si marca[ $v$ ] = noVisitado entonces
```

```

    bpp(v)
finsi
finpara

```

En la figura 5.15 se muestran dos ejemplos de recorridos en profundidad, sobre grafos dirigidos y no dirigidos. Si el procedimiento de búsqueda primero en profundidad da lugar a varios árboles de expansión, entonces hablamos de **bosque de expansión en profundidad**. Está claro que pueden existir distintas búsquedas en profundidad del mismo grafo. Dependiendo del nodo que visitemos en primer lugar y del orden de visita de los adyacentes, tendremos un bosque de expansión u otro.

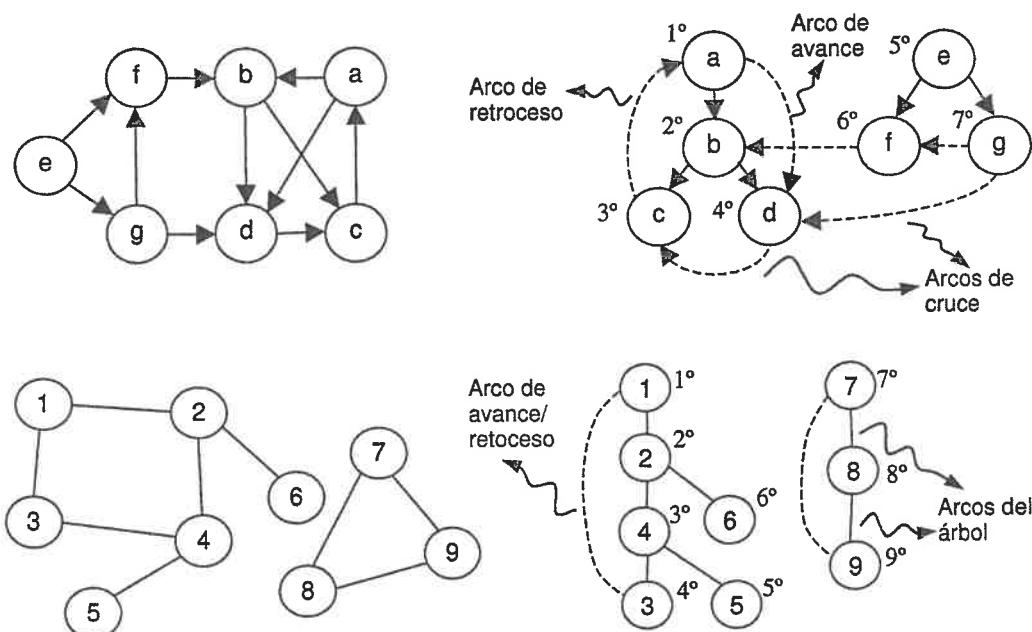


Figura 5.15: Dos grafos (izquierda) y sus bosques de expansión en profundidad (derecha). Los números indican el orden en que son visitados los nodos.

Arcos que no son del árbol

Tal y como se muestra en la figura 5.15, en los grafos dirigidos podemos clasificar los arcos que no son del árbol en tres tipos:

- Los **arcos de avance** van de un vértice a otro que es descendiente del primero dentro de un árbol; en el ejemplo de la figura 5.15, la arista $\langle a, d \rangle$.
- Los **arcos de retroceso** van de un vértice a otro que es ascendiente del primero dentro de un árbol; en el ejemplo de la figura 5.15, la arista $\langle c, a \rangle$.
- Los **arcos de cruce** son todos los demás; por ejemplo, las aristas $\langle d, c \rangle$, $\langle f, b \rangle$ y la $\langle g, d \rangle$.

En los grafos no dirigidos, todas las aristas son de **avance/retroceso**. No puede existir un arco de cruce de v a w , ya que en ese caso se habría realizado la llamada recursiva para w desde v .

Los arcos que no son del árbol pueden servir para diferentes propósitos. Por ejemplo, pueden utilizarse para realizar una **prueba de aciclicidad**, es decir, dado un grafo comprobar si tiene ciclos o no. En el caso de los grafos no dirigidos, habrá un ciclo sí y sólo si aparece algún arco que no sea del árbol. En los grafos dirigidos, tendremos un ciclo si encontramos en el recorrido algún arco $\langle w, v \rangle$ de retroceso⁴.

Ejemplo 5.2 ¿Cómo se puede comprobar si un arco es del árbol abarcador en profundidad y, en caso de no serlo, encontrar de qué tipo es?

Para hacerlo, sería necesario construir el bosque de expansión explícitamente. En concreto, el mismo array *marca* se podría utilizar para almacenar un conjunto de árboles representados mediante punteros al padre. En lugar de ser un array de booleanos, utilizaríamos un array de enteros. En cada posición *marca*[v], indicamos cuál es el padre de v , o bien con un 0 que es una raíz, o un -1 para decir que no está visitado. La inicialización en *BorraMarcas* sería a -1, y el procedimiento *bpp* se debería modificar del siguiente modo:

```

operación bpp2 ( $v$ : Nodo)
    si marca[ $v$ ] = -1 entonces
        marca[ $v$ ]:= 0 //  $v$  es una raíz
    finsi
    para cada nodo  $w$  adyacente a  $v$  hacer
        si marca[ $w$ ] = -1 entonces
            marca[ $w$ ]:=  $v$  //  $v$  es el padre de  $w$ 
            bpp2( $w$ )
        sino
            ArcoQueNoEsDelArbol( $v$ ,  $w$ )
        finsi
    finpar
```

En el procedimiento *ArcoQueNoEsDelArbol*(v , w) tenemos un arco (v, w) que no es del árbol, así que habría que comprobar el caso que ocurre. Con grafos dirigidos, si v es descendiente de w , tenemos un arco de retroceso; si v antecesor de w , tenemos un arco de avance; en otro caso tenemos un arco de cruce.

Si usamos grafos no dirigidos, tendremos un arco de avance/retroceso siempre que w no sea el parente de v en el árbol, es decir si $marca[v] \neq w$.

Tiempo de ejecución de la búsqueda en profundidad

Es difícil obtener el tiempo de ejecución de la búsqueda primero en profundidad, si tratamos de hacer el análisis de la forma tradicional. El procedimiento realiza llamadas recursivas, que dependen del número de adyacentes de cada nodo. Sin embargo, es posible “darle la vuelta” al análisis y encontrar la solución fácilmente. La idea se conoce

⁴ ¿Cuál sería el ciclo? Pues movernos desde v a w por las aristas del árbol, y luego desde w a v a través del arco de retroceso.

como analizar un algoritmo midiendo el trabajo total realizado. Es decir, no contamos las instrucciones según se ejecutan, sino que analizamos cuál será el resultado final del algoritmo.

En concreto, sabemos que la búsqueda en profundidad recorrerá cada nodo exactamente una vez, así que bpp se ejecutará n veces en total. Para cada una de esas ejecuciones, se marca el nodo como visitado y luego se recorren todos sus adyacentes. Ya vimos que el tiempo de recorrer los adyacentes depende de la representación. Con matrices de adyacencia, teníamos un bucle que recorre una fila de la matriz, de tamaño n . Para cada adyacente hay una comparación, que se hace en un tiempo constante. Por lo tanto, cada ejecución de bpp por sí misma tiene un $O(n)$. Multiplicando, el tiempo de ejecución de la búsqueda en profundidad con matrices de adyacencia es un $O(n^2)$.

Con listas de adyacencia, el recorrido de los adyacentes a un nodo implica recorrer una lista. Si el grafo tiene a aristas, el tamaño de cada lista es a/n en promedio. Por lo tanto, si contamos las n ejecuciones de bpp, tenemos que el tiempo con listas de adyacencia es un $O(n(1 + a/n)) = O(n + a)$.

¿Cuál es el trabajo total realizado? Con matrices de adyacencia, el algoritmo acaba recorriendo toda la matriz, ya sea en una u otra llamada. Además por cada celda de la matriz tarda un tiempo constante. De esta forma sale el $O(n^2)$ en matrices de adyacencia. En listas de adyacencia, el algoritmo siempre recorre todas las listas de todos los nodos, gastando también un tiempo constante en cada celda. En consecuencia, el tiempo con listas de adyacencia es proporcional al tamaño de estas listas, $O(n + a)$.

5.3.2. Búsqueda primero en anchura

Si la idea de la búsqueda primero en profundidad es movernos siempre lo más profundo posible, intuitivamente, en la **búsqueda primero en anchura** el recorrido se extiende a lo ancho. De esta forma, si partimos de un nodo v , lo marcamos como visitado en primer lugar. Despues se visitan todos los adyacentes a v . A continuación los adyacentes de estos últimos que no estén ya visitados, y así sucesivamente.

Para hacer el recorrido se utiliza una cola de vértices, que al principio contendrá sólo el nodo inicial. Básicamente, el funcionamiento de la búsqueda en anchura consiste en: sacar un elemento v de la cola, añadir a la cola los adyacentes de v no visitados y acabar cuando se vacíe la cola⁵.

Igual que con la bpp, hacemos uso de un procedimiento que se ocupa de arrancar las llamadas de bpa que sean necesarias. Las operaciones serían las siguientes:

```
operación BusquedaPrimeroEnAnchura
    BorraMarcas
    para  $v := 1, \dots, n$  hacer
        si marca[ $v$ ] = noVisitado entonces
            bpa( $v$ )
        finsi
    finpara
```

⁵Si nos fijamos un poco, la búsqueda en profundidad podría interpretarse de forma parecida, pero usando una pila en lugar de una cola. Ojo, además los nodos se deberían marcar al sacarlos de la pila, y no al meterlos.

```

operación bpa ( $v$ : Nodo)
var  $C$ : Cola[Nodo]
     $marca[v]$ := visitado
     $C$ := CrearCola
    Insertar( $C, v$ )
    mientras NO EsVacíaCola( $C$ ) hacer
         $x$ := Frente( $C$ )
        SuprimirCola( $C$ )
        para cada nodo  $w$  adyacente a  $x$  hacer
            si  $marca[w]$  = noVisitado entonces
                 $marca[w]$ := visitado
                InsertaCola( $C, w$ )
            finsi
        finpara
    finmientras

```

En la figura 5.16 se muestra la ejecución de la búsqueda en anchura para dos grafos. Podemos definir los **árboles** y **bosques de expansión en anchura**, de forma similar a los bosques de expansión en profundidad. En este caso, un nodo v es padre de w en el árbol, si la iteración de v introduce a w en la cola.

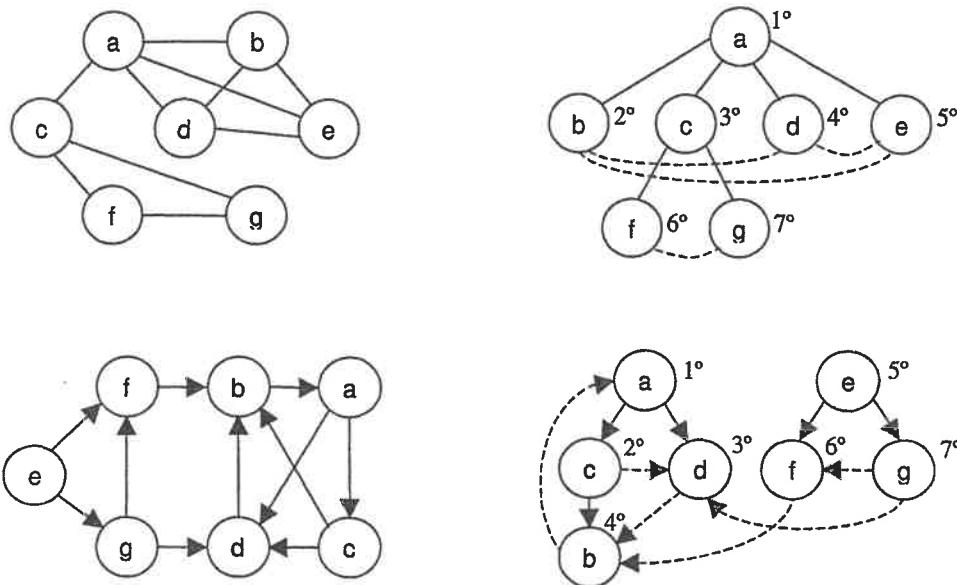


Figura 5.16: Dos grafos (izquierda) y sus bosques de expansión en anchura (derecha). Los números indican el orden en que son visitados los nodos.

Igual que antes, aparecen arcos del árbol y arcos que no son del árbol. Si antes, en grafos no dirigidos, los arcos que no son del árbol eran siempre de avance/retroceso con bpp, ahora sólo aparecen arcos de cruce. Es fácil encontrar la razón si tenemos en cuenta

el funcionamiento del algoritmo.

Tiempo de ejecución de la búsqueda en profundidad

El análisis del tiempo de ejecución es muy parecido al realizado para la búsqueda en anchura. Igual que antes, no seguimos la ejecución del algoritmo sino que consideramos el trabajo total realizado.

En este caso, sabemos que cada uno de los n vértices se visita una vez, se introduce en la cola exactamente una vez, se saca de la cola una vez y se visitan todos sus adyacentes una vez. Así que el bucle **mientras** se ejecuta n veces. El orden de complejidad del algoritmo viene dado por el tiempo que se necesite en recorrer todos los adyacentes. Por lo tanto, podemos decir lo mismo que en la búsqueda en profundidad. Con matrices de adyacencia, el algoritmo acaba recorriendo toda la matriz, dando lugar a un $O(n^2)$. Con listas de adyacencia, se recorren todas las listas, con un coste total de $O(n + a)$.

5.4. Árboles de expansión de coste mínimo

Como vimos en la introducción, existe una relación entre árboles y grafos. Un grafo no dirigido y sin ciclos se puede interpretar casi como un árbol⁶. De hecho, ya hemos usado el concepto de árbol de expansión de un grafo, para describir el orden de recorrido de los nodos en la búsqueda primero en profundidad y en anchura. Podemos definir formalmente el concepto de la siguiente forma.

Definición 5.13 Un **árbol de expansión** de un grafo no dirigido $G = (V, A)$ y conexo es un subgrafo $G' = (V, A')$ no dirigido, conexo y sin ciclos.

Es decir, el árbol de expansión contiene los mismos nodos que G y sigue siendo conexo, pero no tiene ciclos. En consecuencia, tiene el mínimo número posible de aristas para mantener conexo el grafo: $n - 1$, suponiendo un grafo de n nodos. Si el grafo es ponderado, podemos definir el **coste de un árbol de expansión** como la suma de los costes de sus aristas.

Dado un grafo no dirigido con pesos, pueden existir muchos árboles de expansión del mismo. El **problema del árbol de expansión de coste mínimo** de un grafo G consiste en encontrar, de todos los árboles de expansión de G , aquel que tenga el menor coste. Vamos a analizar un caso típico de aplicación donde surge el problema.

Ejemplo 5.3 Supongamos que tenemos varios ordenadores que queremos conectar en red. Hacemos un presupuesto de lo que nos costaría cada conexión punto a punto, obteniendo el grafo de la figura 5.17. No obstante, no todas las conexiones son necesarias, ya que dos ordenadores se pueden comunicar a través de otro. El objetivo es seleccionar las conexiones que debemos contratar con dos requisitos: que todos los ordenadores estén comunicados entre sí y que el coste requerido sea el mínimo.

Tenemos un problema de árbol de expansión de coste mínimo. Los ordenadores son los nodos y las conexiones las aristas. El resultado debe ser un árbol de expansión, para

⁶A no ser porque el árbol tiene raíz y el grafo no.

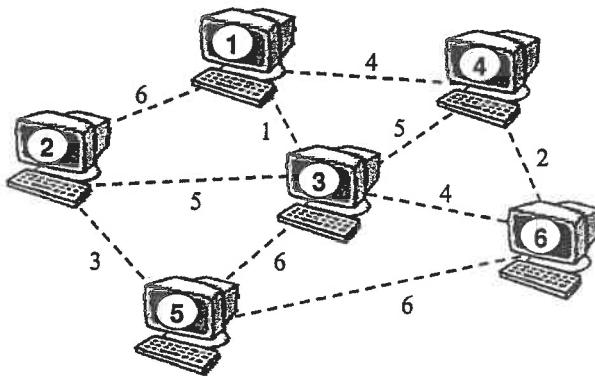


Figura 5.17: Una red de ordenadores, con posibles conexiones. Los números de las aristas indican los costes presupuestados para cada enlace, en miles de euros.

garantizar que no se usan más conexiones de las necesarias, pero garantizando que el grafo es conexo. Y debe ser el de coste mínimo para conseguir el segundo requisito del problema.

Existen dos algoritmos *clásicos* para encontrar el árbol de expansión de coste mínimo de un grafo: los algoritmos de Prim y de Kruskal, conocidos así por sus autores, R. C. Prim y J. B. Kruskal, que los propusieron a mediados del pasado siglo XX.

5.4.1. Algoritmo de Prim

El funcionamiento del algoritmo de Prim es bastante intuitivo. La idea es la siguiente:

- Se toma un vértice v cualquiera del grafo. Inicialmente, el árbol consta sólo de este nodo. Por ejemplo, en el grafo de la figura 5.17 digamos que cogemos el 1.
- Se estudian los pesos de los arcos entre v y todos los demás vértices, y se escoge el menor. Ahora el árbol consta de dos vértices y la arista que los une. En el ejemplo, cogeríamos el nodo 3, cuya arista tiene peso 1.
- Se analizan los costes de todos los arcos que unen cualquiera de esos dos vértices con el resto. De todos ellos se escoge el de menor coste. Ahora tenemos en el árbol tres nodos y dos aristas. En el ejemplo, la arista de menor coste con 1 ó 3 sería la arista (3,6) o la (1,4), ambas con coste 4.
- De esta forma, el proceso se repite sucesivamente hasta seleccionar todos los vértices del grafo, añadiendo en cada paso un vértice y una arista al árbol.

Aún antes de pasar a la implementación, es fácil comprobar manualmente el funcionamiento del algoritmo. En la figura 5.18 se muestra un ejemplo de aplicación.

Vemos que el algoritmo trabaja implícitamente con dos conjuntos de vértices: los que ya hemos incorporado al árbol de expansión y los que quedan por escoger. Sea U el conjunto de los vértices ya escogidos; los que quedan por estudiar serán $V - U$. Inicialmente, U consta sólo del vértice de partida. En cada paso del algoritmo, buscamos la

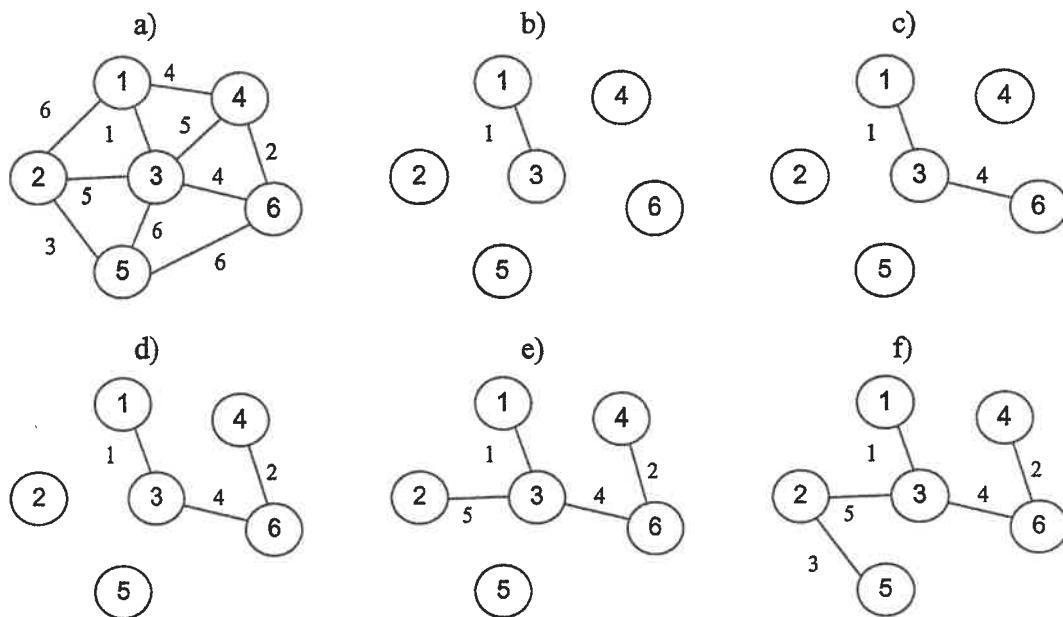


Figura 5.18: Aplicación del algoritmo de Prim. a) Grafo de entrada del algoritmo. b)-e) Pasos de aplicación del algoritmo. f) Árbol de expansión de coste mínimo resultante.

arista de menor coste que vaya de un nodo en U a otro en $V - U$. Informalmente, decimos que se busca el vértice de $V - U$ “más cercano”⁷ a alguno de los de U . El algoritmo acaba cuando hemos añadido todos los vértices a U , es decir, cuando $U = V$.

Implementación del algoritmo de Prim

Para calcular en cada paso la arista de menor coste entre un vértice de U y otro de $V - U$ vamos a utilizar dos arrays:

- *MásCercano[v]*, que da el vértice en U que está más cercano al vértice v de $V - U$.
- *MenorCoste[v]*, que da el coste de la arista entre v y *MásCercano[v]*.

Si empezamos por el nodo 1, entonces inicialmente el *MásCercano* para todos los nodos será el nodo 1, y *MenorCoste* será el coste de las aristas de cada nodo con 1.

En cada paso del algoritmo se recorre *MenorCoste*, hasta encontrar el vértice k de $V - U$ más cercano a U . Se añade al árbol la arista, que será $(k, \text{MásCercano}[k])$, y se actualizan los arrays, teniendo en cuenta que k pertenece ahora a U y no a $V - U$.

Para simplificar, en lugar de usar el tipo grafo trabajaremos directamente con una matriz de costes C , de tamaño $n \times n$, donde $C[i, j]$ es el coste de la arista (i, j) . Si no existe arista entre dos vértices, el valor de la posición correspondiente en la matriz será suficientemente grande como para no ser tenido en cuenta en ningún momento. Denotaremos este valor con “ $+\infty$ ”. Por otro lado, el resultado viene dado en un tipo

⁷Notar que esto supone interpretar los *costes* como *distancias*.

GrafoNDE[T], en el cual se suponen operaciones para añadir nodos y aristas. En definitiva, el algoritmo de Prim sería el siguiente.

```

operación Prim (C: array [1..n, 1..n] de real; var A: GrafoNDE[real])
var
    MenorCoste: array [1..n] de real
    MásCercano: array [1..n] de entero
        // Inicialización: el conjunto U sólo contiene el vértice 1
    A:= CrearGrafo
    InsertaNodo(A, 1)
    para i:= 2 hasta n hacer
        MenorCoste[i]:= C[1, i]
        MásCercano[i]:= 1
    finpara
        // Repetir n-1 veces: encuentra el vértice k de V-U más cercano a alguno de U
    para i:= 2 hasta n hacer
        min:= MenorCoste[2]      // Busca la arista con mínimo MenorCoste
        k:= 2
        para j:= 3 hasta n hacer
            si MenorCoste[j] < min entonces
                min:= MenorCoste[j]
                k:= j
            finsi
        finpara
        InsertaNodo(A, k)      // Inserta k al árbol de expansión
        InsertaArista(A, (k, MásCercano[k]))
        MenorCoste[k]:= +∞
        para j:= 2 hasta n hacer // Recalcular los costes de U, tras añadir k
            si (C[k, j] < MenorCoste[j]) Y (MenorCoste[j] ≠ +∞) entonces
                MenorCoste[j]:= C[k, j]
                MásCercano[j]:= k
            finsi
        finpara
    finpara

```

Análisis del tiempo de ejecución

El análisis del algoritmo de Prim se puede realizar fácilmente por partes. La inicialización del algoritmo lleva un tiempo de $O(n)$. A continuación, tenemos un bucle **para** *i*, que se repite $n - 1$ veces. Dentro de este bucle, tenemos varias instrucciones constantes y dos bucles **para** *j*, no anidados sino consecutivos. Estos bucles requieren claramente un $O(n)$. Por lo tanto, el orden total del algoritmo de Prim es $O(n^2)$.

La implementación propuesta trabaja con matrices de adyacencia. Sería posible hacer algunas modificaciones en el procedimiento para usar listas de adyacencia. Si lo hacemos así, podríamos ahorrarnos algunas instrucciones en el bucle de "recalcular los costes de *U*". En su lugar tendríamos un **para cada** nodo *j* adyacente a *k*.... El problema

es que la búsqueda del mínimo *MenorCoste* sigue necesitando $n - 2$ pasos, con lo que el tiempo seguiría siendo un $O(n^2)$.

5.4.2. Algoritmo de Kruskal

La idea subyacente al algoritmo de Kruskal es también intuitiva. Igual que en el algoritmo de Prim, el árbol de expansión de coste mínimo se construye paso a paso; en cada paso se añade una arista al árbol. El funcionamiento del algoritmo es el siguiente:

- Suponiendo que partimos de un grafo con pesos, $G = (V, A, W)$, comenzamos con un grafo $G' = (V, \emptyset)$. Es decir, el árbol está compuesto por todos los vértices de G y ningún arco.
- En cada paso del algoritmo, se elige la arista de menor coste de A de las que queden por estudiar. Esto es, se van comprobando las aristas de menor a mayor.
- Suponiendo que la arista que está siendo estudiada es (v, w) , entonces debemos comprobar si se puede añadir al árbol de expansión, G' . En concreto, se podrá añadir siempre que no forme un ciclo en G' . Si forma un ciclo, se descarta. En otro caso, se añade (v, w) a G' .
- Repetir el paso anterior hasta que G' sea conexo, es decir, hasta que hayamos añadido $n - 1$ aristas.

En la figura 5.19 se muestra un ejemplo de ejecución del algoritmo de Kruskal, para el mismo grafo de la figura 5.17.

Veamos la aplicación del algoritmo en la figura 5.19. Inicialmente, el árbol de expansión no tendría ninguna arista. Las estudiamos de menor a mayor. La $(1, 3)$ con coste 1 se puede añadir sin problemas (figura 5.19b); igual pasa con la $(4, 6)$ con coste 2 (figura 5.19c), la $(2, 5)$ con coste 3 (figura 5.19d), y la $(3, 6)$ con coste 4 (figura 5.19e). La siguiente arista con menor coste sería la $(1, 4)$ con coste 4. Pero si añadimos esta arista al grafo de la figura 5.19e) se formaría un ciclo. Por lo tanto, esta arista no debe ser añadida, se descarta. Pasamos a la siguiente, la $(3, 4)$ con coste 5. También formaría un ciclo, así que se descarta. A continuación tenemos la $(2, 3)$, también con coste 5, que se puede añadir sin formar un ciclo (figura 5.19f). Al añadirla ya tenemos $n - 1$ aristas, luego tenemos la solución.

En definitiva, la estructura del algoritmo de Kruskal sería como la siguiente. Utilizamos aquí un pseudocódigo de muy alto nivel, dando por supuesto que el alumno es capaz de programarlo por sí mismo.

operación Kruskal (G : GrafoNDE[real]; var A : GrafoNDE[real])

var C : Lista[arista]

$C :=$ ordenar de menor a mayor las aristas de G

$A :=$ CrearGrafo

$A :=$ InsertaNodos($1, 2, \dots, n$)

mientras A contenga menos de $n-1$ aristas **hacer**

elección la siguiente arista (v, w) de C

si la arista (v, w) no forma un ciclo en el grafo A **entonces**

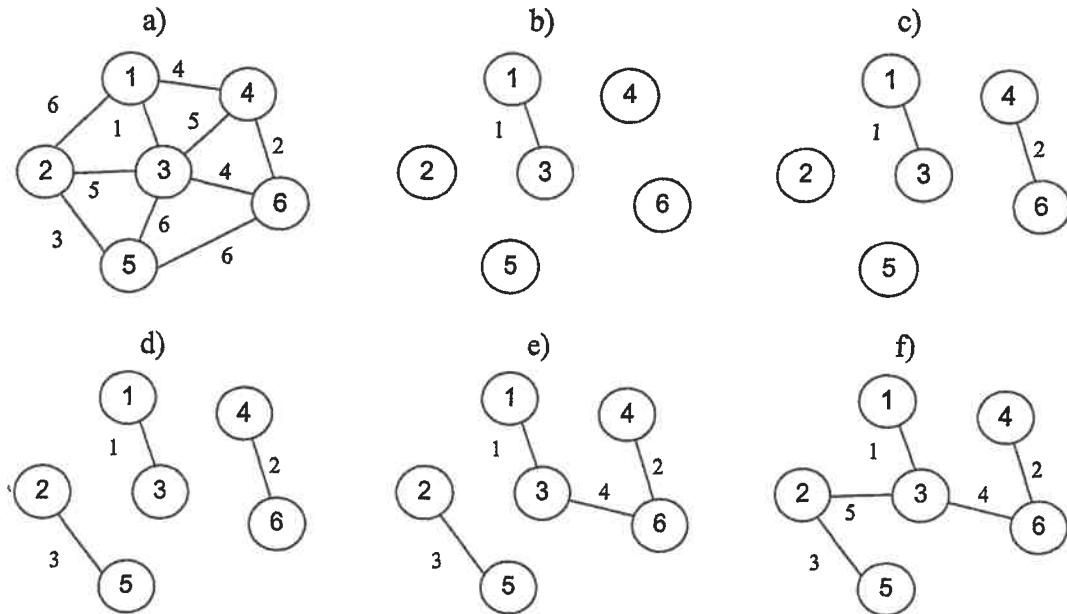


Figura 5.19: Aplicación del algoritmo de Kruskal. a) Grafo de entrada del algoritmo. b)-e) Pasos de aplicación del algoritmo. f) Árbol de expansión de coste mínimo resultante.

```

InsertaArista( $A, (v,w)$ )
finsi
finmientras
  
```

Comprobación de componentes conexos

La cuestión que nos queda por resolver en el algoritmo es ¿cómo comprobar rápidamente si la adición de una arista (v, w) provocará un ciclo o no? Podríamos añadir la arista y aplicar la prueba de aciclicidad vista en el apartado 5.3.1. Pero esa solución sería demasiado lenta. ¿Es posible hacer la comprobación en un tiempo constante?

Si nos fijamos en la figura 5.19e), una arista (v, w) dará lugar a un ciclo en el grafo G' , si antes de añadirla ya existía un camino de v a w , es decir si v y w están conectados. Esto ocurría al intentar añadir $(1, 4)$. Suponiendo que calculamos los componentes conexos de G' —esto es, los subconjuntos de nodos conectados— entonces podemos decir que no se formará un ciclo —y por tanto podemos añadir la arista— si v y w están en componentes conexos distintos.

Inicialmente, cuando empezamos con $G' = (V, \emptyset)$, cada nodo será un componente conexo por sí mismo. Cada vez que añadimos una arista juntando dos componentes conexos. Y, finalmente, el algoritmo acabará cuando sólo tengamos un componente conexo.

¿Cómo implementamos la comprobación “dos nodos están en el mismo componente conexo”? Si nos fijamos, la relación “estar en el mismo componente conexo” es una relación binaria de equivalencia: es *reflexiva* (un nodo está en el mismo componente que sí mismo), *simétrica* (si v está en el mismo componente que w , entonces w está en el mismo que v)

y *transitiva* (también se cumple trivialmente). Por lo tanto, para implementar la comprobación podemos usar la estructura de relaciones de equivalencia del capítulo 4, estudiada en el apartado 4.2. Suponiendo que la relación es almacenada en una variable R de tipo `RelacionEquiv[{1..n}]`, la comprobación:

si la arista (v, w) no forma un ciclo en el grafo A entonces

Se transformaría sencillamente en:

si $\text{Encuentra}(R, v) \neq \text{Encuentra}(R, w)$ entonces

Y al añadir una arista (v, w) a A , deberíamos unir dos clases de equivalencia distintas, ejecutando:

`Union(R, v, w)`

Recordemos que este tipo de datos utilizaba una estructura de representación mediante punteros al nodo padre. En su versión más eficiente, utilizando equilibrado y compresión de caminos, el tiempo de ejecución de las operaciones era *casi* un $O(1)$.

Análisis del tiempo de ejecución

Hemos dejado muchas cosas sin especificar en el algoritmo de Kruskal, así que el análisis del tiempo de ejecución es un tanto orientativo. En primer lugar, el algoritmo requiere una ordenación previa de las aristas del grafo. En el mejor caso, por ejemplo utilizando QuickSort, podemos ordenar las a aristas en un tiempo $O(a \log a)$.

A continuación, el algoritmo entra en un bucle **mientras** que se repite como mínimo $n - 1$ veces, si no se descarta ninguna arista, y como máximo a , si no se acaba hasta la última arista⁸. Además, todas las operaciones que hay dentro del bucle requieren un tiempo constante. Por lo tanto, el tiempo de ejecución del **mientras** estaría entre $\Omega(n)$ y $O(a)$. Si añadimos el tiempo de la ordenación de las aristas, tenemos que en todos los casos, mejor o peor, el tiempo de ejecución sería un $O(a \log a)$.

El análisis anterior supone que usamos listas de adyacencia. Si utilizamos matrices de adyacencia, el paso de extraer todas las aristas necesitaría un $O(n^2)$ adicional; tendríamos un $O(a \log a + n^2)$. La diferencia podría ser sustancial si el grafo está poco conectado.

Por otro lado, podemos comparar el tiempo con el del algoritmo de Prim, que resuelve el mismo problema. El algoritmo de Prim tarda un $O(n^2)$, por lo que siempre sería más adecuado si trabajamos con matrices de adyacencia. Con listas de adyacencia, la relación entre ambos depende de la proporción entre $a \log a$ y n^2 . Cuantas menos aristas haya, más adecuado será el algoritmo de Kruskal.

5.5. Problemas de caminos mínimos

Una variedad muy importante de problemas sobre grafos son los problemas de caminos mínimos. En un grafo con pesos, el coste de un camino es la suma de los costes de las aristas por las que pasa. En grafos no etiquetados se puede asociar el coste de un camino con su longitud, es decir, el número de aristas por las que pasa.

Encontrar los caminos más cortos en un grafo puede tener distintas aplicaciones.

⁸Cuidado! El algoritmo, tal y como está estructurado, requiere que el grafo G sea conexo. En otro caso, acabaría con todas las aristas pero no saldría del bucle.

Por ejemplo, en un mapa de carreteras, como el de la figura 5.1a), el camino más corto indica la forma de gastar menos gasolina; en una red de ordenadores el camino más corto dice por dónde debe circular la información para que la comunicación sea rápida; en un diagrama de flujos de un algoritmo, como el de la figura 5.3, el camino más corto del inicio al fin significa la ejecución más rápida del programa. Según lo que pretendamos calcular, podemos distinguir los siguientes problemas de caminos mínimos:

- Camino mínimo entre un par de nodos dados, v y w .
- Caminos mínimos entre un nodo origen dado, v , y todos los demás. En grafos dirigidos tendríamos dos problemas equivalentes: desde v a todos los demás nodos, y desde todos los demás nodos hasta v .
- Caminos mínimos entre todos los pares de nodos del grafo.

Curiosamente, si intentamos resolver el caso a) nos encontramos con un problema de la misma complejidad que el caso b). Es decir, la solución para a) consistiría básicamente en calcular los caminos mínimos entre v y todos los demás nodos, y después quedarse con el que va de v a w . Así que vamos a empezar viendo el problema conocido como caminos mínimos desde un origen. Para resolverlo se utiliza el algoritmo de Edsger W. Dijkstra, un algoritmo clásico en teoría de grafos. Para solucionar el caso c) simplemente podríamos repetir el algoritmo de Dijkstra por cada origen. En su lugar vamos a ver otro algoritmo, también dentro de los clásicos, debido a Robert W. Floyd.

En la figura 5.20 se muestra un ejemplo de grafo dirigido con pesos y la resolución del problema de los caminos más cortos desde el nodo 1 hasta todos los demás.

5.5.1. Caminos mínimos empezando por un origen

Supongamos, para simplificar, que el nodo origen es $v = 1$ y que los costes de las aristas están almacenados en una matriz de costes C , de tamaño $n \times n$. La matriz C es como la utilizada en el algoritmo de Prim. En la figura 5.20 se muestra un grafo de ejemplo y los caminos más cortos desde el nodo 1 hasta todos los demás.

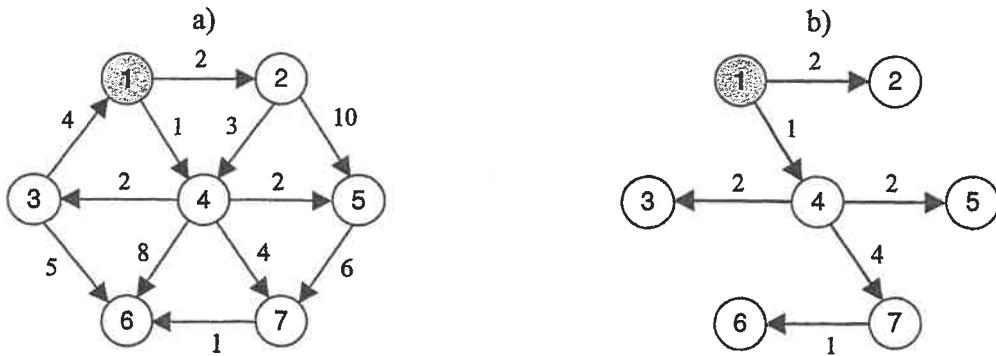


Figura 5.20: Problema de los caminos mínimos empezando por un origen. a) Grafo de entrada del algoritmo. b) Caminos mínimos resultantes.

Para resolver el problema de los caminos más cortos desde un nodo v hasta todos los demás contamos con el **algoritmo de Dijkstra**. El algoritmo garantiza la solución óptima, siempre que los costes de las aristas sean positivos. Vamos a empezar viendo la lógica del algoritmo y después pasaremos a su implementación.

Nodos candidatos, seleccionados y caminos especiales

En el funcionamiento del algoritmo de Dijkstra se considera que el conjunto de vértices V del grafo está descompuesto en dos subconjuntos:

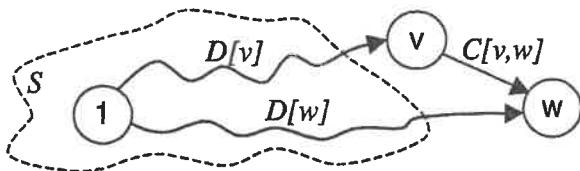
- **Conjunto de nodos escogidos, S .** Son los nodos para los cuales ya se conoce el camino mínimo desde el origen, en cierto momento del algoritmo.
- **Conjunto de nodos candidatos, T .** Está compuesto por todos los demás nodos, para los cuales no se conoce todavía el camino más corto.

En esencia, el algoritmo de Dijkstra consiste en ir sacando nodos del conjunto de candidatos y añadirlos al conjunto de escogidos. Para los nodos de T no sabemos todavía los caminos mínimos, pero lo que sí conocemos es el camino más corto desde el origen pasando por nodos escogidos, es decir los de S . Esto es lo que se conoce como caminos especiales. Un **camino especial** es un camino mínimo desde el origen hasta un nodo cualquiera, pasando sólo por nodos que están en S .

El funcionamiento del algoritmo de Dijkstra es el siguiente:

1. Inicialmente, el conjunto de nodos escogidos $S = \{1\}$, y todos los demás nodos son candidatos. Es fácil calcular los caminos especiales; serán los caminos directos (de longitud 1) puesto que no se puede pasar por otros nodos intermedios. Por lo tanto, el coste del camino especial mínimo a cada w será simplemente $C[1, w]$.
2. Elegir un nodo del conjunto de candidatos T , para añadirlo a S . ¿Qué nodo se elige? Pues aquel cuyo camino especial sea el menor. Para ese nodo, se puede garantizar que el “camino especial mínimo” es, de hecho, el “camino mínimo desde el origen”.
3. Si el nodo seleccionado en el paso anterior es v , entonces al añadirlo a S tenemos que recalcular todos los caminos especiales de los demás nodos de T .
4. El algoritmo acabará cuando todos los nodos hayan sido seleccionados. Cuando se cumpla esto, $S = V$, los caminos especiales pueden pasar por todos los nodos. En consecuencia, los caminos especiales serán los caminos mínimos desde el origen.

Para almacenar los costes de los caminos especiales, el algoritmo de Dijkstra utiliza un array D : **array [2..n] de real**. El punto 1 nos dice que cada posición w se debe inicializar con: $D[w] := C[1, w]$. El punto 2 se traduce en buscar el mínimo valor de D para los nodos que estén en T . Para representar T podemos usar una representación sencilla de conjuntos, por ejemplo con un array de booleanos; $T[w]$ será verdadero o falso si w es candidato o no, respectivamente. Y, finalmente, el punto 3 implica recalcular, para cada nodo w de T el valor $D[w]$, pudiendo pasar el camino por v . Es decir, el camino especial para w será el mínimo entre el que tenía antes ($D[w]$) y el camino mínimo para v ($D[v]$) más el coste de la arista (v, w) ($C[v, w]$).



Así que el paso de actualización al añadir v diría:

```
para cada nodo  $w$  adyacente a  $v$  y siendo Miembro( $T, w$ ) hacer
   $D[w] := \min(D[w], D[v]+C[v,w])$ 
finpara
```

Sólo nos queda un pequeño detalle. Tenemos calculados en D los costes de los caminos mínimos. Pero nos falta conocer cuáles son esos caminos, es decir por qué nodos pasa cada camino mínimo. Para representar los caminos usamos otro array P : array [2..n] de Nodo, tal que $P[v]$ contiene el vértice inmediatamente anterior a v en el camino más corto. Es decir, el camino más corto para un v cualquiera sería:

$$1 \rightarrow \dots P[P[P[v]]] \rightarrow P[P[v]] \rightarrow P[v] \rightarrow v$$

Inicialmente $P[v] = 1$ para todo $v \neq 1$, es decir, los caminos mínimos son los caminos directos. Por otro lado, en el paso de actualización, si el camino más corto para w pasa por v , entonces debemos indicarlo en $P[w]$.

```
para cada nodo  $w$  adyacente a  $v$  hacer
  si Miembro( $T, w$ ) Y  $(D[v]+C[v,w]) < D[w]$  entonces
     $D[w] := D[v]+C[v,w]$ 
     $P[w] := v$ 
  finsi
finpara
```

Algoritmo de Dijkstra

Recopilando todo lo que hemos estudiado hasta ahora del algoritmo de Dijkstra, vamos a ver una posible implementación en pseudocódigo. Damos por supuesto el tipo Conjunto[Rango], implementado mediante arrays de booleanos.

```
operación Dijkstra (C: array [1..n, 1..n] de real; var D: array [2..n] de real; var P: array [2..n] de entero)
var T: Conjunto[2..n]
para i:= 2, ..., n hacer
  D[i]:= C[1, i]
  P[i]:= 1
  Inserta(T, i)
finpara
para i:= 2, ..., n hacer
  v:= vértice con Miembro( $T, v$ )=verdadero y  $D[v]$  mínimo
  Suprime( $T, v$ )
  para cada nodo  $w$  adyacente a  $v$  hacer
    si Miembro( $T, w$ ) Y  $(D[v]+C[v,w]) < D[w]$  entonces
```

```

 $D[w] := D[v] + C[v,w]$ 
 $P[w] := v$ 
finsi
finpara
finpara

```

Para simplificar la estructura, no se desarrolla la operación “ $v :=$ vértice con Miembro(T, v)=**verdadero** y $D[v]$ mínimo”, que consistiría en un simple recorrido secuencial de un array. En la tabla 5.4 se muestra un ejemplo de ejecución del algoritmo de Dijkstra, para el grafo de la figura 5.20.

Iteración	T	$D[2]$	$D[3]$	$D[4]$	$D[5]$	$D[6]$	$D[7]$	$P[2]$	$P[3]$	$P[4]$	$P[5]$	$P[6]$	$P[7]$	v
inicial	{2,3,4,5,6,7}	2	∞	1	∞	∞	∞	1	1	1	1	1	1	-
2	{2,3,5,6,7}	2	3	1	3	9	5	1	4	1	4	4	4	4
3	{3,5,6,7}	2	3	1	3	9	5	1	1	1	1	1	1	2
4	{5,6,7}	2	3	1	3	8	5	1	4	1	4	3	4	3
5	{6,7}	2	3	1	3	8	5	1	4	1	4	3	4	5
6	{6}	2	3	1	3	6	5	1	4	1	4	7	4	7
7	{ }	2	3	1	3	6	5	1	4	1	4	7	4	6

Tabla 5.4: Aplicación del algoritmo de Dijkstra sobre el grafo de la figura 5.20. Aparecen en negrita los valores que se modifican en cada paso.

Demostración de la optimalidad del algoritmo

Es posible demostrar formalmente que el algoritmo de Dijkstra es óptimo –es decir encuentra siempre los caminos mínimos– suponiendo que las aristas tienen peso positivo. Para ver la demostración, vamos a diferenciar primero entre los distintos conceptos que entran en juego:

1. Camino mínimo en el grafo: es lo que buscamos, lo que podríamos llamar el mínimo “absoluto”. Denotamos el coste del camino mínimo por $\text{CosteMin}(v \rightsquigarrow w)$.
2. Camino especial: es un camino mínimo pero restringido a pasar sólo por nodos escogidos, de S . Lo denotamos por $\text{CosteEsp}(1 \rightsquigarrow w)$, siendo 1 el nodo origen.
3. Array D : es el coste del camino que calcula el algoritmo de Dijkstra, para cada nodo distinto de 1.

Por un lado, está claro que cuando $S = V$ entonces $\text{CosteEsp} \equiv \text{CosteMin}$. Así que si demostramos que $D \equiv \text{CosteEsp}$, tendremos demostrado que el algoritmo de Dijkstra

es óptimo. La demostración tiene dos partes: 1) $\text{CosteEsp} \equiv \text{CosteMin}$ para los nodos que están en S ; 2) $D \equiv \text{CosteEsp}$ para todos los nodos antes de añadirlos a S .

1) El menor camino especial hasta un nodo de T es un camino mínimo. Esto implica, de forma directa, que para los nodos añadidos a S los caminos especiales son caminos mínimos. La demostración de este punto garantiza que el orden de selección de los nodos de T debe ser, como ocurre en el algoritmo, por el menor valor de D .

Sea v el nodo de T con menor camino especial. Pero supongamos, por reducción al absurdo, que ese camino especial no es un camino mínimo, es decir $\text{CosteEsp}(1 \rightsquigarrow v) > \text{CosteMin}(1 \rightsquigarrow v)$. Al no ser especial, su camino mínimo debe pasar por algún nodo que no esté en S , como se muestra en la figura 5.21a). Sea k el primero de estos nodos fuera de S .

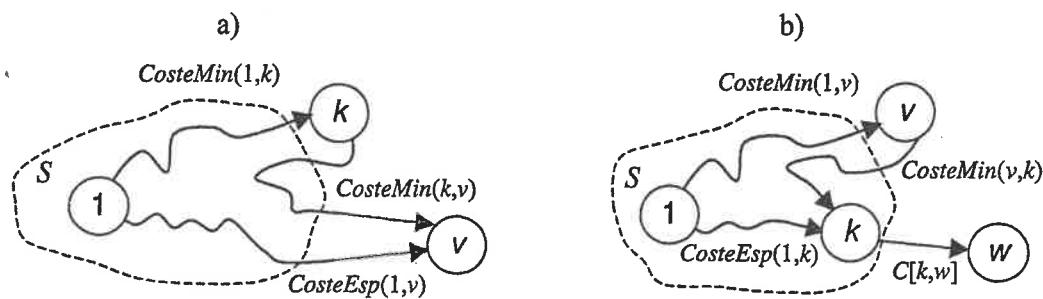


Figura 5.21: Demostración del algoritmo de Dijkstra. a) El menor camino especial es un camino mínimo. b) El camino especial es el que calcula el algoritmo de Dijkstra.

El coste de este hipotético camino mínimo sería: $\text{CosteMin}(1 \rightsquigarrow v) = \text{CosteMin}(1 \rightsquigarrow k) + \text{CosteMin}(k \rightsquigarrow v)$. Pero como los pesos son positivos: $\text{CosteMin}(k \rightsquigarrow v) > 0$; luego: $\text{CosteMin}(1 \rightsquigarrow v) > \text{CosteMin}(1 \rightsquigarrow k)$. Ahora bien, como hemos tomado en k el primero de los nodos fuera de S , entonces su camino mínimo es un camino especial y tenemos que $\text{CosteEsp}(1 \rightsquigarrow v) > \text{CosteEsp}(1 \rightsquigarrow k)$. Llegamos a una contradicción: el menor camino especial no es el de v (como habíamos supuesto al principio) sino el de k . Por lo tanto, el menor camino especial debe ser el camino mínimo de ese nodo.

2) $D[w]$ almacena efectivamente el coste del camino especial a w . La demostración de este punto garantiza que el proceso de actualización del algoritmo de Dijkstra, al añadir un nodo v a S , es correcto. La demostración se puede hacer por inducción.

Inicialmente, cuando $S = \{1\}$ está claro que $D[w]$ contiene los caminos especiales. Supongamos, por hipótesis de inducción, que en cierto momento D almacena los caminos especiales. Veamos que después de añadir un nuevo nodo v a S , el array D sigue guardando los caminos especiales.

Sea w un nodo de T cualquiera. El algoritmo de Dijkstra toma el mínimo entre $D[w]$ y $D[v] + C[v, w]$. Supongamos, por reducción al absurdo, que el camino especial para w pasa por v pero no es el camino: $1 \rightsquigarrow v \rightarrow w$, sino que después de v pasa por otros nodos de S , siendo del tipo: $1 \rightsquigarrow v \rightsquigarrow k \rightarrow w$. En la figura 5.21b) se ilustra esta hipotética situación. El coste de este camino sería: $\text{CosteEsp}(1 \rightsquigarrow v) + \text{CosteMin}(v \rightsquigarrow k) + C[k, w]$.

Ahora bien, como k estaba en S antes que v , entonces el camino especial para k ya era un camino mínimo, así que el coste de $1 \rightsquigarrow v \rightsquigarrow k$ es mayor que $\text{CosteEsp}(1 \rightsquigarrow k)$.

Usando la hipótesis de inducción, $D[k] = \text{CosteEsp}(1 \rightsquigarrow k) < \text{CosteEsp}(1 \rightsquigarrow v) = D[v]$, luego $D[k] + C[k, w] < D[v] + C[v, w]$. Por lo tanto, tenemos una contradicción: el camino especial para w pasaría solo por k y no por v . En definitiva, la actualización del algoritmo de Dijkstra garantiza que D almacena los caminos especiales.

Análisis del tiempo de ejecución

Igual que con la búsqueda en profundidad, vamos a realizar el estudio del trabajo total realizado por el algoritmo. En primer lugar tenemos una inicialización de los arrays D y P , que requiere un $O(n)$, para un grafo de n nodos. A continuación tenemos un **para** i , dentro un **para cada** adyacente y dentro una comparación y algunas operaciones constantes. Razonando como en la búsqueda en profundidad, vemos que el resultado final es que se recorren todas las aristas del grafo, comprobando todas las adyacencias. Por lo tanto, con matrices de adyacencia tenemos $O(n^2)$ y con listas $O(n + a)$.

Pero no podemos olvidar la operación “ $v :=$ vértice con $\text{Miembro}(T, v) = \text{verdadero} \dots$ ”, que implica un recorrido secuencial del array D para cada uno de los n nodos. Por lo tanto, tendríamos que sumar un $O(n^2)$. Con matrices de adyacencia tenemos en total un $O(n^2)$. Con listas de adyacencia: $O(n + a + n^2) = O(n^2)$, ya que siempre $a \leq n^2$.

Se podría mejorar el tiempo con listas de adyacencia, si la operación anterior no necesitara un recorrido secuencial del array para localizar el mínimo. Por ejemplo, si los nodos se almacenan en una estructura de **montículos de mínimos** se podría encontrar en un $O(1)$ y luego suprimirlo en $O(\log n)$. También podríamos usar un árbol AVL, con un $O(\log n)$ para ambas operaciones. No obstante, esta estructura ordenada podría modificarse cada vez que se actualice D , dentro del **si**. La actualización requiere un $O(\log n)$ en ambos casos, y se puede repetir como máximo $a+n$ veces. Por lo tanto, el orden de complejidad del algoritmo con esta implementación sería $O(n+a+(a+n)\log n) = O((a+n)\log n)$. El tiempo será menor que $O(n^2)$ cuando $a \ll n^2$, es decir, cuando el grafo sea disperso.

5.5.2. Caminos más cortos entre todos los vértices

En cierto sentido, el problema de los caminos mínimos entre todos los pares de vértices abarca –como situación más general– al problema de los caminos mínimos desde un origen. Como ya propusimos al principio de la sección, para resolverlo se podría aplicar el algoritmo de Dijkstra repetidas veces, eligiendo cada vez como origen un vértice diferente. El orden de complejidad de esta solución sería $O(n^3)$ si usamos matrices de adyacencia y $O(n(a+n)\log n)$ con listas de adyacencia, siendo n el número de nodos y a el número de aristas.

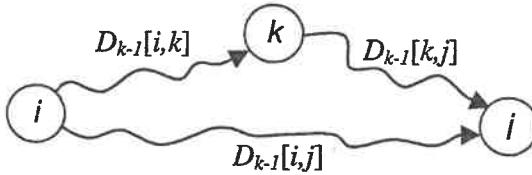
Pero existe una solución más directa y elegante, que es conocida como el **algoritmo de Floyd**. Igual que en el algoritmo de Dijkstra, trabajamos con una matriz de costes C de tamaño $n \times n$, que tendrá un $+\infty$ en las posiciones donde no exista una arista. También como en el algoritmo de Dijkstra, los costes deben ser siempre positivos. El resultado final del algoritmo vendrá dado en otra matriz D , de tamaño $n \times n$, en la que cada posición $D[v, w]$ indica el coste del camino mínimo entre v y w . Si, al final del algoritmo, tenemos que $D[v, w] = +\infty$ entonces quiere decir que no existe ningún camino entre v y w . Obviamente, en un grafo no dirigido siempre será $D[v, w] = D[w, v]$.

Pivotaje sobre un nodo y programación dinámica

El algoritmo de Floyd da lugar a la idea de la programación dinámica, que es una técnica general de diseño de algoritmos. En esencia, la idea del algoritmo de Floyd es la siguiente. Inicialmente, en la matriz D tenemos los costes de las aristas entre todos los nodos, es decir $D = C$. El algoritmo está compuesto por n pasos, iterando con una variable k , desde 1 hasta n . En cada paso k , la matriz D contendrá los costes de los caminos mínimos entre todos los pares de nodos, pudiendo pasar por los k primeros nodos, como nodos intermedios de los caminos. Es decir, cuando $k = 1$ tenemos que calcular los caminos mínimos entre todos los v y w pasando, o no, por el nodo 1. Cuando $k = 2$ calculamos todos los caminos mínimos pudiendo pasar por los nodos 1 y 2. Y así sucesivamente hasta tener $k = n$. Al final tendremos en D los caminos más cortos pudiendo pasar por todos los vértices, es decir lo que buscamos, los caminos mínimos *absolutos*.

En cada paso k del algoritmo, decimos que el vértice k actúa como **pivote**. El proceso de pivotaje consiste en comprobar, para todos los pares de vértices i y j , si su camino mínimo debe pasar por k o no. Es decir, en el caso k se debe elegir para todos los pares i, j una posibilidad de entre:

- No pasar por el vértice k . En ese caso, el coste del camino no varía. Seguiría siendo⁹: $D_{k-1}[i, j]$.
- Pasar por el vértice k . En este caso, el coste sería ir de i a k y luego ir de k a j . Por lo tanto, el coste ahora sería: $D_{k-1}[i, k] + D_{k-1}[k, j]$.



¿Con cuál de las dos opciones nos quedamos? Pues, obviamente, con la que tenga menor coste: $D_k[i, j] := \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$. Así que la matriz D en el paso k se calcula usando la misma D en el paso $k - 1$. Es más, se puede demostrar que en el paso k ni la fila ni la columna k de la matriz se modifican¹⁰. Por lo tanto, podemos usar la misma matriz D para todos los pasos del algoritmo, sin necesidad de distinguir D_{k-1} de D_k . De esta forma, el núcleo del algoritmo de Floyd consistiría en lo siguiente:

```

para  $k := 1, \dots, n$  hacer
  para  $i := 1, \dots, n$  hacer
    para  $j := 1, \dots, n$  hacer
       $D[i, j] := \min(D[i, j], D[i, k] + D[k, j])$ 
    finpara
  finpara
finpara

```

⁹El término D_{k-1} significa: “la matriz matriz D en el paso $k - 1$ ”. Y, según lo que hemos visto, D_x almacena los caminos mínimos entre todos los pares pudiendo pasar por los x primeros nodos.

¹⁰Por ejemplo, la fila k sería $D_k[k, j] := \min(D_{k-1}[k, j], D_{k-1}[k, k] + D_{k-1}[k, j])$. Como hemos supuesto que todos los pesos son positivos, entonces $D_{k-1}[k, j] \leq D_{k-1}[k, k] + D_{k-1}[k, j]$. De esta forma, $D_k[k, j] := D_{k-1}[k, j]$, la fila k no cambia en el paso k .

finpara

Al final del algoritmo tenemos almacenados en D los costes de los caminos mínimos entre todos los pares de nodos. Pero ¿cuáles son esos caminos? Igual que en el algoritmo de Dijkstra, necesitamos una estructura para almacenar los nodos por los que pasan los caminos. Esta estructura será una matriz P , de tamaño $n \times n$. En cada posición $P[i, j]$ tendremos un nodo intermedio por el cual pasa el camino mínimo de i a j . En otras palabras, el camino mínimo de i a j sería:

$$i \rightarrow \dots \rightarrow P[i, j] \rightarrow \dots \rightarrow j$$

Si el camino mínimo es directo, entonces $P[i, j] = 0$. Inicialmente, todos los caminos de D serán caminos directos y $P[i, j] = 0$, para todo i y j . Por otro lado, si en un paso de pivotaje encontramos que el mínimo es $D[i, k] + D[k, j]$, entonces deberíamos indicar que el camino mínimo entre i y j pasa por k : $P[i, j] := k$.

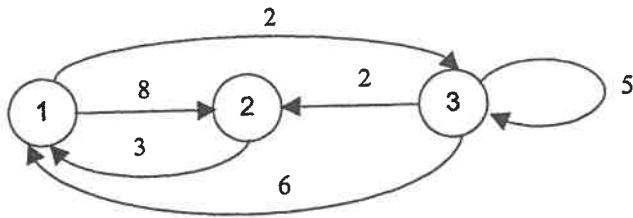
Algoritmo de Floyd

Juntando todo lo anterior, una posible implementación del algoritmo de Floyd sería la siguiente.

```
operación Floyd (C: array [1..n, 1..n] de real; var D: array [1..n, 1..n] de real; var P: array [1..n, 1..n] de entero)
  para i:= 1, ..., n hacer
    para j:= 1, ..., n hacer
      D[i, j]:= C[i, j]
      P[i, j]:= 0
    finpara
    D[i, i]:= 0
  finpara
  para k:= 1, ..., n hacer
    para i:= 1, ..., n hacer
      para j:= 1, ..., n hacer
        si D[i, j] < D[i, k]+D[k, j] entonces
          D[i, j]:= D[i, k]+D[k, j]
          P[i, j]:= k
        finsi
      finpara
    finpara
  finpara
```

La asignación: $D[i, i] := 0$, es requerida en muchas aplicaciones, en las cuales la distancia de un vértice a sí mismo siempre debe ser tomada como cero; y esto aun cuando existan aristas del tipo (i, i) , como en el grafo de la figura 5.22. Sin embargo, la asignación no es imprescindible para que el algoritmo funcione correctamente. ¿Qué pasaría si no la hacemos? En ese caso, el algoritmo calcularía el coste del camino mínimo de i a i pasando por algún otro nodo. Si el grafo es dirigido, sería el ciclo de menor coste que pasa por i .

En la figura 5.23 se muestra la aplicación del algoritmo de Floyd sobre el grafo de la figura 5.22. Se muestra graficamente la idea del pivotaje.

Figura 5.22: Ejemplo de grafo dirigido, donde se permiten aristas del tipo $\langle v, v \rangle$.

	1	2	3	
1	0	8	2	
2	3	0	∞	
3	6	2	0	

$D_0[i,j]$

	1	2	3	
1	0	8	2	
2	3	0	5	
3	6	2	0	

$D_1[i,j]$

	1	2	3	
1	0	8	2	
2	3	0	5	
3	5	2	0	

$D_2[i,j]$

	1	2	3	
1	0	4	2	
2	3	0	5	
3	5	2	0	

$D_3[i,j]$

Figura 5.23: Aplicación del algoritmo de Floyd sobre el grafo de la figura 5.22. En el paso k ni la fila ni la columna k se modifican.

Por otro lado, ¿cómo podemos recuperar los nodos por los que pasa el camino mínimo entre cada i y j ? Sabemos que el camino mínimo pasa por $P[i, j]$. Razonando de manera inductiva, tendríamos que encontrar los trozos de camino de i a $P[i, j]$, y luego de $P[i, j]$ a j . Y así mientras $P[i, j]$ sea distinto de 0. El algoritmo podría ser el siguiente.

operación Camino (P: array [1..n, 1..n] de entero; i, j: entero)

```

k := P[i, j]
si k ≠ 0 entonces
    Camino(P, i, k)
    escribir(k)
    Camino(P, k, j)
finsi

```

En el anterior algoritmo, faltaría por escribir los propios vértices i y j .

```

escribir(i)
Camino(i, j)
escribir(j)

```

Análisis del tiempo de ejecución

El análisis del tiempo de ejecución del algoritmo de Floyd resulta bastante sencillo. Trivialmente, la inicialización requiere $2n^2 + n$ asignaciones, y el proceso de pivotaje n^3 comparaciones, que en caso de ser ciertas añadirían $2n^3$ asignaciones. En total, el orden de complejidad del algoritmo es un $O(n^3)$.

Comparado con el algoritmo de Dijkstra repetido n veces, tenemos los mismos tiempos si usando matrices de adyacencia. Las n ejecuciones del algoritmo de Dijkstra darían un $O(n^3)$. Sin embargo, el algoritmo de Floyd es más directo y sencillo de programar, por lo que será más adecuado si necesitamos los caminos entre todos los pares.

Por otro lado, con listas de adyacencia el algoritmo de Dijkstra repetido n veces tardaría un $O((a+n)n \log n)$. Si suponemos que el grafo es conexo, tendríamos $O(a n \log n)$. Comparado con el $O(n^3)$ del algoritmo de Floyd, tardaría menos tiempo conforme a fuera menor. Es decir, tendríamos una mejora si el grafo está poco conectado.

Cierre transitivo

Un problema muy relacionado con el de los caminos mínimos es el problema del cierre transitivo. Dado un grafo cualquiera, queremos calcular si existe un camino entre dos vértices o no, para todos los vértices del grafo. Por ejemplo, si aplicamos el algoritmo de Floyd sobre un grafo con pesos, entonces si $D[i, j] = +\infty$ al final del algoritmo, no existe ningún camino entre i y j . En otro caso sí existe algún camino.

El cierre transitivo se puede aplicar sobre grafos sin pesos, utilizando la matriz de adyacencia M del grafo. El resultado será una matriz A de conectividad. Es decir, $M[i, j]$ indica si existe una arista (i, j) y $A[i, j]$ indica si existe un camino de i a j .

El problema se puede resolver con el algoritmo de Warshall. El algoritmo es muy parecido al de Floyd, pero en lugar de razonar con costes lo hacemos con booleanos. En cada paso $k - 1$, indicamos en $A[i, j]$ si existe camino entre i y j pudiendo pasar por los $k - 1$ primeros nodos. Entonces, existirá camino en el paso k si existía en el paso $k - 1$ o bien si existe un camino de i a k y otro de k a j . Escrito como una expresión booleana: $A[i, j] := A[i, j] \text{ O } (A[i, k] \text{ Y } A[k, j])$. El algoritmo sería sencillo.

operación Warshall (M : array [1..n, 1..n] de booleano; var A : array [1..n, 1..n] de booleano)

```

A := M      // Inicialización
para k := 1, ..., n hacer
    para i := 1, ..., n hacer
        para j := 1, ..., n hacer
            A[i, j] := A[i, j] O (A[i, k] Y A[k, j])
        finpara
    finpara
finpara

```

Claramente, el tiempo de ejecución del algoritmo de Warshall es un $O(n^3)$. Aunque el algoritmo se puede ver como una sencilla modificación del algoritmo de Floyd, el mérito de Warshall es que su algoritmo es previo al de Floyd.

El centro de un grafo

Vamos a ver un ejemplo de aplicación del problema de los caminos mínimos entre todos los pares de nodos. Podemos definir el concepto de **excentricidad de un nodo** como la mayor de las distancias mínimas de los caminos que llegan a ese nodo. El **centro de un grafo** con pesos se define como el vértice con menor valor de excentricidad. Es decir, es el vértice que tiene más cercano su vértice más distante.

Supongamos, por ejemplo, el grafo de carreteras mostrado a la izquierda en la figura 5.24. La cuestión es ¿cómo calcular el centro del grafo?



<i>D</i>	1	2	3	4	5	6	7	8	Exc.
1 Barcelona	0	620	956	621	590	349	814	296	956
2 Bilbao	620	0	730	395	796	969	280	324	969
3 Jaén	956	730	0	335	368	609	528	660	956
4 Madrid	621	395	335	0	401	642	193	325	642
5 Murcia	590	796	368	401	0	241	594	726	796
6 Valencia	349	969	609	642	241	0	835	645	969
7 Valladolid	814	280	528	193	594	835	0	518	835
8 Zaragoza	296	324	660	325	726	645	518	0	726

Figura 5.24: El centro de un grafo. Izquierda: grafo de carreteras. Derecha: caminos mínimos entre ciudades y excentricidad de cada ciudad. El centro del grafo es Madrid.

El problema se puede resolver fácilmente utilizando los algoritmos que hemos visto. En concreto, deberíamos hacer lo siguiente:

- 1 Calcular la distancia mínima entre cada par de nodos, utilizando el algoritmo de Floyd (o Dijkstra repetido n veces). Obtenemos una matriz de costes mínimos D .
- 2 Calcular las excentricidades: $\text{excentricidad}(v) = \max\{D[w, v] \mid \forall w \in V\}$. Es decir, tomamos los máximos por columnas (o por fila, si el grafo es no dirigido).
- 3 Devolver el nodo v con menor valor de $\text{excentricidad}(v)$.

En la derecha de la figura 5.24 se muestra el resultado del algoritmo para el grafo de la izquierda. El centro del grafo sería el nodo Madrid.

¿Cuál es el orden de complejidad del algoritmo? Si utilizamos el algoritmo de Floyd, el punto 1 tarda un $O(n^3)$, el 2 necesita un $O(n^2)$ y el 3 un $O(n)$. En total, el tiempo sería un $O(n^3)$.

5.6. Algoritmos sobre grafos dirigidos

En esta sección vamos a estudiar tres problemas relacionados con los grafos dirigidos: el cálculo de los componentes fuertemente conexos, el recorrido en orden topológico sobre grafos dirigidos acíclicos y los problemas de flujo en redes.

5.6.1. Componentes fuertemente conexos

Como estudiamos en la introducción, un grafo $G = (V, A)$ se dice que es conexo, o que está conectado, si existen caminos entre v y w , para todo $v, w \in V$. En el caso de los grafos dirigidos añadimos la etiqueta “fuertemente”. Definimos también un componente conexo –o **componente fuertemente conexo**, en grafos dirigidos– como un subgrafo conexo y maximal de G . Obviamente, si un grafo es conexo, entonces sólo tendrá un componente conexo, que será él mismo. Por otro lado, si el grafo no tiene ninguna arista, entonces cada nodo será un componente conexo por sí mismo.

La búsqueda de los componentes conexos, o fuertemente conexos, en un grafo puede tener numerosas utilidades. Por ejemplo, en un grafo como el de la figura 5.25, los nodos representan páginas web y las aristas los enlaces de una páginas con otras. Los componentes fuertemente conexos son todas las páginas que se refieren entre sí, bien directamente o a través de otras.

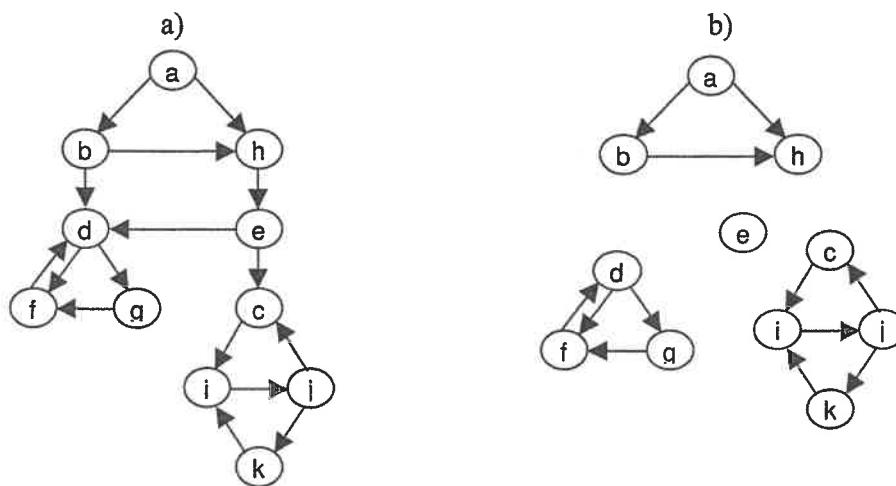


Figura 5.25: El problema de los componentes fuertemente conexos. a) Grafo de páginas web y referencias entre las mismas. b) Componentes fuertemente conexos resultantes.

Componentes conexos en grafos no dirigidos

En grafos no dirigidos, suponiendo que el grafo representa vías de comunicación, los distintos componentes conexos indican nodos entre los cuales es imposible la comunicación. El cálculo de los componentes conexos en grafos no dirigidos es bastante sencillo, así que lo resolveremos como un ejemplo.

Ejemplo 5.4 Dar un algoritmo para encontrar los componentes conexos de un grafo no dirigido, como el mostrado en la figura 5.26.

Suponiendo que arrancamos una búsqueda primero en profundidad, o en anchura, desde un nodo de un componente cualquiera, está claro que se visitarán todos los nodos de ese componente y sólo esos nodos. Por ejemplo, si arrancamos la búsqueda en $v = 1$,

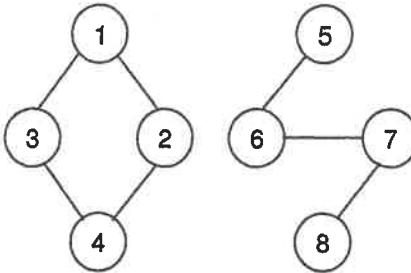


Figura 5.26: Grafo no dirigido. Encontrar los componentes conexos.

se visitarán los nodos 1, 2, 3, 4. Por lo tanto, cada árbol de expansión en profundidad corresponderá a un componente conexo.

La implementación se basaría en una búsqueda en profundidad, con algunas modificaciones. En concreto, el array *marca* será un array de enteros, inicializado a 0, donde almacenaremos el número de componente al que pertenece cada nodo. Cada vez que visitamos un nodo, ponemos el componente que le corresponde. Suponemos que el entero *numeroComponentes* es una variable global. La implementación sería la siguiente:

operación CalculaComponentesConexos

```

BorraMarcas
numeroComponentes:= 0
para v:= 1, ..., n hacer
  si marca[v] = 0 entonces
    numeroComponentes:= numeroComponentes + 1
    marca[v]:= numeroComponentes
    bpp(v)
  finsi
finpara
  
```

operación bpp (v: Nodo)

```

para cada nodo w adyacente a v hacer
  si marca[w] = 0 entonces
    marca[w]:= numeroComponentes
    bpp(w)
  finsi
finpara
  
```

Cálculo de componentes en grafos dirigidos

¿Podemos aplicar para los grafos dirigidos la misma idea que para los no dirigidos? Es decir, ¿bastaría con hacer una búsqueda primero en profundidad para obtener los componentes fuertemente conexos del grafo? Pues, desafortunadamente no. Se puede comprobar con un contraejemplo, como el grafo de la figura 5.27.

Claramente, en el grafo de la figura 5.27 los componentes fuertemente conexos son cada uno de los nodos por separado. Existe un camino, por ejemplo, de 1 a 4, y no de 4

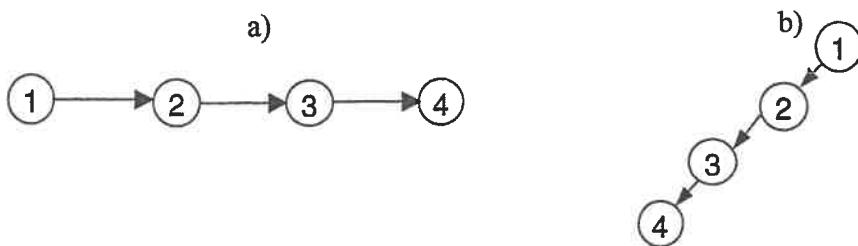


Figura 5.27: Componentes fuertemente conexos. a) Grafo dirigido. b) La búsqueda en profundidad genera solo un árbol, aunque hay cuatro componentes fuertemente conexos.

a 1. Pero la búsqueda en profundidad genera un solo árbol para todo el grafo.

No obstante, razonando sobre este mismo ejemplo, si la búsqueda primero en profundidad hubiera o hubiese empezado al revés (primero por el 4, luego el 3, etc.) sí que habríamos obtenido un árbol de expansión por cada componente fuertemente conexo, que es precisamente lo que buscamos. La conclusión es que tenemos que hacer una búsqueda en profundidad, pero no en un orden cualquiera. ¿En qué orden?

Pues en un orden que garantice que si desde v se visita w , también desde w se puede llegar a v . Para conseguirlo, nos basamos en dos búsquedas en profundidad: en la primera se numeran los nodos, y en la segunda se utiliza esa numeración para hacer el segundo recorrido. Además, la segunda búsqueda se hace invirtiendo el sentido de las aristas, de manera que si una búsqueda garantiza los caminos de v a w , la otra comprueba también los de w a v .

En definitiva, el algoritmo para calcular los componentes fuertemente conexos de un grafo dirigido, $G = (V, A)$, tendría la siguiente estructura:

1. Realizar una búsqueda primero en profundidad de G , numerando los vértices en el orden de terminación de las llamadas recursivas, es decir, justo antes de finalizar la llamada a bpp. Esto es lo que se conoce normalmente como **numeración en orden posterior**.
2. Construir un nuevo grafo dirigido, G_R , invirtiendo las direcciones de los arcos. Es decir, para todo $\langle v, w \rangle \in A(G)$, tenemos que $\langle v, w \rangle \in A(G_R)$, y $V(G_R) = V(G)$.
3. Realizar una búsqueda primero en profundidad de G_R , partiendo del nodo con numeración más alta. Si en el recorrido no se visitan todos los nodos, continuar la búsqueda en profundidad a partir del nodo no visitado con numeración más alta.
4. Cada árbol del bosque de expansión resultante del punto 3, es un componente fuertemente conexo de G .

La implementación sería sencilla, sin más que realizar algunas modificaciones sobre el procedimiento bpp. En la práctica, en el paso 2 no es necesario construir otro árbol, sino que bastaría con cambiar: **para cada** nodo w adyacente a v **hacer** ..., por: **para cada** nodo w adyacente de v **hacer** ... En la figura 5.28 se muestra un ejemplo de aplicación de este algoritmo.

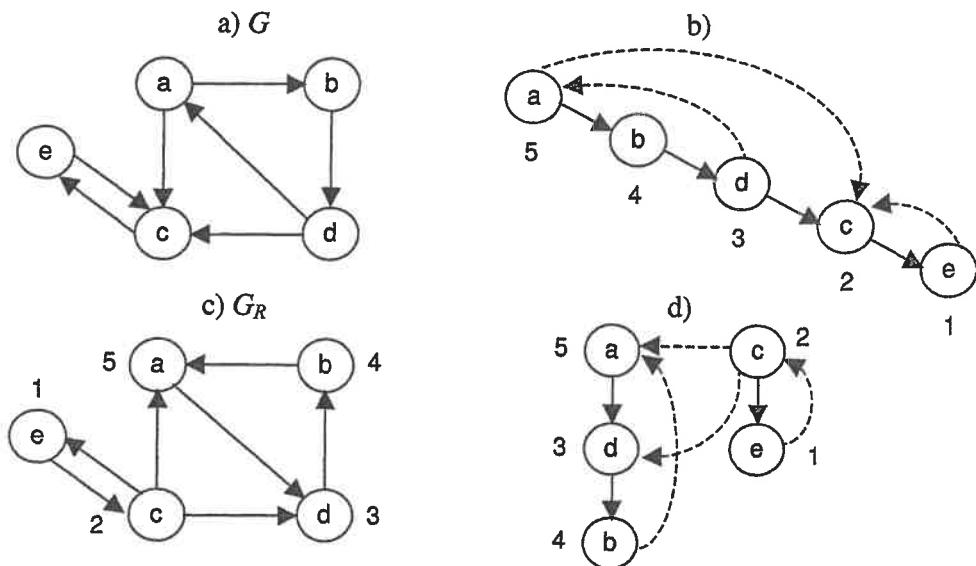


Figura 5.28: Obtención de los componentes fuertemente conexos. a) Grafo del problema. b) Búsqueda en profundidad del paso 1, con numeración en orden posterior. c) Grafo invertido, G_R . d) Búsqueda en profundidad de G_R , del paso 3.

Los componentes fuertemente conexos del grafo de la figura 5.28a) serían: $\{a,b,d\}$ y $\{c,e\}$. Por otro lado, el orden de complejidad del algoritmo viene dado por el orden del procedimiento bpp. Con matrices de adyacencia tendríamos un $O(n^2)$ y con listas un $O(n + a)$.

Grafo reducido de un grafo dirigido

Se puede observar en el ejemplo de la figura 5.28, que todo vértice del grafo G pertenece a algún componente fuertemente conexo. Pero no ocurre lo mismo con las aristas. Estas aristas –que conectan dos vértices que pertenecen a distintos componentes– se llaman **arcos de cruce de componentes**. Utilizando estos arcos de cruce de componentes es posible obtener una representación simplificada de un grafo dirigido, conocida como **grafo reducido**.

El **grafo reducido** asociado a un grafo dirigido G , es un grafo dirigido en el que cada nodo representa un componente fuertemente conexo de G . Además, existirá una arista $\langle v,w \rangle$ en el grafo reducido, si existe una arista entre algunos de los nodos de los componentes conexos de G correspondientes a v y a w . En la figura 5.29 se muestran dos ejemplos de grafos reducidos.

Un grafo reducido sirve para expresar, de manera simplificada, todos los caminos existentes en el grafo. Por ejemplo, el grafo reducido de la figura 5.29b) nos dice que existen caminos de todos los nodos de $\{a, b, h\}$ entre sí, de todos los nodos de $\{d, f, g\}$ entre sí, y también hay caminos desde los $\{a, b, h\}$ hasta los $\{d, f, g\}$, pero no al revés.

Ejemplo 5.5 Comprobar que dentro de un grafo reducido no pueden existir ciclos.

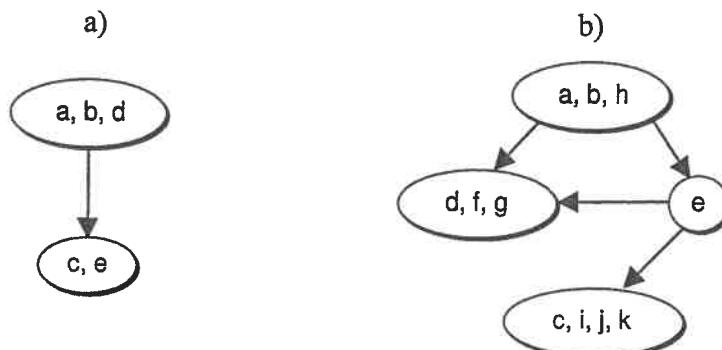


Figura 5.29: Grafos reducidos asociados a los grafos dirigidos de las figuras: a) figura 5.28, b) figura 5.25.

Supongamos, por reducción al absurdo, que existe algún ciclo dentro de un grafo reducido. Sean los nodos v y w parte de este ciclo. Esto significa que existe un camino de v a w y otro camino de w a v , siguiendo el ciclo. Por lo tanto, todos los nodos de v y los de w están fuertemente conectados, y deberían estar en el mismo componente fuertemente conexo. Como el grafo reducido representa cada componente fuertemente conexo con un nodo, no podrían aparecer dos nodos v y w distintos.

5.6.2. Grafos dirigidos acíclicos

Un grafo dirigido acíclico (GDA) no es más que un grafo dirigido y sin ciclos. En muchas aplicaciones que usan grafos dirigidos, las características del problema obligan –de forma implícita o explícita– a que no existan ciclos en el grafo. Por ejemplo, acabamos de ver que el grafo reducido de un grafo dirigido es siempre un grafo acíclico. El interés de los GDA es que existen ciertos conceptos y problemas que sólo tienen sentido sobre este tipo de grafos, como la ordenación topológica.

En la figura 5.30 se muestran dos ejemplos de aplicaciones donde aparecen GDA.

- **Expresiones aritméticas con subexpresiones comunes.** Como vimos en el capítulo 4, un árbol puede utilizarse para representar expresiones aritméticas. Pero si aparecen subexpresiones repetidas dentro del árbol, entonces podemos usar un grafo para evitar duplicaciones. Por ejemplo, en la figura 5.30, la expresión $(a + b)$ aparece tres veces pero sólo se representa una. Si existiera un ciclo, sería imposible evaluar el resultado final de la expresión, habría recursividad infinita.
- **Planificación de tareas.** En los grandes sistemas de gestión empresarial, el desarrollo de un gran proyecto se divide en muchas tareas pequeñas. Existen precedencias entre las tareas: si a precede a b , entonces b no puede empezar hasta que haya acabado a . Los nodos representan tareas –cada una de las cuales tiene un tiempo estimado de ejecución– y cada arista $\langle a, b \rangle$ significa que a precede a b . Por ejemplo, en la figura 5.30b) la tarea “Pintar pirámide” tarda 3 meses, pero sólo puede empezar

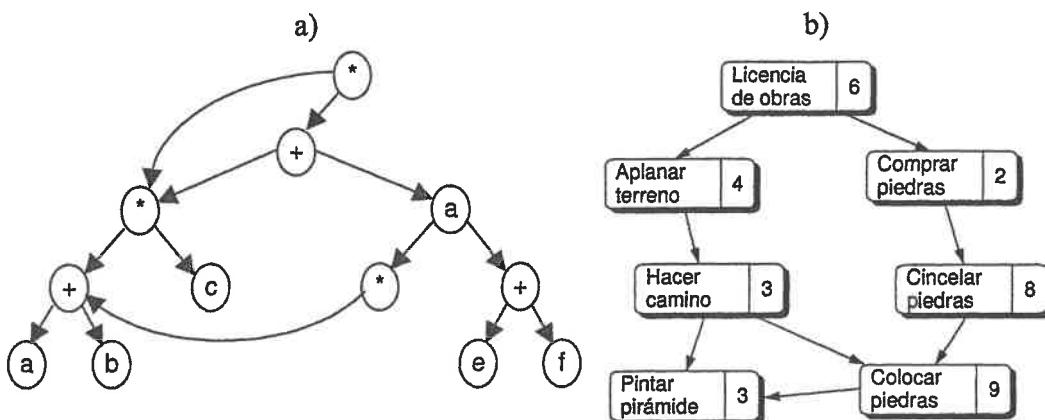


Figura 5.30: Dos aplicaciones que usan GDA. a) Representación de expresiones aritméticas con subexpresiones comunes, para la expresión: $((a+b)c + ((a+b)e)(e+f))((a+b)c)$. b) Grafo de planificación de tareas. Tareas necesarias para construir una pirámide. A la derecha se indican los meses previstos para cada tarea.

después de “Hacer camino” y “Colocar piedras”. Si existiera un ciclo sería imposible ejecutar el proyecto.

- **Prerrequisitos de un plan de estudios.** En este caso, cada nodo representa una asignatura y una arista indica que para hacer cierta asignatura antes hay que acabar otra. Esta aplicación es parecida a la anterior. ¿Qué pasaría si para matricularse en AED antes hay que aprobar Álgebra, pero para matricularse en Álgebra hay que tener aprobada AED? Obviamente, no puede haber ciclos.

En planificación de tareas aparecen cuestiones del tipo: ¿en qué orden se deben ejecutar las tareas para que se cumplan las precedencias? ¿Qué tareas se pueden ejecutar simultáneamente? ¿Cuánto tiempo puede tardar como mínimo la ejecución del plan completo? ¿Cuáles son las tareas críticas, es decir, las que no se pueden retrasar?

Órdenes parciales en conjuntos

El concepto matemático subyacente a los GDA es la representación de órdenes parciales en un conjunto.

Definición 5.14 Un **orden parcial** en un conjunto S es una relación binaria R que cumple las siguientes propiedades:

- **Propiedad irreflexiva.** Para todo $a \in S$, $a R a$ es falso.
- **Propiedad transitiva.** Para todo $a, b, c \in S$, si $a R b$ y $b R c$ entonces $a R c$.

Por ejemplo, la relación de menor estricto, “ $<$ ”, en los reales o la inclusión propia, “ \subset ”, en los conjuntos son órdenes parciales. En la figura 5.31 se muestra un ejemplo de GDA para representar un orden parcial.

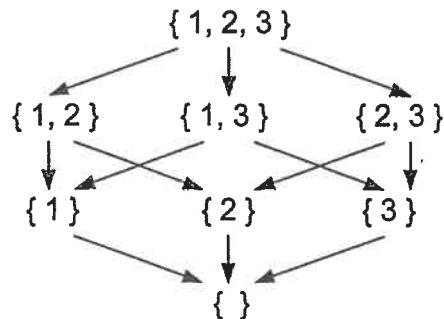


Figura 5.31: Representación de un orden parcial mediante un GDA. Relación “ \subset ” en el conjunto de los subconjuntos de $\{1, 2, 3\}$.

¿Por qué no pueden existir ciclos en el grafo asociado a un orden parcial? Si existiera un ciclo de relaciones, por ejemplo que contenga un elemento a , entonces podríamos aplicar la propiedad transitiva y deducir que $a R a$, lo cual no es posible.

Ordenación topológica de un GDA

Sobre los GDA es posible definir un tipo específico de recorrido, conocido como **recorrido en orden topológico**. En el recorrido en orden topológico, si existe un camino de un nodo v a w entonces v debe visitarse antes que w . El recorrido topológico da lugar a la **numeración en orden topológico**.

Definición 5.15 La **numeración en orden topológico** de un GDA, $G = (V, A)$, es una función $orden : V \rightarrow [1..n]$, tal que si $\langle v, w \rangle \in A$, entonces $orden(v) < orden(w)$.

La ordenación topológica de un grafo puede tener diferentes utilidades. Por ejemplo, en el GDA de expresiones aritméticas el orden topológico, recorrido de mayor a menor, indica el orden en el que deben calcularse las distintas subexpresiones para obtener el resultado final. En el grafo de planificación de tareas, el orden topológico indica una ejecución secuencial de las tareas que respeta las precedencias. Por ejemplo, un orden posible sería: Licencia de obras, Aplanar terreno, Hacer camino, Comprar piedras, Cincelar piedras, Colocar piedras, Pintar pirámide. Obviamente, la ordenación topológica de un grafo no es única.

Un algoritmo sencillo para encontrar una ordenación topológica consiste en buscar primero un vértice al cual no le apunte ningún arco. Este sería el primero dentro de la ordenación. Entonces lo visitamos y lo eliminamos junto con todas sus aristas. Despues se aplica la misma estrategia con el grafo restante. Buscamos otro vértice que no reciba aristas, es decir que tenga grado de entrada cero, lo visitamos y lo borramos. Y así vamos repitiendo hasta visitar todos los nodos del GDA.

Para poder llevar a cabo la idea anterior, necesitamos conocer el grado de entrada de los vértices del grafo; supondremos que lo tenemos calculado en un array *GradoEnt*, de tamaño n . En cada paso del algoritmo, seleccionamos un vértice v no visitado¹¹ y con

¹¹Si no existe ningún vértice con grado de entrada cero, podemos afirmar que el grafo tiene algún ciclo.

$GradoEnt[v] = 0$. Ahora se actualiza el array $GradoEnt$, decrementando en uno el grado de entrada de los vértices adyacentes a v , con una iteración del tipo: **para cada** w adyacente a v ...

Podemos mejorar el algoritmo si utilizamos una cola de nodos, C , en la que mantengamos en cada momento aquellos vértices con grado cero, pero no seleccionados hasta ahora. De esta manera, elegir el siguiente nodo a visitar requiere un $O(1)$, y no tenemos que recorrer el array cada vez. Si la cola está vacía y no hemos seleccionado todos los vértices, el grafo tiene al menos un ciclo.

```

operación OrdenaciónTopológica ( $GradoEnt$ : array [1..n] de entero;  $var orden$ : array [1..n] de entero)
var C: Cola[entero]
    C:= CrearCola
    contador:=1
    para  $v := 1, \dots, n$  hacer
        si  $GradoEnt[v] = 0$  entonces
            InsCola(C, v)
        finsi
    finpara
    mientras NO EsVaciaCola(C) hacer
         $v :=$  Cabeza(C)
        Sacar(C)
         $orden[v] :=$  contador
        contador:= contador + 1
        para cada  $w$  adyacente a  $v$  hacer
             $GradoEnt[w] := GradoEnt[w] - 1$ 
            si  $GradoEnt[w] = 0$  entonces
                InsCola(C, w)
            finsi
        finpara
    finmientras
    si contador  $\leq n$  entonces
        error("El grafo tiene un ciclo")
    finsi
```

En la figura 5.32 se puede ver un ejemplo de ejecución de este algoritmo. La ordenación topológica de este grafo no es única. ¿Cómo se puede comprobar? Si en algún paso de ejecución la cola C contiene más de un vértice, entonces podríamos seguir por cualquiera de ellos. Por ejemplo, cuando $contador = 4$, la cola contiene dos nodos. En ese caso la ordenación no es única. En realidad, en lugar de una cola podríamos usar una pila, una lista o cualquier otro orden de selección.

El análisis del tiempo de ejecución es muy parecido al que realizamos en los recorridos en profundidad y en anchura. De esta forma, si medimos el trabajo total realizado tenemos un $O(n^2)$ con matrices de adyacencia, y un $O(n + a)$ con listas de adyacencia.

Una manera alternativa de hacer la ordenación topológica es mediante una búsqueda primero en profundidad, numerando los vértices decrecientemente en orden de terminación de las llamadas recursivas. Es decir, para el primer nodo que acaba la llamada recursiva

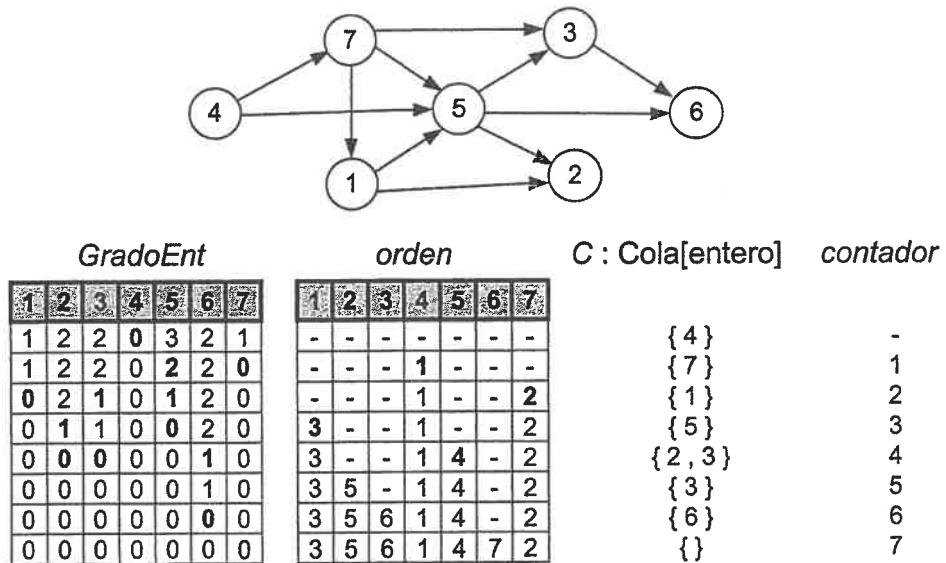


Figura 5.32: Ordenación topológica en un GDA. Arriba: GDA de ejemplo. Abajo: ejecución del algoritmo sobre el ejemplo.

a $\text{bbp}(v)$ tendríamos $\text{orden}[v] = n$, para el siguiente $n - 1$, y así sucesivamente. El orden obtenido es un orden topológico porque un nodo sólo acabará su llamada recursiva cuando todos sus adyacentes hayan sido visitados.

5.6.3. Flujo máximo en redes

En teoría de grafos, un grafo dirigido con pesos es también conocido como una **red**. En los problemas de flujo en redes, las aristas representan canales por los que puede circular cierta cosa: datos, agua, coches, corriente eléctrica, etc. Los pesos de las aristas representan la capacidad máxima de un canal: velocidad de una conexión, volumen máximo de agua, cantidad máxima de tráfico, voltaje de una línea eléctrica, etc.; aunque es posible que la cantidad real de flujo sea menor.

El problema del flujo máximo consiste en lo siguiente: dado un grafo dirigido con pesos, $G = (V, A, W)$, que representa las capacidades máximas de los canales, un nodo de inicio s y otro de fin t en V , encontrar la cantidad máxima de flujo que puede circular desde s hasta t . En la figura 5.33 se muestra un ejemplo de problema y la solución. El grafo de la izquierda, G , pintado con líneas continuas, representa las capacidades máximas; sería la entrada del problema. El grafo de la derecha, F , representado con líneas discontinuas, indica los flujos reales; es una posible solución para el problema.

La solución del problema debe cumplir las siguientes propiedades:

- La suma de los pesos de las aristas que salen de s debe ser igual a la suma de las aristas que llegan a t . Esta cantidad es el **flujo total** entre s y t .
- Para cualquier nodo distinto de s y de t , la suma de las aristas que llegan al nodo

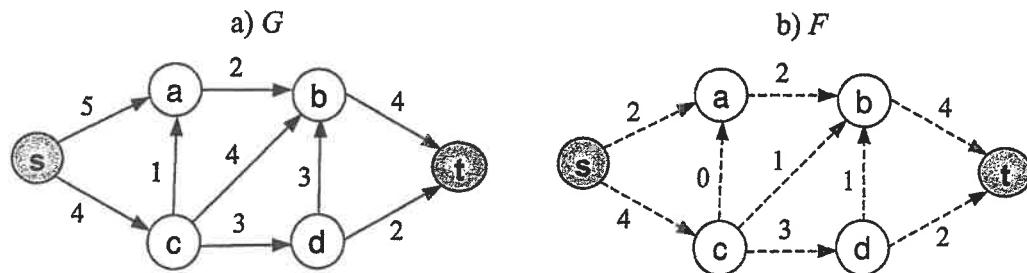


Figura 5.33: Problema de flujo máximo en redes. a) Grafo de capacidades máximas de los canales. b) Solución del problema, grafo de flujos reales.

- debe ser igual a la suma de las aristas que salen al mismo.
- Los pesos de las aristas en F no pueden superar los pesos máximos indicados en G . Es decir, si $C_G(a, b)$ es el peso de la arista $\langle a, b \rangle$ de G y $C_F(a, b)$ es el peso de la misma arista en F , entonces $C_F(a, b) \leq C_G(a, b)$.

Una vez planteado el problema, vamos a estudiar la forma de resolverlo. En primer lugar propondremos un algoritmo intuitivo, y a continuación analizaremos si garantiza la solución óptima o no.

Un posible algoritmo para calcular el flujo máximo

La idea de los flujos, que van desde s hasta t , es muy próxima a la de un camino por el que circula cierto fluido. Cada unidad de flujo que llega hasta un nodo, debe salir por alguna de sus aristas. Por lo tanto, un posible algoritmo podría basarse en encontrar caminos $(s, v_1, v_2, \dots, v_k, t)$ en G . Por ese camino podemos mandar cierta cantidad de flujo. ¿Cuánto? Pues todo lo que quepa. Por ejemplo, si en el grafo G de la figura 5.33 tomamos el camino (s, a, b, t) , vemos que las aristas por las que pasa tienen pesos: 5, 2, 4. El máximo flujo que podemos mandar por ese camino está limitado por el mínimo de las capacidades por las que pasa el camino; en este caso 2. De esta forma, el algoritmo iría encontrando caminos en G , añadiendo los flujos correspondientes al grafo F y quitándolos de G . Y así seguiría hasta que no queden más caminos para enviar flujo.

La estructura del algoritmo que lleva a cabo esta idea sería la siguiente:

- Sea $G = (V, A, C_G)$ el grafo de capacidades máximas. Inicializar el grafo de flujos reales, F , con los mismos nodos y aristas de G , pero con pesos 0. Es decir, $C_F(v, w) = 0; \forall \langle v, w \rangle \in A$. Este grafo guardará el resultado del algoritmo.
- Buscar un camino en G , desde s hasta t , pasando por aristas cuyo peso sea mayor que 0. Este camino es denominado **camino creciente**. Supongamos que el camino es $(s, v_1, v_2, \dots, v_k, t)$. Tomamos $m = \min\{C_G(s, v_1), C_G(v_1, v_2), \dots, C_G(v_k, t)\}$. Es decir, por este camino pueden fluir hasta m unidades de flujo, como máximo.
- Para cada arista $\langle v, w \rangle$ del camino anterior, añadir m al coste de la arista correspondiente en F y quitarlo en G . Es decir, $C_F(v, w) = C_F(v, w) + m; C_G(v, w) =$

$C_G(v, w) - m$; para todo $\langle v, w \rangle$ del camino del paso 2.

4. Volver al paso 2 mientras siga existiendo algún camino creciente entre s y t en G .

Todavía nos queda por determinar la forma de encontrar el camino creciente del paso 2. Una vez más, la búsqueda primero en profundidad puede sernos de utilidad. Para encontrar un camino creciente, podríamos iniciar una búsqueda en profundidad en G a partir del nodo s . Cuando la búsqueda llegue a t ya tenemos un camino de s a t ¹². Además, el procedimiento bpp debería ser modificado para tener en cuenta sólo las aristas con peso mayor que cero.

Por otro lado, está claro que entre s y t puede haber más de un camino creciente. Esta primera versión del algoritmo indica que se encuentre un camino cualquiera. El algoritmo será óptimo si, independientemente de los caminos elegidos en el paso 2, siempre encuentra el flujo máximo. Vamos a ver que no siempre ocurre así.

Ejemplo 5.6 Vamos a aplicar la primera versión del algoritmo del flujo máximo sobre el grafo G de la figura 5.33a). En la figura 5.34 se muestra una ejecución posible del algoritmo, donde no se alcanza la solución óptima.

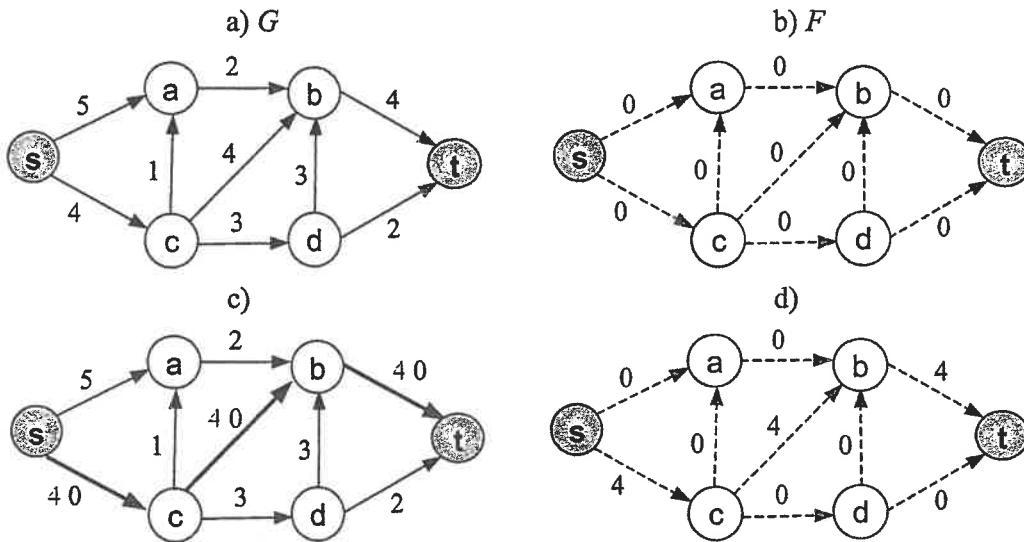


Figura 5.34: Primera versión del algoritmo de flujo máximo en redes. a) Grafo del problema, G . b) Grafo de flujos F inicial. c) Grafo G después de quitar el camino (s, c, b, t) . d) Grafo de flujos después de añadir el mismo camino.

En la primera ejecución del paso 2, se encuentra el camino (s, c, b, t) . Los costes de las aristas son: 4, 4, 4; así que $m = 4$. Esta cantidad se añade en F (figura 5.34d) y se quita de G (figura 5.34c). Si intentamos buscar otro camino entre s y t , en el grafo de la figura 5.34c), que pase por aristas con peso mayor que cero, vemos que no existe ninguno. Por

¹²El camino estaría en la pila de llamadas recursivas. Lo más adecuado sería ir almacenando en un array los nodos que están en la rama actual de la llamada a bpp.

lo tanto, el algoritmo acabaría. El resultado del algoritmo es que el flujo total encontrado es 4.

En consecuencia, el algoritmo no encuentra el óptimo, que como vimos en la figura 5.33 es 6 unidades de flujo. No obstante, si los caminos hubieran sido elegidos en otro orden sí que se habría obtenido el óptimo. En concreto, se puede comprobar que el resultado de la figura 5.33 se alcanzaría si seleccionamos los siguientes caminos, por orden: (s, a, b, t) con peso 2; (s, c, d, t) con peso 2; (s, c, d, b, t) con peso 1; (s, c, b, t) con peso 1.

Algoritmo de flujo máximo deshaciendo caminos

La primera versión del algoritmo es no determinista: en el paso 2 se pueden elegir varios caminos y, dependiendo de cuál se coja, el algoritmo alcanza la solución óptima o no. Para solucionar el problema podemos hacer una pequeña modificación en el algoritmo. El sentido de esta modificación es que si se coge un camino, pero que luego resulta ser una mala decisión, se pueda deshacer el flujo enviado por ese camino.

En particular, la modificación afecta a la forma de actualizar C_G dentro del paso 3. Cada vez que encontramos un camino creciente, quitamos m unidades de flujo de G y las ponemos en F . Ahora, además, vamos a indicar en G que se pueden deshacer m unidades de flujo a través de las aristas del camino. El flujo que se deshace tendrá el sentido opuesto al de añadir; es decir, si se añade m unidades en $\langle v, w \rangle$ en F , entonces se quitan m de $\langle v, w \rangle$ en G y se añaden m unidades de deshacer para la arista $\langle w, v \rangle$ en G .

En definitiva, este cambio sólo implica modificaciones dentro del paso 3 del algoritmo, que ahora debería decir:

- 3 Para cada arista $\langle v, w \rangle$ del camino anterior, añadir m al coste de la arista correspondiente en F , quitarlo en G y ponerlo en G en sentido contrario. Es decir, $C_F(v, w) = C_F(v, w) + m$; $C_G(v, w) = C_G(v, w) - m$; $C_G(w, v) = C_G(w, v) + m$ para todo $\langle v, w \rangle$ del camino del paso 2.

Hay que tener en cuenta que aquí estamos suponiendo que el peso de una arista inexistente es 0. De esta forma, cuando sumamos m a $C_G(w, v)$, pero $\langle v, w \rangle$ no está en G , sería equivalente a crear una nueva arista con peso m .

Esta nueva versión del algoritmo no deja de ser no determinista, pero garantiza siempre la solución óptima. Aunque no lo vamos a demostrar, vamos a ver que se resuelve correctamente el problema que vimos en el ejemplo 5.6.

Ejemplo 5.7 Vamos a aplicar la segunda versión del algoritmo del flujo máximo –la que permite deshacer caminos– sobre el grafo G de la figura 5.33a). Un posible resultado del algoritmo se muestra en la figura 5.35.

Igual que en el ejemplo 5.6, consideraremos que en la primera ejecución del paso 2 se encuentra el camino (s, c, b, t) , con $m = 4$. Esta cantidad se añade en F (figura 5.35d). Ahora, en G se quita esa cantidad en sentido directo y se añade en sentido contrario (figura 5.35c).

A continuación podemos encontrar un nuevo camino, que pasa por la arista de “deshacer” $\langle b, c \rangle$. El camino es (s, a, b, c, d, t) , con pesos: 5, 2, 4, 3, 2. Por lo tanto,

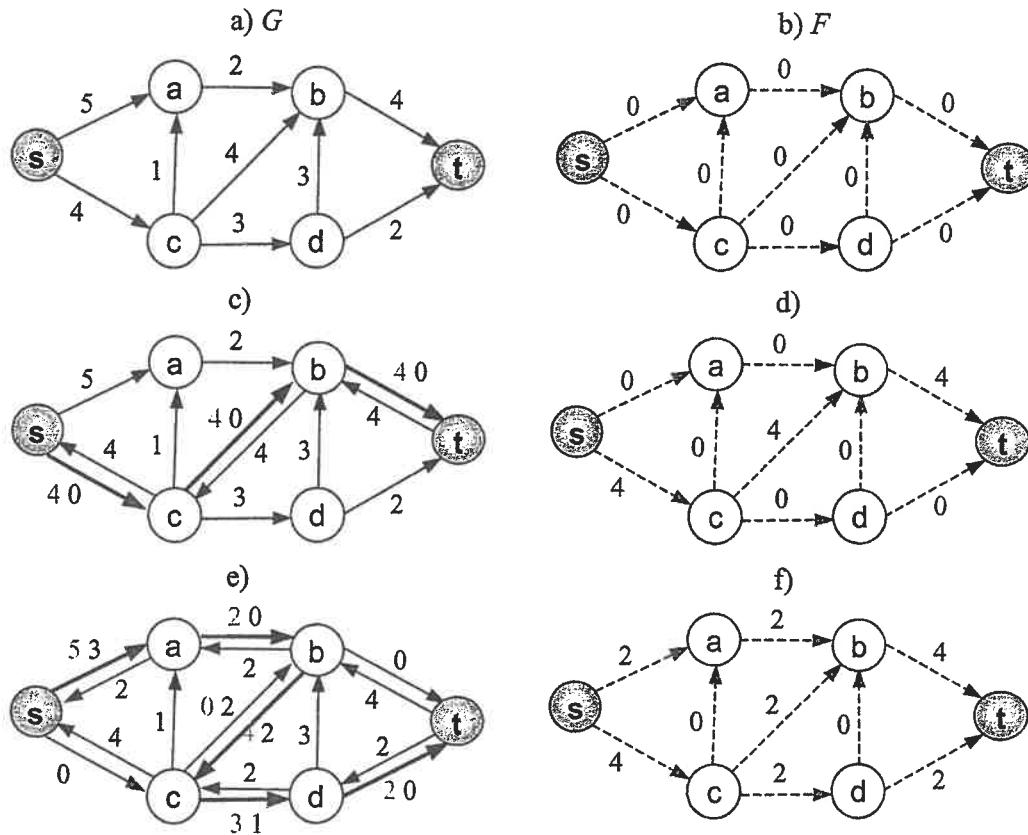


Figura 5.35: Segunda versión del algoritmo de flujo máximo en redes. a) Grafo del problema, G . b) Grafo de flujos F inicial. c),d) Grafos G y F , después de encontrar el camino (s, c, b, t) . e),f) Grafos G y F , después de encontrar el camino (s, a, b, c, d, t) .

$m = 2$. Se añade a F^{13} (figura 5.35f) y se actualiza G (figura 5.35e). En el siguiente paso, ya no existe ningún camino creciente, luego acaba el algoritmo.

Si comparamos la solución obtenida con la mostrada en la figura 5.33, vemos que no coinciden. No obstante, ambas tienen el mismo valor de flujo total, 6, y ambas son óptimas. Es perfectamente posible, como en este ejemplo, que la solución óptima no sea única.

5.7. Algoritmos sobre grafos no dirigidos

En esta sección plantearemos dos problemas específicos de grafos no dirigidos: la búsqueda de los puntos de articulación y los circuitos de Euler. Vamos a ver que ambos problemas se pueden resolver utilizando como herramienta la búsqueda primero en profundidad.

¹³Hay que notar un detalle sutil. Cuando en el grafo de la figura 5.35d) se añade el flujo de “deshacer” entre b y c , no se añade realmente 2 a $C_F(b, c)$, sino que se resta 2 de $C_F(c, b)$. ¿Por qué?

5.7.1. Puntos de articulación y componentes biconexos

Como sabemos, un grafo no dirigido se dice que es conexo si existen caminos entre todos sus nodos. Pero en muchas aplicaciones se requiere un nivel más de conexión; se necesita no sólo que los nodos estén conectados sino que si falla algún nodo o enlace, los nodos sigan conectados. Buscamos lo que se podría denominar *tolerancia a fallos*. En la figura 5.36 se muestran dos ejemplos de aplicación. El grafo de la figura 5.36a) muestra las estrategias de pase del balón de un equipo de fútbol; la figura 5.36b) muestra una red de ordenadores y las conexiones entre los mismos.

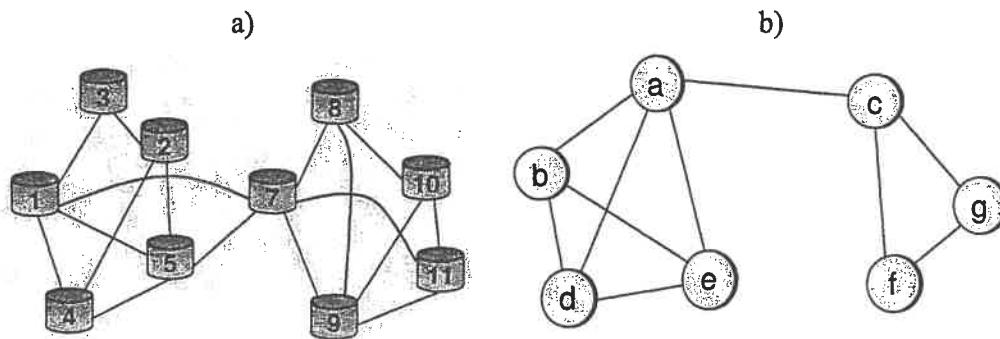


Figura 5.36: Ejemplos de grafos no dirigidos. a) Jugadores de un equipo de fútbol (nodos) y estrategias de pase del balón (aristas). b) Red de ordenadores.

En este tipo de aplicaciones surge el estudio de los puntos de articulación y los componentes biconexos.

Definición 5.16 Sea un grafo no dirigido y conexo, $G = (V, A)$. Un **punto de articulación** es un vértice v tal que cuando se elimina de G junto con todas las aristas incidentes en él, se divide una componente conexa de G en dos o más.

El punto de articulación es un nodo *crítico* del grafo, en el sentido de que si falla tendremos graves problemas. Por ejemplo, en la figura 5.36a) el jugador 7 es un punto de articulación; si conseguimos eliminarlo cortaremos la circulación del balón entre la defensa y la delantera. En la figura 5.36b) los ordenadores a y c son puntos de articulación; si cualquiera de ellos cae, quedarán trozos de red incomunicados.

Un grafo no dirigido se dice que es **biconexo** si no tiene puntos de articulación. En los anteriores apartados hemos hablado informalmente de grafos más o menos conectados. Podemos definir el concepto de conectividad de un grafo de la siguiente forma.

Definición 5.17 Un grafo no dirigido se dice que tiene **conectividad k** si la eliminación de $k - 1$ vértices cualesquiera, junto con las aristas incidentes en ellos, no desconecta el grafo resultante.

De acuerdo con la definición, un grafo tiene conectividad 2 o más si y sólo si no tiene puntos de articulación, es decir, si es biconexo. Por ejemplo, los grafos de la figura 5.36

no son biconexos, por lo que su conectividad es 1. Cuanto mayor sea la conectividad del grafo, más fácil será que *sobreviva* al fallo de alguno de sus vértices. Por otro lado, según la definición, si un grafo tienen conectividad k , también tendrá conectividad $k - 1, k - 2, \dots, 1$.

Ejemplo 5.8 ¿Cuánto es la máxima conectividad posible de un grafo? ¿A qué grafo corresponde? ¿Cuántas aristas debe tener un grafo como mínimo para ser biconexo?

La máxima conectividad posible sería la de un grafo completo. Podemos eliminar todos los vértices sin desconectarlo. Sólo cuando eliminemos todos los nodos conseguimos suprimir un componente conexo. Podemos decir que el grafo completo tiene conectividad n , siendo n el número de nodos del grafo.

En cuanto al mínimo número de aristas, sabemos que un grafo con forma de árbol es el menor grafo conexo posible. Un grafo de ese tipo tiene $n - 1$ aristas, pero no es biconexo. Todos los nodos, excepto las hojas, serían puntos de articulación. Consideremos un grafo con forma de anillo. Tiene n aristas y es biconexo, ya que no tiene puntos de articulación. Por lo tanto, el mínimo número de aristas para un grafo biconexo sería n . Ojo, esto no garantiza que cualquier grafo no dirigido con n aristas sea biconexo.

Algoritmo para calcular los puntos de articulación

A falta de una idea mejor, una posible solución para calcular los puntos de articulación de un grafo podría ser la siguiente: eliminar los vértices del grafo uno por uno; para cada nodo eliminado, comprobar si el grafo resultante sigue siendo conexo o no; en caso negativo, tenemos un punto de articulación. La comprobación de si el grafo resultante es conexo se podría hacer con una búsqueda primero en profundidad. En consecuencia, el tiempo de esta solución sería $O(n^3)$ con matrices de adyacencia y $O(n(a + n))$ con listas. Pero vamos a ver que el problema se puede resolver con una simple búsqueda en profundidad.

Supongamos que hacemos la búsqueda en profundidad de un grafo no dirigido y conexo, como el de la figura 5.36b). Obtenemos un árbol abarcador en profundidad, como el mostrado en la figura 5.37.

En el resultado aparecen dos tipos de arcos: los del árbol y los que no son del árbol, que serán de avance/retroceso. Podemos interpretar que tenemos dos tipos de caminos entre los nodos: los caminos a través del árbol, hacia los padres, y los caminos moviéndonos a través de los arcos de retroceso. Los primeros, a través del árbol, siempre existirán; los segundos, a través de arcos de retroceso, representan caminos *alternativos*. Decimos que son alternativos en el sentido de que si eliminamos un nodo en el camino del primer tipo, tenemos una vía alternativa.

El algoritmo se basa en el cálculo de los caminos alternativos. Se realiza una búsqueda primero en profundidad, numerando los nodos en el orden de recorrido. Al mismo tiempo se calculan los "caminos alternativos". El resultado se guarda en un array *bajo: array [1..n] de entero*. El valor de *bajo* de un nodo v indica lo más arriba que podemos llegar en el árbol a través de un camino alternativo. El camino alternativo para un nodo consistirá en moverse a través de un arco de retroceso, o bien hacia abajo en el árbol y luego hacia arriba por un arco de retroceso. En la figura 5.37b) se muestra un ejemplo.

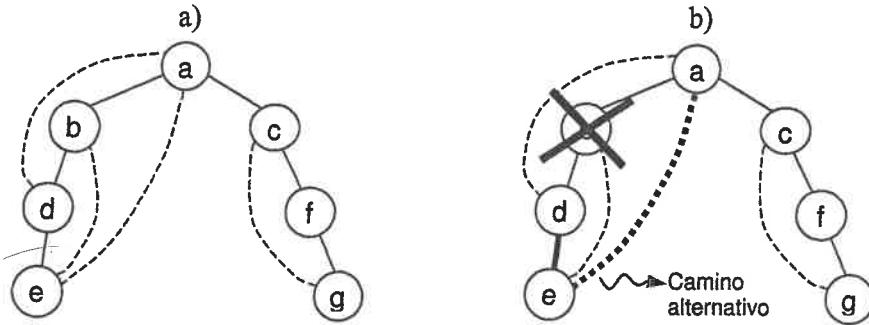


Figura 5.37: Caminos alternativos. a) Árbol de expansión en profundidad del grafo de la figura 5.36b). b) Si eliminamos el nodo b , existe para d y para e un camino alternativo para llegar a la raíz. Conclusión: b no es punto de articulación.

Tendremos un punto de articulación si, al eliminar un nodo, alguno de sus hijos no tiene un camino alternativo para llegar más arriba del nodo eliminado. Es decir, v será un punto de articulación si tiene un hijo tal que su valor de *bajo* es menor o igual que el número de búsqueda en profundidad de v . La condición es distinta para la raíz; será punto de articulación si tiene dos o más hijos. Como ya vimos, no pueden haber arcos de cruce, por lo que si tiene dos o más hijos estos sólo se pueden comunicar a través de la raíz.

En definitiva, el algoritmo para calcular los puntos de articulación de un grafo no dirigido tendría la siguiente estructura:

1. Realizar una búsqueda primero en profundidad, numerando los nodos en el orden en que son recorridos. Supongamos que guardamos en el array *nbpp*: array [1..n] de entero, el orden en que es visitado cada nodo.
2. Calcular los valores $bajo[v]$ para cada nodo visitado, según la fórmula:
 $bajo[v] := \min\{nbpp[z] \mid \text{para todo } z \text{ tal que existe un arco de retroceso } (z,v);$
 $bajo[y] \mid \text{para todo } y \text{ hijo de } v \text{ en el árbol de expansión}\}$
3. La raíz del árbol es un punto de articulación si y sólo si tiene dos o más hijos en el árbol de expansión en profundidad.
4. Un nodo v , distinto de la raíz, es un punto de articulación si y sólo si tiene algún hijo w en el árbol tal que $bajo[w] \geq nbpp[v]$.

Realmente, el cálculo de los valores de *bajo* y la comprobación de las condiciones no se deben hacer como pasos separados, sino dentro del mismo procedimiento de bpp. Como los cálculos añadidos se pueden hacer en tiempo constante, el orden de complejidad del algoritmo viene dado por el orden de la búsqueda primero en profundidad: $O(n^2)$ con matrices de adyacencia y $O(n + a)$ con listas de adyacencia.

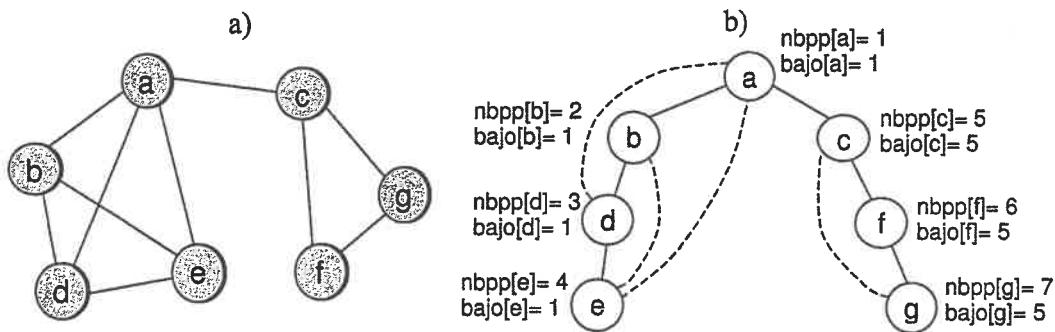


Figura 5.38: Búsqueda de los puntos de articulación. a) Grafo no dirigido. b) Aplicación del algoritmo. Se indican los valores de *bajo* y *nbpp*.

Ejemplo 5.9 En la figura 5.38 se muestra un ejemplo de aplicación de este algoritmo, para el mismo grafo de la figura 5.36a).

¿Cuáles son los puntos de articulación del ejemplo de la figura 5.38? Para la raíz, *a*, se cumple la condición del punto 3, puesto que tiene dos hijos. Por lo tanto, *a* es un punto de articulación. Para los demás nodos, tenemos que comprobar la condición del punto 4. Vemos que se cumple únicamente para el nodo *c*; tiene como hijo el nodo *f*, cuyo *bajo[f]=5 ≥ nbpp[c]=5*. En consecuencia, *f* es otro punto de articulación.

Componentes biconexos

La definición de componente biconexo es similar a la de componente conexo. Un **componente biconexo** de un grafo *G* es un subgrafo biconexo y maximal de *G*. Si *G* es de por sí biconexo, entonces tendrá un sólo componente biconexo. Si *G* tiene puntos de articulación, entonces aparecerán distintos componentes biconexos. ¿Cómo encontrarlos?

En primer lugar, deberíamos calcular los puntos de articulación de *G*. Después, el algoritmo sería similar al cálculo de los componentes conexos que vimos en el ejemplo 5.4. Recordemos que este algoritmo se basaba en una simple búsqueda primero en profundidad. La única diferencia es que no se deberían hacer llamadas recursivas al llegar a un nodo que sea punto de articulación.

En la figura 5.39 se muestran los componentes biconexos del grafo del ejemplo 5.9. Se puede ver que algunos nodos están en más de un componente biconexo. Esto ocurrirá, precisamente, para los nodos que sean puntos de articulación.

5.7.2. Circuitos de Euler

Los problemas de circuitos de Euler aparecen cuando se utilizan grafos para representar dibujos de líneas. En la figura 5.40 se muestran tres de estos ejemplos. En estos grafos, cada nodo representa un punto del dibujo y una arista entre dos nodos indica que existe una línea entre los dos puntos correspondientes.

El problema del circuito de Euler trata de responder a la cuestión: ¿es posible dibujar la figura con un bolígrafo, pintando cada línea una sola vez, sin levantar el bolígrafo y

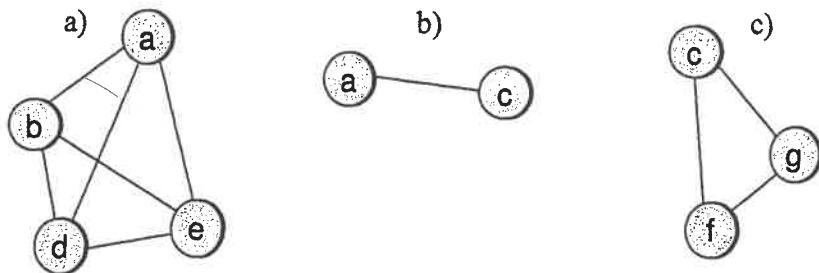


Figura 5.39: Componentes biconexos del grafo del ejemplo 5.9.

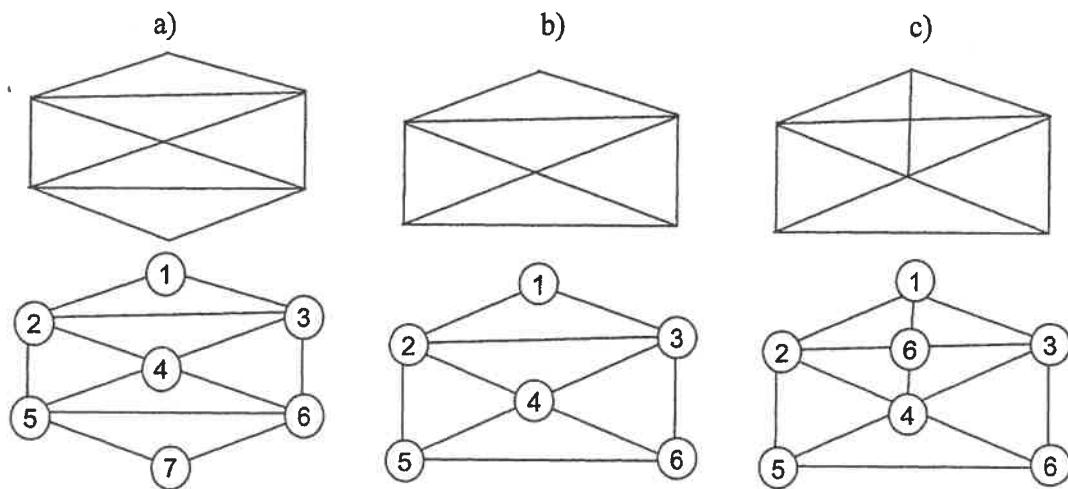


Figura 5.40: Circuitos y caminos de Euler. Arriba: dibujo de líneas. Abajo: grafo correspondiente. a) Tiene un circuito de Euler. b) Tiene un camino de Euler. c) No tiene circuito ni camino de Euler.

acabando en el mismo punto donde se empezó?

Formulado con terminología de grafos, decimos que un **círculo de Euler** es un ciclo, no necesariamente simple, que visita todas las aristas exactamente una vez. Es decir, los nodos se pueden visitar varias veces, pero las aristas se tienen que visitar una y solo una vez. Así que dado un grafo, tenemos que decidir si existe un circuito de Euler.

Hay claramente dos condiciones necesarias para que exista un circuito de Euler:

- El grafo debe ser conexo.
- El grado de todos los nodos –es decir, número de aristas incidentes– debe ser par. La razón es que si el camino pasa varias veces por un nodo, siempre que entra por una arista debe salir por otra.

Un concepto relacionado con el circuito de Euler es el de camino de Euler. Un **camino de Euler** es un camino que visita todas las aristas exactamente una vez, pudiendo empezar y acabar en sitios distintos. Las condiciones necesarias para que exista, en este

caso, serían similares a las del circuito de Euler. La segunda condición permitiría que existan dos nodos con grado impar, o ninguno. Los demás deberían tener grado par.

Se puede comprobar que las anteriores condiciones necesarias son también suficientes. Esto es, si se cumplen entonces existe un circuito de Euler. El objetivo ahora es encontrar un algoritmo para encontrar el circuito o camino de Euler.

Algoritmo para calcular un circuito de Euler

Supongamos un grafo no dirigido G . En primer lugar deberíamos comprobar si se cumplen las condiciones necesarias y suficientes. En ese caso, tenemos garantizado que existe un circuito de Euler, y podemos encontrarlo con el siguiente algoritmo:

1. Buscar un ciclo en G empezando por un vértice v cualquiera. Puede que en este ciclo no todas las aristas hayan sido visitadas. En ese caso, seguimos con el siguiente punto.
2. Si quedan aristas por visitar, seleccionar el primer nodo w del ciclo anterior que tenga una arista no visitada. Buscar un ciclo partiendo de w que pase por aristas no visitadas.
3. Unir el ciclo del paso 1 con el obtenido en el paso 2. Repetir sucesivamente los pasos 2 y 3 hasta que no queden aristas por visitar.

Para encontrar un ciclo, en los pasos 1 y 2, podemos utilizar una búsqueda primero en profundidad, como se comenta en el apartado 5.3.1. No obstante, la diferencia ahora es que lo que se marca como visitado no son los nodos sino las aristas.

Ejemplo 5.10 Vamos a aplicar el algoritmo para calcular el circuito de Euler para el grafo de la figura 5.40.

Supongamos que empezamos por el nodo 1. Podemos tener algo como lo siguiente:

Paso 1. Encontramos el ciclo: $C = (1, 2, 5, 7, 6, 3, 1)$.

Paso 2. No todas las aristas están visitadas. El primer nodo con una arista no visitada es el 2. Encontramos el siguiente ciclo: $D = (2, 3, 4, 2)$.

Paso 3. Unimos los ciclos C y D . Lo que hacemos es, dentro de C en el lugar donde aparece 2 sustituirlo por D . El resultado es: $C = (1, 2, 3, 4, 2, 5, 7, 6, 3, 1)$.

Paso 2. No todas las aristas están visitadas, el primer nodo es 4. Encontramos el ciclo: $D = (4, 5, 6, 4)$.

Paso 3. Unimos el ciclo C con el D , Obtenemos: $C = (1, 2, 3, 4, 5, 6, 4, 2, 5, 7, 6, 3, 1)$. Todas las aristas están ya visitadas, por lo que acaba el algoritmo.

5.8. Otros problemas con grafos

Además de los problemas que hemos estudiado en los apartados anteriores, existen otros muchos tipos de problemas clásicos en teoría de grafos. Todos ellos aparecen como resultado de modelar un problema de la vida real a través de grafos. Así que disponer de una solución eficiente para los mismos resultaría deseable.

Sin embargo, existe una amplia variedad de problemas para los cuales no se conoce hasta la fecha ningún algoritmo capaz de resolverlos de forma eficiente —a pesar de los numerosos años de esfuerzo dedicados y del fenomenal avance que supondría encontrarlos. Entendemos aquí por eficiente un algoritmo con tiempos polinomiales, del tipo $O(n)$, $O(n^2)$, $O(n^{32})$, etc. Esta categoría de problemas para los cuales no se conocen soluciones eficientes —ya sean sobre grafos o no— son conocidos como **problemas NP**. Los algoritmos existentes para resolver problemas de este tipo se basan, en esencia, en comprobar todas las posibles soluciones de manera exhaustiva, dando lugar a tiempos exponenciales o factoriales. El resultado es lo que se conoce como el efecto de **explosión combinatoria**, que hace referencia a la forma en la que se dispara el tiempo de ejecución para tamaños grandes del problema. Alternativamente, se pueden diseñar algoritmos que obtengan soluciones más o menos *buenas*, no necesariamente la óptima, pero en un tiempo reducido. Estos son conocidos como **algoritmos heurísticos**.

Vamos a ver un conjunto de problemas clásicos sobre grafos, que están dentro de la categoría NP. En este apartado únicamente enunciaremos el problema, mostrando las posibles aplicaciones prácticas donde puede ser de utilidad. Por el momento, no daremos algoritmos para resolverlos. Conforme se avance hacia los capítulos de diseño de algoritmos, sería adecuado intentar plantearse la resolución de estos problemas con las técnicas que se vayan estudiando.

5.8.1. Ciclos hamiltonianos

En principio, el problema del ciclo hamiltoniano tiene una formulación muy similar a la del circuito de Euler. Veamos la definición.

Definición 5.18 Dado un grafo no dirigido G , se llama **ciclo de Hamilton** o **ciclo hamiltoniano** a un ciclo simple que visita todos los vértices.

Es decir, el ciclo hamiltoniano pasa por todos los vértices exactamente una vez. El **problema del ciclo hamiltoniano** consiste en: dado un grafo no dirigido G , determinar si posee algún ciclo hamiltoniano. Por ejemplo, en el grafo de la figura 5.41a) existe un ciclo de Hamilton, que ha sido señalado con línea más gruesa. El de la figura 5.41b) no posee ningún ciclo hamiltoniano, lo cual ha sido verificado comprobando exhaustivamente todos los posibles caminos. El de la figura 5.41c) se deja como pasatiempo.

Aunque parece similar al del circuito de Euler, la complejidad implícita del problema del ciclo de Hamilton resulta muchísimo mayor. Mientras que el circuito de Euler se puede resolver aplicando varias búsquedas en profundidad, para el ciclo de Hamilton no se conoce ningún algoritmo capaz de resolverlo en un tiempo polinomial¹⁴. Se encuentra dentro de los problemas NP.

La solución para este problema consistiría en encontrar todos los posibles caminos simples, comprobando si alguno de ellos es un ciclo hamiltoniano. Si el grafo es completo, el número de caminos simples distintos sería un $(n - 1)!$. Para reducir el tiempo, podemos diseñar algún algoritmo heurístico. Por ejemplo, podemos aplicar una búsqueda en profundidad en la cual de todos los adyacentes a un nodo se visitan primero los que tengan

¹⁴Lo cual, como veremos en el último capítulo, no quiere decir que no exista.

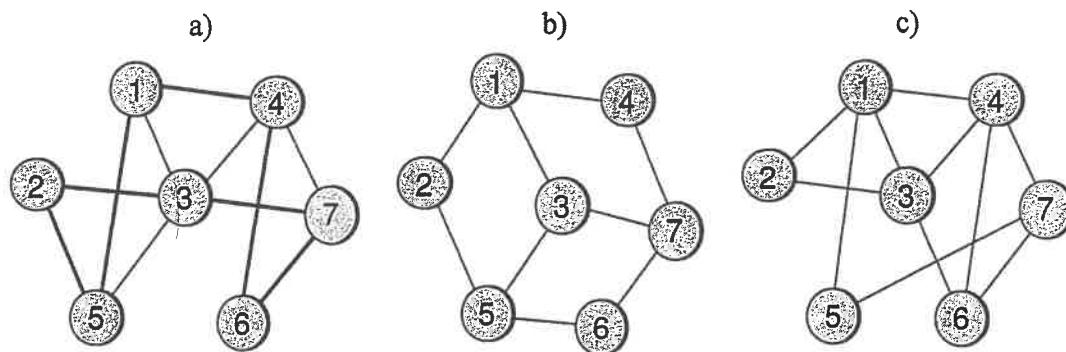


Figura 5.41: Problema del ciclo hamiltoniano. a) Grafo con un ciclo hamiltoniano. b) Grafo que no posee ningún ciclo de Hamilton. c) ¿Existe algún ciclo hamiltoniano?

menor grado. Si lo aplicamos sobre el grafo de la figura 5.41a), encontramos la solución. Empezando por el 1, nos vamos al 5 (grado 3), luego al 2, 3, 7, 6, 4 y cerramos el ciclo. Pero este algoritmo no siempre garantiza la solución. Y si no la encuentra puede que exista o puede que no, como ocurre con el grafo de la figura 5.41c).

5.8.2. Problema del viajante

El **problema del viajante**, también conocido como **problema del agente viajero**, es uno de los más recurridos en las aplicaciones que utilizan grafos para representar caminos con costes asociados¹⁵. La formulación del problema es la siguiente: dado un grafo no dirigido, completo y con pesos, G , encontrar el ciclo simple de coste mínimo que recorra todos los nodos.

El ciclo al que se refiere el problema sería un ciclo hamiltoniano. Pero la dificultad ahora no está en determinar si existe o no ese ciclo –ya que, al ser completo, trivialmente se sabe que existirá siempre– sino en encontrar el que tenga menor coste de todos ellos. En la figura 5.42a) se muestra un grafo de ejemplo, y en la 5.42b) se muestra una posible solución. ¿Es la solución óptima?

Las aristas del grafo pueden tener coste $+\infty$. Así que decir que el grafo debe ser completo es sólo una forma de interpretar los datos: si el grafo no es completo, las aristas faltantes son consideradas con coste $+\infty$.

Ejemplo 5.11 El problema del viajante está subyacente en aplicaciones de tipo “reparto de mercancías”. Por ejemplo, un camionero distribuye pimientos murcianos por toda la región. Tiene que pasar por varios supermercados, n , pero el orden le es indiferente. Eso sí, el camión debe volver al mismo sitio de donde salió. Cada camino necesita un tiempo determinado. El objetivo es planificar la ruta que tiene que seguir, de forma que el tiempo total sea el menor.

La equivalencia con el problema del viajante es inmediata: los n supermercados son los nodos del grafo; los caminos son las aristas, siendo el tiempo el peso de las mismas; y

¹⁵Además de los problemas de caminos mínimos vistos en el apartado 5.5, claro.

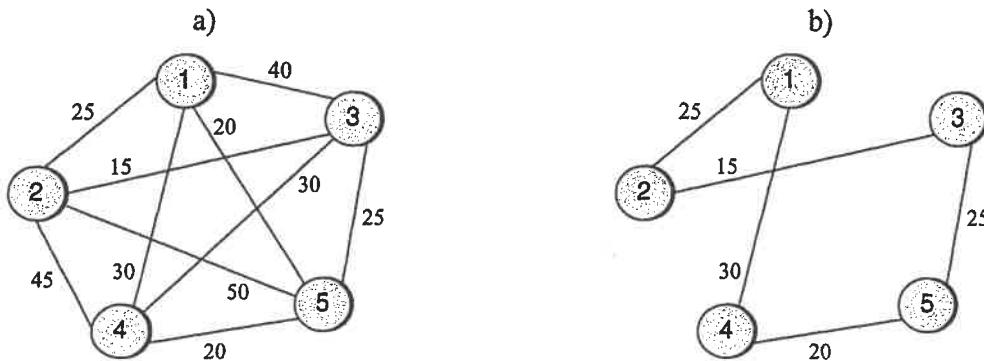


Figura 5.42: Problema del viajante. a) Grafo no dirigido completo y con pesos. b) Posible solución al problema.

el objetivo buscado es el menor ciclo simple que visita todos los nodos.

Teniendo en cuenta que existen $(n - 1)!$ posibles ciclos simples, un algoritmo óptimo sencillo estaría en un orden de complejidad factorial. Se han desarrollado numerosas técnicas heurísticas para abordar este problema, como algoritmos voraces, búsqueda local, algoritmos genéticos e incluso computación con ADN. Pero ninguno de ellos garantiza que se obtenga el óptimo, o que la solución obtenida sea próxima a la óptima.

Igual que para el problema del ciclo hamiltoniano, no se conoce ningún algoritmo eficiente que lo resuelva de forma óptima. Es más, se puede demostrar que ambos problemas son equivalentes. La equivalencia se obtiene viendo que es posible transformar un problema en el otro y viceversa. Por ejemplo, para resolver el problema del ciclo hamiltoniano en un grafo no dirigido $G = (V, A)$, podemos considerar el grafo $G' = (V, A', W)$, donde A' contiene todas las aristas posibles, y $W(v, w) = 1$ si la arista $(v, w) \in A$ y $W(v, w) = +\infty$ en caso contrario. Si resolvemos el problema del viajante en G' , entonces existirá un ciclo hamiltoniano siempre que el coste sea menor que $+\infty$.

5.8.3. Coloración de grafos

En los problemas de coloración de grafos, las aristas no representan *caminos* sino *incompatibilidades*. Los nodos representan cierto tipo de objeto y existe una arista (v, w) entre dos nodos si los objetos v y w son incompatibles. La coloración de un grafo consiste en asignar un **color** o **etiqueta** a cada nodo, de forma que dos nodos incompatibles no tengan el mismo color. Formalmente se puede definir de la siguiente manera.

Definición 5.19 Dado un grafo $G = (V, A)$ no dirigido, una **coloración del grafo** es una función $C : V \rightarrow N$, tal que si $(v, w) \in A$ entonces $C(v) \neq C(w)$.

El **problema de la coloración** de grafos consiste en: dado un grafo no dirigido, encontrar una coloración del mismo utilizando el mínimo número de colores distintos. En la figura 5.43 se muestra un ejemplo del problema. Las figuras 5.43b) y 5.43c) son dos

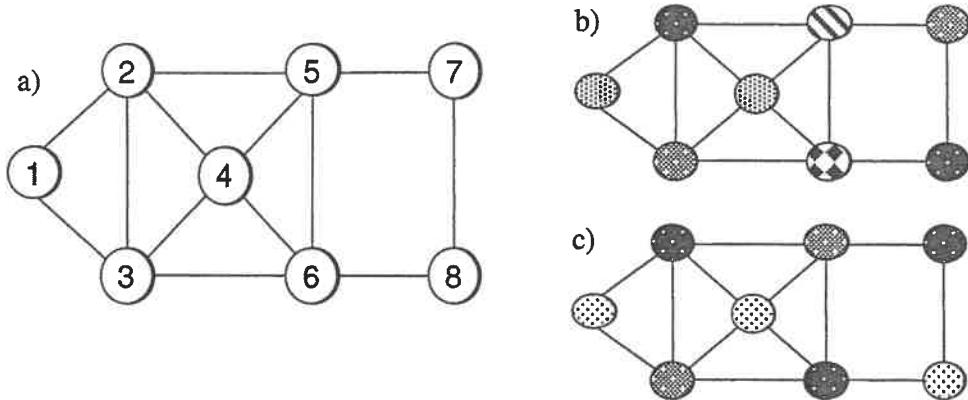


Figura 5.43: Problema de coloración de grafos. a) Grafo no dirigido del problema. b) Coloración usando 5 colores. c) Coloración usando 3 colores.

posibles coloraciones del grafo. La coloración de la figura 5.43c) utiliza menos colores, y se puede comprobar que es la solución óptima de este ejemplo.

El problema de coloración de grafos es también un problema NP. La solución consistiría en comprobar todas las posibles coloraciones y quedarnos con la que use menos colores.

Ejemplo 5.12 En un mapa geopolítico aparecen dibujadas distintas regiones, cada una de las cuales es fronteriza con otras regiones. Queremos llenar de color las regiones del mapa, de forma que dos regiones fronterizas no tengan el mismo color. Para ahorrar costes de producción, la editorial pide que se use el mínimo número de colores distintos. ¿Cuántos colores diferentes necesitamos, como mínimo, para colorear el mapa?

El problema puede ser modelado usando grafos. Cada región del mapa se corresponderá con un nodo del grafo. Habrá una arista entre dos nodos si las regiones asociadas son fronterizas. En la figura 5.44 aparece un ejemplo de transformación de un problema de mapas a un problema de grafos.

Está claro que el resultado del problema será la coloración mínima del grafo. Los grafos que surgen en este tipo de aplicación son llamados **grafos planos**. Un grafo se dice que es plano si se puede dibujar en papel sin que se crucen las aristas. Está demostrado que cualquier grafo plano puede dibujarse usando como máximo cuatro colores. Pero para los grafos no planos el número de colores depende del caso.

5.8.4. Isomorfismo de grafos

El isomorfismo de grafos es una *generalización* de la igualdad entre dos grafos. Por definición, dados dos grafos G y F , se dice que son iguales cuando $V(G) = V(F)$ y $A(G) = A(F)$. Pero lo que normalmente interesa no es la comparación de grafos en igualdad, sino conocer si los grafos tienen una estructura *equivalente*. Esto es lo que llamamos isomorfismo.

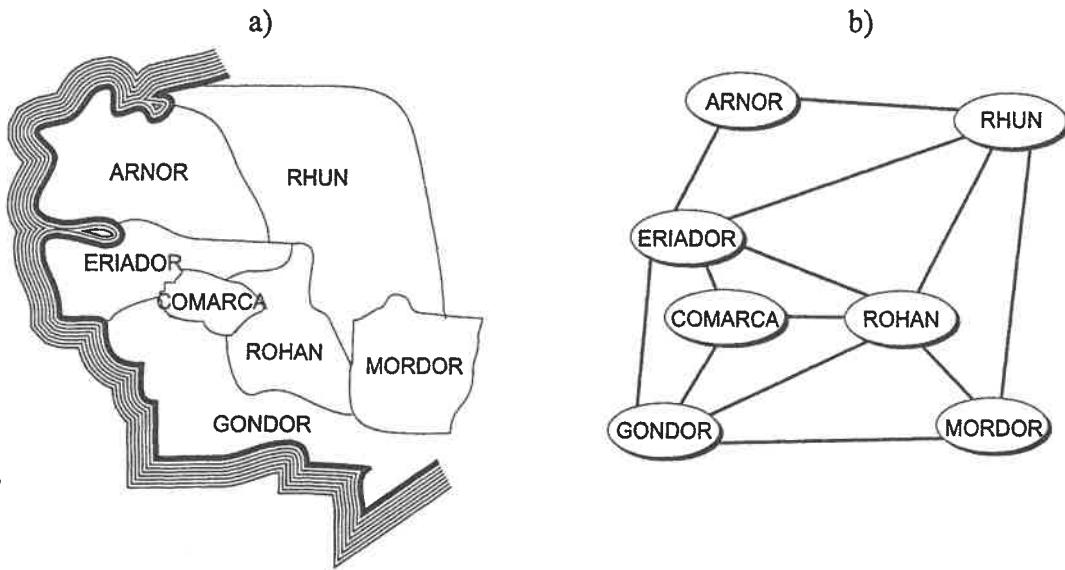


Figura 5.44: Problema de coloración de un mapa. a) Mapa de la Tierra Media. b) Transformación a un problema de grafos.

Definición 5.20 Dados dos grafos G y F , un **isomorfismo** entre grafos es una asignación de los vértices de $V(G)$ con los vértices de $V(F)$ tal que se respetan las aristas. Es decir, es una función biyectiva: $f : V(G) \rightarrow V(F)$, tal que para todo $v, w \in V(G)$, $(v, w) \in A(G) \Leftrightarrow (f(v), f(w)) \in A(F)$.

Dos grafos se dice que son **isomorfos** si existe un isomorfismo entre ellos. Por ejemplo, entre los grafos de la figura 5.45 existen varios posibles isomorfismos. Uno de ello podría ser la asignación: $1 \rightarrow b$; $2 \rightarrow a$; $3 \rightarrow d$; $4 \rightarrow e$; $5 \rightarrow c$; $6 \rightarrow g$; $7 \rightarrow f$.

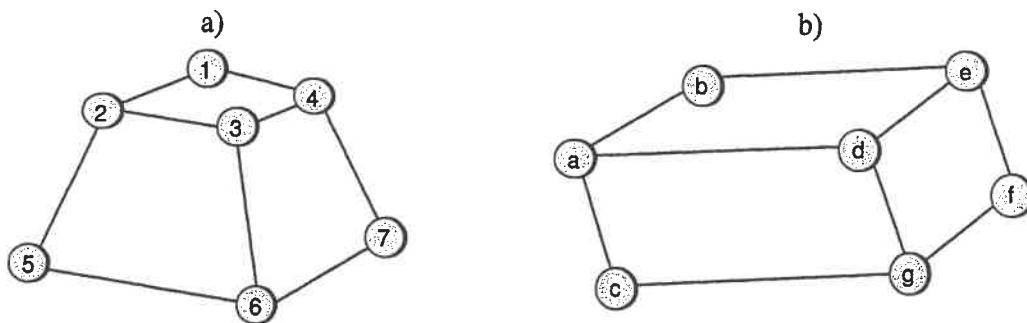


Figura 5.45: Problema del isomorfismo de grafos. a),b) Dos grafos no dirigidos isomorfos.

Nuevamente, el isomorfismo de grafos es un problema NP. La solución consistiría en ir comprobando todas las posibles asignaciones, hasta encontrar alguna válida. Esto requeriría un orden de tipo factorial. Si el grafo es etiquetado, entonces la asignación

debería respetar también los pesos o etiquetas de las aristas. En ciertas aplicaciones, puede surgir el problema del **subisomorfismo** de grafos. Intuitivamente, el subisomorfismo es la mayor asignación posible entre los nodos de los dos grafos, que *respete* las aristas. Vamos a ver un ejemplo.

Ejemplo 5.13 Una aplicación de visión artificial analiza una imagen y debe interpretar qué es lo que está viendo. Su mundo se reduce a un conjunto de p poliedros, de cada uno de los cuales posee un modelo. El problema es: dada una imagen decir dónde se encuentran los objetos.

Para resolverlo creamos un grafo a partir de la imagen, donde los vértices son puntos del dibujo y las aristas del grafo son líneas del dibujo. En la figura 5.46 se muestra un ejemplo con dos modelos de poliedros (figuras 5.46a) y b) y una escena para localizar los modelos (figura 5.46c).

¿Cómo resolver el problema algorítmicamente?

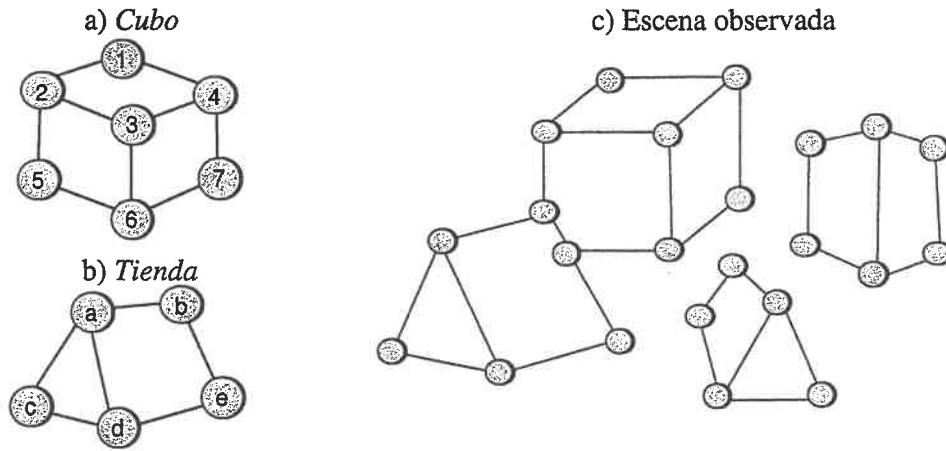


Figura 5.46: Aplicación del isomorfismo de grafos. a),b) Modelos de grafo asociados a dos poliedros. c) Grafo extraído de un dibujo de líneas.

En primer lugar, podríamos separar el grafo del dibujo en sus componentes conexos. Idealmente, cada componente debería ser isomorfo a algún grafo del modelo. Pero, puesto que puede haber ruido, imperfecciones o solapamientos, lo que tenemos realmente es un problema de subisomorfismo. Por ejemplo, de los tres componentes de la figura 5.46c), en uno de ellos podemos encontrar un isomorfismo con la figura denominada *Tienda*, en otro encontramos dos subisomorfismos con *Tienda* y con *Cubo*, y en el tercero no hay ningún isomorfismo adecuado.

Ejercicios resueltos

Ejercicio 5.1 El grafo de la figura 5.47 representa una red de ordenadores, con enlaces entre los mismos. Todos ellos se pueden comunicar entre sí, directamente o a través de otros. Encontrar los ordenadores críticos, es decir los que no pueden fallar para que todos

los ordenadores sigan estando conectados. ¿Es posible añadir algún enlace para que dejen de existir estos nodos críticos?

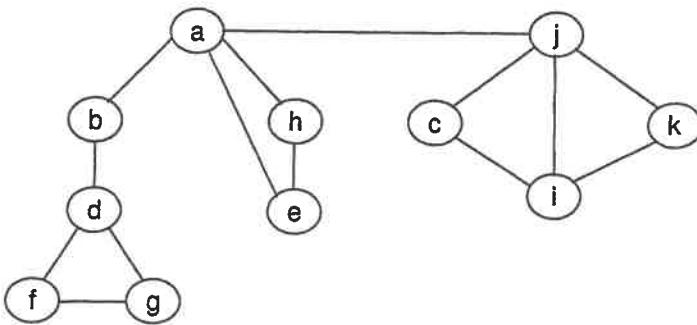


Figura 5.47: Red de ordenadores del ejercicio 5.1.

Solución.

Claramente, tenemos un problema de puntos de articulación. Los puntos de articulación del grafo serán los nodos que no pueden fallar para que la red siga conectada. Aplicando el algoritmo obtenemos las numeraciones de nbpp y bajo mostradas en la figura 5.48.

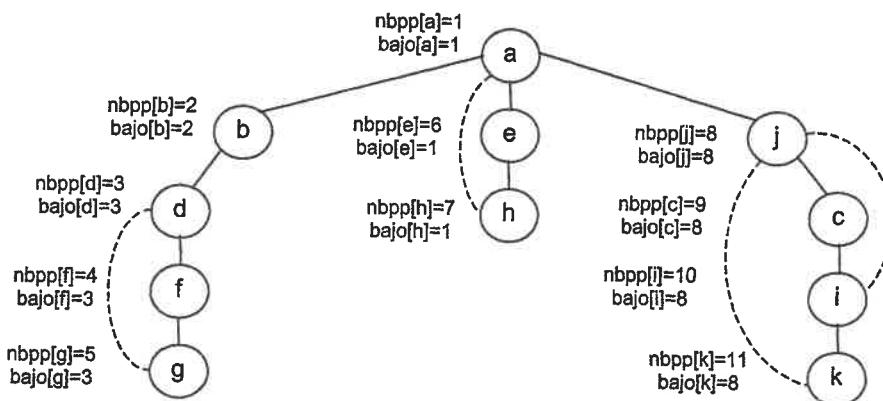


Figura 5.48: Obtención de los puntos de articulación del grafo de la figura 5.47.

Una vez con este resultado, aplicamos las dos condiciones del algoritmo para encontrar los puntos de articulación:

- La raíz, a, es un punto de articulación porque tiene 3 hijos.
- El nodo b es un punto de articulación porque tiene un hijo, d, con $bajo[d] = 3 \geq 2 = nbpp[b]$.
- El nodo d es un punto de articulación porque tiene un hijo, f, con $bajo[f] = 3 \geq 3 = nbpp[d]$.

- El nodo j es un punto de articulación porque tiene un hijo, c, con $bajo[c] = 8 \geq 8 = nbpp[j]$.

Para hacer que dejen de haber puntos críticos, debemos añadir aristas de forma que se eliminan los puntos de articulación. Si nos fijamos en la figura 5.48, la eliminación de b y de d provoca que f y g queden aislados del resto, aunque comunicados entre sí; la eliminación de j aísla a c, i y k; y de los restantes la eliminación de a desconectaría a h y e de los demás. Estos grupos de nodos están en componentes biconexos distintos. Si añadimos ciertas aristas entre ellos, podemos eliminar los puntos de articulación. Por ejemplo, si añadimos las aristas (g,e) y (h,c), el grafo no tendría puntos de articulación. Por lo tanto, sería un grafo biconexo.

Ejercicio 5.2 Dado un grafo con pesos, dirigido o no dirigido, puede que el camino mínimo entre dos nodos v y w no sea único, sino que existan varios. Modificar los algoritmos de Dijkstra y de Floyd para que, además de calcular los caminos mínimos, también calculen el número de caminos mínimos distintos existente entre dos nodos.

Solución.

Para calcular el número de caminos mínimos distintos, en el algoritmo de Dijkstra usaremos un array NC : **array [2..n] de entero**. Inicialmente, los caminos especiales del algoritmo de Dijkstra son los caminos directos, así que sólo existe un camino mínimo. Por lo tanto, NC se inicializaría a 1.

```

para  $i := 2, \dots, n$  hacer
   $NC[i] := 1$ 
   $D[i] := C[1, i]$ 
  ...
finpara

```

Por otro lado, tenemos la actualización del algoritmo de Dijkstra. Cuando añadidos un nuevo nodo v , comprobamos todos los w adyacentes a v . Tenemos tres posibilidades:

- Si $D[w] < (D[v] + C[v, w])$, el camino mínimo de w no se modifica.
- Si $D[w] = (D[v] + C[v, w])$, tenemos varios caminos mínimos más para w . ¿Cuántos más? Pues tantos como hayan para v . Es decir: $NC[w] := NC[w] + NC[v]$.
- Si $D[w] > (D[v] + C[v, w])$, existe un nuevo camino mínimo más para w . Habrán tantos caminos como hayan para v . Es decir: $NC[w] := NC[v]$.

Por lo tanto, el algoritmo de Dijkstra modificado diría:

```

para  $i := 2, \dots, n$  hacer
   $v :=$  vértice con  $Miembro(T, v) =$  verdadero y  $D[v]$  mínimo
  Suprime( $T, v$ )
  para cada nodo  $w$  adyacente a  $v$  hacer
    si  $Miembro(T, w)$  entonces
      si  $(D[v] + C[v, w]) < D[w]$  entonces
         $D[w] := D[v] + C[v, w]$ 
         $P[w] := v$ 
    
```

```

 $NC[w]:= NC[v]$ 
sino si ( $D[v] + C[v, w] = D[w]$ ) entonces
 $NC[w]:= NC[w] + NC[v]$ 
finsi
finsi
finpara
finpara

```

En cuanto al algoritmo de Floyd, en este caso el resultado debería ser una matriz NC : array [1..n, 1..n] de entero. Igual que antes, la inicialización sería $NC[v, w]:= 1$, para todo v y w . Dentro del paso de pivotaje sobre un k , si el camino pasando por k tiene menor coste que $D[i, j]$, entonces tenemos los caminos mínimos existentes pasando por k . Estos caminos serán tantos como: $NC[i, k]*NC[k, j]$. Si el camino pasando por k tiene igual coste que $D[i, j]$, entonces debemos añadir esos caminos mínimos a $NC[i, k]$. Por lo tanto, el bucle principal del algoritmo de Floyd modificado sería el siguiente:

```

para  $k:= 1, \dots, n$  hacer
    para  $i:= 1, \dots, n$  hacer
        para  $j:= 1, \dots, n$  hacer
            si  $D[i, j] < D[i, k]+D[k, j]$  entonces
                 $D[i, j]:= D[i, k]+D[k, j]$ 
                 $P[i, j]:= k$ 
                 $NC[i, j]:= NC[i, k]*NC[k, j]$ 
            sino si  $D[i, j] = D[i, k]+D[k, j]$  entonces
                 $NC[i, j]:= NC[i, j] + NC[i, k]*NC[k, j]$ 
            finsi
        finpara
    finpara
finpara

```

Ejercicio 5.3 En la figura 5.49 se representa un plan de estudios. Los nodos indican las asignaturas del plan. Las aristas indican los prerequisitos para realizar ciertas asignaturas. Es decir, una arista $\langle v, w \rangle$ indica que para matricularse en w hay que tener aprobada v . Encontrar un posible orden secuencial para realizar las asignaturas del plan. El orden obtenido ¿es único?

Solución.

Por las características del problema, tenemos que los grafos usados serán GDA, ya que no pueden existir ciclos. El orden secuencial que respeta los prerequisitos será un orden topológico del grafo. Aplicando el algoritmo estudiado obtenemos el resultado de la tabla 5.5. Ojo, en este caso Array solución no indica el array orden del algoritmo, sino los nodos que se visitan en cada paso del recorrido topológico.

Podemos deducir que orden obtenido no es único si en algún paso de la ejecución la cola tiene más de un vértice. Para este ejemplo ocurre en numerosos pasos, como se puede ver en la tabla 5.5. Por lo tanto, la ordenación topológica obtenida no es única. Otra posible ordenación topológica podría ser: 1, 11, 12, 3, 4, 2, 8, 13, 5, 9, 6, 10, 7.

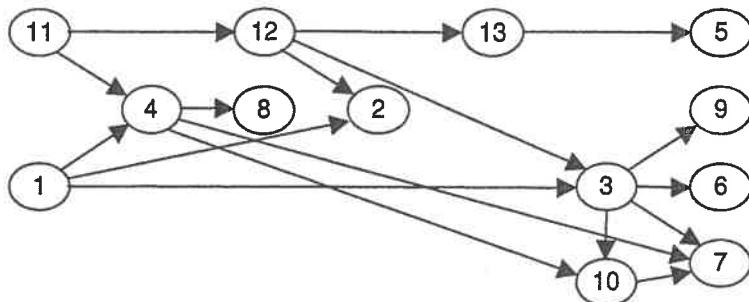


Figura 5.49: Grafo de prerequisitos de un plan de estudios del 5.3.

Array de grados de entrada													Array solución													Cola														
1	2	3	4	5	6	7	8	9	10	11	12	13	1	2	3	4	5	6	7	8	9	10	11	12	13	1	11													
0	2	2	2	1	1	3	1	1	2	0	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	1	11													
0	1	1	1	1	1	3	1	1	2	0	1	1	1	-	-	-	-	-	-	-	-	-	-	-	-	11														
0	1	1	0	1	1	3	1	1	2	0	0	1	1	1	-	-	-	-	-	-	-	-	-	-	-	1	11													
0	1	1	0	1	1	2	0	1	1	0	0	1	1	1	-	-	-	-	-	-	-	-	-	-	-	11	4													
0	0	0	0	1	1	2	0	1	1	0	0	0	1	1	1	-	-	-	-	-	-	-	-	-	-	11	4	12												
0	0	0	0	1	1	2	0	1	1	0	0	0	0	1	1	1	-	-	-	-	-	-	-	-	-	11	4	12	8											
0	0	0	0	1	1	2	0	1	1	0	0	0	0	1	1	1	1	-	-	-	-	-	-	-	-	11	4	12	8	2										
0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	-	-	-	-	-	-	-	11	4	12	8	2	3									
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	-	-	-	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	-	-	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	-	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	-	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	-	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	-	11	4	12	8	2	3	13								
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	-	11	4	12	8	2	3	13							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	-	11	4	12	8	2	3	13						

Tabla 5.5: Obtención de la ordenación topológica del grafo de la figura 5.49.

Ejercicio 5.4 En cierto país petrolífero se supone la existencia de armas de destrucción masiva. La ONU ha decidido planificar una serie de inspecciones exhaustivas, para analizar todos los posibles almacenes de armas. Tenemos un mapa T de localizaciones, como el mostrado en la figura 5.50, con n sitios sospechosos y los caminos existentes entre ellos. En el mapa, $T[i, j] = T[j, i]$ valdrá 1 si existe un camino entre i y j , y 0 en caso contrario. Existen varios equipos de inspectores, que parten inicialmente del mismo sitio. Cada equipo visita un sitio durante un día y se desplaza por la tarde (es decir, en todos los caminos tardamos un tiempo unitario).

Hay que tener en cuenta que las armas se pueden mover de sitio, para evitar ser descubiertas. Las armas se mueven por los mismos caminos que los inspectores, aunque lo hacen por la noche (es decir, no se pueden cruzar por el camino). Para garantizar que el plan de inspecciones sea efectivo debemos garantizar lo siguiente: 1) todos los sitios son inspeccionados, 2) hay que evitar que se puedan mover armas a un sitio que ya ha sido inspeccionado; para ello, debemos garantizar que cualquier posible camino entre un sitio inspeccionado y uno no inspeccionado pase por un sitio que esté siendo inspeccionado.

Diseñar dos algoritmos para resolver los siguientes problemas:

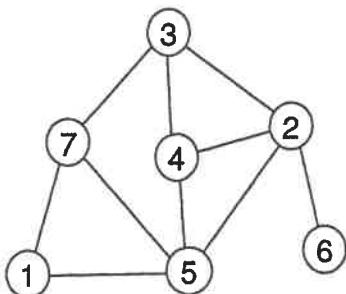


Figura 5.50: Mapa de localizaciones sospechosas del ejercicio 5.4.

- a) Suponiendo que todos los equipos de inspectores parte del sitio I , encontrar cuánto tiempo tardaría el plan de inspecciones como mínimo, qué sitios se deben visitar cada día, cuántos equipos de inspectores necesitamos en total y por dónde se debe mover cada equipo. Ejecutar sobre el grafo de la figura 5.50 con $I = 1$.
- b) Suponiendo que queremos minimizar el tiempo total del plan de inspecciones, encontrar desde qué sitio deben partir las inspecciones, qué sitios se deben visitar cada día y cuántos equipos de inspectores necesitamos.

Solución.

Vamos a dar algunas indicaciones para la resolución de estos problemas, pero sin llegar a escribir los algoritmos. El problema a) se puede plantear de distintas maneras. Por ejemplo, si usamos una búsqueda primero en anchura, el primer día inspeccionamos la raíz del árbol, el segundo día los nodos que están un nivel debajo de la raíz, el tercero los nodos que están a nivel 3, y así sucesivamente. El tiempo que se tarda en el plan sería igual a la profundidad máxima del árbol. Los sitios que visitan cada día i serían los nodos de nivel i . El número de inspectores que se necesitan son el número de hojas del árbol abarcador en anchura. Y los movimientos de cada equipo de inspectores son los caminos desde la raíz hasta esa hoja. Por ejemplo, en el grafo de la figura 5.50 se tardaría 4 días. El día 1 se visita el nodo 1, el día 2 los nodos 7 y 5, el día 3 los nodos 2, 3 y 4, y el día 4 el nodo 6.

El problema también se puede plantear como un problema de caminos mínimos, donde todas las aristas tienen coste 1. Si calculamos el camino mínimo desde el nodo inicial I hasta todos los demás (por ejemplo, con el algoritmo de Dijkstra), el primer día deberíamos visitar la raíz, el segundo los nodos que están a distancia 1, el tercero todos los que están a distancia 2, y así sucesivamente. El tiempo total que se tardaría sería la distancia máxima de los caminos mínimos, más 1. El número de inspectores sería el número de nodos que no aparecen como nodos intermedios de algún camino mínimo. Esta solución garantiza las restricciones del problema: si estamos en un día k cualquiera, para pasar de un nodo no inspeccionado (con distancia mínima mayor que k) a uno ya inspeccionado (con distancia mínima menor que k) se debe pasar por alguno que esté siendo inspeccionado (con distancia mínima k).

Según lo anterior, para resolver el problema b) simplemente deberíamos calcular para cada nodo la distancia máxima de los caminos mínimos desde ese nodo hasta todos

los demás. Esto es lo que se conoce como la **excentricidad** de un nodo en un grafo. Las inspecciones deben comenzar desde el nodo más central, es decir el que tenga excentricidad mínima. La solución se puede programar fácilmente usando el algoritmo de Floyd. Después tomamos máximos por filas (o por columnas, en este caso es equivalente) de la matriz de caminos mínimos, y nos quedamos con el nodo con menor valor máximo. El número de inspectores que se necesitan un día k son los nodos que estén a distancia $k - 1$ del nodo central. Una posible cota inferior para el número de inspectores necesario es el máximo número de inspectores que se necesitan para todos los días.

Ejercicio 5.5 La experimentación con ratones es básica para crear nuevos fármacos, que resultan en importantes mejoras en la calidad de vida de los roedores. En un experimento con fármacos cerebrales, se colocan ratones dentro de un laberinto formado por n celdas, m salidas del laberinto y varios pasadizos entre las celdas o las salidas, como el mostrado en la figura 5.51. Las celdas son numeradas de 1 a n y las salidas de $n + 1$ a $n + m$. En cada pasadizo los ratones tardan un tiempo dado, que llamaremos $P[i, j]$: tiempo de ir de la celda i a la celda o salida j . Se supone que el tiempo que tardan en las celdas es despreciable.

El experimento consiste en colocar inicialmente un ratón en cada celda, y contar el número de ratones que son capaces de salir del laberinto en un tiempo dado t_{max} . Suponiendo que los ratones eligen siempre el camino más corto hacia alguna de las salidas, las preguntas a resolver son: i) ¿Cuántos ratones habrán salido en el instante dado t_{max} ? ii) ¿Cuánto tiempo se necesita para conseguir que todos los ratones hayan salido?

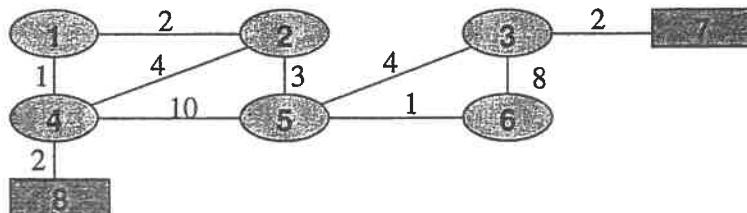


Figura 5.51: Laberinto del ejercicio 5.5.

Diseñar un algoritmo para resolver los dos problemas anteriores. Se supone que el instante inicial es $t = 1$. Hacer una estimación aproximada del orden de complejidad del algoritmo.

Solución.

Claramente, el ejercicio es un problema de caminos mínimos en un grafo. Se puede resolver fácilmente con el algoritmo de Dijkstra. El valor de i) será el número de celdas para las cuales la distancia mínima a alguna de las salidas sea menor o igual que t_{max} . El valor de ii) será el máximo de las distancias mínimas de cada celda a alguna de las salidas.

Para simplificar la implementación, utilizaremos el algoritmo de Floyd, calculando todos los caminos mínimos entre todas las celdas y salidas. El algoritmo sería el siguiente.

operación Ratones ($n, m, tmax$: entero; P : array [1.. $n+m$, 1.. $n+m$] de entero; **var** RatonesQueSalen, MaximoTotal: entero)

variables C : array [1.. $n+m$, 1.. $n+m$] de entero

```

 $C := P$ 
para  $k := 1, \dots, n+m$  hacer
    para  $i := 1, \dots, n+m$  hacer
        para  $j := 1, \dots, n+m$  hacer
             $C[i, j] := \min(C[i, j], C[i, k] + C[k, j])$ 
        finpara
    finpara
finpara
MaximoTotal := 0
para  $i := 1, \dots, n$  hacer
    TiempoMinimo :=  $C[i, n+1]$ 
    para  $j := n+2, \dots, n+m$  hacer
        TiempoMinimo :=  $\min(TiempoMinimo, C[i, j])$ 
    finpara
    si  $TiempoMinimo \leq tmax$  entonces
        RatonesQueSalen := RatonesQueSalen + 1
    finsi
    MaximoTotal :=  $\max(MaximoTotal, TiempoMinimo)$ 
finpara
```

Para el laberinto de la figura 5.51 tenemos el resultado final: RatonesQueSalen=4, MaximoTotal= 7. Se puede ver fácilmente que el tiempo del algoritmo pertenece a $\Theta((n+m)^3)$.

Ejercicios propuestos

Ejercicio 5.6 Implementar la prueba de aciclicidad en un grafo no dirigido de manera eficiente. ¿Se puede conseguir en un tiempo $O(n)$? Modificar el procedimiento para que, en caso de encontrar que existe un ciclo, devuelva en una lista o en un array los elementos que forman el ciclo encontrado.

Ejercicio 5.7 Suponer el grafo no dirigido de siguiente figura 5.52.

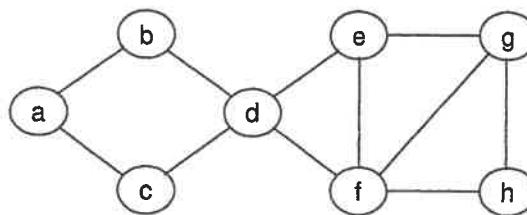


Figura 5.52: Grafo no dirigido del ejercicio 5.7.

Se pide resolver las siguientes cuestiones sobre el mismo:

- a) El bosque de expansión en profundidad, empezando en a y en d.
- b) El bosque de expansión en amplitud, empezando en a y en d.
- c) El árbol de expansión de coste mínimo utilizando el algoritmo de Prim.
- d) El árbol de expansión de coste mínimo utilizando el algoritmo de Kruskal. ¿Son iguales las soluciones obtenidas en ambos algoritmos? En caso contrario, ¿son válidas las distintas soluciones? ¿Por qué?
- e) Encontrar los puntos de articulación y los componentes biconexos. ¿Cuál es la conectividad del grafo?

Ejercicio 5.8 Demostrar que el algoritmo de Dijkstra no funciona cuando las aristas pueden tener coste negativo, aun cuando no existan ciclos en el grafo. Sugerencia: dar un contraejemplo de un grafo dirigido sin ciclos en el que el algoritmo de Dijkstra no encuentre el resultado correcto.

Ejercicio 5.9 Modificar el algoritmo de Dijkstra para que, además de calcular los caminos mínimos del vértice origen al resto, también calcule los siguientes valores:

- Un array L de tamaño n , indicando el número de aristas por las que pasa cada uno de los caminos mínimos. En caso de empate de coste en los caminos mínimos, el algoritmo deberá quedarse con el camino que pase por menos aristas.
- Un array U de booleanos, en el que se indique para cada nodo si su camino mínimo desde el origen es único o no.

Ejercicio 5.10 Considerar el grafo dirigido de la figura 5.53. Resolver las siguientes cuestiones:

- a) Aplicar el algoritmo de Dijkstra para encontrar los caminos más cortos que van desde el nodo a hasta los restantes nodos. Mostrar los valores T , D y P para todos los pasos de ejecución del algoritmo.
- b) Utilizando el resultado del apartado anterior, encontrar la secuencia de nodos del camino mínimo de a a d, y el de a a e.
- c) Aplicar el algoritmo de obtención de los componentes fuertemente conexos. A partir del resultado obtenido, mostrar el grafo reducido correspondiente.
- d) Mostrar el resultado de la aplicación del algoritmo de Floyd. Obtener el nodo centro del grafo.

Ejercicio 5.11 Diseñar un algoritmo para contar el número de ciclos simples existentes en un grafo no dirigido. ¿Cuál es el orden de complejidad de este algoritmo? ¿Cuántos ciclos pueden haber como máximo en un grafo cualquiera?

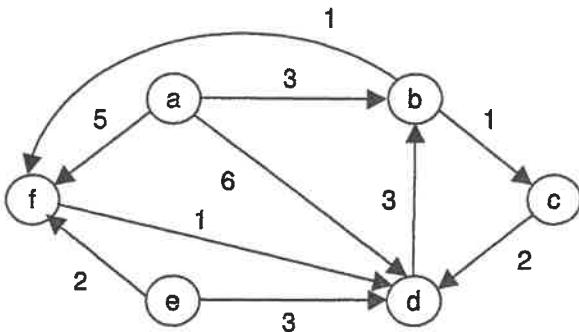


Figura 5.53: Grafo dirigido del ejercicio 5.10.

Ejercicio 5.12 Proporcionar un algoritmo con tiempo $O(n + a)$ para comprobar si un grafo es bipartido o no. Aplicarlo sobre los grafos de las figuras 5.52 y 5.54. Sugerencia: utilizar la búsqueda primero en profundidad.

Ejercicio 5.13 Una raíz de un GDA es un vértice r tal que todos los nodos del grafo pueden alcanzarse desde r , es decir, existen caminos entre r y el resto de nodos. Escribir un procedimiento para determinar si un GDA posee una raíz y, en ese caso, encuentre cuál es.

Ejercicio 5.14 Aplicar el algoritmo de Kruskal sobre el grafo de la figura 5.54, mostrando el orden en que son añadidas las aristas a la solución.

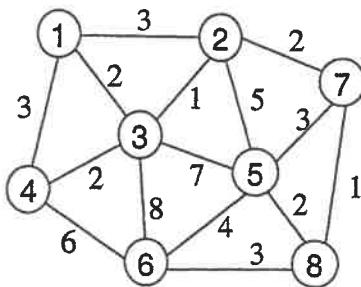


Figura 5.54: Grafo no dirigido con pesos del ejercicio 5.14.

Si aplicáramos el algoritmo de Prim, ¿podemos asegurar que se obtendría siempre la misma solución? ¿Podemos asegurar que el coste de la solución sería el mismo? ¿Por qué? Comprobar las respuestas mostrando la ejecución del algoritmo de Prim sobre el mismo grafo.

Ejercicio 5.15 Para desarrollar la especificación formal del TAD `GrafoNDE[T]`, de los grafos no dirigidos y etiquetados con etiquetas de tipo T , disponemos de los siguientes conjuntos:

G Conjunto de grafos dirigidos con pesos

V Conjunto de vértices

T Conjunto de posibles etiquetas de las aristas

N Conjunto de naturales

B Conjunto de booleanos

U Conjunto de mensajes de error { "Vértice no existente", "Arista no existente" }

No se define el conjunto de aristas puesto que es implícito (una arista es un par de vértices más una etiqueta). Escribir la sintaxis y la semántica de la especificación algebraica del tipo, incluyendo al menos las siguientes operaciones: GrafoVacío, InsertaNodo, InsertaArista, ExisteArista, EtiquetaArista (para consultar la etiqueta de una arista), NumNodos, NumAristas, BorraArista.

Ejercicio 5.16 Se define el radio de un árbol como la distancia máxima de la raíz a cualquiera de los nodos hoja. Dado un grafo no dirigido y conexo, escribir un algoritmo para encontrar un árbol de expansión de radio mínimo.

Ejercicio 5.17 Construir un algoritmo para evaluar expresiones aritméticas representadas mediante GDA. Suponer que tenemos almacenada una etiqueta para cada nodo del grafo $ETIQ[1..n]$ con los valores (+, -, *, A, B, ..., Z). Especifica la estructura de representación del grafo que consideres más oportuna, con las operaciones adecuadas. Los valores para las variables son obtenidos llamando a un función ValorVariable (car: carácter): real. Sugerencia: Almacenar el valor de cada nodo en un array $VALOR[1..n]$ de reales.

Ejercicio 5.18 Un pozo en un grafo dirigido es un vértice al cual llegan aristas de todos los vértices restantes pero del cual no parte ninguna arista. Escribir un algoritmo que determine la existencia de pozos y, en su caso, obtenga cuáles son en un orden $O(n)$.

Ejercicio 5.19 Suponer el grafo dirigido acíclico de la figura 5.55. Resolver las siguientes cuestiones sobre el mismo:

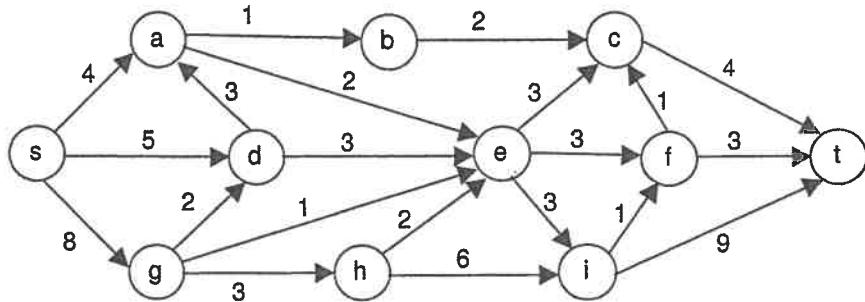


Figura 5.55: Grafo dirigido acíclico con pesos del ejercicio 5.19.

- Encontrar una ordenación topológica del grafo. Mostrar los pasos de ejecución del algoritmo. ¿Es única esta ordenación o existen otras ordenaciones válidas?
- Aplicar el algoritmo de flujo máximo, obteniendo el grafo G_F de flujos resultantes. Aplicar el algoritmo que permite deshacer caminos. ¿Cuánto vale el flujo máximo total?

- c) Encontrar los componentes fuertemente conexos del grafo. A partir de ellos mostrar el grafo reducido correspondiente. A la vista del resultado, ¿podemos establecer alguna relación general entre un GDA y su grafo reducido? ¿Cuál? ¿Por qué?

Ejercicio 5.20 En un problema de planificación de tareas utilizamos GDA, donde los nodos contienen tareas a realizar y las aristas indican la precedencia de tareas. Las aristas son etiquetadas con un coste, que indica el tiempo necesario para acabar la tarea que es cabeza de la arista. Existen dos nodos destacados, *Inicio* y *Fin*, para indicar el principio y el final del plan, respectivamente.

Puesto que todas las tareas se deben realizar necesariamente para acabar el plan, el problema fundamental consiste en calcular la longitud del camino más largo entre *Inicio* y *Fin*, que será el tiempo mínimo en ejecutar el plan.

- Comprobar que una simple modificación del algoritmo de Dijkstra (buscando máximos en lugar de mínimos) no resuelve el problema de calcular el camino más largo entre dos nodos.
- Proponer un algoritmo para calcular la longitud del camino más largo entre los nodos *Inicio* y *Fin*, teniendo en cuenta que trabajamos con GDA. ¿Cuál es el orden de complejidad del algoritmo? Sugerencia: tener en cuenta el orden topológico.
- Aplicar el algoritmo diseñado sobre el grafo de la figura 5.55.

Ejercicio 5.21 Implementar el algoritmo para la búsqueda de puntos de articulación en un grafo no dirigido, modificando las partes adecuadas del procedimiento bpp de la búsqueda primero en profundidad. ¿Cuál es el orden de complejidad del algoritmo?

Cuestiones de autoevaluación

Ejercicio 5.22 Indicar cuál es el número máximo de arcos que puede tener: a) un grafo no dirigido, b) un grafo dirigido, c) un grafo no dirigido sin ciclos, d) un grafo dirigido acíclico (GDA).

Ejercicio 5.23 Dado un árbol de expansión resultante de un recorrido sobre un grafo no dirigido, ¿qué tipo de arcos (aparte de los del árbol) pueden aparecer si el recorrido es una búsqueda en profundidad o una búsqueda en anchura? ¿Qué arcos aparecerán si el recorrido (en profundidad o en anchura) es aplicado sobre un grafo dirigido? Por ejemplo, demostrar que en un bosque de expansión en profundidad de un grafo dirigido, todos los arcos de cruce van de derecha a izquierda.

Ejercicio 5.24 El árbol de expansión en profundidad o en anchura de un grafo ¿es único o existen varias posibilidades? ¿En función de qué surgen las distintas posibilidades?

Ejercicio 5.25 ¿Cómo se podría modificar el procedimiento bpa –utilizado en la búsqueda primero en anchura– para realizar un recorrido en profundidad, realizando un cambio mínimo en el algoritmo? Sugerencia: considera la estructura de datos utilizada.

Ejercicio 5.26 ¿Cuántos árboles de expansión tiene un grafo no dirigido completo de n nodos?

Ejercicio 5.27 Demostrar que el grafo reducido de un grafo dirigido G cualquiera –el que representa cada componente fuertemente conexo de G con un nodo– es siempre un GDA.

Ejercicio 5.28 Suponer que estamos trabajando con GDA. ¿Es posible mejorar la eficiencia obtenida con el algoritmo de Dijkstra para este tipo de grafos? ¿Cómo sería la modificación de este algoritmo para conseguir la mejora? Pista: hacer uso de la ordenación topológica para seleccionar los nodos entre los candidatos.

Ejercicio 5.29 Para los siguientes tipos de grafos decir si son biconexos o no, y cuál es su conectividad: a) grafo con estructura de anillo; b) grafo con estructura de árbol; c) grafo completo con n nodos.

Ejercicio 5.30 Demostrar que todo árbol es un grafo bipartido.

Ejercicio 5.31 Sea G un grafo no dirigido y conexo con n vértices. Demostrar que G debe tener al menos $n - 1$ arcos y que todos los grafos no dirigidos conexos con $n - 1$ arcos son árboles.

Ejercicio 5.32 En una aplicación que usa grafos etiquetados, queremos permitir que pueda existir más de una arista entre dos nodos (v, w). Tendríamos realmente lo que hemos denominado *multigrafos*. ¿Cuál de las estructuras de representación de grafos se adapta más fácilmente a esta modificación? Justificar la respuesta brevemente.

Ejercicio 5.33 ¿Cuáles son las claves de la demostración del funcionamiento óptimo del algoritmo de Dijkstra? Es decir, ¿cómo se puede demostrar que siempre encuentra la solución óptima? ¿Y para el algoritmo de Floyd?

Ejercicio 5.34 En el algoritmo para calcular los puntos de articulación de un grafo no dirigido, puesto que la condición del punto 4 no se puede cumplir para los nodos hoja del árbol abarcador en profundidad, se puede concluir que las hojas nunca serán puntos de articulación. Demostrar que esta conclusión es cierta, sin usar las propiedades del algoritmo comentado.

Ejercicio 5.35 Mostrar un ejemplo de grafo no dirigido con conectividad 3 y que tenga cinco nodos o más. ¿Cuántos nodos hay que eliminar para desconectar el grafo?

Referencias bibliográficas

La mayoría de los problemas y algoritmos sobre grafos estudiados en este capítulo se pueden encontrar en [Aho88], tratados con un nivel de detalle similar. El estudio sobre grafos es descompuesto en grafos dirigidos, capítulo 6, y no dirigidos, capítulo 7, aunque la mayoría de los problemas se pueden aplicar sobre ambos tipos. También dentro de la

bibliografía básica se encuentra [Baase00], aunque en este caso los problemas sobre grafos no son estudiados de forma separada sino a lo largo de todo el libro.

Para profundizar en otros problemas sobre grafos se puede utilizar bibliografía adicional. Por ejemplo, en los capítulos del 22 al 26 de [Cormen90] se tratan los grafos de forma exhaustiva. El capítulo 26 está dedicado a los problemas de flujo en redes. En el capítulo 8 de [Drozdek01] se tratan algunos otros problemas, como el isomorfismo de grafos (en inglés *matching*) y los ciclos hamiltonianos y de Euler.

Apéndices

Apéndice A

Tutorial de ANSI C

Contenido del capítulo:

A.1. Introducción	277
A.2. Primeros pasos	279
A.3. Tipos de datos elementales y variables	280
A.3.1. Tipo char	281
A.3.2. Tipo int	281
A.3.3. Tipos enumerados	282
A.3.4. Tipos float y double	282
A.3.5. Las conversiones de tipo	282
A.3.6. Modos de almacenamiento de las variables	283
A.4. Operadores	285
A.4.1. Aritméticos	285
A.4.2. Incrementales	285
A.4.3. Relacionales	286
A.4.4. Lógicos	286
A.4.5. De manejo de bits	287
A.4.6. De asignación	287
A.4.7. De dirección e indirección	287
A.4.8. Otros operadores	288
A.4.9. Expresiones: precedencia y asociatividad	288
A.5. Sentencias de control de flujo	290
A.5.1. Sentencias de selección	290
A.5.2. Sentencias de iteración	293
A.6. Tipos de datos agregados y punteros	295
A.6.1. Punteros	296
A.6.2. Arrays	297
A.6.3. Relación entre arrays y punteros	299
A.6.4. Estructuras y uniones	301
A.6.5. Definición de tipos	303
A.7. Funciones	303

A.7.1. Función main con argumentos	305
A.7.2. Funciones como parámetros	305
A.8. Funciones de E/S	306
A.8.1. printf	306
A.8.2. scanf	307
A.8.3. putchar y getchar	308
A.8.4. Manejo de ficheros	308
A.9. Gestión dinámica de la memoria	310
A.10. El preprocesador de C	312
A.11. Programas con varios ficheros	313
Ejercicios Propuestos	315
Referencias bibliográficas	317

A.1. Introducción

En este tutorial se introduce al alumno al manejo del lenguaje C de cara a la programación de las prácticas de la asignatura *Algoritmos y Estructuras de Datos*. Se asume la posesión de nociones de programación previas por parte del alumno, así como el manejo de algún lenguaje de programación estructurada como *Pascal* (conocimientos adquiridos en las asignaturas de programación de los primeros cursos de la carrera). De este modo no se explicarán conceptos básicos de programación como el de variable, flujo de ejecución, función, puntero, paso de parámetros, etc., sino que se asumirá su conocimiento y se detallará su implementación y uso en el caso particular de C.

El lenguaje explicado aquí es *ANSI C*, es decir, el estándar de C definido por el *ANSI* (siglas en inglés del Instituto Nacional Americano de Estándares) y posteriormente aceptado por la *ISO* (Organización Internacional para la Estandarización). Se trata, por tanto, de un lenguaje estandarizado, ampliamente utilizado y para el que existen compiladores en casi cualquier plataforma. Dado que el presente tutorial pretende ser ante todo práctico, referiremos todos los ejemplos a una plataforma concreta, la que el alumno tendrá disponible en las salas de prácticas: compilador *gcc* para *Linux* sobre *PC*.

Antes de empezar a dar los primeros pasos con el lenguaje, comentaremos algunas características generales que nos pueden dar una primera idea sobre su funcionamiento:

- Se trata de un lenguaje estructurado. Encontramos en él instrucciones de control de flujo como iteradores de distintos tipos (*while*, *do-while* y *for*), sentencias condicionales (*if-else*) o de selección de casos (*switch*), así como las abstracciones de control *procedimiento* y *función*. En realidad en C formalmente no existen los procedimientos como tales sino que se implementan como funciones que no devuelven nada, lo cual se puede indicar con *void*, según se verá más adelante.
- Al contrario que en lenguajes como *Pascal*, no existen las funciones anidadas. Es decir, no se puede declarar una función local a (dentro de) otra función. Todas las funciones se encuentran al mismo nivel. En cambio sí existen los bloques de instrucciones anidados, donde podemos definir variables locales al bloque. Profundizaremos en el tema más adelante.
- El lenguaje proporciona algunos tipos básicos de datos: enteros y reales de varios tamaños y caracteres; adicionalmente arrays, apuntadores, estructuras y uniones con los que el usuario puede construir tipos compuestos.
- Se trata de un lenguaje de relativo bajo nivel, es decir, no se trata de un lenguaje de bajo nivel, como el ensamblador; pero dentro de los lenguajes de alto nivel tampoco es de los de mayor nivel de abstracción: no hay operaciones que permitan operar de forma transparente y automática con la mayoría de los tipos de datos compuestos, especialmente en lo que se refiere a estructuras dinámicas. Por ejemplo, no existe la operación de asignación entre arrays (ni siquiera para estáticos). El programador se ha de encargar de manejar *a bajo nivel* las distintas estructuras de datos generadas, conociendo perfectamente su disposición en memoria y accediendo directamente a las distintas posiciones que ocupan. Esta filosofía de programación proporciona una

gran potencia al programador, pero al mismo tiempo le supone una complejidad añadida.

- Si a esta forma de manejar las estructuras de datos añadimos el hecho de que el lenguaje – o más bien la forma en que el compilador y el entorno de ejecución lo interpretan –, asume en general que el programador sabe lo que está haciendo y le deja hacer prácticamente cualquier cosa, el resultado que obtenemos es un lenguaje sumamente potente pero al mismo tiempo susceptible a los errores de programación. Por ejemplo no se realiza comprobación dinámica de respeto de los rangos de un array: si escribimos una posición más allá del límite declarado para un array, nadie nos dirá nada. El compilador no sabía nada en su momento (se trata de un error en tiempo de ejecución). Y el entorno de ejecución no se preocupa de ese tipo de cosas, simplemente nos deja hacer. De ese modo probablemente estaremos escribiendo en la variable declarada tras el array, supuesto que la hubiese. Esto podría producir, por ejemplo, un error lógico muy difícilmente detectable. Este tipo de problemas se suelen dar especialmente con el manejo de punteros y la asignación de memoria dinámica. Haremos hincapié en este tema en la sección A.9 (página 310).
- En la mayoría de los lenguajes se distingue entre los conceptos de *expresión* y *sentencia*. Una expresión es algo que al ser evaluado corresponde a un valor. Por ejemplo `2+2` es una expresión que al ser evaluada corresponde al valor 4. Una sentencia es una orden que indica que se ha de llevar a cabo una determinada acción. La asignación es un ejemplo de sentencia: se indica que una variable ha de recibir como valor el resultado de la evaluación de una expresión. En C se expresa mediante el símbolo `'='`: `variable = expresion`, por ejemplo, `a=2`.

En C cualquier expresión puede ser utilizada como sentencia, es decir, puedo invocar la obtención de su valor sin utilizarlo en ninguna sentencia. Por ejemplo, puedo escribir en mitad de un programa la expresión `2+2` sin asignarla a ninguna variable. Este ejemplo no parece tener mucho sentido. Pero sí que lo tiene el que podamos invocar del mismo modo a una función de la cual solo queramos, en un momento dado, utilizar sus efectos colaterales, pero sin que nos interese su valor devuelto. Por ejemplo, imaginemos una función `leer_car` que lee un carácter de un fichero y lo devuelve como resultado, al tiempo que avanza el cursor de lectura de ese fichero. Con `c=lee_car(fich)` almacenamos el carácter leído, pero también podemos utilizar `lee_car(fich)` para obtener el efecto colateral de avance del cursor, ignorando el carácter leído.

En C la sentencia de asignación es también una expresión, es decir, tiene asociado un valor. Se trata del valor asignado a la variable de su parte izquierda. Esto permite escribir cosas como `a=b=c=0`. La interpretación de esta sentencia es `a=(b=0)`, es decir, se ejecuta la sentencia `b=0` que corresponde al valor 0 y a continuación se ejecuta `a=0` (que corresponde de nuevo al valor 0, pero en esta ocasión el valor de la expresión no es utilizado, sino que estamos utilizando una expresión como sentencia).

A.2. Primeros pasos

En este tutorial iremos aprendiendo el lenguaje C paso a paso a mediante su puesta en práctica. Así que empecemos por el principio. Cualquier programa en C está compuesto por funciones. En todo programa siempre existe, al menos, la función principal, denominada `main`, que es la primera en ser ejecutada al ser invocado el programa. Según lo dicho, el programa mínimo en C sería el siguiente:

```
main(){}1
```

Se trata de un programa que sólo contiene la función principal, que en este caso no tiene argumentos (se enumerarían entre los paréntesis vacíos) y devuelve (por defecto, al no especificar otra cosa) un entero. `main` se puede definir de muchas otras formas; de hecho, es habitual que tenga argumentos de entrada en los que recibir los valores introducidos por el usuario en la línea de comandos (explicaremos cómo hacer esto en la sección A.7.1, página 305). Lo que hay entre las llaves es el cuerpo de la función, es decir, las sentencias que se ejecutan cuando se invoca. En este caso está vacío, de modo que la función no hace nada. No parece un programa muy interesante, pero ya es un programa. Para probarlo vamos a utilizarlo como ejemplo para una primera explicación del compilador que vamos a utilizar, el `gcc`. A partir de ahora insertaremos el símbolo \otimes para indicar que debes probar sobre el ordenador lo que se está explicando. \otimes Editamos un fichero de texto que contenga únicamente la línea de código anterior y lo salvamos con nombre `prog01.c`. Si escribimos en la línea de comandos (donde el símbolo `>>` representará el cursor a partir de ahora):

```
>> gcc prog01.c
```

no obtendremos ningún mensaje. Eso quiere decir que la compilación fue bien – aparecerá en el directorio actual un nuevo fichero, `a.out`. Se trata del ejecutable correspondiente a nuestro programa. Si queremos que el ejecutable se cree directamente con un nombre distinto a este usaremos la opción `-o` seguida del nombre deseado. \otimes Por ejemplo, el comando

```
>> gcc -o prog01 prog01.c
```

creará esta vez un ejecutable llamado `prog01`.

El cuerpo de una función es un caso particular de **bloque**: un bloque es una secuencia de sentencias que se irán ejecutando una tras otra en el orden en que están escritas. Los bloques se delimitan mediante llaves (`{}`). Vamos a hacer un nuevo programa en que el cuerpo de la función `main` tenga alguna sentencia. Para ello vamos a utilizar la función `printf`¹ sobre la cual solo comentaremos por el momento que imprime en la salida estándar (en principio la pantalla) la cadena de caracteres que recibe como parámetro. Sobre las cadenas de caracteres solo comentaremos por el momento que se escriben entre comillas dobles ("cadena") y que los caracteres especiales se escriben precedidos por la barra invertida (\), como `\n`, que es el retorno de carro. También hay que comentar aquí que `printf` no es una función básica del lenguaje sino que se encuentra en una librería, `stdio.h` (entrada/salida estándar), de modo que hay que incluir dicha librería en nuestro programa. Eso se hace mediante la palabra `include`. Se puede observar que

¹Para una explicación más completa de la función, ver sección A.8, página 306.

va precedida del símbolo `#`. Esto es porque se trata de un comando para el *preprocesador*. Hablaremos sobre este tipo de comandos más adelante (en la sección A.10, página 312). Dicho todo esto, veamos el siguiente programa:

```
#include <stdio.h>
main(){
    printf("Esto es la sentencia 1 \n");
    printf("Esto es la sentencia 2 \n");
}
```

⊗ Compila y ejecuta este programa para ver el resultado. Observa que ambas sentencias acaban en un punto y coma (`;`). Igual que existe el bloque de sentencias vacío, existe la sentencia vacía que se reduce a ese punto y coma. Un bloque es también una sentencia, de modo que como comentábamos en la introducción se permite la anidación de bloques. Como ejemplo de ambas cuestiones veamos el siguiente programa. Los comentarios en C se escriben entre los símbolos `/*` y `*/` o bien desde la aparición de `//` hasta fin de línea:

```
main(){
    ;      /* esto es una sentencia vacia */
    {}     /* esto es un bloque vacio */
}
```

⊗ Compila el programa para comprobar que es un programa válido.

A.3. Tipos de datos elementales y variables

Los tipos elementales proporcionados por C son:

- `char`: un byte, pensado para almacenar caracteres. Es considerado como un número entero a todos los efectos.
- `int`: un entero; su tamaño habitual es el entero de la máquina.
- `float`: número real en coma flotante, precisión normal.
- `double`: número real en coma flotante, doble precisión.

Algunos de estos tipos pueden ser *calificados* mediante las palabras `short`, `long`, `signed` y `unsigned`. Los modificadores `short` (corto) y `long` (largo) se aplican a enteros para obtener los tipos `short int` y `long int`, en los que habitualmente se suprime la palabra `int`. La idea es que en estos dos nuevos tipos contamos con más o menos bytes que en el caso de un entero normal. Pero el efecto real depende de la máquina y el compilador utilizados. Lo único que nos garantiza el estándar es que:

- $\text{bits}(\text{short}) \leq \text{bits}(\text{int}) \leq \text{bits}(\text{long})$
- $\text{bits}(\text{short}) \text{ y } \text{bits}(\text{int}) \geq 16$
- $\text{bits}(\text{long}) \geq 32$

`long` también puede ser aplicado a `double` para obtener un número en punto flotante de precisión extendida. El estándar no regula cuál es esa precisión, de modo que no tenemos garantizado obtener una mejor que la de `double`.

Respecto a `signed` y `unsigned`, todo `int` o `char` (y derivados) es por defecto un número cuyo rango de valores se reparte simétricamente entre positivos y negativos. Se reserva un bit para indicar el signo. Pero podemos elegir utilizar este bit de signo para representar el doble de números positivos renunciando a los negativos. La primera posibilidad se especifica anteponiendo al nombre del tipo la palabra `signed, con signo`, mientras que en el segundo caso utilizamos `unsigned, sin signo`. Por ejemplo, un `char` tiene 8 bits. Eso quiere decir que cuando es `con signo` (por defecto) podemos representar valores en el rango $[-2^7, 2^7 - 1]$ mientras que si utilizamos un `unsigned char` podemos hacerlo en el rango $[0, 2^8 - 1]$.

En C, al igual que en Pascal, es necesario declarar cada una de las variables que se vayan a utilizar, de modo que se le pueda asignar un espacio en memoria de acuerdo a su tipo. Una variable no declarada hará que el compilador nos dé un error avisándonos de la omisión. La sintaxis de una declaración es *tipo nombre-variable*. En una misma línea de declaración se pueden declarar varias variables del mismo tipo, al tiempo que cualquiera de ellas puede ser opcionalmente inicializada mediante asignación:

tipo nombre-variable-1=valor-inicial-1, ..., nombre-variable-n=valor-inicial-n;

⊗ En las subsecciones siguientes no se proponen ejemplos de código concretos, pero sería interesante que intentaras definir e inicializar variables de los tipos explicados para irte familiarizando con la sintaxis. Pasamos a estudiar individualmente cada uno de estos tipos.

A.3.1. Tipo `char`

Una variable de tipo `char` se utiliza para representar caracteres, pero como decíamos es también un número entero a todos los efectos. Lo que almacena el byte correspondiente es el código ASCII del carácter representado. Un valor de tipo carácter se representa en el código fuente de C encerrado entre comillas simples ('), por ejemplo, 'a'. Dado que los caracteres son en realidad números, podemos escribir, por ejemplo, la expresión '9'-'0'. Su resultado numérico es 9, el carácter asociado es no imprimible (tabulador horizontal). Podemos utilizar esta circunstancia para, por ejemplo, obtener la letra mayúscula correspondiente a otra minúscula del siguiente modo: `may=min+'A'-'a'`.

A.3.2. Tipo `int`

Todo lo que aquí se diga es válido para las variantes `short` y `long`, así como para los `char` y sus variantes tratados como números. Los valores enteros se pueden escribir en diferentes bases:

- Decimal: sin añadidos, como lo haríamos normalmente, por ejemplo 29, -8, +4.
- Octal: añadiendo un 0 la izquierda del número representado, por ejemplo: 027, -012.
- Hexadecimal: añadiendo el prefijo 0x, por ejemplo: 0xFF00, -0x8A, 0xff.

A.3.3. Tipos enumerados

En realidad no se trata de un nuevo tipo, sino más bien una forma de definir constantes enteras. Una enumeración es una lista de valores enteros constantes a los que se asocia un nombre. La enumeración también recibe un nombre, que podrá ser usado como tipo. Por ejemplo puedo definir el tipo enumerado `estaciones`:

```
enum estaciones {primavera, verano, otonno, invierno};
```

El primer nombre, `primavera`, tiene valor 0, el segundo 1, tercero 2 y cuarto 3. A partir de esta definición puedo usar `estaciones` como tipo y asignar a las variables de ese tipo valores por medio de los nombres:

```
enum estaciones a,b;
a = primavera;
b = invierno;
```

Como los nombres corresponden a valores enteros pueden ser usados como operandos enteros, por ejemplo: `num = primavera*invierno`.

Se pueden especificar valores explícitos para los nombres. Para aquellos para los que no se especifica valor explícito su valor será calculado a partir del último nombre para el que sí se especificó, o a partir de 0 para el primero de la lista si no hay valor explícito previo. Por ejemplo: `enum estaciones {primavera, verano=5, otonno, invierno}`. Tenemos 0 para `primavera`, 5 para `verano`, y 6 y 7 para `otonno` e `invierno`.

Los nombres de valores en enumeraciones diferentes deben ser distintos. Los valores dentro de una misma enumeración no tienen por qué serlo.

A.3.4. Tipos float y double

Los valores en punto flotante se escriben en base diez y son reconocidos porque incluyen el punto decimal o bien la notación científica ($mant10^{exp}$, escrito como: *mant* e *exp*, *mant* por mantisa y *exp* por exponente). Algunos ejemplos de constantes en punto flotante son: 1.0, 0.1, .1, 2e3, 2e-2, -2e-3,... Nótese la diferencia entre 1 y 1.0 ó 1.. El primero es un entero y el segundo un real. Este matiz puede llegar a ser muy importante en situaciones como la siguiente: \otimes ¿cuál es el valor de la expresión $3/2$? ² ¿Y de $3.0/2.0$? En el primer caso estamos dividiendo dos números enteros por lo que estamos aplicando una división entera, la respuesta es 1; en el segundo caso dividimos dos reales mediante división real, el resultado es 1.5.

¿Y cuál es el valor de la expresión $3/2.0$? Esta pregunta nos remite a una cuestión fundamental en C: las conversiones de tipo, que desarrollamos en el siguiente apartado.

A.3.5. Las conversiones de tipo

¿Qué ocurre cuando se mezclan en una misma expresión variables o expresiones de tipos distintos, o si se trata de asignar a una variable el resultado de una expresión que tiene un tipo distinto al suyo? Por ejemplo, ¿qué ocurre en el caso de la expresión $3/2.0$?

²Donde / es el operador de división para reales y de división entera para enteros

Es necesario homogeneizar los elementos sobre los que se opera, es decir, transformar la operación en otra en la que los operandos sean del mismo tipo. Esto se puede hacer de dos formas distintas:

- De forma explícita (**casting**): el programador dice explícitamente que cierta expresión ha de ser transformada a cierto tipo. La sintaxis es *(tipo) expresión*. En el ejemplo anterior, si lo que queremos es realizar la división real, la reescribiríamos como `((float)3)/2.0.`
- De forma implícita: el programador no dice nada sobre las conversiones a realizar. El lenguaje se encarga entonces de suponer qué es lo que se pretendía hacer y de establecer las conversiones necesarias. En ese caso el programador debe ser plenamente consciente de las elecciones que hará el lenguaje. El esquema básico seguido por el lenguaje consiste en convertirlo todo al tipo de mayor rango, donde la idea intuitiva y quizás algo imprecisa de rango (ignorando los números negativos) es que un tipo es de más rango que otro cuando cualquier valor del segundo puede ser expresado mediante un valor del primero. La ordenación de rangos en C es la siguiente:

```
long double > double > float > unsigned long > long > unsigned int
> int > char
```

De este modo, la conversión implícita que se llevaría a cabo en la expresión `3/2.0` sería elevar el 3 a float de modo que la división que obtendríamos sería real.

A.3.6. Modos de almacenamiento de las variables

Hasta ahora hemos hablado sobre el tipo (elemental) que puede tener una variable, es decir, sobre lo que almacena. Pero hay otras características de las variables a tener en cuenta: cuándo existen, o dicho de otro modo, cuándo son creadas y cuándo destruidas (**duración**) y desde dónde son accesibles (**visibilidad**). Estos aspectos son modelados en C mediante los *modos de almacenamiento*. Según se declare una variable, le corresponderá un modo u otro, y la conjunción de este y el lugar en que fue declarada definirá su duración y visibilidad:

- **Variables locales (modo auto):** este es el modo por defecto de cualquier variable declarada dentro de un bloque, por lo que no se suele añadir la palabra `auto` en la declaración de la variable para calificarla como tal. Se trata de variables locales al bloque en que están definidas, es decir, solo visibles en él y en bloques anidados a este. Todas las variables declaradas dentro de un bloque han de estar al principio de este, antes del resto de sentencias. Las variables `auto` son creadas al comenzar la ejecución del bloque en que se declaran y destruidas cuando termina, de modo que no mantienen sus valores entre sucesivas ejecuciones del mismo bloque. En el siguiente programa `i` es un ejemplo de variable `auto`:

```
main(){
    int i;
    ...
}
```

- **Variables globales (modo extern):** se definen fuera de cualquier bloque o función. Existen durante toda la ejecución y son visibles desde cualquier función situada entre la declaración de la variable y el final del fichero. Aunque aún no hemos tratado el tema, se pueden tener programas con varios ficheros de código fuente (ver sección A.11, página 313). Una variable global se puede hacer también visible desde otro fichero o desde una posición anterior a su declaración dentro del mismo fichero declarándola: una variable **extern** se define una única vez pero puede ser declarada varias. La definición activa su existencia, la declaración simplemente permite que se conozca su existencia para que pueda ser utilizada. En el siguiente programa **i** es un ejemplo de variable **extern**:

```
int i;
main(){
    ...
}
```

- **Variables híbridas (modo static):** una variable **local** calificada como **static** (por ejemplo, **static int a**) es una mezcla entre local y global: tendrá la visibilidad de una variable local y la duración de una variable global, de modo que mantendrá su valor entre ejecuciones del bloque respecto a la que es local.

Cuando se califica como **static** una variable **global** o una **función** ocurre que su visibilidad ya no puede ser ampliada a zonas anteriores del fichero o a otros ficheros mediante declaraciones **extern**. Esta es una forma de restricción de acceso.

- **Modo register:** se trata de una recomendación al compilador para que trate de ubicar una variable calificada como tal en un registro del procesador con el fin de agilizar los cálculos. Esta es una muestra más de que C es un lenguaje de relativo bajo nivel.

En cualquiera de los cuatro casos se ha de tener en cuenta la circunstancia de que un bloque siempre supone un nuevo espacio de nombres para variables. Es decir, que una variable definida dentro de un bloque siempre ocultará a otra variable que fuese visible dentro del bloque antes de la nueva definición. Lo cual no quiere decir que la variable anterior deje de existir: simplemente no será accesible dentro del bloque hasta que la nueva variable desaparezca. \otimes Por ejemplo, en el siguiente fragmento de código la variable global deja de ser visible en algún punto de la ejecución. ¿Qué valor tendrá **a** al final de la ejecución³?

```
int a;
main() {
    int b=1,c=1;
    a=b;
    { int a;
        a = c + b;
```

³Para visualizar los valores de las variables podemos utilizar la función **printf**; si no lo has hecho ya, consulta la sección A.8, página 306, si quieras utilizarla.

```

    c = a + b;
}
a = c + a;
}

```

A.4. Operadores

Un operador es como una función que actúa sobre una serie de valores para devolver otro, solo que normalmente en vez de adoptar la notación típica de las funciones (*nombre-función(lista-argumentos)*) el operador aparece como uno o varios caracteres junto a los argumentos, sin paréntesis, y a menudo entre ellos (operadores infixos). Aunque, como veremos, hay excepciones a esta descripción y alguno parece una función. Suelen ser binarios (dos argumentos), como la suma o la resta, aunque también los hay unarios (un argumento) e incluso hay uno ternario (tres argumentos). Veamos los distintos tipos de operadores que C nos ofrece.

A.4.1. Aritméticos

En C contamos con los siguientes cinco operadores aritméticos, todos binarios: suma (+), resta (-), multiplicación (*), división (/) y resto o módulo (%). La división será entera o real dependiendo de los operandos. El módulo es obviamente sólo aplicable a operandos enteros. Adicionalmente podemos considerar en esta categoría al operador unario - que cambia el signo de la expresión a la que precede, así como a su homólogo +, incluido en el lenguaje por completitud pero que en realidad no hace nada.

A.4.2. Incrementales

Son operadores unarios que incrementan (++) o decrementan (--) una variable en una unidad. Según vayan antes o después de la variable, el valor de esta que se usará en la expresión en la que aparezca será el correspondiente al momento posterior o anterior a la aplicación del operador, respectivamente. Veámoslo con un ejemplo. En:

```

main() {
    int a=1,b=1,c,d;
    c = a++;
    d = ++b;
}

```

el valor final de c es 1 (valor de a antes del incremento) mientras que el valor final de d es 2 (valor de b tras el incremento). Tanto a como b acaban valiendo 2. Una regla nemotécnica para recordar cuál es el valor de la variable utilizado puede ser: para `++a`, según leo, primero encuentro el operador, luego primero incremento; después encuentro la variable, ya incrementada. Para `a++` primero encuentro a, tal cual estaba, la uso, y luego la incremento.

A.4.3. Relacionales

Para presentar este apartado es necesario explicar primero el tratamiento de los valores booleanos en C. En este lenguaje no existe un tipo booleano como ocurría en Pascal sino que se utilizan los números, tanto enteros como reales, como valores de verdad. Un 0 equivale a FALSE, cualquier otro número a TRUE. Cuando una expresión devuelve un booleano y este es verdadero, el valor numérico devuelto será 1. Este curioso sistema interconecta el ámbito de lo lógico y lo aritmético como se verá en esta y posteriores secciones.

Dicho esto, sigamos con los operadores relacionales. Se trata de operadores binarios que comparan en algún sentido los operandos y devuelven un valor de verdad que indica el resultado de la comparación. En C contamos con los siguientes:

- igual que (==) – no confundir con operador de asignación
- distinto de (!=)
- menor que (<)
- menor o igual que (<=)
- mayor que (>)
- mayor o igual que (>=)

Los operandos son, en general, expresiones. Para aplicar los operadores de comparación, obviamente se necesita que las expresiones correspondan a tipos en los que haya definido un orden. Pero pensemos que en C esto ocurre para cualquier tipo (al menos para cualquier tipo elemental), incluso para los booleanos, ya que en realidad son enteros.

A.4.4. Lógicos

Son operadores que actúan sobre valores lógicos o booleanos para devolver otro. Insistamos: aceptarán como verdadero cualquier valor numérico distinto de 0 y devolverán un valor que será 0 ó 1 para expresar FALSE o TRUE respectivamente. Tenemos uno unario, la negación (!), y dos binarios, la conjunción (&&), también llamada *Y* o *AND* y la disyunción (||), también llamada *O* u *OR*. Recordemos que por ser los valores de verdad números, y por ser cualquier tipo elemental un número, podemos obtener expresiones lógicas tan curiosas como 'a' && 10 (que será evaluada a TRUE y por tanto devolverá un 1).

A la hora de utilizar estos operadores hemos de tener en cuenta que su valor de verdad puede ser conocido tras la evaluación de uno de los operandos. Es decir, la conjunción es falsa en cuanto uno de los operandos sea evaluado como falso y la disyunción es cierta en cuanto uno de los operandos sea evaluado como cierto, de modo que no siempre el otro será evaluado. Esto es lo que se conoce como *evaluación resumida* o *en cortocircuito*. Este punto se ha de tener muy presente en cuanto a que no debemos asumir los efectos colaterales de la evaluación de los operandos implicados.

A.4.5. De manejo de bits

C proporciona seis operadores para actuar sobre los operandos bit a bit. Los operandos sólo pueden ser de tipo `char` o `int` (y derivados). Estos son los operadores disponibles:

operador	símbolo	descripción	ejemplos
AND	&	Conjunción bit a bit	<code>0xFF & 0x0F → 0x0F</code>
OR inclusivo		Disyunción (inclusiva) bit a bit	<code>0xFF & 0x0F → 0xFF</code>
OR exclusivo	^	Disyunción (exclusiva) bit a bit	<code>0xFF & 0x0F → 0xF0</code>
desplazamiento a izquierda	<<	Desplaza <i>n</i> posiciones hacia izq. <i>n</i> debe ser positivo. <i>n</i> primeros bits dchos se llenan con ceros.	<code>0x01 << 1 → 0x02</code> <code>0x11 << 2 → 0x44</code> <code>0x91 << 1 → 0x22</code>
desplazamiento a derecha	>>	Desplaza <i>n</i> posiciones hacia dcha. <i>n</i> debe ser positivo. <i>n</i> primeros bits dchos se llenan con: - variable <code>unsigned</code> , ceros - <code>signed</code> , ceros o bits de signo según la máquina.	<code>0x21 >> 1 → 0x10</code> <code>0x85 >> 2 → 0x21</code> <code>0xFF >> 3 → 0x1F</code> (relleno con ceros)
negación	~	Negación bit a bit	<code>~0xF0 → 0x0F</code>

A.4.6. De asignación

En C el operador de asignación tradicional es `=`. Pero adicionalmente encontramos nuevos operadores de asignación que suponen en realidad un acortamiento sintáctico de operaciones de asignación muy frecuentes en las que la variable que recibe la expresión calculada interviene también en la propia expresión. Hablamos de expresiones del tipo

`variable = variable op expresion`

donde `op` es un operador del conjunto `{ +, -, *, /, %, <<, >>, &, ^, | }`. La simplificación sintáctica es

`variable op= expresion`

De modo que, por ejemplo, el típico incremento unitario de una variable contador `n` pasa de escribirse como `n=n+1` a la nueva forma `n+=1`.

A.4.7. De dirección e indirección

El operador unario de indirección es `&` (no confundir con el operador lógico de conjunción, `&&`, que es doble, ni con el `&` lógico a nivel de bits, que es binario). Aplicado a una variable, devuelve la dirección de memoria en que se encuentra ubicada.

El operador unario `*`, aplicado a una expresión cuyo valor corresponda a una dirección de memoria, devuelve el contenido de esa dirección de memoria. Por supuesto, el contenido a devolver dependerá del tipo de dato almacenado que se considere.

Hay mucho que decir sobre estos dos operadores que han sido incluidos aquí para completar el apartado de operadores, pero que serán tratados con mucho más detalle

cuando hablemos de los **punteros**, elemento básico (¡y de los más complicados!) de la programación en C.

A.4.8. Otros operadores

- Operador **sizeof()**: devuelve el tamaño (en bytes) del tipo que se le pasa como parámetro. Su finalidad es permitir la portabilidad de código (dado que ese tamaño depende de la plataforma). Lo utilizaremos en algún ejemplo cuando trabajemos con *memoria dinámica* en la sección A.9, página 310.
- Operador **coma (,)**: es binario e infijo. Sus parámetros son dos expresiones. El efecto es que se evalúen secuencialmente, primero la de la izquierda y luego la de la derecha. El valor devuelto por el operador es el de la expresión de la derecha. El valor de la primera expresión es ignorado, de modo que el único efecto (y posible utilidad) de su evaluación son los efectos colaterales que pueda contener. Ejemplo:

```
main() {
    int i,j;
    j = ( i=1, 2*i );
}
```

A *j* se le asigna *i**2, donde el valor de *i* ha sido asignado en la propia expresión antes de calcular este producto. El operador coma es el que menos precedencia tiene en C. De ese modo, la utilización de paréntesis en la última sentencia del ejemplo anterior es imprescindible.

- Operador **? y expresiones condicionales**: se trata de un operador ternario e infijo. Una expresión condicional tiene esta estructura:

```
expr1 ? expr2 : expr3
```

Su significado es el siguiente: si *expr1* es evaluada como cierta entonces el valor de la expresión condicional es el resultado de evaluar *expr2*. En otro caso, será el resultado de evaluar *expr3*. Por ejemplo, para calcular la velocidad máxima en una autopista francesa en función del tiempo: *vmax* = *llueve* ? 110 : 130.

A.4.9. Expresiones: precedencia y asociatividad

Una vez conocidos los operadores disponibles en el lenguaje, nos queda entender cómo se evalúan las distintas expresiones que se pueden construir combinando expresiones más simples mediante estos operadores. Acabamos de presentar las reglas básicas que permiten saber cuál es el resultado de la aplicación de un operador a dos expresiones. Pero la cuestión es: ante una expresión larga y complicada, ¿qué operando se aplica primero? Puede haber, a priori, distintas opciones, y el resultado puede depender del orden elegido. Esta cuestión se ve claramente en este ejemplo: ante la expresión 3+3*3, si aplicamos primero la suma obtenemos como resultado 18, pero si aplicamos primero el producto obtenemos 12. Para establecer un criterio fijo para todas las ocasiones, se eligen la reglas de precedencia y asociatividad:

- La **precedencia** determina, ante dos operadores distintos, cuál ha de ser aplicado primero. En el ejemplo anterior, según que la mayor precedencia la tuviese el operador + o *, elegiríamos un orden u otro.
- La **asociatividad** determina, ante una secuencia de operadores binarios iguales (y por tanto con la misma precedencia), si hemos de ir aplicándolos de izquierda a derecha o de derecha a izquierda. Por ejemplo, ante la expresión $10/5/2$, si el operador / tiene asociatividad por la izquierda obtenemos como resultado 1 (10/5 es 2, 2/2 es 1), pero si la tiene por la derecha obtenemos 5 (5/2 es 2, 10/2 es 5).

La utilización de paréntesis evita tener que recurrir a las reglas de precedencia y asociatividad. En expresiones muy complicadas puede ser una buena idea hacer un uso intensivo de estos. En cualquier caso, a continuación se muestra una tabla con las reglas de precedencia y asociatividad utilizadas en C. La precedencia es mayor cuanto menor sea el número asociado. Alguno de los operadores no ha sido comentado; la idea es que esta tabla sirva como referencia:

Operadores	Precedencia	Asociatividad
<code>() [] -> .</code>	1	i-d
<code>++ -- ! ~ sizeof (tipo) + (unario) - (unario)</code>	2	d-i
<code>* (dirección) & (dirección)</code>		
<code>* / %</code>	3	i-d
<code>+ -</code>	4	i-d
<code><< >></code>	5	i-d
<code>< <= > >=</code>	6	i-d
<code>== !=</code>	7	i-d
<code>&</code>	8	i-d
<code>^</code>	9	i-d
<code> </code>	10	i-d
<code>&&</code>	11	i-d
<code> </code>	12	i-d
<code>? :</code>	13	d-i
<code>= += -= *= /= %= ^= = <<= >>=</code>	14	d-i
<code>, (operador coma)</code>	15	i-d

⊗ Veamos un ejemplo de aplicación de estas reglas. Sea la expresión

`1+2*3&&2==5>=3!=4+23||7<2`

Insistimos: ante expresiones tan complejas, lo mejor es utilizar paréntesis. Pero intentemos aplicar las reglas de precedencia y asociatividad para deducir su valor. En la siguiente tabla se va transformando la expresión paso a paso, indicando cuál es el operador elegido en cada paso y comentando por qué.

Operador	Motivo elección	Cálculo	Expresión
*	(expresión original)		$1+2*3\&\&2==5>=3!=4+23 7<2$
+ (izq)	mayor precedencia (3)	$2*3 \rightarrow 6$	$1+6\&\&2==5>=3!=4+23 7<2$
	mayor prec (4), izq-dcha	$1+6 \rightarrow 7$	$7\&\&2==5>=3!=4+23 7<2$
+	mayor prec (4)	$4+23 \rightarrow 27$	$7\&\&2==5>=3!=27 7<2$
>=	mayor prec (6), izq-dcha	$5>=3 \rightarrow 1$	$7\&\&2==1!=27 7<2$
<	mayor prec (6)	$7<2 \rightarrow 0$	$7\&\&2==1!=27 0$
==	mayor prec (7), izq-dcha	$2==1 \rightarrow 0$	$7\&\&0!=27 0$
!=	mayor prec (7)	$1!=27 \rightarrow 1$	$7\&\&1 0$
&&	mayor prec (11)	$7\&\&1 \rightarrow 1$	$1 0$
	mayor prec (12)	$1 0 \rightarrow 1$	1

En C los compiladores normalmente calculan las expresiones constantes en tiempo de compilación. De ese modo, la expresión anterior sería sustituida por un 1 en tiempo de compilación.

A.5. Sentencias de control de flujo

En C contamos con sentencias de control de flujo al estilo estructurado (sentencias de selección e iteración) así como también con la sentencia `goto` que permite saltar a una etiqueta (declarada como `etiqueta:`) que marca un punto del programa. Esta segunda forma de control de flujo se comenta para que el conocimiento del lenguaje sea más completo, pero su uso de desaconseja totalmente ya que va en contra de la programación estructurada y en realidad estas sentencias no son realmente necesarias: dado un programa con `goto` siempre podemos escribir otro equivalente sin ellos, usando solo sentencias de control de flujo estructuradas.

A.5.1. Sentencias de selección

Permiten elegir la acción a realizar en función de una o varias condiciones. En primer lugar tenemos la **sentencia condicional** que en C adopta la forma

```
if (condicion) sentencia
```

Los paréntesis son obligatorios. La condición es cualquier expresión, su resultado será utilizado como un valor de verdad. La sentencia puede ser simple o un bloque; se ejecutará si la condición es evaluada como cierta. \otimes Veamos un ejemplo:

```
main(){
    int i=0;
    if (2==2) i++;
}
```

El valor final de `i` será 1 dado que la condición es cierta y por tanto se ejecuta la sentencia que incrementa el valor inicial de esa variable (cero). También disponemos de la versión con acciones alternativas a ejecutar en caso de que la condición sea falsa:

```
if (condicion) sentencia01
else           sentencia02
```

⊗ Por ejemplo:

```
main(){
    int i=2;
    if (!7) i++;
    else {
        i--;
        i--;
    }
}
```

7 es un número distinto de cero, por tanto verdadero como condición, y su negación falsa. Al no satisfacerse la condición no se ejecuta el incremento sobre *i* sino el decremento (dos veces), de modo que el valor final de *i* es cero. Fíjate que en este caso la sentencia asociada a *else* es un bloque (que como ya se ha dicho también es una sentencia).

Es interesante aquí comentar una cuestión importante: la existencia de una ambigüedad en la construcción *if-else*. Un programa es ambiguo cuando puede ser interpretado de varias formas diferentes. El compilador siempre elegirá una, de modo que es importante que seamos conscientes de la presencia de dichas situaciones y sepamos cómo actúa el compilador ante ellas para evitar interpretaciones indeseadas del código que escribamos. Veamos en qué consiste la ambigüedad introducida por *if-else* mediante un ejemplo:

```
if (condicion01)
if (condicion02) sentencia02
else sentencia03
```

Esta porción de código se ha escrito intencionadamente sin indentación. El compilador no utiliza la indentación para interpretar el código, se trata solo de una ayuda visual para el ser humano. Se puede interpretar de dos formas:

- Un *if-else* con una sentencia para el *if* que a su vez es un *if*:

```
if (condicion01)
    if (condicion02) sentencia02
else sentencia03
```

- Un *if* cuya sentencia es un *if-else*:

```
if (condicion01)
    if (condicion02) sentencia02
    else sentencia03
```

¿A qué *if* quedará asociado el *else*? En C siempre se asocia al *if más cercano*, es decir, el compilador opta por la segunda interpretación. La forma de evitar esta ambigüedad o de conseguir la primera interpretación es utilizar llaves (de forma parecida al uso de paréntesis en las expresiones). Por ejemplo, para conseguir la primera interpretación:

```
if (condicion01) {
    if (condicion02) sentencia02
}
else sentencia03
```

Estamos definiendo para el `if` externo una sentencia que es un bloque constituido por una única sentencia, un `if` sin `else`.

Para la segunda interpretación sería:

```
if (condicion01) {
    if (condicion02) sentencia02
    else sentencia03
}
```

Estamos definiendo para el primer `if` una sentencia que es un bloque constituido por una sentencia `if` con `else`. Las llaves externas no serían necesarias, dado que esta es la interpretación por defecto.

Cuando utilizamos como sentencia de un `else` un nuevo `if-else` y repetimos la operación cierto número de veces, tenemos una construcción muy útil, el **`if-else encadenado`**, que no es más que un caso particular de la generalidad a que da lugar el `if-else`. Veámoslo con un ejemplo, un fragmento de código para elegir postre según ciertas preferencias:

```
if(hay_fresas_con_nata)
    eleccion = fresas_con_nata;
else
    if(hay_tarta_de_chocolate)
        eleccion = tarta_de_chocolate;
    else
        if(hay_flan)
            eleccion = flan;
        else
            eleccion = no_quiero_postre;
```

La lógica de elección es simple: si hay fresas con nata se elige esto y hemos acabado. Si no, se evalúa la siguiente condición. Si es cierta (hay tarta de chocolate) se elige la tarta y se acaba. Si tampoco hay tarta, hacemos un último intento con el flan. Si tampoco hay flan, desistimos desesperados. Es decir, el `if-else` encadenado permite la elección de una opción entre varias mediante una serie de preguntas sucesivas que van descartando posibilidades.

Otra forma de elegir una opción de entre varias es a través de la **sentencia de selección múltiple: `switch`**. En este caso no se evalúa una condición sino una expresión cuyo resultado es tomado como tal, sea del tipo que sea. Se ejecutará una u otra acción según su valor coincida con uno u otro de entre una lista de valores constantes dados. Opcionalmente se puede añadir una acción por defecto, a ejecutar si el valor de la expresión no estaba en la lista. Su forma general es:

```
switch (expresion) {
    case expcte01: sentencias01
```

```

        case expcte02: sentencias02
        (...)

        case expcteN:  sentenciasN
        default: sentencia_por_defecto
    }

```

Fíjate en que en este caso hablamos de **sentencias**, en plural. Puede haber cero, una o varias, sin necesidad de utilizar un bloque para albergar varias sentencias. La línea **default** determina la acción por defecto. Una vez evaluada **expresion**, si su valor coincide con **expcte01** se ejecutará el grupo de sentencias **sentencias01** y todos los demás **grupos**. Si no, si coincide con el valor de **expcte02** se ejecuta el grupo **sentencias02** y todos los grupos tras ella. Y así sucesivamente. Esto nos recuerda a un **if-else** encadenado, con la diferencia de que en aquel, una vez evaluada una condición a verdadero, se ejecutaba su sentencia asociada y se terminaba. Para conseguir ese comportamiento con **switch**, se debe incluir una sentencia **break** en cada **case** que se quiera. Su efecto es que se da por acabada la ejecución del **switch** cuando se ejecuta. Por último, antes de un ejemplo ilustrativo, comentamos que se puede usar una misma sentencia para varios casos distintos. Para ello se enumeran todos los casos (**case expcte01: case expcte02: ...**) y a continuación se indica la sentencia común. Veamos el ejemplo. En este fragmento de código se determinan las tres orígenes distintos⁴:

```

switch (punto_de_partida) {
    case lisboa:
        paso_por_lisboa = 1;
    case madrid:
        paso_por_madrid = 1;
        paso_por_barcelona = 1;
        break;
    case paris:
        paso_por_paris = 1;
    default:
        paso_por_barcelona = 1;
}

```

⊗ En cualquier trayecto siempre paso por la ciudad de origen (distinta en cada caso) y destino (común). Además, desde Lisboa también paso por Madrid. Examina con atención el código para captar todos los detalles. En particular, fíjate en cómo se reutilizan las sentencias de Madrid para Lisboa; o la acción por defecto para París. También fíjate en que si la ciudad de origen es otra, no considerada, se anotará al menos el paso por Barcelona a través de la sentencia **default**.

A.5.2. Sentencias de iteración

En C contamos con tres sentencias para hacer iteraciones: **while**, **do-while** y **for**. Veamos cómo funciona cada una de ellas.

while adopta la siguiente forma:

⁴Se utilizan las constantes enteras **lisboa**, **madrid**, **barcelona** y **paris**, definidas mediante la sentencia **#define**, explicada en la sección A.10, página 312.

```
while (expresion) sentencia
```

La expresión será utilizada como valor de verdad. Será evaluada una primera vez. Si es cierta, se ejecuta la sentencia y se repite el proceso hasta que la expresión sea evaluada a falsa. Por ejemplo, en:

```
main(){
    int i = 0;
    while (i<10) i++;
}
```

la primera vez *i* vale 0, de modo que se incrementa. Esto se hace sucesivamente hasta que, cuando *i* vale 9, se hace un último incremento que hace que la condición ya no sea cierta y el bucle termine.

Por otra parte, tenemos una iteración muy parecida con do-while. La única diferencia es que la condición se evalúa al final. La forma general es:

```
do sentencia while(expresion);
```

Si reescribimos el ejemplo anterior como:

```
main(){
    int i = 0;
    do i++ while(i<10);
}
```

lo que ocurrirá ahora es que se produce un primer incremento y entonces se pregunta por la condición. Esto se repite hasta que *i* alcanza el valor 10, momento en que la condición es falsa y el bucle termina. \otimes En este caso, el efecto ha sido el mismo en ambos bucles, *i* ha recorrido la secuencia de valores 0,1,...,10. Pero ¿qué ocurriría si ahora cambiamos la condición del bucle por (*i*>0 && *i*<10)?

El bucle for merece una explicación algo más detallada. Se trata de una construcción muy potente y que difiere bastante de lo que solemos encontrar en otros lenguajes en los que se limita a hacer que una variable índice recorra un intervalo de enteros, a lo sumo permitiendo un recorrido descendente o con incrementos mayores que la unidad. La estructura de este bucle es

```
for (expr01 ; expr02 ; expr03) sentencia
```

expr01, *expr02* y *expr03* son formalmente expresiones, no sentencias. Como ya se ha comentado, toda expresión puede actuar como sentencia. Teniendo esto en cuenta, podemos entender el funcionamiento de for sabiendo que la estructura anterior es totalmente equivalente a:

```
expr01;
while (expr02) {
    sentencia
    expr03;
}
```

Es decir, `expr01` actúa como una inicialización del bucle; suele ser una asignación (o varias mediante el empleo del operador *coma* (,), ver página 288). `expr02` es una condición de continuación, el bucle se seguirá ejecutando mientras sea cierta. Y `expr03` se ocupa de llevar a cabo algún tipo de actualización (aquí también puede ser útil el operador *coma*), por ejemplo, incrementar un índice que es comprobado por `expr02`. Ninguna de las tres expresiones es obligatoria, solo los dos puntos y coma lo son. Si la primera expresión o la tercera no están, no hay ningún efecto: simplemente no hay inicialización o actualización, respectivamente, expresadas en el propio `for`. Si falta la segunda expresión lo que ocurre es que se asume que la condición es siempre verdadera, de modo que tenemos un bucle infinito salvo que su ejecución sea interrumpida por otros medios (como `break`, que se explica a continuación, `return`, que se explicará cuando hablemos sobre las funciones en C en la sección A.7, página 303, o `goto`). El ejemplo utilizado para explicar la sentencia `while` puede ser reescrito en términos de `for` del siguiente modo:

```
main(){
    int i;
    for(i=0; i<10,i++);
}
```

Este bucle no tiene cuerpo (o, mejor dicho, su cuerpo es una sentencia vacía) ya que todo queda hecho en la propia sentencia `for`.

En C se incluyen facilidades para interrumpir la ejecución de un bucle que se asemejan algo a un `goto` pero no llegan a producir el mismo efecto de *código espagueti* de los programas. Se trata de las sentencias `break` y `continue`. La sentencia `break` produce directamente la interrupción del bucle completo. La ejecución del programa saltará a la siguiente sentencia tras el bucle. La sentencia `continue` interrumpe tan solo la pasada actual del bucle. La ejecución salta a la condición de continuación, en función de la cual se seguirá con una nueva ejecución del bucle o no. La desestructuración que producen no es tanta como la de `goto`, ya que el punto de salto es conocido localmente por el programador y siempre se podría obtener el mismo efecto con una combinación adecuada de condiciones.

A.6. Tipos de datos agregados y punteros

Un **tipo agregado o compuesto** es aquél cuyos valores están constituidos por valores de otros tipos. Es el caso de las tablas, denominadas `array` en C, que almacenan *n* elementos de un mismo tipo, y de los registros o estructuras, `struct` en C, que almacenan elementos de distintos tipos con etiquetas asociadas a cada uno, y de las uniones, `union`, parecidas a las estructuras. Los punteros son direcciones de memoria, apuntan al lugar en que se almacena algo. En realidad no son agregados, pero se parecen a ellos en que el tipo de un puntero se define sobre la base de otro tipo, el tipo de la variable apuntada. De modo que tendremos punteros a entero, a real, a tabla de enteros, etc. Además, en el caso particular de C, existe una relación muy estrecha entre los punteros y los arrays que describiremos en la subsección A.6.2 (página 297).

A.6.1. Punteros

La forma de declarar que una variable es un puntero a un cierto tipo es:

```
tipo *nombre_variable;
```

Es decir, se antepone un asterisco al nombre de la variable, no al del tipo en cuestión. Esto permite encontrar declaraciones de variables como: `int a,*b` donde `a` queda declarada como entero y `b` como puntero a entero. Cualquier puntero ocupa la misma cantidad de memoria independientemente del tipo al que apunte, de modo que, previa conversión explícita (obligatoria), podríamos declarar perfectamente un puntero a `int` y luego asignarle la dirección de un `float`, como veremos enseguida en un ejemplo.

La dirección de una variable se obtiene mediante el **operador dirección**, `&`: devuelve la dirección de memoria en que se ubica la variable a la que se aplica, en forma de puntero al tipo de la variable. Una constante o una expresión no tienen dirección por lo que no se les puede aplicar. \otimes Veamos el siguiente ejemplo, en el que hacemos una conversión de tipo explícita y otra implícita (para repasar estos conceptos, ver la sección A.3.5, página 282). Observa que la forma de referirse al tipo *puntero a tipo* en el casting explícito es `(tipo *)`:

```
main(){
    int i=1,*p;
    float f=2.0;
    p = &i;           // no se necesita conversion
    p = (int *)&f;   // conversion explicita
    p = &f;          // conversion implicita
}
```

El operador inverso del operador dirección es el **operador indirección**, `*`: aplicado a un puntero a tipo *tipo*, devuelve el elemento de tipo *tipo* contenido a partir de la posición de memoria indicada por el puntero. Según se trate de un puntero a un tipo u otro estaremos obteniendo un valor u otro. **Atención:** en C, al contrario que en otros lenguajes, se utiliza en mismo símbolo, el asterisco (`*`), para referirse a un tipo *puntero a tipo* y como operador indirección. \otimes Veamos un ejemplo de utilización del operador indirección:

```
main(){
    int c=1,b,*p;
    p=&c;           /* p apunta a c */
    b=*p;            /* b recibe el valor de c a traves de p */
    *p = ++b;        /* b se incrementa y c recibe su valor a traves de p */
}
```

Una última cuestión a considerar en cuanto a los punteros en C es la **aritmética de punteros**. Los punteros almacenan direcciones de memoria y adicionalmente están asociados a un cierto tipo, el del dato que se almacena en esa dirección. Si conocemos el tipo contenido en esa dirección conocemos el número de bytes que ocupa y por tanto se puede calcular la dirección en la que se encuentra el siguiente dato en memoria. De ese modo, si tenemos varios datos del mismo tipo consecutivos en memoria (como será el caso de los arrays, explicados en la próxima sección) podemos acceder a cualquiera de ellos a partir de la dirección del primero. \otimes Lo ilustramos con un ejemplo:

```
main(){
    int a,b,*p;
    p = &a;
    *p = 1;
    *(p+1) = 2;
}
```

El resultado es que se ha asignado un 1 a a y un 2 a b. ¿Cómo? La asignación de a está clara. Respecto a la de b, como está situado en memoria a continuación de a, *fabricamos* un puntero a b a partir del puntero a a sumándole *uno*. ¿Qué significa ese *uno*? No es un byte, es el tamaño en bytes del tipo al que apunta el puntero. Es decir, consigue que la dirección inicial se convierta en la dirección del siguiente entero en memoria. Esto es lo que se llama aritmética de punteros: se puede sumar o restar a una dirección un número entero, se pueden restar dos direcciones y así calcular el número de elementos del tipo en cuestión entre ambas, etc. Por contra, no se pueden hacer cosas como multiplicar o dividir punteros ni operar entre sí punteros de tipos distintos. Veremos más cuestiones sobre aritmética de punteros en la siguiente sección.

Podemos definir un puntero a *tipo no definido* mediante el tipo void: void *pnodef. En cuanto a aritmética de punteros, para un puntero a void un incremento significa un incremento de un byte. En la librería stdio se define un puntero especial, NULL, como (void *) 0. Tiene diversos usos:

- Un puntero puede ser usado como valor de verdad: NULL equivale a *false*, cualquier otro puntero a *true*.
- NULL apunta a una zona de memoria que nunca debería ser accedida por el usuario, es un puntero *no válido* en el sentido de que no apunta realmente a una variable. Por este motivo NULL puede ser utilizado como código de error en una función cuyo resultado sea un puntero: si el puntero devuelto es NULL algo fue mal.
- Que sea un puntero no válido permite otro uso: el compilador inicializará a NULL todo puntero no inicializado por el usuario. De este modo será fácilmente detectable la circunstancia de que el usuario acceda a lo apuntado por el puntero antes de que éste haya sido inicializado. Esta situación arrojará el mensaje **NULL pointer assignment**.

Respecto a la asignación entre punteros, no se pueden asignar entre sí punteros a tipos distintos (sin casting explícito previo) salvo si uno de ellos es void. A un puntero a void sí que podemos asignarle otro puntero de cualquier tipo o asignarlo a un puntero de cualquier tipo.

A.6.2. Arrays

Un array almacena en posiciones consecutivas de memoria un cierto número de elementos del mismo tipo (básico o no). La forma de declarar una variable array de tipo tipo con n elementos es:

```
tipo nombre_variable[n];
```

Al igual que ocurría con los punteros, los corchetes y el número de elementos acompañan al nombre de la variable, no al tipo base, de modo que son posibles declaraciones como esta: `int a,b[10]` en la que declaramos conjuntamente un entero y un array de 10 enteros. A la hora de acceder mediante un índice a las distintas posiciones del array, los elementos se numeran desde 0 a $n - 1$. Para referirnos a un elemento individual del array se escribe el nombre del array seguido del índice del elemento entre corchetes, por ejemplo, `b[2]` para acceder al tercer entero de `b`. Este elemento puede aparecer en cualquier lugar en que podría hacerlo una variable de un tipo básico.

En C no se puede asignar un array a otro, sino que hay que asignar uno a uno los elementos individuales. Veamos un ejemplo de copia de un array en otro:

```
main(){
    int i,a[10],b[10];
    for(i=0;i<10;i++) a[i]=i;      /* se inicializa a = 0,1,2,...,9 */
    for(i=0;i<10;i++) b[i]=a[i];  /* b=a, elemento a elemento */
}
```

Un array puede ser **inicializado en su declaración**. El array a asignar se escribe como una secuencia de elementos separados por comas y entre llaves, por ejemplo: `int a[3]={1,2,3}`. Cuando un array se inicializa en su declaración, se puede omitir su tamaño ya que queda determinado implícitamente por el array asignado: `int a[]={1,2}`.

Una **cadena de caracteres** en C es un array de `char`. Se utiliza un carácter especial, '`\0`', cuyo valor numérico es cero, para marcar el fin de la cadena, de modo que, aunque el array que da soporte a la cadena tenga `n` caracteres, la cadena está formada solo por los que hay antes de ese carácter especial. A la hora de inicializar un array como cadena de caracteres, se puede hacer según se describió arriba para los arrays en general (como se hace para la cadena `a` en el siguiente ejemplo) o mediante una constante cadena de caracteres, según se hace para `b` en el ejemplo:

```
#include <stdio.h>
main(){
    char a[10]={'c','a','d','e','n','a','','a','\n','\0'};
    char b[13]="cadena b\n"; // \n indica fin de linea
    printf(a); printf(b);
    b[10] = 'X'; b[11]='\n'; b[12] = '\0';
    printf(b);
    b[9]='-'; printf(b);
}
```

En primer lugar se imprimen ambas cadenas por pantalla. Observa que en `a` se utilizan en la inicialización todos los caracteres declarados, pero en `b` se declaran tres caracteres más de los que utiliza la cadena con la que se inicializa el array. Eso significa que hay espacio en memoria para tres caracteres más tras el carácter '`\0`', que se indicó explícitamente en la inicialización de `a` (por usar formato general de arrays) pero es implícito en la de `b` por usar inicialización con formato de cadena. \otimes Si añadimos en esas tres posiciones un carácter X, un nuevo fin de línea ('`\n`') y un nuevo fin de cadena ('`\0`') y volvemos a imprimir `b`, ¿qué obtenemos? Comprueba que nada ha cambiado. Esto es así porque aún hay un fin de cadena previo a este en la posición 10 (índice 9); el

contenido del array que contiene la cadena es "cadenab\n\0X\n", por lo que la cadena es "cadenab\n". Si lo sobrescribimos con un carácter, en el ejemplo -, ahora sí, obtenemos la cadena "cadena b\n-X\n".

Hasta este punto hemos hablado de arrays unidimensionales, también denominados vectores. En C disponemos de arrays n-dimensionales: arrays cuyos elementos son arrays (n-1)-dimensionales. Se declaran como:

```
tipo nombre_array[tam_1]...[tam_n]
```

donde tam_i indica el número de elementos en la dimensión i-ésima. Los arrays unidimensionales son considerados filas (almacenamiento por filas), los bidimensionales son vectores cuyos elementos son filas, los tridimensionales vectores cuyos elementos son arrays bidimensionales y así sucesivamente. Cada elemento del array se especifica mediante notación nombre_array[i_1]...[i_n], donde i_1 especifica el array (n-1)-dimensional, i_2 especifica, dentro de este, el array (n-2)-dimensional,... i_(n-1) especifica la fila dentro del array bidimensional seleccionado y finalmente i_n selecciona el elemento deseado dentro de esa fila.

Veamos un ejemplo. El siguiente programa calcula el máximo y mínimo en cada fila de una matriz a de 3 filas de 4 elementos y los deja en sendos vectores maxs y mins. Fíjate en la inicialización de a en la declaración. Es coherente con la idea de que un array bidimensional es un vector de vectores. Siguiendo la misma idea se puede inicializar cualquier array n-dimensional:

```
main(){
    int a[3][4] = { {1,2,3,4},
                    {9,2,3,6},
                    {1,2,1,0} };
    int maxs[3], mins[3], i, j;
    for(i=0; i<3; i++) {
        maxs[i] = mins[i] = a[i][0];
        for(j=1; j<4; j++) {
            if(maxs[i] < a[i][j]) maxs[i] = a[i][j];
            else if(mins[i] > a[i][j]) mins[i] = a[i][j];
        }
    }
}
```

A.6.3. Relación entre arrays y punteros

Existe una relación muy estrecha entre los arrays y los punteros en C:

- El nombre de un array es, a todos los efectos, un puntero a la posición de memoria en que se almacena su primer elemento, siendo la única diferencia respecto a un puntero que la dirección del array es constante: no puede almacenar otra dirección de memoria distinta. Si A es un array de enteros declarado como int A[10], entonces A es también un puntero que contiene la dirección &A[0]. Por serlo podemos acceder a los elementos de A mediante punteros y aritmética de punteros: al primero mediante *A (en vez de A[0]), al siguiente mediante *(A+1) (en vez de A[1]), y así sucesivamente hasta *(A+9) (en vez de A[9]).

- Por otra parte, un puntero puede ser utilizado para acceder a distintas posiciones de memoria a partir de la que él almacena mediante índices en lugar de mediante aritmética de punteros. Dado el array anterior, si declaramos `int *p` y asignamos `p=A`, podemos acceder a los elementos de `A` mediante `p` de este modo: al primero como `p[0]` (en lugar de `*p`), al segundo como `p[1]` (en lugar de `*(p+1)`) y así sucesivamente. Es importante recordar que cuando se declara un puntero no se está reservando ningún espacio de almacenamiento de elementos del tipo al que está asociado, al contrario de lo que ocurre con un array. Que podamos acceder a posiciones de memoria a través de un puntero al modo en que se hace con los arrays, nos puede llevar a confundir en algún punto de un programa un puntero con un array y de ese modo acceder a posiciones de memoria incorrectas.

⊗ Veamos algunos ejemplos de lo comentado en el siguiente programa:

```
main(){
    int A[5]={0,1,2,3,4},*p;
    *A==*(A+1); // equivale a A[0]=A[1];
    *p = A[1]; // INCORRECTO: p no apunta a nada aun
    p = A; // p apunta a A[0]
    *p = A[1]; // efecto: A[0]=A[1]
    p[1]==*(A+2); // efecto: A[1]=A[2]
    p = A+1; // p apunta a A[1]
    A = p; // INCORRECTO: direccion de A es constante
}
```

¿Qué ocurre con los arrays n-dimensionales? Veámoslo con el caso particular de los bidimensionales (matrices). La generalización es inmediata. Según hemos visto anteriormente, una matriz es un vector de vectores (fila) y cada una de esas filas es un array unidimensional. Es decir, cada fila es un puntero a una zona de memoria donde se almacenan sus elementos. Y el array bidimensional es un puntero a una zona de memoria donde se almacenan todos esos punteros a las filas. Por ejemplo, si definimos:

```
int A[3][3] = { {1,2,3},
                 {4,5,6},
                 {7,8,9} } ;
```

La estructura de punteros que se establece es la que se muestra en la figura A.1:

- Cada uno de los 9 enteros en `A` es accedido como `A[i][j]`, con $i, j \in \{0, 1, 2\}$.
- Visto como array unidimensional, `A` almacena punteros a enteros (que son arrays de enteros). De ese modo, `A[0]` no es el primer elemento de la matriz, que sería `A[0][0]`, sino el primer puntero a entero, que indica el comienzo de la primera fila (el primer vector de enteros).
- Si pensamos en `A` como puntero, es un puntero a puntero a entero. Un puntero `p` de ese tipo se declararía como `int **p`.

⊗ Veamos un ejemplo de manejo del array `A` de diversas formas:

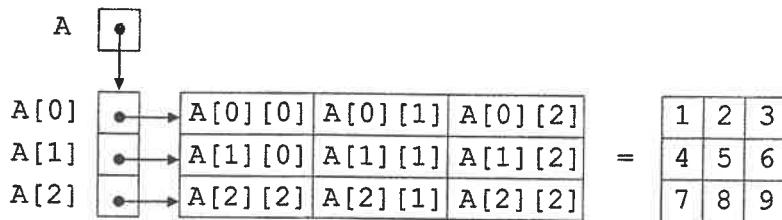


Figura A.1: Estructura de punteros para array bidimensional.

```

main(){
    int A[] []= { {1,2,3}, {4,5,6}, {7,8,9} };
    int **B,*F;
    A[0][0]=A[2][2]; // acceso, tipo array, a elementos
    F = A[1]; // F equivale a segunda fila de A
    *F = 0; // A[1,0]=0 a traves de F como puntero
    F[1] = 0; // A[1,1]=0 a traves de F como array
    B = A; // B equivale a A
    B[1][1]=B[2][2]; // A[1][1]=A[2][2] a traves de B
    B = A+1; // B = subarray de A, 2 ultimas filas
    B[0][2]=B[1][2]; // A[1][0]=A[2][1] a traves de B
    B[1][2]=B[2][2]; // ERROR: A[2][0]=A[3][1], FUERA DE RANGO
}

```

El error que se produce en la última línea consiste en que estamos escribiendo sobre una zona de memoria que no corresponde al array A. Las consecuencias son imprevisibles. Si la zona escrita corresponde a otras variables esto nos llevará probablemente a un error lógico en nuestro programa, ya que esas otras variables recibirán valores arbitrarios que sobrescriben sus verdaderos valores. También podría ocurrir que esa zona corresponda a código o incluso ni siguiera corresponda a nuestro programa. En este último caso el programa será abortado por el sistema operativo, que nos indicará lo que ha ocurrido mediante el mensaje `segmentation fault`.

A.6.4. Estructuras y uniones

Las estructuras nos permiten agrupar en una variable varios datos de tipos distintos, son tipos agregados heterogéneos. Los distintos datos agrupados, de tipo básico o agregado, se denominan miembros de la estructura y son identificados mediante una etiqueta. Por ejemplo, nos puede interesar agrupar en una variable el DNI, nombre y edad de una persona. Para usar variables de este tipo, primero hay que crear un patrón de estructura que actúa como tipo de dato para luego crear instancias de este. Por ejemplo, creamos el patrón de estructura para personas:

```

struct persona {
    unsigned long DNI;
    char nombre[100];
    unsigned short edad;
};

```

persona es el nombre del patrón de estructura creado. La palabra `struct` y el punto y coma final deben aparecer siempre. Entre llaves se incluye la lista de *miembros* que es idéntica en la forma a una declaración de variables donde los nombres de variables son en realidad las etiquetas asociadas a los miembros. Una vez definida la estructura, el tipo a utilizar para declarar variables es `struct persona`, por ejemplo: `struct persona persona1`. También se pueden instanciar variables en la propia definición del patrón de la estructura:

```
struct persona {
    unsigned long DNI;
    char nombre[100];
    unsigned short edad;
} persona1;
```

Para acceder a los distintos miembros de una variable de este tipo, se utiliza la *notación punto*: `persona1.DNI`, `persona1.nombre`. Si tenemos definido un puntero a una estructura, por ejemplo `struct persona *p` y luego `p=&persona1`, existe una notación simplificada para acceder a los campos de la estructura a través del puntero, el **operador flecha**: `p->DNI` en lugar de `(*p).DNI`, `p->nombre` en lugar de `(*p).nombre...`

Las estructuras, al contrario que los arrays, sí pueden asignarse directamente, sin necesidad de hacer una copia campo a campo. Respecto a la inicialización en la declaración, se puede hacer de forma parecida a los arrays, incluyendo entre llaves y separados por comas los valores para los distintos campos en el mismo orden en que se declaran en el patrón. Siguiendo con el mismo ejemplo: `struct persona persona1 = {22000000, "Juan", 27}`.

Las **uniones** son algo así como una estructura en la que todos los miembros comparten la misma memoria, de modo que en realidad solo uno puede existir en cada momento. Según la etiqueta utilizada para acceder cada vez se estará accediendo a la misma posición de memoria pero de modo distinto según el tipo de dato asociado a la etiqueta. Las uniones proporcionan un medio de definir una variable que pueda ser de tipos distintos según nos interese. Por ejemplo, para la estructura persona, ahora vamos a utilizar en lugar de nombre y DNI un único campo *identificador* que será una de las dos cosas. Para ello definimos el siguiente patrón de unión:

```
union identificador {
    unsigned long DNI;
    char nombre[100];
};
```

Y basándonos en este tipo redefinimos el de persona como:

```
struct persona {
    union identificador id;
    unsigned short edad;
};
```

Para una variable de `p` de tipo `persona` ahora podríamos escribir: `p.id.DNI=12345` o bien `p.id.nombre="Juan"` según quisiésemos usar un identificador o el otro.

A.6.5. Definición de tipos

C permite dar un nombre a cualquier tipo, ya sea proporcionado por el sistema o creado por el usuario. Este nombre podrá ser usado en cualquier lugar en el que se espere un tipo, ya sea declaración, casting, etc. La sintaxis es: `typedef tipo nombre_tipo`. Por ejemplo, podemos dar nombre a la estructura `persona` definida anteriormente, momento a partir del cual podremos declarar variables mediante ese nombre:

```
typedef struct persona PERS; // Alternativamente podríamos hacer
PERS persona1,persona2; // struct persona persona1, persona2;
```

A.7. Funciones

La estructura general de la definición de una función en C es

```
tipo_valor_devuelto nombre_funcion(lista_argumentos) cuerpo
```

Como ya se ha comentado, la especificación del tipo del valor devuelto es opcional. Por defecto se asume `int`. Si se quiere que la función no devuelva nada, se usa el tipo especial `void` (nulo). El valor devuelto es único. Se pueden devolver estructuras pero no un array con su contenido, aunque sí el puntero asociado al nombre de un array. La lista de argumentos se especifica como una lista de pares (*tipo nombre*) separados por comas. La parte de la definición anterior al cuerpo se denomina **cabecera**.

El cuerpo de la función es un bloque de código, el código a ejecutar cuando esta se llama. Como en todo bloque, se pueden declarar variables locales a él al principio. Como ya se ha comentado, estas variables serán `auto` por defecto, de modo que son creadas cada vez que la función es llamada y destruidas cuando finaliza su ejecución, así que sus valores se pierden entre llamadas sucesivas. Si las declaramos como `static`, los valores serán conservados entre llamadas. Se ha de tener precaución con el uso de variables `static` en funciones recursivas: una variable local `static` no tendrá una instancia distinta para cada llamada recursiva a la función, sino que todas las llamadas comparten la misma instancia, como si fuese una variable global.

El código del cuerpo de la función se ejecuta secuencialmente hasta el final o hasta que se encuentra la palabra `return`, cuyo efecto es terminar la ejecución de la función; adicionalmente se puede utilizar para especificar el valor de retorno de la función. Puede aparecer en varias veces en el cuerpo. Veamos un ejemplo de definición de función:

```
int division_segura(unsigned int dividendo, unsigned int divisor){
    unsigned int cociente;
    if (divisor==0) return -1;
    cociente = dividendo/divisor;
    return((int)cociente);
}
```

Fíjate en que el `if` no tiene `else`: si se ejecuta la sentencia asociada al `if`, la ejecución de la función termina en ese punto, de modo que no es necesario un `else`. Fíjate también en que, con `return`, en un caso se han utilizado paréntesis y en otro no. Su uso

es indiferente. Por último, observa que se utiliza -1 como código de error para indicar que algo fue mal (el divisor era cero), así como el casting explícito en el segundo `return`.

Al igual que cualquier variable debe estar declarada antes de ser usada, cualquier función debe también estarlo. La declaración de una función puede ocurrir de una de tres formas:

- Mediante una **llamada**: cuando una función es llamada antes de haber sido declarada o definida, se asumirá que devuelve un entero y que sus parámetros tienen por tipo los de los argumentos actuales en la llamada. Este método debe ser evitado, se menciona su existencia para que se sepa que este es un efecto del uso de una función antes de su declaración o definición.
- Mediante **definición** (anterior a la llamada): es correcta, pero si se hace alguna llamada anterior, la llamada pasará a ser la definición.
- Mediante **declaración explícita** (previa a la llamada): se indica explícitamente el prototipo de la función:

```
tipo_valor_devuelto nombre_funcion(lista_tipos_argumentos);
```

Es como la cabecera de la función pero sin dar nombre a los argumentos (se pueden incluir también, pero son ignorados). Si no hay argumentos, se pone entre los paréntesis `void`; si no hay valor de retorno, el tipo del valor devuelto también se indicará como `void`. La idea es agrupar todas las declaraciones de función al principio del código, de modo que nos aseguramos que no hay ninguna llamada previa a las declaraciones.

Veamos como ejemplo la declaración de la función `division_segura`:

```
int division_segura(unsigned int,unsigned int);
```

Respecto a la invocación de una función, consiste en incluir su nombre en una expresión, seguido de los argumentos actuales entre paréntesis y separados por comas (en el mismo orden en que aparecen los formales en la declaración de la función). Los argumentos actuales se evaluarán y una copia del resultado se pasará a la función para que se ejecute, asociando a los nombres de sus argumentos formales las copias de la evaluación de los actuales. El resultado de la función, si es que existe, se utilizará en la expresión en la que aparece la función para continuar con su evaluación.

En C el paso de parámetros es siempre por valor, no existe el paso por referencia. La forma de conseguir el mismo efecto que un paso de parámetros por referencia es gestionar nosotros mismos un mecanismo referencial: los punteros. No podemos pasar un parámetro `int` y que la función modifique su valor directamente, pero sí podemos pasar como parámetro un puntero a `int` de modo que la función pueda acceder a través del puntero al entero propiamente dicho y modificar su valor. ☒ Por ejemplo, queremos implementar una función que multiplica A por B y deja el resultado en C. Observa que en este caso la función no devuelve ningún valor. Como se comentó en la introducción, en C cualquier expresión puede actuar como sentencia. Es el caso de las funciones, independientemente de que devuelvan o no algún valor:

```

void multiplica(int A,int B,int *C) {*C=A*B;}
main(){
    int D=2,E=4,F;
    multiplica(D,E,&F);
}

```

Cuando se pasa un array como argumento actual de una función, en realidad el argumento actual es la dirección del array, es decir, el array como puntero. Esto es lo que se pasará como copia y lo que no podrá ser modificado. De ese modo, el contenido del array no es copiado y sí puede ser modificado desde la función. Del mismo modo, si un array es devuelto como resultado de una función, no se devuelve el array propiamente dicho, sino solo el puntero asociado. Si ese puntero correspondía a un array declarado dentro de la función de modo *automatic* (por defecto), resulta que el array será destruido al acabar la ejecución de la función, por lo que no tiene sentido devolver su puntero como resultado. Sin embargo, si el array está en memoria global (es una variable global o *static*) o dinámica (se estudiará en la sección A.9), sí que tiene sentido devolver su puntero.

A.7.1. Función main con argumentos

Hasta ahora hemos usado funciones *main* sin argumentos y sin sentencia *return*. Ambas cosas pueden ser utilizadas para comunicarnos con el sistema operativo, que es el programa que invoca a nuestro programa como si de una función más se tratase. Si devolvemos un valor mediante *return*, este será utilizado como código de error por el sistema operativo. Lo normal es que sea un número entero. Respecto a los argumentos, sirven para recoger los valores que el usuario teclea en la línea de comandos tras el nombre del programa invocado. Se pueden recoger mediante dos variables habitualmente denominadas *argc* y *argv*. La primera es un *int* que indica el número de palabras introducidas por el usuario tras el nombre del programa. El segundo es un vector de punteros a carácter, es decir, vector de cadenas de caracteres. Almacena en la posición *i* la *i*-ésima palabra introducida en la línea de comandos. ⊗ El siguiente programa muestra por pantalla, con una palabra por línea, la línea de comandos introducida:

```

main(int argc, char *argv[]){ // no se especifica numero de palabras en argv
    int i;
    for(i=0;i<argc;i++) printf("%s\n",argv[i]);
}

```

En este *printf* aparece algo que aún no conocemos, *%s*. Estudiaremos su significado en la sección A.8, página 306.

A.7.2. Funciones como parámetros

Del mismo modo que los arrays son en realidad un puntero a la zona de memoria en que se almacenan sus elementos, una función es un puntero a una zona en que se almacena toda la información necesaria para definirla (código, número de parámetros y tipos, valor devuelto, etc.). Esto nos permite pasar a una función el nombre de otra función como parámetro. ⊗ Veamos un ejemplo: vamos a construir una función que aplica a sus dos

argumentos enteros su tercer argumento. La forma de declarar un puntero a una función que recibe dos enteros y devuelve otro es: `int (*p)(int, int)`. Los paréntesis en torno a `*p` son indispensables, de lo contrario estaríamos declarando una función `p` que devuelve un puntero a entero. Utilizaremos una función `main` con argumentos: al invocar al programa le pasamos como parámetro en la línea de comandos los dos números a operar. Por ejemplo, si llamamos `operar` a nuestro programa⁵:

```
>> operar 3 5
```

Este es el programa:

```
#include <string.h>

int suma(int A,int B){return A+B;}
int mult(int A,int B){return A*B;}
int opera(int A,int B,int (*op)(int,int)){return(*op)(A,B);}

main(int argc, char *argv[]){
    int A,B;
    if (argc<3) return -1; // ERROR: faltan operandos
    A=atoi(argv[1]); B=atoi(argv[2]);
    printf("%d\n",opera(A,B,&suma));
    printf("%d\n",opera(A,B,&mult));
}
```

A.8. Funciones de E/S

En C las funciones de entrada/salida no forman parte del propio lenguaje, sino que se encuentran en una librería estándar que, como ya se ha comentado, es `stdio` (E/S estándar). Se incluye situando al comienzo del programa el comando `#include <stdio.h>` para el preprocesador. En los siguientes apartados estudiaremos las más importantes.

A.8.1. printf

Ya hemos utilizado antes esta función pero aún no hemos explotado toda su potencia. Esta función permite incluir en la cadena que imprime *huecos* a ser rellenados con el contenido de una variable según el formato que se elija. Por ejemplo, si tenemos una variable contador `cont` y queremos escribir "el valor del contador es... \n" donde en los puntos suspensivos escribimos su valor en cada ocasión, utilizaremos la sentencia `printf("el valor del contador es %d\n", cont)`. El hueco es `%d`. Todos los huecos empiezan por el signo `%`. Lo que siga después depende del formato que se quiera dar a la impresión de la variable. Por ejemplo, `d` o `i` para un entero en formato decimal, `s` para un string y `f` para un real en notación decimal. Tras el fin de la cadena, separados por comas, se escriben las expresiones de las que se va a tomar el valor para llenar los huecos, en orden según su correspondencia con los huecos. Por ejemplo, para escribir la hora a partir de las variables `horas`, `minutos` y `segundos`:

⁵Es este programa se utiliza la función `atoi` que obtiene un entero a partir de una cadena de caracteres. Se utiliza la librería `string` que contiene múltiples funciones para manejo de cadenas de caracteres.

```
printf("Son las %d:%d:%d\n", horas,minutos,segundos)
```

Para más detalles sobre esta función consulta la página de `man`. `man` es un programa de Linux que nos permite consultar las páginas del manual de comandos del sistema, programas, etc. Podemos utilizarlo también para obtener información sobre sentencias de C. Por ejemplo, para obtener información sobre `printf` escribe en la línea de comandos `>> man printf`. Para saber más sobre `man` teclea `>> man man`.

Otra función muy útil es `sprintf`. Actúa como `printf` pero en lugar de escribir en la salida estándar, genera la cadena de caracteres sobre un array que se le pasa como primer parámetro. Por ejemplo:

```
char cadena[30];
int horas=22,minutos=10,segundos=30;
sprintf(cadena,"Son las %d:%d:%d\n",horas,minutos,segundos);
```

A.8.2. `scanf`

La función `scanf` se utiliza para leer datos de la entrada estándar (por defecto el teclado). Se parece a `printf` en cuanto a que también se utiliza una cadena de caracteres con huecos asociados a variables. En este caso, la idea es tratar de hacer coincidir lo que se encuentra en la entrada con la cadena con huecos. En la medida en que vayan coincidiendo, se irán asociando huecos a variables. Cuando ya no haya coincidencia o cuando ya se haya recorrido con coincidencia toda la cadena con huecos, la función termina y devuelve un entero que indica el número de emparejamientos hueco-variable realizados con éxito. La forma general de `scanf` es

```
int scanf(cadena_con_huecos,&var_1,&var_2,...,&var_n)
```

Las variables que han de recibir los valores resultantes de los emparejamientos se han de pasar por referencia (se pasan sus direcciones), ya que `scanf` necesitará escribir en ellos; a excepción de si se trata de cadenas de caracteres, ya que las cadenas son tablas y por tanto ya son punteros.

Respecto a los identificadores de formato para los huecos, coinciden con los de `printf` en gran medida, aunque con algunas consideraciones adicionales en algunos casos. Una cuestión a tener en cuenta: un hueco `%s`, cadena de caracteres, se rellena con los caracteres encontrados hasta el primer separador (espacio en blanco, tabulador o intro). De modo que, si queremos leer más caracteres, habrá que utilizar algún otro método. Consulta la página de `man` sobre `scanf` para más información sobre formatos⁶.

Veamos algunos ejemplos del funcionamiento de `scanf`:

- Ante una entrada "27 Pepe Juan" las sentencias

```
int i,n;
char cadena[30];
n=scanf("%d%s",&i,cadena);
```

⁶Existe una función `sscanf` con un papel respecto a `scanf` análogo al de `sprintf` respecto a `printf`.

tendrían como resultado *n* igual a 2, porque se han rellenado 2 huecos con éxito, *i* igual a 27 y *cadena* igual a "Pepe" ("Juan" no se incluye en la cadena asociada al hueco %s).

- Ante la misma entrada, las sentencias:

```
int i,j,n;
n=scanf("%d%d",&i,&j);
```

tendrían como resultado *n* igual a 1, porque se ha rellenado solo el primer hueco con éxito, *i* igual a 27 y *j* sin asignar (indefinido).

A.8.3. putchar y getchar

En stdio contamos con dos funciones⁷, que nos permiten leer un único carácter de la entrada estándar o escribirlo a la salida estándar. Son putchar y getchar. Si declaramos char *c*, tenemos las siguientes equivalencias:

- putchar(*c*) equivale a printf("%c",*c*)
- getchar(*c*) equivale a scanf("%c",&*c*)

A.8.4. Manejo de ficheros

Hasta ahora hemos visto funciones que trabajan sobre la entrada o salida estándar (o sobre arrays de caracteres en su lugar). Ahora vamos a estudiar las funciones de E/S relacionadas con ficheros. Para utilizar un fichero, en primer lugar hay que abrirlo. Esto se hace mediante la función fopen. Para utilizarla, primero necesitamos declarar un puntero a archivo (tipo FILE, declarado en stdio): FILE *f. Este puntero recoge el resultado de la función fopen, el puntero al fichero abierto si la apertura fue exitosa o el puntero nulo (NULL, con valor numérico 0) si hubo algún problema. Los argumentos que recibe son el nombre del fichero a abrir (como cadena de caracteres) y el modo de apertura (también una cadena):

```
f = fopen(nombre_fichero,modo);
```

El modo puede ser: r, *read*, solo lectura; w, *write*, escritura; o a, *append*, escritura desde el final del archivo (añadir). Si vamos a abrir un fichero binario, añadimos al carácter elegido de entre los tres anteriores la letra b en la cadena de modo. Por ejemplo "rb" para leer un fichero binario. En otro caso se supone que el fichero es de texto. Cuando terminamos de utilizar un fichero, hemos de cerrarlo mediante la función fclose, que recibe como parámetro el puntero al fichero. ⊗ Un ejemplo de secuencia de operaciones sobre un fichero de texto a leer:

⁷En realidad son macros, concepto que veremos en la sección A.10, página 312, pero podemos considerarlas funciones a todos los efectos.

```

main(){
    FILE *f;
    f = fopen("datos.txt","r");
    if (!f) printf("ERROR: no se pudo abrir el fichero\n");
    else{
        ... // (operaciones sobre el fichero)
        fclose(f);
    }
}

```

Una vez que sabemos abrir y cerrar un fichero, vamos a estudiar las funciones que podemos utilizar sobre un fichero abierto. Por ejemplo, para un fichero de texto podemos utilizar las funciones `getc` y `putc` que leen y escriben un único carácter. Sus cabeceras son:

```

int getc(FILE *fp)
int putc(int c, FILE *fp)

```

`getc` devolverá la constante `EOF` si ocurre algún error, incluyendo el hecho de que ya se haya leído el último carácter y no queden más por leer. Esta última situación también es detectable mediante la función `feof`, que indica si el fichero cuyo puntero recibe como parámetro ha llegado a su fin.

`fgets` y `fputs` permiten leer o escribir una línea completa, consulta `man`. Las funciones `fprintf` y `fscanf` son análogas a `printf` y `scanf`, con la única diferencia de que se les añade un primer parámetro en el que reciben el puntero al fichero y su acción es ejecutada sobre el fichero en lugar de sobre la salida o entrada estándar. Por ejemplo, podemos escribir una cadena en un fichero apuntado por `f` mediante la sentencia `fprintf(f, "cadena")` o leer un entero mediante `fscanf(f, "%d", &i)`, siendo `i` declarado como `int i`.

⊗ Para acceder en modo directo (binario) a un fichero, podemos utilizar las funciones `fwrite` y `fread`. Permiten escribir o leer un bloque de `n` elementos de un mismo tipo de una vez. Son más eficientes que `fprintf` y `fscanf`. A `fread` se le pasa como parámetro el número de objetos a leer (como máximo) y el tamaño en bytes del tipo de objeto. También hay que indicarle mediante un puntero la zona de memoria en que ha de depositar los objetos leídos. Además de dejar los objetos en el lugar indicado, devuelve el número de objetos leídos, que puede ser menor del máximo indicado. Respecto a `fwrite`, se le indica de dónde leer y cuántos objetos (como máximo) hay que escribir y de qué tamaño son. Devuelve también un número indicando cuántos objetos se escribieron. Si es menor que el máximo indicado hubo algún error. Consulta sus páginas en `man` para obtener más información.

⊗ Para acabar la sección, hay que comentar que la entrada y salida estándares están asignadas por defecto al teclado y a la pantalla, pero esto puede cambiarse. Lo mismo ocurre con la salida de error estándar. Para aprender cómo, consulta las páginas de `man` sobre `stdin`, `stdout` y `stderr`.

A.9. Gestión dinámica de la memoria

Hasta el momento hemos considerado solamente datos que ocupan un tamaño conocido en tiempo de compilación, es decir, datos estáticos. Pero a menudo necesitaremos una cantidad de memoria para datos conocida solo en tiempo de ejecución. Por ejemplo, si queremos ordenar un array que leemos desde un fichero, no sabremos el tamaño del array que necesitamos para almacenar los datos del fichero en memoria hasta que conozcamos el fichero concreto. En estos casos se hace indispensable la utilización de *memoria dinámica*. Es memoria que podemos solicitar al sistema de forma explícita durante la ejecución de un programa. Se trata de una memoria tipo *montón* o *heap*. Es importante tener presente que se sitúa en una zona de memoria independiente, de modo que, si desde dentro de una función solicitamos memoria dinámica, al salir de la función la memoria seguirá estando reservada salvo que la liberemos explícitamente. Es decir, la memoria reservada dinámicamente no es gestionada por el sistema como las variables *auto*.

Las funciones que nos permiten reservar memoria dinámica son `malloc`, `calloc` y `realloc`, las tres definidas en la librería `stdlib`. Las dos primeras son muy parecidas.

- `malloc` se declara como `void *malloc(size_t nbytes)`. El tipo `size_t` es en realidad un entero. Devuelve un puntero (sin tipo definido) a la zona de memoria asignada. `nbytes` especifica el número de bytes a reservar. Este número suele ser calculado como el número de elementos multiplicado por el tamaño del tipo, obtenido con el operador `sizeof`. La memoria asignada no se inicializa. Si no se pudo reservar la cantidad solicitada, se devuelve `NULL`.
- `calloc` se declara como `void *calloc(size_t nmemb, size_t size)`. Funciona como `malloc`, con dos diferencias: primera, que se indican por separado número de elementos y tamaño en bytes de estos; y segunda, la memoria reservada es inicializada con ceros.
- `realloc` se declara como `void *realloc(void *ptr, size_t size)`. `ptr` es un puntero a una zona de memoria reservada previamente. `realloc` se encarga de redimensionar la zona reservada a `size` bytes.

Para liberar la memoria reservada con cualquiera de las tres funciones se utiliza la función `free`, pasándole como parámetro el puntero a la zona de memoria a liberar.

⊗ Estudiemos un ejemplo que nos va a ser de utilidad en las prácticas para entender mejor el manejo de la memoria dinámica. Supongamos que tenemos un fichero de texto en el que en cada línea hay una única palabra. Queremos ordenar las líneas alfabéticamente. Para ello vamos a leer las palabras del fichero, situarlas en un array en memoria, ordenar los elementos del array y finalmente escribirlos ordenados en un nuevo fichero. La forma en que vamos a organizar las palabras en memoria es la siguiente:

- Tendremos un array de caracteres llamado `letras` que contiene todos los caracteres del fichero, leídos secuencialmente. Sustituimos los `\n` y el EOF final por `\0` por las razones dadas en el siguiente punto.
- Tendremos un array de punteros a carácter llamado `palabras` en el que el elemento *i*-ésimo apunta a la posición de `letras` en que comienza la palabra *i*-ésima. Por

estar los caracteres de cada palabra en `letras` seguidos por un `\0`, podemos usar estos punteros directamente como cadenas de caracteres a efectos de visualización por pantalla.

Por ejemplo, suponiendo que el contenido del fichero es:

```
Ana
Pedro
Rafa
```

la representación gráfica de la estructura de punteros vendría dada por la figura A.2.

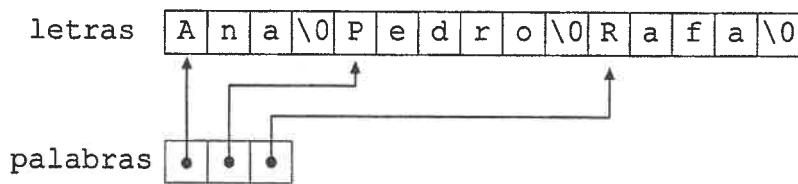


Figura A.2: Estructura de los arrays `letras` y `palabras`.

Una vez que tengamos ambos arrays en memoria, podremos ordenar las palabras mediante la ordenación del array `palabras` sin necesidad de cambiar de lugar los caracteres en sí, manejando únicamente punteros. Pero, ¿cómo construimos dichos arrays? Hasta que no hayamos leído el fichero concreto, no sabremos ni el número de letras (que determina el tamaño de `letras`) ni el número de líneas (que determina el tamaño de `palabras`). Una posibilidad (no la única ni necesariamente la mejor, pero sí sencilla) es la siguiente:

- Abrimos el fichero en modo lectura y damos una primera pasada leyendo carácter a carácter. En esta pasada contamos el número de líneas, `nlin`, y de caracteres, `ncar`.
- Conocidos estos datos, ya podemos reservar espacio para ambos arrays. Se podría hacer del siguiente modo:

```
char *letras;           // puntero a caracter =
                      // = array dinámico de caracteres
char **palabras;       // puntero a puntero a caracter =
                      // = array dinámico de punteros a caracter

letras = (char *) malloc(ncar*sizeof(char ));
palabras = (char **) malloc(nlin*sizeof(char *));
```

Date cuenta de que los arrays para los que se reserva memoria en tiempo de ejecución no se declaran como tales arrays sino como punteros, de modo que no se reserva memoria estática para ellos. Fíjate también en los castings explícitos y en el uso de `sizeof`. El casting sobre el puntero devuelto por `malloc` coincide con el tipo de la variable que ha de almacenarlo. El argumento de `sizeof` es el tipo de elemento a almacenar por el array correspondiente.

- Por último, una vez que tenemos los arrays, podemos leer secuencialmente de nuevo el fichero de entrada. Esta vez iremos copiando los caracteres a letras y rellenando los punteros de palabras a la par para que apunten al comienzo de cada palabra.

⊗ Haz un programa que construya las estructuras descritas a partir de un fichero y que luego muestre por pantalla la lista de palabras en orden contrario a como se encuentran en el fichero.

A.10. El preprocesador de C

El preprocesador es un programa que actúa sobre el fichero fuente antes de la compilación propiamente dicha. Se ocupa de tareas como la inclusión de ficheros, sustitución de macros y compilación condicional atendiendo a los comandos dirigidos a él, caracterizados por estar precedidos por el símbolo `#`. Los más utilizados son `#include` y `#define`.

Cuando el preprocesador encuentra un comando `#include` seguido de un nombre de archivo, el comando es sustituido por el contenido del archivo. El nombre puede ir entrecomillado o entre signos `<>`. En el primer caso, el fichero es buscado primero en el directorio actual y luego en el de librerías. En el segundo caso, directamente en el directorio de librerías. De ese modo, el primer caso es usado normalmente para ficheros escritos por el programador y el segundo para librerías del sistema.

El comando `#define` define una **macro**: una cadena que será sustituida por el preprocesador por otra cosa. Por convención, el nombre de la macro suele escribirse en mayúsculas. Hay dos tipos de macros:

- `#define nombre texto`: cada aparición de `nombre` en el código es sustituida directamente por `texto`. Un uso común es la declaración de constantes, por ejemplo: `#define PI 3.1416`.
- `#define nombre(parametros) texto_con_parametros`: de forma similar al anterior caso, pero ahora el texto que sustituye al nombre de la macro está parametrizado de modo que lo que será escrito en ciertas posiciones depende del parámetro actual en cada utilización de la macro. Por ejemplo: definimos `#define SUMA5(x)` (`x+5`) y luego la utilizamos así: `SUMA5(3)`; esta cadena será sustituida textualmente por `(3+5)`. Los paréntesis se añaden para evitar interacciones indeseadas con el contexto en que aparezca la expresión. Por ejemplo, si definimos `suma5(x)` `x+5` entonces `suma5(a)*suma5(b)` se sustituirá por `a+5*b+5`. Como el operador `*` tiene mayor precedencia que el operador `+` esa expresión es equivalente a `a+(5*b)+5`, no a `(a+5)*(b+5)`, que es la que queríamos.

Es importante tener claro que `#define` provoca tan solo una sustitución textual del nombre de la macro. En ningún caso se están definiendo variables ni funciones.

Los comandos `#ifdef`, `#ifndef`, `#else`, `#endif` y `#undef` permiten el establecimiento de bloques de compilación condicionales, es decir, de bloques de código que serán compilados o no dependiendo de una condición. La condición es siempre que cierta macro esté o no definida. Los primeros cuatro comandos permiten chequear condiciones. `#define` define macros y `#undef` permite eliminar una definición previa. Un uso frecuente de estos

comandos es evitar que un mismo fichero sea incluido varias veces en un mismo programa. He aquí un ejemplo de cómo se puede evitar esta circunstancia:

```
#ifndef INCLUIDO_FICHERO_AUXILIAR
#include "fichero_auxiliar.c"
#define INCLUIDO_FICHERO_AUXILIAR
#endif
```

Otro uso muy común de estos comandos es el de escribir sentencias cuya finalidad es depurar el programa y que por tanto queremos eliminar en la versión definitiva. Podemos escribirlas todas en bloques tipo

```
#ifdef DEPURANDO
... // sentencias de depuración
#endif
```

Al principio del programa definiremos la macro `#define DEPURANDO`. Esto hará que estemos en modo depuración. Para conmutar a modo normal no tenemos más que comentar (o eliminar) esta línea.

A.11. Programas con varios ficheros

En C podemos escribir un programa dividido en varios ficheros fuente. Cada uno de ellos será compilado a fichero objeto por separado (extensión .o) y luego los ficheros objeto resultantes podrán ser enlazados en un ejecutable único. Las librerías son ficheros fuente precompilados que incluimos en nuestro ejecutable final.

Explicaremos cómo crear un programa de este tipo mediante un ejemplo. Supongamos que definimos las siguientes funciones aritméticas:

```
/* arit.c */
int suma(int A,int B){return A+B;}
int mult(int A,int B){return A*B;}
```

y queremos situarlas en un fichero aparte. Será un fichero de funciones aritméticas y lo denominaremos `arit.c`. Su contenido es exactamente lo escrito arriba. Ese fichero no puede compilarse a un ejecutable directamente porque no tiene función `main`. Pero sí que puede compilarse a fichero objeto. Esto se hace mediante la opción de compilación `-c`, solo compilar, sin enlazar para generar ejecutable. La orden sería: `gcc -c arit.c`, tras la cual podemos observar que se ha generado un fichero `arit.o`. Si ahora hacemos un programa, en un fichero llamado `prog.c`, y queremos usar estas funciones en él, debemos declararlas en `prog.c`. Lo que se hace habitualmente es generar, para cada fichero independiente con funciones, un fichero con sus declaraciones o cabeceras. Este fichero tiene el mismo nombre que el fichero de código, pero con extensión `.h` (de *header*, cabecera). En cada fichero en que se usen las funciones definidas en otro se incluirá el fichero de cabeceras correspondiente. De este modo se evita tener que declarar de nuevo las funciones en cada fichero que las use. Este sería el contenido de `arit.h`:

```
/* arit.h */
int suma(int,int);
int mult(int,int);
```

Ahora escribimos nuestro programa, por ejemplo una pequeña calculadora que pide dos números y devuelve su suma y producto. No haremos ningún tipo de control de error sobre la entrada introducida. Llamaremos a este programa calc. El fichero calc.c sería:

```
/* calc.c */
#include <stdio.h>
#include "arit.h"
main(){
    int a,b;
    printf("Introduce operando 1: "); scanf("%d",&a);
    printf("Introduce operando 2: "); scanf("%d",&b);
    printf("%d+%d=%d\n",a,b,suma(a,b));
    printf("%d*%d=%d\n",a,b,mult(a,b));
}
```

Fíjate en el uso diferente de comillas y <> en los #include. Y fíjate en el significado de la inclusión del fichero stdio.h: contiene las declaraciones de las funciones de la librería, cuyas definiciones están ya compiladas. Fíjate también en un detalle: la cadena recibida por scanf para extraer de ella un entero no es pasada a esta función hasta que se pulsa *intro*. Y una vez pulsado, el retorno de carro correspondiente es escrito a la salida estándar, de modo que no es necesario utilizar \n para provocar el salto de línea.

¿Cómo generamos ahora el ejecutable calc? Tenemos que comprender el proceso llevado a cabo por gcc cuando es invocado. Aquí daremos una explicación simplificada para entenderlo al nivel de lo que nos interesa. El proceso consiste en tomar cada uno de los ficheros que recibe como parámetro y si es fuente compilarlo a objeto (si ya es objeto, no hace nada con él); a continuación se enlazan los objetos y produce el ejecutable (salvo utilización de opciones como -c). Para obtener el ejecutable de un programa se han de incluir como parámetros de gcc, ya sea como ficheros fuente o como objetos, todos los ficheros que forman parte el programa. Por ejemplo, para nuestra calculadora podríamos obtener el ejecutable con la orden:

```
>> gcc -o calc calc.c arit.c
```

En este caso estaríamos volviendo a compilar a objeto arit.c. Podríamos evitar repetir este proceso mediante la orden:

```
>> gcc -o calc calc.c arit.o
```

que es equivalente a la secuencia de órdenes:

```
>> gcc -c calc.c
>> gcc -o calc calc.o arit.o
```

Reutilizar los objetos ya compilados puede ser importante, sobre todo en programas grandes en los que la compilación de cada uno de ellos lleva mucho tiempo. Pero hay que tener presente, cada vez que generamos el ejecutable, que puede que hayamos hecho algún cambio en un fichero fuente y en ese caso su fichero objeto debe ser generado de nuevo. Llevar en cuenta qué ficheros han sido modificados y cuáles no puede llegar a convertirse en una tarea complicada.

Para evitar este tipo de complicaciones, podemos utilizar la orden make. Se trata de un programa de uso general del que aquí explicaremos solo una pequeña parte, para

más información consulta `man`. Permite la generación de ficheros objetivo a partir de ficheros **prerrequisito o dependencias**. En nuestro ejemplo, para generar el ejecutable necesitamos los dos objetos y para cada objeto necesitamos su fichero fuente. Esta estructura sería expresada así en un fichero que vamos a llamar `makefile`:

```
# esto es un comentario

calc: calc.o arit.o
<TABULADOR>gcc -o calc calc.o arit.o

calc.o: calc.c
<TABULADOR>gcc -c calc.c

arit.o: arit.c
<TABULADOR>gcc -c arit.c
```

La inclusión del tabulador donde se indica es obligatoria. Cada pareja de líneas constituye un **objetivo**. Lo que hay antes de ':' es la etiqueta que lo identifica y lo que hay detrás expresa el conjunto de dependencias: los ficheros que se necesitan para conseguir el objetivo. Tras el tabulador, se escribe la orden que produce el objetivo a partir de las dependencias. Por ejemplo, las dos primeras líneas tras el comentario significan: "para conseguir el fichero `calc` necesito los ficheros `calc.o` y `arit.o`, y se debe ejecutar el comando `gcc -o calc calc.o arit.o`".

La orden `make` busca por defecto un fichero llamado `makefile`, dentro del cual busca por defecto el primer objetivo y trata de generarlo a partir de la información dada en el fichero. Sigue la cadena de dependencias hasta llegar a nuevos objetivos cuyas dependencias son definidas de nuevo como objetivos y así sucesivamente. Si la marca de tiempo (fecha y hora de la última modificación) de alguna dependencia es posterior a la del objetivo, entonces ese objetivo debe ser recompilado. Esta recompilación hará cambiar su marca de tiempo, lo cual llevará a una recompilación de cualquier objetivo que dependa de él. De este modo, `make` gestiona de forma automática la recompilación de ficheros fuente modificados. Para generar un objetivo distinto al primero, la orden a invocar es `make objetivo`.

Ejercicios Propuestos

Ejercicio A.1 Haz un programa que, dado un fichero de texto, cambie de mayúsculas a minúsculas y viceversa las letras dadas en una tabla. El nombre del fichero será un parámetro dado por el usuario en la línea de comandos. La tabla de letras a cambiar será recogida de teclado mediante sentencias `scanf`.

Ejercicio A.2 Haz un programa que genere en un fichero ASCII un tablero de ajedrez con piezas. Las dimensiones del tablero y las piezas a colocar y posiciones donde hacerlo serán elegidas por el usuario durante la ejecución. Utiliza un tipo enumerado `pieza` para identificar qué pieza hay en cada casilla (puedes incluir un valor vacío para indicar que no hay ninguna). Define con `typedef` un tipo `casilla`, utilizando `struct`, y úsalo para manejar posiciones en el programa. Un ejemplo de fichero generado podría ser este (usamos

* para casilla vacía, Y para rey, y la primera letra de sus nombres para reina, torre, alfil, caballo y peón):

```
* T * A * * R
* * * * * *
P * C * * * *
* * * * * Y *
```

Ejercicio A.3 Programa el método de la *criba de Eratóstenes* para la determinación de qué números entre 1 y n son primos. n es una entrada al programa, dada como parámetro de línea de comandos al invocarlo. Este método consiste en lo siguiente:

- Se crea una tabla con los números impares desde 3 a n y una lista de primos que inicializamos con el 2.
- En una primera pasada se "tachan" de la tabla el 3, el 6... y así sucesivamente todos los múltiplos de 3, y se añade el 3 a la lista de primos.
- En una segunda pasada se busca el primer elemento no tachado de la tabla (será el 5), se añade a la lista de primos, y se tachan de la tabla el 5, el 10 y así todos los múltiplos de 5.
- El punto anterior se repite hasta que no queden elementos sin tachar en la tabla. En este momento en la lista tendremos todos los primos que había entre 1 y n . Tomar inicialmente los números impares significa dar un primer paso implícito en que se tachan todos los múltiplos de dos, por eso se introduce el 2 directamente en la lista de primos encontrados.

Ejercicio A.4 Haz un programa que multiplique las n matrices dadas en un fichero de texto. Las dimensiones de las matrices no son indicadas explícitamente. Tampoco se puede asumir que estas dimensiones son coherentes, es decir, que son tales que para dos matrices consecutivas cualesquiera A y B , de dimensiones $a_1 \times a_2$ y $b_1 \times b_2$ respectivamente, ocurra $a_2 = b_1$. El programa deberá analizar esta circunstancia y abortar su ejecución y dar un mensaje de error si la detecta.

Ejercicio A.5 Haz un programa que genere recursivamente la fila n -ésima del *triángulo de Tartaglia*. El elemento i -ésimo de la fila n -ésima de ese triángulo corresponde al número combinatorio $\binom{i}{n}$, con $i \in [1, n]$. Los valores de una fila se pueden obtener sumando los dos valores encima de él en la anterior, o el único valor sobre él si se trata de un elemento en un extremo de la fila. La primera fila (fila 0) tiene un único elemento con valor 1. El resto pueden generarse recursivamente a partir de esta. Como ejemplo se muestran las cinco primeras filas del triángulo:

$$\begin{array}{cccccc} & & & 1 & & \\ & & & 1 & 1 & \\ & & & 1 & 2 & 1 \\ & & & 1 & 3 & 3 & 1 \\ & & & 1 & 4 & 6 & 4 & 1 \end{array}$$

Referencias bibliográficas

Son muchos los libros disponibles para profundizar en el aprendizaje del lenguaje C. Recomendamos en particular [Kernighan91], escrito por los creadores del lenguaje.

Apéndice B

Introducción básica a C++

Contenido del capítulo:

B.1.	Pequeños cambios respecto a C	320
B.1.1.	Variables y tipos	320
B.1.2.	Funciones	321
B.1.3.	Operadores new y delete	323
B.1.4.	Gestión de E/S	324
B.2.	Introducción a la POO	324
B.3.	Clases y objetos en C++, primeros pasos	325
B.4.	Programa ejemplo: TAD cadena de caracteres	327
B.5.	Plantillas	333
B.6.	Excepciones	335
	Ejercicios Propuestos	337
	Referencias bibliográficas	338
C.1.	Enunciado	339
C.2.	Ánalisis de los requisitos de la aplicación	341
C.3.	Diseño del programa	343
C.4.	Implementación del diseño	348
C.5.	Validación y verificación	349
C.6.	Informe del desarrollo	350

C++ es, salvo por algunos detalles, un superconjunto de C. O dicho al revés, C es un subconjunto de C++: cualquier programa en C es también un programa válido en C++. Es decir, C++ añade cosas a C (y modifica un poco algunas). Algunas de las cosas añadidas son pequeños detalles que hacen más cómoda o eficiente la programación; serán comentados en la siguiente sección. Pero el gran cambio lo constituye la adición a la programación procedural de C – centrada en las funciones– de características de *programación orientada a objetos (POO)*, que tienden a centrar más la programación en los datos. El objetivo de esta pequeña introducción a C++ es, sobre la base del recién aprendido C, explicar las características mínimas de C++ que nos permitan utilizarlo para definir **tipos abstractos de datos (TAD)**. No entraremos en muchos de los detalles de este complicado lenguaje; en particular no entraremos a fondo en la POO, y sobre todo no entraremos en absoluto a hablar de la *herencia*, concepto fundamental en POO. Todas estas cuestiones son abordadas más adelante con más profundidad en asignaturas específicas.

B.1. Pequeños cambios respecto a C

Los ficheros de C++ tienen extensión .cpp en lugar de .c. La distinción es importante, ya que el compilador se fijará en la extensión para determinar si está ante un fichero escrito en C o en C++ y por tanto para decidir cómo ha de hacer la compilación. En general las modificaciones son ayudas a la programación, para realizar ciertas tareas de forma más eficiente. La forma antigua de realizar la misma tarea en C se permite también en C++ para permitir la compatibilidad hacia atrás, de modo que C++ es casi totalmente compatible con C.

B.1.1. Variables y tipos

Un cambio importante introducido en C++ es la aparición del tipo `bool` como tipo predefinido del lenguaje. Puede adoptar dos valores, `true` o `false`. Un valor de tipo `bool` ocupa un byte de memoria. Los booleanos pueden ser utilizados del mismo modo que en C cuando no constituían un tipo independiente: pueden ser utilizados como valor numérico (`true` equivale a 1 y `false` a 0) o pueden recibir un valor numérico (0 se interpretará como `false`, cualquier otro valor como `true`).

En C++ podemos declarar una variable cuyo tipo sea una estructura utilizando simplemente el nombre de la estructura definida, sin necesidad de añadir la palabra `struct`. Es decir, podemos escribir `persona P` en lugar de `struct persona P`.

C++ nos permite declarar una variable en cualquier punto del código, no necesariamente al principio de un bloque, antes de las sentencias, como en el caso de C. De este modo podemos retrasar la declaración de una variable hasta el punto en que va a ser utilizada por primera vez.

Tanto en C como en C++ contamos con el calificador `const` para variables. Para C no fue explicado de modo que ahora en lugar de ver las diferencias respecto a C hablaremos directamente de en qué consiste para C++. Cuando la declaración de un variable es precedida por esta palabra la variable pasa a ser constante: su valor no podrá ser modificado,

ni directamente (la variable aparece a la izquierda en una asignación) ni indirectamente (a través de punteros a la memoria en que se almacena). Este tipo de variables han de ser inicializadas en la propia declaración. Un ejemplo de declaración con inicialización es: `const float pi=3.1416.` Este tipo de constantes es diferente de las macros establecidas mediante `define`: con `const` tenemos variables que no cambian, pero sujetas a todas las consideraciones aplicables a variables, como visibilidad y duración; mientras que un `define` lleva simplemente a una sustitución textual por parte del precompilador.

El calificador `const` puede ser también aplicado a un puntero. Esto puede hacerse de dos maneras:

- Se puede declarar un puntero a una variable `const`, es decir, un puntero a una variable que no podrá ser modificada, pero sí que permitimos que el puntero cambie y apunte a otras variables. Esto se hace anteponiendo `const` a toda la declaración. Por ejemplo: `const char *nombre="Pepe".`
- También podemos definir un puntero `const` que apunta a una variable de cualquier tipo. Por ser el puntero una variable `const` solo puede ser inicializado en su declaración y no podrá apuntar a ninguna otra variable a lo largo de la ejecución del programa. Esta segunda alternativa se consigue introduciendo la palabra `const` entre el tipo y el nombre de la variable: `char* const nombre="Pepe".`

En C++ se introduce el operador de resolución de visibilidad (`::`). Su cometido es permitir el acceso a una variable global desde el interior de un bloque en que esta no es visible por haber sido ocultada por la definición de una variable local con el mismo nombre. Este operador no nos permitirá, por contra, acceder a otra variable local ocultada por otra local en un bloque anidado. He aquí un ejemplo del uso de este operador. Al final de la ejecución a valdrá 1 y b valdrá 2:

```
int i=2;
main(){
    int i=1,a,b;
    a=i;
    b=::i;
}
```

Para acabar esta sección, comentaremos que en C++ se introduce una nueva sintaxis para efectuar una conversión explícita de tipo (casting). El equivalente a una expresión `(float) 2` mediante esta nueva notación sería `float(2)`.

B.1.2. Funciones

En C++ se introduce un concepto fundamental, el de la sobrecarga de funciones. Consiste en que se pueden definir varias funciones distintas con el mismo nombre, siempre y cuando se diferencien en el número o tipo de argumentos. El tipo del valor devuelto no puede ser nunca la única diferencia entre dos versiones de una función, necesariamente estas dos versiones deberán tener alguna diferencia en el número o tipo de los argumentos que la función recibe. En el momento de la llamada se distinguirá automáticamente a qué función nos estamos refiriendo en cada caso mediante un análisis del número y tipo

de parámetros actuales. Por ejemplo, podríamos definir una función `menos` que cuando tiene un único argumento `int` devuelve el parámetro actual con el signo cambiado; en cambio, cuando tiene dos devuelve la diferencia entre ambos:

```
int menos(int a,int b) {return a-b;}
int menos(int a) {return -a;}
main(){
    int c,d;
    c=menos(2); // c = -2
    d=menos(9,6); // d = 3
}
```

Otro concepto nuevo que aparece en C++ es de **operador** definido por el usuario: podemos definir métodos que puedan aplicarse en forma de operador en lugar de como función con parámetros entre paréntesis. Los operadores son funciones a todos los efectos, son solamente una comodidad sintáctica proporcionada al programador por el lenguaje. Se declaran usando la palabra `operator` seguida del símbolo que queramos utilizar como operador. El conjunto de símbolos disponibles es el siguiente:

+	-	*	/	%	^	&		~	!	=	<	>	+=	-=	
*=	/=	%=	^=	&=	=	<<	>>	>>=	<<=	==	!=	<=	>=	&&	
	++	--	->*	,	->	[]	()	new		delete					

No se pueden modificar las reglas de precedencia o asociatividad de los operadores, ni tampoco alterar su sintaxis. Por ejemplo, no se puede definir un + unario.

Un operador binario puede definirse mediante una función miembro¹ que reciba un argumento (además del implícito) o bien como una función global que reciba dos argumentos. Por ejemplo, podemos definir el operador + para una clase `complejo` de modo que `a+b` se interprete como `a.operator+(b)` o bien como `operator+(a,b)`. Esto supone una forma de sobrecarga. Si se definen ambas funciones, ante la utilización del operador se elegirá una u otra según qué argumentos actuales se ajusten mejor a los formales de una de las dos funciones definidas. Si ninguna de las opciones es mejor que la otra, entonces la llamada se considerará ilegal. Algo parecido ocurre con los operadores unarios, que pueden ser interpretados como una función miembro sin argumentos o una global con un argumento. Para más información consulta el manual de C++.

Otra de las características introducidas en C++ es la posibilidad de especificar valores por defecto para los parámetros formales de una función y, en consonancia, la posibilidad de invocar una función con menos parámetros actuales que parámetros formales tiene. Solo se pueden omitir parámetros actuales del final y las omisiones han de corresponder a parámetros consecutivos, es decir, se suprime todos los parámetros a partir del parámetro i-ésimo. Para los parámetros que faltan se tomarán los valores por defecto. Este es un ejemplo de cómo se declaran los valores por defecto. Se trata de un función que calcula la raíz cuadrada de su primer parámetro. El segundo indica si queremos la positiva o la negativa, si no lo especificamos se dará la positiva por defecto. Usamos la función `sqrt` de la librería `math`:

¹Para entender este párrafo necesitas los conocimientos explicados en las dos secciones siguientes. Puedes saltarlo y volver sobre él más adelante si aún no conoces los conceptos utilizados.

```
int raiz_cuadrada(int numero, int negativa=0) {
    if (!negativa) return sqrt(numero);
    else return -sqrt(numero);
}
```

Para acabar esta sección hablaremos sobre una novedad importante en C++: la existencia del tipo **referencia** y el **paso de parámetros por referencia**. Una variable referencia es una referencia a otra variable, un alias, otra forma de acceder al contenido de la primera variable. Por ejemplo, una variable **i2** que sea una referencia a una variable entera **i** se declara como: **int& i2=i**. A partir de ese momento cualquier acción sobre **i** tiene efecto tanto en **i** como en **i2** y viceversa. Vemos un cierto parecido entre las referencias y los punteros, pero hay dos diferencias básicas: primero, una referencia se asocia a una variable en su declaración y esa asociación ya no puede ser cambiada; y segundo, al emplear la referencia nos estamos refiriendo directamente a la variable a la que se asocia, sin necesidad de utilizar el operador indirección, *****. Veamos en este programa la diferencia entre una referencia y un puntero:

```
main(){
    int i,j,*p;
    int& i2=i; // i2 referencia a i
    i2 = 3;    // i=3, asignacion por referencia o alias
    p = &i;    // p recibe la direccion de i
    *p = 4;    // i=4, asignacion a traves de puntero
    p = &j;    // p recibe otra direccion, la de j
    p = i2;    // ERROR, detectado por compilador: error de tipos,
                // asignacion de un entero a un puntero a entero
}
```

El principal uso de las variables referencia es para el paso de parámetros por referencia. Lo que hacíamos en C mediante el uso de puntero de forma explícita puede hacerse ahora de una forma más sencilla mediante el uso de referencias. Por ejemplo, reescribimos la función **multiplica** de la página 304 con referencias de este modo:

```
multiplica(int A,int B,int& C) {C=A*B;}
main(){
    int D=2,E=4,F;
    multiplica(D,E,F);
}
```

Fíjate en que ahora **F** se pasa directamente como parámetro, sin aplicarle el operador dirección (**&**). Cuando se produzca la llamada la variable referencia **C** quedará asociada como alias a **F**, de modo que todo lo que se haga con **C** se estará haciendo también con **F**, todo ello de un modo transparente, sin usar notación de punteros.

B.1.3. Operadores new y delete

En C++ se introducen dos operadores, **new** y **delete**, que realizan funciones equivalentes a **malloc** y **free** y las amplían. La ampliación consiste en que no se limitan a reservar una cantidad de memoria y luego liberarla sino que en realidad **new** crea una

variable del tipo que se le indique como parámetro y `delete` la destruye. Cuando el tipo no sea un tipo sino una *clase*, concepto que será descrito en breve, esto implicará una llamada automática a una función especial denominada *constructor*, en el caso de `new`, y a otra denominada *destructor* en el caso de `delete`. El puntero devuelto por `new` es directamente del tipo de la variable creada, no es necesario un casting como el que se utilizaba con `malloc`. La sintaxis para la creación y posterior destrucción de una variable es:

```
int *p;
p=new int;
delete p;
```

Para un vector de 3 variables del mismo tipo la sintaxis sería:

```
int *p;
p=new int[3];
delete [] p;
```

B.1.4. Gestión de E/S

En C++ se introduce una nueva forma de utilizar los flujos de entrada o salida, es decir, los ficheros o la E/S estándar. Para la E/S estándar, además de los ya conocidos `printf` y `scanf` podemos utilizar los operadores `cin` y `cout`, para lo cual hay que incluir la librería `iostream`. El equivalente a:

```
printf("La suma de %d y %d es %d\n",a,b,c);
```

sería

```
cout << "La suma de " << a << " y " << b << " es " c << "\n";
```

De forma similar, el equivalente a

```
scanf("%s",&cadena);
```

sería

```
cin >> cadena;
```

Fíjate en que no es necesario indicar el tipo de las variables a imprimir o en las que se recogen datos desde la entrada. La asignación de tipos se hace de forma automática en tiempo de ejecución.

Con ficheros se trabaja de forma similar. En este caso habría que incluir la librería `fstream`. Para más detalles, consulta el manual una vez hayas acabado de estudiar este capítulo.

B.2. Introducción a la POO

Un **objeto** es un ente que contiene datos y que tiene asociadas ciertas funciones que pueden actuar sobre sus datos. Las funciones asociadas a un objeto se denominan *métodos*.

Por ejemplo, podríamos tener un objeto `panel_turno` que modela el panel que informa sobre el turno en curso en el sistema de ticket de una charcutería. Este objeto podría tener un dato, el turno actual, y tres funciones asociadas: una que lo ponga a cero, `reset`; otra que lo haga avanzar una unidad, `avance`; y una tercera que nos permita conocer el turno actual, `turno_actual`. Un objeto puede interactuar con otros objetos mediante paso de mensajes. Este paso de mensajes consiste en la invocación de una de las funciones asociadas al objeto con el que nos queremos comunicar. El mensaje de respuesta será el resultado devuelto por esa función. Por ejemplo, podríamos tener un objeto `cliente` que modele a un cliente de la charcutería. Ese objeto tendría un dato que es el turno que le ha sido asignado, `miturno`. Para saber si le toca ya hacer su pedido o no, debe comparar su turno con el turno actual, para lo cual debe consultar cuál es el turno actual. Para ello puede invocar la función `turno_actual` asociada al objeto `panel_turno`. Recibirá como respuesta el turno actual. Envío un mensaje y recibió una respuesta.

En C un programa en ejecución era la ejecución secuencial de una serie de instrucciones. En POO un programa en ejecución se considera más como la interacción de una serie de objetos pasándose mensajes. El objeto acapara el protagonismo. ¿Qué ocurre si tenemos varios objetos idénticos? Por ejemplo, si tenemos varios clientes en la charcutería. En realidad lo que tenemos son varias instancias de un mismo tipo de objeto. Se trata de la misma relación que existe entre las variables y su tipo de dato. Por así decirlo el tipo describe lo que son (valores que pueden tomar o posibles *estados*) y cómo se comportan (operaciones) las variables de ese tipo. En el caso de los objetos un tipo de objeto se denomina **clase**. Una clase describe los datos que almacena un objeto (valores que puede tomar) y los métodos aplicables sobre esos datos (cómo se comporta un objeto de esa clase). Una vez definida la clase se pueden declarar instancias de ella, los objetos. Por ejemplo, en el caso de la charcutería, declararíamos una clase `cliente` y posteriormente declararíamos tantos objetos cliente como sea necesario. La clase define que cada uno tendrá una variable `turno`, variable que será distinta para cada cliente, para cada objeto, de modo que cada objeto tendrá su propio estado.

B.3. Clases y objetos en C++, primeros pasos

En C++ las clases se definen sobre la base de las estructuras. Una clase es una estructura sobre la que además de los datos miembro o atributos se definen métodos (funciones) miembro que pueden actuar sobre esos datos. Cuando un miembro de una clase, ya sea un atributo o un método, es visible desde el exterior se dice que es *público* y cuando no *privado*. Si utilizamos la palabra `struct` para definir la clase todos sus miembros serán públicos por defecto. Podemos sustituir esta palabra por otra nueva, `class`, que hará que los miembros sean privados por defecto. Normalmente utilizaremos esta segunda opción, es más segura. Hacer privados los miembros tiene que ver con el principio de *encapsulación*: esos miembros no serán accesibles desde el exterior directamente, sino únicamente a través de métodos miembro públicos que actuarán de *interface* entre los miembros privados del objeto y quien quiera acceder a ellos desde el exterior. En la misma línea, lo normal es no incluir la definición de los métodos en la propia clase, sino solo su declaración, de modo que la definición quede oculta (podríamos compilar las definiciones y entregar sólo

el fichero objeto junto con el .h con las declaraciones).

Vamos a estudiar un primer ejemplo de definición de clase y métodos miembro. Se trata de la clase `panel_turno` descrita en la sección anterior. Por simplicidad trabajaremos con un único fichero, `turno.cpp`. Fíjate en que la extensión utilizada es `.cpp` en vez de `.c`. En él aparecen algunas cuestiones que aún no han sido explicadas. Se comentan a continuación:

```
#include <iostream.h>

class panel_turno{
private:
    int turno;
public:
    void reset();
    void avance();
    int turno_actual();
};

panel_turno::reset()      {turno=0;}
panel_turno::avance()     {turno++;}
panel_turno::turno_actual() {return turno;}

main(){
    panel_turno P;
    P.reset(); cout << "Reset\n";
    cout << "El turno actual es: " << P.turno_actual() << "\n";
    P.avance(); cout << "Avance\n";
    P.avance(); cout << "Avance\n";
    cout << "El turno actual es: " << P.turno_actual() << "\n";
}
```

En primer lugar, tras el `#include`, encontramos la definición de la clase `panel_turno`. Fíjate en que se han incluido en su definición las palabras `private` y `public`, seguidas de dos puntos. Establecen dos secciones dentro de la definición. Los miembros declarados en la primera serán públicos y los declarados en la segunda privados. Los miembros declarados son: el atributo `turno`, de tipo `int`, privado; y tres métodos, públicos. Tras la definición de la clase hemos incluido la definición de sus métodos.² Fíjate en que en las definiciones de los métodos miembro de una clase sus nombres son precedidos por el nombre de la clase seguido de `::`, el *operador de resolución de visibilidad*. Fíjate también en que estos métodos no tienen parámetros y en que en su cuerpo se accede a la variable `turno`. En los métodos miembro de una clase siempre hay un *argumento implícito* que es el objeto de esa clase sobre el que se ha *aplicado* el método. Se puede acceder directamente a los atributos de dicho objeto implícito desde el cuerpo del método. ¿Cuál es ese objeto implícito? En la función `main` se declara en la primera línea un objeto de clase `panel_turno`. En la segunda se aplica el método `reset` a ese objeto mediante la misma

²Normalmente se situarían, según se ha comentado antes, en un fichero aparte, que se distribuiría como objeto para ocultar la implementación de los métodos.

notación punto usada para acceder a los atributos. La aplicación consiste en la invocación del método con el objeto en cuestión como objeto implícito. Fíjate también en que por ser *turno* un atributo privado es imposible una sentencia como *P.turno=2*. \otimes Analiza el resto del programa para ver qué está ocurriendo y comprueba el resultado de la ejecución.

B.4. Programa ejemplo: TAD cadena de caracteres

En esta sección implementaremos un TAD cadena de caracteres muy simple que nos servirá para ilustrar cómo hacer un TAD en C++ y también nos permitirá la introducción de algunos nuevos conocimientos sobre el lenguaje. La especificación informal sería:

TAD Cadena es Crear, Esvacia, Addcar, Delcar, Concat, Numcars, Elem

Descripción

Los valores de tipo Cadena son cadenas de caracteres, es decir, secuencias $x_1x_2\dots x_n$ donde x_i es un Carácter, $i \in [1, n]$ y $n \in [0, \infty)$.

Operaciones

Operación Crear (sal Cadena)

Calcula: Devuelve una cadena vacía (contiene 0 caracteres).

Operación Esvacia (ent C: Cadena; sal Booleano)

Calcula: Devuelve verdadero si la cadena *C* tiene 0 caracteres, falso en caso contrario.

Operación Addcar (ent C: Cadena; car: Carácter; sal Cadena)

Calcula: Devuelve la cadena *C* con el carácter *car* añadido por la derecha.

Operación Delcar (ent C: Cadena; sal Cadena)

Requiere: *C* es una cadena no vacía.

Calcula: Devuelve la cadena *C* sin su último carácter.

Operación Concat (ent A, B: Cadena; sal Cadena)

Calcula: Devuelve la concatenación de ambas cadenas, es decir, si $A=a_1\dots a_n$ y

$B=b_1\dots b_m$, devuelve la cadena $a_1\dots a_nb_1\dots b_m$.

Operación Numcars (ent C: Cadena; sal Entero)

Calcula: Devuelve el número de caracteres en la cadena *C*.

Operación Elem (ent C: Cadena; pos: Entero; sal Caracter)

Requiere: *pos* es una posición válida de *C*.

Calcula: Devuelve el carácter de *C* situado en la posición *pos*.

Fin Cadena.

Vamos a implementar este TAD mediante una clase *cadena*. Puedes encontrar el código completo correspondiente a dicha implementación en la página 331, separado en dos ficheros, *cadena.h* para la definición de la clase y *cadena.cpp* para la definición de sus funciones miembro (observarás construcciones del lenguaje aún no explicadas). Vamos a explicar paso a paso cómo se ha construido el TAD, introduciendo a medida que nos vayan haciendo falta los nuevos conceptos de C++ utilizados en estos dos ficheros.

En primer lugar decidimos cómo vamos a dar soporte a la cadena en sí. Utilizaremos un array de caracteres. No vamos a limitar la longitud de las cadenas de modo que necesitaremos un array dinámico. Es decir, utilizaremos un puntero a *char* como atributo de la clase *cadena* al que se asignará el array que reservemos dinámicamente, lo llamaremos *cad*. Es decir, nuestro primer atributo será *char *cad*. También utilizaremos un entero

`maxcars` que indique el número de caracteres reservado actualmente en `cad`. Cuando sea necesario se reservará más memoria para `cad`; `maxcars` se actualizará para reflejar la nueva situación. La política será reservar siempre algunos caracteres de más para no tener que estar reservando nueva memoria cada vez que se introduzca un nuevo carácter en la cadena. Como de ese modo la longitud de la cadena podrá ser menor que el número de caracteres reservados usaremos un nuevo atributo para saber cuántos caracteres corresponden realmente a la cadena, un entero que llamaremos `ncars`. Todos estos atributos serán privados, solo podrá accederse a ellos a través del interface que definamos mediante las operaciones del TAD. Resumiendo lo dicho hasta ahora, nuestra clase sería algo así:

```
class cadena {
private:
    char *cad;
    int ncars;
    int maxcars;
};
```

Hasta aquí solo hemos definido los atributos, cuyos valores constituirán el estado. Vamos ahora con los métodos. Es el momento de introducir el concepto de **constructor**. Un constructor es un método especial que se encarga de inicializar los atributos de un objeto en el momento de su creación de modo que el objeto se encuentre en un estado consistente desde el comienzo de su existencia. Por ejemplo, en nuestro caso un objeto `cadena` debe tener inicializado su atributo `ncars` a cero, dado que inicialmente la cadena está vacía. Además en nuestro caso elegimos reservar desde el principio cierta cantidad de memoria para `cad`. El constructor se encargará de hacerlo y de inicializar `maxcars` según esa reserva. Si no declaramos un constructor C++ se encarga de asignar a nuestra clase un constructor *de oficio* que inicializará las variables a 0. Esto puede ser adecuado en algunos casos, como en el de `panel_turno`, pero en otros no, como en el de `cadena`: `cad` no debe ser inicializado a cero (`NULL`), sino que se debe efectuar una asignación de memoria dinámica. Necesitamos definir un constructor propio. En C++ un constructor se declara como un método que se llama igual que la clase y no tiene ningún valor de retorno, ni siquiera `void`. Respecto a los parámetros, tenemos varias posibilidades. Una primera opción es que no tenga ningún parámetro. A este caso se le llama *constructor por defecto*. Es el constructor que se usa cuando simplemente se declara un objeto de la clase en cuestión. Por ejemplo, ante la declaración: `cadena A` el constructor por defecto es invocado. Si no hemos definido ningún constructor C++ generará su constructor de oficio que actuará como constructor por defecto. Veamos cuál sería en nuestro caso la declaración y definición del constructor por defecto:

```
cadena::cadena(){
    cad = new char[LONG_INIC];
    maxcars=LONG_INIC;
    ncars=0;
}
```

`LONG_INIC` es una constante, un entero mayor que cero, e indica la cantidad de espacio reservado inicialmente para `cad`. Como vemos, este constructor coincide con la operación *Crear* definida en la especificación informal del TAD.

A menudo al crear un objeto nos interesaría inicializarlo con un estado distinto al que le asigna el constructor por defecto. En ese caso podemos usar constructores con

parámetros. Por ejemplo, si quisiésemos que al declarar un objeto cadena directamente se incluya ya un primer carácter podríamos tener un constructor:

```
cadena::cadena(char primer_car){
    cad = new char[LONG_INIC];
    maxcars=LONG_INIC;
    cad[0]=primer_car;
    ncars=1;
}
```

¿Cómo se utiliza un constructor con parámetros? Cuando declaremos un objeto de clase **cadena** mediante este segundo constructor deberemos utilizar una de las siguientes notaciones para la declaración: a) **cadena P(car)** o b) **cadena P=cadena(car)**, donde **car** es el carácter con el que queremos inicializar la cadena.

Vamos a incluir este segundo constructor también en nuestro programa. Observa que estamos hablando de dos métodos que tienen el mismo nombre, por lo que estamos utilizando sobrecarga de funciones. El primer método no tiene argumentos y el segundo tiene uno, con lo que la coexistencia de ambas definiciones es compatible.

Como vimos en la sección anterior **new** no sólo reserva memoria dinámica para un objeto sino que adicionalmente invoca un constructor sobre el objeto creado. ¿Cuál? En **new** usaremos una sintaxis parecida a la de la declaración de un objeto en cuanto a la invocación del constructor. En nuestro ejemplo, si declaramos un puntero a **cadena**, **cadena *P**, para crear un objeto dinámicamente con el constructor por defecto escribiríamos **P = new cadena** y con el otro **P = new cadena(letra)**.

Del mismo modo que existen los constructores existe el **destructor** (en singular porque para cada clase solo hay uno). Se trata de una función que es invocada automáticamente cuando un objeto de esa clase deja de existir. Su nombre es el de la clase precedido del carácter ~ (ALT-126). Su misión es liberar memoria reservada por el objeto, cerrar ficheros abiertos, etc. Por ejemplo, en nuestro caso al crear un objeto **cadena** estamos haciendo una reserva de memoria. Cuando ese objeto deje de existir esa memoria debería ser liberada. Será el destructor el que se encargue de hacerlo. Este es el destructor que vamos a utilizar:

```
~cadena();
cadena::~cadena() {delete [] cad;}
```

Acabamos de decir que el destructor es llamado automáticamente cuando deja de existir un objeto. Pero si este objeto es creado dinámicamente por el usuario (con **new**) ese objeto no deja de existir hasta que acaba el programa. Es decir, no es como las variables locales **auto**, que dejan de existir cuando termina de ejecutarse el bloque en que fueron definidas. En estos casos será el usuario el que tenga que invocar explícitamente al destructor cuando el objeto haya dejado de usarse. Lo haría mediante la función **delete**, que se encargaría de liberar la memoria asignada a sus atributos y de invocar al destructor correspondiente.

Veamos un ejemplo de su uso que ilustre el contenido del párrafo anterior. Declaramos un puntero a cadena, **cadena *C**. Se trata solo de un puntero, no lleva asociada llamada a constructor ni reserva de memoria. En un cierto punto del programa creamos una cadena dinámicamente, con un primer carácter 'a' y lo asociamos a este puntero:

`C = new cadena('a');`. Esta invocación a `new` genera la reserva de memoria para los atributos del objeto `cadena` y adicionalmente llama al constructor de `cadena` cuya cabecera tenga un parámetro `char`. Ese constructor se encarga de reservar memoria para el atributo `cad`, de inicializar variables y de escribir el carácter en `cad`. Por último, el puntero devuelto por `new` es asignado a `C`. Tras esto sigue la ejecución del programa normalmente, teniendo presente que `C` es un puntero a objeto, no un objeto. Esto quiere decir que accederemos a sus miembros, tanto atributos como métodos, no con el *operador punto* (`.`) sino con el *operador flecha*, `->`. Una vez que ese objeto deja de sernos útil lo destruimos: `delete C`. La invocación de `delete` activa el destructor de la clase, que en este caso lo único que hace es liberar la memoria asociada al atributo `cad`. Tras esto se libera la memoria destinada a los atributos del objeto.

Es importante tener muy presente lo siguiente: cuando un objeto es pasado como parámetro a una función se hace una copia de este. Lo que maneja localmente la función es la copia. Si el objeto tiene atributos que son arrays o punteros a variables, lo que será copiado es el puntero, pero no el array ni las variables apuntadas. Cuando la función llegue a su fin el objeto copia será destruido. En nuestro caso el destructor liberaría la memoria asignada al atributo `cad` del objeto original a través de la copia, lo cual es un error. Podemos solucionar este problema de dos formas:

- Mediante *paso de parámetros por referencia*: para un objeto que es pasado como parámetro por referencia, ya no se generará copia local, ya que lo que tenemos localmente es sólo un alias a un objeto anterior, con lo que el problema queda solucionado.
- Mediante la definición de un **constructor de copia**: un constructor de copia es un constructor que recibe como parámetro un objeto de la clase de la que es constructor (en realidad una referencia a dicho objeto). Su misión es efectuar una copia del objeto parámetro sobre el objeto creado. Si no definimos explícitamente un constructor de copia, C++ nos proporciona uno por defecto que simplemente copia los valores de los atributos, pero como hemos visto esto no es válido para un objeto con arrays o punteros: si algún atributo es un array o puntero a variable, será necesario reservar memoria para el nuevo objeto y copiar los contenidos correspondientes en esa memoria.

Cuando se pasa un parámetro por valor a una función, C++ aplica implícitamente, al crear la copia local del objeto parámetro, el constructor de copia de la clase a que pertenece este objeto. Para la clase `cadena`, si definimos un constructor de copia que haga una copia adecuada de un objeto `cadena`, el problema queda resuelto. En nuestro caso no incluiremos un constructor de copia para la clase `cadena`, sino que optaremos por la primera solución.

El siguiente método definido es el operador de asignación. Su misión es permitir asignar una cadena a otra. Este sería su código:

```
operator=(cadena &);  
cadena::operator=(cadena &cadB) {  
    delete [] cad;  
    cad = new char[cadB.ncars+INC];
```

```

maxcars=cadB.ncars+INC;
for(int i=0;i<cadB.ncars;i++) cad[i]=cadB.cad[i];
ncars=cadB.ncars;
}

```

El objeto parámetro es pasado como referencia. Ante una asignación A=B ocurre que A será el argumento implícito y B el parámetro. De ese modo, para acceder a un atributo de A en el cuerpo del método escribimos directamente su nombre, por ejemplo, cad en la primera línea, mientras que para acceder a los de B usamos la notación punto: cadB.cad. Fíjate en que el argumento cadB se pasa como referencia para evitar el problema expuesto anteriormente. INC es una constante que indica cuántos caracteres de más se reservan cada vez que hay que hacer una nueva reserva, para evitar nuevas reservas cada vez que se añada un carácter. Estudia el código para comprender qué hace.

Los siguientes métodos no introducen novedades sobre C++, estudia su código y entiende qué hacen y cómo funcionan. Por último, cabe señalar que addcar reservará nueva memoria si es necesario.

```

-- FICHERO CADENA.H -----
class cadena {
private:
    char *cad;
    int ncars;
    int maxcars;
public:
    cadena();
    cadena(char);
    ~cadena();
    operator=(cadena &);
    addcar(char);
    delcar();
    concatenar(cadena &);
    int esvacia();
    int numcars();
    char elem(int);
    imprimir();
};

-- FICHERO CADENA.CPP ----

#include <stdio.h>
#include "cadena.h"
#define LONG_INIC 4
#define INC 2

cadena::cadena(){
    cad = new char[LONG_INIC];
    maxcars=LONG_INIC;
}

```

```
ncars=0;
}

cadena::cadena(char primer_car){
    cad = new char[LONG_INIC];
    maxcars=LONG_INIC;
    cad[0]=primer_car;
    ncars=1;
}

cadena::~cadena() {delete [] cad; }

cadena::operator=(cadena &cadB) {
    delete [] cad;
    cad = new char[cadB.ncars+INC];
    maxcars=cadB.ncars+INC;
    for(int i=0;i<cadB.ncars;i++) cad[i]=cadB.cad[i];
    ncars=cadB.ncars;
}

cadena::addcar(char car) {
    char *cadaux;
    if(ncars==maxcars) {
        cadaux=cad;
        cad = new char[maxcars+INC];
        maxcars+=INC;
        for(int i=0;i<ncars;i++) cad[i]=cadaux[i];
        delete [] cadaux;
    }
    cad[ncars]=car;
    ncars++;
}

cadena::delcar() {ncars--; }

cadena::concatenar(cadena &cadB) {
    int i;
    char *cadaux;
    cadaux = cad;
    cad = new char[ncars+cadB.ncars+INC];
    maxcars=ncars+cadB.ncars+INC;
    for(i=0;i<ncars;i++) cad[i]=cadaux[i];
    for(i=0;i<cadB.ncars;i++) cad[ncars+i]=cadB.cad[i];
    delete [] cadaux;
    ncars+=cadB.ncars;
}

int cadena::esvacia() {return !ncars;}
```

```

int cadena::numcars() {return ncars;}

char cadena::elem(int pos) {return cad[pos-1];}

```

Con lo anterior ya tenemos definido el TAD. \otimes Ponlo a prueba, intenta cualquier combinación de operaciones que se te ocurra. Seguro que encuentras muchas carencias y te planteas posibles mejoras. Lánzate a implementarlas, es la mejor forma de familiarizarse con tantos conceptos nuevos y explicados de forma tan intensiva. Por ejemplo, ¿qué tal definir la función concat como operador, por ejemplo, con el signo +? Y mejorar el operador de asignación para permitir expresiones como $a=b=c$? Una pista sobre esto último: necesitas que la primera aplicación del método devuelva algo para que pueda ser usado como expresión a asignar en la siguiente aplicación del operador asignación. En cualquier caso aquí tienes algunas sugerencias para ver cómo se usa el TAD que hemos implementado:

```

#include "cadena.h"
main(){
    int resp,car;
    cadena A,B('Y'),*C,D;
    resp = A.esvacia();
    A.addcar('a'); A.addcar('b'); A.addcar('c');
    A.delcar();
    car = A.elem(2);
    B.addcar('A'); B.addcar('B'); B.addcar('C');
    A.concatenar(B);
    C=new cadena;
    C->addcar('x'); C->addcar('y'); C->addcar('z');
    delete C;
    C=new cadena[2];
    C[0].addcar('c'); C[0].addcar('i');
    C[1].addcar('c'); C[1].addcar('2');
    delete [] C;
    C=new cadena('X');
    delete C;
    D=B;
}

```

B.5. Plantillas

En C++ se permite dejar un tipo sin especificar en la definición de una función o una clase. En ese caso se dice que se tiene no una función o una clase sino una **plantilla** de la función o la clase. La plantilla especifica una estructura, una forma genérica, que para llegar a ser realmente una función o clase debe ser instanciada. Instanciarla consiste en concretar los tipos no definidos. Esta característica del lenguaje es muy útil para el manejo de TAD genéricos.

Veamos en primer lugar cómo se definen y usan las plantillas de funciones. Vamos a definir una que llamaremos `suma` y que suma dos elementos de cualquier tipo para el que esté definida la operación `+`. La definición de una plantilla de función es igual que la de una función, solo que precedida de la palabra `template` (plantilla) y una lista de los tipos que van a quedar sin especificar entre símbolos `<>`, cada nombre de tipo precedido por la palabra `class`. En nuestro ejemplo tendríamos:

```
template <class T> T suma(T a, T b) { return a+b; }
```

Como vemos `T` se usa exactamente igual que si de un nombre de tipo se tratase. Fíjate en que estamos utilizando implícitamente la sobrecarga del operador `+`. Si por ejemplo, definimos para la clase `cadena` de la sección anterior un operador `operador+` que concatene las cadenas podríamos aplicar `suma` a dos cadenas para obtener su concatenación. Respecto a la instanciación de la plantilla, ocurre cuando es invocada con unos argumentos concretos. El tipo de los argumentos permiten inferir el tipo asignado a `T` en esa llamada concreta. Por ejemplo:

```
int a=1,b=2,c;
float f=1.0,h=2.0,j;
c=suma(a,b);
j=suma(f,h);
```

Una cuestión importante a considerar: con esa definición de `suma` estamos imponiendo que `a` y `b` tengan el mismo tipo, sea el que sea. Para permitir tipos distintos tendríamos que utilizar dos tipos no definidos. Por ejemplo, una función que dados dos números de tipos distintos diga si el primero es mayor que el segundo:

```
template <class T1, class T2> int mayorque(T1 a, T2 b) { return a>b; }
```

Veamos ahora cómo declarar plantillas de clase. Lo haremos mediante un ejemplo, un TAD pila muy sencillo. La capacidad de la pila es fijada en el momento de su creación mediante el parámetro pasado al constructor. Tras el código comentamos las novedades:

```
#include <stdio.h>

template <class T>

class pila {
private:
    int nelems;
    int maxelems;
    T *elems;
public:
    pila(int);
    apilar(T);
    T desapilar();
}

template <class T> pila<T>::pila(int capacidad) {
    maxelems = capacidad;
```

```

nelems = 0;
elems = new T[capacidad];
}

template <class T> pila<T>::apilar(T elem) {
    if (nelems<maxelems) elems[nelems++]=elem;
}

template <class T> T pila<T>::desapilar() {
    if (nelems>0) return elems[--nelems];
}

main() {
    pila <int> a(10);
    pila <char *> b(10);
    char *cad1,cad2;

    a.apilar(15);
    a.apilar(12);
    printf("suma de apilados: %d\n",a.desapilar()+a.desapilar());
    b.apilar("amigos");
    b.apilar("Hola");
    cad1=b.desapilar();
    cad2=b.desapilar();
    printf("%s %s\n",cad1,cad2);
}

```

Fíjate en que la definición de la clase está precedida por la línea `template <class T>`. En esa línea se indican todos los tipos que se van a dejar sin definir en la plantilla de clase. A continuación se define la clase como una clase normal en la que se usa un tipo `T`. Respecto a la definición de las funciones, aquellas que usen un tipo no definido serán definidas como plantillas de funciones, teniendo en cuenta que el nombre de la clase antes del operador `::` cambia a `nombre<lista_tipos_sin_definir>`, lista en la que no aparece ya la palabra `class`. Respecto a la instanciación de las plantillas de clase, podemos ver ejemplos en el código anterior: se escribe el nombre de la plantilla, `pila` en nuestro caso, seguido de la lista de tipos concretos a usar en lugar de los tipos por definir, de nuevo entre signos `<>`.

B.6. Excepciones

En C++ tenemos a nuestra disposición mecanismos para definición y manejo de excepciones. Aquí daremos solo algunas nociones básicas que nos permitan usarlas en la construcción de TAD. Las excepciones son situaciones anómalas que ocurren durante la ejecución de un fragmento de código, por ejemplo, dentro de una función, y que sobrepasan la capacidad de gestión de situaciones anómalas de dicha función. Por así decirlo, es una situación ante la cual la función no sabe qué hacer. Y lo que hace entonces es lanzar

una excepción, es decir, comunicar a quien la haya llamado que ha ocurrido algo a lo que ella no sabe hacer frente. Si el código que la llamó es capaz de gestionar la excepción (de manejarla) ejecutará las acciones pertinentes y la situación quedará controlada. En caso contrario lanzará a su vez la excepción a quién lo hubiese llamado a él. Si esta cadena de lanzamientos continúa hasta llegar al programa principal, el programa será abortado.

⊗ Ilustraremos el uso de este mecanismo en C++ mediante la implementación de un ejemplo, la función dividir, que comentamos en detalle a continuación:

```
#include <iostream.h>

class division_por_cero { };

int dividir(int a, int b) {
    if (!b) throw division_por_cero();
}

main(){
    int a=2,b=3,c;
    try{
        c=dividir(a,b); cout << "a/b=" << c << "\n";
        b=0;
        c=dividir(a,b); cout << "a/b=" << c << "\n";
    }
    catch(division_por_cero){
        cout << "Error: division por cero\n";
    }
}
```

La operación que puede dar problemas es la división de la función dividir. Cuando el divisor es 0 el resultado es indeterminado y se debe lanzar una excepción. Esto se hace mediante la palabra `throw` seguida de la excepción concreta que queremos lanzar. La excepción es un objeto. En este caso hemos creado una clase `division_por_cero` con el único objetivo de poder crear objetos de esa clase para lanzarlos como excepción. La palabra `throw` va seguida en este caso de la invocación al constructor por defecto de la clase definida. Cuando nos interese podremos usar constructores con parámetros para dar más información sobre lo que ocurrió a quien reciba la excepción.

¿Y quién la recibe? Quien llamó a dividir. La llamada está en la función `main`. Fíjate en la estructura:

```
try      { // sentencias }
catch(clase) { // sentencias }
```

En el bloque asociado a `try` (intentar) se coloca cualquier código que pueda generar excepciones, como por ejemplo, en nuestro caso, código que contenga una llamada a `dividir`. Si la excepción ocurre, como es el caso de la segunda llamada a `dividir`, la ejecución del código del bloque `try` es interrumpida y se pasa a ejecutar el o los bloques `catch` (atrapar) situados a continuación. Estos bloques tienen dos partes, la especificación de excepciones atrapadas, entre paréntesis, y el bloque de acciones a ejecutar. Ante una

excepción en el bloque `try` se comprueba si ésta corresponde a la clase especificada entre paréntesis. Si es así la excepción queda atrapada: pasa a ser gestionada por ese bloque `catch`. Esto quiere decir que ya no será lanzada de nuevo y que se ejecutarán las sentencias del bloque para gestionarla. Si la excepción no es atrapada se intentaría con otros bloques `catch`, si es que los hay. Si ninguno la atrapa, la excepción es lanzada de nuevo. En nuestro ejemplo, como ya estamos en la función principal, esto querría decir que el programa sería abortado.

Este mecanismo fue diseñado con objeto de manejar excepciones reales, es decir, situaciones anómalas. Pero una vez definido puede ser utilizado por el programador a su antojo de la forma que prefiera: se pueden lanzar excepciones ante situaciones perfectamente normales, por ejemplo, como mecanismo alternativo a `return` en ciertas circunstancias.

Ejercicios Propuestos

Ejercicio B.1 Crea una clase `persona` con los siguientes atributos:

- `nombre`: cadena de caracteres.
- `cumple`: de tipo `fecha`, tipo a definir con `typedef`.
- `tlf`: de tipo `telefono`, también a definir con `typedef`. Tendrá un campo `long int` con el número, un `int` con el prefijo y un enumerado con el tipo de teléfono: fijo, móvil o fax.

Crea una clase `agenda` que tenga un atributo en que se pueda almacenar una lista de objetos `persona`. Implementa métodos de clase que permitan dar de alta a una persona en la agenda, darla de baja (identificándola mediante su nombre) o generar listados de personas que cumplan una cierta condición (por ejemplo, que su nombre contenga cierta cadena, que tengan una cierta edad, etc.). Implementa un constructor de copia y un destructor para la clase `agenda`. Haz un programa que utilice las operaciones implementadas.

Ejercicio B.2 Crea una clase genérica `cola` que implemente el TAD genérico `Cola[T: tipo]`, definido del siguiente modo:

TAD Cola[T: tipo] es Crear, Esvacia, Añadir, Primero, Eliminar

Los valores de tipo `Cola[T: tipo]` son secuencias de elementos de tipo `tipo`, es decir, secuencias $x_1x_2\dots x_n$ donde x_i es un elemento del tipo `tipo`, $i \in [1, n]$ y $n \in [0, \infty)$.

Operaciones

Operación Crear (sal Cola)

Calcula: Devuelve una cola vacía (contiene 0 elementos).

Operación Esvacia (ent C: Cola; sal Booleano)

Calcula: Devuelve verdadero si la cola `C` tiene 0 elementos, falso en caso contrario.

Operación Añadir (ent C: Cola; elem: tipo; sal Cola)

Calcula: Devuelve la cola `C` con el elemento `elem` añadido por el final, es decir, situado en la posición $(n+1)$ -ésima.

Operación Primero (ent C: Cola; sal tipo)

Requiere: *C* es una cola no vacía.

Calcula: Devuelve el elemento *elem* situado en la primera posición de la cola *C*.

Operación Eliminar (ent *C*: Cola; sal Cola)

Calcula: Devuelve la cola resultante de eliminar el primer elemento de la cola *C*.

Fin Cola.

Programa un constructor de copia y destructor adecuados para la clase *cola*. Haz un programa en el que se lea de teclado los números introducidos por el usuario hasta que se introduzca un 0. Almacena esos números a medida que son leídos en una cola de enteros. Luego genera una cola de cadenas de caracteres constituida por la conversión a cadena de cada número de la primera. Por último muestra por pantalla la secuencia de números a partir del contenido de la segunda cola.

Ejercicio B.3 Crea una clase *matriz* con los siguientes atributos:

- *M*: la matriz propiamente dicha, array bidimensional de enteros.
- *ncols*: número de columnas, entero.
- *nfilas*: número de filas, entero.

Implementa un constructor que permita crear una matriz a partir de un array bidimensional y su número de filas y columnas. Implementa las operaciones de suma y producto de matrices. Ante cualquier discrepancia en las dimensiones de los operandos de deberá generar una excepción que será capturada y tratada por el programa que ejecutó la operación (por ejemplo, se mostrará un mensaje de error por pantalla). Consideraremos exclusivamente matrices de números naturales, de modo que si alguno de los elementos de alguno de los operandos es negativo también generaremos una excepción. Se generarán excepciones diferentes para cada una de las dos situaciones anómalas posibles, teniendo prioridad la excepción debida a la presencia de elementos negativos sobre otra. Pon a prueba la clase *matriz*, especialmente en cuanto al manejo de excepciones, con algunos ejemplos.

Referencias bibliográficas

Al igual que en el caso de C, también para profundizar en el aprendizaje de C++ hay muchos libros disponibles. Recomendamos [Stroustrup98], escrito por el creador del lenguaje.

Apéndice C

Práctica 1: Análisis y diseño de estructuras de datos

En esta primera práctica el objetivo es poner de manifiesto una justa comprensión del funcionamiento y un apropiado uso de las estructuras de datos estudiadas en la primera parte de la asignatura. Para ello, se planteará un problema que requiere un manejo intensivo de información. Se realizará un estudio teórico-práctico de las diferentes alternativas –en cuanto a las estructuras de datos que mejor se ajusten a los requerimientos del problema– hasta llegar al nivel de implementación. Este estudio seguirá un proceso metódico, compuesto por los siguientes pasos: analizar los requisitos de la aplicación, elegir y diseñar los tipos abstractos adecuados a la información usada, especificar formal o informalmente estos tipos, seleccionar la estructura más eficiente para cada tipo, implementar el diseño en un lenguaje concreto, validar el correcto funcionamiento de la implementación (respecto a su especificación) y analizar experimentalmente la eficiencia conseguida.

La memoria de prácticas a entregar por el alumno deberá documentar separadamente cada uno de estos apartados del proceso de desarrollo, indicando las decisiones adoptadas en los mismos. Además, es conveniente realizar un informe de la dedicación personal en cada parte y un estudio del logro de los criterios de calidad estudiados en el capítulo 1 (legibilidad del código, reparto de funcionalidad, robustez, eficiencia, etc.).

C.1. Enunciado

Possiblemente, entre todas las aplicaciones existentes una de las más exigentes, en cuanto a volúmenes de información procesados, son los **buscadores de Internet**. Ya comentamos por encima las características del problema al comienzo del capítulo 3. Vamos a considerar ahora el problema de estructurar la gran cantidad de información usada de la mejor manera posible, con el propósito de optimizar la recuperación de información a través de la búsqueda por palabras clave.

Supongamos que disponemos de una lista de páginas web con las que vamos a trabajar, almacenada en un **fichero de enlaces**. Este fichero es un archivo de texto ASCII, con la siguiente estructura: existe una línea por cada página web existente. Cada

línea contiene tres campos separados por uno o más espacios en blanco (o tabuladores). El primer campo es la **dirección URL** de la página, el segundo es el **nombre del archivo** donde está almacenada la página localmente y el tercero es un entero indicando la **puntuación** relativa de la página. Por ejemplo, podemos tener algo como lo siguiente:

<code>http://dis.um.es/~ginesgm/aaed.html</code>	<code>aaed.html</code>	812
<code>http://www.google.com</code>	<code>google.index.html</code>	950
<code>http://www.elpais.es/</code>	<code>elpais.index.htm</code>	24
<code>http://dis.um.es/~ginesgm/fip/</code>	<code>index3.html</code>	620
.....		

Todos los ficheros del segundo campo están accesibles localmente (en el mismo directorio que el fichero de enlaces) y son también archivos de texto con el contenido de las páginas correspondientes, en formato HTML.

El buscador procesa inicialmente este fichero de enlaces, leyendo todas las páginas referenciadas y almacenando internamente la información adecuada. Además debe permitir realizar las siguientes operaciones:

- Dar de alta una página web nueva.
- Dar de baja una página, especificando su dirección URL.
- Consultar los datos almacenados para una página, dada la URL.
- Hacer una **búsqueda** de páginas por palabras clave. Sin duda alguna, esta es la operación fundamental. Se deja cierta libertad en la forma de realizar las búsquedas, pero por lo menos deben permitir lo siguiente. En cada búsqueda se podrá indicar una lista de palabras clave a buscar, que podrán ser combinadas con los siguientes operadores:
 - **Operador AND.** Las páginas resultantes deberán contener todas las palabras clave especificadas en la búsqueda.
 - **Operador OR.** Las páginas deberán contener alguna de las palabras clave especificadas.
 - **Operador comillas, “ ”.** Las páginas deberán contener todas las palabras especificadas en la búsqueda y además en el mismo orden.

La operación de búsqueda debe producir como resultado una lista de todas las páginas que se ajusten a los criterios de búsqueda. Esta lista tendrá que estar ordenada por la puntuación relativa de las páginas, de mayor a menor, que indica el interés de una página dada. Para cada página de la lista resultante se deberá mostrar la dirección URL, la puntuación de la página y un trozo de texto de la página donde aparezca alguna de las palabras buscadas. Es imprescindible que la implementación de esta operación sea muy eficiente, especialmente para las operaciones de búsqueda y consulta. Además hay que tener en cuenta que el fichero de enlaces puede contener hasta varios miles de millones de páginas, por lo que se deberá cuidar el uso de memoria.

C.2. Análisis de los requisitos de la aplicación

Vamos a ver ordenadamente todas las fases de la resolución de la práctica, que deberían estar documentadas en la memoria final de prácticas. Tal y como estudiamos en el capítulo 1, el primer paso en la resolución de problemas es el estudio de las características y requisitos del problema considerado. El objetivo de este estudio es hacerse una idea más precisa de lo que se pide, antes de empezar a tomar decisiones.

El enunciado de un problema puede ser más o menos preciso y completo. Como resultado del análisis del problema, todo lo que no esté especificado debe especificarse completamente y todas las ambigüedades deben ser eliminadas. ¿Cómo? Pues consultando al *cliente* o tomando las decisiones que se crean más adecuadas. Por ejemplo, en la operación de búsqueda no está claro cómo se pueden combinar los operadores AND, OR y comillas. ¿Se puede utilizar más de un operador por cada consulta o la consulta es por uno de ellos? ¿Se pueden usar expresiones con paréntesis?

En consecuencia, el resultado del análisis es una nueva descripción del problema en nuestros propios términos. En primer lugar, la descripción debería indicar de forma concreta y precisa los datos que se deben almacenar y los que no, definiendo las relaciones existentes entre los datos almacenados. En segundo lugar, se deberían enumerar las funcionalidades que se requieren de la aplicación, es decir, las operaciones que puede utilizar el usuario. Para cada operación habrá que concretar el tipo de los parámetros que recibe y del valor devuelto. Además, ambas descripciones –de datos y de funcionalidades– deben estar acompañadas de las posibles restricciones de uso y requisitos especiales, como por ejemplo, quién puede acceder a determinada información, en qué operaciones la eficiencia es crítica, qué restricciones de memoria pueden aparecer, etc.

Análisis de la información

Vamos a ver como sería el resultado del análisis de requisitos, aplicado sobre el problema del buscador de Internet. En cuanto a la información manejada por la aplicación, tenemos páginas web como el concepto central del problema. Cada página web tiene asociada la siguiente información:

- **Dirección URL.** Es una cadena que puede contener letras, números o caracteres especiales. Es única y exclusiva de cada página, aunque es previsible que haya muchas páginas con un prefijo común en la dirección URL.
- **Nombre del fichero local.** Es una cadena que especifica un fichero, accesible localmente desde la aplicación, donde está el contenido de la página web asociada. Normalmente será de extensión .html o .htm, aunque no necesariamente.
- **Fichero local.** Es un fichero de texto que contiene la página web a la que está asociado. Es decir, se supone que hemos descargado la página de Internet y la hemos grabado en disco. No obstante, el nombre del fichero no tiene por qué coincidir con el nombre original.
- **Puntuación.** La puntuación es un número que indica la importancia o interés relativo de cada página web. No sabemos cómo se ha obtenido este número. No se

especifica el tipo, así que supondremos números enteros con signo. Consideraremos que varias páginas web pueden tener la misma puntuación.

- **Palabras clave.** Una página web puede contener muchas palabras clave y una palabra clave puede aparecer en muchas páginas. El enunciado no especifica cuáles son las palabras clave de una página web. Supondremos que las palabras clave son todas las palabras que aparecen en el contenido de la página. En particular, las palabras clave serán cadenas de letras, separadas por espacios, puntos, comas o cualquier cosa que no sea una letra. Además, excluiremos todo lo que haya entre los caracteres "<" y ">", que se usan en HTML para indicar marcadores¹.

Por otro lado tenemos el **fichero de enlaces**, en el cual se almacenan las páginas web con las que trabajamos inicialmente. El formato de este fichero ya está claramente especificado, así que no lo volveremos a repetir.

Existe una restricción espacial en la información almacenada. El enunciado habla de "varios miles de millones de páginas". Por lo tanto, las estructuras utilizadas deben estar optimizadas en cuanto a uso de memoria. Esto nos lleva a que no se deben almacenar en memoria todos los contenidos de los ficheros locales, ya que el coste resultaría prohibitivo.

Análisis de las operaciones

La funcionalidad que debe proveer el sistema es descrita en el enunciado de manera bastante imprecisa. Existe una funcionalidad adicional, que no está indicada explícitamente, que consiste en procesar el fichero de enlaces para inicializar la aplicación. Así que tenemos las siguientes operaciones, descritas con más detalle:

- **Procesamiento del fichero de enlaces.** Recibe como entrada el nombre de un archivo, que tendrá el formato del fichero de enlaces. El objetivo de esta operación es leer cada una de las páginas web que aparezcan en el fichero y crear en memoria las estructuras necesarias para almacenar los datos adecuados. Puesto que es una operación de inicialización, decidimos que en caso de error se interrumpa la ejecución del programa, indicando el fallo.
- **Alta de una página.** Debe recibir como entrada los mismos datos contenidos en cada línea del fichero de enlaces, es decir: dirección URL, nombre del fichero local y puntuación. El fichero local debe ser accesible por el programa. Su función es añadir la nueva página a las estructuras internas de la aplicación y actualizar el fichero de enlaces. En caso de error (fichero no existente, página ya existente, etc.) devuelve un mensaje de error pero sin interrumpir el programa.
- **Baja de una página.** Según el enunciado, recibe como entrada una dirección URL. Modifica las estructuras internas para suprimir la página web asociada, actualizando el fichero de enlaces. Igual que antes, en caso de error devuelve un mensaje pero sin interrumpir el programa.

¹Observemos que esto no está especificado en el enunciado. Es una decisión que tomamos en la fase de análisis del problema y que podría ser más o menos discutible.

- **Consulta de una página.** Recibe como entrada una dirección URL. El enunciado no indica explícitamente qué datos deben devolverse en la consulta. Supondremos que devuelve el nombre del fichero local asociado, la puntuación y una lista de las palabras clave que aparecen, ordenadas por el número de apariciones en esa página web. Si no existe la página solicitada, se devolverán valores nulos.
- **Búsqueda por palabras clave.** De acuerdo con el enunciado, esta es la operación más importante. Hemos visto que la combinación de los operadores está descrita de forma ambigua, así que tomaremos una decisión sencilla: en cada búsqueda sólo podrá aparecer un tipo de operador. Por lo tanto, la entrada será un operador (AND, OR o comillas) y una lista de una o más palabras clave, siendo cada palabra una cadena de letras. El resultado será una lista de descripciones de páginas web, ordenados de mayor a menor puntuación. Cada descripción contiene la dirección URL y la puntuación de la página. Además, el enunciado pide que se muestre "un trozo de texto de la página donde aparezca alguna de las palabras buscadas". La especificación es ambigua. Supondremos que este trozo de texto corresponde a la primera aparición de alguna de las palabras en la página, más algunos caracteres antes y algunos después.

Existe una restricción muy importante de tiempo sobre la operación de búsqueda por palabras clave. Si nos fijamos en los buscadores reales existentes, una búsqueda no suele tardar más de medio segundo, aun trabajando con unos tres mil millones de páginas. Además, también podemos considerar que la consulta de páginas por URL debe ser rápida. Las otras operaciones son de mantenimiento y no tienen requisitos estrictos de tiempo.

C.3. Diseño del programa

La siguiente fase del proceso es el diseño del programa. Como vimos en el capítulo 1, en el apartado 1.1.3, el diseño es un paso previo a la implementación, cuyo objetivo es producir la estructura de tipos y de operaciones que debe tener la aplicación. El diseño es, por lo tanto, como un plano de la forma global que tendrá el programa final.

La etapa de diseño se puede entender como una serie de toma de decisiones: qué tipos abstractos se usan para almacenar la información necesaria, cómo se organizan estos tipos para no repetir información, qué estructuras de datos son más adecuadas en cada tipo, etc. Además, durante el diseño se deben crear las especificaciones que deberá cumplir la implementación. Todo esto forma parte de lo que se conoce como las **decisiones de diseño**.

Realizar un cuidadoso diseño previo del programa, tomando las decisiones adecuadas, resulta fundamental para evitar problemas durante la fase de implementación. En otro caso, tendremos que "volver atrás", cuando estemos programando, para rectificar decisiones de diseño equivocadas. Vamos a ver, por encima, cómo sería el diseño para el caso del buscador.

Arquitectura de tipos abstractos

Basándonos en las restricciones de tiempo y memoria que surgieron durante la fase de análisis, tomamos la decisión de representar las páginas web de forma abreviada mediante códigos identificativos, por ejemplo enteros de 32 bits. Este código nos permite, al mismo tiempo, referenciar a una página de forma reducida y poder acceder rápidamente a su información si usamos una estructura adecuada.

Utilizando estos códigos, tendríamos los siguientes tipos abstractos:

- **Página.** Es un registro compuesto por el código de la página (de tipo entero), la dirección URL (de tipo cadena), un nombre de fichero (también de tipo cadena) y la puntuación (de tipo entero).
- **Códigos.** Es un diccionario, donde las claves son códigos de páginas y el valor asociado es una referencia a un registro de tipo Página.
- **DireccionesURL.** Es un diccionario, donde las claves son direcciones URL y el valor asociado es una referencia a un registro de tipo Página.
- **Palabras.** Es un diccionario donde tenemos precalculadas las búsquedas de cada palabra. Las claves son palabras, de tipo cadena; los valores son colecciones de códigos de las páginas web donde aparece esa palabra, ordenadas por puntuación de las páginas correspondientes, de mayor a menor.
- **Buscador.** Es un tipo abstracto que engloba toda la información y funcionalidades utilizadas en la aplicación, de cara al usuario. Contendría un registro con un conjunto de páginas y los diccionarios Códigos, DireccionesURL y Palabras. Además, incluye las operaciones que hemos visto en la fase de análisis: procesamiento del fichero de enlaces, alta de página, baja, etc.

En aplicaciones como la del buscador web –donde existen muchas formas posibles de organizar la información– determinar una correcta arquitectura de tipos es esencial. La decisión influirá mucho en los demás pasos del desarrollo. La eficiencia del sistema viene dada no sólo por las estructuras de datos, sino que también está condicionada por la organización de la información. Por ejemplo, la decisión de introducir códigos asociados a las páginas es debida a las necesidades de eficiencia, pero influye en el diseño de los tipos abstractos.

De forma parecida, la utilización del diccionario Palabras surge de la necesidad de conseguir búsquedas rápidas. Al tenerlas precalculadas, si necesitamos buscar una palabra cualquiera simplemente tendremos que acceder en el diccionario a esa palabra. Después, los códigos resultantes deben ser buscados en el diccionario Códigos, para obtener los demás datos de la página. Si la búsqueda tiene varias palabras combinadas con un AND, deberemos hacer una intersección de los conjuntos correspondientes a cada palabra; y si tenemos un OR, tendremos que hacer una unión².

En la figura C.1 se muestran gráficamente los tipos definidos en el problema. Se puede ver que los tipos Códigos y DireccionesURL forman una estructura dual, con dos estructuras de acceso que apuntan a los mismos datos.

² ¿Qué ocurre con el operador comillas, “ ”? Se deja como ejercicio para el lector.

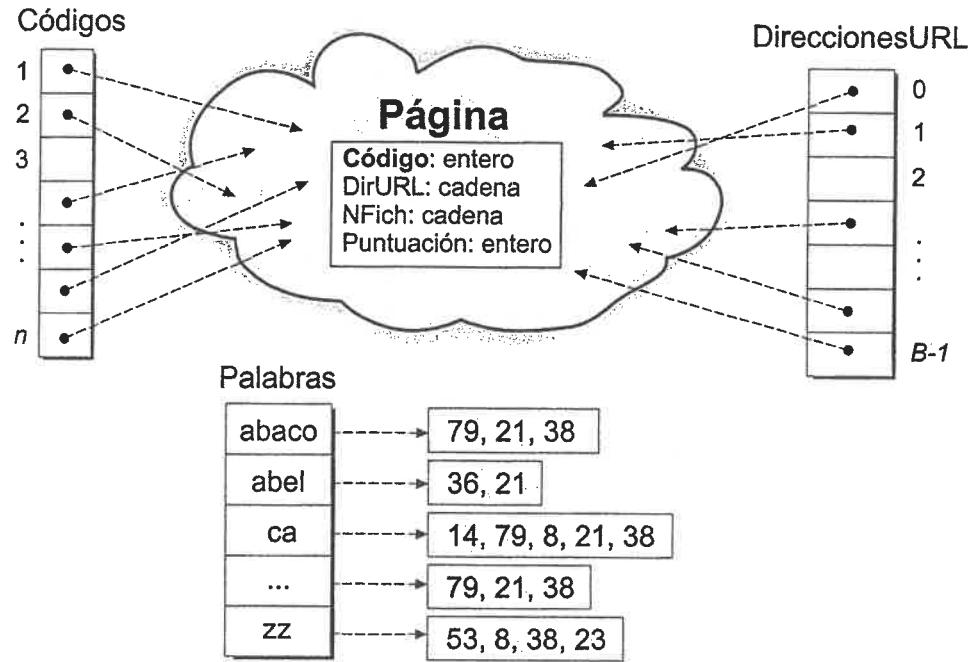


Figura C.1: Representación gráfica de la arquitectura de tipos en el problema del buscador web.

Especificación de los tipos de datos

Después de crear la arquitectura global de tipos, el siguiente paso de refinamiento consiste en especificar de forma más detallada cada uno de los tipos abstractos, por separado. Para ello utilizaremos algún formato de notación como los estudiados en el capítulo 2. La primera decisión sería elegir el formato de especificación, formal o informal, a utilizar. Ya vimos en el capítulo 2 las ventajas de utilizar especificaciones formales (posibilidad de crear prototipos, verificación de automática de la implementación, etc.). No obstante, en la práctica el desarrollo de especificaciones formales para problemas complejos suele ser difícil y se opta por utilizar especificaciones informales, pero descritas de la forma más precisa posible.

Como ejemplo, vamos a mostrar la especificación informal del tipo Página, que representa una única página web.

TAD Página es Crear, Código, DirURL, NFich, Puntuación, Destruir

Descripción

Los valores de tipo Página son páginas web, representadas mediante registros no modificables. La creación de una nueva página se hace con la operación Crear, la cual recibe como parámetros los datos de esa página (dirección URL, nombre del fichero y puntuación). Existe una operación para liberar la memoria y cuatro operaciones para consultar cada uno de los atributos de las páginas. Los códigos son generados de forma automática.

Operaciones

Operación Crear (ent d, n: cadena; p: entero; sal Página)

Requiere: Las cadenas d y n deben ser no vacías.

Calcula: Devuelve una nueva página donde la dirección URL es d , el nombre del fichero local es n y la puntuación es p . El código de la página es asignado automáticamente y de forma única a cada página.

Operación Código (ent P : Página; sal entero)

Calcula: Devuelve el código asignado automáticamente a la página.

Operación DirURL (ent P : Página; sal cadena)

Calcula: Devuelve la dirección URL de la página. Es decir, si la página ha sido creada con $Crear(d, n, p)$ devuelve d .

Operación NFich (ent P : Página; sal cadena)

Calcula: Devuelve el nombre de fichero de la página. Es decir, si la página ha sido creada con $Crear(d, n, p)$ devuelve n .

Operación Puntuación (ent P : Página; sal entero)

Calcula: Devuelve la puntuación de la página. Es decir, si la página ha sido creada con $Crear(d, n, p)$ devuelve p .

Operación Destruir (ent P : Página)

Modifica: P .

Calcula: Libera la memoria asignada a la página P .

Fin Página.

De forma similar, deberíamos desarrollar las especificaciones informales de los tipos abstractos restantes.

Estructuras de datos para los tipos

El siguiente paso en el diseño de la solución es decidir las estructuras de datos que vamos a utilizar para representar cada tipo abstracto. Este paso nos acerca un poco más a la implementación, aunque está todavía dentro del diseño. Por lo tanto, habrá que tener en cuenta consideraciones de eficiencia, tanto con respecto al tiempo de ejecución como al uso de memoria. Por ejemplo, en relación a la representación de conjuntos y diccionarios podemos considerar todas las estructuras que estudiamos en los capítulos 3 y 4 (tablas de dispersión, tries, árboles B, AVL, etc.). Para cada tipo abstracto la mejor solución podrá ser una u otra, en función del problema.

En algunos casos, será más interesante adaptar alguna estructura de datos existente, para ajustarla a la aplicación particular de que se trate; y otros casos tendremos que combinar varias estructuras para obtener una estructura de datos múltiple, como las vistas en el capítulo 3.

Por ejemplo, el tipo Página se puede entender como un simple registro, por lo que su representación es inmediata usando registros³.

tipo

Página = **registro**

Código: entero

DirURL: cadena

³Ojo, estamos hablando sólo de la estructura de datos y no de las operaciones. El que el tipo Página use una estructura de registros no implica que no se implemente con una clase, o que sus atributos deban ser públicos en la clase.

NFich: cadena

Puntuación: entero

finregistro

En cuanto a los diccionarios Códigos, DireccionesURL y Palabras, se presentan todas las alternativas conocidas. ¿Cómo decidimos entre una u otra? Pues teniendo en cuenta las operaciones que se necesitan.

- En el caso del tipo Códigos, necesitamos un acceso rápido a los datos de una página dado un código numérico. Si asignamos los códigos de manera secuencial, 1, 2, 3, ..., entonces podemos usar un array indexado por códigos, de referencias al tipo Página. Es decir, Códigos= **array [1..n] de Puntero[Página]**. La operación de acceso tendría claramente un $O(1)$. El inconveniente es que no sabemos a priori el tamaño, n , que debe tener este array. Además, se pueden añadir y modificar páginas en tiempo de ejecución. Podríamos hacer una estimación del número de páginas o bien obtener ese dato de una primera lectura del fichero de enlaces. Y para tener en cuenta la inserción se podrían reservar algunas posiciones más en el array.

Alternativamente, se podría utilizar una tabla de dispersión abierta con un tamaño suficientemente grande. Esto soluciona, en parte, el problema de estimar el tamaño de la tabla. Ahora deberíamos elegir una función de dispersión adecuada (que podría ser una simple operación de módulo) y el tiempo de ejecución ya no sería constante. Si la tabla es de tamaño B y tenemos n elementos, el orden de complejidad de las consultas sería un $O(1 + n/B)$.

Otra posibilidad sería representar el diccionario Códigos mediante una estructura de árboles AVL. Con esto evitamos los problemas de tamaño de las soluciones anteriores, pero el tiempo de ejecución de las consultas en este caso sería un $O(\log n)$.

- En cuanto al diccionario DireccionesURL, además de las posibilidades anteriores, podríamos usar árboles trie. Hay que tener en cuenta que muchas direcciones comparten prefijos comunes, de manera que se podría reducir el uso de memoria. En los puntos de la marca de fin aparecerían ahora referencias al tipo Página. En contrapartida, el tiempo de ejecución podría ser elevado. Posiblemente, la mejor solución en este caso es utilizar una tabla de dispersión, donde la función de dispersión haga uso de la técnica de extracción.
- Para el diccionario Palabras tenemos que las claves son cadenas y los valores son colecciones de códigos, de tipo entero. Como hemos visto, sobre estas colecciones de códigos necesitamos operaciones de unión, intersección, inserción ordenada, etc. Existirán muchas de estas colecciones, una por cada palabra clave del diccionario. Esto nos puede llevar a la necesidad de crear un nuevo tipo abstracto de datos, lo que implica volver a la fase anterior del diseño para modificar la arquitectura de tipos abstractos. Deberíamos incluir un nuevo tipo abstracto ConjuntoCódigos. Una representación adecuada podría ser mediante listas ordenadas. Aunque la inserción de un nuevo código en una lista (al añadir una página nueva) sea un $O(n)$, lo importante es optimizar las operaciones de unión e intersección.

En cuanto a la representación del diccionario Palabras en sí, aparecen básicamente las mismas posibilidades que antes: tablas de dispersión, árboles trie, AVL, B, etc.

Finalmente, el tipo **Buscador** sería un registro que contiene los tipos **Códigos**, **DireccionesURL** y **Palabras**. Se puede decir que se trata de un tipo de “naturaleza algorítmica”, por lo que la dificultad no está en diseñar su estructura de datos sino en los algoritmos para llevar a cabo las operaciones de consulta, búsqueda, etc., de forma eficiente.

C.4. Implementación del diseño

La implementación siempre debe partir de un diseño previo, donde se indica la especificación de cada tipo y operación, y las estructuras de datos para representar los tipos abstractos. Obviamente, la implementación es dependiente del lenguaje y del paradigma de programación, en nuestro caso C++ y orientación a objetos⁴. Por lo tanto, cada tipo abstracto se corresponderá con una clase que implemente la estructura y operaciones de ese tipo. Además de la estructura de clases, que surgen del diseño, habría que realizar una estructuración, en librerías o paquetes, de todas las funcionalidades, incluyendo las de la interfaz con el usuario.

Por ejemplo, para la aplicación del buscador web podemos decidir crear las siguientes tres librerías:

- **Tipos.** Contiene las clases: **Pagina**, **Codigos**, **DireccionesURL**, **Palabras** y **ConjuntoCodigos**, asociadas a los tipos abstractos correspondientes.
- **Buscador.** Contiene la clase **Buscador**, que implementa el TAD **Buscador**.
- **Interfaz.** Dentro de esta librería estarían todas las operaciones que constituyen la interfaz con el usuario, recepción de datos por teclado, salida en pantalla, etc.

Dentro de la librería **Tipos**, la clase **Pagina** podría tener la siguiente definición:

```
class Pagina {
private:
    int codigo;
    char *dirURL;
    char *nFich;
    int puntuacion;
public:
    Pagina (char *d, char *n, int p); // Inicialización de una página
    ~Pagina (); // Eliminación de una página
    int Código ();
    char *DirURL ();
    char *NFich ();
    int Puntuacion ();
};
```

Por otro lado, la definición de la clase **Buscador** podría ser:

⁴Aunque a un nivel de programación modular exclusivamente, sin usar herencia.

```

class Buscador {
    private:
        Codigos *cods;
        DireccionesURL *dirs;
        Palabras *pals;
        char *nombreFicheroEnlaces;
    public:
        Buscador (char *NombreFich); // Inicializ. usando el fichero de enlaces
        ~Buscador (); // Operación de eliminación del buscador
        void AltaPagina (char *urlPagina, char *nFichLocal, int puntuacion);
        void BajaPagina (char *urlPagina);
        int ConsultaPagina (char *urlPagina, (char *) &nFichLocal,
                            int &puntuacion, Lista<char *> &palabras);
        Lista<DescripBusqueda> *Busqueda (TipoOper op, Lista<char *> palabras);
};

```

Nos damos cuenta de que aparecen algunos tipos nuevos, que no tuvimos en cuenta durante la fase de diseño. Por lo tanto, deberíamos volver atrás para añadir a la arquitectura de tipos abstractos `DescripBusqueda`, `TipoOper` y el tipo genérico `Lista<T>`, siendo `T` cualquier otro tipo.

De la misma forma, se crearían las cabeceras de las demás clases y posteriormente se implementarían las operaciones de cada una.

C.5. Validación y verificación

La última fase de desarrollo consistiría en comprobar que la implementación responde a las especificaciones. Para ello, se diseñará un conjunto de pruebas suficientemente grande y variado, que permita comprobar el correcto funcionamiento de las diversas operaciones que han de trabajar con los datos. Para las diferentes pruebas y para el correspondiente seguimiento del comportamiento del programa, será necesario realizar un breve estudio de la corrección del programa en cuanto a eficacia (el programa hace lo esperado) y eficiencia en tiempo de ejecución (el programa optimiza el recurso tiempo) y en uso de memoria. Se profundizará en el estudio experimental de la eficiencia en la práctica 2.

El programador habrá de realizar una correcta depuración de entradas al programa. Este último debe ser robusto, y no permitir que ninguna operación falle ante una entrada inesperada. El programa debe ser capaz de responder ante una entrada de un tipo diferente al esperado en cualquier momento de su ejecución, ya sea la recepción de datos desde teclado o desde fichero. El comportamiento ante situaciones especiales puede tratarse de diferentes formas: terminar la ejecución del programa⁵, indicar el error mediante un mensaje, o lanzar una excepción.

⁵Aunque esta opción siempre se debe intentar evitar, puesto que puede suponer pérdida de información. Puede ser adecuado, por ejemplo, en fallos que se producen durante el inicio del programa.

C.6. Informe del desarrollo

Durante el desarrollo de la práctica se irán llenando unas tablas con los tiempos empleados en cada una de las diferentes etapas de la confección de la práctica. Esos datos se reflejarán en horas o minutos, y serán una medida del esfuerzo dedicado a cada apartado. Por ejemplo, se puede utilizar una tabla como la mostrada en la tabla C.1. El objetivo último de estas tablas es reflejar la dedicación personal en cada fase de desarrollo, para poder extraer conclusiones de utilidad. Por lo tanto, la estructura de apartados o medidas tomadas podrá adaptarse según lo que más interese en cada proyecto.

Proyecto: Buscador Web			Fecha de inicio: 21/11/2003		
Programador: Nuria Escrivá López			Fecha de fin: 15/12/2003		
Día/Mes	Análisis	Diseño	Implementación	Validación	TOTAL
3/11	21	10			31
7/11	5	20	10		35
2/11		8	35	15	58
1/12		3		43	46
TOTAL (minutos)	26	41	45	58	170
MEDIAS (porcentaje)	15,3%	24,1%	26,5%	34,1%	100%

Tabla C.1: Dedicación temporal al proyecto, en las distintas fases de desarrollo.

En el informe es conveniente incluir otro tipo de información adicional en relación a las medidas de calidad del software. Por ejemplo, al finalizar el programa se puede llenar una tabla como la mostrada en la tabla C.2.

Proyecto: Buscador Web			Fecha de inicio: 21/11/2003					
Programador: Nuria Escrivá López			Fecha de fin: 15/12/2003					
Nombre de la clase	Número de operaciones	Líneas de código	Reutilizada	Líneas por operación				
	Públicas Privadas Total			Min	Max	Media		
Códigos	4	3	7	58	no	3	13	8,3
Páginas	7	5	12	95	no	7	24	7,9
Palabras	5	4	9	97	no	5	31	10,8
Buscador	9	5	14	253	no	12	63	18
TOTAL	25	17	42	503	0	3	63	12,0
MEDIAS	6,25	4,25	10,5	12,58	0	0,8	32,8	12,0

Tabla C.2: Resumen de clases, funciones y líneas de código, del proyecto.

Sobre estas tablas sería interesante intentar extraer conclusiones, por ejemplo sobre el adecuado reparto de la funcionalidad, la dedicación a las fases iniciales de desarrollo, la modularidad, el logro de los objetivos buscados, etc.

Bibliografía

- [Aho74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [Aho88] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.
- [Baase83] Sara Baase. *Computer algorithms. Introduction to design and analysis*. Addison-Wesley, 1983.
- [Baase00] Sara Baase, Allen Van Gelder. *Computer Algorithms. Introduction to Design and Analysis*. Addison-Wesley, 2000.
- [Brassard90] Brassard, Bratley. *Algorítmica. Concepción y análisis*. Masson, 1990.
- [Brassard97] Brassard, Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 1997.
- [Campos95] Javier Campos Laclaustra. *Estructuras de datos y algoritmos*. Colección Textos Docentes, Prensas Universitarias de Zaragoza, 1995.
- [Collado87] Manuel Collado Machuca, Rafael Morales Fernández, Juan José Moreno Navarro. *Estructuras de datos, realización en Pascal*. Ediciones Díaz de Santos, 1987.
- [Cormen90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [Davis83] Martin D. Davis, Elaine J. Weyuker. *Computability, Complexity, and Languages*. Academic Press, 1983.
- [Drozdek01] Adam Drozdek. *Data Structures and Algorithms in Java*. Brooks/Cole, 2001.
- [Franch94] Manuel Franch Gutiérrez. *Estructuras de datos, Especificación, diseño e implementación*. Ediciones UPC, 1994.
- [Galve93] Javier Galve, Juan C. González, Ángel Sánchez, J. Ángel Velázquez. *Algorítmica, diseño y análisis de algoritmos Funcionales e Imperativos*. Ra-Ma, 1993.
- [Garey79] Michael R. Garey, David S. Johnson. *Computers and intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

- [Gonnet91] G. H. Gonnet, R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. The MIT Press, 1991.
- [Gonzalo98] Julio Gonzalo Arroyo, Miguel Rodríguez Artacho. *Esquemas algorítmicos: enfoque metodológico y problemas resueltos*. Universidad Nacional de Educación a Distancia, 1998.
- [Harel97] David Harel. *Algorithms. The Spirit of Computing*. Addison-Wesley, 1997.
- [Heileman98] Gregory L. Heileman. *Estructuras de Datos, Algoritmos y Programación Orientada a Objetos*. McGraw-Hill, 1998.
- [Hernández01] Roberto Hernández, Juan Carlos Lázaro, Raquel Dormido, Salvador Ros. *Estructuras de Datos y Algoritmos*. Prentice Hall, 2001.
- [Hopcroft01] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. Addison-Wesley, 2001.
- [Hopcroft79] John E. Hopcroft, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Horowitz78] Ellis Horowitz, Sartaj Sahni. *Fundamentals of Computer Algorithms*. Pitman, 1978.
- [Horowitz82] Ellis Horowitz, Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1982.
- [Humphrey01] Watt S. Humphrey. *Introducción al proceso software personal*. Pearson Educación, 2001.
- [Johnson90] David S. Johnson. *A Catalog of Complexity Classes*, in *Handbook of Theoretical Computer Science*, Volume A, Algorithms and Complexity, Jan van Leeuwen ed., pp 68-161. Elsevier, 1990.
- [Kernighan91] Brian W. Kernighan, Dennis M. Ritchie. *El lenguaje de programación C*. Prentice-Hall, 1991.
- [Knuth85] D. E. Knuth. *El arte de programar ordenadores. Vol 1: algoritmos fundamentales*. Reverté, 1985.
- [Knuth87] D. E. Knuth. *El arte de programar ordenadores. Vol 3: clasificación y búsqueda*. Reverté, 1987.
- [Kruse89] Robert L. Kruse. *Estructura de datos y diseño de programas*. Prentice-Hall, 1989.
- [Langsam97] Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum. *Estructuras de datos con C y C++*. Prentice-Hall, 1997.
- [Lewis81] Harry R. Lewis, Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 1981.

- [Manber89] Udi Manber. *Introduction to algorithms. A creative approach.* Addison-Wesley, 1989.
- [Meyer99] Bertrand Meyer. *Construcción de Software Orientado a Objetos. Segunda edición.* Prentice Hall, 1999.
- [Peña98] Ricardo Peña Marí. *Diseño de Programas. Formalismo y Abstracción.* Prentice-Hall, 1998.
- [Rabhi99] Fethi Rabhi, Guy Lapalme. *Algorithms, A Functional Programming Approach.* Addison-Wesley, 1999.
- [Stroustrup98] Bjarne Stroustrup. *El lenguaje de programación C++.* Addison Wesley, 1998.
- [Tenenbaum88] Aaron M. Tenenbaum, Moshe J. Augenstein. *Estructuras de datos en Pascal.* Prentice-Hall, 1988.
- [Troya84] José María Troya Linero. *Análisis y diseño de algoritmos.* VI Escuela de Verano de Informática. AEIA, 1984.
- [Weis95] Mark Allen Weiss. *Estructuras de datos y algoritmos.* Addison-Wesley, 1995.
- [Weis00] Mark Allen Weiss. *Estructuras de datos en Java.* Addison-Wesley, 2000.
- [Wirth80] Niklaus Wirth. *Algoritmos+Estructuras de datos=Programas.* Ediciones del Castillo, 1980.
- [Wirth87] Niklaus Wirth. *Algoritmos y estructuras de datos.* Prentice-Hall, 1987.
- [Wood87] Derick Wood. *Theory of Computation.* John Wiley & Sons, 1987.