

ALGORITMOS Y ESTRUCTURA DE DATOS

LO NUEVO DE ESTA VERSIÓN

- El Pascal ha sido sustituido por Modula-2.
- La última parte del Capítulo 1 se dedica a la **búsqueda**.
- Una sección sobre **árboles de búsqueda con prioridad** completa el capítulo referente a estructuras dinámicas de datos.
- El libro refleja la evolución en el uso de las computadoras y de complicados algoritmos para elaborar y editar automáticamente documentos.
- La sintaxis de Modula-2 se resume en un Apéndice para facilitar su consulta.

Este estudio lúcido, sistemático y penetrante que se hace de las estructuras de datos básicas y dinámicas, sobre la clasificación, algoritmos recursivos, estructuras del lenguaje y compilación muestra la forma en que estos principios básicos se aplican a cualquier área científica y de ingeniería.

Entre sus características el libro:

- Explica la elaboración de programas paso a paso y los expresa en una presentación bien estructurada, detallada y no ambigua.
- Resalta la importancia del análisis del rendimiento y demuestra la forma en que la selección y el refinamiento de los algoritmos se utilizan en forma más eficaz en el diseño de programas.

ISBN 968-880-113-5



9 0000

9 789688 801130

WIRTH

ALGORITMOS

Y ESTRUCTURA DE DATOS

PH
PRENTICE
HALL

ALGORITMOS Y ESTRUCTURA DE DATOS

NIKLAUS
WIRTH

**ALGORITMOS
Y
ESTRUCTURAS DE DATOS**

ALGORITMOS Y ESTRUCTURAS DE DATOS

Niklaus Wirth

ETH Zürich

Traducción:
Juan Carlos Vega Fagoaga
Traductor Técnico

Revisión Técnica:
Gerardo Quiroz Vieyra
Ingeniero en Comunicaciones y Electrónica, ESIME, IPN
Profesor de tiempo completo, UAM-Xochimilco

PRENTICE-HALL HISPANOAMERICANA, S.A.
México, Englewood Cliffs, Nueva Delhi, Wellington, Londres,
Río de Janeiro, Sidney, Singapur, Tokio, Toronto

EDICION EN ESPAÑOL

EDITOR:

SUPERVISOR DE TRADUCCION Y CORRECCION

DE ESTILO:

SUPERVISOR DE PRODUCCION:

DIRECTOR:

José de Jesús Muñoz Zazueta

José C. Pecina Hernández

Patricia Díaz Castañeda

Raymundo Cruzado González

CONTENIDO

PROLOGO 9

1 ESTRUCTURAS DE DATOS FUNDAMENTALES 19

- 1.1 Introducción 19
- 1.2 Concepto del tipo de datos 22
- 1.3 Tipos de datos primitivos 25
- 1.4 Tipos primitivos estándar 27
- 1.5 Tipos de subintervalo 31
- 1.6 Estructura del arreglo 32
- 1.7 Estructura del registro 36
- 1.8 Variantes de estructuras del registro 40
- 1.9 Estructura de conjunto 43
- 1.10 Representación de estructuras de arreglo, registro y conjunto 45
 - 1.10.1 Representación de arreglos 45
 - 1.10.2 Representación de estructuras de registro 47
 - 1.10.3 Representación de conjuntos 47
- 1.11 Estructura de la secuencia 50
 - 1.11.1 Operadores elementales de secuencia 51
 - 1.11.2 Manejo por bufer de secuencias 54
 - 1.11.3 Entrada y salida estándar 59
- 1.12 Búsqueda 62
 - 1.12.1 Búsqueda lineal 62
 - 1.12.2 Búsqueda binaria 63
 - 1.12.3 Búsqueda en tabla 65
 - 1.12.4 Búsqueda directa de cadena 66
 - 1.12.5 Búsqueda de cadena de Knuth-Morris-Pratt 68
 - 1.12.6 Búsqueda de cadena de Boyer-Moore 73
- Ejercicios 77

ALGORITMOS Y ESTRUCTURAS DE DATOS

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio o método, sin autorización escrita del editor.
DERECHOS RESERVADOS © 1987 respecto a la primera edición en español por
PRENTICE-HALL HISPANOAMERICANA, S.A.

Enrique Jacob No. 20, Col. El Conde C.P. 53500

Naucalpan de Juárez , Edo. de México.

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1524.
ISBN 968-880-113-5

Traducido de la primera edición en Inglés de
ALGORITHMS & DATA STRUCTURES

Copyright © MCMLXXXVI, by Prentice Hall, Inc.

ISBN 0-13-022005-1

PROGRAMAS EDUCATIVOS
CALZ. CHABACANO N°. 65 LOCAL A
COL. ASTURIAS, DELEG. CUAUHTEMOC, D.F.
C.P. 06650

100

1995

2 CLASIFICACION 81

- 2.1 Introducción 81
- 2.2 Clasificación de arreglos 84
 - 2.2.1 Clasificación por inserción directa 84
 - 2.2.2 Clasificación por selección directa 87
 - 2.2.3. Clasificación por intercambio directo 89
- 2.3 Métodos de clasificación avanzados 93
 - 2.3.1 Inserción por incremento decreciente 93
 - 2.3.2 Clasificación por árbol 95
 - 2.3.3 Clasificación por partición 100
 - 2.3.4 Obtención de la mediana 105
 - 2.3.5 Una comparación de los métodos de clasificación de arreglos 107
- 2.4 Secuencias de clasificación 110
 - 2.4.1 Mezcla directa 110
 - 2.4.2 Mezcla natural 115
 - 2.4.3 Mezcla balanceada múltiple 121
 - 2.4.4 Clasificación polifásica 126
 - 2.4.5 Distribución de las corridas iniciales 135
- Ejercicios 141

3 ALGORITMOS RECURSIVOS 145

- 3.1 Introducción 145
- 3.2 Cuándo no utilizar recursión 148
- 3.3 Dos ejemplos de programas recursivos 151
- 3.4 Algoritmos de rastreo inverso 158
- 3.5 El problema de las ocho reinas 164
- 3.6 El problema del matrimonio estable 169
- 3.7 El problema de selección óptima 176
- Ejercicios 180

4 ESTRUCTURAS DE INFORMACION DINAMICAS 183

- 4.1 Tipos de datos recursivos 183
- 4.2 Apuntadores 187
- 4.3 Listas lineales 193
 - 4.3.1 Operaciones básicas 193
 - 4.3.2 Listas ordenadas y reorganización de listas 196
 - 4.3.3 Una aplicación: Clasificación topológica 202

4.4 Estructuras de árbol 210

- 4.4.1 Conceptos y definiciones básicos 210
- 4.4.2 Operaciones básicas con árboles binarios 218
- 4.4.3 Búsqueda e inserción en árbol 222
- 4.4.4 Eliminación en un árbol 229
- 4.4.5 Análisis de búsqueda e inserción en árbol 231
- 4.5 Árboles balanceados 234
 - 4.5.1 Inserción en árboles balanceados 235
 - 4.5.2 Eliminación en árboles balanceados 240
- 4.6 Árboles de búsqueda óptimos 245
- 4.7 Árboles B 258
 - 4.7.1 Árboles B multicaminos 259
 - 4.7.2 Árboles B binarios 271
- 4.8 Árboles de búsqueda con prioridad 278

Ejercicios 282

5 TRANSFORMACIONES DE LLAVES (HASHING) 287

- 5.1 Introducción 287
- 5.2 Elección de una función de transformación de llaves (HASH) 289
- 5.3 Manejo de colisiones 290
- 5.4 Análisis de la transformación de llaves 295
- Ejercicios 298

APENDICES 299

- A Conjunto de caracteres ASCII 297
- B Sintaxis de Modula-2 300

INDICE 304

PROLOGO

En años recientes el tema de la *programación de computadoras* ha sido reconocido como una disciplina cuyo dominio es fundamental y decisivo para el éxito de muchos proyectos de ingeniería y que es susceptible de estudio y presentación científicos. Ha dejado de ser una simple ocupación para convertirse en una disciplina académica. Las primeras contribuciones sobresalientes para su creación fueron aportadas por E. W. Dijkstra y C. A. R. Hoare. El tratado *Notes on Structured Programming* [1] de Dijkstra abrió un nuevo panorama de la programación como un tema de interés científico y reto intelectual, y además sirvió como base de la denominada “revolución” en la programación. El tratado *Axiomatic Basis of Computer Programming* [2] de Hoare mostró en forma lúcida que los programas se prestan a un análisis riguroso basado en el razonamiento matemático. Ambas obras prueban convincentemente que muchos errores de programación pueden evitarse haciendo que los programadores conozcan los métodos y técnicas que se aplican hasta ahora intuitiva y, con frecuencia, inconscientemente. Estos escritos centraron su atención en los aspectos de la composición y el análisis de los programas o bien, en forma más explícita, en la estructura de los algoritmos representados por textos del programa. No obstante, es evidente que un enfoque sistemático y científico a la construcción de programas tiene conexión primaria en el caso de programas grandes y complejos que incluyan conjuntos de datos complicados. En consecuencia, una metodología de programación se limita asimismo a contener todos los aspectos de la estructuración de los datos. Los *programas*, después de todo, son formulaciones concretas de *algoritmos* abstractos basados en ciertas representaciones y *estructuras de datos*. Una contribución sobresaliente que permitió poner orden en la abrumadora diversidad de terminología y conceptos referentes a estructuras de datos fue hecha por Hoare en su tratado *Notes on Data Structuring* [3]. Este escrito demostró que las decisiones acerca de la estructuración de datos no pueden tomarse sin tener conocimiento de los algoritmos que se aplican a los datos y que, viceversa, la estructura y selección de los algoritmos con frecuencia dependen mucho de la estructura de los datos subyacentes. En resumen, los temas de composición de programas y estructuras de datos se interrelacionan y son inseparables.

No obstante, este libro comienza con un capítulo acerca de las estructuras de datos por dos razones. Primero, uno tiene una sensación intuitiva de que los datos preceden a los

algoritmos: se deben estudiar algunos temas antes de poder efectuar operaciones con ellos. Segundo, y ésta es la razón más inmediata, este libro supone que el lector está familiarizado con las nociones básicas de la programación de computadoras. Sin embargo, es tradicional y razonable que los cursos introductorios de programación se concentren en los algoritmos que operan en estructuras de datos relativamente simples. En consecuencia, parece adecuado un capítulo introductorio sobre las estructuras de datos.

En todo el libro, y particularmente en el capítulo 1, se sigue la teoría y terminología expuesta por Hoare y realizada en el lenguaje de programación PASCAL [4]. La esencia de este aparato teórico es que los datos, ante todo, representan abstracciones de fenómenos reales y se formulan de preferencia como estructuras abstractas que no necesariamente se elaboran en lenguajes comunes de programación. En el proceso de la construcción de programas la representación de los datos se refina gradualmente (junto con el refinamiento del algoritmo) para cumplir cada vez más con las restricciones impuestas por un sistema de programación disponible [5]. Por lo tanto, postulamos varios de los principios básicos de construcción de estructuras de datos, llamados *estructuras fundamentales*. Es más importante que sean construcciones que son fácilmente aplicables en computadoras reales, ya que sólo en este caso pueden considerarse como los verdaderos elementos de una representación de datos real, como las moléculas que emergen de la etapa final de los refinamientos de la descripción de los datos. Estos son el *registro*, el *arreglo* (de tamaño fijo) y el *conjunto*. No es extraño que estos principios básicos de construcción correspondan a nociones matemáticas que son también fundamentales.

Una piedra angular de esta teoría de estructuras de datos es la distinción que se hace entre estructuras fundamentales y “avanzadas”. Las primeras son las moléculas, constituidas por átomos, que son las componentes de las segundas citadas. Las variables de una estructura fundamental solamente cambian su valor, pero nunca su estructura y tampoco el conjunto de valores que pueden tomar. A consecuencia de esto, el espacio de almacenamiento que ocupan permanece constante. Sin embargo, las estructuras “avanzadas” se caracterizan por su cambio de valor y estructura durante la ejecución de un programa. Por tanto, se necesitan técnicas más complejas para su aplicación. La secuencia figura como un elemento híbrido en esta clasificación. Ciertamente varía en longitud, pero ese cambio de estructura es de naturaleza trivial. Ya que la secuencia desempeña un papel verdaderamente fundamental en prácticamente todos los sistemas de computación, su estudio se incluye en el capítulo 1.

El segundo capítulo trata acerca de algoritmos de clasificación. Exhibe una diversidad de métodos, los cuales sirven todos al mismo objetivo. El análisis matemático de algunos de estos algoritmos muestra las ventajas y desventajas de los métodos y entera al programador de la importancia del análisis en la elección de soluciones adecuadas a un problema determinado. La división en métodos para clasificar arreglos y otros para clasificar archivos (que frecuentemente se llaman clasificación interna y externa) muestra la influencia crucial que la representación de los datos ejerce sobre la elección de algoritmos aplicables y sobre su complejidad. El espacio asignado a la clasificación no sería tan grande si no fuera por el hecho de que la clasificación constituye un vehículo ideal para ilustrar tantos principios y problemas de programación que ocurren en muchas otras

aplicaciones. Con frecuencia parece que se podría integrar un curso completo de programación incluyendo sólo ejemplos de clasificación.

Otro tema, que generalmente se omite en cursos introductorios de programación pero que desempeña un papel importante en la concepción de muchas soluciones algorítmicas, es la recursión. Por lo tanto, el tercer capítulo está dedicado a los *algoritmos recursivos*. La recursión se muestra como una generalización de la repetición (iteración) y constituye un concepto importante y poderoso de la programación. En muchas clases de programación que se imparten, desafortunadamente la exemplifican casos en los cuales bastaría la iteración simple. En cambio, el capítulo 3 se concentra en varios ejemplos de problemas en los cuales la recursión hace posible una formulación más natural de una solución, en tanto que el uso de la iteración nos llevaría a programas oscuros y enredados. La clase de algoritmos de *rastreo inverso* surge como una aplicación ideal de la recursión, pero los candidatos más obvios para el uso de la recursión son los algoritmos que operan en datos cuya estructura se define en forma recursiva. Estos casos se estudian en los dos últimos capítulos, para los cuales el tercero ofrece antecedentes de estudio.

El capítulo 4 trata de las *estructuras de datos dinámicas*, es decir, los datos que cambian su estructura durante la ejecución del programa. Se muestra que las estructuras de datos recursivas son una subclase importante de las estructuras dinámicas que comúnmente se utilizan. Aunque una definición recursiva es natural y posible en estos casos, por lo general no se emplea en la práctica. En su lugar, el mecanismo que se usa en su implantación es evidente para el programador ya que se le obliga a utilizar la referencia explícita o bien *variables apuntadoras*. Este libro se apega a esta técnica y refleja el estado actual de ésta: el capítulo 4 está dedicado a la programación con apuntadores, hacia listas, árboles y ejemplos que incluyen combinaciones de datos todavía más complicadas. Presenta lo que con frecuencia se denomina (y un tanto incorrectamente) *procesamiento de listas*. Una cantidad considerable de espacio se dedica a organizaciones de árbol y en particular a árboles de búsqueda. El capítulo finaliza con una presentación de tablas examinadoras, también llamadas códigos de “combinación”, que con frecuencia se prefieren sobre los árboles de búsqueda. Esto hace posible la comparación de dos técnicas fundamentalmente distintas para una aplicación que se encuentra con frecuencia.

La programación es una actividad *constructiva*. ¿Cómo puede impartirse una actividad constructiva e inventiva? Un método consiste en cristalizar los principios básicos de composición de muchos casos y exhibirlos en forma sistemática. Pero la programación es un campo de amplia variedad que a menudo comprende actividades intelectuales complejas. La creencia de que nunca podrá condensarse en una clase de fórmula de enseñanza es equivocada. Lo que persiste en nuestro arsenal de métodos de enseñanza es la cuidadosa selección y presentación de ejemplos modelo. Naturalmente, no debemos creer que todas las personas pueden adquirir el mismo conocimiento del estudio de ejemplos. Es la característica de este enfoque la que se deja al estudiante, a su agilidad e intuición. Este es particularmente el caso de los programas de ejemplo relativamente intrincados y largos. Su inclusión en este libro no es accidental. Los programas más largos son los que prevalecen en la práctica y resultan ser mucho más adecuados para exhibir ese ingrediente elusivo pero esencial denominado estilo y estructura ordenada. También tienen el fin

de servir como ejercicios en el arte de la lectura de programas, que con mucha frecuencia es despreciada en favor de la escritura del programa. Esta es una motivación importante oculta en la inclusión de programas mayores como ejemplos en su totalidad. El lector es guiado a través de una elaboración gradual del programa; se le ofrecen diversas pistas en la evolución de un programa, con lo cual esta elaboración se hace manifiesta como un *refinamiento paso a paso* de los detalles. Considero esencial que los programas se muestren en forma final, con suficiente atención a los detalles ya que en la programación los errores están en los detalles. Aunque la mera presentación del principio de un algoritmo y su análisis matemático pueden ser estimulantes e interesantes para una mente académica, parece deshonesto al ingeniero. Por lo tanto, me he apagado estrictamente a la regla de presentar los programas finales en un lenguaje en el cual puedan ser corridos realmente en una computadora.

Desde luego, esto trae consigo el problema de hallar una forma que al mismo tiempo sea ejecutable en la máquina y lo suficientemente independiente de ella para incluirse en dicho texto. En este aspecto, ningún lenguaje ampliamente utilizado ni notación abstracta probaron ser adecuados. El lenguaje Pascal ofrece un compromiso adecuado: ha sido creado con esta aspiración y por tanto se utiliza en todo el libro. Los programadores pueden entender fácilmente los programas que conocen bien con algún otro lenguaje de alto nivel, como ALGOL 60 o bien PL/I, ya que es fácil comprender la notación del Pascal mientras se avanza en el estudio del libro. Sin embargo, esto no quiere decir que alguna propagación no sería beneficiosa. El libro *Systematic Programming* [6] ofrece un antecedente ideal ya que se basa también en la notación del lenguaje Pascal. Sin embargo, este libro no se escribió para fungir como manual del lenguaje Pascal; existen otros más adecuados para este fin [7].

Este libro es una condensación, y al mismo tiempo una elaboración, de varios cursos de programación que se imparten en el Federal Institute of Technology (ETH) en Zürich. Debo muchas ideas y puntos de vista expresados en este libro a pláticas con mis colaboradores del ETH. En particular, deseo agradecer a Mr. Sandmayr por su cuidadosa lectura del material y a Miss Heidi Theiler y D. Wirth por su cuidado y paciencia en el mecanografiado del libro. También debo hacer mención de la estimulante influencia ejercida por reuniones con los grupos de trabajo 2.1 y 2.3 de IFIP y particularmente de los muchos argumentos memorables que experimenté en estas ocasiones con E. W. Dijkstra y C. A. R. Hoare. Por último, pero no menos importante, ETH ofreció generosamente el medio de trabajo e instalaciones de computación sin los cuales la elaboración de este texto habría sido imposible.

Zürich, agosto de 1975

N. Wirth

4. N. Wirth. The Programming Language Pascal. *Acta Informatica*, 1, No. 1 (1971), 35-63.
5. N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14, No. 4 (1971), 221-27.
6. N. Wirth. *Systematic Programming*. (Englewood Cliffs, N. J. Prentice-Hall, Inc., 1973.)
7. K. Jensen and N. Wirth, *PASCAL-User Manual and Report*. (Berlin, Heidelberg, New York; Springer-Verlag, 1974).

1. In *Structured Programming*. O.-J. Dahl, E. W. Dijkstra, C.A.R. Hoare. F. Genuys, Ed. (New York; Academic Press, 1972), pp. 1-82.
2. In *Comm. ACM*, 12, No. 10 (1969), 576-83.
3. In *Structured Programming*, pp. 83-174.

PROLOGO A LA EDICION DE 1986

Esta nueva edición incorpora muchas revisiones de detalles y varios cambios de naturaleza más significativa. Todos fueron motivados por experiencias vividas en los diez años siguientes a la aparición de la primera edición. La mayor parte del contenido y estilo del libro se han conservado. Se resumen a continuación las innovaciones más importantes.

El cambio principal en todo el libro concierne al lenguaje de programación que se utiliza para expresar los algoritmos. El Pascal ha sido sustituido por *Modula-2*. A pesar que este cambio no ejerce influencia fundamental en la presentación de los algoritmos, la elección se justifica por las estructuras sintácticas más simples y elegantes del Modula-2, que a menudo nos llevan a una representación más lúcida de la estructura de un algoritmo. Aparte de esto, pareció aconsejable utilizar una notación que gane rápida aceptación en parte de una vasta comunidad, ya que se ajusta bien a la elaboración de sistemas de programación grandes. No obstante, el hecho de que Pascal es el ancestro del Modula es muy evidente y facilita la tarea de una transición. La sintaxis de Modula se resume en el apéndice para su fácil localización.

Una consecuencia directa de este cambio del lenguaje de programación es que la sección 1.11 sobre estructura de archivos secuenciales ha sido reescrita. Modula-2 no ofrece un tipo de archivo integrado. La sección 1.11 revisada presenta el concepto de *secuencia* (o sucesión) como una estructura de datos en forma más general y además un conjunto de módulos de programa que incorporan el concepto de secuencia en Modula-2 en forma específica.

La última parte del capítulo 1 es nueva. Está dedicada al tema de la búsqueda y, comenzando con la búsqueda lineal y binaria, nos lleva a algunos algoritmos veloces de búsqueda de cadena recién inventados. En esta sección en particular se utilizan afirmaciones e invariantes de repetición para demostrar la corrección de los algoritmos que se presentan.

Una nueva sección acerca de *árboles de búsqueda con prioridad* redondea el capítulo sobre las estructuras de datos dinámicas. Asimismo, esta especie de árboles era desconocida cuando se publicó la primera edición. Hacen posible una representación económica y una rápida búsqueda de conjuntos de puntos en un plano.

El capítulo 5 en su totalidad, como apareció en la primera edición, se ha omitido. Se sentía que el tema de la construcción del compilador estaba un tanto aislada de los capítulos anteriores y en su lugar ameritaría un estudio más profundo en un libro especializado en el tema.

Finalmente, la aparición de la nueva edición refleja una evolución que ha influido profundamente en las publicaciones de los últimos 10 años: el uso de las computadoras y algoritmos complicados para elaborar y editar automáticamente documentos. Este libro fue editado y organizado por el autor con ayuda de una computadora *Lilith* y su editor de documentos *Lara*. Sin estas herramientas, no sólo el libro hubiera resultado más costoso, sino ciertamente no se habría terminado todavía.

Palo Alto

N. Wirth

NOTACION

Las siguientes notaciones, adoptadas de publicaciones de E. W. Dijkstra, se utilizan en este libro.

En las expresiones lógicas, el carácter $\&$ denota conjunción y se pronuncia como *y*. El carácter \sim representa negación y se pronuncia *no*. Las letras **A** y **E** negritas redondas se utilizan para denotar los calificadores universales y existenciales. En las fórmulas siguientes, la parte de la izquierda es la notación que se usa y se define aquí en términos de la parte derecha. Nótese que las partes de la izquierda no utilizan el símbolo “...”, que estimula la intuición del lector.

$$A_i: m \leq i < n : P_i \equiv P_m \& P_{m+1} \& \dots \& P_{n-1}$$

Los símbolos P_i son predicados y la fórmula afirma que *para todos* los índices i que varían de cierto valor m a pero excluyendo un valor n , P_i se conserva.

$$E_i: m \leq i < n : P_i \equiv P_m \text{ o bien } P_{m+1} \text{ o bien } \dots \text{ o bien } P_{n-1}$$

Los símbolos P_i son predicados y la fórmula afirma que *para algunos* índices i que varían de cierto valor m a pero excluyendo un valor n , P_i se conserva.

$$Si: m \leq i < n : x_i = x_m + x_{m+1} + \dots + x_{n-1}$$

$$\text{MIN } i: m \leq i < n : x_i = \text{mínimo}(x_m, x_{m+1}, \dots, x_{n-1})$$

$$\text{MAX } i: m \leq i < n : x_i = \text{máximo}(x_m, x_{m+1}, \dots, x_{n-1})$$

Utilizando calificadores, los operadores *mín* y *máx* se pueden expresar de la manera siguiente:

$$\begin{aligned} \text{MIN } i: m \leq i < n : x_i = \text{mín} &\equiv \\ (\text{A}_i: m \leq i < n : \text{mín} \leq x_i) \& \& (\text{E}_i: m \leq i < n : \text{mín} = x_i) \end{aligned}$$

$$\begin{aligned} \text{MAX } i: m \leq i < n : x_i = \text{máx} &\equiv \\ (\text{A}_i: m \leq i < n : \text{máx} \geq x_i) \& \& (\text{E}_i: m \leq i < n : \text{máx} = x_i) \end{aligned}$$

1. ESTRUCTURAS DE DATOS FUNDAMENTALES

1.1. INTRODUCCION

La computadora digital moderna fue inventada e ideada como un dispositivo que debe facilitar y acelerar operaciones de cálculo complicadas y que consumen mucho tiempo. En la mayoría de las aplicaciones su capacidad de almacenar y acceder a grandes cantidades de información desempeña la parte dominante y se considera como su característica principal; su capacidad de contar o computar, es decir, calcular y realizar operaciones aritméticas, en muchos casos se ha vuelto casi irrelevante.

En todos estos casos, la enorme cantidad de información que se procesa en algún sentido representa una *abstracción* de una parte de la realidad. La información de que se dispone para procesar en la computadora consta de un conjunto determinado de *datos* acerca del problema real, es decir, el conjunto que se considera relevante para el problema que se tiene a la mano, aquel conjunto del cual se cree pueden derivarse los resultados deseados. Los datos representan una abstracción de la realidad en el sentido de que ciertas propiedades y características de los objetos reales son ignorados por ser periféricos e irrelevantes para el problema específico. Una abstracción es por tanto también una *simplificación* de hechos.

Como ejemplo podemos considerar el archivo personal de una empresa. Todos los empleados se representan (compendian) en él por medio de un conjunto de datos relevante para la empresa o bien para sus procedimientos contables. Este conjunto puede comprender alguna identificación del empleado, por ejemplo, su nombre y salario. Pero muy probablemente no comprenderá información como el color del cabello, peso y estatura.

Al resolver un problema con o sin una computadora se necesita elegir una abstracción de la realidad, o sea definir un conjunto de datos que representará la situación real. Esta elección debe ser guiada por el problema que debe resolverse. Luego sigue una elección de la representación de esta información. Esta elección es guiada por la herramienta que se usa para resolver el problema, es decir, por los recursos que ofrece la computadora. En muchos casos estas dos etapas no son completamente separables.

La elección de la representación de los datos con frecuencia es considerablemente difícil y no se determina sólo por los recursos de que se dispone. Siempre debe tomarse a la luz de las operaciones que se realicen con los datos. Un buen ejemplo de esto es la representación de números, que son por sí mismos abstracciones de propiedades de objetos que se caracterizarán. Si la adición es la única operación (o al menos la dominante) que debe efectuarse, una forma adecuada de representar el número n consiste en escribir n trazos. La regla de adición en esta representación es en realidad muy obvia y simple. Los numerales romanos se basan en el mismo principio de simplicidad y las reglas de adición son análogamente directas para números pequeños. Por el otro lado, la representación con numerales arábigos requiere de reglas que escapan de lo obvio (para números pequeños) y deben ser memorizadas. Sin embargo, la situación se invierte cuando consideramos la adición de números grandes o bien la multiplicación y división. La descomposición de estas operaciones en otras más simples es mucho más sencilla en el caso de representación por medio de numerales arábigos, debido al principio de estructuración sistemática que se basa en el valor posicional de los dígitos.

Generalmente se sabe que las computadoras utilizan una representación interna basada en dígitos binarios (bits). Esta representación es inadecuada para los seres humanos debido al número grande de cifras comprendidas, pero es más adecuada para circuitos electrónicos porque los valores 0 y 1 pueden representarse en forma ventajosa y confiable por la presencia o ausencia de corrientes eléctricas, carga eléctrica o campos magnéticos.

De este ejemplo podemos observar también que la cuestión de la representación a menudo trasciende varios niveles de detalle. Dado el problema de representación, por decir algo, de la posición de un objeto, la primera decisión puede llevar a la elección de un par de números reales en coordenadas cartesianas o bien polares. La segunda decisión puede conducirnos a una representación de punto flotante, donde todo número real x consta de un par de enteros que simbolizan una función f y un exponente e a una cierta base (tal que $x = f \cdot 2^e$). La tercera decisión, basada en el conocimiento de que los datos serán almacenados en una computadora, nos puede llevar a una representación binaria de posición de enteros y la decisión final podría consistir en la representación de dígitos binarios por la dirección del flujo magnético en un dispositivo de almacenamiento magnético. Sin duda la primera decisión en esta cadena se ve principalmente influida por la situación del problema y las sucesivas son progresivamente dependientes de la herramienta y su tecnología. Así, difícilmente puede pedirse que un programador decida qué representación numérica utilizar o las características del dispositivo de almacenamiento. Estas decisiones de nivel inferior pueden dejarse a los diseñadores de equipo de computación, quienes disponen de la información más significativa sobre tecnología en uso con la cual hacer una elección sensible que será aceptable para todas (o casi todas) las aplicaciones donde intervengan los números.

En este contexto, el significado de los *lenguajes de programación* se vuelve apparente. Un lenguaje de programación representa a una computadora abstracta capaz de interpretar los términos que se utilizan en este lenguaje, los cuales pueden contener un cierto nivel de abstracción de los objetos usados por la máquina real. Así, el programador que utiliza un lenguaje de nivel superior quedará liberado de cuestiones referentes a la representación numérica, si el número es un objeto elemental en el reino de este lenguaje.

La importancia de emplear un lenguaje que ofrece un conjunto conveniente de abstracciones básicas comunes a muchos problemas de procesamiento de datos radica principalmente en el área de la confiabilidad de los programas resultantes. Es más fácil diseñar un programa basado en el razonamiento con nociones bien conocidas de números, conjuntos, secuencias y repeticiones que en bits, unidades de almacenamiento y saltos. Por supuesto, una computadora real representa todos los datos, sean números, conjuntos o secuencias como una masa grande de bits. Pero esto es irrelevante para el programador en tanto no tenga que preocuparse por los detalles de la representación de las abstracciones elegidas y en tanto pueda confiar en que la representación correspondiente elegida por la computadora (o compilador) es razonable para los fines propuestos.

Cuanto más próximas estén a una cierta computadora las abstracciones, más fácil será hacer una elección de la representación para el ingeniero o implantador del lenguaje y mayor será la probabilidad de que una sola elección sea adecuada para todas (o casi todas) las aplicaciones imaginables. Este hecho fija límites precisos al grado de abstracción de una computadora real determinada. Por ejemplo, no tendría sentido incluir objetos geométricos como elementos de datos básicos en un lenguaje de uso general, ya que su representación adecuada dependerá considerablemente, debido a su complejidad inherente, de las operaciones que se apliquen a estos objetos. La naturaleza y frecuencia de estas operaciones no será conocida, no obstante, por el diseñador de un lenguaje de uso general y su compilador y cualquier elección que haga el diseñador puede ser inapropiada para algunas aplicaciones.

En este libro estas deliberaciones determinan la elección de la notación para la descripción de algoritmos y sus datos. Con claridad, se desean utilizar nociones bien conocidas de matemáticas, como números, conjuntos, secuencias, etcétera, en vez de entidades que dependen de la computadora como cadenas de bits. Pero igualmente deseamos hacer uso de una notación para la cual hay compiladores efectivos. Es igualmente imprudente utilizar un lenguaje casi orientado a la máquina y dependiente de ella, así como es de poca ayuda describir programas en una notación abstracta que deje problemas de representación sin solución. El lenguaje de programación Pascal fue diseñado tras un intento por hallar o determinar un equilibrio entre estos extremos y el lenguaje Modula-2 es el resultado de la experiencia de 10 años con el Pascal [1-3]. Este conserva los conceptos básicos del Pascal e incorpora algunas mejoras y extensiones; se utiliza en todo este libro [1-5]. Modula-2 se ha realizado con muy buenos resultados en varias computadoras y se ha demostrado que la notación se apega suficientemente a máquinas reales en tanto que las características elegidas y sus representaciones se pueden explicar con claridad. El lenguaje se parece a otros y, por tanto, las lecciones que se imparten aquí pueden aplicarse a su uso.

1.2. CONCEPTO DEL TIPO DE DATOS

En matemáticas se acostumbra clasificar las variables de acuerdo con ciertas características importantes. Se hacen distinciones claras entre variables reales, complejas y lógicas o bien entre variables que representan valores individuales, conjuntos de valores o conjuntos de conjuntos; o bien entre funciones, funcionales, conjuntos de funciones, etcétera. Esta noción de clasificación es igualmente, si no es que más, importante en el procesamiento de datos. Nos apegaremos al principio de que toda constante, variable, expresión o función es de cierto *tipo*. Este tipo caracteriza esencialmente el conjunto de valores al cual pertenece una constante o bien que puede ser tomado por una variable o expresión o quizás que puede ser generado por una función.

En libros de matemáticas el tipo de una variable generalmente se deduce a partir del tipo de letra sin tomar en consideración el contexto; esto no es viable en programas de computadora. Por lo general se dispone de un tipo de letra en el equipo de computación (es decir, letras latinas). Por ello se acepta generalmente la regla de que el tipo asociado se hace explícito en una *declaración* de la constante, variable o función y que esta declaración precede textualmente a la aplicación de esa constante, variable o bien función. Esta regla es muy importante si se considera el hecho de que un compilador tiene que hacer una elección de la representación del objeto en el espacio de almacenamiento de una computadora. Evidentemente, la cantidad de almacenamiento asignada a una variable tendrá que ser elegida de acuerdo con el tamaño del intervalo de valores que la variable pueda tomar. Si el compilador conoce esta información, se puede evitar la así denominada asignación dinámica de almacenamiento. Esta es con frecuencia la clave hacia una solución eficaz de un algoritmo. Las características principales del concepto de tipo que se utiliza en todo este libro y que se incluye en el lenguaje de programación Modula-2, son las siguientes [1-2]:

1. Un tipo de datos determina el conjunto de valores al cual pertenece una constante o el cual puede ser tomado por una variable o por una expresión; o bien que puede ser generado por un operador o por una función.
2. El tipo de un valor representado por una constante, variable o expresión puede derivarse de su forma o de su declaración sin que se necesite ejecutar el proceso de cálculo.
3. Cada operador o función aguarda la llegada de argumentos de un tipo fijo y da como resultado un tipo fijo. Si un operador admite argumentos de varios tipos (por ejemplo, $+$ se utiliza para indicar la adición de números enteros y reales), entonces el tipo del resultado se puede determinar de las reglas específicas del lenguaje.

En consecuencia, un compilador puede emplear esta información referente a tipos para verificar la legalidad de diversas construcciones. Por ejemplo, la asignación equivocada de un valor booleano (lógico) a una variable aritmética puede ser detectada sin ejecutar el programa. Este tipo de redundancia en el texto del programa es de extrema utilidad como auxiliar en la elaboración de programas y debe considerarse como la ventaja principal de lenguajes de alto nivel adecuados sobre el código de máquina (o bien código en-

samblador simbólico). Sin duda los datos se representarán finalmente por medio de un número considerable de dígitos binarios, sin tener en cuenta si el programa había sido inicialmente concebido o no en un lenguaje de alto nivel utilizando el concepto de tipo o en código ensamblador sin tipo. Para la computadora, el almacenamiento es una masa homogénea de bits sin estructura. Pero es exactamente esta estructura abstracta, que por sí sola permite a los programadores humanos reconocer el significado en un paisaje monótono de la memoria de una computadora.

La teoría presentada en este libro y el lenguaje de programación Modula-2 especifican ciertos métodos de definición de los tipos de datos. En muchos casos nuevos tipos de datos se definen en términos de tipos de datos anteriormente definidos. Los valores de un tipo como éste generalmente son conglomerados de valores de componentes de los tipos constituyentes anteriormente definidos y se dice que son *estructurados*. Si hay solamente un tipo constituyente, es decir, si todas las componentes son del mismo tipo constituyente, entonces éste se conoce como el *tipo base*. El número de diferentes valores que pertenecen a un tipo T se llama su *cardinalidad*. Esta da la medida de la cantidad de almacenamiento que se necesita para representar una variable x del tipo T, simbolizada por $x:T$.

Ya que los tipos constituyentes pueden volver a ser estructurados, pueden constituirse jerarquías completas de estructuras pero, con claridad, las componentes últimas de una estructura son atómicas. Por lo tanto, se necesita ofrecer una notación que introduzca de igual manera estos tipos primitivos y no estructurados. Un método directo consiste en *enumerar* los valores que constituirán el tipo. Por ejemplo, en un programa referente a figuras geométricas planas, podemos introducir un tipo primitivo llamado *forma*, cuyos valores pueden representarse por los identificadores *rectángulo*, *cuadrado*, *elipse*, *circunferencia*. Pero además de estos tipos definidos por el programador, tendrá que haber algunos tipos estándar predefinidos. Por lo general incluyen números y valores lógicos. Si existe un ordenamiento entre los valores individuales, entonces se dice que el tipo es *ordenado* o bien *escalar*. En Modula-2, todos los tipos no estructurados son ordenados; en el caso de la enumeración explícita, los valores se suponen ordenados por su secuencia de enumeración.

Con esta herramienta disponible, es posible definir tipos primitivos y construir conglomerados, tipos estructurados que se limitan a un grado arbitrario de inserción o anidamiento. En la práctica, no es suficiente tener sólo un método general de combinación de tipos constituyentes en una estructura. Considerando problemas prácticos de representación y uso, un lenguaje de programación de uso general debe ofrecer varios *métodos de estructuración*. En un sentido matemático, son equivalentes; difieren en los operadores en que se dispone para seleccionar componentes de estas estructuras. Los métodos básicos de estructuración que se presentan aquí son el *arreglo*, el *registro*, el *conjunto* y la *secuencia*. Estructuras más complicadas generalmente no se definen como tipos estáticos, sino que en cambio son generados dinámicamente durante la ejecución del programa, cuando pueden variar en tamaño y forma. Dichas estructuras son el tema del capítulo 4 e incluyen listas, anillos, árboles y gráficas generales finitas.

Las variables y tipos de datos se presentan en un programa a fin de utilizarlos en los procesos de cálculo. Para lograr esto se debe disponer de un conjunto de operadores. Pa-

ra cada tipo de datos estándar un lenguaje de programación ofrece un cierto conjunto de operadores primitivos estándar y, de igual manera, con cada método de estructuración, una operación y notación diferentes para seleccionar una componente. La tarea de la composición de operaciones a menudo se considera como el núcleo del arte de la programación. Sin embargo, más adelante se hará evidente que la adecuada composición de los datos es igualmente fundamental y esencial.

Los operadores básicos más importantes son la *comparación* y la *asignación*, es decir, la prueba de igualdad (y de orden en el caso de tipos ordenados) y el comando para reforzar la igualdad. La diferencia fundamental entre estas dos operaciones se destaca por la clara distinción que se hace en su representación en todo el libro.

Prueba de igualdad:	$x = y$	(una expresión con valor TRUE o FALSE)
Asignación a x :	$x := y$	(un enunciado o instrucción que hace que x sea igual a y)

Estos operadores fundamentales están definidos para muchos tipos de datos, pero debe notarse que su ejecución puede implicar una cantidad sustancial de esfuerzo de cálculo, si los datos son extensos y altamente estructurados.

Para los tipos de datos primitivos estándar, postulamos no sólo la disposición de la asignación y la comparación, sino también un conjunto de operadores para crear (calcular) nuevos valores. Así, se presentan las operaciones estándar de la aritmética para tipos numéricos y los operadores elementales de la lógica proposicional para valores lógicos.

1.3. TIPOS DE DATOS PRIMITIVOS

Un nuevo tipo primitivo se puede definir por la enumeración de los diferentes valores que pertenecen a él. A dicho tipo se le llama *tipo de enumeración*. Su definición tiene la forma

$$\text{TYPE } T = (c_1, c_2, \dots, c_n) \quad (1.1.)$$

T es el nuevo identificador del tipo y c_i son los nuevos identificadores de las constantes. La cardinalidad de T es $\text{card}(T) = n$.

EJEMPLOS

TYPE forma = (rectángulo, cuadrado, elipse, circunferencia)
 TYPE color = (rojo, amarillo, verde)
 TYPE sexo = (masculino, femenino)
 TYPE BOOLEAN = (FALSE, TRUE)
 TYPE diáadelasemana = (lunes, martes, miércoles, jueves, viernes, sábado, domingo)
 TYPE monedas = (franco, marco, libra, dólar, chelín, lira, florín, corona, rublo, cruzeiro, yen)
 TYPE destino = (infierno, purgatorio, paraíso)
 TYPE vehículo = (tren, autobús, automóvil, barco, avión)
 TYPE rango = (recluta, cabo, sargento, teniente, capitán, mayor, coronel, general)
 TYPE objeto = (constante, tipo, variable, procedimiento, módulo)
 TYPE estructura = (arreglo, registro, conjunto, secuencia)
 TYPE condición = (manual, descargada, paridad, torcida)

La definición de estos tipos presenta no sólo un nuevo identificador de tipos, sino que al mismo tiempo el conjunto de identificadores que representan los valores del nuevo tipo. Estos identificadores pueden, por tanto, utilizarse como constantes en el programa completo y resaltan considerablemente su facilidad de entendimiento. Si, para poner un ejemplo, introducimos las variables s , d , r y b :

```

VAR s: sexo
VAR d: diáadelasemana
VAR r: rango
VAR b: BOOLEAN
  
```

entonces son posibles las siguientes instrucciones de asignación:

```

s := masculino
d := domingo
r := mayor
b := TRUE
  
```

Evidentemente, son mucho más informativas que sus contrapartes

$s := 1 \ d := 7 \ r := 6 \ b := 2$

las cuales se basan en la suposición de que s, d, r y b están definidas como enteros y que las constantes se mapean en los números naturales en el orden de su enumeración. Además, un compilador puede verificar contra el uso inconsistente de los operadores. Por ejemplo, dada la declaración de s anterior, la instrucción o enunciado $s := s + 1$ no tendría sentido.

Sin embargo, si recordamos que las enumeraciones son ordenadas, resulta conveniente introducir operadores que generen el sucesor y predecesor de su argumento. Por lo tanto, postulamos los siguientes operadores estándar, los cuales asignan a su argumento su sucesor y predecesor, respectivamente:

$$\text{INC}(x) \quad \text{DEC}(x) \quad (1.2)$$

1.4. TIPOS PRIMITIVOS ESTÁNDAR

Los tipos primitivos estándar son aquellos que se tienen a disposición en muchas computadoras como características integradas. Incluyen los números enteros, valores lógicos de verdad y un conjunto de caracteres imprimibles. En muchas computadoras los números fraccionarios se incorporan también, junto con las operaciones aritméticas comunes. Estos tipos se simbolizan por medio de los identificadores

INTEGER, CARDINAL, REAL, BOOLEAN, CHAR

El tipo INTEGER (entero) comprende un subconjunto de números enteros cuyo tamaño puede variar entre los diversos sistemas de computación. Si una computadora utiliza n bits para representar un entero en notación de complemento de 2, entonces los valores posibles de x deben cumplir la desigualdad $-2^{n-1} \leq x < 2^{n-1}$. Se supone que todas las operaciones con datos de este tipo son exactas y corresponden a las leyes ordinarias de la aritmética y que el proceso de cálculo será interrumpido en el caso de un resultado que quede fuera del subconjunto representable. Este evento se llama *desbordamiento*. Los operadores estándar son las cuatro operaciones aritméticas básicas de adición (+), sustracción (-), multiplicación (*) y división (/, DIV).

Aquí distinguiremos entre *aritmética entera Euleriana* y aritmética de módulos. En la primera, la división se simboliza mediante una diagonal y el residuo de la división por el operador REM. Sean q = m/n el cociente y r = m REM n el residuo. Entonces, las relaciones

$$q * n + r = m \quad y \quad 0 \leq ABS(r) < ABS(n) \quad (1.3)$$

siempre se cumplen. El signo del residuo es el mismo que el del dividendo (o bien el residuo es cero). En consecuencia, la división euleriana de enteros es simétrica con respecto a cero, como lo expresan las siguientes igualdades;

$$(-m)/n = m/(-n) = -(m/n)$$

Algunos ejemplos de esto son:

$31 / 10 = 3$	$31 \text{ REM } 10 = 1$
$-31 / 10 = -3$	$-31 \text{ REM } 10 = -1$
$31 / -10 = -3$	$31 \text{ REM } -10 = 1$
$-31 / -10 = 3$	$-31 \text{ REM } -10 = -1$

En la aritmética de módulos (o de congruencia), el valor m MOD n es en realidad una clase de congruencia, es decir, un conjunto de enteros en vez de un solo número. Este conjunto contiene todos los enteros m-Qn para enteros arbitrarios Q. Con claridad, las clases pueden representarse por medio de un miembro específico; por ejemplo, por su miembro no negativo menor. De aquí que se defina R = m MOD n y al mismo tiempo Q = m DIV n tal que se cumplan las relaciones siguientes:

$$Q \cdot n + R = m \quad y \quad 0 \leq R < n \quad (1.4)$$

Algunos ejemplos de esto son:

31 DIV 10 = 3	31 MOD 10 = 1
-31 DIV 10 = -4	-31 MOD 10 = 9

Observese que la división entre 10^n se puede lograr con sólo recorrer los dígitos decimales n espacios a la derecha e ignorando con esto las cifras perdidas. El mismo método se aplica si los números se representan en forma binaria en vez de decimal. Si se emplea la representación en complemento de 2 (como se hace prácticamente en todas las computadoras modernas), entonces los corrimientos efectúan una división como lo definió el operador DIV anterior (mas no el operador $/$). Por tanto, los compiladores moderadamente complicados representarán una operación de la forma $m \text{ DIV } 2^n$ por medio de una operación de corrimiento rápida, mientras que este atajo no está permitido en expresiones de la forma $m/2^n$.

Si se sabe que una variable (como un contador) no toma valores negativos, entonces este hecho puede expresarse mediante el uso de un tipo más estándar llamado CARDINAL. En una computadora que usa n bits para representar enteros (sin signo), a las variables de este tipo se les puede asignar valores x que cumplan la desigualdad $0 \leq x < 2^n$.

El tipo REAL representa un subconjunto de los números reales. Se supone que la aritmética con operandos de los tipos INTEGER y CARDINAL producen resultados exactos, pero la aritmética sobre valores de tipo REAL puede ser inexacta dentro de los límites de errores de redondeo ocasionados por operaciones de cálculo con un número finito de cifras. Esta es la razón principal de la distinción explícita entre los tipos INTEGER y REAL, como se hace en muchos lenguajes de programación.

Los dos valores del tipo BOOLEAN (booleano) estándar se simbolizan por los identificadores TRUE (verdadero) y FALSE (falso). Los operadores booleanos son la conjunción, disyunción y negación lógicas cuyos valores se definen en la tabla 1.1. La conjunción lógica se denota por el símbolo AND (o bien $\&$, y), la disyunción lógica por OR y la negación lógica por NOT (o bien \sim). Obsérvese que las comparaciones son operaciones que producen un resultado de tipo BOOLEAN. Así pues, el resultado de una comparación puede asignarse a una variable o bien puede usarse como operando de un operador lógico de una expresión booleana. Por ejemplo, dadas las variables booleanas p y q , y las variables enteras $x = 5$, $y = 8$, $z = 10$, las asignaciones

$$\begin{aligned} p &:= x = y \\ q &:= (x < y) \& (y < z) \end{aligned}$$

producen $p = \text{FALSE}$ y $q = \text{TRUE}$.

p	q	$p \text{ AND } q$	$p \text{ OR } q$	$\text{NOT } p$
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Tabla 1.1. Operadores booleanos.

Los operadores booleanos AND y OR tienen una propiedad adicional en Modula-2 (y en algunos lenguajes), la cual los distingue de otros operadores diádicos. En tanto que, por ejemplo, la suma $x + y$ no está definida si x o bien y está indefinida, la conjunción $p \& q$ está definida aun cuando q no lo esté, siempre que p sea FALSE. Esta condicionalidad es una propiedad importante y útil. La definición exacta de AND y OR está dada, pues, por las siguientes ecuaciones:

$$\begin{aligned} p \text{ AND } q &= \text{if } p \text{ then } q \text{ else FALSE} \\ p \text{ OR } q &= \text{if } p \text{ then TRUE else } q \end{aligned} \quad (1.5)$$

El tipo estándar CHAR comprende un conjunto de caracteres imprimibles. Desafortunadamente, no existe un conjunto de caracteres generalmente aceptado que se use en todos los sistemas de computación. En consecuencia, el uso del atributo "estándar" puede en este caso ser casi engañoso; se entenderá en el sentido de "estándar en el sistema de computación en el cual se ejecute cierto programa".

El conjunto de caracteres definido por la Organización Internacional de Normas (ISO) y particularmente su versión estadounidense ASCII (Código Estándar Estadounidense para el Intercambio de Información) es el conjunto más ampliamente aceptado. El conjunto ASCII se tabula por tanto en el apéndice A. Consta de 99 caracteres imprimibles (gráficos) y 33 caracteres de control; los últimos se usan principalmente en la transmisión de datos y para el control del equipo de impresión.

A fin de poder diseñar algoritmos con caracteres (es decir, valores del tipo CHAR) que sean independientes del sistema, debemos poder suponer ciertas propiedades de los conjuntos de caracteres como grupo, a saber.

1. El tipo CHAR contiene las 26 letras mayúsculas latinas, las 26 letras minúsculas, los 10 numerales arábigos y otros caracteres gráficos, como los signos de puntuación.
2. Los subconjuntos de letras y dígitos son ordenados y contiguos, o sea,

$$\begin{aligned} ("A" \leq x) \& (x \leq "Z") &\rightarrow x \text{ es una letra mayúscula} \\ ("a" \leq x) \& (x \leq "z") &\rightarrow x \text{ es una letra minúscula} \\ ("0" \leq x) \& (x \leq "9") &\rightarrow x \text{ es un dígito} \end{aligned} \quad (1.6)$$

3. El tipo CHAR contiene un carácter en blanco que no se imprime y que se puede utilizar como separador.

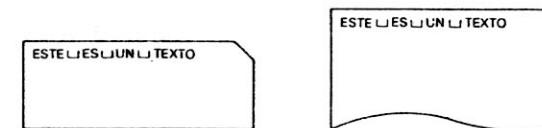


Fig. 1.1. Representación de un texto.

La disponibilidad de dos funciones de transferencia de tipo estándar entre los tipos CHAR y CARDINAL es particularmente importante cuando se quiere escribir programas en forma independiente de la máquina. Los llamaremos ORD(ch), que representa el número ordinal de *ch* en el conjunto de caracteres CHR(i), simbolizando el carácter con número ordinal *i*. De este modo, CHR es la función inversa de ORD y viceversa; es decir,

$$\begin{aligned} \text{ORD(CHR}(i)\text{)} &= i \text{ (si } \text{CHR}(i) \text{ está definido)} \\ \text{CHR(ORD}(c)\text{)} &= c \end{aligned} \quad (1.7)$$

Además, postulamos una función estándar CAP(ch). Su valor se define como la letra mayúscula correspondiente a *ch*, siempre que *ch* sea una letra.

$$\begin{aligned} \text{ch es una letra minúscula} &\rightarrow \text{CAP(ch)} = \text{letra mayúscula correspondiente} \\ \text{ch es una letra mayúscula} &\rightarrow \text{CAP(ch)} = \text{ch} \end{aligned} \quad (1.8)$$

1.5. TIPOS DE SUBINTERVALO

Con frecuencia sucede que una variable toma valores de un cierto tipo únicamente dentro de un intervalo específico. Esto se puede expresar definiendo la variable como de un tipo de subintervalo de acuerdo con la forma

$$\text{TYPE T} = [\text{mín} .. \text{máx}] \quad (1.9)$$

donde *mín* y *máx* son expresiones que especifican los límites del intervalo. Nótese que estas expresiones deben contener constantes sólo como operandos.

EJEMPLOS

```
TYPE año    = [1900 .. 1999]
TYPE letra  = ["A" .. "Z"]
TYPE dígito = ["0" .. "9"]
TYPE oficial = [teniente .. general]
TYPE índice = [0 .. 2*N-1]
```

Dadas las variables

```
VAR y: años
VAR L: letra
```

las asignaciones *y* := 1984 y *L* := "L" son admisibles, pero *y* := 1291 y *L* := "9" no lo son. Sin embargo, la licitud de las asignaciones puede verificarse sin ejecutar el programa solamente si el valor asignado es una constante. La posibilidad de realizar asignaciones del tipo *y* := *i* y *L* := *c*, donde *i* es del tipo INTEGER y *c* es del tipo CHAR, no pueden verificarse a través de un mero examen textual realizado por el compilador. En la práctica, los sistemas que efectúan estas verificaciones durante la ejecución del programa han resultado ser considerablemente valiosos en la elaboración del programa. Su uso de información redundante para detectar posibles errores es una vez más una motivación primordial para utilizar lenguajes de alto nivel.

1.6. ESTRUCTURA DEL ARREGLO

El arreglo es probablemente la estructura de datos mejor conocida; en algunos lenguajes es la única de que se dispone. Un arreglo consta de componentes que son todas del mismo tipo, llamado su *tipo base*; por lo tanto, se denomina estructura *homogénea*. El arreglo es también una estructura así llamada *con acceso al azar*; todos los componentes pueden ser seleccionados al azar y son igualmente accesibles. A fin de representar una componente individual, el nombre de la estructura completa es aumentado por el así llamado *índice* de selección de los componentes. Este índice será el valor del tipo definido como el *tipo de índice* del arreglo. La definición de un tipo de arreglo T especifica, por tanto el tipo T0 y un tipo de índice TI.

(1.10)

EJEMPLOS

```
TYPE Renglón = ARRAY [1 .. 5] OF REAL
TYPE Tarjeta = ARRAY [1 .. 80] OF CHAR
TYPE alfa = ARRAY [0 .. 15] OF CHAR
```

Un cierto valor de una variable

VAR x: Renglón

con todas las componentes que cumplen la ecuación $x_i = 2^{-i}$, pueden visualizarse como se muestra en la figura 1.2.

Un componente individual de un arreglo puede ser seleccionado por un índice. Dada una variable de arreglo x, representamos un selector de arreglo por medio del nombre de dicho arreglo seguido del índice de la componente respectivo i y se escribe x_i o bien $x[i]$. Debido a la primera notación convencional, una componente de una componente de arreglo se llama, pues, *variable suscrita*.

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

Fig. 1.2. Arreglo del tipo renglón.

La manera común de operar con arreglos, particularmente con elementos grandes, consiste en actualizar selectivamente componentes simples en vez de construir valores estructurados completamente nuevos. Esto se expresa tomando en consideración *una variable de arreglo como un arreglo de variables de componente* y permitiendo asignaciones a componentes seleccionadas, como por ejemplo $x[i] := 0.125$. Aunque la actualización selectiva ocasiona solamente que cambie un solo valor del componente, desde un punto de vista conceptual debemos considerar el valor compuesto en su totalidad como si también hubiera cambiado.

El hecho de que los índices de arreglos, es decir, los nombres de componentes del arreglo, deban ser de un tipo de datos definidos (escalar) tiene una consecuencia fundamentalmente importante: *los índices pueden ser calculados*. Una expresión de índice general puede ser sustituida en lugar de una constante de índice; esta expresión será evaluada y el resultado identifica la componente seleccionada. Esta generalidad no sólo ofrece un recurso de programación más significativo y poderoso, sino que al mismo tiempo también origina uno de los errores de programación que se encuentran con mayor frecuencia: el valor resultante puede estar fuera del intervalo especificado como el ámbito de los índices del arreglo. Supondremos que los sistemas de computación decentes dan un aviso en el caso de que se realice dicho acceso equivocado a una componente del arreglo no existente.

La cardinalidad de un tipo estructurado, es decir, el número de valores que pertenecen a este tipo, es el producto de la cardinalidad de sus componentes. Ya que todas las componentes de un tipo de arreglo T son del mismo tipo base T0, se obtiene, dado el tipo de índice TI

card(T) = card(T0)card(TI) (1.11)

Los constituyentes de los tipos de arreglo se pueden estructurar por sí solos. Cualquier variable de arreglo cuyas componentes sean otra vez arreglos se llama *matriz*. Por ejemplo, se tiene

M: ARRAY [1 .. 10] OF Renglón

es un arreglo que consta de diez componentes (renglones), cada una de las cuales consta a su vez de cinco componentes de tipo REAL y se llama matriz 10×5 con componentes reales. Los selectores pueden eslabonarse siguiendo este modelo, tal que M_{ij} y $M[i][j]$ representen la j-ésima componente del renglón M_i , que es la i-ésima componente de M. Esto se abrevia generalmente como $M[i, j]$ y con el mismo espíritu la declaración

M: ARRAY [1 .. 10] OF ARRAY [1 .. 5] OF REAL

puede escribirse más concisamente como

M: ARRAY [1 .. 10], [1 .. 5] OF REAL.

Si una cierta operación tiene que ser efectuada con todas las componentes de un arreglo o con componentes adyacentes de una sección del arreglo, entonces este hecho puede destacarse convenientemente mediante el uso de la proposición FOR, como se muestra en los siguientes ejemplos para calcular la suma y para determinar el elemento máximo de un arreglo declarado como

```

VAR a: ARRAY [0 .. N-1] OF INTEGER
sum := 0;
FOR i := 0 TO N-1 DO sum := a[i] + sum END

k := 0; max := a[0];
FOR i := 1 TO N-1 DO
  IF max < a[i] THEN k := i; max := a[k] END
END.

```

En un ejemplo adicional, supóngase que una fracción f se representa en su forma decimal con $k-1$ dígitos, es decir, por medio de un arreglo d tal que

$$f = \text{Si: } 1 \leq i < k: d_i * 10^{-i}.$$

Ahora supóngase que se desea dividir f entre 2. Esto se hace repitiendo la bien conocida operación de división para los $k-1$ dígitos d_i , comenzando con $i=1$. Consiste en dividir cada dígito entre 2 tomando en cuenta un posible corrimiento de la posición anterior y en conservar un posible residuo r para la siguiente posición:

$$r := 10 * r + d[i]; d[i] := r \text{ DIV } 2; r := r \text{ MOD } 2$$

Este procedimiento se aplica en el programa 1.1 para calcular una tabla de potencias negativas de 2. La repetición de la división entre dos para calcular $2^{-1}, 2^{-2}, \dots, 2^{-N}$ se vuelve a expresar adecuadamente por medio de una instrucción FOR, con lo cual se llega a una inserción o anidamiento de dos instrucciones FOR.

```

MODULE Power;
(*calcular la representacion decimal de potencias negativas de 2*)
FROM InOut IMPORT Write, WriteLn;
CONST N = 10;
VAR i, k, r: CARDINAL;
d: ARRAY [1 .. N] OF CARDINAL;
BEGIN
  FOR k := 1 TO N DO
    Write(".");
    FOR i := 1 TO k-1 DO
      r := 10 * r + d[i]; d[i] := r DIV 2; r := r MOD 2;
      Write(CHR(d[i] + ORD("0")));
    END;
    d[k] := 5; Write("5");
  END;
END Power.

```

Programa 1.1. Cálculo de potencias de 2.

La salida resultante para $n = 10$ es

```

.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625

```

1.7. ESTRUCTURA DEL REGISTRO

El método más general para obtener tipos estructurados consiste en asociar elementos de tipos arbitrarios, que sean posiblemente tipos estructurados por sí solos, en uno compuesto. Ejemplos de matemáticas son los números complejos, compuestos por dos números reales, y coordenadas de puntos, compuestas por dos o más números de acuerdo con la dimensionalidad del espacio ocupado por el sistema de coordenadas. Un ejemplo de procesamiento de datos es la descripción de personas por medio de unas cuantas características relevantes, como su nombre y apellido, fecha de nacimiento, sexo y estado civil.

En matemáticas a dicho tipo compuesto se le denomina *producto Cartesiano* de sus tipos constituyentes. Esto surge del hecho de que el conjunto de valores definido por este tipo compuesto consta de todas las posibles combinaciones de valores, tomadas de cada conjunto definido por cada tipo constituyente. Así, el número de dichas combinaciones, también llamadas *n-adas*, es el producto del número de elementos de cada conjunto constituyente, es decir, la cardinalidad del tipo compuesto es el producto de las cardinalidades de los tipos constituyentes.

En el procesamiento de datos, los tipos compuestos, como las descripciones de personas u objetos, generalmente se presentan en archivos o bancos de datos y registran las características relevantes de una persona u objeto. La palabra *registro* se ha vuelto por tanto ampliamente aceptada para describir un compuesto de datos de esta naturaleza y adoptamos esta nomenclatura en lugar del término *producto Cartesiano*. En términos generales, un tipo de registro T con componentes de los tipos T_1, T_2, \dots, T_n se define como sigue:

```
TYPE T = RECORD s1 : T1;
               s2 : T2;
               ...
               sn : Tn
           END
card(T) = card(T1) * card(T2) * ... * card(Tn)
```

(1.12)

Ejemplos:

```
TYPE Complejo = RECORD re: REAL;
                  im: REAL
              END
TYPE Fecha = RECORD dia: [1 .. 31];
                 mes: [1 .. 12];
                 año: [1 .. 2000]
```

```
END
TYPE Persona = RECORD nombre, primernombre: alfa;
                fechadenacimiento: Fecha;
                sexo: (masculino, femenino);
                estadocivil: (soltero, casado, viudo, divorciado)
            END
```

Podemos visualizar ciertos valores estructurados de registro (por ejemplo) de las variables

z: Complejo
d: Fecha
p: Persona

como se muestra en la figura 1.3.

Los identificadores s_1, s_2, \dots, s_n , introducidos por la definición de un tipo de registro, son los nombres dados a las componentes individuales de variables de ese tipo. Así como a las componentes de registros se les llama *campos*, también los nombres se denominan *identificadores de campos*. Se utilizan en selectores de registros aplicados a variables estructuradas de registros. Dada una variable $x: T$, su i -ésimo campo se simboliza por $x.s_i$. La actualización selectiva de x se logra utilizando la misma denotación selectora en el lado izquierdo de una instrucción de asignación:

$x.s_i := e$

donde e es un valor (expresión) de tipo T_i . Dadas, por ejemplo, las variables de registro z , d y p que se declararon antes, los siguientes son selectores de componentes:

z.im	(de tipo REAL)
d.mes	(de tipo [1 .. 12])
p.nombre	(de tipo alfa)
p.fechadenacimiento	(de tipo Fecha)
p.fechadenacimiento.dia	(de tipo [1 .. 31])

Complejo z	Fecha d	Persona p
1.0	1	WIRTH
– 1.0	4	CHRIS
	1973	18 1 1966
		masculino
		soltero

Fig. 1.3. Registro de los tipos Complejo, Fecha y Persona.

El ejemplo del tipo *Persona* muestra que un constituyente de un registro puede estructurarse por sí mismo. Así, los selectores pueden eslabonarse. Naturalmente, pueden utilizarse también diferentes tipos de estructuración en forma anidada. Por ejemplo, la *i*-ésima componente de un arreglo *a* que es una componente de una variable de registro *r* se simboliza por *r.a[i]* y la componente con el nombre selector *s* de la *i*-ésima componente estructurada del registro *a* se representa por *a[i].s*.

Es característica del producto Cartesiano que contenga todas las combinaciones de elementos de los tipos constituyentes. Pero debe observarse que en aplicaciones prácticas no todas pueden ser significativas. Por ejemplo, el tipo *Fecha* como antes se definió incluye el 31 de abril así como el 29 de febrero de 1985, fechas que nunca ocurrieron. De este modo, la definición de este tipo no refleja la situación real en forma correcta por completo; pero se apegó lo suficiente con fines prácticos y es responsabilidad del programador asegurar que nunca se presenten valores sin sentido durante la ejecución de un programa.

El siguiente extracto breve de un programa muestra el uso de variables de registro. Su objetivo es el de contar el número de personas representadas por la variable de arreglo *familia* que son del sexo femenino y solteras:

```

VAR cuenta: CARDINAL;
familia: ARRAY [1 .. N] OF Persona;
cuenta := 0;

FOR i := 1 TO N DO
  IF (familia[i].sexo = femenino) & (familia[i].estadocivil = soltero) THEN
    cuenta := cuenta + 1
  END
END

```

Una variante notacional del enunciado anterior usa una construcción llamada *with-proposición*:

```

FOR i := 1 TO N DO
  WITH familia[i] DO
    IF (sexo = femenino) & (estadocivil = soltero) THEN
      cuenta := cuenta + 1
    END
  END
END

```

El significado de “WITH *r* DO *s* END” es que los identificadores de campo del tipo de la variable *r* se pueden utilizar sin prefijo dentro de la instrucción *s* y que se toman para hacer referencia a la variable *r*. La construcción *with*-proposición sirve de este modo para abreviar el texto del programa, así como para prevenir la frecuente reevaluación de la dirección de almacenamiento de la componente indizada *familia[i]*.

Las estructuras de registro y arreglo tienen la propiedad común de que ambas son estructuras con *acceso al azar*. El registro es más general en el sentido de que no es requisito que todos los tipos constituyentes deban ser idénticos. A su vez, el arreglo ofrece mayor flexibilidad al permitir que sus selectores de componentes sean valores (expresiones) calculables, en tanto que los selectores de componentes de registro son identificadores de campo declarados en la definición del tipo de registro.

1.8. VARIANTES DE ESTRUCTURA DE REGISTRO

En la práctica, con frecuencia conviene, y es natural, considerar dos tipos simplemente como variantes del mismo tipo. Por ejemplo, un tipo *Coordenada* de la sección anterior puede considerarse como la unión de sus dos variantes de coordenadas cartesianas y polares cuyos constituyentes son (a) dos longitudes y (b) una longitud y un ángulo, respectivamente. A fin de identificar la variante realmente supuesta por una variable, se introduce una tercera componente, el *tipo discriminador* o *campo objetivo*.

```
TYPE CoordMode = (Cartesian, polar);
TYPE Coordinate =
  RECORD
    CASE kind: CoordMode OF
      Cartesian: x, y: REAL |
      polar: r: REAL; phi: REAL
    END
  END
```

Aquí, el nombre del campo objetivo es *kind* (*clase*) y los nombres de las coordenadas son *x* y *y* en el caso de un valor cartesiano o bien, *r* y *phi* si se trata de un valor polar. El conjunto de valores representado por este tipo *Coordenada* es la unión de los tipos

$$\begin{aligned} T_1 &= (x, y: \text{REAL}) \\ T_2 &= (r: \text{REAL}; \text{phi: REAL}) \end{aligned}$$

y su cardinalidad es la suma de las cardinalidades de T_1 y T_2 .

$$\text{card}(T) = \text{card}(T_1) + \text{card}(T_2) \quad (1.13)$$

Sin embargo, con relativa frecuencia las variantes no son tipos completamente distintos, sino dos tipos con componentes parcialmente idénticas. Es esta situación predominante la que dio origen al término estructura de registro *variante*. Un ejemplo de esto lo constituye el tipo *Persona*, que se definió en la sección anterior y en el cual las características relevantes que se registrarán en un archivo dependen del sexo de la persona. Por ejemplo, en relación con una persona del sexo masculino, su peso y el hecho de si tiene barba o no pueden considerarse relevantes en una situación específica; sin embargo, en relación con una persona del sexo femenino, las tres medidas características pueden considerarse como datos relevantes. La siguiente es una definición de tipo que refleja estas circunstancias:

```
TYPE Persona =
  RECORD nombre, primernombre: alfa;
    fechadenacimiento: Fecha;
    estado civil: (soltero, casado, viudo, divorciado);
    CASE s: sexo OF
```

```
    masculino: peso: REAL; barbado: BOOLEAN
    femenino: medidas ARRAY [1 .. 3] OF INTEGER
  END
  END
```

La forma general de una definición de tipo de registro variante con m variantes es

```
TYPE T =
  RECORD s1: T1; s2: T2; ... ; sn-1: Tn-1;
    CASE sn: Tn OF
      v1: s11: T11; ... ; s1,n1: T1,n1 |
      v2: s21: T21; ... ; s2,n2: T2,n2 |
      ...
      vm: sm,1: Tm,1; ... ; sm,nm: Tm,nm
    END
  END
```

Los elementos s_i (para $i = 1 \dots n-1$) son los identificadores de campo que pertenecen a la *parte común* del registro, s_{ij} son los nombres del selector de los campos que pertenecen a la *parte variante* y s_n es el nombre del *campo objetivo* discriminante con tipo T_n . Las constantes v_1, v_2, \dots, v_m denotan los valores del tipo del campo objetivo (escalar). Cada variante i tiene n_i campos en su parte variante. Una variable x de tipo T consta de los componentes

$$x.s_1, x.s_2, \dots, x.s_n, x.s_{k,1}, \dots, x.s_{k,n_k}$$

si y sólo si el valor (actual) de $x.s_n = v_k$. Por lo tanto, el uso de un selector de componentes $x.s_{k,h}$ ($1 \leq h \leq n_k$) cuando $x.s_n \neq v_k$ debe considerarse como un grave error de programación y (en referencia al tipo *Persona* definido antes) es igual a preguntar si una mujer tiene o no barba o bien (en el caso de actualización selectiva) ordenarle que la tenga. Cuando se utilizan registros variantes, se requiere por tanto sumo cuidado y las operaciones correspondientes con las variantes individuales se agrupan mejor en una proposición selectiva, la así llamada *proposición case*, cuya estructura refleja fielmente la de la definición del tipo de registro variante.

```
CASE x.sn OF
  v1: S1;
  v2: S2;
  ...
  vm: Sm
END
```

S_k representa la provisión de proposiciones en el caso en que x toma la forma de la variante k ; es decir, se selecciona para su ejecución si y sólo si el campo objetivo $x.s_n$ tiene el valor v_k . Como consecuencia, es relativamente fácil salvaguardarse en contra del mal uso de los nombres selectores verificando que cada S_k contenga sólo los selectores

$x.s_1 \dots x.s_{n-1}$ and $x.s_{k,1} \dots x.s_{k,n_k}$

El siguiente programa corto de ejemplo ilustra el uso de la proposición case. Su objetivo es el de calcular la distancia d entre los puntos A y B dadas las variables a y b del tipo de registro variante *Coordenada*. El procedimiento de cálculo difiere de acuerdo con las cuatro posibles combinaciones de coordenadas cartesianas y polares (véase la figura 1.4).

```
CASE a.kind OF
    Cartesian: CASE b.kind OF
        Cartesian: d := sqrt(sqrt(a.x-b.x) + sqrt(a.y-b.y)) |
        Polar:      d := sqrt(sqrt(a.x - b.r*cos(b.phi)) + sqrt(a.y - b.r*sin(b.phi)))
    END |
    Polar:      CASE b.kind OF
        Cartesian: d := sqrt(sqrt(a.r*cos(a.phi) - b.x) + sqrt(a.r*sin(a.phi) - b.y)) |
        Polar:      d := sqrt(sqrt(a.r + sqrt(b.r)) - 2*a.r*cos(a.phi-b.phi)))
    END
END
```

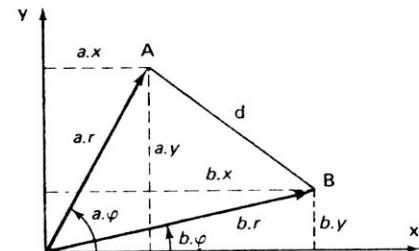


Fig. 1.4. Coordenadas cartesianas y polares.

1.9. ESTRUCTURA DEL CONJUNTO

La tercera estructura fundamental de datos es la del conjunto. Se define por medio del siguiente modelo de declaración:

TYPE T = SET OF T₀ (1.16)

Los posibles valores de una variable x de tipo T son conjuntos de elementos de un conjunto T₀ llamado *conjunto de potencias* de T₀. El tipo T comprende así el conjunto de potencias de su tipo base T₀.

Ejemplos:

TYPE BITSET = SET OF [0 .. 15]
TYPE TapeStatus = SET OF exception (véase sección 1.3)

Dadas las variables

b: BITSET
t: ARRAY [1 .. 6] OF TapeStatus

determinados valores estructurados en conjunto pueden ser construidos y asignados, por ejemplo, de la manera siguiente:

b := {2, 3, 5, 7, 11, 13}
t[3] := TapeStatus{manual}
t[5] := TapeStatus{}
t[6] := TapeStatus{descargado .. torcida}

Aquí, el valor asignado a t₃ es el *conjunto único* que consta del elemento único *manual*; a t₅ se asigna el conjunto vacío, lo que significa que la quinta unidad de cinta se devuelve a su condición operacional (no excepcional), en tanto que a t₆ se asigna el conjunto de estas tres excepciones.

La cardinalidad de un tipo de conjunto T es

card(T) = 2^{card(T₀)} (1.17)

Esto puede deducirse fácilmente del hecho de que cada uno de los elementos card(T₀) de T₀ debe simbolizarse por uno de los dos valores *presente* o bien *ausente* y que todos los elementos sean independientes el uno del otro. Es evidentemente esencial para una implantación efectiva y económica que el tipo base no sólo sea finito, sino que su cardinalidad sea razonablemente pequeña.

Los siguientes operadores elementales están definidos en todos los tipos de conjunto:

- * intersección de conjuntos
- + unión de conjuntos

- diferencia de conjuntos
- IN membresía de conjuntos

A la construcción de la *intersección* o la *unión* de dos conjuntos se les llama *multiplicación de conjuntos* o bien *adición de conjuntos*, respectivamente. Las prioridades de los operadores de conjuntos se definen de acuerdo con lo estipulado, con el operador de intersección teniendo prioridad sobre los operadores de unión y diferencia, que a su vez tienen prioridad sobre el operador de membresía, el cual se clasifica como operador relacional. Los siguientes son ejemplos de expresiones de conjuntos y sus equivalentes con paréntesis:

$$\begin{aligned} r * s + t &= (r * s) + t \\ r - s * t &= r - (s * t) \\ r - s + t &= (r - s) + t \\ x \text{ IN } s + t &= x \text{ IN } (s + t) \end{aligned}$$

1.10. REPRESENTACION DE ESTRUCTURAS DE ARREGLO, REGISTRO Y CONJUNTO

La esencia del uso de abstracciones en la programación es que un programa puede ser concebido, entendido y verificado sobre la base de las leyes que rigen las abstracciones, y que no se necesita conocer más acerca de las formas en que las abstracciones se instrumentan y representan en una cierta computadora. No obstante, es esencial que un programador profesional logre un entendimiento de las técnicas ampliamente utilizadas para representar los conceptos básicos de las abstracciones de programación, como las estructuras de datos fundamentales. Es de utilidad en cuanto a que podría facultar al programador para tomar decisiones sensatas acerca del diseño del programa y datos, a la luz no sólo de las propiedades abstractas de las estructuras, sino también de sus instrumentaciones en computadoras reales, tomando en cuenta las facultades y limitaciones específicas de la computadora.

El problema de la representación de datos consiste en el mapeo de la estructura abstracta almacenada en la memoria de una computadora. Los almacenamientos en la computadora son (en una primera aproximación) arreglos de celdas individuales de almacenamiento llamadas *palabras*. Los índices de las palabras se llaman *direcciones*.

VAR almacenamiento: ARRAY ADDRESS OF WORD

Las cardinalidades de los tipos ADDRESS (dirección) y WORD (palabra) varían de una computadora a otra. Un problema determinado es la considerable variabilidad de la cardinalidad de la palabra. Su logaritmo se llama *tamaño de palabra*, ya que es el número de bits del cual consta una celda de memoria.

1.10.1. Representación de arreglos

La representación de la estructura de un arreglo es un mapeo del arreglo (abstracto) con componentes de tipo T en el almacenamiento que es un arreglo con componentes de tipo WORD (palabra). El arreglo debe trazarse en tal forma que el cálculo de la dirección de las componentes del arreglo sea lo más simple (y por tanto eficiente) posible. La dirección i del j -ésimo componente del arreglo se calcula por medio de la función de mapeo lineal

$$i = i_0 + j * s \quad (1.18)$$

donde i_0 es la dirección del primer componente y s es el número de palabras que ocupa un componente. Suponiendo que la palabra es la unidad individualmente accesible más pequeña, evidentemente se desea que s sea un número entero; el caso más simple es $s = 1$. Si s no es un número entero (y éste es el caso normal), entonces s por lo general se redondea al siguiente entero mayor $|s|$. Cada componente del arreglo ocupa por tanto $|s|$ palabras con lo cual $|s| - s$ palabras quedan sin usarse (véanse las figuras 1.5 y 1.6). El redondeo del número de palabras que se necesitan para el siguiente número entero se denomina *relleno*. El factor de utilización de la memoria u es el cociente de las cantidades

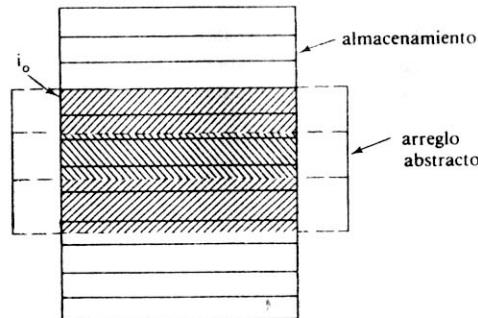


Fig. 1.5. Mapeo de un arreglo en la memoria.

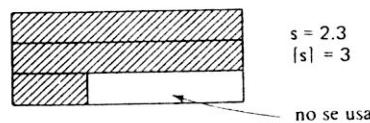


Fig. 1.6. Representación rellenada de un registro.

mínimas de espacio en almacenamiento que se necesitan para representar una estructura y de la cantidad que realmente se usa:

$$u = s / \lceil s \rceil \quad (1.19)$$

Como un implantador tiene que procurar una utilización de almacenamiento lo más próxima a 1 que sea posible, y ya que el acceso de partes de palabras es un proceso endoso y relativamente ineficaz, el citado implantador debe comprometerse. Las siguientes consideraciones son relevantes:

1. El relleno reduce la utilización del almacenamiento.
2. La omisión del relleno puede necesitar un acceso parcial de palabras ineficiente.
3. El acceso parcial de palabras puede ocasionar que el código (programa compilado) se extienda y por tanto neutralice la ganancia obtenida por la omisión del relleno.

De hecho, las consideraciones 2 y 3 generalmente son tan dominantes que los compiladores siempre emplean el relleno en forma automática. Se observa que el factor de utilización es siempre $u > 0.5$, si $s > 0.5$. Sin embargo, si $s \leq 0.5$, el factor de utilización puede verse significativamente reducido por la asignación de una componente de arreglo más a cada palabra. Esta técnica se llama *empaqueamiento*. Si se empaquetan n componentes en una palabra, el factor de utilización es (véase la figura 1.7)

$$u = n * s / \lceil n * s \rceil \quad (1.20)$$

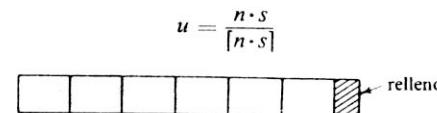


Fig. 1.7. Empacado de seis componentes en una palabra.

El acceso a la i -ésima componente de una arreglo empaquetado comprende el cálculo de la dirección de palabra j en la cual se ubica la componente deseada; además implica el cálculo de la posición del componente k respectivo dentro de la palabra.

$$j = i \text{ DIV } n \quad k = i \text{ MOD } n$$

En muchos lenguajes de programación, el programador no tiene control sobre la representación de las estructuras de datos abstractas. Sin embargo, debe ser posible indicar la deseabilidad de empaquetado por lo menos en aquellos casos en los cuales más de una componente se ajustaría a una sola palabra, es decir, cuando pudiera lograrse una economía de almacenamiento en un factor de 2 y más. Se propone la convención de indicar la deseabilidad del empaquetado anteponiendo la palabra PACKED en la declaración ARRAY, arreglo (o RECORD, registro).

1.10.2. Representación de estructuras de registro

Los registros se mapean en la memoria de una computadora por la simple yuxtaposición de sus componentes. La dirección de un componente (campo) r_i relativa a la dirección de origen del registro r se llama *compensación* k_i del campo. Esta se calcula como

$$k_i = s_1 + s_2 + \dots + s_{i-1} \quad (1.21)$$

donde s_j es el tamaño (en palabras) del j -ésimo componente. Ahora comprendemos que el hecho de que todos los componentes de un arreglo sean del mismo tipo tiene la bienvenida consecuencia de que $k_i = (i-1)*s$. La generalidad de la estructura del registro por desgracia no admite dicha función lineal simple para calcular la compensación de la dirección y ésta es por tanto la razón real del requisito de que los componentes del registro sean seleccionables sólo por identificadores fijos. Esta restricción tiene el beneficio deseable de que las compensaciones respectivas se conocen en el momento de la compilación. La mayor eficiencia resultante del acceso del campo de registro es bien conocida.

La técnica de empaquetado puede ser ventajosa si varias componentes del registro pueden ajustarse en una sola palabra de almacenamiento (véase la figura 1.8). Ya que las compensaciones pueden calcularse por medio del compilador, la compensación de un campo contenido en una palabra puede ser asimismo determinada por el compilador. Esto significa que en muchas computadoras el empaquetamiento de registros ocasiona un deterioro en la efectividad del acceso considerablemente menor que el producido por el empaquetamiento de arreglos.

1.10.3. Representación de conjuntos

Un conjunto s se representa adecuadamente en la memoria de una computadora por medio de su *función característica* $C(s)$. Este es un arreglo de valores lógicos cuyo i -ésimo

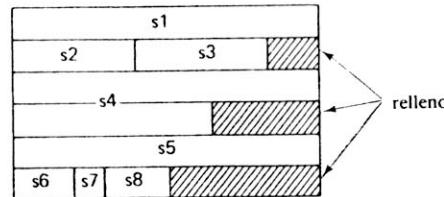


Fig. 1.8. Representación de un registro empacado.

componente tiene el significado *i* está presente en *s*. El tamaño del arreglo está determinado por la cardinalidad del tipo del conjunto.

$$C(s_i) = (i \text{ IN } s) \quad (1.22)$$

Para poner un ejemplo, el conjunto de enteros pequeños $s = 2, 3, 5, 7, 11, 13$ está representado por la secuencia de valores lógicos F(alse), falso y T(rue), verdadero

$$C(s) = (\text{FFTTFTFTFFFFFTFTFF})$$

si el tipo base de *s* es el subintervalo de enteros 0 .. 15. En la memoria de una computadora la secuencia de valores lógicos se representa como una *cadena de bits*, es decir, una secuencia de ceros y unos.

La representación de conjuntos por su función característica tiene la ventaja de que las operaciones de cálculo de la unión, intersección y diferencia de dos conjuntos pueden realizarse como operaciones lógicas elementales. Las siguientes equivalencias, que se cumplen para todos los elementos *i* del tipo base de los conjuntos *x* y *y*, relacionan las operaciones lógicas con operaciones sobre conjuntos:

$$\begin{aligned} i \text{ IN } (x+y) &= (i \text{ IN } x) \text{ OR } (i \text{ IN } y) \\ i \text{ IN } (x*y) &= (i \text{ IN } x) \text{ AND } (i \text{ IN } y) \\ i \text{ IN } (x-y) &= (i \text{ IN } x) \text{ AND NOT } (i \text{ IN } y) \end{aligned} \quad (1.23)$$

Estas operaciones lógicas están a disposición del usuario en todas las computadoras digitales y además, operan concurrentemente en todos los elementos correspondientes (bits) de una palabra. Por lo tanto, parece que para poder realizar las operaciones básicas de conjunto en forma eficaz, los conjuntos deben representarse en un número pequeño fijo de palabras, con lo cual no sólo se disponga de las operaciones lógicas básicas, sino también de las de corrimiento. La verificación de la membresía se instrumenta por tanto por medio de un sólo cambio y una operación de prueba de bits (signo) subsecuente. Como consecuencia, una prueba de la forma $x \text{ IN } \{c_1, c_2, \dots, c_n\}$ puede realizarse con considerable mayor eficacia que la expresión booleana equivalente

$$(x = c_1) \text{ OR } (x = c_2) \text{ OR } \dots \text{ OR } (x = c_n)$$

Un corolario es que la estructura de conjunto debe usarse solamente en el caso de tipos base pequeños. El límite de la cardinalidad de tipos base para los cuales puede garantizarse una implantación considerablemente efectiva se determina por la longitud de pa-

labra de la computadora subyacente; y está claro por qué se prefieren las computadoras con longitudes de palabras grandes en este aspecto. Si el tamaño de la palabra es relativamente pequeño, puede optarse por una representación que utilice palabras múltiples para un conjunto.

1.11. ESTRUCTURA DE LA SECUENCIA

El cuarto método elemental de estructuración es la *secuencia*. Un tipo de secuencia podría enunciarse como

$$\text{TYPE T} = \text{SEQUENCE OF } T_0 \quad (1.24)$$

Esta forma aclara que todos los elementos de la secuencia son del mismo tipo, llamado *tipo base* T_0 de la secuencia. Una secuencia s la representaremos con n elementos por medio de

$$s = \langle s_0, s_1, s_2, \dots, s_{n-1} \rangle$$

a n se le denomina *longitud* de la secuencia. Esta estructura parece ser muy semejante al arreglo. La diferencia esencial es que, en el caso del arreglo, el número de elementos es fijado por la declaración del arreglo, en tanto que en la secuencia se deja abierto. Esto implica que puede variar durante la ejecución del programa. Aunque toda secuencia tiene en cualquier instante una longitud específica y finita, debemos considerar la cardinalidad de un tipo de secuencia como infinito, ya que no hay límite fijo para la longitud potencial de las variables de una secuencia.

Una consecuencia directa de la cardinalidad infinita de los tipos de secuencia es la imposibilidad de asignar una cantidad determinada de espacio de almacenamiento a variables de secuencia. En cambio, el almacenamiento tiene que ser asignado durante la ejecución del programa, sobre todo siempre que la secuencia crezca. Quizá la memoria pueda ser reclamada cuando la secuencia se contraiga. En cualquier caso, debe emplearse un esquema de asignación de almacenamiento dinámica. Todas las estructuras con cardinalidad infinita comparten esta propiedad, que es tan esencial que las clasificamos como *estructuras avanzadas* en contraste con las estructuras fundamentales que se han estudiando hasta ahora.

Entonces, ¿cuál es el motivo por el cual estudiamos las secuencias en este capítulo acerca de estructuras fundamentales? La razón principal es que la estrategia de manejo de la memoria es suficientemente simple para las secuencias (en contraste con otras estructuras avanzadas), si damos vigor a una cierta disciplina en el uso de las secuencias. De hecho, en esta condición el manejo del almacenamiento puede delegarse con toda seguridad a un mecanismo que se garantice como razonablemente efectivo. La razón secundaria es que las secuencias son en realidad ubicuas en todas las aplicaciones de las computadoras. Esta estructura prevalece en todos los casos donde se comprenden diferentes clases de medios de almacenamiento, es decir, a donde los datos serán transferidos de un medio a otro, como de disco a cinta o bien de cinta a almacenamiento primario o viceversa.

La disciplina mencionada es la limitación a utilizar solamente *acceso secuencial*. Con esto queremos decir que una secuencia se inspecciona procediendo estrictamente de un elemento a su sucesor inmediato, y que se genera agregando repetidamente un elemento

en su extremo. La consecuencia inmediata es que los elementos no son directamente accesibles, con excepción del elemento que se encuentra regularmente en espera de inspección. Es esta disciplina de acceso la que distingue en forma fundamental las secuencias de los arreglos. Como observaremos en el capítulo 2, la influencia de una disciplina de acceso en los programas es profunda.

La ventaja de apegarse al acceso secuencial que, después de todo, es una restricción grave, es la relativa simplicidad del manejo de la memoria que se requiere. Pero aún más importante es la posibilidad de utilizar técnicas de manejo por buffer efectivas cuando se desplazan datos hacia o desde dispositivos secundarios de almacenamiento. El acceso secuencial nos permite alimentar flujos de datos a través de tubos entre los diferentes medios. El manejo por buffer implica la colección de secciones de un flujo en un buffer y la transportación subsiguiente del contenido total del buffer una vez que éste se ha llenado. Esto produce un uso significativamente más eficaz de la memoria secundaria. Dado solamente un acceso secuencial, el mecanismo de manejo por buffer es razonablemente directo para todas las secuencias y todos los medios. Por lo tanto, puede convertirse con toda seguridad en un sistema de uso general y el programador no tiene que verse agobiado por tener que incorporarlo en su programa. A dicho sistema por lo general se le llama *sistema de archivo*, ya que los dispositivos de acceso secuencial de alto volumen se utilizan para el almacenamiento permanente de datos y los conservan aun cuando la computadora sea desconectada. La unidad de datos en estos medios es la secuencia o *archivo secuencial*.

Existen ciertos medios de almacenamiento en los cuales el acceso secuencial es en realidad el único posible. Entre ellos se encuentran evidentemente todos los tipos de cintas. Pero aun en discos magnéticos cada pista de grabación constituye un elemento de almacenamiento que admite sólo el acceso secuencial. En forma estricta el acceso secuencial es la característica más importante de todo dispositivo mecánicamente móvil y de algunos otros de igual manera.

Resumimos la esencia de lo anterior como sigue:

1. Los arreglos, registros y conjuntos son estructuras con acceso al azar. Se utilizan cuando se alojan en memoria primaria con acceso al azar.
2. Las secuencias se utilizan para acceder a datos en memoria secundaria con acceso secuencial, como discos y cintas.

1.11.1. Operadores elementales de secuencia

La disciplina del acceso secuencial puede reforzarse ofreciendo un conjunto de operadores de secuencia a través de los cuales se puede acceder a las variables de las secuencias *exclusivamente*. En consecuencia, aunque podamos aquí referirnos al i -ésimo elemento de una secuencia s escribiendo s_i , esto no será posible en un programa. Evidentemente, el conjunto de operadores debe contener un operador para generar y uno para inspeccionar una secuencia. Como ya se mencionó, una secuencia se genera agregando elementos en su extremo, y se inspecciona procediendo al siguiente elemento. Por lo tanto, una cierta posición siempre está asociada con toda secuencia. Ahora postularemos y describiremos informalmente el siguiente conjunto de operadores primitivos:

1. Open(s), abrir(s), define a s como la secuencia vacía, o sea, la secuencia de longitud 0.
2. Write(s, x), escribir(s, x), agrega un elemento con valor x a la secuencia s.
3. Reset(s), re establecer(s), coloca la posición actual (indicador de posición) al inicio de s.
4. Read(s, x), leer(s, x), asigna el elemento designado por la posición actual a la variable x y adelanta la posición hacia el siguiente elemento.

Para lograr un entendimiento más preciso de los operadores de secuencia, se da el siguiente ejemplo. Muestra la forma en que podrían expresarse cuando las secuencias se representan en términos de arreglos. Este ejemplo de implantación intencionalmente forja conceptos presentados y analizados con anterioridad, y no involucra memorias secuenciales o de manejo por buffer que, como se mencionó antes, hacen que el concepto de secuencia sea realmente atractivo.

No obstante, este ejemplo exhibe todas las características esenciales de los operadores primitivos de secuencia cuando representan un sistema de archivo secuencial para respaldar almacenamientos.

Primero, los operadores se presentan en términos de procedimientos convencionales. Esta colección de definiciones se llama *módulo de definición*. Ya que el tipo de secuencia figura en las listas de parámetros formales, este tipo también debe declararse. La declaración es un buen ejemplo de una aplicación de una estructura de registro ya que, además del campo que simboliza el arreglo que representa la secuencia, se requieren más campos para denotar la longitud y la posición actuales, es decir, el *estado* de la secuencia. Además, se agrega otro campo que indicará si una operación de lectura ha tenido buenos resultados o bien, si se llenó debido a que no hubo otro elemento que leer.

```
DEFINITION MODULE FileSystem;
  FROM SYSTEM IMPORT WORD; (1.25)
  CONST MaxLength = 4096;
  TYPE Sequence = RECORD pos, length: CARDINAL;
    eof: BOOLEAN;
    a: ARRAY [0 .. MaxLength-1] OF WORD
  END;

  PROCEDURE Open(VAR f: Sequence);
  PROCEDURE WriteWord(VAR f: Sequence; w: WORD);
  PROCEDURE Reset(VAR f: Sequence);
  PROCEDURE ReadWord(VAR f: Sequence; VAR w: WORD);
  PROCEDURE Close(VAR f: Sequence);
END FileSystem.
```

Nótese que en este ejemplo la longitud máxima que las secuencias pueden alcanzar es una constante arbitraria. Si un programa llega a ocasionar que una secuencia se alargue, éste no sería un error del programa, sino una inadecuado de esta implantación. Por el otro lado, una operación de lectura que procede más allá del extremo actual de la secuencia en realidad sería error del programa. A fin de evitarlo, debe ofrecerse un recurso para inspeccionar si se llega al extremo. Este está presente en la forma del campo booleano eof (por end of, fin de). Es el único campo que se supone debe conocer el usuario de esta

implantación de la abstracción del tipo de secuencia. El tipo base se elige aquí como WORD (palabra). Se utilizará para cualquier tipo en general. En el lenguaje Modula-2 es un tipo compatible en parámetros con varios tipos estándar, inclusive INTEGER.

Las proposiciones que constituyen los procedimientos se especifican en el módulo de implantación correspondiente:

```
IMPLEMENTATION MODULE FileSystem;
  FROM SYSTEM IMPORT WORD; (1.26)
  PROCEDURE Open(VAR f: Sequence);
  BEGIN f.length := 0; f.pos := 0; f.eof := FALSE
  END Open;

  PROCEDURE WriteWord(VAR f: Sequence; w: WORD);
  BEGIN
    WITH f DO
      IF pos < MaxLength THEN
        a[pos] := w; pos := pos+1; length := pos
      ELSE HALT
      END
    END
  END WriteWord;

  PROCEDURE Reset(VAR f: Sequence);
  BEGIN f.pos := 0; f.eof := FALSE
  END Reset;

  PROCEDURE ReadWord(VAR f: Sequence; VAR w: WORD);
  BEGIN
    WITH f DO
      IF pos = length THEN f.eof := TRUE
      ELSE w := a[pos]; pos := pos+1
      END
    END
  END ReadWord;

  PROCEDURE Close(VAR f: Sequence);
  BEGIN (*vacío*)
  END Close;
END FileSystem.
```

Dado este conjunto de operadores de secuencias, el siguiente esquema de programa emerge para escribir y después leer un archivo secuencial s:

```
Open(s);
  WHILE B DO P(x); WriteWord(s, x) END (1.27)
  Reset(s); ReadWord(s, x);
  WHILE ~s.eof DO Q(x); ReadWord(s, x) END
```

Aquí, B es la condición que debe cumplirse antes de que la instrucción P sea ejecutada para calcular el valor del siguiente elemento, que se agrega después a la secuencia s.

Cuando se lee, Q es la proposición que se aplica a cada valor leído de la secuencia. Q es protegida por la condición $\sim s.eof$ que garantiza que en realidad se ha leido el siguiente elemento. Se observa una ligera asimetría entre los dos esquemas. La proposición adicional de lectura se debe al hecho de que para leer n elementos deben ejecutarse $n + 1$ operaciones de lectura, de las cuales la última falla. Esta proposición de lectura fallida se necesita, debido a que el operador de lectura es el único que altera el valor de la variable *eof*. Esto podría evitarse postulando que el operador de re establecer (reset) definiría el valor del campo *eof* como *longitud* = 0. El lenguaje Pascal, por ejemplo, define el operador de re establecer en esta forma sensata. Pero muchos sistemas de archivo ampliamente utilizados no lo hacen, y ésta es la razón por la cual nos apegamos a esta estrategia al demostrar el uso de las secuencias.

Con mucha frecuencia, los sistemas de archivo admiten alguna relajación de la disciplina del acceso estrictamente secuencial sin penalización grave. En particular, permiten el posicionamiento de la secuencia en un sitio arbitrario en vez de sólo en el inicio. Esta relajación de las reglas se logra mediante la introducción de un procedimiento adicional de secuencia llamado *SetPos(s, k)* con $0 \leq k < s.longitud$. Evidentemente, *SetPos(s, 0)* es equivalente a *Reset(s)*. La facilidad de implantación de esta extensión se debe a nuestra definición de la operación de escritura, la cual postula que la escritura siempre ocurre en el extremo de la secuencia. Consecuentemente, si una operación de escritura ocurre cuando la posición del archivo no está en el extremo, el nuevo elemento sustituye la cola anterior en su totalidad, es decir, la secuencia se trunca.

1.11.2. Manejo por buffer de secuencias

Cuando los datos son transferidos hacia o desde un dispositivo de almacenamiento secundario, los bits individuales se transfieren como un flujo. Por lo general, un dispositivo impone restricciones estrictas de temporización en la transmisión. Por ejemplo, si los datos se escriben sobre una cinta, la cinta se desplaza a una velocidad fija y requiere que los datos sean alimentados a una razón fija. Cuando la fuente se detiene, el movimiento de la cinta se suspende y la velocidad desciende rápida, pero no instantáneamente. Así, se deja un espacio entre los datos transmitidos y los que siguen en un instante posterior. A fin de lograr una alta densidad de datos, el número de espacio tiene que conservarse pequeño y por tanto, los datos se transmiten en *bloques* relativamente grandes una vez que la cinta se mueve. Condiciones semejantes se cumplen con los discos magnéticos, donde los datos se asignan a pistas con un número fijo de bloques de tamaño específico, el así llamado *tamaño del bloque*. De hecho, un disco debe considerarse como un arreglo de bloques, con cada bloque siendo leído o escrito como un todo; contiene comúnmente 2^k bytes con $k = 8, 9$ o 10 .

Sin embargo, nuestros programas no observan ninguna de estas restricciones de temporización. A fin de permitirles ignorar las restricciones, los datos por transferir son *manejados por buffer*. Se colectan en una así llamada *variable buffer* (en memoria principal) y se transfieren cuando se acumula una cantidad suficiente de datos para formar un bloque del tamaño que se pide.

El manejo por buffer tiene una ventaja adicional al permitir que el proceso que genera (recibe) datos proceda concurrentemente con el dispositivo que escribe (lee) los datos de

(hacia) el buffer. De hecho, conviene considerar al dispositivo como un proceso en sí que meramente copia bloques de datos. La finalidad del buffer consiste en asignar un cierto grado de desacoplamiento entre los dos procesos, a los que llamaremos *productor* y *consumidor*. Si, por ejemplo, el consumidor es lento en un cierto momento, puede encontrarse con el productor más adelante. Este desacoplamiento es a menudo esencial para lograr una buena utilización de dispositivos periféricos, pero sólo tiene efecto si los índices del productor y consumidor son casi iguales en promedio, pero fluctúan en tiempos. El grado de desacoplamiento crece cuando aumenta el tamaño del buffer.

Ahora tornamos nuestra atención a la cuestión de cómo representar un buffer y se pondrá, en el tiempo qué sea, que los elementos de datos se depositan y extraen en forma individual en vez de en bloques. Un buffer esencialmente constituye una cola de primero en entrar, primero en salir (*FIFO, first-in-first-out*). Si se declara como un arreglo dos variables de índice, por ejemplo, *in* (dentro) y *out* (fuera), marcan las posiciones de la siguiente localidad que se escribirá y que será leída. En teoría, dicho arreglo no debe tener límites de índice. Un arreglo finito es muy adecuado, considerando el hecho de que los elementos una vez traídos ya no son relevantes. Su localidad puede bien volver a usarse. Esto nos conduce a la idea del *buffer circular*. Las operaciones de depositar y traer un elemento se expresan en el siguiente módulo, el cual exporta estas operaciones como procedimientos, pero oculta el buffer y sus variables de índice (y con lo cual eficazmente el mecanismo de manejo por buffer) de los procesos del cliente. Este mecanismo implica asimismo una variable *n* que cuenta el número de elementos que hay regularmente en el buffer. Si *N* simboliza el tamaño del buffer, la condición $0 \leq n \leq N$ es una invariante obvia. Por consiguiente, la operación *fetch (traer)* debe ser protegida por la condición $n > 0$ (buffer no vacío) y la operación *deposit (depositar)* por la condición $n < N$ (buffer no lleno). Si no se cumple la primera condición esto debe considerarse como un error de programación; una violación de la segunda debe considerarse como una falla de la implantación sugerida (buffer demasiado pequeño).

```
MODULE Buffer;
  EXPORT deposit, fetch;
  CONST N = 1024; (*tamaño del buffer*)
  VAR n, in, out: CARDINAL;
    buf: ARRAY [0 .. N-1] OF WORD;
  PROCEDURE deposit(x: WORD);
  BEGIN
    IF n = N THEN HALT END;
    n := n+1; buf[in] := x; in := (in + 1) MOD N
  END deposit;
  PROCEDURE fetch(VAR x: WORD);
  BEGIN
    IF n = 0 THEN HALT END;
    n := n-1; x := buf[out]; out := (out + 1) MOD N
  END fetch;
  BEGIN n := 0; in := 0; out := 0
  END Buffer.
```

(1.28)

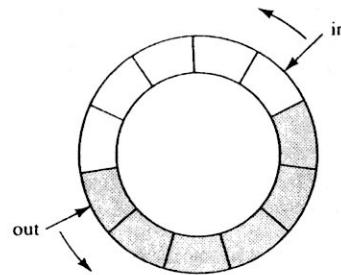


Fig. 1.9. Buffer circular con índices *in* (dentro) y *out* (fuera)

Esta implantación de un buffer es solamente aceptable, si los procedimientos *deposit* y *fetch* son activados por un solo agente (una vez actuando como productor y una como consumidor). Sin embargo, si son activados por procesos concurrentes individuales, este esquema es demasiado simple. La razón es que el intento que se hace por depositar en un buffer lleno o bien el que se hace por extraer (traer) de uno vacío, son bien legítimos. La ejecución de estas acciones meramente tendrá que ser demorada hasta que se establezcan las condiciones de seguridad. Dichas demoras constituyen en esencia la *sincronización* necesaria entre procesos concurrentes. Podemos representar estos retrasos, respectivamente, mediante las proposiciones

```
REPEAT UNTIL n < N
  REPEAT UNTIL n > 0
```

que deben ser sustituidas por las dos proposiciones HALT condicionadas en (1.28). Sin embargo, esta solución no se recomienda, aun si se sabe que los dos procesos son impulsados por dos motores individuales. La razón es que los dos procesadores necesariamente acceden a la misma variable *n* y por tanto, el mismo espacio en memoria. El proceso repetitivo, que constantemente solicita el valor *n*, obstaculiza a su compañero, ya que en ningún momento la memoria puede ser accedida por más de un proceso. Este tipo de *espera larga* debe evitarse en realidad y por tanto postulamos un recurso que hace menos explícitos los detalles de sincronización, de hecho los oculta. A este recurso lo llamaremos *signal (señal)*, y supondremos que está a disposición de un módulo de servicio (denominado *Processes (Procesos)* junto con un conjunto de operadores primitivos de señales.

Toda señal *s* se asocia con un guardia (condición) P_s' . Si un proceso necesita ser retrasado hasta que P_s' se establezca (por medio de algún otro proceso), debe, antes de proceder, esperar a que aparezca la señal *s*. Esto se expresará por la proposición *Wait(s)*, esperar(*s*). Si, por el otro lado, un proceso establece P_s , de aquí en adelante señala este hecho por medio de la proposición *Send(s)*, enviar(*s*). Si P_s es una condición pre establecida para cualquier proposición *Send(s)*, entonces P_s puede considerarse como una postcondición de *Wait(s)*.

```
DEFINITION MODULE Processes;
  TYPE Signal;
```

(1.29)

```
PROCEDURE Wait(VAR s: Signal);
PROCEDURE Send(VAR s: Signal);
PROCEDURE Init(VAR s: Signal);
END Processes.
```

Ahora podemos expresar el módulo buffer (1.28) en una forma que funcione adecuadamente cuando sea utilizado por procesos concurrentes individuales:

```
MODULE Buffer;
  IMPORT Signal, Wait, Send, Init;
  EXPORT deposit, fetch;
  CONST N = 1024; (*tamaño del buffer*)
  VAR n, in, out: CARDINAL;
    nonfull: Signal; (*n < N*)
    nonempty: Signal; (*n > 0*)
    buf: ARRAY [0 .. N-1] OF WORD;
  PROCEDURE deposit(x: WORD);
  BEGIN
    IF n = N THEN Wait(nonfull) END ;
    n := n + 1; buf[in] := x; in := (in + 1) MOD N;
    IF n = 1 THEN Send(nonempty) END
  END deposit;
  PROCEDURE fetch(VAR x: WORD);
  BEGIN
    IF n = 0 THEN Wait(nonempty) END ;
    n := n - 1; x := buf[out]; out := (out + 1) MOD N;
    IF n = N-1 THEN Send(nonfull) END
  END fetch;
  BEGIN n := 0; in := 0; out := 0; Init(nonfull); Init(nonempty)
  END Buffer.
```

(1.30)

Sin embargo, debe hacerse una advertencia adicional. El esquema falla miserablemente, si por coincidencia el consumidor y el productor buscan y traen el valor del contador *n* simultáneamente para su actualización. En forma impredecible, su valor resultante será *n+1* o bien *n-1*, pero no *n*. En realidad se necesita proteger los procesos de una interferencia peligrosa. En general, todas las operaciones que *alteran los valores de variables compartidas* constituyen peligros potenciales.

Una condición suficiente (pero no siempre necesaria) es que todas las variables compartidas sean declarados locales a un módulo cuyos procedimientos garanticen ser ejecutados en exclusión mutua. A dicho módulo se le llama *monitor* [1-7]. La provisión de exclusión mutua garantiza que en cualquier momento cuando mucho un proceso se compromete activamente en la ejecución de un procedimiento del monitor. Si algún otro proceso llega a solicitar un procedimiento del (mismo) monitor, automáticamente será demorado hasta que el primer proceso haya terminado su procedimiento.

Nota: Por activamente comprometido nos referimos a que un proceso ejecuta una proposición diferente que la proposición wait (de espera).

Por último volvemos a considerar el problema donde el productor o el consumidor (o bien ambos) necesitan disponer de los datos en un cierto tamaño de bloque. El siguiente módulo es una variante de (1.30), suponiendo un tamaño de bloque de N_p elementos de datos para el productor y de N_c elementos para el consumidor. En estos casos, el tamaño del bloque N se elige por lo general como un múltiplo común de N_p y N_c . A fin de destacar que la simetría existente entre las operaciones de traer (fetch) y depositar (deposit) datos, el contador individual n se representa ahora por dos contadores, a saber ne y nf . Estos especifican los números de ranuras del buffer vacías y llenas, respectivamente. Cuando el consumidor está inactivo, nf indica el número de elementos que se necesitan para que proceda el consumidor; y cuando el productor espera, ne especifica el número de elementos que se necesitan para que el productor reanude su labor. (Por lo tanto, $ne + nf = N$ no siempre es válido.)

```

MODULE Buffer;
  IMPORT Signal, Wait, Send, Init;
  EXPORT deposit, fetch;

  CONST Np = 16; (*tamaño del bloque del productor*)
    Nc = 128; (*tamaño del bloque del consumidor*)
    N = 1024; (*tamaño del buffer, múltiplo común de Np y Nc*)

  VAR ne, nf: INTEGER;
    in, out: CARDINAL;
    nonfull: Signal; (*ne >= 0*)
    nonempty: Signal; (*nf >= 0*)
    buf: ARRAY [0 .. N-1] OF WORD;

  PROCEDURE deposit(VAR x: ARRAY OF WORD);
  BEGIN ne := ne - Np;
    IF ne < 0 THEN Wait(nonfull) END ;
    FOR i := 0 TO Np-1 DO buf[in] := x[i]; in := in + 1 END ;
    IF in = N THEN in := 0 END ;
    nf := nf + Np;
    IF nf >= 0 THEN Send(nonempty) END
  END deposit;

  PROCEDURE fetch(VAR x: ARRAY OF WORD);
  BEGIN nf := nf - Nc;
    IF nf < 0 THEN Wait(nonempty) END ;
    FOR i := 0 TO Nc-1 DO x[i] := buf[out]; out := out + 1 END;
    IF out = N THEN out := 0 END ;
    ne := ne + Nc;
    IF ne >= 0 THEN Send(nonfull) END
  END fetch;

  BEGIN
    ne := N; nf := 0; in := 0; out := 0; Init(nonfull); Init(nonempty)
  END Buffer.

```

(1.31)

1.11.3. Entrada y salida estándar

Por entrada y salida estándar se entiende la transferencia de datos hacia (desde) un sistema de computación a partir de (hacia) genuinamente agentes externos, en particular su operador humano. La entrada puede originarse comúnmente en un teclado y la salida puede figurar en una pantalla de exhibición. En cualquier caso, su característica es que es legible y consta por lo general de una secuencia de caracteres. No obstante, esta condición de legibilidad es responsable de otra complicación incurrida en muchas operaciones genuinas de entrada y salida. Además de la transferencia de datos real, también implican una *transformación de la representación*. Por ejemplo, los números, que por lo general se consideran como unidades atómicas y se representan en forma binaria, necesitan transformarse en notación decimal legible. Las estructuras necesitan representarse en un esquema adecuado, cuya generación se le llama *formateo*.

Cualquiera que sea la transformación, el concepto de secuencia sigue siendo un medio para lograr una simplificación considerable de la tarea. La clave es la observación de que, si el conjunto de datos puede considerarse como una secuencia, la transformación de la secuencia puede realizarse como una sucesión de transformaciones (idénticas) de elementos.

$$T(s_0, s_1, \dots, s_{n-1}) = \langle T(s_0), T(s_1), \dots, T(s_{n-1}) \rangle \quad (1.32)$$

Investigaremos brevemente las operaciones necesarias para transformar las representaciones de números naturales para la entrada y salida. Esto se basa en que un número x representado por la secuencia de dígitos decimales $d = \langle d_{n-1}, \dots, d_1, d_0 \rangle$ tiene el valor

$$x = \sum_{i=0}^{n-1} d_i * 10^i$$

Supóngase ahora que la secuencia d será leída y transformada, y que el valor numérico resultante se asignará a x . El algoritmo simple se da en (1.33); termina con la lectura del primer carácter que no es un dígito. (No se considera el desbordamiento aritmético.)

```

x := 0; Read(ch);
WHILE ("0" <= ch) & (ch <= "9") DO
  x := 10*x + (ORD(ch) - ORD("0")); Read(ch)
END

```

(1.33)

En el caso de la salida la transformación es complicada por el hecho de que la descomposición de x en cifras decimales las genera en orden invertido. El dígito menor se genera primero calculando $x \bmod 10$. Esto requiere un buffer intermedio en la forma de una cola de primero en entrar, último en salir (FILO, *first-in-last-out*). Lo representamos como un arreglo d con índice i y se obtiene el siguiente programa:

```

i := 0;
REPEAT d[i] := x MOD 10; x := x DIV 10; i := i+1
UNTIL x = 0;
REPEAT i := i-1; Write(CHR(d[i] + ORD("0")))
UNTIL i = 0

```

(1.34)

Nota: Una sustitución consistente de la constante 10 en (1.33) y (1.34) por una constante B(>0) producirá rutinas de conversión numérica a y a partir de representaciones con base B. Algunos casos que se utilizan a menudo son B = 8 (octal) y B = 16 (hexadecimal), ya que las multiplicaciones y divisiones implicadas pueden realizarse por medio de simples corrimientos de los números binarios.

Con claridad, no debe necesitarse especificar estas operaciones ubicuas en todos los programas con todos sus detalles. Por tanto postulamos un módulo de servicio que ofrece las operaciones de entrada y salida más comunes en todo este libro, y lo llamamos *InOut*. Sus procedimientos sirven para leer y escribir un carácter, un entero, un número cardinal o bien, una cadena. Debido a su uso frecuente en este libro, la parte de la definición del módulo se enlista aquí en su totalidad.

```
DEFINITION MODULE InOut;
  FROM SYSTEM IMPORT WORD;
  FROM FileSystem IMPORT File;
```

```
CONST EOL = 36C;
VAR Done: BOOLEAN;
  termCH: CHAR; (*caracter de terminacion en ReadInt, ReadCard*)
  in, out: File; (*solo en casos excepcionales*)
```

```
PROCEDURE OpenInput(defext: ARRAY OF CHAR);
(*pedir un nombre de archivo y abrir el archivo de entrada "in".
  Done := "el archivo se abrio con buenos resultados".
  Si se abre, la entrada subsiguiente se lee de este archivo.
```

```
  Si el nombre termina con ".", agregar extension de defext*)
```

```
PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
(*pedir un nombre de archivo y abrir el archivo de salida "out".
  Done:= "el archivo se abrio con buenos resultados".
  Si se abre, la salida subsiguiente se escribe en este archivo*)
```

```
PROCEDURE CloseInput;
(*cierra el archivo de entrada; regresa la entrada a la terminal*)

PROCEDURE CloseOutput;
(*cierra el archivo de salida; regresa la salida a la terminal*)
```

```
PROCEDURE Read(VAR ch: CHAR);
(*Done := NOT in.eof*)

PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
(*leer la cadena, es decir, secuencia de caracteres que no contiene
  espacios en blanco ni caracteres de control; se ignoran los espacios finales.
  La entrada termina con cualquier caracter <= " ";
  este caracter se asigna a termCH.
  DEL se utiliza para retroceder espacios cuando se introduce desde la terminal*)

PROCEDURE ReadInt(VAR x: INTEGER);
(*leer la cadena y convertirla en entera. Sintaxis:
  entero = ["+" | "-"] % digito {digito }.
  Los espacios en blanco finales se ignoran.
  Done := "se leyó el entero"*)
```

```
PROCEDURE ReadCard(VAR x: CARDINAL);
(*leer la cadena y convertirla en cardinal. Sintaxis:
  cardinal = digito { digito } .
  Los espacios finales en blanco se ignoran.
  Done := "se leyó el cardinal"*)
PROCEDURE ReadWrd(VAR w: WORD);
(*Done := NOT in.eof*)

PROCEDURE Write(ch: CHAR);
PROCEDURE WriteLn; (*terminar linea*)
PROCEDURE WriteString(s: ARRAY OF CHAR);
PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
(*escribir el entero x con (al menos) n caracteres en el archivo "out".
  Si n es mayor que el numero de digitos que se necesitan,
  se agregan espacios en blanco antes del numero*)
PROCEDURE WriteCard(x, n: CARDINAL);
PROCEDURE WriteOct(x, n: CARDINAL);
PROCEDURE WriteHex(x, n: CARDINAL);
PROCEDURE WriteWrd(w: WORD);
END InOut.
```

Estos procedimientos de lectura y escritura del módulo no tienen un parámetro que indica la secuencia (archivo) afectada. En cambio, se entiende implicitamente que los datos son leídos del dispositivo de entrada estándar (teclado) y que se escriben en el dispositivo de salida estándar. Sin embargo, esta suposición puede ser anulada invocando los procedimientos *OpenInput* y *OpenOutput*. Piden un nombre de archivo al operador y reemplazan el archivo especificado por el dispositivo estándar, con lo cual se restituye el flujo de los datos. Los procedimientos *CloseInput* y *CloseOutput* retornan la lectura y escritura a los dispositivos estándar.

1.12. BUSQUEDA

La tarea de búsqueda es una de las operaciones más frecuentes en la programación de computadoras. También ofrece un terreno ideal para la aplicación de las estructuras de datos que hasta ahora se han encontrado. Existen diversas variaciones básicas del tema de la búsqueda y se han creado muchos algoritmos referentes a este campo de aplicación. La suposición básica en las presentaciones que siguen es que la colección de datos, entre los cuales se buscará un elemento determinado, es *fija*. Supondremos que este conjunto de N elementos se representa como un arreglo, es decir, como

a: ARRAY [0 .. N-1] OF item

Comúnmente, el tipo *elemento* tiene una estructura de registro con un campo que acuña como una llave. La tarea consiste por tanto en hallar un elemento de *a* cuyo campo llave sea igual a un cierto argumento de búsqueda *x*. El índice resultante *i*, que cumple la igualdad $a[i] = x$, permite por tanto el acceso a los otros campos del registro localizado. Ya que aquí nos interesamos solamente en la tarea de búsqueda, y no nos preocupamos por los datos de los cuales se buscó el elemento en el primer sitio, supondremos que el tipo *elemento* consta sólo de la llave, o sea, *es la llave*.

1.12.1. Búsqueda lineal

Cuando no se brinda más información acerca de los datos buscados, el punto de vista obvio consiste en proceder secuencialmente a través del arreglo a fin de incrementar, paso a paso, el tamaño de la sección, donde se sabe no existe el elemento que se busca. Este enfoque se llama *búsqueda lineal*. Existen dos condiciones que ponen fin a la búsqueda:

1. Se halla el elemento, es decir, $a_i = x$.
2. Se ha rastreado todo el arreglo y no se halló concordancia.

Esto produce el siguiente algoritmo:

```
i := 0;
WHILE (i < N) & (a[i] # x) DO i := i+1 END
```

(1.36)

Nótese que el orden de los términos en la expresión booleana es relevante. La invariante, o sea la condición que se cumplió antes de cada incremento del índice *i*, es

$$(0 \leq i < N) \& (Ak : 0 \leq k < i : a_k \neq x)$$
(1.37)

lo cual expresa que, para todos los valores de *k* menores que *i*, no existe concordancia. De esto y del hecho de que la búsqueda termina sólo si la condición de la cláusula while es falsa, la condición resultante se deduce como

$$((i = N) \text{ OR } (a_i = x)) \& (Ak : 0 \leq k < i : a_k \neq x)$$

Esta condición no sólo es el resultado que se buscaba, sino que también implica que cuando el algoritmo halló una concordancia, encontró aquella con el índice menor, es decir, el primero. $i = N$ implica que no existe concordancia.

La terminación de la repetición es evidentemente segura, ya que en cada etapa del proceso *i* se incrementa y por consiguiente llegará con certeza al límite *N* después de un número finito de etapas; de hecho, después de *N* etapas, si no hay concordancia.

Cada paso requiere evidentemente se incremente el índice, y la evaluación de una expresión booleana. ¿Podría simplificarse esta tarea y podría en consecuencia acelerarse la búsqueda? La única posibilidad yace en determinar una simplificación de la expresión booleana que consta notablemente de dos factores. De aquí, la única oportunidad de obtener una solución más simple radica en establecer una condición consistente en un solo factor que implique ambos factores. Esto es posible sólo si se garantiza que se hallará concordancia, y esto se logra colocando un elemento adicional con valor *x* al final del arreglo. A este elemento auxiliar lo llamamos *sentinel* (*centinela*), ya que previene a la búsqueda de exceder el límite del índice. El arreglo a se declara ahora como

a: ARRAY [0 .. N] OF INTEGER

y el algoritmo de búsqueda lineal con centinela se expresa por

```
a[N] := x; i := 0;
WHILE a[i] # x DO i := i+1 END
```

(1.38)

La condición resultante, que se deriva de la misma invariante anterior, es

$$(a_i = x) \& (Ak : 0 \leq k < i : a_k \neq x)$$

Evidentemente $i = N$ implica que no se encontró concordancia (excepto la del centinela).

1.12.2. Búsqueda binaria

Con claridad no hay forma de acelerar un proceso de búsqueda, a menos que se disponga de más información acerca de los datos que se investigan. Se sabe bien que una búsqueda puede hacerse mucho más efectiva si los datos están ordenados. Imaginese, por ejemplo, un directorio telefónico en el cual los nombres no se enlistaran alfabéticamente. Sería inutilizable en absoluto. Por lo tanto presentaremos un algoritmo que hace uso del conocimiento de que está ordenado, o sea, de la condición

$$Ak: 1 \leq k < N : a_{k-1} \leq a_k$$
(1.39)

La idea clave consiste en inspeccionar un elemento elegido al azar, por decir algo a_m , y compararlo con el argumento de búsqueda *x*. Si es igual a *x*, la búsqueda termina; si es menor que *x*, inferimos que todos los elementos con índices menores que *m* iguales a *m* pueden ser eliminados. Esto produce el siguiente algoritmo llamado *búsqueda binaria*; utiliza dos variables de índice, *L* y *R*, que marcan los extremos Izquierdo (por Left) y Derecho (por Right) de la sección de *a* en los cuales todavía puede hallarse un elemento.

```

L := 0; R := N-1; found := FALSE;
WHILE (L ≤ R) & ~found DO
  m := cualquier valor entre L y R;
  IF a[m] = x THEN found := TRUE
  ELSIF a[m] < x THEN L := m+1
  ELSE R := m-1
  END
END

```

(1.40)

La invariante de iteración, es decir, la condición que se cumplió antes de cada etapa, es

$$(L \leq R) \& (Ak : 0 \leq k < L : a_k < x) \& (Ak : R < k < N : a_k > x) \quad (1.41)$$

de la cual el resultado se deduce como

$$\text{found OR } ((L > R) \& (Ak : 0 \leq k < L : a_k < x) \& (Ak : R < k < N : a_k > x))$$

lo cual implica que

$$(a_m = x) \text{ OR } (Ak : 0 \leq k < N : a_k \neq x)$$

La elección de m es arbitraria en el sentido de que la corrección no depende de ella. Pero si ejerce influencia sobre la efectividad del algoritmo. Nuestro objetivo debe ser el de eliminar tantos elementos como sea posible en cada etapa de búsquedas posteriores, independientemente del resultado de la comparación. La solución óptima consiste en elegir el elemento del medio, ya que esto elimina la mitad del arreglo en cualquier caso. Como resultado, el número máximo de etapas es $\log N$, redondeado al entero más próximo. En consecuencia, este algoritmo ofrece una mejora drástica sobre la búsqueda lineal, donde el número estimado de comparaciones es $N/2$.

La eficiencia puede mejorarse un poco intercambiando las dos cláusulas if. La prueba de la igualdad debe dejarse en segundo término, ya que sólo ocurre una vez y ocasiona la terminación del proceso. Pero más relevante es la pregunta si (como en el caso de la búsqueda lineal) podría hallarse una solución que admita que una condición más simple dé la terminación. En realidad se obtiene dicho algoritmo más rápido si abandonamos el deseo ingenuo de poner fin a la búsqueda en cuanto se establezca una concordancia. Esto parece insensato a simple vista, pero al inspeccionarlo más a fondo comprendemos que lo que se gana en eficiencia tras cada etapa es mayor que la perdida en que se incurre al comparar unos cuantos elementos extra. Recuérdese que el número de etapas es a lo más $\log N$. La solución más rápida se basa en la siguiente invariante:

$$(Ak : 0 \leq k < L : a_k < x) \& (Ak : R \leq k < N : a_k \geq x) \quad (1.42)$$

y la búsqueda continúa hasta que las dos secciones cubren el arreglo en su totalidad.

```

L := 0; R := N;
WHILE L < R DO
  m := (L+R) DIV 2;
  IF a[k] < x THEN L := m+1 ELSE R := m END
END

```

(1.43)

La condición de terminación es $L \geq R$. ¿Se garantiza que se llegará a ella? A fin de establecer esta garantía, debemos demostrar que en todas las circunstancias la diferencia $R - L$ es reducida en cada etapa. $L < R$ es válida al inicio de cada etapa. La media aritmética m satisface, pues, la condición $L \leq m < R$. En consecuencia, la diferencia es en realidad disminuida asignando $m + 1$ a L (incrementando L) o bien m a R (disminuyendo R), y la repetición finaliza con $L = R$. Sin embargo, la invariante $L = R$ todavía no establecen una concordancia. Es verdad que, si $R = N$, no existe concordancia. En caso contrario debemos tomar en consideración que el elemento a [R] nunca había sido comparado. Por lo tanto, se necesita una prueba adicional de igualdad a [R] = x. En contraste con la primera solución (1.40), este algoritmo (al igual que la búsqueda lineal) halla el elemento de concordancia con el índice menor.

1.12.3. Búsqueda en tabla

Una búsqueda a través de un arreglo algunas veces se llama asimismo *búsqueda en tabla*, particularmente si las llaves son por sí mismas objetos estructurados, como arreglos de números o caracteres. El último es el caso que más frecuentemente se presenta; los arreglos de caracteres se denominan *cadenas* o bien palabras. Definiremos un tipo *Cadena* como

$$\text{String} = \text{ARRAY [0..M-1]} \text{OF CHAR} \quad (1.44)$$

y sea que el orden en las cadenas x y y se defina como sigue:

$$(x = y) = (Aj: 0 \leq j < M : x_j = y_j)$$

$$(x < y) = Ei: 0 \leq i < N : ((Aj: 0 \leq j < i : x_j = y_j) \& (x_i < y_i))$$

A fin de establecer una concordancia, evidentemente debemos hallar todos los caracteres de los comparandos iguales. Dicha comparación de operandos estructurados resulta ser por tanto una búsqueda de una pareja desigual de comparandos, o sea, una búsqueda de desigualdad. Si no existe un par desigual, se establece la igualdad. Suponiendo que la longitud de las palabras es muy pequeña, a saber menor que 30, utilizaremos una búsqueda lineal en la siguiente solución.

En muchas aplicaciones prácticas, se desea considerar las cadenas como poseedoras de *longitud variable*. Esto se logra asociando una indicación de longitud con cada valor individual de la cadena. Utilizando el tipo que se declaró antes, la longitud no debe exceder la longitud máxima M. Este esquema ofrece la flexibilidad suficiente en muchos casos, no obstante que evita las complicaciones de la asignación dinámica de almacenamiento. Se utilizan más comúnmente dos representaciones de longitudes de cadena y son:

1. La longitud se especifica en forma implícita agregando al final un carácter de terminación que no ocurre en caso contrario. Por lo general, el valor no imprimible OC se utiliza con este fin. (Es importante para las aplicaciones subsiguientes que sea el *menor* carácter del conjunto de caracteres.)
2. La longitud se almacena explícitamente como el primer elemento del arreglo, o sea la cadena s tiene la forma

$$s = s_0, s_1, s_2, \dots, s_{N-1}$$

donde $s_1 \dots s_{N-1}$ son los caracteres reales de la cadena y $s_0 = \text{CHR}(N)$. Esta solución tiene la ventaja de que la longitud está disponible y la desventaja de que la longitud máxima se limita al tamaño del conjunto de caracteres (256).

Para el siguiente algoritmo de búsqueda, nos apegaremos al primer esquema. Una comparación de la cadena toma por tanto la forma

```
i := 0;
WHILE (x[i] = y[i]) & (x[i] ≠ 0C) DO i := i+1 END
```

(1.45)

El carácter de terminación funciona ahora como un centinela y la invariante de iteración es

$Aj: 0 \leq j < i : x_j = y_j \neq 0C,$

y la condición resultante es por consiguiente

$((x_i \neq y_i) \text{ OR } (x_i = 0C)) \& (Aj: 0 \leq j < i : x_j = y_j \neq 0C)$

Esta establece una concordancia entre x y y , siempre que $x_1 = y_1$, y establece $x < y$, si $\bar{x}_1 < y_1$.

Ahora estamos preparados para volver a la tarea de la búsqueda en tabla. Requiere una búsqueda anidada, es decir, una búsqueda a través de los valores contenidos en la tabla, y para cada valor una secuencia de comparaciones entre componentes. Por ejemplo, sea la tabla T y el argumento de búsqueda x se definan como

```
T: ARRAY [0 .. N-1] OF String;
x: String
```

Suponiendo que N puede ser muy grande y que la tabla está ordenada alfabéticamente, utilizaremos una búsqueda binaria. Utilizando los algoritmos de búsqueda binaria (1.43) y comparación de cadenas (1.45) que se elaboraron antes, obtenemos el siguiente segmento de programa.

```
L := 0; R := N;
WHILE L < R DO
  m := (L+R) DIV 2; i := 0;
  WHILE (T[m,i] = x[i]) & (x[i] ≠ 0C) DO i := i+1 END ;
  IF T[m,i] < x[i] THEN L := m+1 ELSE R := m END
END ;
IF R < N THEN i := 0;
  WHILE (T[R,i] = x[i]) & (x[i] ≠ 0C) DO i := i+1 END
END
(* (R < N) & (T[R,i] = x[i]) establecen un ajuste*)
```

(1.46)

1.12.4. Búsqueda directa de cadena

Un tipo de búsqueda que se halla a menudo es el así llamado *búsqueda de cadena*. Se caracteriza de la manera siguiente. Dado un arreglo s de N elementos y un arreglo p de M elementos, donde $0 < M \leq N$, declarados como

$s: \text{ARRAY}[0 .. N-1] \text{ OF item}$
 $p: \text{ARRAY}[0 .. M-1] \text{ OF item}$

la búsqueda de cadena es la tarea de hallar la primera presencia de p en s . Comúnmente, los elementos son caracteres; entonces s puede considerarse como un texto y p como un patrón de búsqueda o palabra, y deseamos encontrar la primera presencia de la palabra en el texto. Esta operación es básica para todo sistema de procesamiento de textos, y existe un interés obvio en determinar un algoritmo eficiente para realizar esta tarea. Sin embargo, antes de poner particular atención a la eficiencia, presentaremos primero un algoritmo de búsqueda directa. Lo llamaremos *búsqueda directa de cadena*.

Una formulación más precisa del resultado que se busca de una búsqueda es indispensable antes de que intentemos especificar un algoritmo para calcularlo. Sea que el resultado sea el índice i que apunta hacia la primera presencia de una concordancia del patrón con la cadena. Con este fin, presentamos un atributo $P(i, j)$.

$P(i,j) = Ak : 0 \leq k < j : s_{i+k} = p_k$

(1.47)

Por tanto, evidentemente nuestro índice i resultante debe cumplir $P(i, M)$. Pero esta condición no es suficiente. Ya que la búsqueda debe ubicar la *primera* aparición del patrón, $P(k, M)$ debe ser falsa para toda $k < i$. Esta condición se simboliza por $Q(i)$.

$Q(i) = Ak : 0 \leq k < i : \neg P(k, M)$

(1.48)

El problema planteado sugiere de inmediato formular la búsqueda como una iteración de comparaciones y se propone el siguiente punto de vista:

```
i := -1;
REPEAT i := i+1; (* Q(i) *)
  hallado := P(i,M)
UNTIL hallado OR (i = N-M)
```

El cálculo de P vuelve a producir naturalmente una iteración de comparaciones de caracteres individuales. Cuando se aplica el teorema de DeMorgan a P , parece que la iteración debe ser una búsqueda de desigualdad entre el patrón y los caracteres de la cadena correspondientes.

$P(i,j) = (Ak : 0 \leq k < j : s_{i+k} = p_k) = (\neg Ek : 0 \leq k < j : s_{i+k} \neq p_k)$

El resultado del siguiente refinamiento es una repetición dentro de una repetición. Los atributos P y Q se insertan en lugares adecuados en el programa como comentarios. Actúan como invariantes de los ciclos de iteración.

```
i := -1;
REPEAT i := i+1; j := 0; (* Q(i) *)
  WHILE (j < M) & (s[i+j] = p[j]) DO (* P(i,j+1) *) j := j+1 END
  (* Q(i) & P(i,j) & ((j = M) OR (s[i+j] ≠ p[j])) *)
UNTIL (j = M) OR (i = N-M)
```

(1.49)

El término $j = M$ en la condición de terminación corresponde en realidad a la condición *hallado*, ya que implica a $P(i, M)$. El término $i = N - M$ implica a $Q(N - M)$ y por tanto la no existencia de una concordancia en ninguna parte de la cadena. Si la iteración continúa con $j < M$, entonces debe hacerlo también con $s_i + j \neq p_j$. Esto implica, de acuerdo con (1.47), a $\sim P(i, j)$, que de acuerdo con (1.48) implica a $Q(i + 1)$, que establece $Q(i)$ después del siguiente incremento de i .

Análisis de la búsqueda directa de cadena. Este algoritmo opera muy eficazmente si podemos suponer que ocurre una inconcordancia entre los pares de caracteres después de cuando más unas cuantas comparaciones en el ciclo interno. Este es probablemente el caso, si la cardinalidad del tipo elemento es grande. Para búsquedas de textos con un tamaño del conjunto de caracteres de 128 podemos bien suponer que ocurre una inconcordancia después de inspeccionar solamente 1 o 2 caracteres. No obstante, el rendimiento en el peor de los casos es alarmante. Considérese, por ejemplo, que la cadena consta de $N - 1$ Aes seguidas de una sola B, y que el modelo consta de $M - 1$ Aes seguidas de una B. Por tanto en el orden de $N * M$ comparaciones se necesitan para hallar la concordancia al final de la cadena. Como después se observará, afortunadamente existen métodos que mejoran en forma drástica el comportamiento en el peor de los casos.

1.12.5. Búsqueda de cadena de Knuth-Morris-Pratt

Por el año de 1970, D. E. Knuth, J. H. Morris y V. R. Pratt inventaron un algoritmo que requiere esencialmente sólo N comparaciones de caracteres, aun en el peor de los casos [1-8]. El nuevo algoritmo se basa en la observación de que iniciando la siguiente comparación del patrón en su inicio cada vez, podemos estar descartando información valiosa. Después de una concordancia parcial del inicio del patrón con caracteres correspondientes de la cadena, en realidad conocemos la última parte de la cadena y quizás podríamos precompilar algunos datos (del patrón) que podrían utilizarse para lograr un más rápido avance en la cadena de texto. El ejemplo siguiente de una búsqueda de la palabra *Hooligan* (revoltoso) ilustra el principio del algoritmo. Los caracteres subrayados son aquellos que fueron comparados. Nótese que cada vez que dos caracteres comparados no concuerdan, el modelo es totalmente cambiado, ya que un cambio menor posiblemente no podría conducirnos a una concordancia completa.

```

Hoola-Hoola girls like Hooligans.
Hooligan
Hooligan
Hooligan
Hooligan
Hooligan
Hooligan
Hooligan
.....
```

Utilizando los atributos P y Q definidos por (1.47) y (1.48), el algoritmo KMP es el siguiente:

```

i := 0; j := 0;
WHILE (j < M) & (i < N) DO
  (* Q(i-j) & P(i-j, j) *)
  WHILE (j >= 0) & (s[i] # p[j]) DO j := D END ;
  i := i+1; j := j+1
END

```

(1.50)

Esta formulación no es muy completa según se admite, ya que contiene un valor de cambio D no especificado. Volveremos a considerarlo brevemente, pero primero nótese que las condiciones $Q(i-j)$ y $P(i-j, j)$ se conservan como invariantes globales, a las cuales debemos añadir las relaciones $0 \leq i < N$ y $0 \leq j < M$. Esto sugiere que debemos abandonar la noción de que i siempre marca la posición actual del primer carácter del patrón en el texto. En su lugar, la posición de alineación del patrón es ahora $i-j$.

Si el algoritmo termina debido a la condición $j = M$, el término $P(i-j, j)$ de la invariante implica a $P(i-M, M)$, es decir, de acuerdo con (1.47) una concordancia en la posición $i-M$. En caso contrario termina con $i = N$, y ya que $j < M$, la invariante $Q(i)$ implica que no existe en absoluto ninguna concordancia.

Ahora debemos demostrar que el algoritmo nunca falsifica la invariante. Es fácil probar que se establece al inicio con los valores $i = j = 0$. Primero investiguemos el efecto de las dos proposiciones que incrementan i y j en 1. Es evidente que ninguna representa un corrimiento del modelo hacia la derecha, ni tampoco falsifican $Q(i-j)$, ya que la diferencia permanece inalterada. Pero ¿podrían falsificar $P(i-j, j)$, segundo factor de la invariante? Observamos que en este punto la negación de la cláusula while interna es válida, o sea, $j < 0$ o bien, $s_i = p_j$. La última asignación amplía la concordancia parcial y establece $P(i-j, j+1)$. En el primer caso, postulamos que $P(i-j, j+1)$ se cumple también. En consecuencia, el incremento de i y j en 1 no puede falsificar tampoco la invariante. La única asignación que queda en el algoritmo es $j := D$. Simplemente postularemos que el valor D siempre sea tal que la sustitución j por D conserve la invariante.

A fin de hallar una expresión adecuada para D , debemos primero entender el efecto de la asignación. Siempre que $D < j$, representa un *corrimiento del patrón hacia la derecha* de $j-D$ posiciones. Naturalmente, se desea que este corrimiento sea lo más grande posible, es decir, que D sea lo más pequeña posible. Esto se ilustra en la figura 1.10.

Evidentemente, la condición $P(i-D, D) \& Q(i-D)$ debe cumplirse antes de asignar D a j , si la invariante $P(i-j, j) \& Q(i-j)$ debe cumplirse de aquí en adelante. Esta precondition

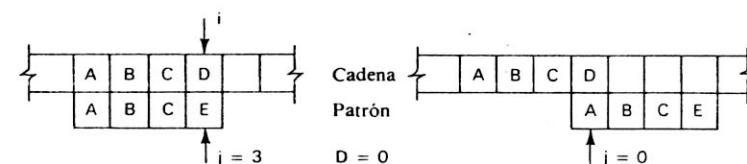


Fig. 1.10. Asignación $j := D$ desplaza el modelo $j-D$ posiciones.

es, pues, nuestra guía para hallar una expresión adecuada para D. La observación importante es que gracias a $P(i-j, j)$ sabemos que

$$s_{i-j} \dots s_{i-1} = p_0 \dots p_{j-1}$$

(solamente habíamos examinado los primeros j caracteres del modelo y observamos que concuerdan). Por lo tanto, la condición $P(i-D, D)$ con $D \leq j$, es decir,

$$p_0 \dots p_{D-1} = s_{i-D} \dots s_{i-1}$$

se traduce en

$$p_0 \dots p_{D-1} = p_{j-D} \dots p_{j-1} \quad (1.51)$$

y (con el fin de establecer la invariancia de $Q(i-D)$) el atributo $\sim P(i-k, M)$ para $k = 1 \dots j-D$ se traduce en

$$p_0 \dots p_{k-1} \neq p_{j-k} \dots p_{j-1} \quad \text{para toda } k = 1 \dots j-D \quad (1.52)$$

El resultado esencial es que el valor D aparentemente es determinado sólo por el patrón y no depende de la cadena de texto. Las condiciones (1.51) y (1.52) nos indican que para hallar D debemos, para toda j, buscar la D más pequeña y por lo tanto, la secuencia más larga de caracteres del patrón que anteceden la posición j, la cual concuerda un número

Ejemplos	
Cadena	
Patrón	
Patrón desplazado	
	$j = 5, d_5 = 4, (\text{despl. m\'ax} = j - d_j = 1)$ $p_0 \dots p_3 = p_1 \dots p_4$
	$j = 5, d_5 = 2, (\text{despl. m\'ax} = j - d_j = 3)$ $p_0 \dots p_1 = p_3 \dots p_4$ $p_0 \dots p_2 \neq p_2 \dots p_4$ $p_0 \dots p_3 \neq p_1 \dots p_4$
	$j = 5, d_5 = 0, (\text{despl. m\'ax} = j - d_j = 5)$ $p_0 \dots p_0 \neq p_4 \dots p_4$ $p_0 \dots p_1 \neq p_3 \dots p_4$ $p_0 \dots p_2 \neq p_2 \dots p_4$ $p_0 \dots p_3 \neq p_1 \dots p_4$

Fig. 1.11. Coincidencias del patrón parcial y cálculo de d_j .

igual de caracteres al inicio del patrón. A D la simbolizaremos por d_j para una j dada. Ya que estos valores dependen únicamente del patrón, la tabla auxiliar d puede calcularse antes de dar inicio a la búsqueda real; este cálculo cuenta para la *precompilación* del patrón. Este esfuerzo sólo es útil evidentemente si el texto es considerablemente más largo que el patrón ($M < N$). Si han de hallarse múltiples presencias del mismo patrón, pueden reutilizarse los mismos valores de d. Los ejemplos que siguen ilustran la función de d.

El último ejemplo de la figura 1.11 sugiere que podemos mejorar ligeramente nuestra posición; si el carácter p_j hubiera sido una A en vez de una F, sabríamos que el carácter de cadena correspondiente posiblemente no podría ser una A, ya que $s_i \neq p_j$ pone fin al ciclo. En consecuencia, un corrimiento de 5 podría no conducirnos a una concordancia después, y podríamos de igual manera incrementar los corrimientos a 6 (véase la figura 1.11, parte superior). Tomando esto en consideración, se vuelve a definir el cálculo de d_j como la búsqueda de la más larga secuencia de concordancia

$$p_0 \dots p_{d_j-1} = p_{j-d_j} \dots p_{j-1}$$

con la restricción adicional de $p_{d_j} \neq p_j$. Si no hay concordancia en absoluto, hacemos $d_j = -1$, con lo cual se indica que el patrón completo es corrido más allá de su posición actual (véase la figura 1.12, parte inferior).

Evidentemente, el cálculo de d_j nos presenta la primera aplicación de la búsqueda de cadena, y podemos de igual manera usar la versión KMP rápida. Esta se muestra en el programa 1.2, el cual consta de las partes siguientes: Primero, se lee la cadena s; luego sigue una repetición que empieza con la lectura de un patrón, seguida de la precompilación del mismo y finalmente, de la búsqueda misma.

```

MODULE KMP;
FROM InOut IMPORT
  OpenInput, CloseInput, Read, Write, WriteLn, Done;

CONST Mmax = 100; Nmax = 10000; ESC = 33C;

VAR i, j, k, k0, M, N: INTEGER;
  ch: CHAR;
  p: ARRAY [0 .. Mmax-1] OF CHAR; (*patr\'on*)
  s: ARRAY [0 .. Nmax-1] OF CHAR; (*cadena*)
  d: ARRAY [0 .. Mmax-1] OF INTEGER;

BEGIN OpenInput("TEXT"); N := 0; Read(ch);
  WHILE Done DO
    Write(ch); s[N] := ch; N := N+1; Read(ch)
  END ;
  CloseInput;
  LOOP WriteLn; Write(">"); M := 0; Read(ch);
  WHILE ch > " " DO
    Write(ch); p[M] := ch; M := M+1; Read(ch)
  END;
  
```

```

END ;
WriteLn;
IF ch = ESC THEN EXIT END ;
j := 0; k := -1; d[0] := -1;
WHILE j < M-1 DO
  WHILE (k >= 0) & (p[j] # p[k]) DO k := d[k] END ;
  j := j+1; k := k+1;
  IF p[j] = p[k] THEN d[j] := d[k] ELSE d[j] := k END
END ;
i := 0; j := 0; k := 0;
WHILE (j < M) & (i < N) DO
  WHILE k <= i DO Write(s[k]); k := k+1 END ;
  WHILE (j >= 0) & (s[i] # p[j]) DO j := d[j] END ;
  i := i+1; j := j+1
END ;
IF j = M THEN Write("!!") (* hallado *) END
END
END KMP.

```

Programa 1. Búsqueda de cadena de Knuth-Morris-Pratt.

Análisis de la búsqueda KMP. El análisis exacto del rendimiento de la búsqueda KMP es, al igual que el algoritmo mismo, muy complicado. En [1-8] sus inventores demuestran que el número de comparaciones de caracteres está en el orden de $M + N$, lo cual sugiere una mejora sustancial sobre $M \cdot N$ de la búsqueda directa. También indican la propiedad bienvenida de que el apuntador de rastreo i nunca da marcha atrás, en tanto que en la búsqueda directa de cadena el análisis siempre comienza en el primer carácter.

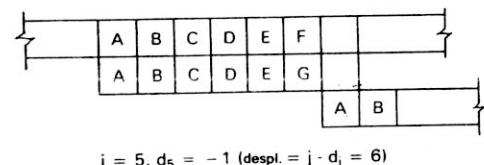
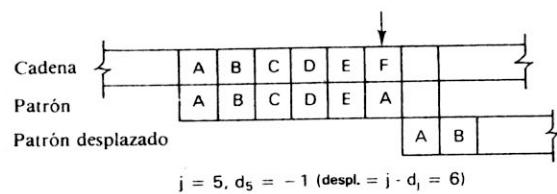


Fig. 1.12. Desplazamiento del patrón en la posición después del último carácter.

ter del modelo después de una inconcordancia, y por lo tanto puede comprender caracteres que en realidad ya habían sido examinados. Esto puede ocasionar problemas difíciles de manejar cuando la cadena es leída del almacenamiento secundario donde los retrocesos son costosos. Aun cuando la entrada sea manejada por buffer el patrón puede ser tal que el retroceso vaya más allá del contenido del buffer.

1.12.6. Búsqueda de cadena de Boyer-Moore

El ingenioso esquema de la búsqueda KMP produce genuinos beneficios sólo si una inconcordancia se viera precedida por una concordancia parcial de alguna longitud. Sólo en este caso se incrementa el corrimiento del patrón a más de 1. Por desgracia, ésta es la excepción en vez de la regla; las concordancias ocurren más rara vez que las inconcordancias. Por lo tanto la ventaja de usar la estrategia KMP es marginal en muchos casos de búsqueda de texto normal. El método que se analizará aquí en realidad no sólo mejora el rendimiento en el peor de los casos, sino también en el caso promedio. Fue inventado por R. S. Boyer y J. S. Moore por el año de 1975 y lo llamaremos *búsqueda BM*. Aquí presentaremos una versión simplificada de la búsqueda BM antes de proseguir a la dada por Boyer y Moore.

La búsqueda BM se basa en la idea no convencional de empezar a comparar caracteres al *final* del patrón en vez de al inicio. Como en el caso de la búsqueda KMP, el patrón se precompila en una tabla d antes de que comience la búsqueda real. Sea d_x , para todo carácter x del conjunto de caracteres, la distancia de la presencia de más a la derecha de x en el patrón desde su extremo. Ahora supóngase que se descubrió una inconcordancia entre la cadena y el patrón. En consecuencia el patrón puede correrse de inmediato hacia la derecha d_{pM-1} posiciones, cantidad que muy probablemente es mayor que 1. Si p_{M-1} no se presenta en el patrón en lo absoluto, el corrimiento es todavía mayor, o sea, igual a toda la longitud del citado patrón. El ejemplo que sigue ilustra este proceso.

Hoola-Hoola girls like Hooligans.
Hooligan
Hooligan
Hooligan
Hooligan
Hooligan
Hooligan

Ya que las comparaciones de caracteres individuales proceden ahora de derecha a izquierda, las siguientes versiones ligeramente modificadas de los atributos P y Q son más adecuados.

$$\begin{aligned} P(i,j) &= Ak: j \leq k < M : s_{i-j+k} = p_k \\ Q(i) &= Ak: 0 \leq k < i : \sim P(i,0) \end{aligned} \quad (1.53)$$

Estos atributos se emplean en la siguiente formulación del algoritmo BM para representar las condiciones invariantes.

$$\begin{aligned} i &:= M; j := M; \\ \text{WHILE } (j > 0) &\& (i < N) \text{ DO} \end{aligned} \quad (1.54)$$

```
(* Q(i-M) *) j := M; k := i;
WHILE (j > 0) & (s[k-1] = p[j-1]) DO
  (* P(k-j, j) & (k-j = i-M) *)
  k := k-1; j := j-1
END ;
i := i + d[s[i-1]]
END
```

Los índices cumplen las desigualdades $0 \leq j \leq M$ y $0 \leq i, k \leq N$. Por lo tanto, la terminación con $j=0$, junto con $P(k-j, j)$, implica a $P(k, 0)$, o sea una concordancia en la posición k . La terminación con $j>0$ demanda que $j=N$; en consecuencia $Q(i-M)$ implica a $Q(N-M)$, lo cual señala que no hay concordancia. Por supuesto, todavía tenemos que convencernos de que $Q(i-M)$ y $P(k-j, j)$ son realmente invariantes de las dos repeticiones. Estas se cumplen de manera trivial cuando la repetición empieza, ya que $Q(0)$ y $P(x, M)$ siempre son verdaderos.

Consideremos primero el efecto de las dos proposiciones que disminuyen k y j . $Q(i-M)$ no es afectada y, ya que $s_{k-1} = p_{j-1}$ se había establecido, $P(k-j, j-1)$ se mantiene como precondición, garantizando a $P(k-j, j)$ como postcondición. Si el ciclo interno termina con $j>0$, el hecho de que $s_{k-1} \neq p_{j-1}$ implica $\sim P(k-j, 0)$, porque

$$\sim P(i, 0) = \exists k: 0 \leq k < M : s_{i+k} \neq p_k$$

Además, como $k-j=M-i$, $Q(i-M) \& \sim P(k-j, 0) = Q(i+1-M)$, estableciendo una inconcordancia en la posición $i-M+1$.

A continuación debemos demostrar que la proposición $i := i + d_{s_{i-1}}$ nunca falsifica la invariante. Este es el caso, siempre y cuando la asignación anterior $Q(i + d_{s_{i-1}}, M)$ se garantice. Ya que sabemos que $Q(i+1-M)$ se cumple, basta establecer $\sim P(i+h-M)$ para $h=2, 3, \dots, d_{s_{i-1}}$. Ahora recordaremos que d_x se define como la distancia de la presencia de más a la derecha de x en el patrón desde el extremo. Esto se expresa formalmente como

$$\forall k: M-d_x \leq k < M-1 : p_k \neq x$$

Sustituyendo s_j para x , se obtiene

$$\begin{aligned} \text{Ah: } M-d_{s_{i-1}} &\leq h < M-1 : s_{i-1} \neq p_h \\ \text{Ah: } 1 < h &\leq d_{s_{i-1}} : s_{i-1} \neq p_{h-M} \\ \text{Ah: } 1 < h &\leq d_{s_{i-1}} : \sim P(i+h-M) \end{aligned}$$

El siguiente programa incluye la estrategia simplificada de Boyer-Moore que se presentó en un medio similar al del programa anterior de búsqueda KMP. Nótese como detalle que se usa una proposición repeat (de repetición) en el ciclo interno, que incrementa k y j antes de comparar s y p . Esto elimina los términos -1 contenidos en las expresiones de índice.

```
MODULE BM; (1.60)
FROM InOut IMPORT
  OpenInput, CloseInput, Read, Write, WriteLn, Done;
CONST Mmax = 100; Nmax = 10000;
VAR i, j, k, i0, M, N: INTEGER;
ch: CHAR;
p: ARRAY [0 .. Mmax-1] OF CHAR; (*patrón*)
s: ARRAY [0 .. Nmax-1] OF CHAR; (*cadena*)
d: ARRAY [0C .. 177C] OF INTEGER;

BEGIN OpenInput("TEXT"); N := 0; Read(ch);
WHILE Done DO
  Write(ch); s[N] := ch; N := N + 1; Read(ch)
END;
CloseInput;
LOOP WriteLn; Write(">"); M := 0; Read(ch);
  WHILE ch > " " DO
    Write(ch); p[M] := ch; M := M + 1; Read(ch)
  END;
  WriteLn;
  IF ch = 33C THEN EXIT END ;
FOR ch := 0C TO 177C DO d[ch] := M END ;
FOR j := 0 TO M-2 DO d[p[j]] := M-j-1 END ;

i := M; i0 := 0;
REPEAT
  WHILE i0 < i DO Write(s[i0]); i0 := i0 + 1 END ;
  j := M; k := i;
  REPEAT k := k-1; j := j-1
  UNTIL (j < 0) OR (p[j] # s[i]);
  i := i + d[s[i-1]];
  UNTIL (j < 0) OR (i >= N);
  IF j < 0 THEN Write("!")
END
END BM.
```

Programa 1. Búsqueda simplificada de cadena de Boyer-Moore.

Análisis de la búsqueda Boyer-Moore. La publicación original de este algoritmo [1-9] contiene un análisis detallado de su rendimiento. La propiedad enfatizable es que en todos excepto los casos especialmente construidos, requiere sustancialmente menos de N comparaciones. En el caso más afortunado, donde el último carácter del modelo siempre corresponde a un carácter deseigual del texto, el número de comparaciones es N/M .

Los autores dan varias ideas acerca de posibles mejoras adicionales. Una consiste en combinar la estrategia que se explicó antes, la cual brinda mayores etapas de corrimiento cuando ocurre una *inconcordancia*, con la estrategia de Knuth-Morris-Pratt, la cual admite corrimientos mayores después de detectar una *concordancia* (parcial). Este método requiere dos tablas precalculadas; d1 es la tabla que se usó antes y d2 es la que corresponde a la del algoritmo KMP. La etapa que se emprende es, pues, la mayor de las dos y ambas indican que quizá ninguna etapa menor podría llevarnos a una concordancia de caracteres. Evitamos seguir extendiéndonos en el tema, ya que la complejidad adicional de la generación de la tabla y la búsqueda misma no parecen producir ninguna mejora apreciable de eficiencia. De hecho, el “exceso” adicional es mayor y crea incertidumbre de si la complicada extensión es una mejora o un deterioro.

EJERCICIOS

- 1.1. Suponga que las cardinalidades de los tipos estándar INTEGER, REAL y CHAR se simbolizan por c_I , c_R y c_{CH} . ¿Cuáles son las cardinalidades de los siguientes tipos de datos que se definieron como ejemplos en este capítulo: sexo, BOOLEANO, día de la semana, letra, dígito, oficial, renglón, alfa, complejo, fecha, persona, coordenada, condición de cinta?
- 1.2. ¿Cómo representaría usted variables de los tipos que se mencionan en el ejercicio 1.1:
 - (a) En la memoria de su computadora?
 - (b) En FORTRAN?
 - (c) En su lenguaje de programación favorito?
- 1.3. ¿Cuáles son las secuencias de instrucciones (en su computadora) para hacer los siguientes:
 - (a) Traer y almacenar operaciones para elementos de registros y arreglos empacados?
 - (b) Operaciones de conjunto, inclusive la prueba de membresía?
- 1.4. ¿Puede verificarse el uso correcto de registros variantes en el instante de la corrida del programa? ¿Puede verificarse en el momento de la compilación?
- 1.5. ¿Cuáles son las razones para definir ciertos conjuntos de datos como secuencias en vez de arreglos?
- 1.6. Se tiene un itinerario de salidas y llegadas de un tren que enlista los servicios diarios en varias líneas de un sistema ferroviario. Obtenga una representación de estos datos en términos de arreglos, registros o secuencias, que sea adecuada para buscar tiempos de llegada y salida, dada cierta estación y la dirección deseada del tren.
- 1.7. Dado un texto T en la forma de secuencia y listas de un número pequeño de palabras en la forma de dos arreglos A y B. Suponga que las palabras son arreglos breves de caracteres de una longitud pequeña y máxima fija. Escriba un programa que transforme el texto T en un texto S sustituyendo cada presencia de una palabra A_i por su palabra correspondiente B_i .
- 1.8. Compare las tres versiones siguientes de la búsqueda binaria con (1.43). ¿Cuáles de los tres programas son correctos? Determine las invariantes relevantes. ¿Cuáles versiones son más eficientes? Se suponen las siguientes variables y la constante $N > 0$:

VAR i, j, k, x: INTEGER;
a: ARRAY[1 .. N] OF INTEGER;

Programa A:

i := 1; j := N;

```

REPEAT k := (i+j) DIV 2;
  IF a[k] < x THEN i := k ELSE j := k END
UNTIL (a[k] = x) OR (i ≥ j)

```

Programa B:

```

i := 1; j := N;
REPEAT k := (i+j) DIV 2;
  IF x ≤ a[k] THEN j := k-1 END ;
  IF a[k] ≤ x THEN i := k+1 END
UNTIL i > j

```

Programa C:

```

i := 1; j := N;
REPEAT k := (i+j) DIV 2;
  IF x < a[k] THEN j := k ELSE i := k+1 END
UNTIL i ≥ j

```

Sugerencia: Todos los programas deben finalizar con $a_k \neq x$, si tal elemento existe o bien $a_k \neq x$, si no hay ningún elemento con valor x .

- 1.14. Una compañía organiza una votación para determinar el éxito de sus productos. Sus productos son discos y cintas de éxitos musicales y los más populares serán difundidos en un desfile de éxitos. La población votante será dividida en cuatro categorías de acuerdo con el sexo y la edad (a saber, menor que o igual a 20 y mayores de 20). A todas las personas se les pide mencionen cinco éxitos musicales. Las melodías se identifican con los números 1 a N (por decir algo, N=30). Los resultados de la votación se codificarán adecuadamente como una secuencia de caracteres. Sugerencia: utilice los procedimientos *Read* y *ReadInt* para leer los valores de la votación.

```

TYPE hit = [0 .. N-1];
  sexo = (masculino, femenino);
  respuesta =
    RECORD nombrecompleto, primernombre: alfa;
      s: sexo;
      edad: INTEGER;
      eleccion: ARRAY [0 .. 4] OF hit
    END;
  VAR escrutinio: sequence

```

Este archivo es la entrada de un programa que calcula los siguientes resultados:

1. Una lista de éxitos musicales en orden de su popularidad. Cada valor consta del número de la melodía y del número de veces que se mencionó en la votación. Las piezas que nunca se mencionaron están omitidas de la lista.

2. Cuatro listas por separado con los nombres y apellidos de todos los votantes quienes hayan mencionado en primer término uno de los tres éxitos más populares en su categoría.

Las cinco listas tendrán antepuestos títulos adecuados.

BIBLIOGRAFIA

- 1-1. O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare. *Structured Programming* (New York: Academic Press, 1972), pp. 155-65.
- 1-2. C.A.R. Hoare. Notes on data structuring; in *Structured Programming* Dahl, Dijkstra, and Hoare, pp. 83-174.
- 1-3. K. Jensen and N. Wirth. *Pascal user Manual and Report* (Berlin: Springer-Verlag, 1974).
- 1-4. N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14, No. 4 (1971), 221-27.
- 1-5. ----- *Programming in Modula-2*. (Berlin, New York: Springer-Verlag, 1982).
- 1-6. -----, On the composition of well-structured programs. *Computing Surveys*, 6, No. 4, (1974) 247-59.
- 1-7. C.A.R. Hoare. The Monitor: A operating systems structuring concept. *Comm. ACM*.
- 1-8. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM H. Comput.*, 6, 2 (June 1977), 323-349.
- 1-9. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20, 10 (Oct. 1977), 762-772.

La dependencia en la elección de un algoritmo respecto a la estructura de los datos por procesar (un fenómeno general) es tan grande en el caso de la clasificación, que los métodos de clasificación suelen dividirse en dos categorías, a saber: *clasificación de arreglos* y *clasificación de archivos (secuenciales)*. Las dos clases a menudo reciben el nombre de *clasificación interna* y *externa*, porque los arreglos se guardan en la memoria “interna” que es rápida, de alta velocidad y acceso aleatorio y los archivos se localizan adecuadamente en la memoria “externa”, más lenta pero más “espaciosa”, que se basa en los dispositivos de movimiento mecánico (discos y cintas). La importancia de esta distinción es evidente en el ejemplo de la clasificación de tarjetas numeradas. Estructurarlas en forma de arreglo equivale a ponerla delante del clasificador, de manera que cada tarjeta sea visible y accesible individualmente (véase la figura 2.1).

La estructuración de tarjetas como un archivo implica que, en cada pila, sólo la tarjeta de la cima es visible (véase la figura 2.2). Tal restricción tendrá sin duda consecuencias graves en el método de clasificación que se aplique, pero es inevitable si el número de tarjetas que debe mostrarse es mayor que la mesa disponible.

Antes de seguir adelante, incluiremos un poco de terminología y notación que se usará en todo este capítulo. Si tenemos los elementos

$$a_1, a_2, \dots, a_n$$

la clasificación consistirá en permutarlos y ponerlos en un arreglo

$$a_{k_1}, a_{k_2}, \dots, a_{k_n}$$

tal que, dada una función de ordenación f ,

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \dots \leq f(a_{k_n}). \quad (2.1)$$

Normalmente la función de ordenación no se evalúa conforme a una regla de computación especificada, sino que se guarda como un componente explícito (campo) del ele-



Fig. 2.1 Clasificación por arreglos.



Fig. 2.2 Clasificación por archivos.

mento. Su valor recibe el nombre de *llave* (key) del elemento. En consecuencia, la estructura de registro es especialmente idónea para representar los elementos y pudiera, por ejemplo, ser declarada así:

```
TYPE item = RECORD llave: INTEGER;
              (*otros componentes declarados aquí*)
            END
```

(2.2)

Los otros componentes representan datos importantes referentes a los elementos de la colección; la llave se limita a suponer el propósito de identificar los elementos. En lo tocante a nuestros algoritmos de clasificación, la llave es el único componente relevante, sin que haya necesidad de definir ninguno de los restantes. En las siguientes explicaciones, prescindiremos por eso de cualquier información conexa y supondremos que el tipo del elemento se define como INTEGER. La elección de INTEGER como el tipo de la llave es un poco arbitraria. Sobra decir que podría usarse también cualquier tipo en el cual se define una relación de ordenación total.

Un método de clasificación se llama *estable* si el orden relativo de elementos con iguales llaves permanece inalterado en el proceso de clasificación. La estabilidad de la clasificación resulta a veces conveniente si los elementos ya están ordenados (clasificados) conforme a algunas llaves secundarias, o sea propiedades que no se reflejan en la llave (primaria).

Este capítulo no pretende ser un examen completo de las técnicas de clasificación, sino que se ejemplifican con mayor detalle algunos métodos selectos y específicos. El lector que desee una exposición amplia sobre la clasificación deberá consultar el excelente y exhaustivo compendio hecho por D.E. Knuth [2-7] (véase también a Lorin [2-10]).

2.2. CLASIFICACION DE ARREGLOS

El requisito más importante que han de satisfacer los métodos de clasificación de arreglos es usar con medida el almacenamiento disponible. Ello significa que la permutación de elementos a fin de ponerlos en orden debe efectuarse *in situ* y que los métodos con que se transportan elementos de un arreglo *a* a un arreglo resultado *b* son de menor interés intrínseco. Una vez limitada así nuestra elección de métodos entre muchas soluciones posibles por el criterio de economía de almacenamiento, procedemos a hacer una primera clasificación basada en su eficiencia, esto es, en su economía de tiempo. Una buena medida de la eficiencia se logra al contar los números *C* de comparaciones que se necesitan de llaves y *M* de movimientos (transposiciones) de elementos. Estos números están en función del número *n* de elementos por clasificar. Los buenos algoritmos de clasificación requieren en el orden de $n * \log n$ comparaciones, antes explicaremos las técnicas más sencillas y obvias, llamadas *métodos directos*, todas las cuales exigen en el orden de n^2 comparaciones de llaves. Hay tres buenas razones que justifican describir esos métodos antes de pasar a algoritmos más rápidos.

1. Los métodos directos son muy adecuados para dilucidar las características de los principales principios de la clasificación.
2. Sus programas son fáciles de entender y son cortos. Recuérdese que los programas también ocupan memoria.
3. Aunque los métodos refinados requieren menor número de operaciones, éstas suelen ser más complejas en sus detalles; por consiguiente, los métodos directos son más rápidos para *n* suficientemente pequeño, aunque no deben utilizarse para un *n* grande.

Los métodos de clasificación que ordenan los elementos *in situ* pueden dividirse en tres categorías principales según el método en que se basan:

1. Clasificación por inserción.
2. Clasificación por selección.
3. Clasificación por intercambio.

A continuación estudiaremos y compararemos esos tres principios. Los programas operan con la variable *a* cuyos componentes han de clasificarse *in situ* y se refieren a los tipos de dato (2.2) e *index*, definidos como

```
TYPE index = INTEGER;
VAR a: ARRAY[1 .. n] OF item
```

(2.3)

2.2.1. Clasificación por inserción directa

Este método lo usan mucho los jugadores de naipes. Los elementos (naipes) se dividen conceptualmente en una secuencia destino $a_1 \dots a_{i-1}$. En cada paso, comenzando con $i = 2$ y aumentando i en una unidad, el i -ésimo elemento de la secuencia fuente se toma y se transfiere a la secuencia destino insertándolo en el lugar correspondiente.

El proceso de clasificación por inserción aparece en un ejemplo de ocho números escogidos en forma aleatoria (véase la tabla 2.1). El algoritmo de inserción directa es

```
FOR i := 2 TO n DO
  x := a[i];
  insertar x en el sitio correspondiente en a1 ... ai
END
```

En el proceso de encontrar el lugar apropiado, es aconsejable alternar entre comparaciones y movimientos, es decir, dejar que *x* se saque del arreglo comparando *x* con el siguiente elemento *a*_j e insertando *x* o moviendo *a*_j hacia la derecha y avanzando luego hacia la izquierda, es decir, se saca *x* del arreglo a fin de reubicarla.

Notamos que hay dos condiciones distintas que pueden causar la terminación del proceso de sacar para reubicar:

1. Se encuentra un elemento *a*_j que tiene una llave menor que la de *x*.
2. El extremo izquierdo de la secuencia destino es alcanzado.

Este caso típico de repetición con dos condiciones de terminación presenta a nuestra atención la conocida técnica del centinela. Se aplica fácilmente a este caso poniendo un centinela *a*₀ con el valor de *x*. (Observe que esto exige ampliar el intervalo de *index* en la declaración de *a* a $0 \dots n$.) El algoritmo completo se formula en el programa 2.1.

```
PROCEDURE StraightInsertion;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; a[0] := x; j := i;
    WHILE x < a[j-1] DO a[j] := a[j-1]; j := j-1 END ;
    a[j] := x
  END
END StraightInsertion
```

Programa 2.1 Clasificación por inserción directa.

Llaves iniciales	44	55	12	42	94	18	06	67
<i>i=2</i>	44	55	12	42	94	18	06	67
<i>i=3</i>	12	44	55	42	94	18	06	67
<i>i=4</i>	12	42	44	55	94	18	06	67
<i>i=5</i>	12	42	44	55	94	18	06	67
<i>i=6</i>	12	18	42	44	55	94	06	67
<i>i=7</i>	06	12	18	42	44	55	94	67
<i>i=8</i>	06	12	18	42	44	55	67	94

Tabla 2.1 Un proceso muestra de la clasificación por inserción directa.

Análisis de la inserción directa. El C_1 de las comparaciones de llaves en la i -ésima extracción es a lo sumo $i-1$ y como mínimo 1; suponiendo que todas las permutaciones de n llaves sean igualmente probables ($i/2$ en promedio). El número M_j de movimientos (asignaciones de elementos) es $C_i + 2$ (incluido el centinela). Por tanto, los números totales de comparaciones y movimientos son:

$$\begin{aligned} C_{\min} &= n-1 & M_{\min} &= 3*(n-1) \\ C_{\text{prom}} &= (n^2 + n - 2)/4 & M_{\text{prom}} &= (n^2 + 9n - 10)/4 \\ C_{\max} &= (n^2 + n - 4)/4 & M_{\max} &= (n^2 + 3n - 4)/2 \end{aligned} \quad (2.4)$$

Los números mínimos ocurren si los elementos se hallan en orden al inicio; el peor caso se presenta cuando los elementos se hallan originalmente en orden inverso. En este sentido, la clasificación por inserción muestra un comportamiento verdaderamente natural. Es evidente que el algoritmo dado describe además un proceso de clasificación estable: respeta el orden de los elementos con llaves iguales.

El algoritmo de la inserción directa se mejora fácilmente al notar que la secuencia destino $a_j \dots a_{i-1}$, donde debe insertarse el nuevo elemento, ya está ordenada. Por eso puede ser empleado un método más rápido para determinar el punto de inserción. La elección obvia es una búsqueda binaria que prueba la secuencia destino en la mitad y continúa buscando hasta encontrar el punto de inserción. El algoritmo de clasificación modificado recibe el nombre de *inserción binaria* y se muestra en el programa 2.2.

```
PROCEDURE BinaryInsertion;
  VAR i, j, m, L, R: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; L := 1; R := i;
    WHILE L < R DO
      m := (L+R) DIV 2;
      IF a[m] <= x THEN L := m+1 ELSE R := m END;
    END;
    FOR j := i TO R+1 BY -1 DO a[j] := a[j-1] END;
    a[R] := x;
  END;
END BinaryInsertion
```

Programa 2.2 Clasificación por inserción binaria.

Análisis de la inserción binaria. La posición de la inserción se encuentra si $L = R$. En consecuencia, el intervalo de búsqueda debe ser, al final, de longitud 1; y ello supone dividir a la mitad el intervalo de longitud $i \log i$ veces. Por tanto,

$$C = \sum_{i=2}^n i \log i$$

Esta suma la aproximamos por la integral

$$\text{Integral}(1:n) \log x \, dx = n * (\log n - c) + c \quad (2.5)$$

donde $c = \log e = 1/\ln 2 = 1.44269\dots$. El número de comparaciones es esencialmente independiente del orden inicial de los elementos. No obstante, dado el carácter truncador de la división que interviene en la bisección del intervalo de búsqueda, el verdadero número de comparaciones que se necesitan con i elementos puede ser hasta 1 más de lo previsto. La naturaleza de este error es tal que las posiciones de la inserción en el extremo inferior están sobre el promedio localizado ligeramente más pronto que los que se hallan en el extremo superior, con lo cual se favorecen los casos en que los elementos están muy fuera de orden al inicio. En efecto, el número mínimo de comparaciones se necesita si los elementos están inicialmente en orden inverso y se necesita el número máximo cuando ya están en orden. En consecuencia, se trata de un caso de comportamiento no natural de un algoritmo de clasificación.

$$C \approx n * (\log n - \log e \pm 0.5)$$

Por desgracia, el mejoramiento conseguido utilizando un método de búsqueda binaria se aplica exclusivamente al número de comparaciones, pero no al de los movimientos requeridos. En efecto, como mover los elementos, es decir, las llaves y la información conexa, es en general más lento que comparar dos llaves, el mejoramiento no es en absoluto radical: el término importante M es todavía del orden de n^2 . Y en efecto, clasificar los arreglos ya ordenados tarda más tiempo que la inserción directa con la búsqueda secuencial.

El ejemplo anterior demuestra que un “mejoramiento obvio” a menudo produce consecuencias menos profundas de lo que estamos inclinados a estimar al inicio; también demuestra que, en ciertos casos (que sí ocurren) el “mejoramiento” en realidad resulta ser un deterioro. Después de todo, la clasificación por inserción no parece ser un método muy adecuado para las computadoras digitales: la inserción de un elemento con el cambio posterior de un renglón entero de elementos por una sola posición no es económica. Cabe esperar resultados más satisfactorios de un método en el cual los movimientos de elementos se realizan sólo sobre elementos individuales y en distancias más largas. Esta idea nos lleva a la clasificación por selección.

2.2.2. Clasificación por selección directa

Este método se basa en los siguientes principios:

1. Seleccionar el elemento que tenga la llave menor.
2. Intercambiarlo con el primer elemento a_1 .
3. Repetir después estas operaciones con los $n-1$ elementos restantes, luego con $n-2$ elementos hasta que no quede más que un elemento (el más grande).

Este método se muestra con las mismas ocho llaves que en la tabla 2.1.

El algoritmo se formula así:

```
FOR i := 1 TO n-1 DO
  asignar el índice del elemento más pequeño de  $a_i \dots a_n$  a  $k$ ;
  intercambiar  $a_i$  con  $a_k$ 
END
```

Llaves iniciales.	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Tabla 2.2 Un proceso muestra de la clasificación por selección directa.

Este método, denominado *selección directa*, es en cierto modo lo contrario de la inserción directa: ésta considera en cada paso sólo *un* elemento siguiente de la secuencia fuente y *todos* los elementos del arreglo destino para encontrar el punto de inserción; la selección directa considera *todos* los elementos del arreglo fuente para encontrar *el que* tenga la llave más pequeña que debe depositarse como el siguiente elemento de la secuencia destino. El programa completo de selección directa se muestra en el programa 2.3.

```
PROCEDURE StraightSelection;
  VAR i, j, k: index; x: item;
BEGIN
  FOR i := 1 TO n-1 DO
    k := i; x := a[i];
    FOR j := i+1 TO n DO
      IF a[j] < x THEN k := j; x := a[k] END
    END;
    a[k] := a[i]; a[i] := x
  END
END StraightSelection
```

Análisis de selección directa. Es evidente que el número C de comparaciones de llaves es independiente del orden inicial de llaves. En este sentido, podemos afirmar que el método se conduce en forma menos natural que la inserción directa. Obtenemos:

$$C = (n^2 - n)/2$$

El número M de movimientos es por los menos

$$M_{\min} = 3*(n-1) \quad (2.6)$$

en el caso de llaves iniciales ordenadas y a lo sumo

$$M_{\max} = n^2/4 + 3*(n-1)$$

cuando al inicio las llaves están en orden inverso. A fin de determinar M_{prom} hacemos el siguiente razonamiento: el algoritmo rastrea el arreglo, comparando cada elemento con el valor mínimo descubierto hasta ese momento y, si es más pequeño que el mínimo, hace una asignación. La probabilidad de que el segundo elemento sea menor que el primero es 1/2, y la de que el cuarto sea el más pequeño es 1/4 y así sucesivamente. En conse-

cuencia, el número total esperado de movimiento es H_{n-1} , donde H_n es el n-ésimo número armónico

$$H_n = 1 + 1/2 + 1/3 + \dots + 1/n \quad (2.7)$$

H_n puede expresarse como

$$H_n = \ln n + g + 1/2n - 1/12n^2 + \dots \quad (2.8)$$

donde $g = 0.577216\dots$ es la constante de Euler. Para n suficientemente grande, podemos ignorar los términos fraccionarios y, por lo mismo, aproximar el número promedio de asignaciones en el i-ésimo pase como

$$F_i = \ln i + g + 1$$

El número promedio de movimientos M_{prom} en la clasificación de selección es, pues, la suma de F_i con i desde 1 hasta n.

$$M_{\text{prom}} = n*(g+1) + (\sum_{i=1}^n F_i)$$

Aproximando ulteriormente la suma de los términos discretos mediante la integral

$$\int_{1:n} \ln x \, dx = x * (\ln x - 1) = n * \ln(n) - n + 1$$

obtenemos el valor aproximado

$$M_{\text{prom}} \approx n * (\ln(n) + g) \quad (2.9)$$

Podemos concluir que en general el algoritmo de selección directa ha de preferirse a la inserción directa, pese a que ésta es todavía un poco más rápida en los casos en que las llaves se clasifican o casi se clasifican al principio.

2.2.3. Clasificación por intercambio directo

La clasificación de un método de clasificación rara vez está enteramente bien definida. Pero los métodos antes descritos pueden considerarse también como clasificaciones por intercambio. En la presente sección explicaremos un método en que el intercambio de dos elementos constituye la característica principal del proceso. El algoritmo subsiguiente de intercambio directo se funda en el principio de comparar e intercambiar pares de elementos contiguos hasta clasificar todos los elementos.

Igual que en los métodos anteriores de selección directa, hacemos varios pases sobre el arreglo, y en cada recorrido desplazamos el elemento más pequeño del conjunto restante hacia el extremo final del arreglo. Si en un cambio advertimos que el arreglo está en posición vertical y no en una posición horizontal y si (con ayuda de un poco de imaginación) vemos que los elementos están como burbujas en un tanque de agua con pesos correspondientes a sus llaves, cada pase sobre el arreglo produce el ascenso de una burbuja hasta su nivel adecuado de peso (véase la tabla 2.3). El método se conoce generalmente con el nombre de *clasificación por burbuja*. Su forma más sencilla aparece en el programa 2.4.

i=1	2	3	4	5	6	7	8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Tabla 2.3 Una muestra de clasificación por el método de la burbuja.

```

PROCEDURE BubbleSort;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    FOR j := n TO i BY -1 DO
      IF a[j-1] > a[j] THEN
        x := a[j-1]; a[j-1] := a[j]; a[j] := x
      END
    END
  END
END BubbleSort

```

Programa 2.4 Clasificación por burbuja.

Este algoritmo admite fácilmente un poco de mejoramiento. El ejemplo de la tabla 2.3 muestra que por lo menos tres pases no afectan al orden de los elementos, pues éstos ya están clasificados. Una técnica obvia que permite perfeccionar este algoritmo consiste en recordar si durante un pase ha tenido lugar algún intercambio. Un último pase sin operaciones ulteriores de intercambio será, pues, indispensable para determinar si el algoritmo puede haber terminado. No obstante, este mejoramiento es perfectible si recordamos no sólo el hecho de que hubo un intercambio, sino además la posición (índice) del último intercambio. Por ejemplo, es evidente que todos los pares de elementos adyacentes situados debajo del índice k se encuentran en el orden deseado. Los rastreos posteriores terminarán por eso en este índice, en vez de tener que pasar al límite inferior previamente determinado i. El programador prudente advierte una asimetría característica: una burbuja mal colocada en el extremo pesado de un arreglo ya clasificado quedará en orden en un solo pase; en cambio, un elemento mal colocado en el extremo ligero se hundirá hacia su posición correcta únicamente un paso en cada recorrido. Por ejemplo, el arreglo

12 18 42 44 55 67 94 06

se ordena mediante la clasificación por burbuja en un solo pase, pero el arreglo

94 06 12 18 42 44 55 67

requiere siete pases. Esta asimetría no natural sugiere un tercer mejoramiento: alternar la dirección de los pases consecutivos. *Clasificación por vibración* es el nombre con que adecuadamente conocemos el algoritmo resultante. Su comportamiento se muestra en la tabla 2.4, aplicándolo a las mismas ocho llaves que se emplearon en la tabla 2.3.

```

PROCEDURE ShakerSort;
  VAR j, k, L, R: index; x: item;
BEGIN L := 2; R := n; k := n;
REPEAT
  FOR j := R TO L BY -1 DO
    IF a[j-1] > a[j] THEN
      x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
    END
  END ;
  L := k+1;
  FOR j := L TO R BY +1 DO
    IF a[j-1] > a[j] THEN
      x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
    END
  END ;
  R := k-1
UNTIL L > R .
END ShakerSort

```

Programa 2.5 Clasificación por vibración.

Análisis de clasificación por burbuja y clasificación por vibración. El número de comparaciones en el algoritmo de intercambio directo es

$$C = (n^2 - n)/2 \quad (2.10)$$

L=	2	3	3	4	4
R=	8	8	7	7	4
dir= ↑	↓	↑	↓	↑	
	44	06	06	06	06
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	06	18	67	67	67
	67	67	94	94	94

Tabla 2.4 Un ejemplo de clasificación por vibración.

y los números mínimo, promedio y máximo de movimientos (asignaciones de elementos) son

$$M_{\min} = 0, \quad M_{\text{prom}} = 3*(n^2 - n)/2, \quad M_{\max} = 3*(n^2 - n)/4 \quad (2.11)$$

El análisis de los métodos mejorados, en especial el de clasificación por vibración es intrincado. El número mínimo de comparaciones es $C_{\min} = n-1$. En el caso del mejoramiento de la clasificación por burbuja, Knuth llega a un número promedio de pasos proporcional a $n-k_1n^{1/2}$ y a un número promedio de comparaciones proporcional a $1/2(n^2 - n(k_2 + \ln n))$. Pero notamos que todos los mejoramientos antes mencionados no afectan de ninguna manera al número de intercambios; tan sólo reducen el número de comparaciones dobles redundantes. Por desgracia, un intercambio de dos elementos suele ser una operación más costosa que una comparación de llaves; de ahí que nuestros mejoramientos más ingeniosos ejercen un efecto mucho menos profundo de lo que cabría suponer intuitivamente.

El análisis precedente muestra que la clasificación por intercambio y sus mejoramientos menores son inferiores a la clasificación por inserción y por selección; en efecto, la clasificación por burbuja no tiene otro aspecto en su favor que no sea el nombre tan original. El algoritmo de clasificación por vibración se utiliza con mejores resultados en los casos en que se sabe que los elementos ya están casi en orden: un caso poco frecuente en la práctica.

Puede demostrarse que la distancia promedio que cada uno de los n elementos debe recorrer durante una clasificación es $n/3$ lugares. Esta cifra proporciona una señal en la búsqueda de métodos de clasificación más refinados, es decir, más eficaces. Todos los de clasificación directa mueven esencialmente cada elemento una posición en cada paso elemental. Por consiguiente, es necesario que requieran en el orden n^2 de tales pasos. Cualquier perfeccionamiento ha de fundarse en el principio de mover los elementos sobre grandes distancias en saltos individuales.

Más adelante explicaremos tres métodos mejorados, a saber uno para cada método básico de clasificación: inserción, selección e intercambio.

2.3. METODOS DE CLASIFICACION AVANZADOS

2.3.1. Inserción por incremento decreciente

Un refinamiento de la inserción directa fue propuesto por D.L. Shell en 1959. El método se explica y demuestra en nuestro ejemplo estándar de ocho elementos (véase la tabla 2.5). Primero, todos los elementos que estén cuatro posiciones aparte se agrupan y se clasifican por separado. A este proceso se le llama *clasificación-4*. En este ejemplo de ocho elementos, cada grupo contiene exactamente dos elementos. Luego del primer paso, los elementos se reagrupan en grupos con los elementos colocados dos posiciones aparte y luego se clasifican otra vez. Este proceso recibe el nombre de *clasificación-2*. Por último en un tercer paso todos los elementos se clasifican en una clasificación ordinaria o *clasificación-1*.

Al inicio nos preguntaremos si la necesidad de realizar varios pasos, cada uno de los cuales incluye todos los elementos, no ocasiona más trabajo del que ahorra. Con todo, cada paso de clasificación sobre una cadena incluye relativamente pocos elementos o bien éstos ya están muy bien ordenados y se requieren pocos reacomodos.

Está claro que con el método se logra un arreglo ordenado, y es bastante evidente que cada paso se sirve de los anteriores (pues cada *clasificación-i* combina dos grupos ordenados en la *clasificación-2i* precedente). También es evidente que cualquier secuencia de incrementos resulta aceptable mientras el último sea la unidad, ya que en el peor caso el último paso hará todo el trabajo. Sin embargo, no es tan evidente que el método de incrementos decrecientes aporte resultados aun mejores con aumentos que no sean potencias de 2.

El programa se desarrolla, pues, sin recurrir a una secuencia específica de incrementos. Estos se denotan por

$$h_1, h_2, \dots, h_t$$

con las condiciones

$$h_t = 1, \quad h_{t+1} < h_t \quad (2.12)$$

44 55 12 42 94 18 06 67

Produce clasificación-4

44 18 06 42 94 55 12 67

Produce clasificación-2

06 18 12 42 44 55 94 67

Produce clasificación-1

06 12 18 42 44 55 67 94

Tabla 2.5 Una clasificación por inserción con incrementos decrecientes.

Cada *clasificación h* está programada como una clasificación por inserción directa utilizando la técnica del centinela para proporcionar una condición sencilla de terminación en la búsqueda del sitio de inserción. Es obvio que cada clasificación necesita colocar su propio centinela y que habrá que hacer lo más sencillo posible el programa para que determine su posición. Así pues, el arreglo a debe ser extendido no sólo en un solo componente a_0 sino también en los componentes h_i en forma tal que ahora se declara como

a: ARRAY [- h_1 .. n] OF item

El algoritmo se describe con el procedimiento llamado *clasificación de Shell* [2.11] en el programa 2.6 para $t = 4$.

```
PROCEDURE ShellSort;
CONST t = 4;
VAR i, j, k, s: index;
x: item; m: 1 .. t;
h: ARRAY [1 .. t] OF INTEGER;
BEGIN h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
FOR m := 1 TO t DO
  k := h[m]; s := -k; (* posición centinela *)
  FOR i := k+1 TO n DO
    x := a[i]; j := i-k;
    IF s = 0 THEN s := -k END;
    s := s + 1; a[s] := x; (*poner centinela*)
    WHILE x < a[j] DO a[j+k] := a[j]; j := j-k END;
    a[j+k] := x
  END
END
END ShellSort
```

Programa 2.6 Clasificación de Shell.

Análisis de la clasificación de Shell. El análisis de este algoritmo plantea algunos problemas matemáticos de gran dificultad, muchos de los cuales todavía no han sido resueltos. En particular, no se sabe cuál elección de incrementos produce los mejores resultados. Con todo, he aquí un hecho sorprendente: no deben ser múltiplos de sí mismos. Con esto se evita el fenómeno evidente en el ejemplo anterior, en el cual cada pase combina dos cadenas que antes no tenían interacción alguna. En efecto, conviene que la interacción entre varias cadenas tenga lugar lo más a menudo posible; se cumple el siguiente teorema: si una secuencia clasificada con k se clasifica con i , debe permanecer clasificada con k . Knuth [2.8] aporta evidencia de que una elección razonable de incrementos es la secuencia (escrita en orden inverso)

1, 4, 13, 40, 121, ...

donde $h_{k-1} = 3h_k + 1$, $h_t = 1$ y $t = \lceil \log_3 n \rceil - 1$. También recomienda la secuencia

1, 3, 7, 15, 31, ...

donde $h_{k-1} = 3h_k + 1$, $h_t = 1$ y $t = \lceil \log_2 n \rceil - 1$. En la segunda opción, el análisis matemático produce un esfuerzo proporcional a $n^{1.2}$ necesario para clasificar n elementos con el algoritmo de clasificación de Shell. Aunque es un mejoramiento importante respecto a n^2 , no nos detendremos más en este método, puesto que se conocen otros aún mejores.

2.3.2. Clasificación por árbol.

El método de clasificación por selección directa se basa en selección repetida de la llave más pequeña entre n elementos, luego entre los restantes $n-1$ elementos. Desde luego, encontrar esa llave entre n elementos exige $n-1$ comparaciones, encontrarla entre $n-1$ elementos requiere $n-2$ comparaciones, etc.; y la suma de los primeros $n-1$ enteros es $\frac{1}{2}(n^2 - n)$. Cabe preguntarse entonces: ¿cómo es posible mejorar esta clasificación por selección? Podemos hacerlo sólo conservando en cada rastreo más información que la mera identificación del elemento menor. Por ejemplo, con $n/2$ comparaciones podemos determinar la llave más pequeña de cada par de elementos, con otras $n/4$ comparaciones podemos escoger la más pequeña de cada par de esas llaves y así sucesivamente. Con sólo $n-1$ comparaciones, podemos construir un árbol de selección como el de la figura 2.3 e identificar la raíz como la llave deseada más pequeña [2.2].

Y ahora el segundo paso consiste en descender por la trayectoria marcada con la llave menor y eliminarla, reemplazándola sucesivamente con un hoyo vacío en la parte inferior o con el elemento situado en la rama alterna en los nodos intermedios (véase las figuras 2.4 y 2.5). También aquí, el elemento que brota en la raíz del árbol tiene la (ahora segunda) llave más pequeña y puede suprimirse. Luego de n pasos de esta selección, el árbol queda vacío (es decir, lleno de hoyos) y concluye el proceso de clasificación. Conviene señalar que cada uno de los n pasos de selección requiere sólo $\log n$ comparaciones. Por consiguiente, el proceso total exige sólo en el orden de $n * \log n$ operaciones elementales además de los n pasos que se necesitan en la construcción del árbol. Esto constituye un mejoramiento muy significativo respecto a los métodos directos que exigen n^2 pasos y respecto a la clasificación de Shell que requiere $n^{1.2}$ pasos. Por supuesto, la tarea de administración de datos se ha vuelto más intrincada y, en consecuencia, la complejidad de los pasos individuales es mayor en el método de clasificación por árbol; después de todo, a fin de conservar la mayor cantidad de información obtenida con el paso inicial, hay

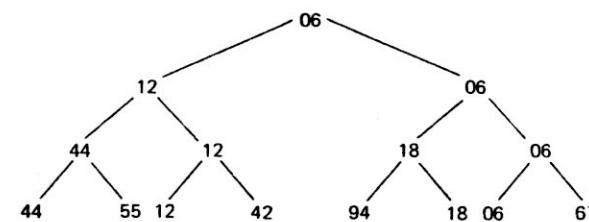


Fig. 2.3 Selección repetida entre dos llaves.

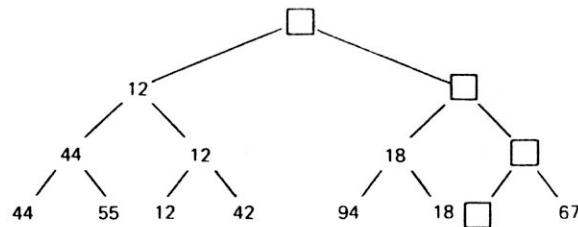


Fig. 2.4 Selección de la llave más pequeña.

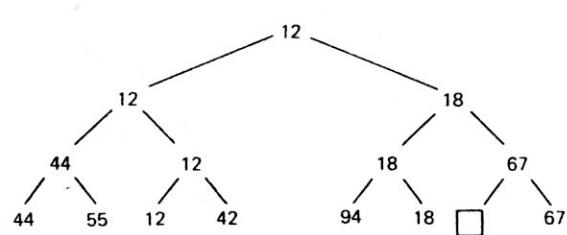


Fig. 2.5 Relleno de los hoyos.

que crear alguna especie de estructura de árbol. Nuestra siguiente tarea consiste en encontrar métodos para organizar de modo eficiente esa información.

Desde luego parece muy conveniente poder prescindir de los hoyos que a la poste pueblan el árbol entero y que son causa de muchas comparaciones superfljas. Más aún, hay que encontrar la manera de representar el árbol de n elementos en n unidades de almacenamiento, en vez de hacerlo en $2n-1$ unidades como se mostró antes. Y estas metas se logran con un método que su inventor J. Williams [2.14] llamó *clasificación por montón*. Salta a la vista que este método representa un notable mejoramiento sobre los métodos más comunes de la clasificación por árbol. Un *montón* se define como una secuencia de llaves h_L, h_{L+1}, \dots, h_R tales que

$$h_i \leq h_{2i} \text{ y } h_i \leq h_{2i+1} \text{ para } L \dots R/2. \quad (2.13)$$

Si un árbol binario se representa como el arreglo que aparece en la figura 2.6, se concluye que los árboles de clasificación de las figuras 2.7 y 2.8 son montones y, en particular, que el elemento h_1 de un montón es su elemento más pequeño:

$$h_1 = \min(h_1, h_2, \dots, h_n)$$

Supongamos que un montón con los elementos $h_{L+1} \dots h_R$ se da para algunos valores L y R y que un nuevo elemento x debe sumarse para formar el montón ampliado $h_L \dots h_R$. Tomemos por ejemplo, el paso inicial $h_1 \dots h_7$ mostrado en la figura 2.7 y extendamos el montón a la izquierda en un elemento $h_1 = 44$. Se obtiene un nuevo montón poniendo

primero x sobre la cima de la estructura del árbol y luego dejando que se desplace hacia abajo a lo largo de la trayectoria de los comparandos menores, que al mismo tiempo suben. En el ejemplo, el valor 44 se intercambia primero con 06, luego con 12 y así forma el árbol mostrado en la figura 2.8. Ahora formulamos del modo siguiente este algoritmo de desplazamiento: i, j son el par de índices que denotan los elementos que es preciso intercambiar en cada paso del desplazamiento. Recomendamos encarecidamente al lector convencirse de que el método propuesto de desplazamiento en realidad preserva las condiciones (2.13) que definen el montón.

Una manera muy adecuada de construir un montón *in situ* la ha propuesto R.W. Floyd. En ella se aplica el procedimiento de desplazamiento que viene en el programa 2.7. Se da un arreglo $h_1 \dots h_n$; sin duda los elementos $h_m \dots h_n$ (con $m = (n \text{ DIV } 2) + 1$) forman ya un montón, pues no hay dos índices i, j tales que $j = 2i$ (*o sea* $j = 2i + 1$).

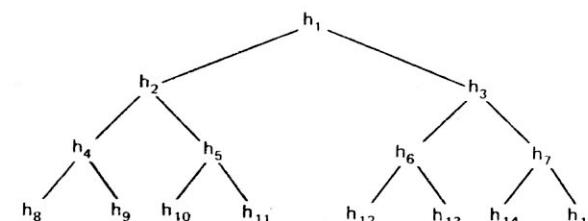
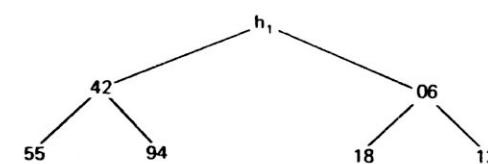
Fig. 2.6 Arreglo h visto como un árbol binario.

Fig. 2.7 Montón con siete elementos.

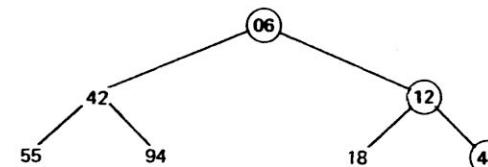


Fig. 2.8 Desplazamiento de la llave 44 a través del montón.

Estos elementos constituyen lo que puede considerarse el renglón de la parte inferior del árbol binario asociado (véase la figura 2.6), entre los cuales no se requiere ninguna relación de ordenación. El montón se amplía ahora a la izquierda, con lo cual en cada paso se incluye un nuevo elemento y se posiciona correctamente con un desplazamiento. Este proceso se muestra en la tabla 2.6 y produce el montón que aparece en la figura 2.6.

```

PROCEDURE sift(L, R: index);
  VAR i, j: index; x: item;
BEGIN i := L; j := 2*L; x := a[L];
  IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END ;
  WHILE (j <= R) & (a[j] < x) DO
    a[i] := a[j]; i := j; j := 2*j;
    IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END
  END
END sift

```

Programa 2.7 Desplazamiento.

En consecuencia, el proceso de generar un montón de n elementos $h_1 \dots h_n$ *in situ* se describe así:

```

L := (n DIV 2) + 1;
WHILE L > 1 DO L := L-1; sift(L, n) END

```

Con objeto de obtener no sólo una ordenación parcial sino completa entre los elementos, han de realizarse n pasos de desplazamiento; de ese modo después de cada paso el siguiente elemento (más pequeño) puede quitarse de la cima del montón. Una vez más, se plantea la cuestión sobre el sitio donde guardar los elementos que emergen en la cima y si será posible o no la clasificación *in situ*. Desde luego hay esta solución: en cada paso se quita del montón el último componente (digamos x), se guarda el elemento de la cima del montón en la localización ahora libre de x y se deja que x se desplace hacia abajo hasta alcanzar su posición correspondiente. En el montón de la tabla 2.7 se ejemplifican los pasos $n-1$ necesarios. El proceso se describe con ayuda del procedimiento *Sift* (programa 2.7) como sigue:

```

R := n;
WHILE R > 1 DO
  x := a[1]; a[1] := a[R]; a[R] := x;
  R := R-1; sift(1, R)
END

```

44	55	12	42		94	18	06	67
44	55	12		42	94	18	06	67
44	55		06	42	94	18	12	67
44		42	06	55	94	18	12	67
06	42	12	55	94	18	44	67	

Tabla 2.6 Construcción de un montón

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

Tabla 2.7 Ejemplo de un proceso de clasificación por montón.

El ejemplo de la tabla 2.7 muestra que el orden resultante está en realidad invertido. No obstante, eso puede remediarlo fácilmente cambiando la dirección de las relaciones de ordenación en el procedimiento *Sift*. Ello nos da el procedimiento *Heapsort* del programa 2.8.

```

PROCEDURE HeapSort;
  VAR L, R: index; x: item;

PROCEDURE sift(L, R: index);
  VAR i, j: index; x: item;
BEGIN i := L; j := 2*L; x := a[L];
  IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END ;
  WHILE (j <= R) & (x < a[j]) DO
    a[i] := a[j]; i := j; j := 2*j;
    IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END
  END
END sift;

BEGIN L := (n DIV 2) + 1; R := n;
  WHILE L > 1 DO L := L-1; sift(L, R) END ;
  WHILE R > 1 DO
    x := a[1]; a[1] := a[R]; a[R] := x;
    R := R-1; sift(L, R)
  END
END HeapSort

```

Programa 2.8 Clasificación por montón

Análisis de clasificación por montón. A primera vista no es evidente que este método de clasificación aporte buenos resultados. Después de todo, los elementos grandes primero son desplazados hacia la izquierda antes de depositarlos finalmente en el extremo derecho. En efecto, el procedimiento no se recomienda en el caso de números pequeños de elementos, como se aprecia en el ejemplo. Sin embargo, este método es sumamente eficiente para n grande, y su eficiencia aumenta al crecer n (incluso en comparación con la clasificación de Shell).

En el peor caso, hay $n/2$ pasos de desplazamiento necesarios, pasando los elementos por $\log(n/2)$, $\log(n/2-1)$, ..., $\log(n-1)$ posiciones, donde el logaritmo (de base 2) queda

truncado en el siguiente entero más bajo. Después, la fase de clasificación hace $n-1$ desplazamientos, con un máximo de $\log(n-1)$, $\log(n-2)$, ..., 1 movimientos. Además, hay $n-1$ movimientos para guardar a la derecha el elemento desplazado. Este argumento muestra que la clasificación por montón toma del orden de $n \cdot \log n$ pasos aun en el peor caso posible. Este excelente rendimiento en el peor caso constituye una de las cualidades más notables de la clasificación por montón.

No está muy claro en cuál caso el peor (o el mejor) rendimiento debe esperarse. Pero en general la clasificación por montón parece gustar de las secuencias iniciales donde los elementos están más o menos clasificados en orden inverso; por tanto, muestra entonces un comportamiento no natural. La fase de creación del montón no requiere movimiento alguno si está presente el orden inverso. El número promedio de movimientos es aproximadamente $n/2 * \log(n)$ y las desviaciones respecto a este valor son relativamente pequeñas.

2.3.3. Clasificación por partición

Luego de haber explicado los dos métodos avanzados de clasificación que se basan en los principios de inserción y clasificación, hablaremos ahora de un tercer método mejorado que se funda en el principio de intercambio. Dado que la clasificación por burbuja fue, en general, el menos eficaz de los tres algoritmos de clasificación directa, cabe esperar un perfeccionamiento relativamente importante. Con todo, nos sorprende el hecho de que el mejoramiento basado en los intercambios, que veremos luego, produzca el método óptimo de clasificación en los arreglos conocidos hasta ahora. Su rendimiento es tan impresionante que su inventor, C.A.R. Hoare, lo llamó *clasificación rápida* [2.5 y 2.6].

Este método se basa en el hecho de que los intercambios han de efectuarse, de preferencia, en distancias largas para que logren su máxima eficiencia. Supongamos que n elementos se dan en orden inverso de sus llaves. Es posible clasificarlos realizando sólo $n/2$ intercambios, tomando primero el extremo izquierdo y el extremo derecho, avanzando luego poco a poco hacia adentro a partir de ambos lados. Por supuesto, es posible lo anterior sólo si sabemos que su orden es exactamente inverso. Pero este ejemplo nos proporciona enseñanzas sin duda.

Probemos el siguiente algoritmo. Selecciónese un elemento cualquiera al azar (y llamémoslo x); se rastrea el arreglo a partir de la izquierda hasta encontrar un elemento $a_i > x$ y después se rastrea desde la derecha hasta hallar un elemento $a_j < x$. A continuación se intercambian los dos elementos y se prosigue este proceso de *rastreo e intercambio* hasta que los dos rastreos se encuentren en alguna parte de la mitad del arreglo. El resultado es que éste está ahora dividido en una parte izquierda con llaves menores (o iguales) que x y en una parte derecha con llaves mayores (o iguales) que x . Este proceso de partición se enumera mediante un procedimiento en el programa 2.9. Nótese que las relaciones $>$ y $<$ han sido reemplazadas por \geq y \leq , cuyas negaciones en la cláusula while son $<$ y $>$. Con este cambio, x sirve de centinela para ambos rastreos.

```
PROCEDURE partition;
  VAR w, x: item;
BEGIN i := 1; j := n;
```

```
seleccionar un elemento x al azar;
REPEAT
  WHILE a[i] < x DO i := i+1 END ;
  WHILE x < a[j] DO j := j-1 END ;
  IF i <= j THEN
    w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
  END
UNTIL i > j
END partition
```

Programa 2.9 Partición.

He aquí un ejemplo: si la llave 42 de la mitad se selecciona como comparando de x , entonces el arreglo de llaves

44 55 12 42 94 06 18 67

requiere los dos intercambios $18 \leftrightarrow 44$ y $6 \leftrightarrow 55$ para producir el arreglo dividido

18 06 12 42 94 55 44 67

y los valores finales índice $i = 5$ y $j = 3$. Las llaves $a_1 \dots a_{i-1}$ son menores o iguales que la llave $x = 42$ y las llaves $a_{j+1} \dots a_n$ son mayores o iguales que la llave x . En consecuencia, hay tres particiones, a saber:

$$\begin{aligned} &Ak : 1 \leq k < i : a_k \leq x \\ &Ak : j < k \leq n : x \leq a_k \end{aligned} \quad (2.14)$$

Este algoritmo es muy sencillo y eficiente porque los comparandos esenciales i , j y x pueden conservarse en registros rápidos durante todo el rastreo. Sin embargo, también puede crear problemas, como lo atestigua el caso con n llaves idénticas que produce $n/2$ intercambios. Estos intercambios innecesarios pueden eliminarse fácilmente modificando las proposiciones de rastreo a

```
WHILE a[i] <= x DO i := i+1 END ;
WHILE x <= a[j] DO j := j-1 END
```

Sin embargo, en este caso el elemento de elección x , que está presente como miembro del arreglo, ya no funge como centinela en los dos rastreos. El arreglo con todas las llaves idénticas hará que los rastreos vayan más allá de los límites del arreglo a menos que se recurra a condiciones más complejas de terminación. La simplicidad de las que se utilizan en el programa 2.9 justifica los intercambios adicionales que ocurren con relativa poca frecuencia en el caso aleatorio promedio. No obstante, puede lograrse un pequeño ahorro al cambiar la cláusula que controla el paso de intercambio a $i < j$ en vez de $i \geq j$. Pero este cambio no debe extenderse sobre las dos proposiciones

$i := i+1; j := j-1$

que, por tanto, requiere una cláusula condicional aparte. Para tener confianza en la corrección del algoritmo de partición se verifican que las relaciones (2.14) sean invariantes de la proposición repeat. Al inicio, con $i = 1$ y $j = n$, son trivialmente verdaderas y después de la salida con $i > j$ implican el resultado deseado.

Recordamos ahora que nuestra meta no es sólo encontrar particiones del arreglo original de elementos, sino también clasificarlo. Sin embargo, hay sólo un pequeño paso entre la partición y la clasificación: luego de dividir el arreglo, se aplica el mismo proceso a ambas particiones, después a las particiones de las particiones y así sucesivamente hasta que cada partición consta de un solo elemento exclusivamente. Este procedimiento se describe en el programa 2.10.

```

PROCEDURE QuickSort;
  PROCEDURE sort(L, R: index);
    VAR i, j: index; w, x: item;
    BEGIN i := L; j := R;
      x := a[(L+R) DIV 2];
      REPEAT
        WHILE a[i] < x DO i := i+1 END ;
        WHILE x < a[j] DO j := j-1 END ;
        IF i <= j THEN
          w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
        END
      UNTIL i > j;
      IF L < j THEN sort(L, j) END ;
      IF i < R THEN sort(i, R) END
    END sort;

    BEGIN sort(1, n)
  END QuickSort

```

Programa 2.10 Clasificación rápida.

El procedimiento *sort* se activa a sí mismo recursivamente. Tal empleo de la recursión en algoritmos es una herramienta muy poderosa y se explicará más a fondo en el capítulo 3. En algunos lenguajes de programación de origen antiguo, la recursión no se usaba por razones técnicas. A continuación mostraremos cómo este mismo algoritmo puede expresarse como un procedimiento no recursivo, con lo cual se vuelven necesarias algunas operaciones de administración de datos adicionales.

La clave de una solución iterativa consiste en mantener una lista de solicitudes de partición que todavía deben realizarse. Después de cada paso surgen dos tareas de partición. Únicamente una de ellas puede ser emprendida por la siguiente iteración; la otra se guarda en la lista. Desde luego, resulta indispensable que la lista de peticiones se cumpla en una secuencia específica, es decir, en secuencia inversa. Esto supone que la primera petición es la que se cumple por último y a la inversa; la lista se trata como una pila pulsante.

En la siguiente versión no recursiva de la clasificación rápida, cada petición se representa simplemente por un índice izquierdo y otro derecho que especifican los límites de la partición que debe dividirse ulteriormente. Por tanto, introducimos una variable arreglo denominada *stack* y un índice *s* que designa su entrada más reciente (véase el programa 2.11). La elección apropiada del tamaño de la pila, *M*, se explicará al analizar la clasificación rápida.

```

PROCEDURE NonRecursiveQuickSort;
  CONST M = 12;
  VAR i, j, L, R: index; x, w: item;
  s: [0 .. M];
  stack: ARRAY [1 .. M] OF RECORD L, R: index END ;
  BEGIN s := 1; stack[1].L := 1; stack[s].R := n;
  REPEAT (*tomar petición de la cima de la pila*)
    L := stack[s].L; R := stack[s].R; s := s-1;
    REPEAT (*particionar a[L] ... a[R]*)
      i := L; j := R; x := a[(L+R) DIV 2];
      REPEAT
        WHILE a[i] < x DO i := i+1 END ;
        WHILE x < a[j] DO j := j-1 END ;
        IF i <= j THEN
          w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
        END
      UNTIL i > j;
      IF i < R THEN (*apilar petición para clasificar partición de la derecha*)
        s := s+1; stack[s].L := i; stack[s].R := R
      END ;
      R := j (*ahora L y R delimitan la partición de la izquierda*)
    UNTIL L >= R
  UNTIL s = 0
END NonRecursiveQuickSort

```

Programa 2.11 Fusión no recursiva de la clasificación rápida.

Análisis de la clasificación rápida. A fin de analizar el rendimiento de la clasificación rápida, antes necesitamos investigar el comportamiento del proceso de partición. Tras haber seleccionado un límite *x*, recorre todo el arreglo. De ahí que se efectúen exactamente *n* comparaciones. El número de intercambios puede determinarse con el siguiente argumento probabilístico.

Con un límite fijo *x*, el número previsto de operaciones de intercambio es igual al de elementos en la parte izquierda de la partición, o sea $n-1$ multiplicado por la probabilidad de que ese elemento haya alcanzado su lugar mediante un intercambio. Habrá tenido lugar un intercambio si el elemento ha sido antes parte de la partición de la derecha; la probabilidad de ello es $(n-(x-1))/n$. El número esperado de intercambios es, pues, el promedio de esos valores previstos sobre todos los límites posibles *x*.

$$\begin{aligned}
 M &= [Sx: 1 \leq x \leq n : (x-1)*(n-(x-1))/n]/n \\
 &= [Su: 0 \leq u \leq n-1 : u*(n-u)]/n^2 \\
 &= n*(n-1)/2n - (2n^2 - 3n + 1)/6n = (n - 1/n)/6
 \end{aligned} \tag{2.15}$$

Suponiendo que seamos muy afortunados y que siempre escogamos como límite la mediana, cada proceso de partición divide el arreglo en dos mitades y el número de pasos necesarios será $\log n$. El número total resultante de comparaciones será entonces $n \cdot \log n$ y el de intercambios, $n \cdot \log(n)/6$.

Desde luego, no podemos esperar obtener la mediana todas las veces. En efecto, la probabilidad de lograrlo es apenas $1/n$. Sin embargo, es interesante señalar que el rendimiento promedio de la clasificación rápida es inferior al caso óptimo por un factor de apenas $2 \cdot \ln(2)$, si se escoge al azar el límite.

Pero la clasificación rápida tiene sus limitaciones. Ante todo, da un rendimiento moderado en el caso de valores pequeños de n , igual que todos los métodos avanzados. Su ventaja sobre ellos consiste en la facilidad con que un método de clasificación directa puede incorporarse para manejar las particiones pequeñas. Ello es de gran utilidad cuando consideramos la versión recursiva del programa.

Queda todavía la cuestión del caso peor. ¿Cuál es el rendimiento de la clasificación rápida entonces? Por desgracia la respuesta es desalentadora y revela una de las debilidades de este procedimiento. Tomemos, por ejemplo, el desafortunado caso en el cual siempre se escoge casualmente como comparando x el valor más grande de una partición. Después cada paso divide un segmento de n elementos en una partición izquierda con $n-1$ elementos y una partición derecha con un solo elemento. El resultado es que n (en vez de $\log n$) divisiones se tornan necesarias y que el rendimiento en el peor caso es del orden n^2 .

De lo anterior se deduce que el paso decisivo lo constituye la selección del comparando x . En nuestro programa ejemplo, se escoge como el elemento de la mitad. Nótese que casi podríamos seleccionar el primer elemento o el último. En tales circunstancias, el caso peor es el arreglo clasificado al inicio; la clasificación rápida muestra entonces una abierta aversión por la tarea trivial y preferencia por los arreglos desordenados. Al seleccionar el elemento de la mitad, la característica extraña del procedimiento resulta menos notoria dado que el arreglo clasificado inicialmente se convierte en el caso óptimo. De hecho, también el rendimiento promedio es ligeramente mejor, si el elemento de la mitad se escoge. Hoare recomienda realizar en forma aleatoria la elección de x o escogerlo como la media de una pequeña muestra de, digamos, tres llaves [2.12 y 2.13]. Esa elección tan sensata difícilmente influirá en el rendimiento promedio de la clasificación rápida, pero mejora mucho el del peor caso. Resulta evidente que la clasificación basada en este procedimiento se parece a un juego de azar en el cual hay que saber cuánto está uno dispuesto a perder si tiene mala suerte y no gana.

Esta experiencia nos da una gran enseñanza, la cual concierne directamente al programador. ¿Cuáles son las consecuencias que el comportamiento en el peor caso, mencionado antes, tiene en el rendimiento del programa 2.11? Hemos visto que cada división da origen a una partición derecha de un solo elemento; la petición de clasificar esa partición

se guarda para ejecución posterior. En consecuencia, el máximo número de peticiones, y por lo mismo el tamaño total requerido de pilas, es n . Y ello es, por supuesto, absolutamente inaceptable. (Nótese que no esperamos algo mejor [por el contrario, esperamos algo peor] con la versión recursiva pues un sistema que permite la activación recursiva de los procedimientos deberá almacenar automáticamente los valores de las variables y parámetros locales de todas las activaciones de procedimientos y se servirá de una pila implícita para ello.)

El remedio consiste en apilar la petición de clasificación para la partición más larga y en iniciar de inmediato la partición ulterior de la sección más pequeña. En este caso, el tamaño de la pila M puede limitarse a $\log n$.

El cambio necesario para el programa 2.11 se encuentra en la sección que impone nuevas peticiones. Y ahora el programa es éste:

```

IF j - L < R - i THEN
  IF i < R THEN (*apilar petición para clasificar partición de la derecha*)
    s := s+1; stack[s].L := i; stack[s].R := R
  END ;
  R := j (*continuar clasificando partición de la izquierda*)
ELSE
  IF L < j THEN (*apilar petición para clasificar partición de la derecha*)
    s := s+1; stack[s].L := L; stack[s].R := j
  END;
  L := i (*continuar clasificando partición de la izquierda*)
END .

```

(2.16)

2.3.4. Obtención de la mediana

La *mediana* de n elementos se define como el elemento que es menor (o igual) que la mitad de n elementos y que es mayor (o igual que) la otra mitad. Por ejemplo la mediana de

16 12 99 95 18 87 10

es 18. El problema de encontrar la mediana suele estar ligado al de la clasificación puesto que el método obvio de determinarla consiste en clasificar n elementos y luego escoger el de la mitad. Pero la partición mediante el programa 2.9 produce una manera potencialmente mucho más rápida de obtener la mediana. El método que se mostrará se generaliza y aplica fácilmente al problema de calcular el k -ésimo más pequeño de n elementos y luego seleccionar el de la mitad. Encontrar la mediana constituye el caso especial $k = n/2$.

El algoritmo inventado por C.A.R. Hoare [2-4] funciona del modo siguiente. Primero, la operación de partición de la clasificación rápida se aplica con $L = 1$ y $R = n$ y con a_k seleccionado como el valor divisor x . Los valores índice resultantes i y j son tales que

1. $a_h < x$ para toda $h < i$
 2. $a_h > x$ para toda $h > j$
 3. $i > j$
- (2.17)

Pueden presentarse tres casos posibles:

1. El valor divisor x fue demasiado pequeño; en consecuencia, el límite entre las dos particiones está debajo del valor deseado k . El proceso de partición ha de repetirse con los elementos $a_i \dots a_R$ (véase la figura 2.9).
2. El límite escogido x fue demasiado grande. La operación divisoria ha de repetirse en la partición $a_L \dots a_j$ (véase la figura 2.10).
3. $j < k < i$: el elemento a_k divide el arreglo en dos particiones según las proporciones especificadas y , en consecuencia, es el cuartil deseado (véase la figura 2.11).

El proceso de división debe repetirse hasta que se presente el caso.

3. Esta iteración se expresa con el siguiente fragmento de programa:

```
L := 1; R := n;
WHILE L < R DO
    x := a[k]; particion (a[L] ... a[R]);
    IF j < k THEN L := i END ;
    IF k < i THEN R := j END
END
```

(2.18)

El lector que desee una prueba formal de la corrección de este algoritmo puede consultar el artículo original de Hoare. El programa *Find* completo se obtiene fácilmente de éste.

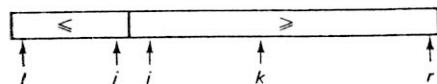


Fig. 2.9 Límite demasiado pequeño.

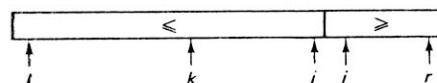


Fig. 2.10 Límite demasiado grande.

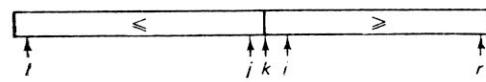


Fig. 2.11 Límite correcto.

```
PROCEDURE Find(k: INTEGER);
  VAR L, R, i, j: index; w, x: item;
BEGIN L := 1; R := n;
  WHILE L < R DO
    x := a[k]; i := L; j := R;
    REPEAT
      WHILE a[i] < x DO i := i+1 END ;
      WHILE x < a[j] DO j := j-1 END ;
      IF i <= j THEN
        w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
      END
    UNTIL i > j;
    IF j < k THEN L := i END ;
    IF k < i THEN R := j END
  END
END Find
```

Programa 2.12 Obtención de k -ésimo elemento más grande.

Si suponemos que, en promedio, cada división divide a la mitad el tamaño de la partición donde se encuentra el cuartil deseado, el número de comparaciones necesarias es

$$n + n/2 + n/4 + \dots + 1 \leq 2n$$
(2.19)

es decir, es del orden n . Esto explica el poder del programa *Find* para encontrar medianas y cuartiles similares, explicando además su superioridad sobre el método sencillo de clasificar el conjunto total de candidatos antes de seleccionar el k -ésimo (donde el mejor es del orden $n \cdot \log(n)$). En el peor caso, cada paso de la partición reduce el tamaño del conjunto de candidatos apenas en 1, lo cual da por resultado el número necesario de comparaciones de orden n^2 . También aquí apenas hay una ventaja pequeña al utilizar este algoritmo, si el número de elementos es reducido, digamos, menos de 10.

2.3.5. Una comparación de los métodos de clasificación de arreglos

Para concluir esta reseña de los métodos de clasificación, trataremos de comparar su eficacia. Si n denota el número de elementos por clasificar, C y M representarán otra vez el número de comparaciones requeridas de llaves y movimientos de elementos, respectivamente. Se pueden dar fórmulas analíticas cerradas para los tres métodos de clasificación directa. Se tabulan en la tabla 2.8. Los encabezados de columna Mín, Pro y Máx, especifican respectivamente los valores mínimos, máximos y promedio de todas las $n!$ permutaciones de n elementos.

No se dispone de fórmulas exactas razonablemente simples para los métodos avanzados. Los hechos esenciales son que el esfuerzo computacional necesario es $c \cdot n^{1.2}$ en el caso de la clasificación de Shell y $c \cdot n \cdot \log n$ en el caso de clasificación por montón y clasificación rápida, donde c son los coeficientes apropiados.

		Mín.	Prom.	Máx.
Inserción directa	C =	n-1	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	M =	$2(n-1)$	$(n^2 - 9n - 10)/4$	$(n^2 - 3n - 4)/2$
Selección directa	C =	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	M =	$3(n-1)$	$n*(\ln n + 0.57)$	$n^2/4 + 3(n-1)$
Intercambio directo	C =	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M =	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

Tabla 2.8 Comparación de métodos de clasificación directa:

Esas fórmulas se limitan a ofrecer una medida aproximada del rendimiento en función de n , permitiendo además la clasificación de los algoritmos de ordenación en métodos primitivos y directos (n^2) y los avanzados o "logarítmicos" ($n*\log(n)$). Sin embargo, en la práctica conviene disponer de algunos datos experimentales que arrojen luz sobre los coeficientes c , los cuales distinguen ulteriormente los métodos. Más aún, las fórmulas no tienen en cuenta el trabajo de computación dedicado a otras operaciones además de las comparaciones de llaves y los movimientos de elementos, como el control de ciclos. Es patente que tales factores dependen, en cierta medida, de los sistemas individuales, pero pese a ello es muy informativo un ejemplo de datos recabados con técnicas experimentales. La tabla 2.9 muestra el tiempo (en segundos) que tardan los métodos de clasificación expuestos antes, ejecutados por el sistema Modula-2 en una computadora personal Lith. Las tres columnas contienen el tiempo destinado a clasificar el arreglo ya ordenado, una permutación aleatoria y el arreglo ordenados inversamente. La figura de la izquierda en cada columna es para 256 elementos, las de la derecha para 512 elementos. Los datos separan con claridad los métodos n^2 y los métodos $n*\log(n)$. Conviene señalar los siguientes puntos:

1. El mejoramiento de la inserción binaria sobre la directa es secundario, e incluso negativo en el caso de un orden ya existente.
2. La clasificación por burbuja es sin duda el peor método entre todos los que se comparan. Su versión perfeccionada de clasificación por vibración es todavía peor que la inserción directa y la selección directa (salvo en el caso "patológico" de clasificar un arreglo ordenado).
3. La clasificación rápida supera a la clasificación por montón en un factor de 2 a 3. Clasifica el arreglo inversamente ordenado con una velocidad casi idéntica a la que tiene el que ya está ordenado.

	Ordenado	Aleatorio	Inverso
$n = 256$			
Inserción directa	0.02	0.82	1.64
Inserción binaria	0.12	0.70	1.30

Selección directa	0.94	0.96	1.18
Clasificación por burbuja	1.26	2.04	2.80
Clasificación por vibración	0.02	1.66	2.92
Clasificación de Shell	0.10	0.24	0.28
Clasificación por montón	0.20	0.20	0.20
Clasificación rápida	0.08	0.12	0.08
Clasificación rápida no recursiva	0.08	0.12	0.08
Mezcla directa	0.18	0.18	0.18
$n = 2048$			
Inserción directa	0.22	50.74	103.80
Inserción binaria	1.16	37.66	76.06
Selección directa	58.18	58.34	73.46
Clasificación por burbuja	80.18	128.84	178.66
Clasificación por vibración	0.16	104.44	187.36
Clasificación de Shell	0.80	7.08	12.34
Clasificación por montón	2.32	2.22	2.12
Clasificación rápida	0.72	1.22	0.76
Clasificación rápida no recursiva	0.72	1.32	0.80
Mezcla directa	1.98	2.06	1.98

Tabla 2.9 Tiempos de ejecución de los programas de clasificación.

2.4. SECUENCIAS DE CLASIFICACION

2.4.1. Mezcla (combinación) directa

Por desgracia, los algoritmos de clasificación presentados en el capítulo 1 son inaplicables si la cantidad de datos por ordenar no caben en la memoria principal de la computadora, pero se representa, por ejemplo, en un dispositivo de almacenamiento periférico y secuencial como una cinta o disco. En este caso describimos los datos como un archivo (secuencial) cuya característica es que en cada momento un componente, y solamente uno, es accesible directamente. Esta es una restricción severa si se compara con las posibilidades que ofrece la estructura de arreglo; por consiguiente, hay que aplicar otras técnicas de clasificación. La más importante es la de *mezcla*. Mezclar significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento. Mezclar es una operación mucho más sencilla que la de clasificar, y sirve como una operación auxiliar en el proceso más complejo de la clasificación secuencial. Una manera de clasificar con base en la mezcla, llamada *mezcla directa*, es la siguiente:

1. Dividir la secuencia a en dos mitades, denominadas b y c .
2. Mezclar b y c combinando cada elemento en pares ordenados.
3. Llamar a a la secuencia mezclada y repetir los pasos 1 y 2, esta vez combinando los pares ordenados en cuádruplos ordenados.
4. Repetir los pasos anteriores, combinando los cuádruplos en octetos y seguir haciendo esto (cada vez duplicando las longitudes de las subsecuencias combinadas) hasta que quede ordenada toda la secuencia.

Para dar un ejemplo examinemos la secuencia

44 55 12 42 94 18 06 67

En el paso 1, los resultados divididos en las secuencias

44 55 12 42
94 18 06 67

La mezcla de los componentes individuales (que son secuencias ordenadas de longitud 1) en pares ordenados nos da

44 94' 18 55' 06 12' 42 67

Dividiendo otra vez por la mitad y combinando los pares ordenados, obtenemos

06 12 44 94' 18 42 55 67

Una tercera operación de división y mezcla produce al fin el resultado deseado

06 12 18 42 44 55 67 94

Cada operación que trata el conjunto entero de datos una vez recibe el nombre de *fase* y el proceso más corto que, por repetición, constituye el proceso de clasificación recibe el nombre de *pase* o *etapa*. En el ejemplo precedente la clasificación tuvo tres pasos, cada uno consistente en una fase de división y una fase de combinación. A fin de realizar la clasificación, se requieren tres cintas: el proceso se llama, pues, *mezcla de tres cintas*.

En realidad, las fases de división no contribuyen a la clasificación pues de ninguna manera permutan los elementos; en cierto modo son improductivas pese a que constituyen la mitad de todas las operaciones de copiado. Pueden eliminarse por completo al combinar la fase de división con la de mezcla. En vez de combinar en una sola secuencia, el resultado del proceso se redistribuye al instante en dos cintas, que constituyen las fuentes del pase subsecuente. En contraste con la clasificación *por mezcla de dos fases*, esta técnica se conoce con el nombre de *mezcla de una sola fase* o *mezcla balanceada*. Es evidentemente superior porque sólo la mitad de las operaciones de copiado se necesitan; el precio de esa ventaja es la necesidad de una cuarta cinta.

Desarrollaremos un programa de combinación con detalle e inicialmente dejaremos que los datos se representen como en un arreglo, pero éste se rastrea en una estricta forma secuencial. Una versión posterior de la clasificación por fusión se basará después en la estructura secuencial, permitiendo con ello una comparación de los dos programas y demostrando la fuerte dependencia que tiene la forma de un programa con la representación de sus datos.

Un arreglo individual puede usarse fácilmente en lugar de dos secuencias, si se considera como de doble extremo. En vez de hacer la mezcla con dos archivos fuente, tomaremos elementos de los dos extremos del arreglo. En consecuencia, la forma general de la fase combinada de mezcla-división puede ilustrarse como se hace en la figura 2.12. El destino de los elementos combinados se cambia después de cada par ordenado en la primera fase, después de cada cuádruplo ordenado en la segunda fase, etc.; de ese modo se llenan uniformemente las dos secuencias destino, representadas por los dos extremos de un solo arreglo. Luego de cada pase, los dos arreglos intercambian su papel: la fuente se convierte en el nuevo destino y viceversa.

Una simplificación ulterior del programa puede lograrse uniendo los dos arreglos conceptualmente distintos en un solo de tamaño duplicado. Así pues, los datos pueden representarse por

a: ARRAY[1 .. 2*n] OF item (2.20)

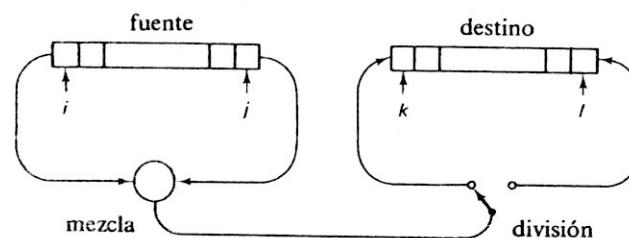


Fig. 2.12 Clasificación directa por mezcla, con dos arreglos.

y dejemos que los índices i y j denotan los dos elementos fuente, en tanto que k y L designan los dos destinos (véase la figura 2.12). Desde luego, los datos iniciales son $a_1 \dots a_n$. Salta a la vista que se necesita una variable booleana up para denotar la dirección del flujo de datos; esa variable significa que en los componentes del pase actual $a_1 \dots a_n$ será transferido hacia abajo a $a_1 \dots a_n$. El valor de up alterna estrictamente entre pasos consecutivos. Y finalmente una variable p se introduce para indicar la longitud de las subsecuencias por combinar. Su valor inicial es 1 y se duplica antes de cada pase sucesivo. Para simplificar un poco las cosas, supondremos que n siempre es una potencia de 2. Así pues, la primera versión del programa de mezcla directa supone la forma siguiente:

```

PROCEDURE MergeSort;
  VAR i, j, k, L: index; up: BOOLEAN; p: INTEGER;
BEGIN up := TRUE; p := 1;
  REPEAT inicializar variables índice;
    IF up THEN i := 1; j := n; k := n+1; L := 2*n
    ELSE k := 1; L := n; i := n+1; j := 2*n
    END ;
    mezclar p-uplos de las fuentes i y j hacia los destinos k y L;
    up := ~up; p := 2*p
  UNTIL p = n
END MergeSort

```

(2.21)

En el siguiente paso del desarrollo refinaremos más las proposiciones expresadas en letras cursivas. Es evidente que el pase de combinación que incluye n elementos es también una secuencia de combinaciones de secuencias, es decir, p -túplos. Entre cada mezcla parcial de ese tipo, el destino se cambia del extremo inferior al superior del arreglo destino o a la inversa para garantizar una distribución igual en ambos destinos. Si el destino de los elementos combinados es el extremo inferior del arreglo destino, entonces el índice destino es k , que se incrementa tras cada movimiento de un elemento. Si los elementos han de ser movidos al extremo superior del arreglo destino, el índice destino es L y se incrementa tras cada movimiento. A fin de simplificar la verdadera proposición de mezcla, decidimos que el destino se designe con k siempre, cambiando los valores de las variables k y L tras cada combinación p -tupla; el incremento que se usará siempre se denota con h , mientras que h es 1 o -1. Estas explicaciones sobre el diseño nos llevan al siguiente refinamiento:

```

h := 1; m := n; (*m = numero de elementos por mezclar*)          (2.22)
REPEAT q := p; r := p; m := m - 2*p;
  mezclar q elementos de las fuentes i con r elementos procedentes
  de la fuente j el índice destino k se aumenta en h.
  h := -h; k ↔ L
UNTIL m = 0

```

En el paso de refinamiento ulterior hay que formular la verdadera proposición de la mezcla. Aquí hemos de tener presente que el cabo (extremo) de una subsecuencia, que no se deja vacío luego de la mezcla debe anexarse a la secuencia de salida mediante simples operaciones de copiado.

```

WHILE (q # 0) & (r # 0) DO
  IF a[i] < a[j] THEN
    mover un elemento de la fuente i al destino k, avanzar i y k; q := q-1      (2.23)
  ELSE
    mover un elemento de la fuente j al destino k, avanzar j y k; r := r-1
  END
END ;
copiar cabo de la secuencia i, copiar cabo de la secuencia j

```

Una vez terminado este refinamiento de las operaciones de copiado de cabos, el programa se presenta con todos sus pormenores. Antes de escribirlo enteramente, deseamos eliminar la restricción de que n debe ser una potencia de 2. ¿Qué partes del algoritmo se ven afectadas por esta medida? No nos cuesta mucho convencernos de que la mejor manera de afrontar la situación más general consiste en seguir el antiguo método en lo posible. En este ejemplo, ello significa que seguimos combinando p -tuplos antes que los restantes de la secuencia fuente tengan una longitud menor que p . La única y sola parte que queda afectada son las proposiciones que determinan los valores de q y r , o sea las longitudes de las secuencias que han de ser fusionadas. Las siguientes cuatro proposiciones reemplazan a las tres proposiciones

$q := p; r := p; m := m - 2*p$

y, como el lector deberá convencerse a sí mismo, representa una buena aplicación de la estrategia antes especificada; nótese que m denota el número total de elementos en las dos secuencias fuente que quedan por combinar:

```

IF m >= p THEN q := p ELSE q := m END ;
m := m-q;
IF m >= p THEN r := p ELSE r := m END ;
m := m-r

```

Además, a fin de garantizar la terminación del programa, la condición $p = n$, que controla la repetición externa, debe ser cambiada a $p \geq n$. Hechas esas modificaciones, podemos pasar a describir el algoritmo entero a partir de un programa completo (véase el programa 2.13).

```

PROCEDURE StraightMerge;
  VAR i, j, k, L, t: index; (*el intervalo del índice es 1 .. 2*n*)
  h, m, p, q, r: INTEGER; up: BOOLEAN;
BEGIN up := TRUE; p := 1;
  REPEAT h := 1; m := n;
    IF up THEN i := 1; j := n; k := n+1; L := 2*n
    ELSE k := 1; L := n; i := n+1; j := 2*n
    END ;
    REPEAT (*mezclar una corrida de las fuentes i y j al destino k*)
      IF m >= p THEN q := p ELSE q := m END ;
      m := m-q;

```

```

IF m >= p THEN r := p ELSE r := m END ;
m := m-r;
WHILE (q # 0) & (r # 0) DO
  IF a[i] < a[j] THEN
    a[k] := a[i]; k := k+h; i := i+1; q := q-1
  ELSE
    a[k] := a[j]; k := k+h; j := j-1; r := r-1
  END
END ;
WHILE r > 0 DO
  a[k] := a[j]; k := k+h; j := j-1; r := r-1
END ;
WHILE q > 0 DO
  a[k] := a[i]; k := k+h; i := i+1; q := q-1
END ;
h := -h; t := k; k := L; L := t
UNTIL m = 0;
up := ~up; p := 2*p
UNTIL p >= n;
IF ~up THEN
  FOR i := 1 TO n DO a[i] := a[i+n] END
END
END StraightMerge

```

Programa 2.13 Clasificación directa por mezcla en su arreglo.

Análisis de clasificación por mezcla. Cada paso duplica p y la clasificación se termina en cuanto $p \geq n$, por lo cual incluye $\lceil \log n \rceil$ pasos. Por definición cada pase copia el conjunto entero de n elementos exactamente una vez. En consecuencia, el número total es exactamente

$$M = n * \lceil \log n \rceil \quad (2.24)$$

El número C de comparaciones de llaves es incluso menor que M , porque no intervienen comparaciones en las operaciones de copiado de cabos. Sin embargo, como la técnica de clasificación por mezcla suele aplicarse cuando se utilizan dispositivos de almacenamiento periférico, la computación que participa en las operaciones de movimiento domina la actividad de comparaciones a menudo en diversos órdenes de magnitud. El análisis detallado del número de comparaciones será por ello de poco interés práctico.

El algoritmo de clasificación por mezcla no es inferior a las técnicas avanzadas que se comentaron en el capítulo anterior. No obstante, los "excesos" administrativos para manipular los índices son relativamente altos, y la gran desventaja es la necesidad de guardar $2n$ elementos. A ello se debe que la clasificación por mezcla rara vez se use en arreglo, esto es, con datos localizados en la memoria principal. En la última línea de la tabla 2.9 se dan cifras que comparan el comportamiento temporal real de este algoritmo. En ella se advierte su superioridad sobre la clasificación por montón pero su inferioridad respecto a la clasificación rápida.

2.4.2. Mezcla natural

En la mezcla directa no se obtiene ventaja alguna cuando los datos al inicio ya están parcialmente clasificados. La longitud de todas las subsecuencias combinadas en el k -ésimo pase es menor o igual que 2^k , sin importar si las subsecuencias más largas ya están ordenadas o podrían también combinarse. En efecto, dos subsecuencias ordenadas de longitudes m y n podrían combinarse directamente en una sola secuencia de $m + n$ elementos. Una clasificación por mezcla que en cualquier momento combina las dos subsecuencias más largas posibles recibe el nombre de *clasificación por mezcla natural*.

A menudo a la subsecuencia ordenada se le llama *cadena*. Pero como este término se emplea comúnmente para designar secuencias de caracteres, seguiremos el criterio de Knuth en nuestra terminología y utilizaremos la palabra *corrida* en vez de cadena al referirnos a subsecuencias ordenadas. Llamamos *corrida máxima* o, simplemente, *corrida* a una subsecuencia $a_1 \dots a_j$ tal que

$$(a_{i-1} > a_i) \& (a_k : i \leq k < j : a_k \leq a_{k+1}) \& (a_j > a_{j+1}) \quad (2.25)$$

Una clasificación por mezcla natural combina, pues, corridas (máximas) en vez de secuencias de longitud fija previamente determinada. Las corridas tienen la propiedad de que, si dos secuencias de n corridas se combinan, se produce una sola secuencia de exactamente n corridas. Por tanto, el número total se divide a la mitad en cada paso y el número de movimientos requeridos de elementos es $n * \lceil \log n \rceil$ en el peor caso, pero en el caso promedio es menos todavía. El número previsto de comparaciones es mucho mayor porque además de las que se necesitan para seleccionar elementos, se requieren más entre los elementos consecutivos de cada archivo para determinar el final de cada corrida.

Nuestro siguiente ejercicio de programación desarrolla un algoritmo de mezcla natural en la misma forma gradual que se utilizó al aplicar el algoritmo de mezcla directa. Se sirve de una estructura secuencia y no de un arreglo; representa una clasificación no balanceada de mezcla bifásica de tres cintas. Suponemos que la variable c representa la secuencia inicial de elementos. (Naturalmente, en la aplicación concreta del procesamiento de datos, los datos iniciales se copian de la fuente original a c por razones de seguridad.) a y b son dos variables de secuencia auxiliar. Cada pase consiste en una fase que distribuye las corridas uniformemente de c hasta a y b y una fase que mezcla las corridas desde a hasta b y c . Este proceso se describe en la figura 2.13.

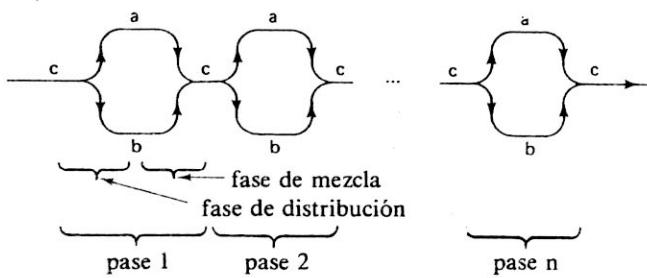


Fig. 2.13 Fases y pasos de clasificación.

He aquí un ejemplo: la tabla 2.11 muestra el archivo c en su estado original (lineal) y, tras cada pase (líneas 2-4), en una clasificación de mezcla natural que comprende 20 números. Nótese que sólo se necesitan tres pasos. La clasificación finaliza en cuanto el número de corridas en c es 1. (Suponemos que existe por lo menos una corrida no vacía en la secuencia inicial.) Por eso dejamos que la variable L sea usada para contar el número de corridas combinadas en c. Usando el tipo *sequence* definido en la sección 1.11, el programa se formulará como se muestra en (2.27).

```
VAR L: INTEGER; a, b, c: Sequence
REPEAT Reset(a); Reset(b); Reset(c);
  distribuir; (*c hacia a y b*)
  Reset(a); Reset(b); Reset(c);
  L := 0; mezclar (*a y en c*)
UNTIL L = 1
```

(2.27)

Las dos fases surgen claramente como dos proposiciones distintas. Ahora es preciso definirlas, esto es, expresarlas con mayor detalle. La descripción refinada de *distribuir* se da en (2.28) y la de *mezclar* en (2.29).

```
REPEAT copyrun(c, a);
  IF ~c.eof THEN copyrun(c, b) END
UNTIL c.eof
```

(2.28)


```
REPEAT mergerun; L := L+1
UNTIL b.eof;
IF ~a.eof THEN copyrun(a, c); L := L+1 END
```

(2.29)

Este método de distribución supuestamente produce un número igual de corridas tanto en a como en b o en la secuencia a que contiene una corrida más que b. Puesto que los pares correspondientes de corridas se combinan, una corrida restante puede estar todavía en el archivo a, el cual simplemente debe copiarse. Las proposiciones *mezclar* y *distribuir* se formulan en términos de una proposición refinada *mergerun* (*mezclar* corrida) y un procedimiento subordinado *copyrun* (copia corrida), que tienen tareas obvias. Cuando se intenta hacer esto, nos topamos con una grave dificultad: para determinar el final de una corrida, hay que comparar dos llaves de elementos consecutivos. Sin embargo, las secuencias son tales que sólo un elemento es accesible de inmediato. Salta a la vista que es imposible no *ver adelante*, es decir, asociar un buffer con todas las secuencias. El buffer debe contener el primer elemento de la secuencia que ha de leerse todavía y que constituye una especie de ventana que se desliza a lo largo de la secuencia.

```
17 31'05 59'13 41 43 67'11 23 29 47'03 07 71'02 19 57'37 61
05 17 31 59'11 13 23 29 41 43 47 67'02 03 07 19 57 71'37 61
05 11 13 17 23 29 31 41 43 47 59 67'02 03 07 19 37 57 61 71
02 03 05 07 11 13 17 19 23 29 31 37 41 43 47 57 59 61 67 71
```

Tabla 2.11 Ejemplo de clasificación por mezcla natural.

En vez de programar este mecanismo explícitamente en nuestro programa, preferimos definir otro nivel más de abstracción. Lo representamos con un nuevo módulo denominado *Sequence*, que reemplaza el módulo *FileSystem* para los clientes y define un nuevo y apropiado tipo de secuencia, pero se refiere al tipo *Sequence* de *FileSystem*. Este nuevo tipo no sólo indicará el final de una secuencia, sino también el fin de una corrida y, desde luego, el primer elemento de la parte restante de la secuencia. El nuevo tipo, lo mismo que sus operadores, se presentan mediante el siguiente módulo de definición:

```
DEFINITION MODULE Sequences;
IMPORT FileSystem;
TYPE item = INTEGER;
Sequence =
  RECORD first: item;
  eor, eof: BOOLEAN;
  f: FileSystem.Sequence
END;

PROCEDURE OpenSeq(VAR s: Sequence);
PROCEDURE OpenRandomSeq(VAR s: Sequence; length, seed: INTEGER);
PROCEDURE StartRead(VAR s: Sequence);
PROCEDURE StartWrite(VAR s: Sequence);
PROCEDURE copy(VAR x, y: Sequence);
PROCEDURE CloseSeq(VAR s: Sequence);
PROCEDURE ListSeq(VAR s: Sequence);
END Sequences.
```

Unas cuantas explicaciones más de la elección del procedimiento son indispensables. Como veremos luego, los algoritmos de clasificación explicados aquí y más adelante se basan en la copia de elementos de una a otra secuencia. Por consiguiente, un procedimiento *copy* toma el lugar de las operaciones individuales *leer* y *escribir*. Una vez abierto un archivo, un mecanismo de previsión ha de recibir instrucciones sobre si la secuencia se escribirá o se leerá. En el segundo caso el primer elemento debe ser introducido *antes* en el buffer de previsión. Los procedimientos *StartRead* y *StartWrite* adoptarán, pues, el papel de *Reset* en *FileSystem*.

Exclusivamente por razones de comodidad se incluyen otros dos procedimientos. *OpenRandomSeq* toma el lugar de *OpenSeq*, si hay que inicializar una secuencia con números en orden aleatorio, y se supone que el procedimiento *ListSeq* genera un listado de la secuencia especificada. Estos dos procedimientos servirán para probar los algoritmos que deben explicarse luego. La aplicación de este concepto se presenta por medio del siguiente módulo. Observamos que en el procedimiento *copy* el campo *first* de la secuencia destino contiene el valor del último (*last*) elemento descrito. Por otra parte, los valores de los campos *eof* y *eor* se definen como resultados de *copy* por analogía con *eof*, luego de haber sido definidos como resultados de una operación de leer.

```
IMPLEMENTATION MODULE Sequences;
FROM FileSystem IMPORT
  File, Open, ReadWord, WriteWord, Reset, Close;
FROM InOut IMPORT WriteInt, WriteLn;
```

```

PROCEDURE OpenSeq(VAR s: Sequence);
BEGIN Open(s.f)
END OpenSeq;
PROCEDURE OpenRandomSeq(VAR s: Sequence; length, seed: INTEGER);
  VAR i: INTEGER;
BEGIN Open(s.f);
  FOR i := 0 TO length-1 DO
    WriteWord(s.f, seed); seed := (31*seed) MOD 997 + 5
  END
END OpenRandomSeq;
PROCEDURE StartRead(VAR s: Sequence);
BEGIN Reset(s.f); ReadWord(s.f, s.first); s.eof := s.f.eof
END StartRead;

PROCEDURE StartWrite(VAR s: Sequence);
BEGIN Reset(s.f)
END StartWrite;
PROCEDURE copy(VAR x, y: Sequence);
BEGIN y.first := x.first;
  WriteWord(y.f, y.first); ReadWord(x.f, x.first);
  x.eof := x.f.eof; x.eor := x.eof OR (x.first < y.first)
END copy;
PROCEDURE CloseSeq(VAR s: Sequence);
BEGIN Close(s.f)
END CloseSeq;
PROCEDURE ListSeq(VAR s: Sequence);
  VAR i, L: CARDINAL;
BEGIN Reset(s.f); i := 0; L := s.f.length;
  WHILE i < L DO
    WriteInt(INTEGER(s.f.a[i]), 6); i := i+1;
    IF i MOD 10 = 0 THEN WriteLn END
  END;
  WriteLn
END ListSeq;
END Sequences.

```

Retornamos ahora al proceso de refinamiento sucesivo del proceso de mezcla (combinación) natural. El procedimiento *copyrun* y la proposición *mezclar* pueden expresarse ahora adecuadamente como se muestra en (2.30) y (2.31).

```

PROCEDURE copyrun(VAR x, y: Sequence);
BEGIN (*de x hacia y*)
  REPEAT copy(x, y) UNTIL x.eor
END copyrun
(2.30)

```

```

(*cambiar de a y b en c*)
REPEAT
(2.31)

```

```

IF a.first < b.first THEN
  copy(a, c);
  IF a.eor THEN copyrun(b, c) END
ELSE copy(b, c);
  IF b.eor THEN copyrun(a, c) END
END
UNTIL a.eor OR b.eor

```

El proceso de comparación y selección de llaves en la combinación de una corrida termina en cuanto una de las dos corridas llega a su fin. Después de eso, la otra corrida (que todavía no ha sido agotada) se transferirá a la corrida resultante con sólo copiar su cabo. Y eso se hace mediante una llamada al procedimiento *copyrun*.

Con esto se supone que finalizará el desarrollo de un procedimiento de clasificación por mezcla (combinación) natural. Por desgracia el programa no es correcto, como seguramente lo habrá ya advertido el lector perspicaz. Y no es correcto porque no clasifica debidamente en algunos casos. Tomemos, por ejemplo, la siguiente secuencia de datos de entrada:

03 02 05 11 07 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

Al distribuir las corridas consecutivas de modo alterno entre a y b, obtenemos

a = 03'07 13 19'29 37 43'57 61 71'
b = 02 05 11'17 23 31'41 47 59'67

Estas secuencias se combinan fácilmente en una sola corrida, tras lo cual la clasificación llega a feliz término. Aunque el ejemplo no propicia un comportamiento erróneo del programa, nos revela que la mera distribución de corridas en diversas secuencias puede producir un número de corridas de salida menor que el de corridas de entrada. Ello se debe a que el primer elemento de la corrida $i + 2$ da puede ser mayor que el último elemento de la i -ésima corrida, lo cual hace que las dos se combinen automáticamente en una sola.

Si bien el procedimiento *distribute* debe producir corridas en número iguales a las dos secuencias, la consecuencia más importante es que el número real de corridas resultantes en a y b puede diferir de modo significativo. Nuestro procedimiento de combinación sólo mezcla pares de corridas y termina en cuanto b se lee, con lo cual se pierde el cabo de una de las secuencias. Examinemos los siguientes datos de entrada que se clasifican (y truncan) en dos pasos subsecuentes:

17 19 13 57 23 29 11 59 31 37 07 61 41 43 05 67 47 71 02 03
13 17 19 23 29 31 37 41 43 47 57 71 11 59
11 13 17 19 23 29 31 37 41 43 47 57 59 71

Tabla 2.12 Resultado incorrecto de un programa de clasificación por mezcla.

El ejemplo de este error de programación caracteriza muchos problemas en esta disciplina. El error se debe a que no se toma en cuenta una de las posibles consecuencias de una operación supuestamente sencilla. También es típico porque se cuenta con varias maneras de corregirlo y es preciso escoger una de ellas. A menudo se dan dos posibilidades que difieren en un aspecto fundamental:

1. Reconocemos que la operación de distribución no está bien programada y que no satisface el requisito de que el número de corridas difiere a lo sumo en 1. Nos ajustamos al esquema original de la operación y corregimos a partir de él el procedimiento defectuoso.
2. Reconocemos que la corrección de la parte defectuosa exige modificaciones muy amplias, y procuramos encontrar las formas en que las otras partes del algoritmo puedan cambiarse y dar cabida a la parte momentáneamente incorrecta.

En general, la primera opción parece más segura, depurada y honesta, pues ofrece un grado satisfactorio de invulnerabilidad contra las consecuencias posteriores debida a los efectos colaterales intrincados e inadvertidos. De ahí que este intento de solución no suela recomendarse.

Con todo, conviene señalar que la segunda posibilidad en ocasiones no debe ignorarse del todo. Es por ello que analizaremos más a fondo este ejemplo e ilustraremos una solución modificando el procedimiento de mezcla (combinación) en vez del procedimiento de distribución, el cual es el que presenta la falla.

Lo anterior significa que no tocamos el esquema de distribución y renunciamos a la condición de que las corridas tengan una distribución uniforme. Con ello se obtiene un rendimiento que no es óptimo. Sin embargo, permanece inalterado el rendimiento en el peor caso y, además, el caso de una distribución muy desigual tiene poca probabilidad estadística. De ahí que las consideraciones de la eficiencia no sean un argumento válido contra esta solución.

Si la condición de distribución igual (uniforme) de las corridas ya no existe, el procedimiento de combinación debe cambiarse de modo que, luego de llegar al fin de un archivo, la cola entera del archivo restante se copia en lugar de una corrida a lo máximo. Tal cambio es sencillo y directo en comparación con los que se dan en el esquema de distribución. (Recomendamos encarecidamente al lector que se convenza a sí mismo de la veracidad de nuestra aseveración.) La versión revisada del algoritmo de mezcla queda incluida en el programa completo 2.14, en el cual los procedimientos llamados una sola vez han sido sustituidos directamente.

```
MODULE NaturalMerge;
FROM Sequences IMPORT item, Sequence, OpenSeq, OpenRandomSeq,
  StartRead, StartWrite, copy, CloseSeq, ListSeq;
VAR L: INTEGER; (*num. de corridas mezcladas*)
  a, b, c: Sequence;
  ch: CHAR;
PROCEDURE copyrun(VAR x, y: Sequence);
BEGIN (*de x hacia y*)
  REPEAT
```

```
REPEAT copy(x, y) UNTIL x.eor
END copyrun;
BEGIN OpenSeq(a); OpenSeq(b); OpenRandomSeq(c, 16, 531);
ListSeq(c);
REPEAT StartWrite(a); StartWrite(b); StartRead(c);
REPEAT copyrun(c, a);
  IF ~c.eof THEN copyrun(c, b) END
UNTIL c.eof;
StartRead(a); StartRead(b); StartWrite(c);
L := 0;
REPEAT
  LOOP
    IF a.first < b.first THEN
      copy(a, c);
      IF a.eor THEN copyrun(b, c); EXIT END
    ELSE copy(b, c);
      IF b.eor THEN copyrun(a, c); EXIT END
    END
  END ;
  L := L+1
UNTIL a.eof OR b.eof;
WHILE ~a.eof DO copyrun(a, c); L := L+1 END ;
WHILE ~b.eof DO copyrun(b, c); L := L+1 END
UNTIL L = 1;
ListSeq(c); CloseSeq(a); CloseSeq(b); CloseSeq(c)
END NaturalMerge.
```

Programa 2.14 Clasificación por mezcla natural.

2.4.3. Mezcla balanceada múltiple

El esfuerzo que requiere una clasificación secuencial es proporcional al número requerido de pasos ya que, por definición, cada paso supone el copiado del conjunto total de datos. Una manera de reducir ese número consiste en distribuir las corridas en más de dos secuencias. Mezclar r corridas que están distribuidas igualmente en N secuencias origina una secuencia de r/N corridas. Un segundo paso reduce el número a r/N^2 , un tercer paso a r/N^3 y al cabo de k pasos quedan r/N^k corridas. El número total de pasos que se requiere para clasificar n elementos por mezcla N -uple será entonces $k = \lceil \log_N n \rceil$. Puesto que cada paso requiere n operaciones de copiado, el número total de operaciones de copiado es, en el peor caso, $M = n * \lceil \log_N n \rceil$.

En el siguiente ejercicio de programación, desarrollaremos un programa de clasificación basado en una combinación múltiple. A fin de contrastar el programa proveniente del procedimiento anterior de combinación natural en dos fases, formularemos la mezcla múltiple como una *clasificación por mezcla balanceada de una sola fase*. Ello significa que en cada paso hay un número igual de archivos de entrada y salida en los cuales se distribuyen en forma alterna las corridas consecutivas. Utilizando N secuencias (N siendo par), el algoritmo se basará entonces en la combinación $N/2$ -uple. Siguiendo la estrategia adoptada antes, no nos detendremos a detectar la combinación automática de

dos corridas consecutivas, distribuidas en la misma secuencia. Por ello, nos vemos obligados a diseñar el programa sin suponer números estrictamente iguales de corridas en las secuencias de entrada.

En este programa nos encontramos por primera vez con una aplicación natural de la estructura de datos, la cual consiste en un arreglo de archivos. En realidad, no deja de ser sorprendente la gran diferencia entre el siguiente programa y el anterior, atribuible a la transición de una combinación de mezcla doble a otra de mezcla múltiple. El cambio proviene primordialmente de la circunstancia de que el proceso de combinación ya no puede terminarse al finalizar una de las corridas de entrada. Por el contrario, una lista de entradas que todavía esté activa, o sea que no esté agotada, ha de ser conservada. Otra complicación proviene de la necesidad de cambiar los grupos de entrada y las secuencias de salida una vez terminado cada pase.

Comenzamos definiendo, además de los dos conocidos tipos *item* y *sequence*, un tipo

$$\text{seqno} = [1 .. N] \quad (2.33)$$

Sin duda los números de secuencia se usan para indizar el arreglo de las secuencias de los elementos. Supongamos, pues, que la secuencia inicial de elementos está dada como una variable

$$f_0: \text{Sequence} \quad (2.34)$$

y que en el proceso de clasificación se dispone de n cintas, donde n es par

$$f: \text{ARRAY seqno OF Sequence} \quad (2.35)$$

Una técnica que se recomienda para abordar el problema de cambiar cintas consiste en introducir un mapa de índice de cintas. En vez de direccionar directamente una cinta por su índice i , esto se hace mediante un mapa t , es decir, en lugar de f_j escribimos f_{t_j} , donde el mapa se define como

$$t: \text{ARRAY seqno OF seqno} \quad (2.36)$$

Si inicialmente $t_i = i$ para todos los i , entonces un cambio consiste en intercambiar sencillamente los pares de los componentes del mapa $t_i \leftrightarrow t_{N/2+i}$ para todos los $i = 1 \dots N/2$, donde $N/2$ es el número de cintas. En consecuencia, siempre consideraremos $f_{t_1}, \dots, f_{t_{N/2}}$ como consecuencias de entrada y siempre consideraremos $f_{t_{N/2+1}}, \dots, f_{t_N}$ como consecuencias de salida. (En lo sucesivo simplemente llamaremos *secuencia j* a f_{t_j} .) Ahora el algoritmo puede formularse inicialmente como sigue:

```
MODULE BalancedMerge;
  VAR i, j: seqno;
  L: INTEGER; (*num. de corridas distribuidas*)
  t: ARRAY seqno OF seqno;
BEGIN (*distribuir corridas iniciales a t[1] ... t[N/2]*)
  j := Nh; L := 0;
  REPEAT
    IF j < Nh THEN j := j+1 ELSE j := 1 END;
    REPEAT
      MODULUS
```

```
copiar una corrida de f0 a la secuencia j;
L := L+1
UNTIL f0.eof;
FOR i := 1 TO N DO t[i] := i END ;
REPEAT (*mezclar de t[1] ... t[N/2] a t[N/2+1] ... t[N]*)
  restablecer secuencias de entrada;
  L := 0;
  j := Nh+1; (*j = indice de la secuencia de salida*)
  REPEAT L := L+1;
    mezclar una corrida de las entradas a t[j]
    IF j < N THEN j := j+1 ELSE j := Nh+1 END
    UNTIL agotar todas las entradas;
    cambiar secuencias;
  UNTIL L = 1
  (*la secuencia clasificada es t[1]*)
END BalancedMerge.
```

A continuación refinamos la proposición para la distribución inicial de corridas; usando la definición de secuencia (2.26) y de copiado (2.32), reemplazamos *copiar una corrida de f0 o a la secuencia j* mediante

$$\text{REPEAT copy}(f0, f[j]) \text{ UNTIL } f0.eor \quad (2.38)$$

El copiado de una corrida termina cuando encontramos el primer elemento en la siguiente corrida o cuando se llega al fin del archivo de entrada.

El algoritmo real de la clasificación, quedan por especificar más a fondo las siguientes proposiciones:

1. Restablecer las secuencias de entrada.
2. Mezclar una corrida a partir de las entradas a t_j .
3. Cambiar secuencias.
4. Agotadas todas las entradas.

Primero, hemos de identificar exactamente las secuencias de entrada actuales. Recuérdese que el número de entradas *activas* puede ser menor que $N/2$. En efecto, puede haber a lo sumo el mismo número de fuentes que de corridas; la clasificación finaliza tan pronto quede una sola secuencia. Con ello se abre la posibilidad de que al inicio de la última clasificación haya menos de $N/2$ corridas. De ahí que introduzcamos una variable, digamos $k1$, para denotar el número real de entradas utilizadas. Incorporamos del modo siguiente la inicialización de $k1$ a la proposición *reestablecer secuencias de entrada*.

```
IF L < Nh THEN k1 := L ELSE k1 := Nh END ;
FOR i := 1 TO k1 DO StartRead(f[t[i]]) END
```

Desde luego, la proposición (2) hace que disminuya $k1$ cada vez que cesa una fuente de entrada. Por eso, el predicado (4) puede expresarse fácilmente como la relación $k1 = 0$.

No obstante, la proposición (2) es más difícil de refinar; consta de la selección repetida de la llave más pequeña entre las fuentes disponibles y su transporte subsecuente al destino, es decir, la secuencia de la salida actual. El proceso se complica aún más por la necesidad de determinar el final de cada corrida. El final de una corrida puede alcanzarse porque 1) la llave subsecuente es menor que la actual o 2) se alcanza el final de la fuente. En el segundo caso la fuente se elimina al disminuir k_1 ; en el primer caso la corrida se concluye al excluir la secuencia en la selección ulterior de elementos, pero sólo hasta terminar la creación de la corrida de la salida actual. Y así resulta evidente que una segunda variable, digamos k_2 , se necesita para denotar el número de fuentes disponibles en ese momento para seleccionar el siguiente elemento. Este valor se hace inicialmente que sea igual a k_1 y se disminuye cada vez que termina una corrida debido a la condición (1).

Por desgracia, la introducción de k_2 no es suficiente. Debemos conocer no sólo el número de secuencias, sino también cuáles se emplean todavía. Una solución obvia consiste en usar un arreglo con componentes booleanos que indican las disponibilidad de las secuencias. Sin embargo, escogemos un método diferente que permite un procedimiento más eficaz de selección, el cual después de todo es la parte más repetida del algoritmo entero. En lugar de utilizar un arreglo booleano, se introduce un segundo mapa de cinta, digamos ta . Este mapa se emplea en lugar de t tal que $ta_1 \dots ta_{k_2}$ sean los índices de las secuencias disponibles. Así pues, la proposición (2) puede formularse así:

```
K2 := k1;
REPEAT seleccionar la llave mínima
    sea ta[mx] el número de secuencia en que ocurre;
    copy(f[ta[mx]], f[t[j]]);                                (2.39)
    IF f[ta[mx]].eof THEN eliminar cinta
    ELSIF f[ta[mx]].eor THEN terminar corrida
    END
UNTIL k2 = 0
```

Puesto que el número de secuencias será bastante pequeño en la práctica, el algoritmo de selección que debe especificarse con más detalle en el siguiente paso de refinamiento pudiera ser una búsqueda lineal sencilla. La proposición *eliminar cinta* supone una disminución de k_1 y también de k_2 ; además supone una reasignación de los índices del mapa ta . La proposición *cierra corrida* simplemente disminuye k_2 y rearregla, en consecuencia, los componentes de ta . Los detalles se muestran en el programa 2.15, que es un último refinamiento de (2.37) a (2.39). La proposición *cambia secuencias* se refina conforme a las explicaciones dadas antes.

```
MODULE BalancedMerge;
FROM Sequences IMPORT item, Sequence, OpenSeq, OpenRandomSeq,
    StartRead, StartWrite, copy, CloseSeq, ListSeq;

CONST N = 4; Nh = N DIV 2;
TYPE seqno = [1 .. N];
VAR i, j, mx, tx: seqno;
    L, k1, k2: CARDINAL;
    min, x: item;
```

```
t, ta: ARRAY seqno OF seqno;
f0: Sequence;
f: ARRAY seqno OF Sequence;

BEGIN OpenRandomSeq(f0, 100, 737); ListSeq(f0);
FOR i := 1 TO N DO OpenSeq(f[i]) END ;
(*distribuir corridas iniciales a t[1] ... t[Nh]*)
FOR i := 1 TO Nh DO StartWrite(f[i]) END ;
j := Nh; L := 0; StartRead(f0);
REPEAT
    IF j < Nh THEN j := j+1 ELSE j := 1 END ;
    REPEAT copy(f0, f[j]) UNTIL f0.eof;
    L := L+1
UNTIL f0.eof;
FOR i := 1 TO N DO t[i] := i END ;
REPEAT (*mezclar de t[1] ... t[nh] a t[nh+1] ... t[n]*)
    IF L < Nh THEN k1 := L ELSE k1 := Nh END ;
    FOR i := 1 TO k1 DO
        StartRead(f[t[i]]); ta[i] := t[i]
    END ;
    L := 0; (*num. de corridas combinadas*)
    j := Nh+1; (*j = índice de la secuencia de salida*)
    REPEAT (*mezclar una corrida de entradas a t[j]*)
        L := L+1; k2 := k1;
        REPEAT (*seleccionar la llave mínima*)
            i := 1; mx := 1; min := f[ta[1]].first;
            WHILE i < k2 DO
                i := i+1; x := f[ta[i]].first;
                IF x < min THEN min := x; mx := i END
            END ;
            copy(f[ta[mx]], f[t[j]]);
            IF f[ta[mx]].eof THEN (*eliminar cinta*)
                StartWrite(f[ta[mx]]); ta[mx] := ta[k2];
                ta[k2] := ta[k1]; k1 := k1-1; k2 := k2-1
            ELSIF f[ta[mx]].eor THEN (*terminar corrida*)
                tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx; k2 := k2-1
            END
        UNTIL k2 = 0;
        IF j < N THEN j := j+1 ELSE j := Nh+1 END
    UNTIL k1 = 0;
    FOR i := 1 TO Nh DO
        tx := t[i]; t[i] := t[i+Nh]; t[i+Nh] := tx;
    END
UNTIL L = 1;
ListSeq(f[t[1]])
```

```
(*la secuencia clasificada es t[1]*)
END BalancedMerge.
```

Programa 2.15 Clasificación por mezcla balanceada.

2.4.4. Clasificación polifásica

Ya hemos explicado las técnicas necesarias y hemos dado suficiente información para investigar y programar otro algoritmo más de clasificación, cuyo rendimiento es superior a la clasificación balanceada. Hemos visto que la mezcla balanceada elimina las puras operaciones de copiado que se requieren cuando se unen en una sola fase las operaciones de distribución y mezcla. Surge la cuestión de si las secuencias dadas pueden procesarse con mayor eficiencia aún, y si es factible eso. La clave de este mejoramiento ulterior consiste en abandonar la rígida noción de pasos estrictos, o sea en utilizar las secuencias de modo más complejo que el simple hecho de tener $N/2$ fuentes y otros tantos destinos, e intercambiar fuentes y destinos al final de cada pase. En cambio, el concepto de pase se torna vago. El método lo inventó R.L. Gilstad [2-3] y se conoce con el nombre de *clasificación polifásica*.

Lo ilustramos primero con un ejemplo en que intervienen tres secuencias. En todo momento, los elementos se mezclan a partir de dos fuentes en una tercera variable de secuencia. Cada vez que se agota una de las secuencias fuente, de inmediato se convierte en el destino de las operaciones de combinación de los datos a partir de la fuente no agotada y de la anterior secuencia destino.

Sabemos que n corridas en cada entrada se transforman en n corridas en la salida, por lo cual necesitamos listar sólo el número de las corridas presentes en cada secuencia (en vez de especificar las llaves reales). En la figura 2.14 suponemos que al inicio las dos se-

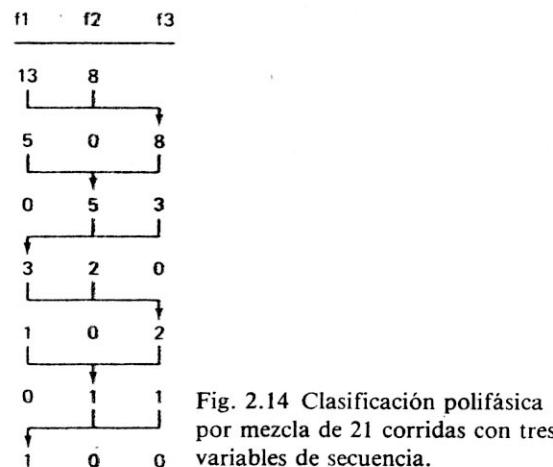


Fig. 2.14 Clasificación polifásica por mezcla de 21 corridas con tres variables de secuencia.

cuencias de entrada f_1 y f_2 contienen 13 y 8 corridas, respectivamente. Así pues, en el primer pase 8 corridas se mezclan a partir de f_1 y f_2 con f_3 , en el segundo pase las 5 corridas restantes se mezclan a partir de f_3 y f_1 con f_2 , etc. Al final, f_1 es la secuencia clasificada.

Un segundo ejemplo muestra el método de clasificación polifásica con 6 secuencias. Supongamos que al inicio hay 16 corridas en f_1 , 15 en f_2 , 14 en f_3 , 12 en f_4 y 8 en f_5 . En el primer pase parcial, 8 corridas se combinan en f_6 ; al final, f_2 contiene el conjunto ordenado de los elementos (véase la figura 2.15).

La clasificación polifásica es más eficiente que la combinación balanceada porque, con N secuencias, siempre opera con una mezcla de $N-1$ -ple en vez de con $N/2$ -ple. A medida que el número de pasos requeridos se approxima a $\log_N n$, donde n es el número de elementos por clasificar y N es el grado de las operaciones de combinación. La clasificación polifásica promete un notable mejoramiento respecto a la mezcla balanceada.

Desde luego, la distribución de las corridas iniciales se escogió cuidadosamente en los ejemplos citados. A fin de averiguar cuáles distribuciones iniciales de corridas llevan a un funcionamiento apropiado, procedemos en sentido contrario, comenzando con la última distribución (última línea en la figura 2.15). Reescribir las tablas de los dos ejemplos y al girar cada renglón una posición respecto al renglón anterior produce las tablas 2.13 y 2.14 para seis pasos y para tres y seis secuencias respectivamente.

Cinta	f_1	f_2	f_3	f_4	f_5	f_6
	16	15	14	12	8	
	8	7	6	4	0	8
	4	3	2	0	4	4
	2	1	0	2	2	2
	1	0	1	1	1	1
	0	1	0	0	0	0

Fig. 2.15 Clasificación polifásica por mezcla de 65 corridas con seis variables de secuencia.

L	$a_1(L)$	$a_2(L)$	Suma $a_i(L)$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Tabla 2.13 Distribución perfecta de corridas en dos secuencias.

L	$a_1^{(L)}$	$a_2^{(L)}$	$a_3^{(L)}$	$a_4^{(L)}$	$a_5^{(L)}$	Suma $a_i^{(L)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Tabla 2.14 Distribución perfecta de corridas en cinco secuencias.

De la tabla 2.13 podemos deducir para $L > 0$ las relaciones

$$\begin{aligned} a_2^{(L+1)} &= a_1^{(L)} \\ a_1^{(L+1)} &= a_1^{(L)} + a_2^{(L)} \end{aligned} \quad (2.40)$$

y $a_1(0) = 1$, $a_2(0) = 0$. Definiendo $f_{i+1} = a_1^{(i)}$, obtenemos para $i > 0$

$$f_{i+1} = f_i + f_{i-1}, \quad f_1 = 1, \quad f_0 = 0 \quad (2.41)$$

Estas son las reglas recursivas (o relaciones de recurrencia) que definen a los *números de Fibonacci*:

$$f = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Cada número de Fibonacci es la suma de sus dos predecesores. En consecuencia, los números de las corridas iniciales en las dos secuencias de salida deben ser dos números consecutivos de Fibonacci, para que la clasificación polifásica funcione bien con tres secuencias.

¿Y cómo afecta esto al segundo ejemplo (tabla 2.14) con seis secuencias? Las reglas de formación se derivan fácilmente como

$$\begin{aligned} a_5^{(L+1)} &= a_1^{(L)} \\ a_4^{(L+1)} &= a_1^{(L)} + a_5^{(L)} = a_1^{(L)} + a_1^{(L-1)} \\ a_3^{(L+1)} &= a_1^{(L)} + a_4^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} \\ a_2^{(L+1)} &= a_1^{(L)} + a_3^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + a_1^{(L-3)} \\ a_1^{(L+1)} &= a_1^{(L)} + a_2^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + a_1^{(L-3)} + a_1^{(L-4)} \end{aligned} \quad (2.42)$$

Sustituyendo f_i por $a_1^{(i)}$ se obtiene

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \quad \text{para } i \geq 4 \\ f_4 &= 1 \\ f_i &= 0 \quad \text{para } i < 4 \end{aligned} \quad (2.43)$$

Estos números son los llamados números de Fibonacci de orden 4. En general, los *números de Fibonacci de orden p* se definen como sigue:

$$\begin{aligned} f_{i+1}(p) &= f_i(p) + f_{i-1}(p) + f_{i-2}(p) + f_{i-3}(p) + f_{i-4}(p) \quad \text{para } i \geq p \\ f_p(p) &= 1 \\ f_i(p) &= 0 \quad \text{para } 0 \leq i < p \end{aligned} \quad (2.44)$$

Nótese que los números ordinarios de Fibonacci son los de orden 1.

Ya hemos visto que los números iniciales de corrida para una clasificación polifásica perfecta con N secuencias son las sumas de cualquier $N-1$, $N-2$, ..., 1 (véase la tabla 2.15) números consecutivos de Fibonacci de orden $N-2$. Ello significa evidentemente que este método sólo es aplicable a las entradas cuyo número de corridas sea la suma de $N-1$ sumas de números de Fibonacci. Y así surge una importante pregunta: ¿qué debemos hacer cuando el número de corridas iniciales no es una suma ideal como éste? La respuesta es sencilla (y típica de tales situaciones): simulamos la existencia de corridas hipotéticas vacías, tal que la suma de las corridas reales e hipotéticas sea un total perfecto. Las corridas vacías reciben el nombre de *corrida simulada (ficticia)*.

Pero esta no es realmente una respuesta satisfactoria porque de inmediato plantea una pregunta más difícil: ¿cómo reconocemos las corridas ficticias durante la mezcla? Antes de contestar tal pregunta hemos de investigar el problema previo de la distribución inicial de las corridas y escoger una regla para distribuir las corridas reales y las simuladas en las $n-1$ cintas.

Para encontrar una regla apropiada de la distribución, es preciso saber cómo se mezclan las corridas reales y ficticias. Por supuesto, la selección de una corrida del segundo tipo en una secuencia i significa exactamente que la secuencia i se ignora durante esta combinación dando por resultado una mezcla proveniente de menos de $N-1$ fuentes. Combinar una corrida simulada de todas las fuentes $N-1$ no implica una operación de combinación real, sino el registro de la corrida simulada resultante en la secuencia de salida. De esto concluimos que las corridas simuladas deben distribuirse en las $n-1$ secuencias con la mayor uniformidad posible, puesto que nos interesan las combinaciones (mezclas) activas de cuantas fuentes sea posible obtenerlas.

2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001

Tabla 2.15 Números de corridas que admiten una distribución perfecta.

Olividémonos por un momento de las corridas simuladas y examinemos el problema de distribuir un número *desconocido* de corridas en $N-1$ secuencias. Está claro que los números de Fibonacci del orden $N-2$ que especifican los números deseados de corridas en cada fuente pueden generarse mientras se efectúa la distribución. Suponiendo, por ejemplo, $N = 6$ y consultando la tabla 2.14, comenzamos por distribuir las corridas como lo indica el renglón con el índice $L = 1$ ($1, 1, 1, 1, 1$); si se dispone de más corridas, pasamos al segundo renglón ($2, 2, 2, 2, 1$); si la fuente todavía no está agotada, la distribución avanza de acuerdo con el tercer renglón ($4, 4, 4, 3, 2$) y así sucesivamente. Llamaremos *level* (nivel) al índice de renglón. Es patente que, cuanto más grande sea el número de corridas, más alto será el nivel de los números de Fibonacci. A propósito, éste es igual al número de pases de combinación o de cambios necesarios para la siguiente clasificación. El algoritmo de distribución puede formularse ahora en una primera versión como sigue:

1. Sea la meta de la distribución los números de Fibonacci de orden $N-2$, nivel 1.
2. Distribuir conforme a la meta establecida.
3. Si se alcanza la meta, se calcula el siguiente nivel de los números de Fibonacci; la diferencia entre ellos y los del nivel anterior constituye la nueva meta de distribución. Vuelva al paso 2. Si la meta no puede lograrse por haberse agotado la fuente, termine el proceso de distribución.

Las reglas para calcular el siguiente nivel de los números de Fibonacci están contenidas en su definición (2.44). Podemos concentrarnos, pues, en el paso 2, donde dada una meta determinada las corridas posteriores deben distribuirse una tras otra en las $N-1$ secuencias de salida. Es aquí donde las corridas simuladas tienen que reaparecer en nuestras consideraciones.

Supongamos que, cuando elevamos el nivel, registramos la siguiente meta por las diferencias d_i para $i = 1 \dots N-1$, donde d_i denota el número de corridas por poner la secuencia i en este paso. Ahora podemos suponer que inmediatamente ponemos d_i corridas simuladas en la secuencia i y después considerar la distribución siguiente como la sustitución de las corridas simuladas por las reales; registramos cada vez una sustitución restándole 1 al conteo d_i . Así pues, d_i indica el número de corridas simuladas en la secuencia i cuando la fuente queda vacía.

No se sabe cuál algoritmo produce la distribución óptima, pero el siguiente método ha resultado bueno. Se llama *distribución horizontal* (cf. Knuth, vol. 3, p. 270), designación que puede entenderse imaginando que las corridas están apiladas en forma de silos, como se advierte en la figura 2.16 para $N = 6$, nivel 5 (cf. tabla 2.14).

Para alcanzar una distribución igual de las restantes corridas simuladas a la mayor brevedad posible, su sustitución por corridas reales reduce el tamaño de las pilas al extraer corridas simuladas en los niveles horizontales, de izquierda a derecha. De este modo, las corridas se distribuyen en las secuencias indicadas por su número, como se aprecia en la figura 2.16.

Ya estamos en condiciones de describir el algoritmo en forma de un procedimiento llamado *select*, el cual es activado cada vez que se ha copiado una corrida y se escoge una

8				
7	1			
6	2	3	4	
5	5	6	7	8
4	9	10	11	12
3	13	14	15	16
2	18	19	20	21
1	23	24	25	26
	28	29	30	31
				32

Fig. 2.16 Distribución horizontal de las corridas.

nueva fuente para la siguiente. Suponemos la existencia de una variable j que denota el índice de la actual secuencia destino. Los elementos a_i y d_i denotan los números de la distribución ideal y la simulada para la secuencia i .

```
j:      seqno;
a, d:  ARRAY seqno OF INTEGER;
level: INTEGER
```

Estas variables se inicializan con los siguientes valores:

```
ai = 1, di = 1   for i = 1 ... N-1
aN = 0, dN = 0 (dummy)
j = 1, level = 1
```

Nótese que el procedimiento *select* sirve para calcular el siguiente renglón de la tabla 2.14, esto es, los valores $a_1^{(L)} \dots a_{N-1}^{(L)}$ cada vez que aumente el nivel. La siguiente meta, o sea calcular las diferencias $d_i = a_i^{(L)} - a_i^{(L-1)}$ también se consigue en ese momento.

El algoritmo indicado se basa en el hecho de que la d_i resultante decrece al aumentar el índice (escalera descendente en la figura 2.16). Nótese que la excepción es la transición del nivel 0 al nivel 1; este algoritmo debe, pues, utilizarse comenzando en el nivel 1. El procedimiento *select* termina disminuyendo d_j en 1; esta operación sustituye al reemplazo de una corrida simulada en la secuencia j por una corrida verdadera.

Suponiendo la disponibilidad de una rutina para copiar una corrida de la fuente f_0 en la fuente f_j , podemos formular la fase inicial de la distribución como sigue (suponiendo siempre que la fuente contiene una corrida por lo menos):

```
REPEAT select; copyrun
UNTIL f0.eof
```

(2.47)

Sin embargo, aquí hemos de hacer una breve pausa y recordar el efecto que encontramos al distribuir las corridas en el algoritmo de combinación natural explicando antes: el hecho de que dos corridas lleguen consecutivamente al mismo destino y constituyan una sola corrida hace que los números supuestos de corridas sean incorrectos. Al diseñar un algoritmo de clasificación tal que su corrección no dependa del número de corridas este

efecto secundario puede ignorarse sin riesgo alguno. Sin embargo, en la clasificación polifásica, nos interesa llevar un control del número exacto de corridas en cada archivo. De ahí que no podamos darnos el lujo de ignorar el efecto de esa combinación casual. Por ello, otra complicación más del algoritmo de distribución no podrá ser evitada. Se hace necesario conservar las llaves del último elemento de la corrida anterior en cada secuencia. Por fortuna, nuestra realización de secuencias hace exactamente eso. En el caso de las secuencias de salida, *f.first* representa el último elemento escrito. Por tanto, un siguiente intento por describir el algoritmo de distribución será

```
REPEAT select; (2.48)
  IF f[j].first <= f0.first THEN continuar corrida anterior END ;
  copyrun
UNTIL f0.eof
```

El error obvio consiste en olvidar que *f[j].first* ha obtenido un valor sólo después de copiar la primera corrida. Una solución correcta será pues, distribuir antes una corrida en cada una de las *N-1* secuencias destino sin examinar *f[j].first*. El resto de las corridas son distribuidas conforme a (2.49).

```
WHILE ~f0.eof DO (2.49)
  select;
  IF f[j].first <= f0.first THEN
    copyrun;
    IF f0.eof THEN d[j] := d[j] + 1 ELSE copyrun END
  ELSE copyrun
  END
END
```

Y ahora estamos finalmente en condiciones de abordar el principal algoritmo de clasificación polifásica por mezcla. Su principal estructura es semejante a la parte principal del programa de mezcla *N-ple*: un ciclo externo cuyo cuerpo mezcla las corridas hasta que quedan agotadas las fuentes, un ciclo interno cuyo cuerpo mezcla una sola corrida proveniente de cada fuente y un ciclo más interno cuyo cuerpo selecciona la llave inicial y transmite el elemento en cuestión al archivo en blanco. Las principales diferencias son las siguientes:

1. En vez de *N/2*, sólo hay una secuencia de salida en cada pase.
2. En vez de cambiar las *N/2* secuencias de entrada y las *N/2* secuencias de salida al terminar cada pase, las secuencias se rotan. Y esto se consigue usando un mapa índice de secuencias *t*.
3. El número de secuencias de entrada varía entre una corrida y la siguiente; al inicio de cada corrida, depende de los conteos *d_i* de las corridas simuladas. Si *d_i>0* para toda *i*, entonces *N-1* corridas simuladas son seudocombinadas en una corrida simulada con sólo incrementar el conteo *d_N* de la secuencia de salida. De lo contrario, una corrida se mezcla a partir de todas las fuentes cuando *d_i=0* y *d_i* se disminuye en todas las otras secuencias, lo cual indica que se extrajo otra corrida simulada. Denotamos con *k* el número de las secuencias de salida que participan en una combinación.

4. Es imposible derivar la terminación de una fase mediante la condición final de la secuencia *N-1*-ésima, ya que pueden requerirse más combinaciones que incluyan corridas simuladas provenientes de la fuente. Por el contrario, el número teóricamente necesario de corridas depende de los coeficientes *a_j*. Esos coeficientes se calcularon durante la fase de distribución; ahora pueden recalcularse en sentido inverso.

La parte principal de la clasificación polifásica puede formularse ahora con base en las reglas expuestas antes, suponiendo que todas las *N-1* secuencias con corridas iniciales están creadas para ser leídas y que el mapa de cintas se establezca inicialmente en *t_i=i*.

```
REPEAT (*mezclar de t[1]...t[N - 1] a t[N]*) (2.50)
  z := a[N-1]; d[N] := 0; StartWrite(f[t[N]]);
  REPEAT k := 0; (*mezclar una corrida*)
    (*determinar num. de secuencias de entrada activas*)
    FOR i := 1 TO N-i DO
      IF d[i] > 0 THEN d[i] := d[i] - 1
      ELSE k := k+1; ta[k] := t[i]
    END
    END ;
    IF k = 0 THEN d[N] := d[N] + 1
    ELSE mezclar una corrida real de t[1]...t[k] a t[N]
    END ;
    z := z-1
  UNTIL z = 0;
  StartRead(f[t[N]]);
  girar secuencia en mapa t; calcular a [i] para siguiente nivel;
  StartWrite(f[t[N]]); level := level - 1
  UNTIL level = 0
(*la salida clasificada es t[1]*)
```

La operación de la verdadera mezcla es casi idéntica a la de la clasificación por combinación *N-ple*, consistiendo la única diferencia en que el algoritmo de eliminación de secuencias es un poco más sencillo. La rotación del mapa índice de las secuencias y los conteos correspondientes *d_i* (y el recálculo de los coeficientes *a_i* hacia abajo del nivel) es sencillo y puede estudiarse detenidamente en el programa 2.16, que representa el algoritmo para la clasificación polifásica en su totalidad.

```
MODULE Polyphase;
  FROM Sequences IMPORT item, Sequence, OpenSeq, OpenRandomSeq,
    StartRead, StartWrite, copy, CloseSeq, ListSeq;

  CONST N = 6;
  TYPE seqno = [1 .. N];

  VAR i, j, mx, tn: seqno;
    k, dn, z, level: CARDINAL;
```

```

x, min: item;
a, d: ARRAY seqno OF CARDINAL;
t, ta: ARRAY seqno OF seqno;
f0: Sequence;
f: ARRAY seqno OF Sequence;

PROCEDURE select;
  VAR i, z: CARDINAL;
BEGIN
  IF d[j] < d[j+1] THEN j := j+1
  ELSE
    IF d[j] = 0 THEN
      level := level + 1; z := a[1];
      FOR i := 1 TO N-1 DO
        d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
      END
    END;
    j := 1
  END;
  d[j] := d[j] - 1
END select;

PROCEDURE copyrun; (* de f0 a f[j]*)
BEGIN
  REPEAT copy(f0, f[j]) UNTIL f0.eor
END copyrun;

BEGIN OpenRandomSeq(f0, 100, 561); ListSeq(f0);
FOR i := 1 TO N DO OpenSeq(f[i]) END ;
(*distribuir corridas iniciales*)
FOR i := 1 TO N-1 DO
  a[i] := 1; d[i] := 1; StartWrite(f[i])
END ;
level := 1; j := 1; a[N] := 0; d[N] := 0; StartRead(f0);
REPEAT select; copyrun
UNTIL f0.eof OR (j = N-1);
WHILE ~f0.eof DO
  select; (* primero = ultimo elemento escrito sobre f[j]*)
  IF f[j].first <= f0.first THEN
    copyrun;
    IF f0.eof THEN d[j] := d[j] + 1 ELSE copyrun END
  ELSE copyrun
  END
END ;
FOR i := 1 TO N-1 DO t[i] := i; StartRead(f[i]) END ;
t[N] := N;

```

```

REPEAT (*mezclar de t[1] ... t[N-1] a. t[N]*) 
z := a[N-1]; d[N] := 0; StartWrite(f[t[N]]));
REPEAT k := 0; (*mezclar una corrida*)
  FOR i := 1 TO N-1 DO
    IF d[i] > 0 THEN d[i] := d[i] - 1
    ELSE k := k+1; ta[k] := t[i]
  END
  END;
  IF k = 0 THEN d[N] := d[N] + 1
  ELSE (*mezclar una corrida real de t[1] ... t[k] a. t[N]*)
    REPEAT i := 1; mx := 1; min := f[ta[1]].first;
      WHILE i < k DO
        i := i+1; x := f[ta[i]].first;
        IF x < min THEN min := x; mx := i END
      END ;
      copy(f[ta[mx]], f[t[N]]);
      IF f[ta[mx]].eor THEN (*omitar esta fuente*)
        ta[mx] := ta[k]; k := k-1
      END
      UNTIL k = 0
    END;
    z := z-1
  UNTIL z = 0;
  StartRead(f[t[N]]); (*girar secuencias*)
  tn := t[N]; dn := d[N]; z := a[N-1];
  FOR i := N TO 2 BY -1 DO
    t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
  END ;
  t[1] := tn; d[1] := dn; a[1] := z;
  StartWrite(f[t[N]]); level := level - 1
  UNTIL level = 0 ;
  ListSeq(f[t[1]]));
  FOR i := 1 TO N DO CloseSeq(f[i]) END
END Polyphase.

```

Programa 2.16 Clasificación polifásica.

2.4.5. Distribución de las corridas iniciales

Hemos tenido que presentar los complejos programas de la clasificación secuencial, dado que los métodos más simples que operan con arreglos se basan en la disponibilidad de una memoria de acceso aleatorio lo suficientemente grande para alojar el conjunto entero de los datos que deben clasificarse. Muchas veces esa memoria no está disponible y entonces hay que utilizar dispositivos de memoria secuencial que sean lo bastante grandes como las cintas o discos. Sabemos que los métodos de clasificación secuencial diseñados hasta hoy prácticamente no necesitan memoria primaria en absoluto, salvo los

buffers de archivo y, naturalmente, el programa. Sin embargo, en realidad hasta las computadoras más pequeñas incluyen una memoria primaria de acceso aleatorio que casi siempre es mayor de la que requieren los programas que hemos desarrollado aquí. No se justifica el hecho de no aprovechar esto al máximo.

La solución consiste en combinar las técnicas de clasificación de arreglos y de secuencias. En particular, una clasificación de arreglos adaptada puede emplearse en la fase de distribución de las corridas iniciales con el efecto de que esas corridas ya tienen una longitud L aproximadamente del tamaño del almacenamiento primario de datos con que se cuenta. Es evidente que en los pasos posteriores de combinación ninguna clasificación adicional de arreglos podría mejorar el rendimiento puesto que las corridas aumentan constantemente de tamaño y, en consecuencia, siempre son más grandes que la memoria principal disponible. Por ello nos concentraremos de preferencia en mejorar el algoritmo que genera las corridas iniciales.

Por supuesto, de inmediato centramos nuestra investigación en los métodos logarítmicos de la clasificación de arreglos. El más adecuado de ellos es el método de clasificación por árbol o por montón (consúltese la sección 2.2.5). El montón puede considerarse como un túnel por donde todos los elementos han de pasar, algunos con mayor rapidez y otros más lentamente. La llave más pequeña se toma fácilmente de la cima del montón y su sustitución constituye un proceso de gran eficiencia. La acción de dirigir un componente desde la secuencia de entrada f_0 , hacerlo pasar por un túnel de un montón completo H y llevarlo a la secuencia de salida $f_{[j]}$ puede describirse sencillamente así:

$$\text{WriteWord}(f_{[j]}, H[1]); \text{ReadWord}(f_0, H[1]); \text{sift}(1, n) \quad (2.51)$$

Sift es el proceso descrito en la sección 2.3.2 para desplazar el componente H_1 hasta su sitio correspondiente. Nótese que dicho componente es el elemento más pequeño en el montón. Un ejemplo de esto se muestra en la figura 2.17. Con el tiempo el programa se torna mucho más complejo por las siguientes razones:

1. El montón H está inicialmente vacío y debe ser llenado.
2. Hacia el final, el montón está parcialmente lleno y a la poste se queda vacío.
3. Debemos llevar un control del inicio de las nuevas corridas, con objeto de cambiar el índice de salida j en el momento oportuno.

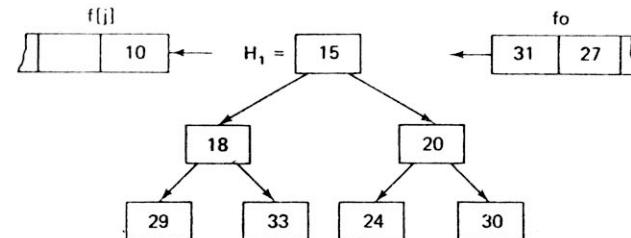
Antes de proseguir, declaremos formalmente las variables que intervienen en el proceso:

```
VAR f0: Sequence;
f: ARRAY seqno OF Sequence;
H: ARRAY [1 .. m] OF item;
L, R: INTEGER
```

m es el tamaño del montón H . Usamos la constante mh para denotar $m/2$; L y R son índices que delimitan el montón. El proceso de túnel puede dividirse entonces en cinco partes distintas.

1. Leer los primeros mh elementos a partir de f_0 y ponerlos en la mitad superior del montón donde no se prescriben ordenación entre las llaves.

Estado antes de una transferencia:



Estado después de la siguiente transferencia:

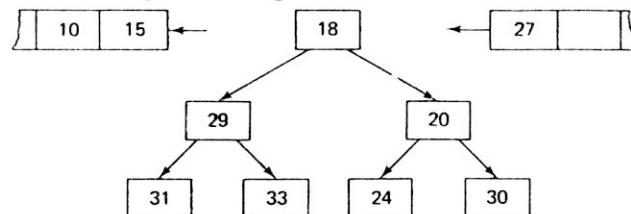


Fig. 2.17 Desplazamiento de una llave a través de un montón.

2. Leer otros mh elementos y colocarlos en la mitad inferior del montón, desplazando cada elemento hacia su posición correspondiente (construir montón).
3. Poner L en m y repetir el siguiente paso para todos los elementos restantes en f_0 : introducir H_1 en la secuencia apropiada de salida. Si su llave es menor o igual que la del siguiente elemento en la secuencia de entrada, este elemento pertenece a la misma corrida y puede ponerse en su posición apropiada. En caso contrario, se reduce el tamaño del montón y se pone el nuevo elemento en el segundo, o sea el montón superior que se construye para que contenga la siguiente corrida. Indicamos el límite entre los dos montones con el índice L . Por tanto, el montón inferior (actual) consta de los elementos $H_1 \dots H_L$, el siguiente montón superior de $H_{L+1} \dots H_m$. Si $L = 0$, entonces cambiamos la salida y volvemos a poner L en m .
4. Ahora está agotada la fuente. Primero, ponemos R en m ; luego sacamos y reubicamos la parte inferior que termina la corrida actual y , al mismo tiempo, construimos la parte superior y gradualmente volvemos a ponerla en las posiciones $H_{L+1} \dots H_R$.
5. La última corrida se genera del resto de los elementos del montón.

Ahora estamos en condiciones de describir las cinco etapas en detalle como un programa completo, llamando a un procedimiento *select* cada vez que se detecta el final de una corrida y alguna acción para alterar el índice de la secuencia de salida ha de ser invocada. En el programa 2.17, se utiliza a cambio una rutina ficticia, la cual se limita a contar el número de corridas generadas. Todos los elementos se escriben en la secuencia f_1 .

Si ahora tratamos de integrar este programa, por ejemplo, con clasificación polifásica, nos topamos con una seria dificultad, debida a las siguientes circunstancias: el

programa de clasificación consta en su parte inicial de una rutina bastante complicada para cambiar entre variables de secuencia, y se basa en la disponibilidad de un procedimiento *copyrun* que proporciona exactamente una corrida al destino seleccionado. El programa *Heapsort*, en cambio, es una rutina compleja que se basa en la disponibilidad de un procedimiento cerrado *select*, el cual simplemente selecciona un nuevo destino. No habría problema si en uno o ambos programas el procedimiento requerido fuese llamado en un solo lugar; pero sucede que se les llama en diversos sitios de ambos programas.

La situación se refleja mejor por el uso de una *corrutina*; es conveniente en los casos en que coexisten varios procesos. El representante más típico es la combinación de un proceso que produce una corriente de información en sus entidades distintas y un proceso que consume esta corriente. Esta relación entre productor y consumidor puede expresarse en función de dos corrutinas; una de ellas bien pudiera ser el programa principal. La corrutina puede concebirse como un proceso que contiene uno o más puntos de ruptura. Si nos encontramos con uno de ellos, el control retornará al programa que ha activado la corrutina. Cada vez que ésta se llama nuevamente, la ejecución es reanudada en el punto de ruptura. En nuestro ejemplo, podríamos considerar la clasificación polifásica como el programa principal, llamando a *copyrun*, que se formula como una corrutina. Consta del cuerpo principal del programa 2.17 en el cual cada llamada de *select* representa ahora un punto de ruptura. La prueba del final de archivo habrá entonces de ser reemplazada sistemáticamente por una prueba de si la corrutina a alcanzado o no su punto final.

```
MODULE Distribute; (*para clasificacion por monton*)
  FROM InOut IMPORT WriteInt, WriteLn;
  FROM FS IMPORT File, Open, ReadWord, WriteWord, Reset, Close;
  CONST m = 16; mh = m DIV 2; (*tamaño del monton*)
  TYPE item = INTEGER;
  VAR L, R: CARDINAL;
    count: CARDINAL;
    x: item;
    H: ARRAY [1 .. m] OF item; (*montón*)
    f0, f1: File;

  PROCEDURE select;
  BEGIN count := count + 1 (*ficticio*)
  END select;

  PROCEDURE sift(L, R: CARDINAL);
    VAR i, j: CARDINAL; x: item;
    BEGIN i := L; j := 2*L; x := H[L];
    IF (j < R) & (H[j] > H[j+1]) THEN j := j+1 END ;
    WHILE (j <= R) & (x > H[j]) DO
      H[i] := H[j]; i := j; j := 2*j;
      IF (j < R) & (H[j] > H[j+1]) THEN j := j+1 END
    END ;
  END;
```

```
H[i] := x
END sift;

PROCEDURE OpenRandomSeq(VAR s: File; length, seed: INTEGER);
  VAR i: INTEGER;
BEGIN Open(s);
  FOR i := 0 TO length-1 DO
    WriteWord(s, seed); seed := (31*ficticio ) MOD 997 + 5
  END
END OpenRandomSeq;

PROCEDURE List(VAR s: File);
  VAR i, L: CARDINAL;
BEGIN Reset(s); i := 0; L := s.length;
  WHILE i < L DO
    WriteInt(INTEGER(s.a[i]), 6); i := i+1;
    IF i MOD 10 = 0 THEN WriteLn END
  END ;
  WriteLn
END List;

BEGIN count := 0; OpenRandomSeq(f0, 200, 991); List(f0);
  Open(f1); Reset(f0);
  select;
(*paso 1; llenar la mitad superior del monton *)
  L := m;
  REPEAT ReadWord(f0, H[L]); L := L-1
  UNTIL L = mh;
(*paso 2; llenar la mitad inferior del monton*)
  REPEAT ReadWord(f0, H[L]); sift(L,m); L := L-1
  UNTIL L = 0;
(*paso 3; pasar elementos a traves del monton*)
  L := m; ReadWord(f0, x);
  WHILE ~f0.eof DO
    WriteWord(f1, H[1]);
    IF H[1] <= x THEN
      (*x pertenece a la misma corrida*) H[1] := x; sift(1,L)
    ELSE (*comenzar siguiente corrida*)
      H[1] := H[L]; sift(1, L-1); H[L] := x;
      IF L <= mh THEN sift(L,m) END ;
      L := L-1;
      IF L = 0 THEN
        (*monton lleno; comenzar siguiente corrida*) L := m; select
      END
    END;
  ReadWord(f0, x)
```

```

END ;
(*paso 4; nivelar mitad inferior del monton*)
R := m;
REPEAT WriteWord(f1, H[1]);
  H[1] := H[L]; sift(1, L-1); H[L] := H[R]; R := R-1;
  IF L <= mh THEN sift(L,R) END ;
  L := L-1
UNTIL L = 0;
(*paso 5; nivelar mitad superior del monton*)
select;
WHILE R > 0 DO
  WriteWord(f1, H[1]); H[1] := H[R]; R := R-1; sift(1,R)
END ;
List(f1);
Close(f0); Close(f1)
END Distribute.

```

Programa 2.17 Distribución de corridas iniciales a través de un montón.

Análisis y conclusiones. ¿Qué rendimiento cabe esperar de una clasificación polifásica con su distribución inicial de corridas mediante una clasificación por montón? Primero examinaremos los mejoramientos que se logran al introducir el montón.

En una secuencia con llaves aleatoriamente distribuidas la longitud promedio esperada de corridas es 2. ¿Cuál es su longitud después de pasar la secuencia por un montón de tamaño m ? Estamos tentados a decir que m , pero por fortuna el resultado verdadero del análisis probabilístico es mucho mejor, a saber $2m$ (véase Knuth, vol. 3, p. 254). Por ello el factor del mejoramiento previsto es m .

Una estimación del rendimiento de la clasificación polifásica se obtiene de la tabla 2.15, el cual indica el número máximo de corridas que puede clasificarse en un determinado número de pasos parciales (niveles) con un determinado número N de secuencias. He aquí un ejemplo: con seis secuencias y un montón de tamaño $m = 100$, un archivo de hasta 165 680 100 corridas iniciales puede clasificarse sin rebasar 10 pasos parciales. Y éste es un rendimiento excelente.

Al repasar otra vez la combinación de clasificación polifásica y clasificación por montón, no podemos menos de sorprendernos ante la complejidad de este programa. Después de todo, ejecuta la misma tarea fácilmente definida de permutar un conjunto de elementos como lo hace uno de los programas de clasificación basados en los principios de la clasificación directa de arreglos. En esencia, el contenido de este capítulo puede compendiarse en dos puntos:

1. La íntima conexión entre el algoritmo y la estructura de datos utilizada, y en particular la influencia que ésta tiene en aquélla.
2. El refinamiento con que puede mejorarse el rendimiento de un programa, aun cuando la estructura disponible de sus datos (secuencia en vez de arreglo) no sea muy idónea para la tarea.

EJERCICIOS

- 2.1. ¿Cuál de los algoritmos dados en los programas 2.1 a 2.6, 2.8, 2.10 y 2.13 son métodos de clasificación estables?
- 2.2. ¿Funcionará correctamente el programa 2.2 si $L \leq R$ fuera sustituido por $L < R$ en la cláusula while? ¿Sería correcto si las proposiciones $R := m-1$ y $L := m+1$ fueran simplificadas a $L := m$ y $R := m$? De no ser así, encuentre conjuntos de valores $a_1 \dots a_n$ en los cuales el programa alterado fracasará.
- 2.3. Programe y mida el tiempo de ejecución de los tres métodos de clasificación directa en su computadora; obtenga los coeficientes por los cuales los factores C y M han de ser multiplicados para producir estimaciones del tiempo real.
- 2.4. Especifique invariantes de las repeticiones en los algoritmos de clasificación directa.
- 2.5. Examine la siguiente versión “obvia” del programa de partición 2.9 y encuentre conjuntos de valores $a_1 \dots a_n$ a los cuales no se aplica esta versión:

```

i := 1; j := n; x := a[n DIV 2];
REPEAT
  WHILE a[i] < x DO i := i+1 END ;
  WHILE x < a[j] DO j := j-1 END ;
  w := a[i]; a[i] := a[j]; a[j] := w
UNTIL i > j

```

- 2.6. Escriba un programa que combine los algoritmos de clasificación rápida y clasificación por burbuja en la forma siguiente: use el primer tipo de clasificación para obtener particiones (sin clasificar de longitud m ($1 \leq m \leq n$)); después recurra al segundo tipo de clasificación para terminar la tarea. Note que la clasificación rápida puede recorrer todo el arreglo de n elementos, por tanto, puede minimizar el esfuerzo de administración de datos. Encuentre el valor de m que minimiza el tiempo total de clasificación. Nota: sin duda el valor óptimo de m será muy pequeño. Por tanto, conviene dejar que la clasificación por burbuja recorra exactamente $m-1$ veces el arreglo en vez de incluir un último pase que establezca el hecho de que no se necesitan más intercambios.
- 2.7. Realice el mismo experimento que en el ejercicio 2.6 con una clasificación por selección directa y no con la clasificación por burbuja. Naturalmente, la clasificación por selección directa no puede recorrer el arreglo entero; de ahí que la cantidad prevista de manejo de índice sea un poco mayor.
- 2.8. Escriba un algoritmo de clasificación rápida conforme a la especificación de que la clasificación de la partición más corta se efectuará antes de clasificar la más larga. Efectúe la primera tarea mediante una proposición iterativa, la segunda apli-

cando una llamada recursiva. (Por tanto, su procedimiento de clasificación contendrá una llamada recursiva en vez de dos en el programa 2.10 y ninguna en el programa 2.11.)

- 2.9. Encuentre una permutación de las llaves $1, 2, \dots, n$ para la cual la clasificación rápida da su peor (mejor) rendimiento ($n = 5, 6, 8$).
- 2.10. Construya un programa de combinación natural, semejante al programa de combinación directa 2.13, operando sobre un arreglo de doble longitud partiendo de ambos extremos hacia adentro; compare su rendimiento con el del programa 2.13.
- 2.11. Note que en una combinación natural (de dos fases) no seleccionamos a ciegas el valor mínimo entre las llaves disponibles. Por el contrario, al encontrarse con el final de una corrida, nos limitamos a copiar el cabo de la otra corrida en la secuencia de salida. Por ejemplo, la combinación de

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

nos da la secuencia

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

en vez de

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

que parece estar mejor ordenado. ¿Cuál es la causa de esta estrategia?

- 2.12. Un método de clasificación semejante a la clasificación polifásica es la clasificación por combinación *en cascada* [2.1 y 2.9]. Se vale de un patrón distinto de combinación. Por ejemplo, si tenemos seis secuencias T_1, \dots, T_6 , dicha clasificación, comenzando también con una “distribución perfecta” de corridas en $T_1 \dots T_5$, realiza una combinación en cinco fases de $T_1 \dots T_5$ hasta que queda vacía T_5 , después (sin que intervenga T_6) una combinación en cuatro fases hasta T_5 , luego una combinación en tres fases hasta T_4 , una combinación en dos fases hasta T_3 y finalmente una operación de copiada de T_1 a T_2 . El siguiente paso opera en la misma forma comenzando con una combinación de cinco fases a T_1 y así sucesivamente. Pese a que este esquema parece inferior a la clasificación polifásica porque algunas veces opta por dejar ociosas algunas secuencias y porque contiene sencillas operaciones de copiada, nos sorprende su superioridad a la clasificación polifásica para archivos (muy) grandes y para seis o más secuencias. Escriba un programa bien estructurado para ilustrar el principio de combinación en cascada.

REFERENCIAS

- 2-1. B.K. Betz and Carter. *Proc. ACM National Conf.* 14 (1959), Paper 14.
- 2-2. R.W. Floyd. Treesort (Algorithms 113 and 243). *Comm. ACM*, 5, No. 8 (1962), 434, and *Comm. ACM*, 7, No. 12 (1964), 701.
- 2-3. R.L. Gilstad. Polyphase Merge Sorting - An Advanced Technique. *Proc. AFIPS Eastern Jt. Comp. Conf.*, 18 (1960), 143-48.
- 2-4. C.A.R. Hoare. Proof. of a Program: FIND. *Coom ACM*, 13, No. 1 (1970), 39-45.
- 2-5. ----- Proof of a Recursive Program: Quicksort. *Comp. J.*, J4, No. 4 (1971), 391-95.
- 2-6. ----- Quicksort. *Comp. J.*, 5, No. 1 (1962), 10-15.
- 2-7. D.E. Knuth. *The Art of Computer Programming*. Vol. 3 (Reading, Mass.: Addison-Wesley, 1973).
- 2-8. ----- *The Art of Computer Programming*. Vol. 3, pp. 86-95.
- 2-9. ----- *The Art of Computer Programming*. Vol. 3, p. 289.
- 2-10. H. Lorin. A Guided Bibliography to Sorting. *IBM Syst. J.*, 10, No. 3 (1971), 244-54.
- 2-11. D.L. Shell. A Highspeed Sorting Procedure. *Comm ACM*, 2, No. 7 (1959), 30-32.
- 2-12. R.C. Singleton. An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347). *Comm ACM*, 12, No. 3 (1969), 185.
- 2-13. M.H. Van Emden. Increasing the Efficiency of Quicksort (Algorithm 402). *Comm ACM*, 13, No. 9 (1970), 563-66, 693.
- 2-14. J.W.J. Williams. Heapsort (Algorithm 232). *Comm. ACM*, 7, No. 6 (1964), 347-48.

3. ALGORITMOS RECURSIVOS

3.1. INTRODUCCION

Se dice que un objeto es *recursivo*, si en parte está formado por sí mismo o se define en función de sí mismo. La recursión se encuentra no sólo en matemáticas, sino también en la vida diaria. ¿Quién no ha visto alguna vez una imagen publicitaria que se contiene a sí misma?

La recursión es una técnica especialmente eficaz en las definiciones matemáticas. Unos ejemplos muy conocidos son los números naturales, las estructuras de árbol y ciertas funciones:

1. Números naturales:
 - a) 0 es un número natural.
 - b) El sucesor de un número natural es otro número natural.



Fig. 3.1 Una imagen recursiva

2. Estructuras de árbol:

- a) 0 es un árbol (llamado árbol vacío).
- b) Si t_1 y t_2 son árboles, entonces las estructuras formadas por un nodo con dos árboles descendientes también son un árbol.

3. La función factorial $n!$ (para enteros no negativos):

- (a) $0! = 1$
- (b) $n > 0: n! = n * (n - 1)!$

El poder de la recursión radica, sin duda, en la posibilidad de definir un conjunto infinito de objetos mediante una proposición finita. De la misma manera, un número infinito de cálculos puede describirse con un programa recursivo finito, pese a que no contenga repeticiones explícitas. Sin embargo, los algoritmos recursivos son idóneos principalmente cuando el problema por resolver, la función por calcular o la estructura de datos por procesar ya están definidos en términos recursivos. En general, un programa recursivo P puede expresarse como una composición P de un conjunto de proposiciones S (que se contiene P) y P .

$$P \equiv P[S, P] \quad (3.1)$$

La herramienta necesaria y suficiente para expresar los programas recursivamente es el procedimiento o subrutina, pues permite dar a una proposición un nombre con el cual puede ser llamada. Si un procedimiento P contiene una referencia explícita a sí mismo, se dice que es *directamente recursivo*; si P contiene una referencia a otro procedimiento Q , que incluye una referencia (directa o indirecta) a P , entonces se dice que P es *indirectamente recursivo*. El uso de la recursión puede, pues, no ser evidente en el texto del programa.

Se acostumbra asociar un conjunto de objetos locales a un procedimiento, esto es, un conjunto de variables, constantes, tipos y procedimientos que se definen localmente a este procedimiento y que carecen de existencia o significado fuera de este procedimiento. Cada vez que ese procedimiento se activa de modo recursivo, se crea un nuevo conjunto de variables locales acotadas. Aunque tienen el mismo nombre que sus elementos correspondientes en el conjunto local al caso anterior del procedimiento, sus valores son específicos y se evita cualquier conflicto en la imposición de nombre por medio de las reglas de la cobertura (ámbito) de los identificadores: éstos siempre se refieren al conjunto de variables de creación más reciente. La misma regla se aplica a los parámetros de procedimiento, los cuales por definición están vinculados al procedimiento.

A semejanza de las proposiciones repetitivas, los procedimientos recursivos introducen la posibilidad de computaciones que no terminan y, con ello, también la necesidad de tener presente el problema de la terminación. Un requisito fundamental es, sin duda, que las llamadas recursivas de P estén sujetas a una condición B , que en un momento dado resulta falsa. El esquema de los algoritmos recursivos puede expresarse más exactamente en una de las dos formas siguientes:

$$P \equiv \text{IF } B \text{ THEN } P[S, P] \text{ END} \quad (3.2)$$

$$P \equiv P[S, \text{IF } B \text{ THEN } P \text{ END}] \quad (3.3)$$

En las repeticiones, la técnica básica de demostrar la terminación consiste en

1. Definir una función $f(x)$ (x será el conjunto de variables) tal que $f(x) \leq 0$ implica la condición de terminación (de la cláusula `while` o `repeat`) y
2. Probar que $f(x)$ disminuye durante cada paso de la repetición.

De la misma manera, la terminación de una recursión puede probarse, demostrando que cada ejecución de P disminuye un poco $f(x)$ y que $f(x) \leq 0$ significa $\sim B$. Una manera muy evidente de garantizar la terminación consiste en asociar un parámetro (valor), digamos n , con P y de llamar recursivamente P con $n-1$ como valor del parámetro. Sustituir $n > 0$ para B asegura entonces la terminación. Esto puede expresarse con los siguientes esquemas de programa:

$$P(n) \equiv \text{IF } n > 0 \text{ THEN } P[S, P(n-1)] \text{ END} \quad (3.4)$$

$$P(n) \equiv P[S, \text{IF } n > 0 \text{ THEN } P(n-1) \text{ END}] \quad (3.5)$$

En las aplicaciones prácticas es obligatorio demostrar que la profundidad final de la recursión no sólo es finita, sino que en realidad es muy *pequeña*. Ello se debe a que, luego de cada activación recursiva de un procedimiento P , cierta cantidad de memoria se necesita para alojar sus variables. Además de estas variables locales, el estado actual de la computación debe registrarse a fin de que sea recuperable cuando finalice la nueva activación de P y se reanude la anterior. Ya nos hemos topado con esta situación en el desarrollo del procedimiento *Quicksort* en el capítulo 2. Se descubrió que, al componer ingenuamente el programa a partir de una proposición que divide los n elementos en dos particiones y a partir de las dos llamadas recursivas que clasifican las dos particiones, la profundidad de la recursión puede aproximarse a n en el peor caso. Mediante una reevaluación más inteligente de la situación, fue posible limitar la profundidad a $\log n$. La diferencia entre n y $\log n$ es suficiente para convertir un caso muy inapropiado para la recursión en otro en el cual ésta resulta perfectamente práctica.

3.2. CUANDO NO UTILIZAR LA RECURSION

Los algoritmos recursivos son convenientes cuando el problema o los datos que deben manipularse están definidos en términos recursivos. Con todo, ello no significa que tales definiciones recursivas garanticen que un algoritmo recursivo es la mejor manera de resolver el problema. En efecto, la explicación del concepto del algoritmo recursivo mediante ejemplos inadecuados ha sido la causa principal de una inquietud general y antipatía por el uso de la recursión en la programación, al punto que la recursión se ha vuelto sinónimo de inefficiencia.

Los programas en los cuales hemos de abstenernos de usar la recursión algorítmica pueden caracterizarse como esquemas que exhiben el patrón de su composición. El esquema es el de (3.6) y, equivalentemente, de (3.7). Su característica es que hay sólo una llamada de P al final (o al inicio) de la composición.

$$P \equiv \text{IF } B \text{ THEN } S; P \text{ END} \quad (3.6)$$

$$P \equiv S; \text{IF } B \text{ THEN } P \text{ END} \quad (3.7)$$

Estos esquemas son naturales en los casos en que deben calcularse valores que están definidos en función de relaciones simples de recurrencia. Examinemos el conocido ejemplo de los números factoriales $f_i = i!$:

$$i = 0, 1, 2, 3, 4, 5, \dots \quad (3.8)$$

$$f_i = 1, 1, 2, 6, 24, 120, \dots$$

El primer número se define explícitamente como $f_0 = 1$, en tanto que los siguientes se definen recursivamente a partir de su predecesor:

$$f_{i+1} = (i+1) * f_i \quad (3.9)$$

La relación de recurrencia indica un algoritmo recursivo que calcula el n -ésimo número factorial. Si introducimos las dos variables I y F para denotar los valores i y f_i en el i -ésimo nivel de la recursión, encontramos que el cálculo necesario para pasar a los siguientes números en las secuencias (3.8) es

$$I := I + 1; F := I * F \quad (3.10)$$

y, al sustituir (3.10) para Σ en (3.6), obtenemos el programa recursivo

$$\begin{aligned} P &\equiv \text{IF } I < n \text{ THEN } I := I + 1; F := I * F; P \text{ END} \\ &I := 0; F := 1; P \end{aligned} \quad (3.11)$$

La primera linea de (3.11) se expresa en términos de nuestra notación ordinaria de programación, así:

```
PROCEDURE P;
BEGIN
  IF I < n THEN I := I + 1; F := I * F; P END
END P
```

(3.12)

Una forma de mayor uso, pero equivalente en lo esencial, es la que se da en (3.13). P es reemplazada por un *procedimiento de función*, es decir, el que tiene un valor resultante está asociado explícitamente y por lo mismo puede emplearse directamente como un constituyente de las expresiones. La variable F se torna superflua entonces; y el papel de I es asumido por el parámetro explícito del procedimiento.

```
PROCEDURE F(I: INTEGER): INTEGER;
BEGIN
  IF I > 0 THEN RETURN I * F(I - 1) ELSE RETURN 1 END
END F
```

(3.13)

Y ahora es evidente que en este ejemplo la recursión puede sustituirse sencillamente por la iteración. Esto lo expresa el programa

```
I := 0; F := 1;
WHILE I < n DO I := I + 1; F := I * F END
```

(3.14)

En general, los programas correspondientes a los esquemas (3.6) o (3.7) deben transcribirse en uno conforme al esquema (3.15)

```
P ≡ [x := x0; WHILE B DO S END]
```

(3.15)

También existen esquemas más complicados de composición recursiva, los cuales pueden y deben traducirse a una forma iterativa. Un ejemplo es el cálculo de los *números de Fibonacci*, los cuales se definen mediante la relación recurrente

$$\text{fib}_{n+1} = \text{fib}_n + \text{fib}_{n-1} \text{ cuando } n > 0 \quad (3.16)$$

y $\text{fib}_1 = 1$, $\text{fib}_0 = 0$. Una transcripción directa y sencilla da origen al programa recursivo

```
PROCEDURE Fib(n: INTEGER): INTEGER;
BEGIN
  IF n = 0 THEN RETURN 0
  ELSIF n = 1 THEN RETURN 1
  ELSE RETURN Fib(n-1) + Fib(n-2)
  END
END Fib
```

(3.17)

El cálculo de fib_n por una llamada Fib(n) hace que este procedimiento de la función sea activado en forma recursiva. ¿Con qué frecuencia? Advertimos que cada llamada con $n > 1$ lleva a 2 llamadas posteriores, o sea que el número total de llamadas crece exponencialmente (véase la figura 3.2). Ese programa resulta evidentemente impráctico.

Pero, por fortuna, los números de Fibonacci pueden calcularse por medio de un esquema iterativo que evita el recálculo de los mismos valores usando variables auxiliares tales que $x = \text{fib}_i$ y $y = \text{fib}_{i-1}$.

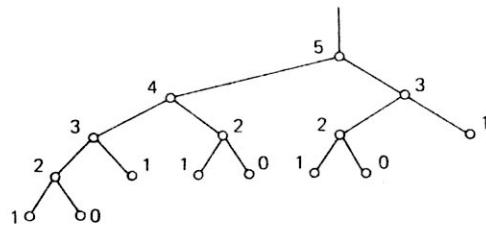


Fig. 3.2 Las 15 llamadas de Fib(5).

```
i := 1; x := 1; y := 0; (3.18)
WHILE i < n DO z := x; x := x + y; y := z; i := i + 1 END
```

Nota: Las asignaciones a x, y, z puede expresarse por medio de dos asignaciones sólo si no hay necesidad de una variable auxiliar: z: x := x + y; y := x - y.

Por tanto, la lección que debemos aprender de esto es no utilizar la recursión cuando la iteración ofrece una solución obvia. Sin embargo, esto no debe hacer que evitemos sistemáticamente el uso de la recursión. La recursión se presta a muchas aplicaciones, como se verá en los siguientes párrafos y capítulos. El hecho de que los procedimientos recursivos se realicen en máquinas esencialmente no recursivas demuestra que, en la práctica, todo programa recursivo puede transformarse en uno totalmente iterativo. Sin embargo, para ello se requiere el manejo explícito de una pila de recursión, y esas operaciones a menudo oscurecen la esencia de un programa, al grado que se torna difícil de entender. Esto significa que han de formularse como procedimientos recursivos los algoritmos que por su naturaleza son más bien recursivos que iterativos. A fin de captar mejor este punto, recomendamos al lector consultar el programa 2.10 y el 2.11, y que los compare.

El resto de este capítulo está dedicado al desarrollo de algunos programas recursivos en situaciones donde se justifica el uso de la recursión. También en el capítulo 4 se utiliza ampliamente esa técnica en los casos en que las estructuras de datos hacen obvia y natural la elección de soluciones recursivas.

3.3. DOS EJEMPLOS DE PROGRAMAS RECURSIVOS

El atractivo patrón gráfico de la figura 3.5 es la superposición de cinco curvas. Estas siguen un patrón regular e indican que pueden dibujarse mediante una pantalla o un graficador bajo el control de una computadora. Aquí nos proponemos descubrir el esquema de la recursión, a partir del cual puede construirse el programa de dibujo. La observación revela que tres de las curvas sobrepuertas tienen las formas que aparecen en la figura 3.3; las denotamos con H_1 , H_2 y H_3 . Las figuras muestran que H_{j+1} se obtiene por la composición de cuatro casos de H_j de medio tamaño y rotación apropiada; además hay que unir las cuatro H_j mediante tres líneas de conexión. Nótese que puede considerarse que H_1 consta de cuatro casos de una H_0 vacía unida por tres líneas rectas. H_j recibe el nombre de *curva Hilbert* de orden j , en honor de su inventor, el matemático D. Hilbert (1891).

Puesto que cada curva H_j está formada por cuatro copias de medio tamaño de H_{j-1} , el procedimiento lo expresamos dibujando H_j como una composición de cuatro llamadas para dibujar H_{j-1} en medio tamaño y con la rotación idónea. Para dar un ejemplo denotaremos las cuatro partes por A, B, C y D, y las rutinas con que se trazan las líneas de interconexión las representaremos con flechas que apunten a la dirección correspondiente. Se obtiene así el siguiente esquema de recursión (véase la figura 3.3).

A:	$D \leftarrow A \downarrow A \rightarrow B$	(3.19)
B:	$C \uparrow B \rightarrow B \downarrow A$	
C:	$B \rightarrow C \uparrow C \leftarrow D$	
D:	$A \downarrow D \leftarrow D \uparrow C$	

Supongamos que, para trazar los segmentos de línea, disponemos de un procedimiento *line* que mueve a una pluma de dibujo en determinada dirección por cierta distancia. Para nuestra comodidad, supondremos que la dirección está indicada por un parámetro entero i como $45i$ grados. Si la longitud de la línea unitaria se denota con u , el procedimiento correspondiente al esquema A se expresa fácilmente utilizando activaciones recursivas de los procedimientos B y D diseñados de modo análogo y de sí mismo.

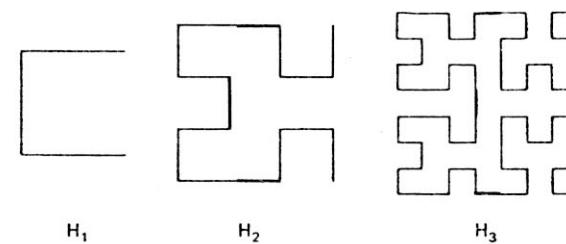


Fig. 3.3 Curvas de Hilbert de orden 1, 2 y 3.

```

PROCEDURE A(i: INTEGER);
BEGIN
  IF i > 0 THEN
    D(i-1); line(4, u);
    A(i-1); line(6, u);
    A(i-1); line(0, u);
    B(i-1)
  END
END A

```

(3.20)

Este procedimiento lo inicia el programa principal⁶, una vez para cada curva de Hilbert que debe sobreponerse. El programa determina el punto inicial de la curva, es decir, las coordenadas iniciales de la pluma denotada por Px y Py, así como el incremento unitario u. El cuadrado en el cual se dibujan las curvas está colocado en la mitad de la página, con determinada anchura y altura. Estos parámetros, lo mismo que la línea del procedimiento de dibujo se toman de un módulo llamado *LineDrawing*. El programa entero dibuja las n curvas de Hilbert $H_1 \dots H_n$ (véanse el programa 3.1 y la figura 3.4).

```

MODULE Hilbert;
FROM Terminal IMPORT Read;
FROM LineDrawing IMPORT width, height, Px, Py, clear, line;
CONST SquareSize = 512;
VAR i,x0,y0,u: CARDINAL; ch: CHAR;
PROCEDURE A(i: CARDINAL);
BEGIN
  IF i > 0 THEN
    D(i-1); line(4,u); A(i-1); line(6,u);
    A(i-1); line(0,u); B(i-1)
  END
END A;

PROCEDURE B(i: CARDINAL);
BEGIN
  IF i > 0 THEN
    C(i-1); line(2,u); B(i-1); line(0,u);
    B(i-1); line(6,u); A(i-1)
  END
END B;

PROCEDURE C(i: CARDINAL);
BEGIN
  IF i > 0 THEN
    B(i-1); line(0,u); C(i-1); line(2,u);
    C(i-1); line(4,u); D(i-1)
  END
END C;

```

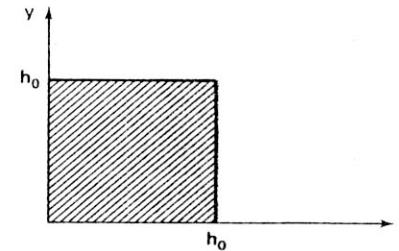


Fig. 3.4 El marco unitario.

```

PROCEDURE D(i: CARDINAL);
BEGIN
  IF i > 0 THEN
    A(i-1); line(6,u); D(i-1); line(4,u);
    D(i-1); line(2,u); C(i-1)
  END
END D;

BEGIN clear;
x0 := width DIV 2; y0 := height DIV 2;
u := SquareSize; i := 0;
REPEAT i := i+1; u := u DIV 2;
  x0 := x0 + (u DIV 2); y0 := y0 + (u DIV 2);
  Px := x0; Py := y0; A(i); Read(ch)
UNTIL (ch = 33C) OR (i = 6);
clear
END Hilbert.

```

Programa 3.1 Curvas de Hilbert.

Un ejemplo similar, aunque un poco más complejo y más elegante desde el punto de vista estético, se da en la figura 3.7. También este patrón se obtiene sobreponiendo varias curvas, dos de las cuales se muestran en la figura 3.6. Si se llama la *curva Sierpinski* de orden i. ¿Cuál es el esquema de recursión? Estamos tentados a escoger la hoja S_i como la estructura fundamental, posiblemente con un borde dejado fuera. Pero esto no nos lleva a una solución. La principal diferencia entre las curvas de Sierpinski y las de Hilbert radica en que las segundas están cerradas (sin entrecruzamientos). Ello significa que el esquema básico debe ser una curva abierta y que las cuatro partes están unidas por ligas que no pertenecen al patrón de recursión propiamente dicho. En efecto, esas ligas constan de cuatro líneas rectas en los cuatro ángulos más externos, dibujados con líneas más gruesas en la figura 3.6. Pueden considerarse como pertenecientes a una curva inicial no vacía S_0 , la cual es un cuadrado que está en un ángulo. Ahora ya podemos establecer fácilmente el esquema de la recursión. Los cuatro patrones constitutivos se denotan otra vez por A, B, C y D y las líneas de conexión se dibujan explícitamente. Ad-

viértase que los cuatro patrones son realmente idénticos salvo por las rotaciones de 90 grados.

El patrón básico de las curvas de Sierpinski es

$$S: \quad A \searrow B \swarrow C \nwarrow D \nearrow \quad (3.21)$$

y los patrones de recursión son (las flechas horizontales y verticales denotan líneas de doble longitud).

$$\begin{aligned} A: & \quad A \searrow B \rightarrow D \nearrow A \\ B: & \quad B \swarrow C \downarrow A \searrow B \\ C: & \quad C \nwarrow D \leftarrow B \swarrow C \\ D: & \quad D \nearrow A \uparrow C \nwarrow D \end{aligned} \quad (3.22)$$

Si usamos los mismos procedimientos primitivos de dibujo que en el ejemplo de la curva de Hilbert, el esquema anterior de la recursión se transforma sin dificultad en un algoritmo (directa e indirectamente) recursivo.

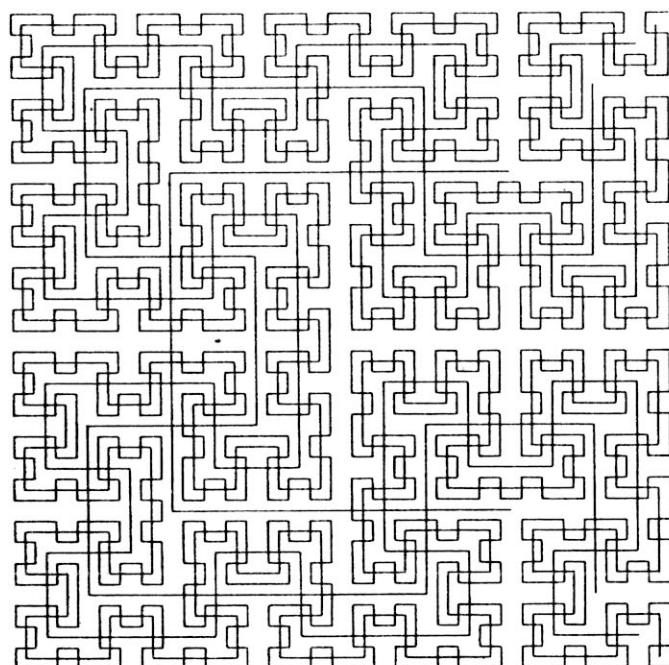


Fig. 3.5 Curvas de Hilbert $H_1 \dots H_5$.

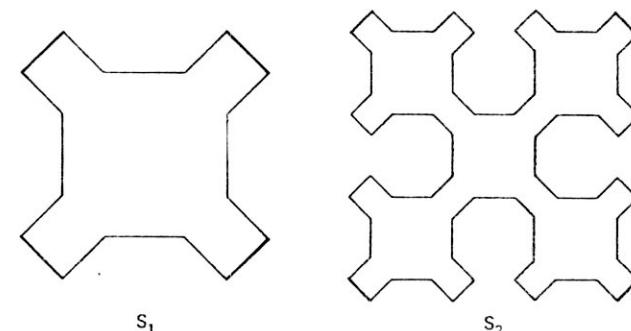


Fig. 3.6 Curvas Sierpinski de los órdenes 1 y 2.

```
PROCEDURE A(k: INTEGER);
BEGIN
  IF k > 0 THEN
    A(k-1); line(7, h); B(k-1); line(0, 2*h);
    D(k-1); line(1, h); A(k-1)
  END
END A
```

(3.23)

Este procedimiento se deriva de la primera línea del esquema de recursión (3.22). Los procedimientos correspondientes a los patrones B, C y D se derivan de modo análogo. El programa principal se compone de acuerdo con el patrón (3.21). Su tarea es establecer los valores iniciales de las coordenadas del dibujo y determinar la longitud unitaria de línea h conforme al tamaño del plano, como se advierte en el programa 3.2. El resultado de la ejecución de este programa con $n = 4$ se muestra en la figura 3.7.

La elegancia del uso de la recursión en este ejemplo es evidente y convincente. La corrección de los programas puede deducirse fácilmente de su estructura y patrones de composición. Más aún, el uso de un parámetro explícito de nivel según el esquema (3.5) garantiza la terminación, puesto que la profundidad de la recursión no puede ser mayor que n . En contraste con esta formulación recursiva, los programas equivalentes que prescinden del uso explícito de la recursión son en extremo difíciles y oscuros. Si el lector trata de entender los programas mostrados en [3-3] se convencerá de ello.

```
MODULE Sierpinski;
  FROM Terminal IMPORT Read;
  FROM LineDrawing IMPORT width, height, Px, Py, clear, line;
  CONST SquareSize = 512;
  VAR i,h,x0,y0: CARDINAL; ch: CHAR;
  PROCEDURE A(k: CARDINAL);
  BEGIN
    IF k > 0 THEN
```

```

A(k-1); line(7, h); B(k-1); line(0, 2*h);
D(k-1); line(1, h); A(k-1)
END
END A;

PROCEDURE B(k: CARDINAL);
BEGIN
IF k > 0 THEN
  B(k-1); line(5, h); C(k-1); line(6, 2*h);
  A(k-1); line(7, h); B(k-1)
END
END B;

PROCEDURE C(k: CARDINAL);
BEGIN
IF k > 0 THEN
  C(k-1); line(3, h); D(k-1); line(4, 2*h);
  B(k-1); line(5, h); C(k-1)
END
END C;

```

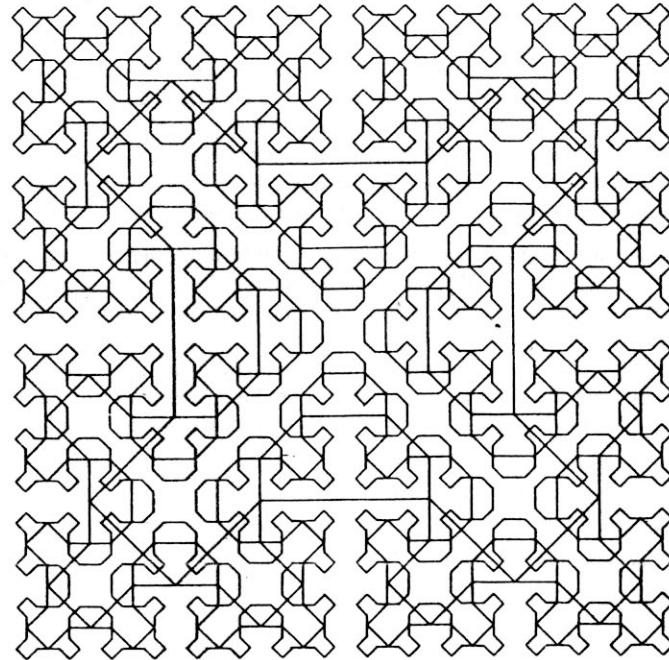


Fig. 3.7 Curvas Sierpinski $S_1 \dots S_4$.

```

PROCEDURE D(k: CARDINAL);
BEGIN
IF k > 0 THEN
  D(k-1); line(1, h); A(k-1); line(2, 2*h);
  C(k-1); line(3, h); D(k-1)
END
END D;

BEGIN clear; i := 0; h := SquareSize DIV 4;;
x0 := CARDINAL(width) DIV 2; y0 := CARDINAL(height) DIV 2 + h;
REPEAT i := i+1; x0 := x0-h;
  h := h DIV 2; y0 := y0+h; Px := x0; Py := y0;
  A(i); line(7,h); B(i); line(5,h);
  C(i); line(3,h); D(i); line(1,h); Read(ch)
UNTIL (i = 6) OR (ch = 33C);
clear
END Sierpinski.

```

Programa 3.2 Curvas de Sierpinski.

3.4. ALGORITMOS DE RASTREO INVERSO

Un aspecto particularmente interesante es el tema de la *solución general de problemas*. La tarea consiste en determinar los algoritmos para encontrar las soluciones de problemas específicos sin seguir una regla fija de cálculo sino por ensayo y error (tanteo). El patrón común consiste en descomponer el proceso de *tanteo* en tareas parciales. A menudo éstas se expresan espontáneamente en términos recursivos y consisten en explorar un número finito de subtareas. Generalmente vemos el proceso entero como un proceso de ensayo o búsqueda que poco a poco construye y rastrea (poda) un árbol de subtareas. En algunos problemas el árbol de búsqueda crece muy rápidamente, a menudo de modo exponencial, según un parámetro determinado. Y la búsqueda aumenta con base en eso. Muchas veces el árbol de búsqueda puede rastrearse aplicando sólo la heurística, con lo cual se reduce el cálculo a límites tolerables.

En este libro no tenemos la intención de explicar las reglas generales de la heurística. Por el contrario, el principio general de dividir esas tareas de solución de tareas en subtareas y la aplicación de la recursión será el tema de este capítulo. Empezaremos por demostrar la técnica subyacente recurriendo a un ejemplo, a saber, el conocido *recorrido del caballo*.

Tenemos un tablero $n \times n$ con n^2 campos. Un caballo (a quien se le permite moverse conforme a las reglas del ajedrez) se pone en el campo con las coordenadas iniciales x_0, y_0 . El problema radica en encontrar una cobertura de todo el tablero (si es que existe), o sea calcular un recorrido de $n^2 - 1$ movimientos (jugadas) tales que cada campo del tablero sea visitado exactamente una vez.

La manera obvia de reducir el problema de abarcar n^2 campos consiste en considerar el problema de realizar una jugada siguiente o descubrir que ninguna es posible. Por consiguiente, definamos un algoritmo que trate de hacer una jugada siguiente. Una primera aproximación se observa en (3.24).

```

PROCEDURE TryNextMove; (3.24)
BEGIN inicializar la selección de jugadas;
  REPEAT seleccionar siguiente candidato de la lista de jugadas siguientes;
    IF aceptable THEN
      registrar jugada
      IF tablero no lleno THEN
        TryNextMove;
        IF sin éxito THEN borrar registro anterior END
      END
    END
  UNTIL (la jugada fue exitosa) OR (no más candidatos)
END TryNextMove

```

Si deseamos ser más precisos al describir este algoritmo, estamos obligados a tomar algunas decisiones sobre la representación de datos. Un paso evidente consiste en representar el tablero con una matriz, digamos h . Introduzcamos también un tipo para denotar los valores índice:

```

TYPE index = [1 .. n]; (3.25)
VAR h: ARRAY index, index OF INTEGER

```

La decisión de representar cada campo del tablero con un entero y no con un valor booleano que denote la ocupación se debe a que deseamos llevar un control de la historia de las ocupaciones sucesivas del tablero. La siguiente convención es una elección obvia:

```

h[x,y] = 0: el campo <x,y> no ha sido visitado
h[x,y] = i: el campo <x,y> ha sido visitado en el i-esimo movimiento (3.26)

```

La siguiente decisión se refiere a la elección de los parámetros idóneos. Debe determinar las condiciones iniciales de la siguiente jugada y también dar a conocer su éxito. La primera tarea se resuelve adecuadamente especificando las coordenadas x, y , a partir de las cuales se debe hacer la jugada, y especificando el número i de la jugada (con fines de registro). La segunda tarea requiere un parámetro de resultados booleanos con el significado *la jugada fue exitosa*.

¿Cuáles proposiciones pueden refinarse ahora con base en estas decisiones? Sin duda *tablero no lleno* puede expresarse como $i < n^2$. Más aún, si introducimos dos variables locales u y v que representan las coordenadas de posibles destinos de jugada, determinados a partir de un patrón de salto de caballos, entonces el predicado *aceptable* puede expresarse como la unión lógica de las condiciones de que el nuevo campo se encuentre en el tablero, esto es, $1 \leq u \leq n$ y $1 \leq v \leq n$ y de que el campo no haya sido visitado con anterioridad, es decir, $h_{uv} = 0$.

La operación de registrar la jugada legal se expresa mediante la asignación $h_{uv} := i$ y la cancelación de este registro de información se expresa como $h_{uv} := 0$. Si una variable local $q1$ se introduce y se utiliza como el parámetro del resultado en la llamada recursiva de este algoritmo, $q1$ puede sustituir a *la jugada fue exitosa*.

```

PROCEDURE Try(i: INTEGER; x, y: index; VAR q: BOOLEAN);
  VAR u, v: INTEGER; q1: BOOLEAN;
  BEGIN inicializar selección de jugadas;
    REPEAT sean <u, v> las coordenadas de la siguiente jugada definidas
      por las reglas del ajedrez;
      IF  $(1 \leq u \leq n) \& (1 \leq v \leq n) \& (h[u,v] = 0)$  THEN
        h[u,v] := i;
        IF  $i < n^2$  THEN Try(i+1, u, v, q1);
        IF  $\sim q1$  THEN h[u,v] := 0 ELSE q1 := TRUE END
      END
    END
  END

```

```

UNTIL q1 OR no más candidatos;
q := q1
END Try

```

Un paso más de refinamiento nos llevará a un programa expresado enteramente en función de nuestra notación básica de programación. Hemos de advertir que hasta ahora el programa se desarrolló de manera totalmente independiente de las reglas que rigen los saltos del caballo. A propósito hemos pospuesto el examen de los detalles del problema. Pero ahora es el momento de examinarlos.

Si tenemos un par de coordenadas iniciales x, y , hay ocho posibles candidatos, u, v del destino. Se numera de 1 a 8 en la figura 3.8. Un simple método de obtener u, v a partir de x, y consiste en sumar las diferencias de las coordenadas guardadas en un arreglo de los pares de la diferencia o en dos arreglos de diferencias individuales. Supongamos que esos arreglos están denotados por dx y dy , debidamente inicializados. Entonces puede usarse un índice k para numerar el *siguiente candidato*. Los detalles aparecen en el programa 3.3. El procedimiento recursivo se inicia por una llamada con las coordenadas x_0, y_0 de ese campo como parámetros de donde se inicia el recorrido. A este campo se le debe asignar un valor; todos los otros deben marcarse como libres.

No ha de omitirse un detalle más: una variable h_{uv} existe sólo si tanto u como v se encuentran en el intervalo índice 1 ... n . En consecuencia, la expresión en (3.27), que sustituyó a *aceptable* en (3.24), es válida sólo si sus cuatro términos constituyentes son verdaderos. Por tanto, es importante que el término $h_{uv} = 0$ aparezca al último. La tabla 3.1 indica las soluciones obtenidas con las posiciones iniciales $<3,3>$, $<2,4>$ para $n = 5$ y $<1,1>$ para $n = 6$.

```

MODULE KnightsTour;
FROM InOut IMPORT
  ReadInt, Done, WriteInt, WriteString, WriteLn;
VAR i, j, n, Nsqr: INTEGER; q: BOOLEAN;
  dx, dy: ARRAY [1 .. 8] OF INTEGER;
  h: ARRAY [1 .. 8], [1 .. 8] OF INTEGER;
  h[x0, y0]:= 1; try(2, x0, y0, q)

```

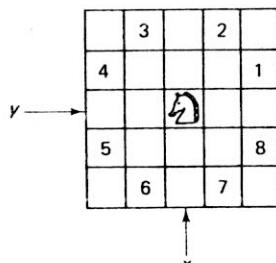


Fig. 3.8 Las ocho posibles jugadas de un caballo.

¿Qué abstracciones es posible hacer en este ejemplo? ¿Qué patrón exhibe que sea típico de esta clase de algoritmos para la solución de problemas? ¿Qué nos enseña? El rasgo distintivo es que los pasos hacia la solución total se intentan y registran; más tarde se vuelven a tomar y se borran en los registros cuando se descubre que el paso no puede

```

PROCEDURE Try(i, x, y: INTEGER; VAR q: BOOLEAN);
  VAR k, u, v: INTEGER; q1: BOOLEAN;
BEGIN k := 0;
  REPEAT k := k+1; q1 := FALSE;
    u := x + dx[k]; v := y + dy[k];
    IF (1 <= u) & (u <= n) & (1 <= v) & (v <= n) & (h[u,v] = 0) THEN
      h[u,v] := i;
      IF i < Nsqr THEN Try(i+1, u, v, q1);
        IF ~q1 THEN h[u,v] := 0 END
      ELSE q1 := TRUE
      END
    END
  UNTIL q1 OR (k = 8);
  q := q1
END Try;

BEGIN
  a[1]:= -2; a[2]:= 1; a[3]:= -1; a[4]:= -2;
  a[5]:= -2; a[6]:= -1; a[7]:= 1; a[8]:= 2;
  b[1]:= 1; b[2]:= 2; b[3]:= 2; b[4]:= 1;
  b[5]:= -1; b[6]:= -2; b[7]:= -2; b[8]:= -1;
  LOOP ReadInt(n);
  IF ~Done THEN EXIT END ;
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO h[i,j] := 0 END
  END ;
  ReadInt(i); ReadInt(j); WriteLn;
  Nsqr := n*n; h[1,1]:= 1; Try(2, i, j, q);
  IF q THEN
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO WriteInt(h[i,j], 5) END
      WriteLn
    END
    ELSE WriteString("sin trayectoria")
    END
  END
END KnightsTour.

```

Programa 3.3 Recorrido del rey.

23	10	15	4	25		23	4	9	14	25
16	5	24	9	14		10	15	24	1	8
11	22	1	18	3		5	22	3	18	13
6	17	20	13	8		16	11	20	7	2
21	12	7	2	19		21	6	17	12	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Tabla 3.1 Tres recorridos del caballo.

llevar a la solución total o que lleva a un callejón sin salida. A esta acción se le llama *rastreo inverso*. El patrón general (3.28) se deriva de (3.24), suponiendo que el número de candidatos potenciales en cada paso sea finito.

```

PROCEDURE Try;
BEGIN inicializar selección de coordenadas;
REPEAT seleccionar siguientes; (3.28)
  IF aceptable THEN
    registrarlo;
    IF solución incompleta THEN Try;
      IF no exitosa THEN cancelar registro END
    END
  END
UNTIL exitosa OR no más candidatos
END Try

```

Desde luego, los programas reales asumen varias formas derivadas del esquema (3.28). Un patrón con el que nos encontramos a menudo se sirve de un parámetro explícito de nivel, el cual indica la profundidad de la recursión y tiene en cuenta una sencilla condición de terminación. Si además en cada paso el número de candidatos por investigar es fijo, digamos m, entonces se aplica el esquema derivado (3.29); se invoca mediante la proposición Try (l).

```

PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN k := 0;
REPEAT k := k+1; seleccionar k-ésimo candidato;
  IF aceptable THEN
    record it;
    IF i < n THEN Try(i+1);
    IF sin éxito THEN cancelar registro END
  END
END Try

```

```

  END
  END
UNTIL exitosa OR (k = m)
END Try

```

El resto de este capítulo se dedica al tratamiento de tres ejemplos más. Los tres muestran varias encarnaciones del esquema abstracto (3.29) y se incluyen como ilustraciones más completas del uso adecuado de la recursión.

3.5. EL PROBLEMA DE LAS OCHO REINAS

Este problema es un conocido ejemplo del uso de los métodos de tanteo y de los algoritmos de rastreo inverso. Lo investigó C. F. Gauss en 1850, pero sin resolverlo por completo. Ello no debe causarnos sorpresa. Después de todo, la propiedad especial de estos problemas es que desafían la solución analítica. Por el contrario, requieren grandes cantidades de trabajo exhaustivo, paciencia y exactitud. De ahí que esos algoritmos hayan adquirido importancia casi exclusivamente gracias a la computadora automática, la cual posee esas propiedades en un grado mucho más alto que las personas en general e incluso que los genios.

El problema de las ocho reinas se formula en los siguientes términos (véase también [3-4]): hay que colocar ocho reinas en un tablero de ajedrez en una forma tal que ninguna reina “se coma” a otra. Aplicando como plantilla el esquema (3.29), nos es fácil obtener la siguiente versión rudimentaria de una solución:

```

PROCEDURE Try(i: INTEGER);
BEGIN
  inicializar selección de posiciones para la i-ésima reina;
  REPEAT hacer siguiente selección;
    IF seguro THEN ColocaReina
      IF i < 8 THEN Try(i+1);
      IF sin éxito THEN EliminaReina END
    END
  UNTIL exitosa OR no más posiciones
END Try

```

A fin de empezar la solución del problema, es preciso tomar algunas opciones respecto a la representación de datos. Sabemos por las reglas de ajedrez que una reina “se come” a todas las otras piezas que están en la misma columna, renglón o diagonal del tablero, por lo cual deducimos que cada columna puede contener una sola reina y que la elección de una posición para la *i*-ésima reina puede limitarse a la *i*-ésima columna. El parámetro *i* se convierte, pues, en el índice de la columna y el proceso de selección de las posiciones se extiende a lo largo de los ocho posibles valores de un índice de renglón *j*.

Queda por resolver la cuestión de representar las ocho reinas sobre el tablero. Una elección obvia sería otra vez una matriz cuadrada para representar el tablero, pero con un poco de observación nos damos cuenta de que esa representación llevaría a operaciones bastante arduas para verificar la disponibilidad de las posiciones. Ello es muy inconveniente puesto que es la operación que se efectúa con mayor frecuencia. Por tanto, hemos de escoger una representación de datos que simplifique al máximo la comprobación. El mejor método consiste en representar lo más directamente posible la información

que en realidad sea relevante y de mayor uso. En nuestro caso ésa no es la posición de las reinas, sino el hecho de que una reina haya sido colocada ya a lo largo de cada renglón y diagonales. (Ya sabemos que exactamente una se pone en cada columna *k* para $1 \leq k \leq i$.) Esto nos lleva a la siguiente elección de variables:

```

VAR x: ARRAY [1 .. 8] OF INTEGER;
a: ARRAY [1 .. 8] OF BOOLEAN;
b: ARRAY [b1 .. b2] OF BOOLEAN;
c: ARRAY [c1 .. c2] OF BOOLEAN;

```

(3.31)

donde

x_i denota la posición de la reina en la *i*-ésima columna;
 a_j significa “ninguna reina se encuentra en el renglón *j*-ésimo”;
 b_k significa “ninguna reina ocupa la *k*-ésima diagonal -/”;
 c_k significa “ninguna reina ocupa la *k*-ésima diagonal -\”.

La elección de los índices límite *b1*, *b2*, *c1*, *c2* depende de la manera en que se calculen los índices *b* y *c*; denotamos que en una diagonal -/ todos los campos tienen la misma suma de sus coordenadas *i* y *j*, y que en diagonal -\ las diferencias de coordenadas *i*-*j* son constantes. Las soluciones correspondientes se muestran en el programa 3.4. Con estos datos, la proposición *ColocaReina* se transforma en

```

x[i] := j; a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE

```

(3.32)

la proposición *EliminaReina* se refina a

```

a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE

```

(3.33)

y la condición *safe* se cumple si el campo $\langle i, j \rangle$ está en un renglón y en diagonales que todavía están libres. Así pues, esto puede indicarse por la expresión lógica

```

a[j] & b[i+j] & c[i-j]

```

(3.34)

Y con esto se termina el desarrollo de este algoritmo, que se muestra íntegramente como programa 3.4. La solución calculada es $x = (1, 5, 8, 6, 3, 7, 2, 4)$ y aparece en la figura 3.9.

```

MODULE Queens;
  FROM InOut IMPORT WriteInt, WriteLn;
  VAR i: INTEGER; q: BOOLEAN;
  a: ARRAY [1 .. 8] OF BOOLEAN;
  b: ARRAY [2 .. 16] OF BOOLEAN;
  c: ARRAY [-7 .. 7] OF BOOLEAN;
  x: ARRAY [1 .. 8] OF INTEGER;

  PROCEDURE Try(i: INTEGER; VAR q: BOOLEAN);
    VAR j: INTEGER;
    BEGIN j := 0;
    REPEAT j := j+1; q := FALSE;

```

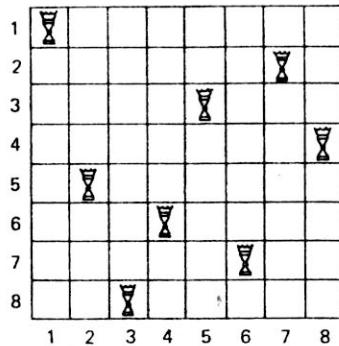


Fig. 3.9 Una solución al problema de las ocho reinas.

```

IF a[j] & b[i+j] & c[i-j] THEN
  x[i] := j;
  a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
  IF i < 8 THEN
    Try(i+1, q);
    IF ~q THEN
      a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
    END
    ELSE q := TRUE
    END
  END
  UNTIL q OR (j = 8)
END Try;
```

```

BEGIN
  FOR i := 1 TO 8 DO a[i] := TRUE END ;
  FOR i := 2 TO 16 DO b[i] := TRUE END ;
  FOR i := -7 TO 7 DO c[i] := TRUE END ;
  Try(1,q);
  FOR i := 1 TO 8 DO WriteInt(x[i], 4) END ;
  WriteLn
END Queens.
```

Programa 3.4 Ocho reinas.

Antes de abandonar el contexto del tablero de ajedrez, el ejemplo de las ocho reinas servirá como una ilustración de una importante extensión del algoritmo de tanteo. En términos generales, la extensión consiste en encontrar no sólo una solución del problema sino *todas*.

La extensión se logró con facilidad. Debemos recordar el hecho de que la generación de candidatos ha de efectuarse de un modo sistemático que garantiza que ningún candidato sea generado más de una vez. Esta propiedad del algoritmo corresponde a la búsqueda del árbol de candidatos en una forma sistemática en la cual cada nodo se visita exactamente una vez. Y permita, una vez encontrada y registrada debidamente una solución, pasar de inmediato al siguiente candidato producido por el proceso de selección sistemática. El esquema general se obtiene de (3.29) y se advierte en (3.35).

```

PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN
  FOR k := 1 TO m DO
    seleccionar k-ésimo candidato;
    IF aceptable THEN registrarlo
      IF i < n THEN Try(i+1) ELSE imprimir solución END ;
      cancelar registro
    END
  END
END Try;
```

(3.35)

Nótese que, dada la simplificación de la condición de terminación en el proceso de selección al término individual $k = m$, la proposición repeat se reemplaza correctamente con una proposición for. Nos causa sorpresa que la búsqueda de todas las soluciones posibles se logra con un programa más simple que la búsqueda de una sola solución.

El algoritmo extendido para determinar las 92 soluciones del problema de las ocho reinas se muestra en el programa 3.5. En realidad, hay sólo 12 soluciones significativamente diferentes; nuestro programa reconoce las simetrías. Las 12 soluciones generadas se enumeran primero en la tabla 3.2. Los números n a la derecha indican la frecuencia de ejecución de la prueba para los campos “seguros”. Su promedio en las 92 soluciones es 161.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	n
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	072
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Tabla 3.2 Doce soluciones al problema de las ocho reinas.

```

MODULE AllQueens;
FROM InOut IMPORT WriteInt, WriteLn;

VAR i: INTEGER;
a: ARRAY [1 .. 8] OF BOOLEAN;
b: ARRAY [2 .. 16] OF BOOLEAN;
c: ARRAY [-7 .. 7] OF BOOLEAN;
x: ARRAY [1 .. 8] OF INTEGER;

PROCEDURE print;
VAR k: INTEGER;
BEGIN
FOR k := 1 TO 8 DO WriteInt(x[k], 4) END ;
WriteLn
END print;

PROCEDURE Try(i: INTEGER);
VAR j: INTEGER;
BEGIN
FOR j := 1 TO 8 DO
IF a[j] & b[i+j] & c[i-j] THEN
x[i] := j;
a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
IF i < 8 THEN Try(i+1) ELSE print END ;
a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
END
END
END Try;

BEGIN
FOR i := 1 TO 8 DO a[i] := TRUE END ;
FOR i := 2 TO 16 DO b[i] := TRUE END ;
FOR i := -7 TO 7 DO c[i] := TRUE END ;
Try(1)
END AllQueens.

```

Programa 3.5 Ocho reinas.

3.6. EL PROBLEMA DEL MATRIMONIO ESTABLE

Supongamos que dos conjuntos disjuntos (ajenos) A y B sean de igual tamaño. Encuentremos un conjunto de n pares $\langle a, b \rangle$ tal que a en A y b en B satisfagan algunas restricciones. Se cuenta con muchos criterios para tales pares; uno de ellos es la regla denominada *regla del matrimonio estable*.

Supongamos que A es un conjunto de varones y B es un conjunto de mujeres. Cada hombre y cada mujer tienen preferencias especiales por su posible consorte. Si se escogen n parejas tales que haya un hombre y una mujer que no estén casados pero que se prefieren entre sí y no a la persona con que van a casarse, entonces la asignación será inestable. Si no existe tal par, la asignación se llama *estable*. Esta situación caracteriza a muchos problemas afines en los cuales es preciso hacer asignaciones conforme a preferencias como, por ejemplo, la elección de escuela para estudiantes, la selección de reclutas para diversas ramas de los servicios militares, etc. El ejemplo del matrimonio es muy revelador; pero nótese que la lista de preferencias es invariable y no cambia luego de realizar una asignación particular. Tal suposición simplifica el problema, pero también representa una grave distorsión de la realidad (llamada abstracción).

Una manera de buscar una solución consiste en tratar de formar parejas entre los miembros de los dos conjuntos, uno tras otro, hasta que se agoten los dos conjuntos. Si queremos encontrar todas las asignaciones estables, podemos bosquejar una solución empleando el esquema del programa (3.35) como plantilla. Supongamos que *Try(m)* denota el algoritmo para encontrar la esposa de un hombre m y hagamos que la búsqueda se realice en el orden de la lista de preferencias manifestadas por el hombre. La primera versión basada en estas suposiciones es (3.36).

```

PROCEDURE Try(m: man);
VAR r: rank;
BEGIN
FOR r := 1 TO n DO
  seleccionar la r-ésima preferencia del hombre m;
  IF acceptable THEN registrar el matrimonio
    IF m no es el último hombre THEN Try(successor(m))
    ELSE registrar el conjunto estable
  END;
  cancelar el matrimonio
END
END
END Try

```

(3.36)

También en este caso hemos llegado al punto donde no podemos proseguir sin tomar antes algunas decisiones respecto a la representación de los datos. Introducimos tres tipos escalares y, por razones de simplicidad, su designación mediante nombres específicos

mejora mucho la claridad. En particular, se puede indicar más fácilmente lo que representa una variable.

```
TYPE man = [1 .. n];
woman = [1 .. n];
rank = [1 .. n]
```

(3.37)

Los datos iniciales se representan con dos matrices que indican las preferencias de los hombres y las mujeres.

```
VAR wmr: ARRAY man, rank OF woman
mwr: ARRAY woman, rank OF man
```

(3.38)

Así pues, wmr_m denota la lista de preferencias del hombre m , esto es, $wmr_{m,r}$ es la mujer que ocupa el r -ésimo rango en la lista del hombre m . De manera análoga, mwr_w es la lista de preferencias de la mujer w y $mwr_{w,r}$ es su r -ésima elección. Un conjunto muestra de datos se incluye en la tabla 3.3.

El resultado se representa con un arreglo de mujeres x , tal que x_m denote a la compañera del hombre m . A fin de mantener la simetría entre hombres y mujeres, se introduce un arreglo más y , tal que y_w denote al compañero de la mujer w .

```
VAR x: ARRAY man OF woman;
y: ARRAY woman OF man
```

(3.39)

En realidad, el arreglo y es redundante pues representa información con que ya se cuenta gracias a la existencia de x . En efecto, las relaciones

$$x_{yw} = w, y_{xm} = m \quad (3.40)$$

se aplican a todos los m y w que estén casados. Por consiguiente, el valor y_w puede determinarse por una simple búsqueda de x ; pero el arreglo y mejora notablemente la eficiencia del algoritmo. La información representada por x y y se necesita para determinar la estabilidad de un conjunto propuesto de matrimonios. Dado que ese conjunto se construye gradualmente al casar a las personas y probar la estabilidad tras cada matrimonio propuesto, x y y se requieren aun antes de definir todos sus componentes. A fin de llevar un control de los componentes definidos, introducimos los arreglos booleanos

$r =$	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8
$m = 1$	7 2 6 5 1 3 8 4	w = 1 4 6 2 5 8 1 3 7
2	4 3 2 6 8 1 7 5	2 8 5 3 1 6 7 4 2
3	3 2 4 1 8 5 7 6	3 6 8 1 2 3 4 7 5
4	3 8 4 2 5 6 7 1	4 3 2 4 7 6 8 5 1
5	8 3 4 5 6 1 7 2	5 6 3 1 4 5 7 2 8
6	8 7 5 2 4 3 1 6	6 2 1 3 8 7 4 6 5
7	2 4 6 3 1 7 5 8	7 3 5 7 2 4 1 8 6
8	6 1 4 2 7 5 3 8	8 7 2 8 4 5 6 3 1

Tabla 3.3 Datos muestra de entrada para wmr y mwr .

```
singlem: ARRAY man OF BOOLEAN
singlew: ARRAY woman OF BOOLEAN
```

(3.41)

lo cual significa que $singlem_m$ implica que x_m está definido y $singlew_w$ implica que y_w está definido. Sin embargo, basta observar el algoritmo propuesto para darse cuenta de que el estado civil de un hombre se determina mediante el valor m a través de la relación

$$\sim singlem[k] = k < m \quad (3.42)$$

Lo anterior sugiere que el arreglo $singlem$ puede omitirse; de ahí que simplifiquemos el nombre $singlew$ a $single$. Estas convenciones nos llevan al refinamiento mostrado en (3.4). El predicado *acceptable* puede refinarse en la unión de *single* y *stable*, donde *stable* es una función que todavía debe refinarse más.

```
PROCEDURE Try(m: man);
  VAR r: rank; w: woman;
  BEGIN
    FOR r := 1 TO n DO
      w := wmr[m,r];
      IF single[w] & stable THEN
        x[m] := w; y[w] := m; single[w] := FALSE;
        IF m < n THEN Try(successor(m)) ELSE record set END ;
        single[w] := TRUE
      END
    END
  END Try
```

(3.43)

En este momento todavía es perceptible la gran semejanza de esta solución con el programa 3.5. La tarea decisiva consiste ahora en refinar el algoritmo para determinar la estabilidad. Por desgracia es imposible representar la estabilidad por una expresión tan simple como la seguridad de la posición de una reina en el programa 3.5. El primer detalle que es preciso tener presente consiste en que la estabilidad se obtiene, por definición, de las comparaciones de rangos. Los rangos de hombres y mujeres no están disponibles en ninguna parte dentro de nuestra colección de datos establecidos hasta ahora. Sin duda puede calcularse el rango de la mujer w en la mente del hombre m , pero con una costosa búsqueda de w en wmr_m . Dado que el cálculo de la estabilidad es una operación muy común, conviene hacer que esta información sea más directamente accesible. Para ello introducimos las dos matrices

```
rmw: ARRAY man, woman OF rank;
rwm: ARRAY woman, man OF rank
```

(3.44)

tales que $rmw_{m,w}$ denota el rango de la mujer w en la lista de preferencias del hombre m y $rwm_{x,m}$ denota el rango del hombre m en la lista de w . Es patente que los valores de esos arreglos auxiliares son constantes y pueden determinarse inicialmente a partir de los valores wmr y mwr .

El proceso de determinar el predicado *stable* se realiza ahora en estricto apego a su definición original. Recuérdese que si estamos probando la factibilidad de casar a m y w , donde $w = wmr_{m,r}$, esto es, w es la r -ésima elección del hombre. Siendo optimistas, pri-

mero suponemos que todavía prevalece la estabilidad, y luego empezamos a encontrar las posibles fuentes de dificultades. ¿Dónde pueden estar? Se dan posibilidades simétricas:

1. Puede haber una mujer pw , preferida a w por m , quien también prefiera a m sobre su esposo.
2. Puede haber un hombre pm , preferido a m por w , quien también prefiera a w sobre su esposa.

Rastreando la fuente del problema 1, comparamos los rangos $rwm_{pw,m}$ y $rwm_{pw,y_{pw}}$ para todas las mujeres preferidas a w por m , o sea para todas las $pw = wmr_{m,i}$ tales que $i < r$. Sucede que conocemos que todas las candidatas ya están casadas pues, si hubiera alguna soltera, m ya hubiera sido escogido. El proceso descrito antes podemos formularlo mediante una simple búsqueda lineal; s denota la estabilidad.

```
s := TRUE; i := 1;
WHILE (i < r) & s DO
    pw := wmr[m,i]; i := i+1;
    IF ~single[pw] THEN s := rwm[pw,m] > rwm[pw,y[pw]] END
END
```

(3.45)

Al buscar la fuente del problema 2, debemos investigar a todos los candidatos pm que son preferidos por w a su asignación actual m , es decir, todos los hombres preferidos $pm = mwr_{w,i}$ tales que $i < rwm_{w,m}$. Por analogía con la detección de la fuente del problema 1, se necesita la comparación entre los rangos $rmw_{pm,x}$ y $rmw_{pm,x_{pm}}$. No obstante, tenemos de tener cuidado y omitir las comparaciones que incluyen x_{pm} donde pm es todavía simple. La medida de seguridad que se requiere consiste en probar $pm < m$, pues sabemos que todos los hombres que preceden a m ya están casados.

El algoritmo completo se muestra en el programa 3.6. La tabla 3.4 especifica las nueve soluciones estables calculadas a partir de los datos de entrada wmr y mwr que aparecen en la tabla 3.3.

```
MODULE Marriage;
FROM InOut IMPORT
    ReadCard, WriteCard, WriteLn;

CONST n = 8;
TYPE man = [1 .. n];
    woman = [1 .. n];
    rank = [1 .. n];

VAR m: man; w: woman; r: rank;
    wmr: ARRAY man, rank OF woman;
    mwr: ARRAY woman, rank OF man;
    rmw: ARRAY man, woman OF rank;
    rwm: ARRAY woman, man OF rank;
    x: ARRAY man OF woman;
    y: ARRAY woman OF man;
```

```
single: ARRAY woman OF BOOLEAN;
h: CARDINAL;

PROCEDURE print;
    VAR m: man; rm, rw: CARDINAL;
BEGIN rm := 0; rw := 0;
FOR m := 1 TO n DO
    WriteCard(x[m], 4);
    rm := rmw[m, x[m]] + rm; rw := rwm[x[m], m] + rw
END ;
WriteCard(rm, 8); WriteCard(rw, 4); WriteLn
END print;

PROCEDURE stable(m: man; w: woman; r: rank): BOOLEAN;
    VAR pm: man; pw: woman;
        i, lim: rank; S: BOOLEAN;
BEGIN S := TRUE; i := 1;
WHILE (i < r) & S DO
    pw := wmr[m,i]; i := i+1;
    IF ~single[pw] THEN S := rwm[pw,m] > rwm[pw,y[pw]] END
END ;
i := 1; lim := rwm[w,m];
WHILE (i < lim) & S DO
    pm := mwr[w,i]; i := i+1;
    IF pm < m THEN S := rmw[pm,w] > rmw[pm,x[pm]] END
END ;
RETURN S
END stable;

PROCEDURE Try(m: man);
    VAR w: woman; r: rank;
BEGIN
FOR r := 1 TO n DO w := wmr[m,r];
    IF single[w] & stable(m,w,r) THEN
        x[m] := w; y[w] := m; single[w] := FALSE;
        IF m < n THEN Try(m+1) ELSE print END ;
        single[w] := TRUE
    END
END
END Try;

BEGIN
FOR m := 1 TO n DO
    FOR r := 1 TO n DO
        ReadCard(h); wmr[m,r] := h; rmw[m, wmr[m,r]] := r
    END
END
```

```

END ;
FOR w := 1 TO n DO
  single[w] := TRUE;
  FOR r := 1 TO n DO
    ReadCard(h); mwr[w,r] := h; rwm[w, mwr[w,r]] := r
  END
END ;
Try(1)
END Marriage.

```

Programa 3.6 Matrimonios estables.

Este algoritmo se basa en un esquema sencillo de rastreo inverso. Su eficiencia depende primordialmente de la complejidad del plan de poda del árbol. Un algoritmo un poco más rápido, aunque también más intrincado y menos transparente, ha sido propuesto por McVitie y Wilson [3-1 y 3-2] quienes también lo ampliaron al caso de dos conjuntos (de hombres y mujeres) de tamaño desigual.

Los algoritmos del tipo de los dos últimos ejemplos, que generan todas las soluciones posibles a un problema (dadas ciertas restricciones) a menudo sirven para seleccionar una o varias de las soluciones que son óptimas en algún sentido. En nuestro ejemplo, podríamos querer la solución que, en promedio, satisface mejor a los hombres, a las mujeres o a todos.

Obsérvese que la tabla 3.4 indica las sumas de los rangos de todas las mujeres en las listas de preferencia de sus esposos y las sumas de los rangos de todos los hombres en las listas de preferencia de sus esposas. He aquí los valores:

$$\begin{aligned} rm &= \sum_{m=1}^n rwm_{m,x_m} & (3.46) \\ rw &= \sum_{m=1}^n rwm_{x_m,m} \end{aligned}$$

La solución con el valor mínimo rm se llama solución estable de *hombre óptimo*; la que tiene el rw más pequeño es la solución estable de *mujer óptima*. La naturaleza de la

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rm	rw	c
1	7	4	3	8	1	5	2	6	16	32	21
2	2	4	3	8	1	5	7	6	22	27	449
3	2	4	3	1	7	5	8	6	31	20	59
4	6	4	3	8	1	5	7	2	26	22	62
5	6	4	3	1	7	5	8	2	35	15	47
6	6	3	4	8	1	5	7	2	29	20	143
7	6	3	4	1	7	5	8	2	38	13	47
8	3	6	4	8	1	5	7	2	34	18	758
9	3	6	4	1	7	5	8	2	43	11	34

c = número de evaluaciones de estabilidad.

Solución 1 = Solución de hombre óptimo; Solución 9 = Solución de mujer óptima.

Tabla 3.4 Resultado del problema de los matrimonios estables.

estrategia de búsqueda seleccionada implica que las buenas soluciones desde el punto de vista de los hombres se generan primero y las buenas soluciones desde el punto de vista de las mujeres aparecen hacia el final. En este sentido, el algoritmo se basa en la población masculina. Esto puede cambiarse de inmediato intercambiando sistemáticamente los papeles de hombres y mujeres, o sea con sólo intercambiar mwr con wmr y rmw con rwm .

Nos abstendremos de extender más este programa y dejaremos la incorporación de una búsqueda de la solución óptima para el siguiente y último ejemplo del algoritmo de rastreo inverso.

3.7 EL PROBLEMA DE SELECCION OPTIMA

El último ejemplo del algoritmo de rastreo inverso es una extensión lógica de los dos ejemplos anteriores representados por el esquema general (3.35). Primero estuvimos aplicando el principio de rastreo inverso para encontrar una *sola* solución a determinado problema. Esto se exemplificó con el recorrido del caballo y el problema de las ocho reinas. Luego nos propusimos obtener *todas* las soluciones de un problema en particular; los ejemplos fueron los de las ocho reinas y el de los matrimonios estables. Ahora queremos encontrar una solución óptima.

Para ello es preciso generar todas las soluciones posibles y, al hacerlo, conservar la que sea óptima en un sentido específico. Suponiendo que lo óptimo se define a partir de alguna función valuada positiva $f(s)$, el algoritmo se deriva del esquema (3.35) reemplazando la proposición *imprimir solución* por la proposición

IF $f(solution) > f(optimum)$ THEN $optimum := solution$ END (3.47)

La variable *optimum* registra la mejor solución encontrada hasta ahora. Por supuesto debe ser inicializada adecuadamente; más aún, se acostumbra registrar el valor $f(optimum)$ mediante otra variable a fin de evitar su recálculo frecuente.

A continuación se da un ejemplo del problema general de encontrar una solución óptima a determinado problema. Escogemos el problema más importante con que nos topamos a menudo: obtener la selección óptima entre un conjunto de objetos con restricciones. Las selecciones que constituyen las soluciones aceptables poco a poco se construyen investigando los objetos individuales a partir del conjunto base. El procedimiento *Try* describe el proceso de investigar la adecuación de un objeto individual y se llama recursivamente (para investigar el siguiente objeto) hasta que todos quedan incluidos.

Observamos que el examen de cada objeto (llamados candidatos en los ejemplos precedentes) tiene dos resultados posibles, a saber: la inclusión del objeto investigado en la selección actual o su exclusión. Esto se sirve de una proposición repeat o for inapropiadas; en cambio, los dos casos pueden escribirse explícitamente. Esto se advierte si suponemos que los objetos están numerados 1, 2, ..., n.

```

PROCEDURE Try(i: INTEGER);
BEGIN
  IF inclusión es aceptable THEN incluir el i-ésimo objeto
    IF i < n THEN Try(i+1) ELSE verificar la optimización END ;
    eliminar el i-ésimo objeto          (3.48)
  END ;
  IF exclusión es aceptable THEN
    IF i < n THEN Try(i+1) ELSE verificar la optimización END
  END
END Try

```

En este patrón es evidente que hay 2^n conjuntos posibles; desde luego, es preciso aplicar criterios de aceptabilidad adecuados para reducir el número de candidatos investigados. A fin de dilucidar ese proceso, escogeremos un ejemplo concreto de un problema de selección: supongamos que los n objetos a_1, \dots, a_n se caracterizan por su peso y su valor. Sea el conjunto óptimo el que tiene la suma más grande de los valores de sus componentes y sea la restricción un límite sobre la suma de su peso. Este problema lo conocen muy bien todos los viajeros que empacan maletas seleccionando de n elementos, en forma tal que su valor total sea óptimo y que su peso total no rebase el peso permitido.

Y así estamos ya en condiciones de escoger la representación de los hechos en función de los datos. Las elecciones de (3.49) se derivan fácilmente de los desarrollos anteriores.

```

TYPE index = [1 .. n];
    object = RECORD weight, value: INTEGER END ;
VAR obj: ARRAY index OF object;
    limw, totv, maxv: INTEGER;
    s, odds: SET OF index

```

(3.49)

Las variables *limw* y *totv* denotan el límite de peso y el valor total de todos los *n* objetos. Estos dos valores son constantes durante todo el proceso de selección. *s* representa la selección actual de hechos en que cada objeto está representado por su nombre (índice). *opts* es la selección óptima encontrada hasta ese momento y *maxv* su valor.

¿Cuáles son ahora los criterios de aceptabilidad de un objeto para la selección actual? Si consideramos *inclusion*, un objeto es seleccionable cuando encaja en el peso permitido. Pero si consideramos *exclusion*, el criterio de aceptabilidad, o sea de la continuación de la construcción de la selección actual, es que el valor total todavía alcanzable tras esta exclusión no sea menor que el valor del óptimo encontrado hasta ahora. En efecto, si es menor, la continuación de la búsqueda no aportará la solución óptima aunque sí produzca alguna solución. Con estas dos condiciones determinaremos las cantidades relevantes que deben calcularse en cada paso durante el proceso de selección:

1. El peso total tw de la selección hecha hasta este momento.
 2. El valor todavía alcanzable av de la selección actual s .

Estas dos entidades se representan adecuadamente como parámetros del procedimiento *Try*. La condición *la inclusión es aceptable* en (3.48) ahora puede formularse así:

$$tw + a[i].weight \leq limw \quad (3.50)$$

y la siguiente comprobación de optimización se formula como

IF $a_v > \max v$ THEN (* nuevo optimo, registrarlo *)
 $opts := s$; $\max v := a_v$
 END (3.51)

La última asignación se basa en el razonamiento de que el valor alcanzable es el valor obtenido, una vez que todos los objetos han sido calculados. La condición la *exclusión es aceptable* en (3.48) se expresa por

$$av - a[i].value > maxv \quad (3.52)$$

Puesto que se usará de nuevo más adelante, el valor $av[i]$. valor recibe el nombre avI a fin de evitar su reevaluación.

A continuación el programa entero sigue de (3.48) a (3.52) agregándole las proposiciones correspondientes de inicialización para las variables globales. Conviene señalar la facilidad con que se expresan la inclusión y exclusión a partir del conjunto s utilizando los operadores de conjunto. Los resultados de la ejecución del programa 3.7, con sus pesos permitidos que oscilan entre 10 y 120 vienen en la tabla 3.5.

```

MODULE Selection;
(*encontrar seleccion optima de objetos con restricciones*)
FROM InOut IMPORT
  ReadCard, Write, WriteCard, WriteString, WriteLn;

CONST n = 10;
TYPE index = [1 .. n];
object = RECORD value, weight: CARDINAL END ;
ObjSet = SET OF index;

VAR i: index;
obj: ARRAY index OF object;
limw, totv, maxv: CARDINAL;
s, opts: ObjSet;
WeightInc, WeightLimit: CARDINAL;
tick: ARRAY [FALSE .. TRUE] OF CHAR;

PROCEDURE Try(i: index; tw, av: CARDINAL);
VAR av1: CARDINAL;
BEGIN (*probar inclusión*)
  IF tw + obj[i].weight <= limw THEN
    s := s + ObjSet{i};
    IF i < n THEN Try(i+1, tw + obj[i].weight, av)
    ELSIF av > maxv THEN maxv := av; opts := s
    END;
    s := s - ObjSet{i}
  END;
  (*probar exclusión*)
  IF av > maxv + obj[i].value THEN
    IF i < n THEN Try(i+1, tw, av - obj[i].value)
    ELSE maxv := av - obj[i].value; opts := s
    END
  END
END Try;

BEGIN totv := 0; limw := 0;
  tick[FALSE] := " "; tick[TRUE] := "*";
  FOR i := 1 TO n DO

```

```

  ReadCard(obj[i].weight); ReadCard(obj[i].value);
  totv := totv + obj[i].value
END;
ReadCard(WeightInc); ReadCard(WeightLimit);
WriteString ("Peso")
FOR i := 1 TO n DO WriteCard(obj[i].weight, 5) END ;
WriteLn; WriteString ("Valor")
FOR i := 1 TO n DO WriteCard(obj[i].value, 5) END ;
WriteLn;
REPEAT limw := limw + WeightInc; maxv := 0;
  s := ObjSet{}; opts := ObjSet{}; Try(1, 0, totv);
  WriteCard(limw, 6);
  FOR i := 1 TO n DO
    WriteString(" "); Write(tick[i IN opts])
  END;
  WriteCard(maxv, 8); WriteLn
UNTIL limw >= WeightLimit
END Selection.

```

Programa 3.7 Selección óptima.

Este esquema de rastreo inverso con un factor de limitación que dificulta el crecimiento del árbol potencial de búsqueda se llama también algoritmo de *rama y límite*.

Valor de peso	10	11	12	13	14	15	16	17	18	19
	18	20	17	19	25	21	27	23	25	24
10	*									18
20								*		27
30					*			*		52
40	*				*			*		70
50	*	*		*				*		84
60	*	*	*	*	*					99
70	*	*			*			*	*	115
80	*	*	*		*			*	*	130
90	*	*			*			*	*	139
100	*	*		*	*			*	*	157
110	*	*	*	*	*	*		*		172
120	*	*			*	*	*	*	*	183

Tabla 3.5 Salida muestra del programa de selección óptima.

EJERCICIOS

3.1. (Torres de Hanoi). Se dan tres agujas y n discos de distintos tamaños. Los discos pueden meterse en las agujas y formar torres. Supongamos que n discos están puestos inicialmente en la aguja A por orden de tamaño decreciente, como se aprecia en la figura 3.10 para $n = 3$. La tarea consiste en mover n discos de la aguja A y ponerlos en la C de modo que queden ordenados en la forma original. Esto se debe lograr observando las siguientes restricciones:

1. En cada paso exactamente un disco se mueve de una aguja a la siguiente.
2. Un disco nunca puede colocarse encima de otro más pequeño.
3. La aguja B puede usarse como un almacenamiento auxiliar.

Encuentre un algoritmo que realice esta tarea. Nótese que una torre puede considerarse adecuadamente como formada por un solo disco en la cima y que la torre conste de los discos restantes. Describa el algoritmo como un programa recursivo.

3.2. Escriba un procedimiento que genere todas las $n!$ permutaciones de n elementos a_1, \dots, a_n *in situ*, o sea sin ayuda de otro arreglo. Luego de generar la siguiente permutación, debe llamarse un procedimiento paramétrico Q que, entre otras cosas, puede producir la permutación generada.

Sugerencia: considere la tarea de generar todas las permutaciones de los elementos a_1, \dots, a_m como formadas por las m subtareas consistentes en generar todas las permutaciones de a_1, \dots, a_{m-1} , seguidas por a_m , donde en la i -ésima subtarea los dos elementos a_i y a_m habían sido intercambiados.

3.3. Deduzca el esquema de recursión de la figura 3.11, que es una sobreposición de las cuatro curvas W_1, W_2, W_3, W_4 . La estructura es similar a la de las curvas Sierpinski (3.21) y (3.22). A partir del patrón de recursión, derive un programa recursivo que dibuje esas curvas.

3.4. Sólo 12 de las 92 soluciones calculadas en el programa de las ocho reinas son esencialmente diferentes. Las otras pueden derivarse por reflexiones alrededor de los

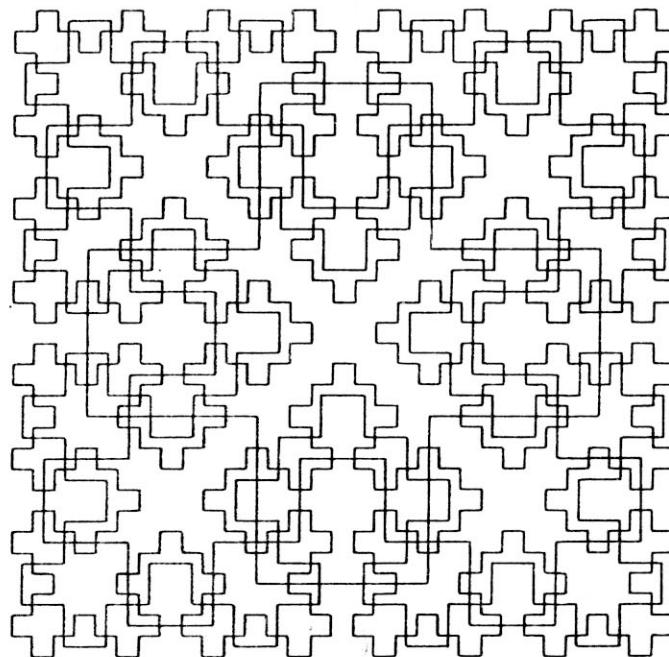
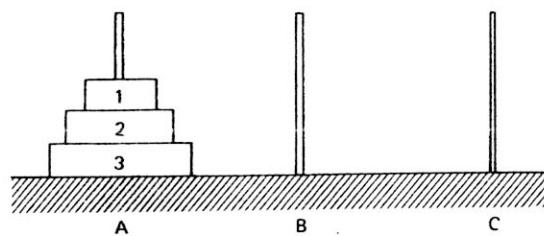


Fig. 3.11 Curvas W del orden 1 a 4.

ejes o el punto del centro. Prepare un programa que determine las 12 soluciones principales. Nótese que, por ejemplo, la búsqueda en la columna 1 puede restringirse a las posiciones 1-4.

- 3.5. Cambie el programa de matrimonios estables de modo que determine la solución óptima (hombre o mujer). Se convierte, pues, en un programa de ramas y límites como los representados en el programa 3.7.
- 3.6. Cierta empresa ferrocarrilera pasa por las n estaciones S_1, \dots, S_n . Proyecta mejorar el servicio de información a los clientes mediante terminales de información computarizada. El cliente teclea la estación de salida S_A y su destino S_D ; de inmediato debe recibir el programa de las conexiones del tren con el tiempo mínimo total de duración del viaje. Idee un programa que calcule la información deseada. Suponga que el horario (que es su banco de datos) viene en una estructura de datos adecuada que contiene las horas de salida (= llegada) de todos los trenes. Por supuesto, no todas las estaciones están conectadas por líneas directas (véase también el ejercicio 1.8).
- 3.7. La función Ackermann A se define para todos los argumentos enteros no negativos m y n del modo siguiente:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m, 0) &= A(m-1, 1) \quad (m > 0) \\ A(m, n) &= A(m-1, A(m, n-1)) \quad (m, n > 0) \end{aligned}$$

Diseñe un programa que calcule $A(m, n)$ sin utilizar la recursión. A manera de pauta use el programa 2.11, la versión no recursiva de clasificación rápida. Prepare un conjunto de reglas que rijan la transformación de programas recursivos en programas iterativos en general.

BIBLIOGRAFIA

- 3-1. D.G. McVitie and L.B. Wilson. The Stable Marriage Problem. *Comm ACM*, 14, No. 7 (1971), 486-92.
- 3-2. -----. Stable Marriage Assignment for Unequal Sets. *Bit*, 10 (1970), 295-309.
- 3-3. Space Filling Curves, or How to Waste Time on a Plotter. *Software - Practice and Experience*, 1, No. 4 (1971), 403-40.
- 3-4. N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14, No. 4 (1971), 221-27.

4. ESTRUCTURAS DE INFORMACION DINAMICAS

4.1. TIPOS DE DATOS RECURSIVOS

En el capítulo 2 se presentaron las estructuras arreglo, registro y conjunto como estructuras de datos fundamentales. Se llaman fundamentales debido a que constituyen los bloques elementales de los cuales se forman las estructuras más complejas y ya que en la práctica ocurren muy frecuentemente. El objeto de definir un tipo de datos y por consiguiente especificar que ciertas variables son de ese tipo, es que la gama de valores tomados por estas variables y por tanto su patrón de almacenamiento, se fija una sola vez y para todos. En consecuencia, se dice que las variables declaradas en esta forma son *estáticas*. Sin embargo, hay muchos problemas en los que intervienen estructuras de información mucho más complicadas. La característica de estos problemas es que no sólo los valores, sino también las estructuras de las variables, cambian durante el proceso de cálculo. Por lo tanto, se las llama estructuras *dinámicas*. Naturalmente, las componentes de estas estructuras son (en algún nivel de resolución) estáticas, es decir, de uno de los tipos de datos fundamentales. Este capítulo está dedicado a la construcción, análisis y manejo de estructuras de información dinámicas.

Es digno de mención que existen algunas analogías entre los métodos utilizados para estructurar algoritmos y aquellos para estructurar datos. Como sucede con todas las analogías, puede seguir habiendo algunas diferencias, pero una comparación de los métodos de estructuración de programas y datos es ilustrador.

La proposición no estructurada elemental es la *asignación* de un valor de una expresión a una variable. Su miembro correspondiente en la familia de las estructuras de datos es el tipo escalar no estructurado. Estos dos son los bloques elementales atómicos de proposiciones compuestas y tipos de datos. Las estructuras más simples, que se obtienen a través de la enumeración o secuenciación, son la proposición compuesta y la estructura de registro. Ambas constan de un número finito (generalmente pequeño) de componentes enumeradas en forma explícita, las cuales pueden ser diferentes entre sí. Si todas las componentes son idénticas no necesitan escribirse individualmente: se utiliza la proposición *for* y la estructura de arreglo para indicar réplica en un factor finito conocido. Una elección entre dos o más elementos se expresa por medio de la proposición condicional o

la case y por medio de la estructura de registro variante, respectivamente. Y, por último, una repetición por un factor inicialmente conocido (y potencialmente infinito) se expresa por medio de las proposiciones while o bien repeat. La estructura de datos correspondiente es la secuencia (archivo), el tipo más simple que permite la construcción de tipos de cardinalidad infinita.

Surge la interrogante de si existe o no una estructura de datos que corresponda en forma análoga a la proposición procedure. Naturalmente, la propiedad más interesante y novedosa de los procedimientos en este aspecto es la *recursión*. Los valores de este tipo de datos recursivo contendrán una o más componentes que pertenecerán al mismo tipo que éste, en analogía con un procedimiento que contiene una o más solicitudes de él. Al igual que los procedimientos, las definiciones de los tipos de datos podrán ser directa o indirectamente recursivas.

Un simple ejemplo de un objeto que se representará adecuadamente como un tipo definido en forma recursiva es la expresión aritmética que se encuentra en los lenguajes de programación. La recursión se utiliza para reflejar la posibilidad de anidamiento, es decir, de utilizar subexpresiones entre paréntesis como operandos en expresiones. En consecuencia, definiremos aquí una expresión informal como sigue:

Una *expresión* consta de un término, seguido de un operador y después un término. (Los dos términos constituyen los operandos del operador.) Un *término* es una variable (representada por un identificador) o bien una expresión entre paréntesis.

Un tipo de datos cuyos valores representan estas expresiones pueden describirse fácilmente utilizando las herramientas de que ya se dispone con la adición de recursión:

```

TYPE expression = RECORD op: operator;
    opd1, opd2: term
END

TYPE term = RECORD
    CASE t: BOOLEAN OF
        TRUE: id: alfa |
        FALSE: subex: expression
    END
END

```

(4.1)

Nota: Utilizando Modula-2, debemos usar aquí una construcción case, ya que el lenguaje no admite una construcción if en definiciones de registros.

Por consiguiente, toda variable del tipo *término* consta de dos componentes, es decir, el campo identificador *t* y, si *t* es real, el campo *id* o bien del campo *subex* en caso contrario. Consideremos ahora, por ejemplo, las cuatro expresiones que siguen:

1. $x + y$
2. $x - (y * z)$
3. $(x + y) * (z - w)$
4. $(x/(y + z)) * w$

(4.2)

Estas expresiones pueden ser visualizadas por medio de los modelos de la figura 4.1, los cuales exhiben su estructura recursiva anidada y determinan la proyección o mapeo de estas expresiones en la memoria.

Un segundo ejemplo de una estructura de información recursiva es la genealogía de la familia: sea una genealogía definida por (el nombre de) una persona y las dos genealogías de los padres. Esta definición nos lleva inevitablemente a una estructura infinita. Las genealogías reales son limitadas debido a que falta información en algún nivel de ascendencia. Esto puede tomarse en cuenta utilizando una vez más una estructura variante como la que se muestra en (4.3).

```

TYPE ped = RECORD
    CASE known: BOOLEAN OF
        TRUE: name: alfa; father, mother: ped |
        FALSE: (*vacío*)
    END
END

```

(4.3)

Nótese que toda variable del tipo *ped* tiene al menos una componente, es decir, el campo identificador llamado *known*. Si su valor es TRUE, entonces hay tres campos más; en caso contrario, no hay ninguno. Aquí se muestra un valor particular en las formas de una expresión anidada y de un diagrama que puede sugerir un posible patrón de almacenamiento (véase la figura 4.2).

$(T, Ted, (T, Fred, (T, Adam, (F), (F)), (F)), (T, Mary, (F), (T, Eva, (F), (F))))$

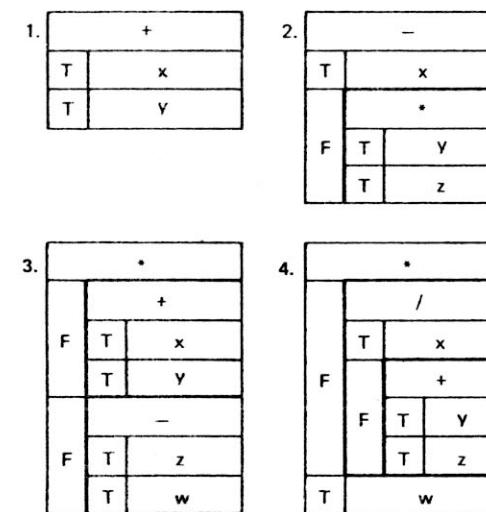


Fig. 4.1 Patrón de almacenamiento para estructuras de registro recursivas.

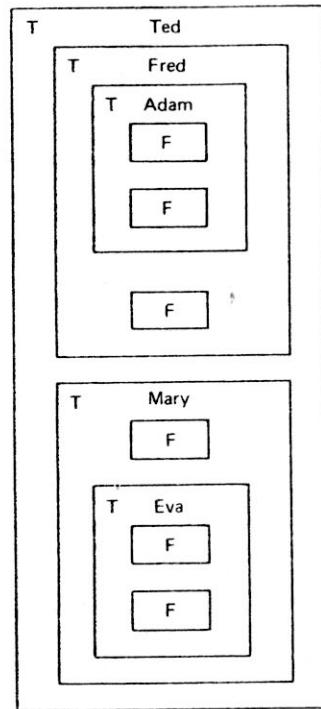


Fig. 4.2 Un ejemplo de una estructura de datos recursiva

El papel importante del recurso variante se hace evidente; es el único medio por el cual una estructura de datos recursiva puede ser limitada y por lo tanto es una compañía inevitable de toda definición recursiva. La analogía entre los conceptos de estructuración de programas y datos es particularmente notable en este caso. Una proposición condicional (o selectiva) debe ser parte de todo procedimiento recursivo a fin de que la ejecución del procedimiento pueda terminar. La terminación de la ejecución corresponde evidentemente a la cardinalidad finita.

4.2. APUNTADORES

La propiedad característica de las estructuras recursivas que las distingue claramente de las estructuras fundamentales (arreglos, registros, conjuntos) es su capacidad de cambiar de tamaño. En consecuencia, es imposible asignar una cantidad fija de espacio en la memoria a una estructura definida recursivamente y como consecuencia un compilador no puede asociar direcciones específicas a las componentes de estas variables. La técnica que se utiliza en forma más común para dominar este problema implica la *asignación dinámica* de almacenamiento, es decir, asignación de espacio en la memoria para componentes individuales en el momento en que entran en existencia durante la ejecución de un programa, en vez de en un instante de transición. El compilador asigna después una cantidad fija de almacenamiento para contener la dirección de la componente asignada dinámicamente y no de la componente misma. Por ejemplo, el árbol genealógico que se ilustra en la figura 4.2 será representado por registros individuales (posiblemente no contiguos), uno por cada persona. Estas personas se vinculan después por sus direcciones asignadas a los campos de *father* y *mother* respectivos. Gráficamente, esta situación se expresa de la mejor manera mediante el uso de flechas o apuntadores (véase la figura 4.3).

Debe destacarse que el uso de apuntadores para instrumentar estructuras recursivas es meramente una técnica. El programador no necesita tener conocimiento de su existencia. El espacio de almacenamiento puede asignarse en forma automática la primera vez que se haga referencia de una nueva componente. Sin embargo, si la técnica de utilizar referencias o apuntadores se hace explícita, pueden construirse estructuras más generales que aquellas definibles por una definición de datos puramente recursiva. En particular, es posible después definir estructuras potencialmente infinitas o circulares y dictaminar que ciertas estructuras se comparten. En consecuencia se ha vuelto común en lenguajes de programación avanzados hacer posible la manipulación explícita de referencias de datos además de los datos mismos. Esto implica que debe existir una distinción notacional clara entre los datos y las referencias de éstos y que en consecuencia deben introducirse ti-

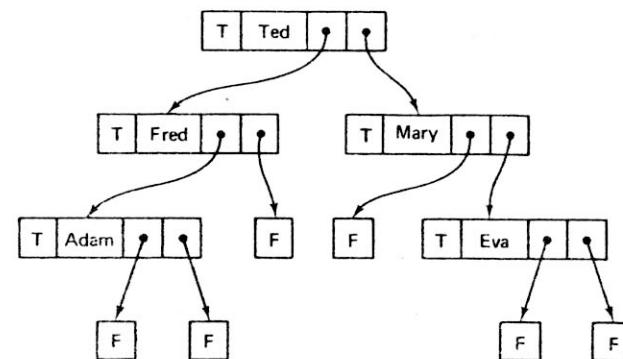


Fig. 4.3 Estructura ligada por apuntadores

pos de datos cuyos valores sean *apunadores* (referencias) de otros datos. La notación que se emplea con este objeto es la siguiente:

TYPE T = POINTER TO T0 (4.4)

La declaración de tipo (4.4) expresa que los valores del tipo T son apuntadores de datos del tipo T0. Es muy importante que el tipo de los elementos referidos sea evidente a partir de la declaración de T. Se dice que T es *límite* de T0 . Esta ligadura distingue los apuntadores de lenguajes de nivel superior de las direcciones en códigos ensambladores y se trata de un recurso de considerable importancia para incrementar la seguridad en la programación a través de la redundancia de la notación subyacente.

Los valores de los tipos apuntadores se generan siempre que un elemento de datos es asignado dinámicamente. Nos apegaremos a la convención de que este suceso se mencione explícitamente en todas las ocasiones. Esto se hace en contraste a la situación en la cual la primera vez que se menciona un elemento, éste se distribuye automáticamente. Con este fin, se presenta un procedimiento *Allocate* (Asignar). Dada una variable apuntadora de tipo T, la proposición *Allocate(p)* asigna efectivamente una variable de tipo T0 y asigna el apuntador que hace referencia de esta nueva variable a p (véase la figura 4.4). El valor del apuntador mismo puede referirse ahora como p (es decir, como el valor de la variable apuntadora p). En cambio, la variable que es referida por p se representa por p^* .

Nota: Si `Allocate` es un procedimiento tomado de un módulo de manejo de almacenamiento general es necesario especificar el tamaño de la variable en forma explícita por medio de un segundo parámetro:

Allocate(p, SIZE(T0)) (4.5)

Se mencionó antes que una componente variante es indispensable en todo tipo recursivo para asegurar la cardinalidad finita. El ejemplo del árbol genealógico familiar es de un modelo que exhibe una constelación que ocurre con mayor frecuencia [véase (4.3)], es decir, el caso en el cual el campo identificador tiene dos valores (booleanos) y en el cual su valor con calidad de falso implica la ausencia de otras componentes. Esto lo expresa el esquema de declaración (4.6).

TYPE T = RECORD (4.6)

```

CASE terminal: BOOLEAN OF
    FALSE: S(T) |
    TRUE: (* vacio *)
END
END
```

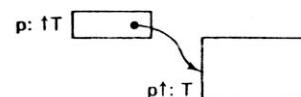


Fig. 4.4 Asignación dinámica de variable.

$S(T)$ denota una secuencia o sucesión de definiciones de campo que incluye uno o más campos del tipo T, con lo cual se asegura la recursividad. Todas las estructuras de un tipo formado según (4.6) exhiben una estructura de árbol (o lista) análoga a la que se muestra en la figura 4.3. Su propiedad peculiar es que contiene apuntadores de componentes de datos con un solo campo identificador, es decir, sin más información relevante. La técnica de instrumentación que utiliza apuntadores sugiere una forma sencilla de ahorrar espacio para almacenamiento, haciendo que la información identificadora se incluya en el valor del apuntador mismo. La solución común consiste en ampliar el intervalo de valores de todos los tipos apuntadores en un solo valor que no apunte a ningún elemento en absoluto. Este valor se representa por el símbolo especial NIL, y se entiende que NIL es automáticamente un elemento de todos los tipos apuntadores declarados. Esta extensión del intervalo de valores del apuntador explica por qué pueden generarse estructuras finitas sin la presencia explícita de variantes (condiciones) en su declaración (recursiva).

Las nuevas formulaciones de los tipos de datos declarados en (4.1) y (4.3), que se basan en apuntadores explícitos, se dan en (4.7) y (4.8), respectivamente. Nótese que en el último caso [que originalmente correspondía al esquema (4.6)] la componente del registro variante se ha desvanecido, ya que $\sim p.\text{conocida}$ ahora se expresa como $p = \text{NIL}$. La renominación del tipo *ped* a *person* refleja la diferencia en el punto de vista originado por la introducción de valores de apuntador explícitos. En vez de primero considerar la estructura en su totalidad y después investigar su subestructura y sus componentes, la atención se centra en las componentes en primer término y su interrelación (representada por apuntadores) no es evidente a partir de alguna declaración fija.

TYPE termPtr = ^ POINTER TO term; (4.7)

TYPE expPtr = **POINTER TO expression:**

TYPE expression = RECORD op: operator;
 opd1, opd2: termPtr
END;

```

TYPE term = RECORD
  CASE t: BOOLEAN OF
    TRUE: id: alfa |
    FALSE: sub: expPtr
  END
END

```

TYPE PersonPtr = ^Person;

```
TYPE person = RECORD name: alfa;
               father, mother: PersonPtr
             END
```

La estructura de datos que representa el árbol genealógico que se muestra en las figuras 4.2 y 4.3 se vuelve a mostrar en la figura 4.5; en él los apunadores de personas desco-

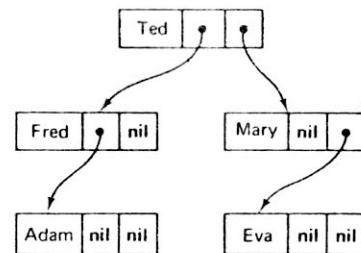


Fig. 4.5 Estructura con apuntadores NIL.

nocidas se simbolizan por NIL. La mejora resultante en economía de almacenamiento es obvia.

Volviendo a hacer referencia de la figura 4.5, supóngase que Fred y Mary son hermanos, es decir, tienen el mismo padre y madre. Esta situación se expresa fácilmente reemplazando los dos valores NIL en los campos respectivos de los dos registros. Una instrumentación que oculta el concepto de apunadores o bien utiliza una técnica diferente de manejo de almacenamiento forzaría al programador a representar los registros de ascendencia de Adam y Eva dos veces. Aunque al acceder a sus datos con fines de inspección no importa si los dos padres (y las dos madres) se duplican o representan con un solo registro, la diferencia es esencial cuando se permite actualización selectiva. Considerar a los apunadores como elementos de datos explícitos en vez de como auxiliares ocultos en la instrumentación permite al programador expresar con claridad dónde se intenta establecer compartición de almacenamiento y dónde no.

Una consecuencia adicional de la claridad de los apunadores es que es posible definir y manipular estructuras de datos cíclicas. Esta flexibilidad adicional origina, desde luego, no sólo potencias crecientes sino también requiere mayor cuidado por parte del programador, dado que la manipulación de estructuras de datos cíclicas nos puede llevar fácilmente a procesos sin terminación.

Este fenómeno de potencia y flexibilidad que se vinculan íntimamente con el peligro de uso equivocado es bien conocido en la programación y recuerda en particular a la proposición GOTO. En realidad, si la analogía entre las estructuras de programas y de datos ha de ampliarse, la estructura de datos puramente recursiva podría colocarse bien en el nivel correspondiente con el procedimiento, en tanto que la introducción de apunadores se compara con el uso de proposiciones GOTO. Por esto, como la proposición GOTO permite la construcción de cualquier tipo de modelo de programa (incluyendo ciclos), también los apunadores permiten la composición de cualquier estructura de datos (incluyendo anillos). La elaboración simultánea de estructuras de programas y datos correspondientes se muestra en forma condensada en la tabla 4.1.

En el capítulo 3 se ha observado que la iteración es un caso especial de recursión y que una solicitud de un procedimiento recursivo llamado P se define según el esquema (4.9).

Patrón de construcción	Proposición de programa	Tipo de datos
Elemento atómico	Asignación	Tipo escalar
Enumeración	Proposición compuesta	Tipo registro
Repetición (factor conocido)	Proposición for	Tipo arreglo
Elección	Proposición condicional	Tipo unión (reg. variable)
Repetición	Proposición while o repeat	Tipo de secuencia
Recursión	Proposición procedure	Tipo de datos recursivos
Gráfica general	Proposición GO TO	Estructura ligada por apunadores

Tabla 4.1 Correspondencia de Programa y Estructura de Datos.

```

PROCEDURE P;
BEGIN
  IF B THEN P0; P END
END
  
```

(4.9)

donde P0 es una proposición que no implica a P, es equivalente a, y sustituible por, la proposición iterativa

```
WHILE B DO P0 END
```

Las analogías que se delinean en la tabla 4.1 revelan que se sostiene una relación semejante entre los tipos de datos recursivos y la secuencia. De hecho, un tipo recursivo definido según el esquema

```

TYPE T = RECORD
  CASE B: BOOLEAN OF
    TRUE: t0: T0; t: T |
    FALSE:
  END
END
  
```

(4.10)

donde T0 es un tipo que no incluye a T, es equivalente y reemplazable por un tipo de datos secuenciales

```
SEQUENCE OF T0
```

El resto de este capítulo se dedica a la generación y manipulación de estructuras de datos cuyas componentes se vinculan por medio de apunadores explícitos. Las estructuras con modelos específicos simples se destacan en particular; pueden derivarse fórmulas para manejar estructuras más complejas que aquellas que manipulan formaciones básicas. Estas son la lista lineal o secuencia encadenada (el caso más simple) y los árboles. Nuestra preocupación con estos bloques elementales de estructuración de datos no implica que en la práctica no ocurran estructuras más complicadas. De hecho, la siguiente historia apareció en un diario de Zúrich en julio de 1922 y es una prueba de que puede ocurrir irregularidad aun en casos que generalmente sirven como ejemplos de estructuras

regulares, como los árboles (de familias). La historia habla de un hombre que lamenta la miseria de su vida en las palabras siguientes:

Me casé con una viuda que tenía una hija mayor. Mi padre, quien nos visitaba muy a menudo, se enamoró de mi hijastra y se casó con ella. En consecuencia, mi padre se convirtió en mi yerno y mi hijastra pasó a ser mi madre. Algunos meses después, mi esposa dio a luz a un hijo, quien se convirtió en el cuñado de mi padre así como en mi tío. La esposa de mi padre, que es mi hijastra, también tuvo un hijo. Por lo tanto, tuve un hermano y al mismo tiempo un nieto. Mi esposa es mi abuela, ya que ella es la madre de mi madre. Por consiguiente, soy el marido de mi esposa y al mismo tiempo su medio nieto; en otras palabras, soy mi propio abuelo.

4.3. LISTAS LINEALES

4.3.1. Operaciones básicas

La manera más simple de interrelacionar o vincular un conjunto de elementos consiste en alinearlos en una sola lista o fila. En este caso, sólo se necesita un vínculo por cada elemento para hacer referencia de su sucesor.

Supóngase que un tipo *Node* y un tipo *Ptr* se definen como se muestra en (4.11). Toda variable de este tipo consta de tres componentes, es decir, una llave identificadora, el apuntador de su sucesor y posiblemente otra información asociada que se omite en (4.11).

```
TYPE Ptr =  POINTER TO Node;
TYPE Node = RECORD key: INTEGER;
            next: Ptr;
            data: ...
          END ;
VAR p, q: Ptr
```

(4.11)

Una lista de nodos, con un apuntador a su primera componente asignada a una variable *p*, se ilustra en la figura 4.6. Quizá la operación más simple que se realiza con una lista como se muestra en la figura 4.6 es la inserción de un elemento en su cabeza o parte superior. Primero, se asigna un elemento de tipo *Node* y su referencia (apuntador) se asigna a una variable apuntadora auxiliar, por decir algo *q*. Según esto una simple reasignación de apuntadores completa la operación, que se programa en (4.12). Obsérvese que el orden de estas tres proposiciones es esencial.

```
Allocate(q, SIZE(Node)); q^.next := p; p := q
```

(4.12)

La operación de inserción de un elemento en la parte superior de una lista sugiere inmediatamente la forma en que una lista de este tipo puede ser generada: comenzando con la lista vacía, se añade varias veces. El proceso de *generación de lista* se expresa en (4.13); aquí, el número de elementos por unir es *n*.

```
p := NIL; (*empezar con lista vacia*)
WHILE n > 0 DO
  Allocate(q, SIZE(Node)); q^.next := p; p := q;
```

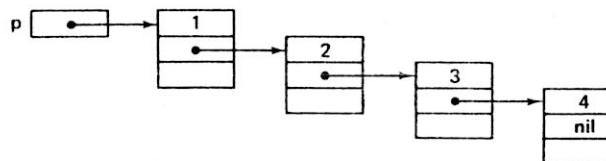
(4.13)


Fig. 4.6 Ejemplo de una lista.

```

q↑.key := n; n := n-1
END

```

Esta es la forma más simple de formar una lista. Sin embargo, el orden resultante de los elementos es el recíproco del orden de su inserción. En algunas aplicaciones esto es indecible y en consecuencia, deben agregarse nuevos elementos al final en vez de en la parte superior de la lista. Aunque el final se puede determinar fácilmente por medio de un examen de la lista, este punto de vista ingenuo implica un esfuerzo que bien puede evitarse utilizando un segundo apuntador, por decir algo q , que siempre designe el último elemento. El método se aplica, como ejemplo, en el programa 4.4, que genera referencias cruzadas de un texto dado. Su desventaja es que el primer elemento insertado tiene que ser tratado de forma diferente de los últimos.[†]

La disponibilidad explícita de los apuntadores hace muy simples ciertas operaciones que en otras circunstancias son engorrosas: entre las operaciones de lista elementales se encuentran las de inserción y borrado de elementos (actualización selectiva de una lista) y, desde luego, la revisión o recorrido de una lista. Primero investigaremos la *inserción en lista*.

Supóngase que un elemento designado por un apuntador (variable) q se insertará en una lista *después de* el elemento designado por el apuntador p . Las asignaciones de apuntadores que se necesitan se expresan en (4.14) y su efecto se exhibe en la figura 4.7.

$$q↑.next := p↑.next; p↑.next := q \quad (4.14)$$

Si se desea la inserción *antes de* en vez de después del elemento designado $p↑$, la cadena de unión unidireccional parece ocasionar un problema, ya que no ofrece ninguna clase de trayectoria hacia los predecesores de un elemento. Sin embargo, un simple truco resuelve nuestro dilema: éste se expresa en (4.15) y se ilustra en la figura 4.8. Supóngase que la llave del nuevo elemento es 8.

$$\text{Allocate}(q, \text{SIZE}(\text{Node})); q↑ := p↑; \\ p↑.key := k; p↑.next := q \quad (4.15)$$

El truco evidentemente consiste en insertar una nueva componente *después de* $p↑$ y después intercambiar los valores del nuevo elemento y $p↑$.

A continuación, se considera el proceso del *borrado de lista*. La eliminación del sucesor de un elemento $p↑$ es directo. En (4.16) se muestra esto en combinación con la rein-

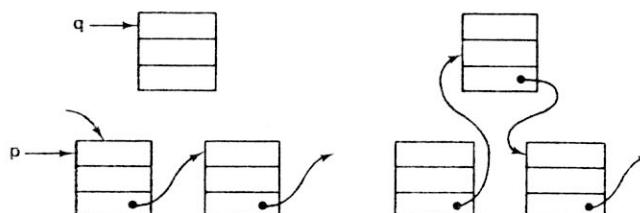


Fig. 4.7 Inserción en lista después de $p↑$.

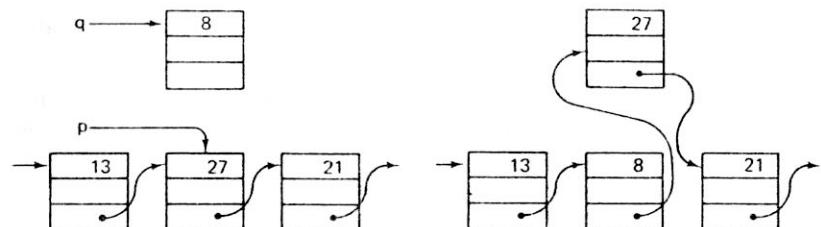


Fig. 4.8 Inserción en lista antes de $p↑$.

serción del elemento borrado que está en la parte superior de otra lista (designada por q). La figura 4.9 ilustra el efecto de las proposiciones (4.16) y muestra que se trata de un intercambio cíclico de tres apuntadores.

$$r := p↑.next; p↑.next := r↑.next; r↑.next := q; q := r \quad (4.16)$$

La eliminación de un elemento designado (en vez de su sucesor) es más difícil, ya que se encuentra el mismo problema que con la inserción: retornar al predecesor del elemento denotado es imposible. Pero la eliminación del sucesor después de adelantar su valor es una solución relativamente obvia y simple. Puede aplicarse siempre que $p↑$ tenga un sucesor, es decir, no sea el último elemento en la lista.

Ahora nos concentraremos en la operación fundamental de *recorrido de la lista*. Supóngase que tiene que realizarse una operación $P↑(x)$ por todos y cada uno de los elementos de la lista cuyo primer elemento es $p↑$. Esta tarea se puede expresar como sigue:

```

WHILE lista designada por  $p$  no este vacía DO
  realizar la operación  $P$ ;
  proceder hacia el sucesor
END

```

En detalle, esta operación se describe por medio de la proposición (4.17).

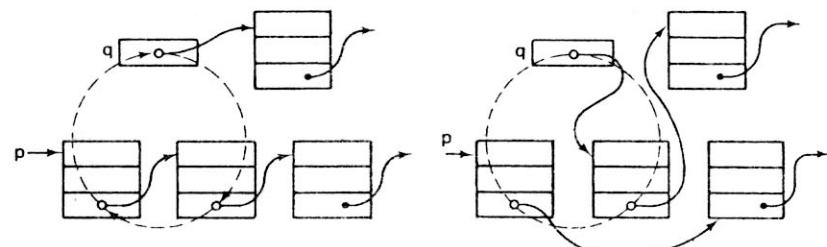
$$\text{WHILE } p \neq \text{NIL DO} \\ P(p↑); p := p↑.next \\ \text{END} \quad (4.17)$$


Fig. 4.9 Eliminación de lista y reinserción.

De las definiciones de la proposición while y de la estructura ligada se sabe que P se aplica a todos los elementos de la lista y no a otros.

Una operación muy frecuente que se efectúa es la *búsqueda en lista* de un elemento con una llave dada x . A diferencia de los arreglos, el proceso de búsqueda debe ser aquí puramente secuencial. La búsqueda termina si se halla un elemento o bien si se llega al final de la lista. Esto lo refleja una conjunción lógica que consta de dos términos. Una vez más, se supone que la parte superior de la lista se designa por medio de un apuntador p .

WHILE(p ≠ NIL) & (pt.key ≠ x) DO p := pt.next END (4.18)

$p = \text{NIL}$ implica que $p.t$ no existe y, en consecuencia, que la expresión $p.t.key \# x$ es indefinida. El orden de los dos términos es esencial por lo antes dicho.

4.3.2. Listas ordenadas y reorganización de listas

El algoritmo (4.18) se asemeja mucho a las rutinas de búsqueda para examinar un arreglo o bien una secuencia. De hecho, una secuencia es precisamente una lista lineal para la cual la técnica de enlace (liga) hacia el sucesor no se especifica o es implícita. Ya que los operadores de secuencia primitivos no permiten la inserción de nuevos elementos (excepto al final) o eliminación (excepto cuando se retiran todos los elementos), la elección de la representación se deja abierta al instrumentador y éste puede bien utilizar la asignación secuencial, dejando componentes sucesivas en áreas de almacenamiento contiguas. Las listas lineales con apuntadores explícitos ofrecen mayor flexibilidad y, por lo tanto, deben utilizarse siempre que se necesite esta flexibilidad adicional.

Para dar un ejemplo, consideremos ahora un problema que ocurrirá en todo este capítulo a fin de ilustrar soluciones y técnicas alternativas. Se trata del problema de la lectura de un texto, colectando todas sus palabras y contando la frecuencia de su ocurrencia. A esto se le denomina construcción de una *concordancia* o bien generación de una *lista de referencia cruzada*.

Una solución obvia consiste en construir una lista de palabras halladas en el texto. La lista se examina para encontrar cada palabra. Si se encuentra la palabra, su conteo de frecuencia se incrementa; en caso contrario la palabra se agrega a la lista. A este proceso simplemente lo llamaremos *búsqueda*, aunque en realidad puede incluir también una inserción. A fin de poder concentrar nuestra atención en la parte esencial del manejo de una lista, se supone que las palabras ya han sido extraídas del texto en investigación; se han codificado como enteros y se tienen a disposición en forma de una secuencia de entrada.

La formulación del procedimiento llamado *Search* sigue en forma directa de (4.18). La variable *root* se refiere a la parte superior de la lista en la cual se insertan nuevas palabras según (4.12). El algoritmo completo se enlista como el programa 4.1; incluye una rutina para tabular la lista con referencia cruzada construida. El proceso de tabulación es un ejemplo en el cual se ejecuta una acción una vez por cada elemento de la lista, como se muestra en forma de esquema en (4.17).

El algoritmo de examen lineal del programa 4.1 se asemeja al procedimiento de búsqueda para los arreglos y nos recuerda una técnica simple que se utiliza para simplificar

```

MODULE List; (* insercion directa en lista *)
  FROM InOut IMPORT ReadInt, Done, WriteInt, WriteLn;
  FROM Storage IMPORT Allocate;

  TYPE Ptr = POINTER TO Word;
  Word =
    RECORD key: INTEGER;
      count: CARDINAL;
      next: Ptr
    END;
  VAR k: INTEGER; root: Ptr;

  PROCEDURE search(x: INTEGER; VAR root: Ptr);
    VAR w: Ptr;
  BEGIN w := root;
    WHILE (w # NIL) & (w^.key # x) DO w := w^.next END
    (* (w = NIL) OR (w^.key = x) *)
    IF w = NIL THEN (*nueva entrada*)
      w := root; Allocate(root, SIZE(Word));
      WITH root^ DO
        key := x; count := 1; next := w
      END
    ELSE w^.count := w^.count + 1
    END
  END search;

  PROCEDURE PrintList(w: Ptr);
  BEGIN
    WHILE w # NIL DO
      WriteInt(w^.key, 8); WriteInt(w^.count, 8); WriteLn;
      w := w^.next
    END
  END PrintList;

  BEGIN root := NIL; ReadInt(k);
    WHILE Done DO
      search(k, root); ReadInt(k)
    END ;
    PrintList(root)
  END List.

```

Programa 4.1 Inserción directa en lista

la condición de terminación del ciclo: el uso de un centinela. Un centinela puede servir bien en la búsqueda en una lista; ésta se representa por medio de un elemento ficticio al final de la lista. El nuevo procedimiento es (4.21), el cual sustituye el procedimiento de

búsqueda del programa 4.1, siempre y cuando se agregue una variable global *sentinel* y que la inicialización de *root* se reemplace por las proposiciones

```
Allocate(sentinel, SIZE(Node)); root := sentinel
```

que genera el elemento que se usará como centinela.

```
PROCEDURE search(x: INTEGER; VAR root: Ptr); (4.21)
  VAR w: Ptr;
BEGIN w := root; sentinel^.key := x;
  WHILE w^.key # x DO w := w^.next END ;
  IF w = sentinel THEN (*nueva entrada*)
    w := root; Allocate(root, SIZE(Node));
    WITH root DO
      key := x; count := 1; next := w
    END
  ELSE w^.count := w^.count + 1
  END
END search
```

Obviamente, el poder y flexibilidad de la lista vinculada se usan escasamente en este ejemplo y el examen lineal de toda la lista sólo se puede aceptar en casos en los cuales el número de elementos es limitado. Sin embargo, una mejora sencilla se tiene fácilmente a la mano: la *búsqueda de la lista ordenada*. Si la lista está ordenada (por decir, por llaves crecientes), entonces la búsqueda puede terminarse hasta que se encuentre la primera llave que sea mayor que la nueva. El ordenamiento de una lista se logra insertando nuevos elementos en el sitio adecuado en vez de en la parte superior. En efecto, el ordenamiento se obtiene prácticamente libre de carga. Esto se debe a la facilidad con la cual se hace la inserción en una lista ligada, es decir, haciendo uso total de su flexibilidad. Es una posibilidad no ofrecida por las estructuras de arreglo o secuencia. (Sin embargo, nótese que aun en las listas ordenadas no se dispone de un equivalente de la búsqueda binaria de arreglos.)

La búsqueda en lista ordenada es un ejemplo común de la situación que se describe en (4.15) en la cual debe insertarse un elemento adelante de un elemento dado, es decir, en frente del primero cuya llave es demasiado grande. Sin embargo, la técnica que se muestra aquí difiere de la que se emplea en (4.15). En vez de calcar valores, *dos* apunadores se llevan a todo lo largo del recorrido de la lista; *w2* se rezaga un paso antes que *w1* y, por lo tanto, identifica el sitio de inserción adecuado cuando *w1* ha hallado una llave demasiado grande. La etapa de inserción general se muestra en la figura 4.10. El apuntador del nuevo elemento (*w3*) se asignará a *w2^.next*, excepto cuando la lista siga estando vacía. Por razones de simplicidad y claridad, preferimos evitar esta distinción utilizando una proposición condicional. La única manera de evitar esto consiste en introducir un elemento ficticio en la parte superior de la lista. La proposición de inicialización *root := NIL* en el programa 4.1 se sustituye por

```
Allocate(root, SIZE(Node)); root^.next := NIL
```

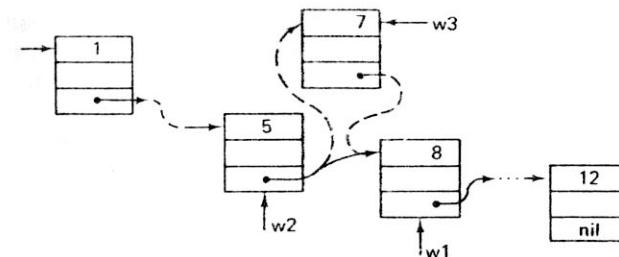


Fig. 4.10 Inserción en lista ordenada.

Haciendo referencia de la figura 4.10, se determina la condición en la cual el examen continúa para proceder hacia el siguiente elemento; consta de dos factores, es decir,

$$(w1 \neq \text{NIL}) \& (w1^.key < x)$$

El procedimiento de búsqueda resultante se muestra en (4.23).

```
PROCEDURE search(x: INTEGER); VAR root: Ptr; (4.23)
  VAR w1, w2, w3: Ptr;
BEGIN (*w2 # NIL*)
  w2 := root; w1 := w2^.next;
  WHILE (w1 # NIL) & (w1^.key < x) DO
    w2 := w1; w1 := w2^.next
  END ;
  (* (w1 = NIL) OR (w1^.key >= x) *)
  IF (w1 = NIL) OR (w1^.key >= x) THEN (*nueva entrada*)
    Allocate(w3, SIZE(Node)); w2^.next := w3;
    WITH w3 DO
      key := x; count := 1; next := w1
    END
  ELSE w1^.count := w1^.count + 1
  END
END search
```

A fin de acelerar la búsqueda, la condición de continuación de la proposición while puede simplificarse una vez más utilizando un centinela. Esto requiere la presencia inicial de una cabeza ficticia así como de un centinela en la parte final.

Ahora ya es tiempo de preguntar qué puede esperarse de la búsqueda en una lista ordenada. Recordando que la complejidad adicional es pequeña, uno debe esperar un notable mejoramiento.

Supóngase que todas las palabras del texto ocurren con la misma frecuencia. En este caso lo que se gana con el ordenamiento lexicográfico también es en realidad nulo, una vez que todas las palabras se enlistan, debido a que la posición de una palabra no interesa si sólo el total de todas las etapas de acceso es significativo y si todas las palabras

tienen la misma frecuencia de ocurrencia. Sin embargo, se obtiene una ganancia siempre que se inserta una nueva palabra. En vez de primero rastrear toda la lista, en promedio solamente se rastrea la mitad de ella. En consecuencia, la inserción de lista ordenada se omite solamente si se genera una concordancia con muchas palabras distintas en comparación con su frecuencia de ocurrencia. Los ejemplos anteriores se adaptan por tanto principalmente como ejercicios de programación en vez de como aplicaciones prácticas.

La disposición de los datos en una lista ligada se recomienda cuando el número de elementos es relativamente pequeño (menor que 50), varía y, más aún, cuando no se da información acerca de sus frecuencias de acceso. Un ejemplo común es la tabla de símbolos en compiladores de los lenguajes de programación. Cada declaración ocasiona que se realice la adición de un nuevo símbolo y en la salida de su cobertura de validez, se borra de la lista. El uso de listas ligadas simples es adecuado para aplicaciones con programas relativamente cortos. Aun en este caso, una mejora considerable en el método de acceso se puede lograr por medio de una técnica muy simple que se vuelve a mencionar aquí debido a que constituye un buen ejemplo para demostrar la flexibilidad de la estructura de la lista ligada.

Una propiedad característica de los programas es que las ocurrencias del mismo identificador a menudo son muy agrupadas, es decir, a menudo una ocurrencia va seguida de una o más reincidencias de la misma palabra. Esta información es una invitación a reorganizar la lista después de cada acceso desplazando la palabra que se encontró en la parte superior de la lista, con lo cual se minimiza la longitud de la trayectoria de búsqueda la próxima vez que se busque. A este método de acceso se le denomina *búsqueda en lista con reordenamiento* o bien (un tanto pomposo) búsqueda en lista con autoorganización. Al presentar el algoritmo correspondiente en la forma de procedimiento que se puede sustituir en el programa 4.1, se aprovecha la experiencia lograda hasta ahora y se introduce un centinela desde el mismo inicio. De hecho, un centinela no sólo acelera la búsqueda, sino que en este caso también simplifica el programa. La lista debe no estar vacía inicialmente, ya que contiene el elemento centinela. Las proposiciones de inicialización son

```
Allocate(sentinel, SIZE(Node)); root := sentinel;
```

Nótese que la diferencia principal entre el nuevo algoritmo y la búsqueda directa en la lista (4.21) es la acción de reordenamiento cuando se ha hallado un elemento. Después se borra o elimina de su posición anterior y se inserta en la parte superior. La eliminación requiere una vez más el uso de dos apuntadores de persecución, tal que el predecesor w2 de un elemento identificado w1 siga siendo localizable. Esto, a su vez, pide el trato especial del primer elemento (o sea, la lista vacía). Para concebir el proceso de enlace, nos referimos a la figura 4.11. Esta muestra los dos apuntadores cuando w1 se identificó como el elemento deseado. La configuración después del reordenamiento correcto se representa en la figura 4.12 y el nuevo procedimiento de búsqueda se muestra en (4.26).

```
PROCEDURE search(x: integer; VAR root: Ptr);
  VAR w1, w2: Ptr;
BEGIN w1 := root; sentinel.key := x;
  IF w1 = sentinel THEN (*primer elemento*)
    (4.26)
```

```
Allocate(root, SIZE(Node));
WITH root DO
  key := x; count := 1; next := sentinel
END
ELSIF w1.key = x THEN w1.count := w1.count + 1
ELSE (*busqueda*)
  REPEAT w2 := w1; w1 := w2.next
  UNTIL w1.key = x;
  IF w1 = sentinel THEN (*nueva entrada*)
    w2 := root; Allocate(root, SIZE(Node));
    WITH root DO
      key := x; count := 1; next := w2
    END
  ELSE (*encontrado, ahora reordenar*)
    w1.count := w1.count + 1;
    w2.next := w1.next; w1.next := root; root := w1
  END
END
END search
```

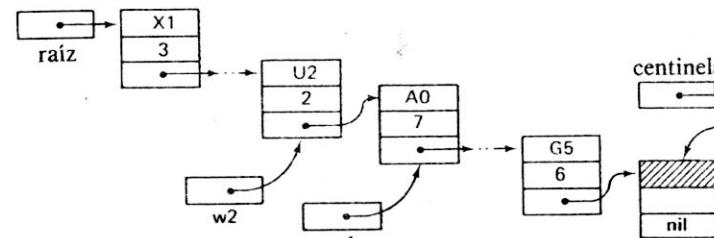


Fig. 4.11 Lista antes de la reordenación.

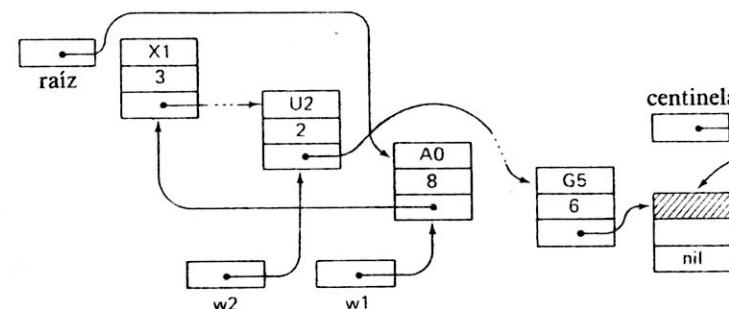


Fig. 4.12 Lista después de la reordenación.

La mejora en este método de búsqueda depende del grado de amontonamiento en los datos de entrada. Para un factor de amontonamiento dado, la mejora será más pronunciada en listas grandes. Para ofrecer una idea de cuánto puede esperarse ganar con esta mejora, se hizo una medición empírica aplicando el programa anterior de referencia cruzada y después comparando los métodos de ordenamiento de lista lineal (4.21) y la reorganización de una lista (4.26). Los datos medidos se condensan en la tabla 4.2. Por desgracia, la mejora es máxima cuando se necesita de cualquier modo una organización diferentes de los datos. Volveremos a este ejemplo en la sección 4.4.

4.3.3. Una aplicación: clasificación topológica

Un ejemplo adecuado del uso de una estructura de datos dinámica y flexible es el proceso de la *clasificación topológica*. Este es un proceso de clasificación de elementos en el cual se define un *ordenamiento parcial*, es decir, donde se da un ordenamiento de algunos pares de elementos más no entre todos ellos. Los que siguen son ejemplos de ordenamientos parciales:

1. En un diccionario o glosario, las palabras se definen en términos de otras palabras. Si una palabra v se define en términos de una palabra w , esto se representa con $v \prec w$. La clasificación topológica de las palabras en un diccionario se refiere a su disposición en un orden tal que no habrá más referencias extra.
2. Una tarea (es decir, un proyecto de ingeniería) se divide en subtareas. La terminación de ciertas subtareas debe preceder por lo general la ejecución de otras subtareas. Si una subtarea v debe preceder a una subtarea w , se escribe $v \prec w$. Clasificación topológica significa su disposición en un orden tal que a la iniciación de cada subtarea todas sus subtareas de prerequisito se hayan completado.
3. En un currículum universitario, deben tomarse ciertos cursos antes de otros ya que éstos se basan en el material presentado en sus prerequisitos. Si un curso v es un prerequisito del curso w , se escribe $v \prec w$. Clasificación topológica significa disposición de los cursos en un orden tal que ningún curso enliste un curso posterior como prerequisito.
4. En un programa, algunos procedimientos pueden contener solicitudes de otros procedimientos. Si un procedimiento v es solicitado por uno w , se escribe $v \prec w$. La clasificación topológica implica la disposición de declaraciones de procedimientos en tal forma que no haya más referencias posteriores.

Prueba 1 Prueba 2

Número de llaves distintas	53	582
Número de ocurrencias de llaves	315	14341
Tiempo de búsqueda con ordenación	6207	3200622
Tiempo de búsqueda con reordenación	4529	681584
Factor de mejoramiento	1.37	4.70

Tabla 4.2 Comparación de métodos de búsqueda en lista.

En general, un ordenamiento parcial de un conjunto S es una relación entre los elementos de S . Se representa por el símbolo \prec , que se verbaliza como *precede a* y cumple las tres propiedades (axiomas) siguientes para cualesquier elementos distintos x, y, z de S :

- (1) si $x \prec y$ y $y \prec z$, entonces $x \prec z$ (transitividad)
 - (2) si $x \prec y$, entonces no $y \prec x$ (asimetría)
 - (3) no $z \prec z$ (irreflexibilidad)
- (4.27)

Por razones evidentes, se supondrá que los conjuntos S que serán clasificados topológicamente por un algoritmo son finitos. En consecuencia, un ordenamiento parcial se puede ilustrar trazando un diagrama o gráfica en el cual los vértices simbolicen los elementos de S y las aristas dirigidas representen relaciones de ordenamiento. En la figura 4.13 se muestra un ejemplo de esto.

El problema de la clasificación topológica integrará el orden parcial en un orden lineal. Gráficamente, esto implica la disposición de los vértices de la gráfica en una hilera tal que todas las flechas apunten a la derecha, como se muestra en la figura 4.14. Las propiedades (1) y (2) de los ordenamientos parciales aseguran que la gráfica no contenga ciclos o repeticiones. Esta es exactamente la condición previa en la cual es posible tal integración en un orden lineal.

¿Cómo se encuentra uno de los posibles ordenamientos lineales? La solución es muy simple. Se inicia eligiendo cualquier elemento que no esté precedido por otro (debe haber cuando menos uno; en caso contrario, existiría un ciclo). Este objeto se coloca en la cabeza de la lista resultante y se retira del conjunto S . El conjunto restante sigue parcialmente ordenado y de esta manera puede volver a aplicarse el mismo algoritmo hasta que el conjunto esté vacío.

A fin de describir este algoritmo en forma más rigurosa, debe fijarse una estructura de datos y una representación de S y su ordenamiento. La elección de esta representación se determina por las operaciones que se realizarán, particularmente la de selección de elementos con cero predecesores. Todo elemento debe, pues, representarse por medio de tres características: su llave de identificación, su conjunto de sucesores y un conteo de sus predecesores. Ya que el número n de elementos en S no se da *a priori*, el conjunto se organiza adecuadamente como una lista ligada. En consecuencia, una entrada adicional

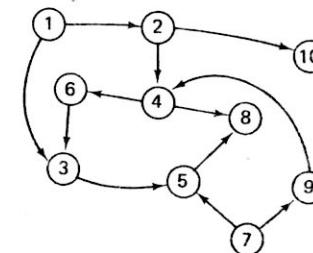


Fig. 4.13 Conjunto parcialmente ordenado.

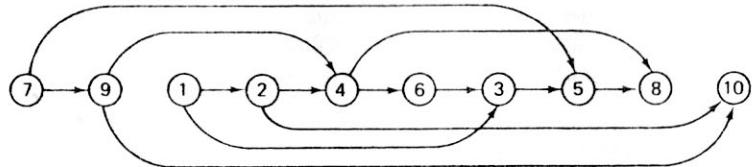


Fig. 4.14 Arreglo lineal del conjunto parcialmente ordenado de la figura.

en la descripción de cada elemento contiene el enlace al siguiente elemento en la lista. Se supondrá que las llaves son enteros (pero no necesariamente los enteros consecutivos de 1 a n). Análogamente, el conjunto de sucesores de cada elemento se representa en forma debida como una lista ligada. Cada elemento de la lista de sucesores se describe por medio de una identificación y un enlace al siguiente elemento de esta lista. Si a los descriptores de la lista principal se les llama *líderes*, en los cuales cada elemento de S ocurre exactamente una vez, y a los descriptores de elementos en las cadenas sucesoras, *seguidores*, se obtienen las siguientes declaraciones de tipos de datos.

```

TYPE LPtr =  POINTER TO leader;
TPtr =  POINTER TO trailer;

leader = RECORD key, count: INTEGER;
          trail: TPtr; next: LPtr
        END;

trailer = RECORD id: LPtr; next: TPtr
         END
    
```

(4.28)

Supóngase que el conjunto S y sus relaciones de ordenamiento se representan inicialmente como una secuencia de pares de llaves en el archivo de entrada. Los datos de entrada del ejemplo de la figura 4.13 se muestran en (4.29) donde se agregan los símbolos \prec con fines de claridad.

1 < 2	2 < 4	4 < 6	2 < 10	4 < 8	6 < 3	1 < 3		(4.29)
3 < 5	5 < 8	7 < 5	7 < 9	9 < 4	9 < 10			

La primera parte del programa de clasificación topológica debe leer la entrada y transformar los datos en una estructura de lista. Esto se lleva a cabo leyendo sucesivamente un par de llaves x y y ($x \prec y$). Los apuntadores de sus representaciones en una lista ligada de líderes se representarán por p y q. Estos registros deben ser localizados por medio de una búsqueda en lista y, si no están presentes aún, deben insertarse en la lista. Esta tarea se realiza por medio de un procedimiento de función llamado *find* (hallar). Subsiguentemente, se agrega una nueva entrada en la lista de seguidores de x, junto con una identificación de y; el conteo de predecesores de y se incrementa en 1. A este algoritmo se le denomina *fase de entrada* (4.30). La figura 4.15 ilustra la estructura de datos generada durante el procesamiento de los datos de entrada (4.29) realizado por (4.30). La función *find(w)* produce la referencia de la componente de la lista con la llave w (véase también el programa 4.2).

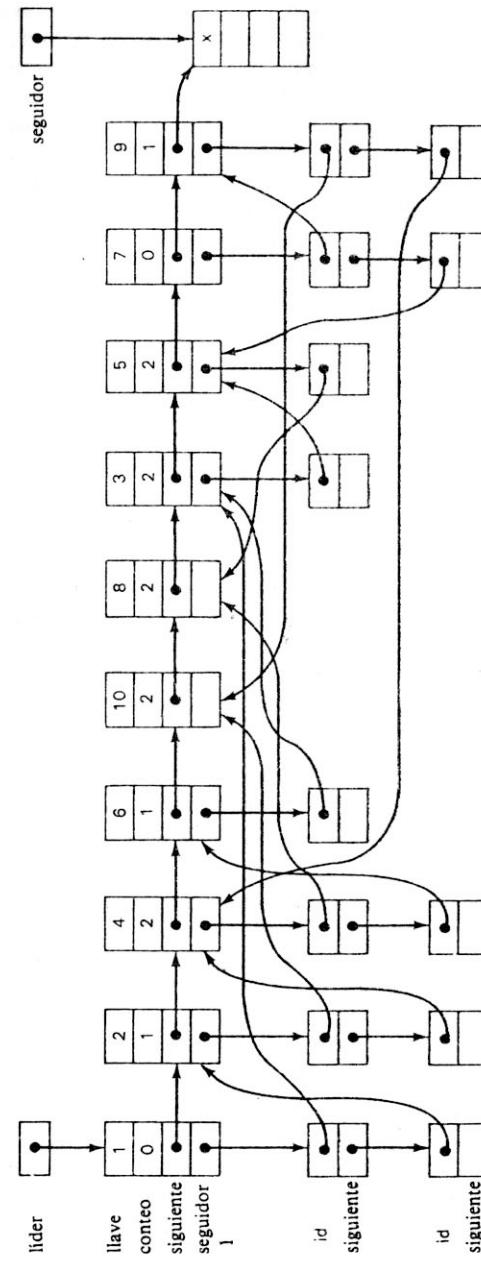


Fig. 4.15 Estructura de la lista generada por el programa clasificación topológica.

```
(*fase de entrada*)
Allocate(head, SIZE(leader)); tail := head; z := 0; ReadInt(x); (4.30)
WHILE Done DO
  ReadInt(y); p := find(x); q := find(y);
  Allocate(t, SIZE(trailer)); tt.id := q; tt.next := pt.trail;
  pt.trail := t; qt.count := qt.count + 1; ReadInt(x)
END
```

Después de construir la estructura de datos de la figura 4.16 en esta fase de entrada, el proceso real de la clasificación topológica puede emprenderse como se describió antes. Pero como consiste en la selección repetida de un elemento con conteo cero de predecesores, parece sensible a considerar primero todos los elementos contenidos en una cadena ligada. Ya que se observa que la cadena original de líderes ya no se necesitará más después, puede volver a usarse el mismo campo llamado *next*(siguiente) para analizar los líderes con cero predecesores. Esta operación de sustitución de una cadena por otra ocurre frecuentemente en el procedimiento de listas. Se expresa en detalle en (4.31) y por razones de comodidad construye la nueva cadena en el orden contrario.

```
(*busqueda de líderes sin predecesores*)
p := head; head := NIL;
WHILE p # tail DO
  q := p; p := qt.next; (4.31)
  IF qt.count = 0 THEN (*insertar qt en nueva cadena*)
    qt.next := head; head := q
  END
END
```

Haciendo referencia a la figura 4.16, se observa que la siguiente cadena de líderes se reemplaza por la de la figura 4.15 en la cual los apuntadores no representados se dejan sin variaciones.

Después de todo este establecimiento preparatorio de una representación adecuada del conjunto parcialmente ordenado S, se puede proceder finalmente a la tarea real de la clasificación topológica, es decir, de generación de la secuencia de salida. En una primera versión, poco elaborada, puede describirse como sigue:

```
q := head;
WHILE q # NIL DO (*sacar este elemento y luego eliminarlo*)
  WriteInt(qt.key, 8); n := n - 1;
  t := qt.trail; q := qt.next; (4.32)
  restar al conteo del predecesor todos sus sucesores en la lista de seguidores t; si un conteo se convierte en 0, insertar este elemento en la lista líder q
END
```

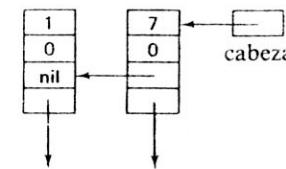


Fig. 4.16 Lista de líderes con cero conteos.

La proposición en (4.32) que todavía se refinará más constituye un rastreo más de una lista [véase el esquema (4.17)]. En cada etapa, la variable auxiliar p designa el elemento líder cuyo conteo tiene que ser disminuido y probado.

```
WHILE t # NIL DO
  p := tt.id; pt.count := pt.count - 1; (4.33)
  IF pt.count = 0 THEN (*insertar p en la lista principal*)
    pt.next := q; q := p
  END ;
  t := tt.next
END
```

Esto completa el programa de clasificación topológica. Nótese que se introdujo un contador n para contar los líderes generados en la fase de entrada. Este conteo se disminuye cada vez que un elemento líder se genera en la fase de salida. Por lo tanto, debe retornar a cero al final del programa. Si no retorna a cero, ésta es una indicación de que hay elementos que se dejaron en la estructura cuando ninguno está sin predecesor. En este caso el conjunto S evidentemente no se ordena en forma parcial. La fase de salida programada antes es un ejemplo de un proceso que mantiene una lista que “vibra”, es decir, en el cual los elementos se insertan y retiran en un orden impredecible. Este es por tanto un ejemplo de un proceso que utiliza toda la flexibilidad ofrecida por la lista ligada explícitamente.

```
MODULE TopSort;
  FROM InOut IMPORT OpenInput, CloseInput,
    ReadInt, Done, WriteInt, WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  TYPE LPtr = POINTER TO leader;
    TPtr = POINTER TO trailer;
    leader = RECORD key, count: INTEGER;
      trail: TPtr; next: LPtr
    END ;
    trailer = RECORD id: LPtr; next: TPtr
    END ;
```

```

VAR p, q, head, tail: LPtr;
t: TPtr;
x, y, n: INTEGER;

PROCEDURE find(w: INTEGER): LPtr;
  VAR h: LPtr;
BEGIN h := head; tail^.key := w; (*centinela*)
  WHILE h^.key # w DO h := h^.next END ;
  IF h = tail THEN
    ALLOCATE(tail, SIZE(leader)); n := n+1;
    ht^.count := 0; ht^.trail := NIL; ht^.next := tail
  END ;
  RETURN h
END find;

BEGIN
(*inicializar lista de líderes con un ficticio que actue como centinela*)
ALLOCATE(head, SIZE(leader)); tail := head; n := 0;

OpenInput("TEXT"); ReadInt(x);
WHILE Done DO
  WriteInt(x, 8); ReadInt(y); WriteInt(y, 8); WriteLn;
  p := find(x); q := find(y);
  ALLOCATE(t, SIZE(trailer)); tt^.id := q; tt^.next := pt^.trail;
  pt^.trail := t; qt^.count := qt^.count + 1; ReadInt(x)
  END ;
CloseInput;

(*busqueda de líderes sin predecesores*)
p := head; head := NIL;
WHILE p # tail DO
  q := p; p := qt^.next;
  IF qt^.count = 0 THEN (*inserta qt en una nueva cadena*)
    qt^.next := head; head := q
  END
END ;

(*fase de salida*) q := head;
WHILE q # NIL DO
  WriteLn; WriteInt(q^.key, 8); n := n-1;
  t := qt^.trail; q := qt^.next;
  WHILE t # NIL DO
    p := tt^.id; pt^.count := pt^.count - 1;
    IF pt^.count = 0 THEN (*insertar pt en lista de líderes*)
      pt^.next := q; q := p
    END ;
    t := tt^.next
  END ;

```

```

    END
  END ;
  IF n # 0 THEN WriteString ("Este conjunto no esta parcialmente ordenado") END
  WriteLn
END TopSort.

```

Programa 4.2 Clasificación topológica.

4.4. ESTRUCTURAS DE ARBOL

4.4.1. Conceptos y definiciones básicos

Se ha observado que las secuencias y las listas pueden definirse adecuadamente en la forma siguiente: Una secuencia (lista) con tipos base T es

1. La secuencia (lista) vacía.
2. El estabonamiento (cadena) de una T y una secuencia con tipo de base T.

En consecuencia, la recursión se utiliza como una ayuda en la definición de un principio de estructuración, secuenciación o iteración. Las secuencias y las iteraciones son tan comunes que generalmente se consideran como modelos fundamentales de estructura y comportamiento. Pero debe tenerse en mente que pueden definirse en términos de recursión, con lo cual el recíproco no es verdadero, ya que la recursión puede usarse efectiva y elegantemente para definir estructuras mucho más complejas. Los árboles son un ejemplo bien conocido. Sea que una estructura de árbol se defina como sigue: Una *estructura de árbol* con tipo base T es

1. La estructura vacía.
2. Un nodo de tipo T con un número finito de estructuras de árbol disjuntas asociadas de tipo de base T, llamadas *subárboles*.

Dada la similitud de las definiciones recursivas de secuencias y estructuras de árbol, es evidente que la secuencia (lista) es una estructura de árbol en la cual cada nodo tiene al más un subárbol. La secuencia (lista) se llama por tanto también árbol *degenerado*.

Existen varias maneras de representar una estructura de árbol. Por ejemplo, una estructura de árbol con su tipo base T que varía según las letras se muestra en diversas formas en la figura 4.17. Estas representaciones muestran la misma estructura y por tanto son equivalentes. Es la estructura de la gráfica la que ilustra de manera explícita las relaciones de bifurcación, las cuales, por razones obvias, condujeron al nombre que se utiliza generalmente de *árbol*. Extrañamente suficiente, se acostumbra representar los árboles de cabeza o bien (si se prefiere expresar este hecho en distinta forma) mostrar las raíces de los árboles. Sin embargo, la última formulación es insuficiente, ya que el nodo superior (A) se llama comúnmente *raíz*.

Un *árbol ordenado* es un árbol en el cual las ramas de cada nodo están ordenadas. En consecuencia, los dos árboles ordenados de la figura 4.18 son objetos diferentes. Un nodo y que está directamente debajo del nodo x se denomina *descendiente* (directo) de x; si x está en el nivel i, entonces se dice que y es un nivel i + 1. A la inversa, se dice que el nodo x es el *ancestro* (directo) de y. La *raíz* de un árbol se define como localizada en el nivel 0. Se dice que el nivel máximo de cualquier elemento de un árbol es su *profundidad* o *altura*.

Si un elemento no tiene descendientes, se le denomina *nodo terminal* o bien *hoja*, y un elemento que no es terminal es un *nodo interior*. El número de descendientes (directos)

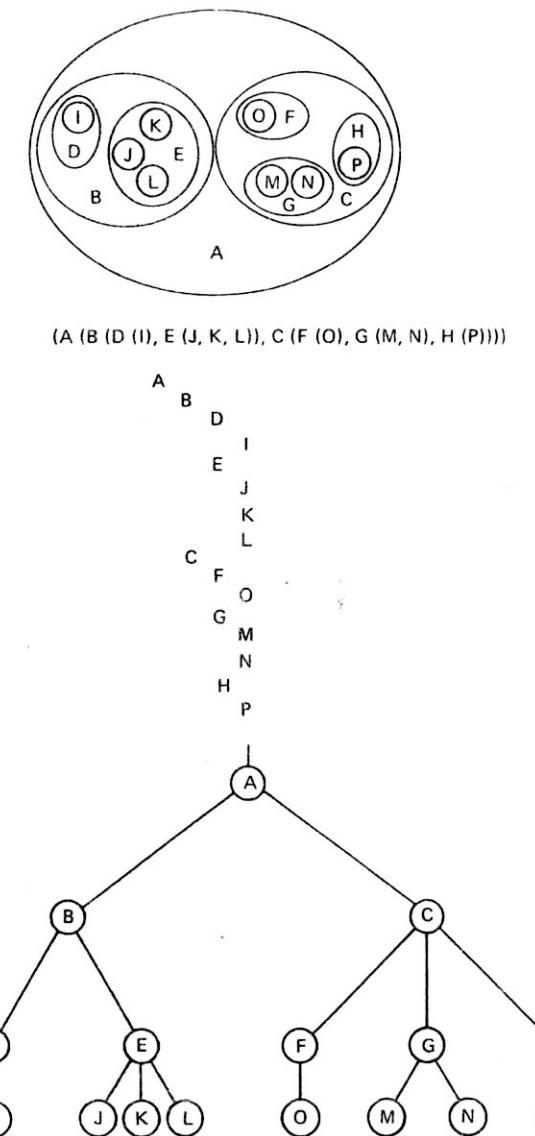


Fig. 4.17 Representación de una estructura de árbol:
 a) conjuntos anidados; b) paréntesis anidados; c) sangría (escalonamiento);
 d) gráfica

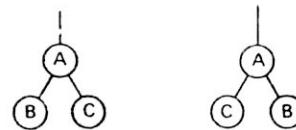


Fig. 4.18 Dos árboles binarios bien definidos.

de un nodo interior se conoce como su *grado*. El grado máximo en todos los nodos es el grado del árbol. El número de ramas o aristas que tienen que ser recorridas a fin de proceder desde la raíz hasta un nodo x se llama *longitud de trayectoria* de x . La raíz tiene una longitud de trayectoria 0, sus descendientes directos tienen una longitud de trayectoria 1, etcétera. En general, un nodo en el nivel i tiene la longitud de trayectoria i . La longitud de trayectoria de un árbol se define como la suma de las longitudes de trayectoria de todas sus componentes. También se le denomina su *longitud de trayectoria interna*. La longitud de trayectoria interna del árbol que se muestra en la figura 4.17, por ejemplo, es 36. Evidentemente, la longitud de trayectoria media es

$$P_I = (\text{Si: } 1 \leq i \leq n : n_i * i) / n \quad (4.34)$$

donde n_i es el número de nodos en el nivel i . A fin de definir lo que se denomina *longitud de trayectoria externa*, se amplía el árbol por medio de un nodo especial donde falte un subárbol en el árbol original. Al hacer esto, se supone que todos los nodos tendrán el mismo grado, es decir, el grado del árbol. Extendiendo el árbol en esta forma se consigue llenar por tanto las ramas vacías, con lo cual los nodos especiales, desde luego, no tienen más descendientes. El árbol de la figura 4.17 extendido con nodos especiales se muestra en la figura 4.19 en el cual los nodos especiales se representan por cuadrados. La longitud de trayectoria externa se define ahora como la suma de las longitudes de trayectoria en todos los nodos especiales. Si el número de nodos especiales en el nivel i es m_i , la longitud de trayectoria externa media es

$$P_E = (\text{Si: } 1 \leq i \leq m : m_i * i) / m \quad (4.35)$$

En el árbol que se muestra en la figura 4.19 la longitud de la trayectoria externa es 120. El número de nodos especiales m que se añadirá a un árbol de grado d depende directamente del número n de nodos originales. Nótese que todo nodo tiene exactamente una arista apuntando hacia él. Por lo tanto, hay $m + n$ aristas en el árbol extendido. Por el otro lado, d aristas emanen de cada nodo original, ninguno de los nodos especiales. Por consiguiente, existen $d * n + 1$ aristas, el 1 resultante de la arista que apunta hacia la raíz. Los dos resultados producen la siguiente ecuación entre el número m de nodos especiales y n de nodos originales: $d * n + 1 = m + n$, o bien,

$$m = (d-1)*n + 1 \quad (4.36)$$

El número máximo de nodos en un árbol de una altura dada h se alcanza si todos los nodos tienen d subárboles, excepto los situados en el nivel h , que no tienen ninguno. Para un árbol de grado d , el nivel 0 contiene por tanto 1 nodo (es decir, la raíz), el nivel 1

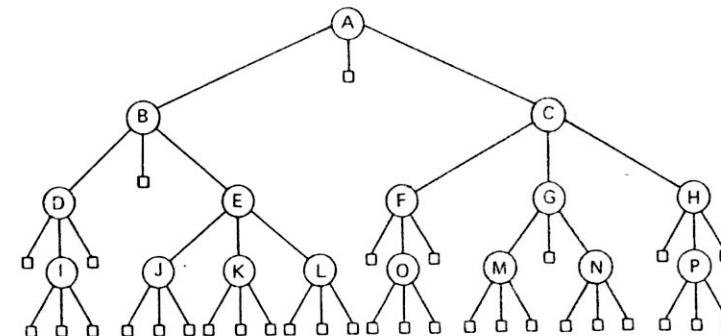


Fig. 4.19 Árbol ternario extendido con nodos especiales.

contiene sus d descendientes, el nivel 2 contiene los d^2 descendientes de los d nodos en el nivel 2, etc. Esto produce

$$N_d(h) = \text{Si: } 0 \leq i < h : d^i \quad (4.37)$$

como el número máximo de nodos para un árbol con altura h y grado d . Para $d = 2$, se obtiene

$$N_2(h) = 2^h - 1 \quad (4.38)$$

De particular importancia son los árboles ordenados de grado 2. A éstos se les llama *árboles binarios*. Un árbol binario ordenado se define como un conjunto finito de elementos (nodos) que es vacío o bien consta de una raíz (nodo) con dos árboles binarios disjuntos llamados *subárbol izquierdo* y *derecho* de la raíz. En las secciones que siguen trataremos exclusivamente con árboles binarios y por lo tanto se utilizará la palabra árbol para referirnos a un *árbol binario ordenado*. Los árboles con grado mayor que 2 se denominan *árboles multicamino (multimodales)* y se estudian en la sección 5 de este capítulo.

Algunos ejemplos conocidos de árboles binarios son el árbol familiar (genealógico) con el padre y madre de una persona como descendientes (!), la historia de un torneo de tenis con cada juego siendo un nodo representado por su ganador y los dos juegos anteriores de los competidores como sus descendientes o bien una expresión aritmética con operadores diádicos, donde cada operador representa un nodo de rama con sus operandos como subárboles (véase la figura 4.20).

Ahora nos concentraremos en el problema de la representación de árboles. Es fácil que la ilustración de dichas estructuras recursivas en términos de estructuras de ramificación sugiera inmediatamente el uso de nuestro recurso de apuntadores. Evidentemente no tiene uso la declaración de variables con una estructura de árbol fija; en cambio, se definen los nodos como variables con una estructura fija, es decir, de un tipo fijo, en la cual el grado del árbol determina el número de componentes apuntadores que se refieren a los

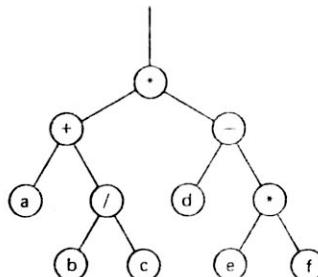


Fig. 4.20 Representación de árbol de la expresión $(a + b/c) * (d - e*f)$.

subárboles del nodo. Evidentemente, la referencia del árbol vacío se simboliza por NIL. En consecuencia, el árbol de la figura 4.20 consta de componentes de un tipo definido como sigue y después puede construirse como se muestra en la figura 4.21.

```
TYPE Ptr = POINTER TO Node;
TYPE Node = RECORD op: CHAR;
           left, right: Ptr
END
```

(4.39)

Antes de investigar la forma en que podrían usarse los árboles ventajosamente y cómo realizar operaciones con los árboles, se da un ejemplo de la forma en que puede construirse un árbol por medio de un programa. Supóngase que se generará un árbol que contendrá nodos de los tipos definidos en (4.39), con los valores de los nodos siendo n números que se leen de un archivo de entrada. A fin de hacer más desafiante el problema,

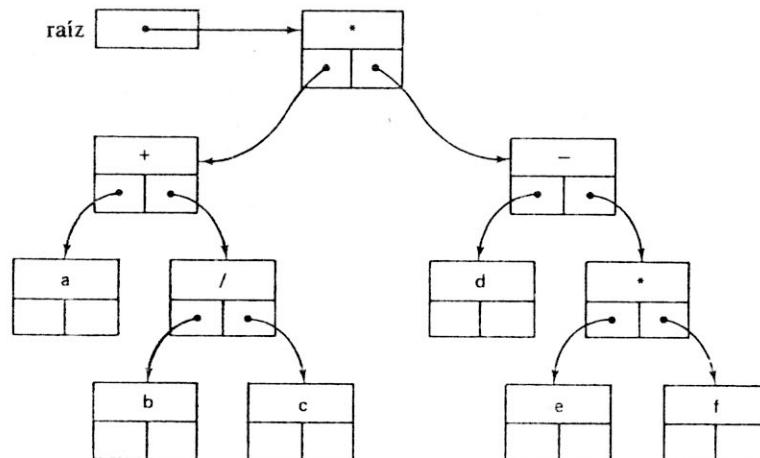


Fig. 4.21 Árbol representado como estructura de datos.

sea que la tarea consista en la construcción de un árbol con n nodos y altura mínima. A fin de obtener una altura mínima para un número de nodos dado, se debe asignar el número máximo posible de nodos en todos los niveles excepto el más inferior. Esto puede lograrse con claridad distribuyendo los nodos entrantes de igual manera a la izquierda y derecha en cada nodo. Esto implica que se estructure el árbol de n dada como se muestra en la figura 4.22, para $n = 1, \dots, 7$.

La regla de igual distribución con un número conocido n de nodos se forma mejor en forma recursiva:

1. Utilizar un nodo para la raíz.
2. Generar el subárbol de la izquierda con $nl = n \text{ DIV } 2$ nodos en esta forma.
3. Generar el subárbol de la derecha con $nr = n - nl - 1$ nodos en esta forma.

La regla se expresa como un procedimiento recursivo que forma parte del programa 4.3, el cual lee el archivo de entrada y construye el árbol perfectamente balanceado. Anotamos la siguiente definición: Un árbol es *perfectamente balanceado*, si para cada nodo los números de nodos en sus subárboles izquierdo y derecho difieren cuando más en 1.

```
MODULE BuildTree;
  FROM InOut IMPORT OpenInput, CloseInput,
             ReadInt, WriteInt, WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  TYPE Ptr = POINTER TO Node;

  Node = RECORD key: INTEGER;
         left, right: Ptr
  END;
```

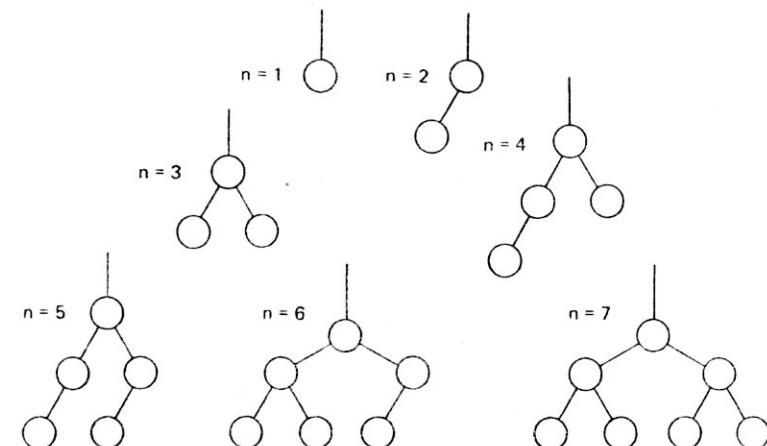


Fig. 4.22 Árboles perfectamente balanceados.

```

VAR n: INTEGER; root: Ptr;
PROCEDURE tree(n: INTEGER): Ptr;
  VAR newnode: Ptr;
  x, nl, nr: INTEGER;
BEGIN (*construir un arbol perfectamente balanceado con n nodos*)
  IF n = 0 THEN newnode := NIL
  ELSE nl := n DIV 2; nr := n-nl-1;
    ReadInt(x); ALLOCATE(newnode, SIZE(Node));
    WITH newnode DO
      key := x; left := tree(nl); right := tree(nr)
    END
  END;
  RETURN newnode
END tree;
PROCEDURE PrintTree(t: Ptr; h: INTEGER);
  VAR i: INTEGER;
BEGIN (*imprimir arbol t con sangria h*)
  IF t # NIL THEN
    WITH t DO
      PrintTree(left, h+1);
      FOR i := 1 TO h DO WriteString("   ") END ;
      WriteInt(key, 6); WriteLn;
      PrintTree(right, h+1)
    END
  END
END PrintTree;
BEGIN (*el primer entero es el numero de nodos*)
  OpenInput("TEXT"); ReadInt(n);
  root := tree(n);
  PrintTree(root, 0); CloseInput
END BuildTree.

```

Programa 4.3 Construcción de un árbol perfectamente balanceado.

Supóngase, por ejemplo, los siguientes datos de entrada para un árbol con 21 nodos:

21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

El programa 4.3 construye después el árbol perfectamente balanceado que se muestra en la figura 4.23. Nótese la simplicidad y transparencia de este programa que se obtiene a través del uso de procedimientos recursivos. Es obvio que los algoritmos recursivos son particularmente adecuados cuando un programa debe manipular información cuya estructura se define por sí mismas en forma recursiva. Esto se vuelve a manifestar en el procedimiento que imprime el árbol resultante: el árbol vacío no origina impresión, el subárbol en el nivel L es el primero en imprimir su subárbol izquierdo, después el nodo

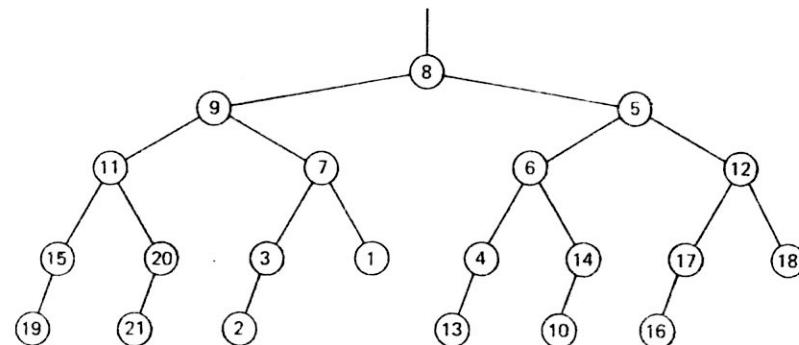


Fig. 4.23 Árbol generado por el programa 4.3.

adecuadamente insertado anteponiéndole L espacios en blanco y, finalmente, imprime su subárbol derecho.

4.4.2. Operaciones básicas con árboles binarios

Hay muchas tareas que pueden realizarse con una estructura de árbol; una común es la de ejecutar una operación P dada con cada elemento del árbol. Así pues, se entiende que P es un parámetro de la tarea más general de visitar todos los nodos o bien, como se llama generalmente, el *recorrido del árbol*. Si la tarea se considera como un solo proceso secuencial, entonces los nodos individuales son visitados en algún orden específico y pueden considerarse como colocados en una disposición lineal. De hecho, la descripción de muchos algoritmos se facilita considerablemente si se puede hablar acerca del procedimiento del siguiente elemento en el árbol con base en un orden subyacente. Existen tres ordenamientos principales que surgen en forma natural de las estructuras de los árboles. Al igual que la estructura del árbol, se expresan adecuadamente en términos recursivos. Refiriéndose al árbol binario de la figura 4.24 en la cual R representa la raíz y A y B simbolizan los subárboles izquierdo y derecho, los tres ordenamientos son

1. Preorden: R, A, B (visitar el nodo antes de los subárboles)
2. En orden: A, R, B
3. Postorden: (visitar la raíz después de los subárboles)

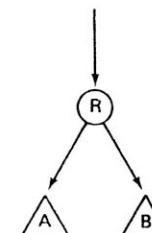


Fig. 4.24 Árbol binario.

Al recorrer el árbol de la figura 4.20 y grabar los caracteres observados en los nodos en la secuencia de encuentro, se obtienen los siguientes ordenamientos:

1. Preorden: * + a/bc - d*ef
2. En orden: a + b/c*d - e*f
3. Postorden: abc/ + def* - *

Reconocemos las tres formas de expresiones: el recorrido *preorden* del árbol de expresión produce la notación *prefija*; el recorrido *postorden* genera la notación *postfija*; y el recorrido *en orden* produce la notación convencional *infija*, aunque sin los paréntesis que se necesitan para aclarar las precedencias del operador.

Ahora formularemos los tres métodos de recorrido por medio de tres programas concretos con el parámetro explícito *t* que simboliza el árbol en que se operará y con el parámetro implícito *P* que representa la operación que se efectuará con cada nodo. Súpongase las definiciones siguientes:

```
TYPE Ptr = POINTER TO Node;
TYPE Node = RECORD ...
    left, right: Ptr
END
```

(4.42)

Los tres métodos se formulan fácilmente como procedimientos recursivos; demuestran una vez más el hecho de que las operaciones con estructuras de datos definidas recursivamente se definen en forma más adecuada como algoritmos recursivos.

```
PROCEDURE preorder(t: Ptr);
BEGIN
    IF t # NIL THEN
        P(t); preorder(t^.left); preorder(t^.right)
    END
END preorder
```

(4.43)

```
PROCEDURE inorder(t: Ptr);
BEGIN
    IF t # NIL THEN
        inorder(t^.left); P(t); inorder(t^.right)
    END
END inorder
```

(4.44)

```
PROCEDURE postorder(t: Ptr);
BEGIN
    IF t # NIL THEN
        postorder(t^.left); postorder(t^.right); P(t)
    END
END postorder
```

(4.45)

Nótese que el apuntador *t* se pasa como parámetro de valor. Esto expresa el hecho de que la entidad relevante es la referencia del subárbol considerado y no la variable cuyo valor es el apuntador y que podría ser cambiado en caso que *t* se pasara como parámetro de variable.

Un ejemplo de una rutina de recorrido de árbol es aquella de la impresión de un árbol, con la indentación adecuada indicando el nivel de cada nodo (véase el programa 4.3).

Los árboles binarios se usan para representar un conjunto de datos cuyos elementos se recuperarán a través de una llave única. Si un árbol se organiza en tal forma que para cada nodo t_i todas las llaves en el subárbol izquierdo de t_i sean menores que la llave de t_i y aquellas en el subárbol derecho sean mayores que la llave de t_i , entonces a este árbol se le llama *árbol de búsqueda*. En un árbol de búsqueda es posible localizar una llave arbitraria comenzando en la raíz y prosiguiendo a lo largo de una trayectoria de búsqueda cambiando a un subárbol izquierdo o derecho de un nodo por medio de una decisión basada en la inspección de esa llave del nodo solamente. Como se ha observado, n elementos pueden organizarse en un árbol binario de una altura tan pequeña como $\log n$. Por lo tanto, una búsqueda entre n elementos puede realizarse con tan pocas como $\log n$ comparaciones si el árbol está perfectamente balanceado. Con claridad, el árbol es una forma mucho más adecuada para organizar un conjunto de datos de este tipo que la lista lineal que se utilizó en la sección anterior. Como esta búsqueda sigue una sola trayectoria de la raíz al nodo deseado, puede ser programada fácilmente por la iteración (4.46).

```
PROCEDURE locate(x: INTEGER; t: Ptr): Ptr;
BEGIN
    WHILE (t # NIL) & (t^.key # x) DO
        IF t^.key < x THEN t := t^.right ELSE t := t^.left END
    END ;
    RETURN t
END locate
```

(4.46)

La función *locate(x, t)* produce el valor NIL, si no se halla ninguna llave con el valor *x* en el árbol con raíz *t*. Como sucede en el caso de búsqueda en una lista, la complejidad de la condición de terminación sugiere que puede haber una mejor solución, es decir, el uso de un centinela. Esta técnica es igualmente aplicable en caso de tratarse de un árbol. El uso de apuntadores hace posible que todas las ramas del árbol terminen con el mismo centinela. La estructura resultante ya no es un árbol, sino que en su lugar es un árbol con todas las hojas sujetadas por cadenas a un solo punto de apoyo (figura 4.25). El centinela puede considerarse como un representante común compartido de todos los nodos externos por el cual el árbol original se extendió (véase la figura 4.19). La rutina de búsqueda simplificada resultante aparece en (4.47).

```
PROCEDURE locate(x: INTEGER; t: Ptr): Ptr;
BEGIN st.key := x; (*centinela*)
    WHILE t^.key # x DO
        IF t^.key < x THEN t := t^.right ELSE t := t^.left END
    END
```

(4.47)

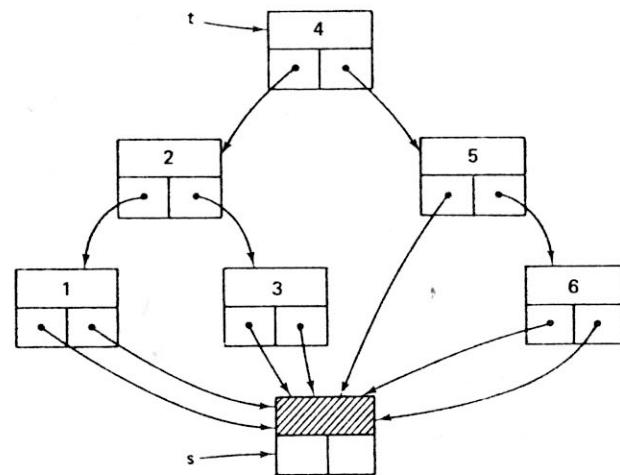


Fig. 4.25 Árbol de búsqueda con centinela.

```

END ;
RETURN t
END locate

```

Nótese que en este caso *locate(x, t)* produce el valor *s* en vez de NIL, es decir, el apuntador hacia el centinela, si no se halla ninguna llave con el valor *x* en el árbol con raíz *t*.

4.4.3. Búsqueda e inserción en árbol

La potencia total de la técnica de asignación dinámica con acceso a través de apuntadores es exhibida difícilmente por aquellos ejemplos en los cuales se construye un conjunto dado de datos y por lo tanto se mantienen inalterados. Algunos ejemplos más adecuados son aquellas aplicaciones en las cuales la estructura del árbol mismo varía, es decir, crece y/o se contrae durante la ejecución del programa. Este también es el caso en el cual otras representaciones de datos, como el arreglo, fallan y en el cual el árbol con elementos enlazados por apuntadores surge como la solución más adecuada.

Primero consideremos solamente el caso de un árbol que crece en forma constante pero que nunca se contrae. Un ejemplo común es el problema de concordancia que ya se investigó en relación con las listas ligadas. Ahora se volverá a considerar. En este problema se da una secuencia de palabras y tiene que determinarse el número de ocurrencias. Esto significa que, comenzando con un árbol vacío, cada palabra se busca en el árbol. Si se encuentra, su conteo de ocurrencias se incrementa; en caso contrario, se inserta como una nueva palabra (con un conteo inicializado en 1). A la tarea subyacente se la llama *búsqueda en árbol con inserción*. Se suponen las siguientes definiciones de tipos de datos:

```

TYPE WPtr = POINTER TO Word;
Word = RECORD
  key: INTEGER;
  count: CARDINAL;
  left, right: WPtr
END

```

(4.48)

Suponiendo además una fuente de llaves y una variable que denota la raíz del árbol de búsqueda, se puede formular el programa como

```

ReadInt(x);
WHILE Done DO search(x, root); ReadInt(x) END

```

Una vez más, la obtención de la trayectoria de búsqueda es directa. Sin embargo, si esto nos lleva a punto muerto (es decir, a un subárbol vacío designado por un valor de apuntador (NIL), la palabra dada debe insertarse en el árbol en el sitio del subárbol vacío. Considerese, por ejemplo, el árbol binario que se muestra en la figura 4.26 y la inserción del nombre *Paul*. El resultado se presenta en líneas punteadas en la misma figura.

Toda la operación se muestra en el programa 4.4. El proceso de búsqueda se formula como un procedimiento recursivo. Nótese que este parámetro *p* es un parámetro de variable y no un parámetro de valor. Esto es esencial debido a que en el caso de la inserción debe asignarse un nuevo valor de apuntador a la variable que antes contenía el valor NIL. Utilizando la secuencia de entrada de 21 números que se habían aplicado al programa 4.3 para construir el árbol de la figura 4.23, el programa 4.4 produce el árbol de búsqueda binaria que se puede apreciar en la figura 4.27.

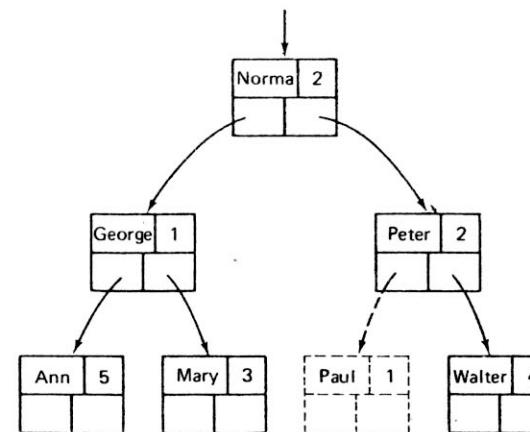


Fig. 4.26 Inserción en un árbol binario ordenado.

```

MODULE TreeSearch;
FROM InOut IMPORT OpenInput, CloseInput,
  ReadInt, Done, WriteInt, WriteString, WriteLn;

```

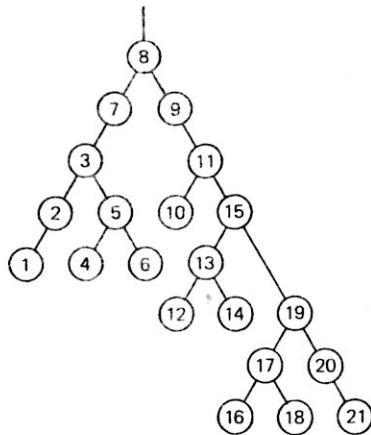


Fig. 4.27 Árbol de búsqueda generado por el programa 4.4.

```

FROM Storage IMPORT ALLOCATE;
TYPE WPtr = POINTER TO Word;
Word = RECORD key: INTEGER;
  count: INTEGER;
  left, right: WPtr
END ;
VAR root: WPtr; n, key: INTEGER;
PROCEDURE PrintTree(t: WPtr; h: INTEGER);
  VAR i: INTEGER;
BEGIN (*imprimir árbol t con sangria h*)
  IF t # NIL THEN
    WITH t^ DO
      PrintTree(left, h+1);
      FOR i := 1 TO h DO WriteString("  ") END ;
      WriteInt(key, 6); WriteLn;
      PrintTree(right, h+1)
    END
  END
END PrintTree;
PROCEDURE search(x: INTEGER; VAR p: WPtr);
BEGIN
  IF p = NIL THEN (*la palabra no está en árbol; insertar*)
    ALLOCATE(p, SIZE(Word));
    WITH p^ DO
  
```

```

    key := x; count := 1; left := NIL; right := NIL;
  END
  ELSIF x < p^.key THEN search(x, p^.left)
  ELSIF x > p^.key THEN search(x, p^.right)
  ELSE p^.count := p^.count + 1
  END
END search;

BEGIN root := NIL;
(*el primer entero es el número de nodos*)
OpenInput("TEXT"); ReadInt(n);
WHILE n > 0 DO
  ReadInt(key); search(key, root); n := n-1
END ;
CloseInput; PrintTree(root, 0)
END TreeSearch.
  
```

Programa 4.4 Búsqueda e inserción en árbol.

El uso de un centinela vuelve a simplificar la tarea un poco, como se muestra en (4.50). Con claridad, al inicio del programa la variable *raíz* debe ser inicializada por el apuntador al centinela en vez del valor NIL y antes de que se especifique cada búsqueda el valor *x* debe asignarse al campo de llave del centinela.

```

PROCEDURE search(x: INTEGER; VAR p: WPtr);
BEGIN
  IF x < p^.key THEN search(x, p^.left)
  ELSIF x > p^.key THEN search(x, p^.right)
  ELSIF p # s THEN p^.count := p^.count + 1
  ELSE (*insertar*) Allocate(p, SIZE(Word));
    WITH p^ DO
      key := x; left := s; right := s; count := 1
    END
  END
END
  
```

(4.50)

Aunque el objetivo de este algoritmo es la búsqueda, se puede utilizar también para clasificación. De hecho, se parece a la clasificación por inserción y debido al uso de una estructura de árbol de un arreglo, la necesidad de recolocación de las componentes anteriores al punto de inserción se desvanece. La clasificación de un árbol puede programarse para ser tan efectiva casi como los mejores métodos de clasificación de arreglos que se conocen. Pero deben tomarse algunas precauciones. Después de hallar una coincidencia, el nuevo elemento también debe insertarse. Si el caso $x = p^.key$ se maneja idénticamente al caso $x > p^.key$, entonces el algoritmo representa un método de clasificación estable, es decir, los elementos con llaves idénticas se forman en la misma secuencia cuando se rastrea el árbol en orden normal, tal como se insertaron.

En general, hay mejores maneras de clasificar pero, en aplicaciones en las cuales se necesitan la búsqueda y la clasificación, el algoritmo de búsqueda e inserción en árbol se recomienda grandemente. De hecho, éste se aplica con mucha frecuencia en compiladores y en bancos de datos para organizar los objetos que se almacenarán y recuperarán. Un ejemplo adecuado es la construcción de un índice con referencias cruzadas para un texto determinado, ejemplo que ya se ha utilizado al ilustrar la generación de listas.

Nuestra tarea consiste en construir un programa que (mientras lee un texto y lo imprime después de proporcionar números de línea consecutivos) colecte todas las palabras de este texto, con lo cual retenga los números de las líneas en las cuales ocurrió cada palabra. Cuando este rastreo llega a su fin, se genera una tabla que contiene todas las palabras colectadas en orden alfabético con listas de sus incidencias.

Con claridad, el árbol de búsqueda (también llamado árbol lexicográfico) es un candidato más adecuado para representar las palabras encontradas en el texto. Cada nodo ahora contiene no sólo una palabra como valor de llave, sino también la cabeza de una lista de números de línea. A cada registro de una incidencia lo llamaremos *elemento*. En consecuencia, en este ejemplo se encuentran árboles y listas lineales. El programa consta de dos partes principales (véase el programa 4.5), es decir, la fase de rastreo y la fase de impresión de la tabla. La segunda es una aplicación directa de una rutina de recorrido de árbol en la cual la visita de cada nodo implica la impresión del valor de la llave (palabra) y el rastreo de su lista asociada de números de línea (elementos). Las siguientes son aclaraciones adicionales referentes al generador de referencias cruzadas del programa 4.5. La tabla 4.4 muestra los resultados del procesamiento del texto del programa (4.50).

1. Una palabra es cualquier secuencia de letras y dígitos que comienzan con una letra.
2. Ya que las palabras pueden ser de longitudes muy diferentes, los caracteres reales se almacenan en un arreglo *buffer* y los nodos del árbol contienen el índice del primer carácter de la llave.
3. Es deseable que los números de línea se impriman en orden ascendente en el índice con referencia cruzada. Por tanto, las listas de elementos deben generarse en el mismo orden en que son rastreadas hasta la impresión. Este requisito sugiere el uso de dos apuntadores en cada nodo de la palabra: uno se refiere al primero y otro al último elemento en la lista.

```
MODULE CrossRef;
  FROM InOut IMPORT OpenInput, OpenOutput, CloseInput,
    CloseOutput, Read, Done, EOL, Write, WriteCard, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  CONST BufLeng = 10000; WordLeng = 16;

  TYPE WordPtr = POINTER TO Word;
    ItemPtr = POINTER TO Item;

    Word = RECORD key: CARDINAL;
      first, last: ItemPtr;
      left, right: WordPtr
    END;
```

```
      END;

    Item = RECORD lno: CARDINAL;
      next: ItemPtr
    END;

    VAR root: WordPtr;
      k0, k1, line: CARDINAL;
      ch: CHAR;
      buffer: ARRAY [0 .. BufLeng-1] OF CHAR;

    PROCEDURE PrintWord(k: CARDINAL);
      VAR lim: CARDINAL;
      BEGIN lim := k + WordLeng;
        WHILE buffer[k] > OC DO Write(buffer[k]); k := k + 1 END;
        WHILE k < lim DO Write(" "); k := k + 1 END
      END PrintWord;

    PROCEDURE PrintTree(t: WordPtr);
      VAR i, m: INTEGER; item: ItemPtr;
      BEGIN
        IF t # NIL THEN
          WITH t^ DO
            PrintTree(left);
            PrintWord(key); item := first; m := 0;
            REPEAT
              IF m = 8 THEN
                WriteLn; m := 0;
                FOR i := 1 TO WordLeng DO Write(".") END
              END;
              m := m + 1; WriteCard(item^.lno, 6); item := item^.next
            UNTIL item = NIL;
            WriteLn;
            PrintTree(right)
          END
        END
      END PrintTree;

    PROCEDURE Diff(i, j: CARDINAL): INTEGER;
    BEGIN
      LOOP
        IF buffer[i] # buffer[j] THEN
          RETURN INTEGER(ORD(buffer[i])) - INTEGER(ORD(buffer[j]));
        ELSIF buffer[i] = OC THEN RETURN 0
        END;
        i := i + 1; j := j + 1
      END
    END;
```

```

END Diff;

PROCEDURE search(VAR p: WordPtr);
  VAR item: ItemPtr; d: INTEGER;
BEGIN
  IF p = NIL THEN (*la palabra no esta en arbol; insertar*)
    ALLOCATE(p, SIZE(Word)); ALLOCATE(item, SIZE(Item));
    WITH p DO
      key := k0; first := item; last := item;
      left := NIL; right := NIL
    END;
    item^.lno := line; item^.next := NIL; k0 := k1
  ELSE d := Diff(k0, p^.key);
    IF d < 0 THEN search(p^.left)
    ELSIF d > 0 THEN search(p^.right)
    ELSE ALLOCATE(item, SIZE(Item));
      item^.lno := line; item^.next := NIL;
      p^.last^.next := item; p^.last := item
    END
  END
END search;

PROCEDURE GetWord;
BEGIN k1 := k0;
  REPEAT Write(ch); buffer[k1] := ch; k1 := k1 + 1; Read(ch)
  UNTIL (ch < "0") OR (ch > "9") & (CAP(ch) < "A")
  OR (CAP(ch) > "Z");
  buffer[k1] := 0C; k1 := k1 + 1; (*terminador*)
  search(root)
END GetWord;

BEGIN root := NIL; k0 := 0; line := 0;
  OpenInput("TEXT"); OpenOutput("XREF");
  WriteCard(0, 6); Write(" "); Read(ch);

  WHILE Done DO
    CASE ch OF
      0C .. 35C: Read(ch) |
      36C .. 37C: WriteLn; Read(ch); line := line + 1;
                    WriteCard(line, 6); Write(" ") |
      " " .. "@": Write(ch); Read(ch) |
      "A" .. "Z": GetWord |
      "[" .. "]": Write(ch); Read(ch) |
      "a" .. "z": GetWord |
      "{" .. "}": Write(ch); Read(ch)
    END
  END;

```

```

WriteLn; WriteLn; CloseInput;
PrintTree(root); CloseOutput
END CrossRef.

```

Programa 4.5 Generador de referencias cruzadas.

```

0 PROCEDURE search(x: INTEGER; VAR p: WPtr);
1 BEGIN
2   IF x < p^.key THEN search(x, p^.left)
3   ELSIF x > p^.key THEN search(x, p^.right)
4   ELSIF p # s THEN p^.count := p^.count + 1
5   ELSE Allocate(p, SIZE(Word));
6     WITH p DO
7       key := x; left := s; right := s; count := 1
8     END
9   END
10 END

Allocate          5
BEGIN            1
DO               6
ELSE             5
ELSIF            3   4
END              8   9   10
IF               2
INTEGER          0
PROCEDURE        0
SIZE              5
THEN              2   3   4
VAR               0
WITH              6
WPtr              0
Word              5
count             4   4   7
key               2   3   7
left              2   7
p                 0   2   2   3   3   4   4
                           4
                           5   6
derecha           3   7
s                  4   7   7
izquierda         0   2   3
x                 0   2   2   3   3   3   7

```

Tabla 4.4 Salida muestra del programa 4.5.

4.4.4. Eliminación en un árbol

Ahora consideraremos el problema inverso de la inserción: la *eliminación*. Nuestra tarea consiste en definir un algoritmo para eliminación, es decir, para retirar el nodo con la

llave x en un árbol con llaves ordenadas. Por desgracia, la eliminación de un elemento en general no es tan simple como la inserción. Es directo si el elemento por eliminar es un nodo terminal o uno con un solo descendiente. La dificultad radica en la eliminación de un elemento con dos descendientes, ya que no se puede apuntar en dos direcciones con un solo apuntador. En esta situación, el elemento eliminado será sustituido por el elemento de más a la derecha de su subárbol izquierdo o bien por el nodo de más a la izquierda de su subárbol derecho, los cuales tienen a lo más un descendiente. Los detalles se muestran en el procedimiento recursivo llamado *delete* (4.52.) Este procedimiento distingue tres casos:

1. No hay una componente con llave igual a x.
2. La componente con la llave x tiene a lo más un descendiente.
3. La componente con la llave x tiene dos descendientes.

```

PROCEDURE delete(x: INTEGER; VAR p: Ptr);
  VAR q: Ptr;
  PROCEDURE del (VAR r: Ptr);
  BEGIN
    IF r^.right # NIL THEN del(r^.right)
    ELSE q^.key := r^.key; q^.count := r^.count;
      q := r; r := r^.left
    END
  END del;

  BEGIN (*eliminar*)
    IF p = NIL THEN (*la palabra no esta en el arbol*)
    ELSIF x < p^.key THEN delete(x, p^.left)
    ELSIF x > p^.key THEN delete(x, p^.right)
    ELSE (*eliminar p^*) q := p;
      IF q^.right = NIL THEN p := q^.left
      ELSIF q^.left = NIL THEN p := q^.right
      ELSE del(q^.left)
    END ;
    (*Desasignar (q)*)
  END
END delete

```

El procedimiento recursivo auxiliar *del* se activa en el caso 3 solamente. Desciende a lo largo de la rama de más a la derecha del subárbol izquierdo del elemento *q* que se eliminará y luego sustituye la información relevante (llave y conteo) en *q* por los valores correspondientes de la componente de más a la derecha *r* de ese subárbol izquierdo, desde donde puede disponerse de *r*. El procedimiento no especificado *Deallocate* puede considerarse el inverso o recíproco de *Allocate*. El segundo asigna almacenamiento para una nueva componente, pero el primero puede utilizarse para indicar a un sistema de computación que el almacenamiento ocupado por *q* vuelve a estar libre y se dispone para otros usos.

A fin de ilustrar el funcionamiento del procedimiento (4.52), nos referimos a la figura 4.28. Considérese el árbol (a); después eliminense sucesivamente los nodos con las llaves 13, 15, 5, 10. Los árboles resultantes se muestran en la figura 4.28 (b-e).

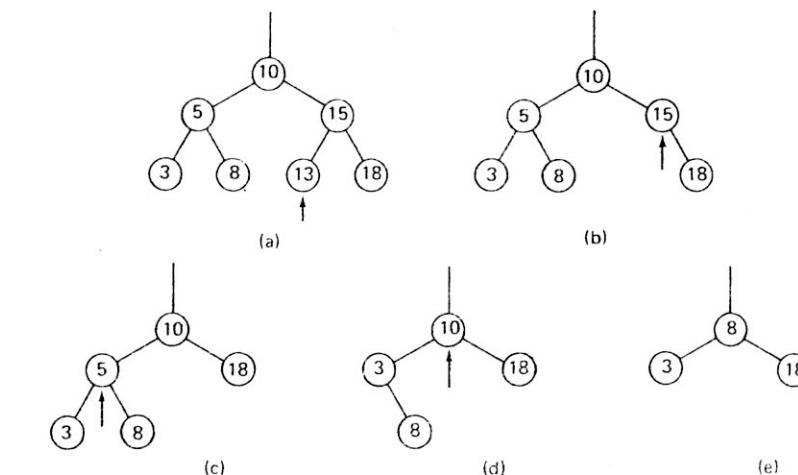


Fig. 4.28 Eliminación en árbol.

4.4.5. Análisis de búsqueda e inserción en árbol

Es una reacción natural sospechar del algoritmo de búsqueda e inserción en un árbol. Por lo menos uno debe mantener cierto escepticismo hasta que se hayan dado algunos detalles más acerca de su comportamiento. Lo que al principio preocupa a muchos programadores es el hecho peculiar de que en general no sabemos cómo crecerá el árbol; no tenemos idea alguna acerca de la forma que tomará. Sólo podemos imaginar que muy probablemente no será el árbol perfectamente balanceado. Ya que el número promedio de comparaciones que se necesitan para localizar una llave en un árbol perfectamente balanceado con n nodos es aproximadamente $\log n$, el número de comparaciones en un árbol generado por este algoritmo será mayor. ¿Pero qué tan mayor?

Antes que nada, es fácil determinar el peor de los casos. Supóngase que todas las llaves llegan en orden aun estrictamente ascendente (o descendente). Después cada llave se une de inmediato a la derecha (o izquierda) de su predecesor y el árbol resultante se vuelve completamente degenerado, o sea, cambia para ser una lista lineal. El esfuerzo promedio de búsqueda es por tanto $n/2$ comparaciones. Este caso, que es el peor, evidentemente nos lleva a un muy pobre desempeño del algoritmo de búsqueda y parece justificar por completo nuestro escepticismo. La pregunta que resta es, desde luego, qué apariencia tendrá esto. Con más precisión, debemos desear conocer la longitud a_n de la trayectoria de búsqueda promediada entre todas las n llaves y todos los $n!$ árboles que se generan a partir de las $n!$ permutaciones de las n llaves originales diferentes. Este problema de análisis algorítmico tiende a ser muy directo y se presenta aquí como un ejemplo común del análisis de un algoritmo así como de la importancia práctica de su resultado.

Dadas n llaves distintas con valores $1, 2, \dots, n$. Supóngase que arrivan en un orden al azar. La probabilidad de la primera llave (que notablemente se convierte en el nodo raíz) que tiene el valor i es $1/n$. Su subárbol izquierdo contendrá por último $i-1$ nodos y su subárbol derecho $n-i$ nodos (véase la figura 4.29). Sea que la longitud de la trayectoria media se denote por a_{i-1} en el subárbol izquierdo y la del subárbol derecho sea a_{n-i} , suponiendo una vez más que todas las permutaciones posibles de las $n-1$ llaves restantes tengan la misma probabilidad. La longitud de trayectoria media en el árbol con n nodos es la suma de los productos del nivel de cada nodo y su probabilidad de acceso. Si se supone que todos los nodos se buscan con la misma probabilidad, entonces

$$a_n = (\text{Si: } 1 \leq i \leq n : p_i) / n \quad (4.53)$$

donde p_i es la longitud de trayectoria del nodo i .

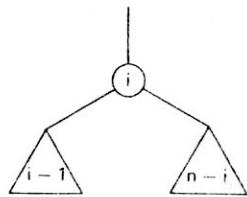


Fig. 4.29 Distribución de ramas por peso.

En el árbol de la figura 4.29 se dividen los nodos en tres clases:

1. Los nodos $i-1$ en el subárbol izquierdo tienen una longitud de trayectoria media a_{i-1}
2. La raíz tiene una longitud de trayectoria de 0.
3. Los nodos $n-i$ en el subárbol derecho tienen una longitud de trayectoria media a_{n-i}

$$a_n(i) = ((i-1)*a_{i-1} + (n-i)*a_{n-i}) / n \quad (4.54)$$

La cantidad deseada a_n es el promedio de $a_n(i)$ para $i = 1 \dots n$, en todos los árboles con la llave $1, 2, \dots, n$ en la raíz.

$$\begin{aligned} a_n &= (\text{Si: } 1 \leq i \leq n : (i-1)*a_{i-1} + (n-i)*a_{n-i}) / n^2 \\ &= 2 * (\text{Si: } 1 \leq i \leq n : (i-1)*a_{i-1}) / n^2 \\ &= 2 * (\text{Si: } 1 \leq i \leq n : i*a_i) / n^2 \end{aligned} \quad (4.55)$$

La ecuación (4.55) es una relación de recurrencia de la forma $a_n = f_1(a_1, a_2, \dots, a_n)$. De esto se deriva una relación de recurrencia más simple de la forma $a_n = f_2(a_{n-1})$. De (4.55) se deriva directamente (1) dividiendo el último término y (2) sustituyendo $n-1$ para n :

$$(1) a_n = 2 * (n-1)*a_{n-1} / n^2 + 2 * (\text{Si: } 1 \leq i \leq n-1 : i*a_i) / n^2$$

$$(2) a_{n-1} = 2 * (\text{Si: } 1 \leq i \leq n-1 : i*a_i) / (n-1)^2$$

Multiplicando (2) por $(n-1)^2/n^2$ se produce

$$(3) 2 * (\text{Si: } 1 \leq i \leq n-1 : i*a_i) / n^2 = a_{n-1} * (n-1)^2/n^2$$

y sustituyendo la parte derecha de (3) en (1), se halla

$$\begin{aligned} a_n &= 2 * (n-1)*a_{n-1} / n^2 + a_{n-1} * (n-1)^2/n^2 \\ &= a_{n-1} * (n-1)^2/n^2 \end{aligned} \quad (4.56)$$

Resulta que a_n puede expresarse en forma cerrada no recursiva en términos de la función armónica

$$\begin{aligned} H_n &= 1 + 1/2 + 1/3 + \dots + 1/n \\ a_n &= 2 * (H_n * (n+1)/n - 1) \end{aligned} \quad (4.57)$$

De la fórmula de Euler (usando la constante de Euler $g = 0.577\dots$)

$$H_n = g + \ln n + 1/12n^2 + \dots$$

se deduce, para n grande, la relación

$$a_n \doteq 2 * (\ln n + g - 1)$$

Como la longitud de trayectoria media en el árbol perfectamente balanceado es aproximadamente

$$a_n' \doteq \log n - 1 \quad (4.58)$$

se obtiene, despreciando los términos constantes que se vuelven insignificantes para n grande,

$$\lim(a_n/a_n') = 2*\ln(n)/\log(n) = 2*\ln(2) \doteq 1.386\dots \quad (4.59)$$

¿Qué nos enseña el resultado (4.59) de este análisis? Nos indica que tomando los contratiempos de construir siempre un árbol perfectamente balanceado en vez del árbol al azar obtenido del programa 4.4, podríamos (siempre que todas las llaves se busquen con la misma probabilidad) esperar una mejora promedio en la longitud de la trayectoria de búsqueda de cuando mucho 39%. Se destacará el promedio de palabras, ya que la mejora puede ser mucho mayor en el caso imprevisto en el cual el árbol generado se ha degenerado completamente hasta una lista que casi nunca ocurrirá. En este aspecto es digno de atención que la longitud de trayectoria media estimada del árbol generado al azar crezca también estrictamente en forma logarítmica con el número de sus nodos, aunque la longitud de la trayectoria en el peor de los casos crezca de manera lineal.

La cifra de 39% impone un límite sobre la cantidad de esfuerzo adicional que puede hacerse útil en cualquier clase de reorganización de la estructura del árbol hasta la inserción de llaves. Naturalmente, la razón entre las frecuencias de acceso (recuperación) de nodos (información) y de inserción (actualización) influencia significativamente los límites de utilidad de cualquier compromiso de este tipo. Mientras mayor es esta razón, mayor es la utilidad de un procedimiento de reorganización. La cifra de 39% es lo suficientemente baja para que en muchas mejoras de aplicaciones del algoritmo de inserción directa en árbol no tenga rendimiento a menos que el número de nodos y el acceso contra la razón de inserción sean grandes.

4.5. ARBOLES BALANCEADOS

En la explicación precedente se advierte que un procedimiento de inserción, el cual siempre re establece la estructura de los árboles en un equilibrio perfecto, apenas si podrá ser rentable pues el restablecimiento del equilibrio perfecto trae una inserción aleatoria es una operación bastante intrincada. Las mejoras posibles se encuentran en la formulación de definiciones menos estrictas del equilibrio. Tales criterios imperfectos del equilibrio darán origen a procedimientos más sencillos en la reorganización de los árboles, con un ligero deterioro del rendimiento promedio de la búsqueda. Esta definición del equilibrio ha sido propuesta por Adelson-Velski y Landis [4.1]. He aquí el criterio de equilibrio:

Un árbol está *balanceado* si y sólo si en cada nodo las alturas de sus dos subárboles difieren a lo máximo en 1.

Los árboles que satisfacen esta condición suelen recibir el nombre de árboles AVL (en honor de sus inventores). Aquí los llamaremos simplemente *árboles balanceados* pues este criterio de equilibrio parece ser el más adecuado. (Nótese que todos los árboles perfectamente balanceados también son árboles AVL balanceados.)

La definición anterior no sólo es simple, sino que además nos lleva a un procedimiento de rebalanceo manejable y a una longitud promedio de trayectoria de búsqueda que es prácticamente idéntica a la del árbol perfectamente balanceado. Las siguientes operaciones pueden efectuarse en árboles balanceados en $O(\log n)$ unidades de tiempo, aun en el peor caso:

1. Localizar un nodo con determinada llave.
2. Insertar un nodo con determinada llave.
3. Eliminar el nodo con una llave determinada.

Los enunciados anteriores son consecuencia directa de un teorema demostrado por Adelson-Velski y Landis, el cual garantiza que un árbol balanceado nunca tendrá una altura mayor que el 45% respecto a su equivalente perfectamente balanceado, por muchos nodos que haya. Si denotamos la altura de un árbol balanceado con n nodos por $h_b(n)$, entonces

$$\log(n+1) \leq h_b(n) < 1.4404 \cdot \log(n+2) - 0.328 \quad (4.60)$$

Desde luego se alcanza el óptimo si el árbol está perfectamente balanceado para $n = 2^k - 1$. ¿Pero cuál es la estructura del *peor* árbol balanceado AVL? A fin de encontrar la altura máxima h de todos los árboles balanceados con n nodos, consideremos una altura fija h y tratemos de construir el árbol balanceado con el número mínimo de nodos. Se recomienda esta estrategia porque, como en el caso de la altura mínima, el valor puede obtenerse sólo para valores específicos de n . Denotemos por T_h este árbol de altura h . Sin duda, T_0 es el árbol vacío y T_1 es el árbol con un solo nodo. A fin de construir el árbol T_h

para $h > 1$, proporcionaremos a la raíz dos subárboles que también tengan un número mínimo de nodos. De ahí que los subárboles también sean T . Está claro que un subárbol debe tener la altura $h-1$, y entonces se permite que el otro tenga una altura menor, esto es, $h-2$. La figura 4.30 muestra los árboles con altura 2, 3 y 4. Puesto que su principio de composición se asemeja mucho al de los números de Fibonacci, se les llama *árboles de Fibonacci*. Se definen del modo siguiente:

1. El árbol vacío es el árbol de Fibonacci con altura 0.
2. El nodo individual es el árbol de Fibonacci con altura 1.
3. Si T_{h-1} y T_{h-2} son árboles de Fibonacci con altura $h-1$ y $h-2$, entonces $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ es un árbol de Fibonacci.
4. No hay otros árboles que sean árboles de Fibonacci.

El número de nodos de T_h se define por la siguiente relación de recurrencia:

$$\begin{aligned} N_0 &= 0, \quad N_1 = 1 \\ N_h &= N_{h-1} + 1 + N_{h-2} \end{aligned} \quad (4.61)$$

Los N_i son los números de nodos en los cuales puede darse el peor caso (límite superior de h) de (4.60) y se les llama *números de Leonardo*.

4.5.1. Inserción en árboles balanceados

Consideremos ahora lo que sucede cuando un nuevo nodo se inserta en un árbol balanceado. Pueden distinguirse tres casos en una raíz r con subárboles de izquierda y derecha L y R . Supongamos que el nuevo nodo se inserta en L haciendo que su altura aumente en 1:

1. $h_L = h_R$: L y R tendrán una altura desigual, pero sin que se viole el criterio de equilibrio.
2. $h_L < h_R$: L y R tendrán igual altura, esto es, el equilibrio ha mejorado incluso.
3. $h_L > h_R$: el criterio de equilibrio se viola y hay que reestructurar el árbol.

Examinemos detenidamente la figura 4.31. Los nodos con las llaves 9 y 11 pueden insertarse sin rebalanceo; el árbol con la raíz 10 será de un solo lado (caso 1); el de la raíz 8 mejorará su equilibrio (caso 2). La inserción de los nodos 1, 3, 5 o 7 requiere balanceo ulterior.

Un cuidadoso análisis de la situación revela que hay dos constelaciones esencialmente diferentes que requieren tratamiento individual. Las restantes pueden derivarse de esas

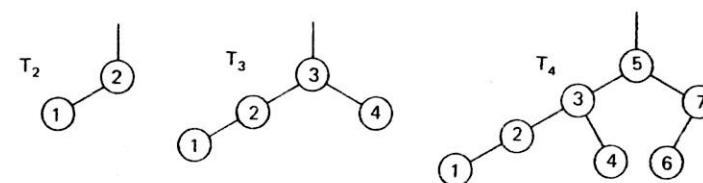


Fig. 4.30 Árboles de Fibonacci de altura 2, 3 y 4.

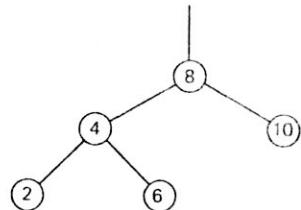


Fig. 4.31 Arbol balanceado.

dos por medio de consideraciones simétricas. El caso 1 se caracteriza por el hecho de insertar las llaves 1 o 3 en el árbol de la figura 4.31; el caso dos por la inserción de los nodos 5 o 7.

Los dos casos se generalizan en la figura 4.32, en la cual las casillas rectangulares denotan subárboles, y la altura agregada con la inserción se indica con cruces. Las simples transformaciones de las dos reestructuras reestablecen el equilibrio deseado. Su resultado se muestra en la figura 4.33; nótese que sólo los movimientos permitidos son los que ocurren en la dirección vertical, en tanto que permanecen inalteradas las posiciones horizontales relativas de los nodos y subárboles mostrados.

Un algoritmo de la inserción y rebalanceo depende principalmente de la manera en que se guarda la información referente al equilibrio del árbol. Una solución extrema consiste en conservar esa información enteramente implícita en la estructura del árbol. En este caso, hay que redescubrir un factor de balance del nodo cada vez que se vea afectado por una inserción, lo cual produce "excesos" excesivamente altos. El otro extremo consiste en atribuir a cada nodo un factor de balance explicitamente almacenado. La definición (4.48) del tipo *Node* se extiende a

```

TYPE Ptr = POINTER TO Node;
TYPE Balance = [-1 .. +1];
TYPE Node = RECORD key: INTEGER;
            count: INTEGER;
            left, right: Ptr;
            bal: Balance;
            END;
  
```

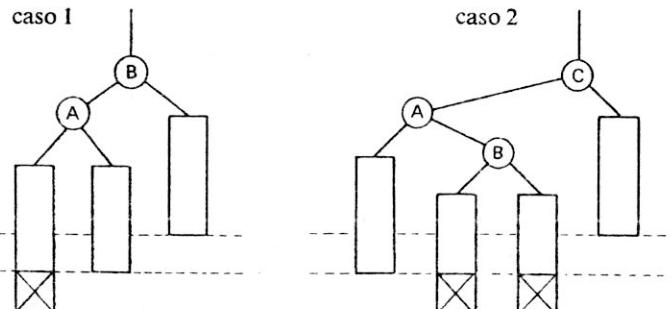
(4.62)


Fig. 4.32 Desequilibrio resultante de la inserción.

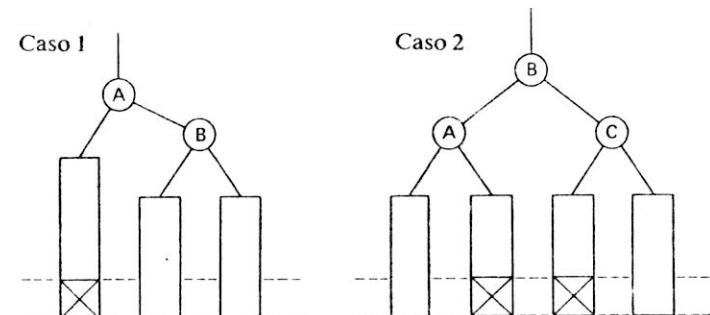


Fig. 4.33 Restablecimiento del equilibrio.

```

left, right: Ptr
bal: Balance
END
  
```

Más adelante interpretaremos el factor de balance (equilibrio) de un nodo como la altura de su subárbol de la derecha menos la de su subárbol de la izquierda; basaremos el algoritmo resultante en el tipo de nodo (4.62). El proceso de la inserción de nodos consta esencialmente de tres partes consecutivas:

1. Seguir la trayectoria de búsqueda hasta verificar que la llave no esté ya (que ya existe) en el árbol.
2. Insertar el nodo y determinar el resultante factor de equilibrio.
3. Retroceder a lo largo de la trayectoria de búsqueda y verificar el factor de equilibrio en cada nodo. Se hace el rebalanceo en caso necesario.

Aunque este método requiere un poco de verificación redundante (una vez establecido el equilibrio, no es necesario comprobar los ancestros del nodo), primero seguiremos este esquema evidentemente correcto ya que puede utilizarse mediante una simple extensión del procedimiento ya conocido de búsqueda e inserción que aparece en el programa 4.4. Este procedimiento describe la operación de búsqueda que se requiere en cada nodo, y por su formulación recursiva le es fácil admitir una operación más al retornar a lo largo de la trayectoria de búsqueda. En cada paso, hay que transmitir información sobre si ha aumentado la altura del subárbol (en el cual se realizó la inserción). Por tanto, aplicamos la lista de parámetros del procedimiento mediante la *h* booleana, lo cual significa que *ha aumentado la altura del subárbol*. Es evidente que *h* debe denominar un parámetro variable puesto que se utiliza para transmitir un resultado.

Supongamos ahora que el proceso está retornando a un nodo *p†* a partir de la rama de la izquierda (véase la figura 4.32), con la indicación de que ha aumentado su altura. A continuación hemos de distinguir entre las tres condiciones relativas a la altura del subárbol antes de la inserción:

1. $h_L < h_R$, $p \uparrow .bal = +1$, el desequilibrio anterior en p ha sido equilibrado.
2. $h_L = h_R$, $p \uparrow .bal = 0$, el peso se inclina ahora a la izquierda.
3. $h_L > h_R$, $p \uparrow .bal = -1$, se necesita el rebalanceo.

En el tercer caso, con la observación del factor de equilibrio de la raíz del subárbol de la izquierda (digamos, $p1 \uparrow .bal$) se determina si el caso 1 o el caso 2 de la figura 4.32 están presentes. Si ese nodo tiene además un subárbol de la izquierda más alto que el de la derecha, tendremos que ocuparnos del caso 1; de lo contrario, nos ocuparemos del caso 2. (Convénzase el lector de que un subárbol de la izquierda con un factor de equilibrio igual a 0 en su raíz no puede presentarse en este caso.) Las operaciones de rebalanceo necesarias se expresan enteramente como secuencias de las reasignaciones de apuntadores. En efecto, éstos se intercambian cíclicamente y dan por resultado una rotación sencilla o doble de los dos o tres nodos que intervienen. Además de la rotación de apuntadores, hay que actualizar los respectivos factores del equilibrio de nodos. Los detalles se muestran en el procedimiento de búsqueda, inserción y rebalanceo (4.63).

El principio que se aplica aquí se advierte en la figura 4.34. Consideremos un árbol binario (a) que consta de dos nodos únicamente. La inserción de la llave 7 primero nos da un árbol desbalanceado (esto es, una lista lineal). Su balanceo requiere una sola rotación RR, que produce el árbol perfectamente balanceado (b). Una inserción adicional de los nodos 2 y 1 da origen a un desequilibrio en el subárbol de raíz 4. Este está balanceado por una simple rotación LL (d). La subsiguiente inserción de la llave 3 de inmediato viola el criterio de equilibrio en la raíz del nodo 5. Después el equilibrio se restablece con una doble rotación más complicada LR; el resultado es el árbol (3). El único candidato para perder el equilibrio tras la siguiente inserción es el nodo 5. En efecto, la inserción del nodo 6 debe invocar el cuarto caso de rebalanceo descrito en (4.63), o sea la doble rotación RL. El árbol final se observa en la figura 4.34(f).

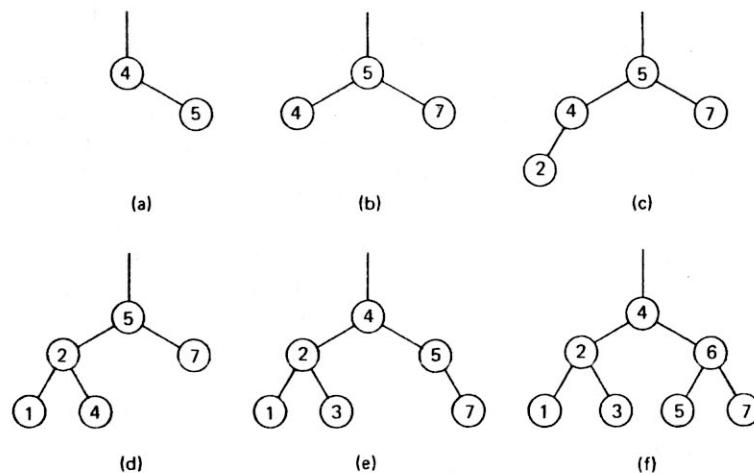


Fig. 4.34 Inserciones en árbol balanceado.

PROCEDURE search(x: INTEGER; VAR p: Ptr; VAR h: BOOLEAN);

VAR p1, p2: Ptr; (*~h*)

BEGIN

IF p = NIL THEN (*insertar*)

ALLOCATE(p, SIZE(Node)); h := TRUE;

WITH p DO

key := x; count := 1; left := NIL; right := NIL; bal := 0

END

ELSIF p1.key > x THEN

search(x, p1.left, h);

IF h THEN (*la rama de la izquierda creció*)

CASE p1.bal OF

1: p1.bal := 0; h := FALSE |

0: p1.bal := -1 |

-1: (*rebalancear*) p1 := p1.left;

IF p1.bal = -1 THEN (*rotacion simple LL*)

p1.left := p1.right; p1.right := p;

p1.bal := 0; p := p1

ELSE (*rotacion doble LR*) p2 := p1.right;

p1.right := p2.left; p2.left := p1;

p1.left := p2.right; p2.right := p;

IF p2.bal = -1 THEN p1.bal := 1 ELSE p1.bal := 0 END ;

IF p2.bal = +1 THEN p1.bal := -1 ELSE p1.bal := 0 END ;

p := p2

END ;

p1.bal := 0; h := FALSE

END

END

ELSIF p1.key < x THEN

search(x, p1.right, h);

IF h THEN (*ha crecido la rama de la derecha*)

CASE p1.bal OF

-1: p1.bal := 0; h := FALSE |

0: p1.bal := 1 |

1: (*rebalancear*) p1 := p1.right;

IF p1.bal = 1 THEN (*rotacion simple RR*)

p1.right := p1.left; p1.left := p;

p1.bal := 0; p := p1

ELSE (*rotacion doble RL*) p2 := p1.left;

p1.left := p2.right; p2.right := p1;

p1.right := p2.left; p2.left := p;

IF p2.bal = +1 THEN p1.bal := -1 ELSE p1.bal := 0 END ;

IF p2.bal = -1 THEN p1.bal := 1 ELSE p1.bal := 0 END ;

p := p2

(4.63)

```

    END ;
    pt.bal := 0; h := FALSE
  END
  END
ELSE pt.count := pt.count + 1
END
END search

```

He aquí dos preguntas particularmente interesantes sobre el rendimiento del algoritmo para inserción de árboles balanceados:

1. Si todas las $n!$ permutaciones de n llaves ocurren con igual probabilidad, ¿cuál es la altura esperada del árbol balanceado construido?
2. ¿Cuál es la probabilidad de que una inserción requiera rebalanceo?

El análisis matemático de este algoritmo tan complicado todavía constituye un problema sin resolver. Las pruebas empíricas apoyan la conjectura de que la altura esperada del árbol balanceado que se genera con (4.63) es $h = \log(n) + c$, donde c es una constante pequeña ($c = 0.25$). Ello significa que, en la práctica, el árbol balanceado AVL da un rendimiento tan satisfactorio como el árbol perfectamente balanceado, aunque es mucho más sencillo de mantener. Los datos empíricos también señalan que, en general, el rebalanceo es necesario sólo aproximadamente en cada dos inserciones. Aquí las rotaciones simples y dobles tienen igual probabilidad. El ejemplo de la figura 4.34 sin duda fue escogido con mucho cuidado para demostrar el mayor número posible de rotaciones con un mínimo de inserciones.

La complejidad de las operaciones de balanceo indica que los árboles balanceados deben utilizarse sólo si las recuperaciones de información son mucho más frecuentes que las inserciones. Ello se debe sobre todo a que los nodos de esos árboles de búsqueda suelen instrumentarse como registros densamente compactados a fin de ahorrar espacio. La velocidad de acceso y la de actualización de los factores de balanceo (ambas requieren dos bits solamente) suelen ser un factor decisivo en la eficiencia de la operación de rebalanceo. Las evaluaciones empíricas muestran que los árboles balanceados pierden gran parte de su atractivo si es obligatoria una fuerte compactación en los registros. En efecto, es difícil superar el algoritmo sencillo de inserción en el árbol.

4.5.2. Eliminación en árboles balanceados

Nuestra experiencia con la eliminación en árboles nos revela que, en el caso de los árboles balanceados, esa maniobra será más complicada que la inserción. Y así es, aunque la operación de rebalanceo sigue siendo, en lo esencial, idéntica a la de inserción. En particular, el rebalanceo consta de una rotación simple o doble de los nodos.

La base de la eliminación en árboles balanceados es el algoritmo (4.52). Los casos fáciles son los nodos terminales y los que tienen un solo descendiente. Si el nodo que debe suprimirse tiene dos subárboles, otra vez lo reemplazaremos con el nodo del extremo derecho de su subárbol de la izquierda. Como en el caso de la inserción (4.63), se agrega un parámetro de variables booleanas h , lo cual significa que la altura del subárbol ha sido

reducida. El rebalanceo debe considerarse sólo cuando h sea verdadera. Se hace que h sea verdadera luego de encontrar y suprimir un nodo o bien si con el rebalanceo se disminuye la altura de un subárbol. En (4.64) introducimos las dos operaciones (simétricas) de balanceo mediante procedimientos, porque es preciso invocarlos desde más de un punto en el algoritmo de eliminación. Nótese que *balanceL* se aplica cuando se ha reducido la altura de la rama izquierda *balanceR* después de la derecha.

La operación del procedimiento se muestra en la figura 4.35. En un árbol balanceado (a), la eliminación sucesiva de los nodos con las llaves 4, 8, 6, 5, 2, 1 y 7 produce los árboles (b) ... (h). La eliminación de la llave 4 es fácil pues representa un nodo terminal. Sin embargo, produce un nodo desbalanceado 3. Su operación de rebalanceo requiere una simple rotación LL. El rebalanceo se torna necesario otra vez tras la eliminación del nodo 6. Esta vez el subárbol de la derecha de la raíz (7) se rebalancea mediante una simple

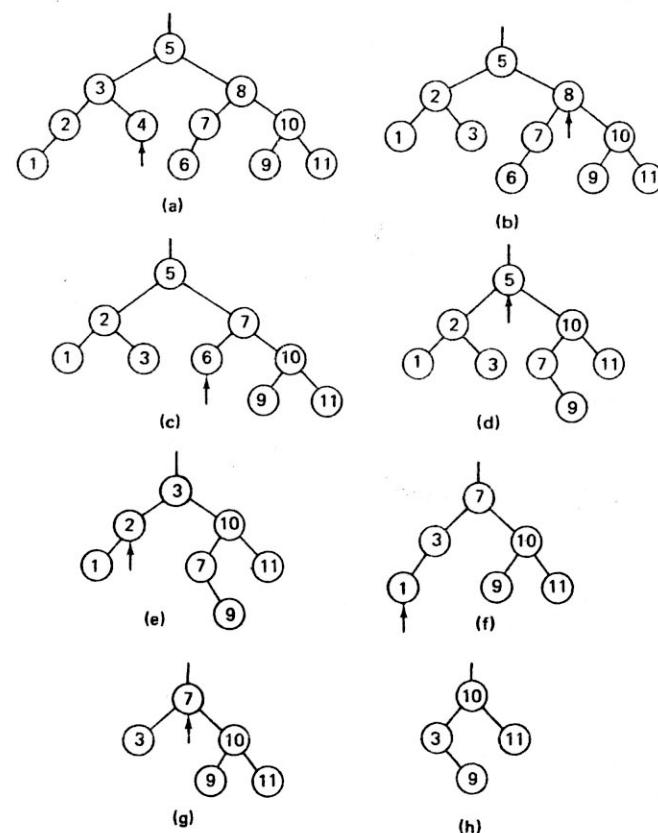


Fig. 4.35 Eliminaciones en árbol balanceado.

rotación RR. La eliminación del nodo 2, aunque también sencilla por tener un solo descendiente, requiere una rotación doble y complicada RL. El cuarto caso, una rotación doble LR, se invoca por último luego de suprimir el nodo 7, que al inicio fue sustituido por el elemento del extremo derecho de su subárbol de la izquierda, o sea por el nodo con la llave 3.

```

PROCEDURE balanceL(VAR p: Ptr; VAR h: BOOLEAN);
  VAR p1, p2: Ptr; b1, b2: Balance;
BEGIN (*h; la rama de la izquierda se ha encogido*)
  CASE pt.bal OF
    -1: pt.bal := 0 |
    0: pt.bal := 1; h := FALSE |
    1: (*rebalanceo*) p1 := pt.right; b1 := plt.bal;
        IF b1 >= 0 THEN (*rotacion simple RR*)
          pt.right := plt.left; plt.left := p;
          IF b1 = 0 THEN pt.bal := 1; plt.bal := -1; h := FALSE
          ELSE pt.bal := 0; plt.bal := 0
        END ;
        p := p1
      ELSE (*rotacion doble RL*)
        p2 := plt.left; b2 := p2t.bal;
        plt.left := p2t.right; p2t.right := plt;
        pt.right := p2t.left; p2t.left := p;
        IF b2 = +1 THEN pt.bal := -1 ELSE pt.bal := 0 END ;
        IF b2 = -1 THEN plt.bal := 1 ELSE plt.bal := 0 END ;
        p := p2; p2t.bal := 0
      END
    END
  END balanceL;

PROCEDURE balanceR(VAR p: Ptr; VAR h: BOOLEAN);
  VAR p1, p2: Ptr; b1, b2: Balance;
BEGIN (*h; la rama de la derecha se ha encogido*)
  CASE pt.bal OF
    1: pt.bal := 0 |
    0: pt.bal := -1; h := FALSE |
    -1: (*rebalanceo*) p1 := pt.left; b1 := plt.bal;
        IF b1 <= 0 THEN (*rotacion simple LL*)
          pt.left := plt.right; plt.right := p;
          IF b1 = 0 THEN pt.bal := -1; plt.bal := 1; h := FALSE
          ELSE pt.bal := 0; plt.bal := 0
        END ;
        p := p1
      ELSE (*rotacion doble RL*)
        p2 := plt.right; b2 := p2t.bal;
        plt.right := p2t.left; p2t.left := plt;
        pt.left := p2t.right; p2t.right := pt;
        IF b2 = +1 THEN pt.bal := 1 ELSE pt.bal := 0 END ;
        IF b2 = -1 THEN plt.bal := -1 ELSE plt.bal := 0 END ;
        p := p2; p2t.bal := 0
      END
    END
  END balanceR;

```

(4.64)

```

pt.left := p2t.right; p2t.right := p;
IF b2 = -1 THEN pt.bal := 1 ELSE pt.bal := 0 END ;
IF b2 = +1 THEN plt.bal := -1 ELSE plt.bal := 0 END ;
p := p2; p2t.bal := 0
END
END balanceR;

PROCEDURE delete(x: INTEGER; VAR p: Ptr; VAR h: BOOLEAN);
  VAR q: Ptr;
PROCEDURE del(VAR r: Ptr; VAR h: BOOLEAN);
BEGIN (*~h*)
  IF rt.right # NIL THEN
    del(rt.right, h);
    IF h THEN balanceR(r, h) END
  ELSE qt.key := rt.key; qt.count := rt.count;
    q := r; r := rt.left; h := TRUE
  END
END del;
BEGIN (*~h*)
  IF p = NIL THEN (*la llave no esta en el arbol*)
  ELSIF pt.key > x THEN
    delete(x, pt.left, h);
    IF h THEN balanceL(p, h) END
  ELSIF pt.key < x THEN
    delete(x, pt.right, h);
    IF h THEN balanceR(p, h) END
  ELSE (*eliminar pt*) q := p;
    IF qt.right = NIL THEN p := qt.left; h := TRUE
    ELSIF qt.left = NIL THEN p := qt.right; h := TRUE
    ELSE del(qt.left, h);
      IF h THEN balanceL(p, h) END
    END ;
    (*desasignar (q)*)
  END
END delete

```

Por fortuna, la eliminación de un elemento en un árbol balanceado también puede hacerse con $O(\log n)$ operaciones en el peor caso. Una diferencia fundamental entre el comportamiento de los procedimientos de inserción y los de eliminación no debe pasar inadvertida. Mientras que la inserción de una sola llave puede producir a lo máximo una rotación (de dos o tres nodos), la eliminación puede requerir una rotación en cada nodo a lo largo de la trayectoria de búsqueda. Pongamos el caso, por ejemplo, de la supresión del nodo del extremo derecho en un árbol de Fibonacci. En este caso, la eliminación de un nodo cualquiera lleva a reducir la altura del árbol; además, la eliminación de su nodo

del extremo derecho requiere el número máximo de rotaciones. Así pues, esto representa la elección menos adecuada del nodo en el peor caso de un árbol balanceado, más que una afortunada combinación de probabilidades. Cabe preguntar: ¿qué grado de probabilidad tienen en general las rotaciones?

El resultado sorprendente de las pruebas empíricas consiste en que, mientras que una rotación sea invocada aproximadamente cada dos inserciones, se necesita una para cada cinco eliminaciones. Así pues, la eliminación en árboles balanceados es tan fácil (o complicada) como la inserción.

4.6. ARBOLES DE BUSQUEDA OPTIMOS

Hasta ahora nuestro estudio de la organización de los árboles de búsqueda se ha basado en la suposición de que la frecuencia de acceso es igual en todos los nodos, o sea que todas las llaves tienen la misma probabilidad de ocurrir como un argumento de búsqueda. Esa es quizás la mejor suposición si no se tiene idea de la distribución de los accesos. Sin embargo, hay casos (son la excepción, no la regla) en que la información relativa a las probabilidades de acceso a llaves individuales está disponible. Tales casos suelen tener la característica de que las llaves siempre permanecen inalteradas, esto es, el árbol de búsqueda no está sujeto ni a inserción ni a eliminación, sino que conserva una estructura constante. Un ejemplo típico es el rastreador (escáner) de un compilador que determina para cada palabra (identificador) si es o no una palabra reservada. Las mediciones estadísticas hechas en cientos de programas compilados puede entonces aportar información confiable sobre las frecuencias relativas de ocurrencia y, por lo mismo, también proporciona información sobre el acceso de llaves individuales.

Supongamos que en un árbol de búsqueda la probabilidad con que se accede al nodo i es

$$\Pr \{x = k_i\} = p_i, \quad (\text{Si: } 1 \leq i \leq n : p_i) = 1 \quad (4.65)$$

Ahora queremos organizar el árbol de búsqueda de modo tal que el número total de pasos (contados luego de un número suficiente de ensayos) sea el mínimo. A tal efecto la definición de la longitud de trayectoria (4.34) se modifica 1) atribuyendo cierto peso a cada nodo y 2) suponiendo que la raíz se halla en el nivel 1 (en vez de 0), pues representa la primera comparación a lo largo de la trayectoria de búsqueda. Los nodos a los cuales se accede frecuentemente se convierten en nodos pesados; los que se visitan rara vez se convierten en nodos ligeros. La *longitud de trayectoria pesada* (interna) es la suma de todas las que van de la raíz a cada nodo, ponderadas por la probabilidad de acceso del nodo

$$P = \sum_{i=1}^n p_i * h_i \quad (4.66)$$

h_i es el nivel del nodo i . La meta consiste ahora en minimizar la longitud de trayectoria pesada para determinada distribución de probabilidad. A manera de ejemplo, consideremos el conjunto de llaves 1, 2, 3, con probabilidades de acceso $p_1 = 1/7$, $p_2 = 2/7$ y $p_3 = 4/7$. Esas tres llaves pueden arreglarse en cinco formas diferentes como árboles de búsqueda (véase la figura 4.36).

Las longitudes de trayectoria pesadas de los árboles (a) a (e) se calculan conforme a (4.66) así:

$$P(a) = 11/7, P(b) = 12/7, P(c) = 12/7, P(d) = 15/7, P(e) = 17/7$$

Por tanto, en este ejemplo resulta ser óptimo no el árbol perfectamente balanceado (c) sino el árbol degenerado (a).

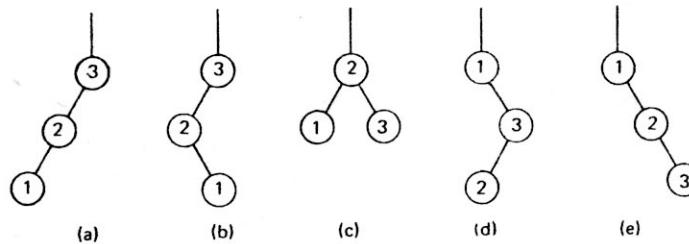


Fig. 4.36 Árboles de búsqueda con tres nodos.

El ejemplo del rastreador del compilador sugiere de inmediato que este problema debiera verse bajo una condición un poco más general: las palabras que figuran en el texto fuente no son siempre palabras reservadas; en realidad, el hecho de que sean palabras reservadas constituye la excepción. El hecho de encontrar que determinada palabra k no es una llave en el árbol de búsqueda debe considerarse como un acceso a un “nodo especial” hipotético, insertado entre la siguiente llave más baja y más alta (véase la figura 4.19), con la longitud de trayectoria externa. Si la probabilidad q_i de un argumento de búsqueda x , situada entre las dos llaves k_i y k_{i+1} , también se conoce, esta información puede modificar radicalmente la estructura del árbol de búsqueda óptimo. De ahí que, para generalizar el problema, se tomen en cuenta también las búsquedas infructuosas. La longitud promedio global de la trayectoria pesada es ahora

$$P = (\text{Si: } 1 \leq i \leq n : p_i * h_i) + (\text{Si: } 0 \leq j \leq m : q_j * h'_j) \quad (4.67)$$

donde

$$(\text{Si: } 1 \leq i \leq n : p_i) + (\text{Si: } 0 \leq j \leq m : q_j) = 1$$

y donde h_i es el nivel del nodo (interno) i y donde h'_j es el nivel del nodo externo j . La longitud promedio de trayectoria ponderada puede llamarse el *costo* del árbol de búsqueda, pues representa una medida de la magnitud prevista del esfuerzo dedicado a la búsqueda. El árbol de búsqueda que requiere el mínimo costo entre todos los árboles con determinado conjunto de llaves k_i y probabilidades p_i recibe el nombre de *árbol óptimo*.

Para encontrar el árbol óptimo, no hay necesidad de requerir que las p y las q sumen 1. En efecto, esas probabilidades se determinan generalmente mediante experimentos en los cuales se cuentan los accesos a los nodos. En lugar de utilizar las probabilidades p_i y q_j , emplearemos en lo sucesivo esos conteos de frecuencia y los denotaremos por

a_i = número de veces que el argumento de búsqueda x es igual a k_i

b_j = número de veces que el argumento de búsqueda x se encuentra entre k_j y k_{j+1}

Por convención, b_0 es el número de veces que x es menor que k_1 y b_n es la frecuencia con que x es mayor que k_n (véase la figura 4.37). Más adelante usaremos P para denotar la *longitud acumulada de trayectoria pesada* en vez de la longitud promedio de trayectoria:

$$P = (\text{Si: } 1 \leq i \leq n : a_i * h_i) + (\text{Si: } 0 \leq j \leq m : b_j * h'_j) \quad (4.68)$$

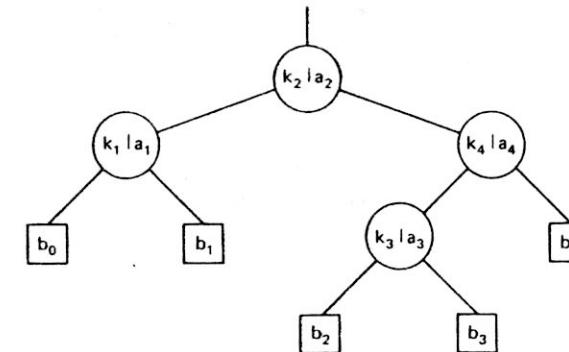


Fig. 4.37 Árboles de búsqueda con frecuencias de acceso asociadas.

Así pues, además de evitar el cálculo de las probabilidades a partir de los conteos de la frecuencia medida, logramos la ventaja adicional de poder emplear enteros en lugar de fracciones en la búsqueda del árbol óptimo.

Si recordamos el hecho de que el número de posibles configuraciones de n nodos crece exponencialmente con n , la tarea de encontrar el óptimo parece imposible en el caso de n grande. Sin embargo, los árboles óptimos tienen una propiedad importante que ayuda a encontrarlos: todos sus subárboles son óptimos también. Por ejemplo, si el árbol en la figura 4.37 es óptimo, el subárbol con las llaves k_3 y k_4 lo es también. Esta propiedad indica un algoritmo que sistemáticamente encuentra árboles cada vez más grandes, comenzando con nodos individuales como los subárboles más pequeños posibles. Y así el árbol crece de las hojas hacia la raíz que es la dirección ascendente [4-6], pues estamos acostumbrados a dibujar los árboles al revés [4-6].

La ecuación que sirve de llave a este algoritmo es (4.69). Sea P la longitud de trayectoria pesada (ponderada) de un árbol y sean P_L y P_R las de los subárboles de la izquierda y derecha de su raíz. Está claro que P es la suma de P_L y P_R y el número de veces que una búsqueda pasa de la rama a la raíz, que es simplemente el número total W de los intentos de búsqueda. A W lo llamamos el *peso* del árbol. Su longitud promedio de trayectoria es, pues, P/W .

$$P = P_L + W + P_R \quad (4.69)$$

$$W = (\text{Si: } 1 \leq i \leq n : a_i) + (\text{Si: } 0 \leq j \leq m : b_j) \quad (4.70)$$

Las consideraciones anteriores muestran la necesidad de una denotación de los pesos y las longitudes de trayectoria de todo árbol que conste de varias llaves contiguas. Sea T_{ij} el subárbol óptimo constituido por los nodos con las llaves $k_{i+1}, k_{i+2}, \dots, k_j$. A continuación w_{ij} denota el peso y p_{ij} denota la longitud de trayectoria de T_{ij} . Evidentemente, $P = p_{0,n}$ y $W = w_{0,n}$. Estas cantidades se definen mediante las relaciones de recurrencia (4.71) y (4.72)

$$w_{ii} = b_i \quad (0 \leq i \leq n) \quad (4.71)$$

$$w_{ij} = w_{i,j-1} + a_j + b_j \quad (0 \leq i < j \leq n)$$

$$p_{ii} = w_{ii} \quad (0 \leq i \leq n) \quad (4.72)$$

$$p_{ij} = w_{ij} + \min k : i < k \leq j : (p_{i,k-1} + p_{kj}) \quad (0 \leq i < j \leq n)$$

La última ecuación se deduce inmediatamente de (4.69) y de la definición de óptimo. Puesto que hay unos $n^2/2$ valores p_{ij} y (4.72) exige una elección entre todos los casos tal que $0 < j-i \leq n$, la operación de minimización incluirá aproximadamente $n^3/6$ operaciones. Knuth señaló cómo puede eliminarse un factor n , lo que por sí solo salva este algoritmo desde un punto de vista práctico.

Sea r_{ij} el valor de k que logra el mínimo en (4.72). Es posible limitar la búsqueda de r_{ij} a un intervalo mucho menor, o sea reducir el número de los pasos de la evaluación $j-1$. Para ello la clave es la observación de que, si hemos encontrado la raíz r_{ij} del subárbol óptimo T_{ij} , ni ampliar el árbol agregando un nodo a la derecha ni acortarlo suprimiendo el nodo del extremo izquierdo podrá hacer que la raíz óptima se mueva a la izquierda. Esto se expresa con la relación

$$r_{i,j-1} \leq r_{ij} \leq r_{i+1,j} \quad (4.73)$$

que limita la búsqueda de posibles soluciones de r_{ij} al rango $r_{i,j-1} \dots r_{i+1,j}$. Esto da por resultado el número total de pasos elementales en el orden de n^2 .

Y ya estamos listos para construir con detalle el algoritmo de optimización. Recorremos la siguiente definición, basada en los árboles óptimos T_{ij} que constan de nodos con las llaves $k_{i+1} \dots k_j$.

1. a_j : la frecuencia de una búsqueda de k_j .
2. b_j : la frecuencia de un argumento de búsqueda x entre k_j y k_{j+1} .
3. w_{ij} : el peso de T_{ij} .
4. p_{ij} : la longitud de trayectoria pesada de T_{ij} .
5. r_{ij} : el índice de la raíz de T_{ij} .

Dado que

TYPE index = [0 .. n]

declaramos los siguientes arreglos:

a: ARRAY [1 .. n] OF CARDINAL;	
b: ARRAY index OF CARDINAL;	
p,w: ARRAY index, index OF CARDINAL;	
r: ARRAY index, index OF index	(4.74)

Supongamos que los pesos w_{ij} se han calculado de a y b en forma sencilla [véase (4.71)]. Ahora consideremos w como el argumento del procedimiento *OptTree* (árbol óptimo) que debemos desarrollar y r como su resultado, pues r describe enteramente la estructura del árbol. Podemos pensar que p es un resultado intermedio. Comenzando

por considerar los subárboles más pequeños, a saber los que no tienen nodos en absoluto, pasamos a árboles cada vez más grandes. Denotemos con h la anchura $j-i$ del subárbol T_{ij} . Es fácil obtener los valores p_{ii} para todos los árboles con $h = 0$ conforme a (4.72).

FOR i := 0 TO n DO p[i,i] := b[i] END (4.75)

En el caso $h = 1$ nos ocupamos de árboles que tienen un solo nodo, el cual evidentemente también es la raíz (véase la figura 4.38).

FOR i := 0 TO n-1 DO
j := i+1; p[i,j] := w[i,j] + p[i,i] + p[j,j]; r[i,j] := j
END (4.76)

Nótese que i denota el límite del índice de la izquierda y j el del índice de la derecha en el árbol en cuestión T_{ij} . Para los casos $h > 1$ usamos una proposición repetitiva que oscila entre 2 y n; el caso $h = n$ cubre todo el árbol $T_{0,n}$. En todos los casos, la longitud mínima de trayectoria p_{ij} y el índice asociado de raíz r_{ij} se determinan mediante una simple proposición repetitiva con un índice k que se extiende sobre el intervalo dado por (4.73).

FOR h := 2 TO n DO
FOR i := 0 TO n-h DO
j := i+h;
encontrar k y min = MIN k : i < k ≤ j : (p_{i,k-1} + p_{kj})
tal que r_{i,j-1} ≤ k ≤ r_{i+1,j};
p[i,j] := min + w[i,j]; r[i,j] := k
END
END (4.77)

Los detalles del refinamiento de la proposición en cursivas se encuentran en el programa 4.6. La longitud promedio de trayectoria $T_{0,n}$ está dada ahora por el cociente $p_{0,n}/w_{0,n}$ y su raíz es el nodo que tiene el índice $r_{0,n}$.

Describamos en seguida la estructura del programa 4.6. Sus dos principales componentes son los procedimientos con que se encuentra el árbol óptimo de búsqueda, dada

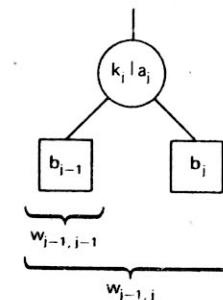


Fig. 4.38 Árbol óptimo con un nodo.

una distribución de peso w , y que sirve para mostrar el árbol conociendo los índices r . Primero, los conteos a y b y las llaves se leen de una fuente de entrada. Las llaves no intervienen realmente en el cálculo de la estructura del árbol; tan sólo se utilizan en la exhibición posterior del árbol. Luego de imprimir los datos estadísticos de la frecuencia, el programa empieza a calcular la longitud de la trayectoria del árbol perfectamente balanceado; y al hacerlo también determina las raíces de sus subárboles. Después se imprime la longitud promedio de la trayectoria pesada y se muestra el árbol.

En la tercera parte, el procedimiento *OptTree* es activado a fin de calcular el árbol óptimo de búsqueda; después, el árbol se exhibe. Y por último los mismos procedimientos se emplean para calcular y mostrar el árbol óptimo considerando tan sólo las frecuencias e ignorando las frecuencias de las no llaves.

La tabla 4.5 y las figuras 4.40 a 4.42 contienen los resultados generados por el programa 4.6 cuando se aplica a su propio texto. Las diferencias en las tres figuras indican que el árbol balanceado no puede ni siquiera considerarse cercano a lo óptimo y que las frecuencias de las no llaves influyen profundamente en la selección de la estructura óptima.

```

MODULE OptTree;
FROM InOut IMPORT
  OpenInput, OpenOutput, Read, ReadCard, ReadString, WriteCard,
  WriteString, Write, WriteLn, Done, CloseInput, CloseOutput;
FROM Storage IMPORT ALLOCATE;

CONST N = 100; (*num. max. de palabras reservadas*)
  WL = 16; (*longitud max. de palabras reservadas*)

TYPE Word = ARRAY [0 .. WL-1] OF CHAR;
  index = [0 .. N];

VAR ch: CHAR;
  i, j, n: CARDINAL;
  key: ARRAY index OF Word;
  a: ARRAY index OF CARDINAL;
  b: ARRAY index OF CARDINAL;
  p,w: ARRAY index, index OF CARDINAL;
  r: ARRAY index, index OF CARDINAL;

PROCEDURE BalTree(i,j: CARDINAL): CARDINAL;
  VAR k: CARDINAL;
BEGIN k := (i+j+1) DIV 2; r[i,j] := k;
  IF i >= j THEN RETURN 0
  ELSE RETURN BalTree(i,k-1) + BalTree(k,j) + w[i,j]
  END
END BalTree;

PROCEDURE OptTree;
  VAR x, min: CARDINAL;
  i, j, k, h, m: CARDINAL;

```

```

BEGIN (*argumento: W, resultados: p, r*)
  FOR i := 0 TO n DO p[i,i] := 0 END ;
  FOR i := 0 TO n-1 DO
    j := i+1; p[i,j] := w[i,j]; r[i,j] := j
  END ;
  FOR h := 2 TO n DO
    FOR i := 0 TO n-h DO
      j := i+h; m := r[i,j-1]; min := p[i,m-1] + p[m,j];
      FOR k := m+1 TO r[i+1,j] DO
        x := p[i,k-1] + p[k,j];
        IF x < min THEN
          m := k; min := x
        END
      END ;
      p[i,j] := min + w[i,j]; r[i,j] := m
    END
  END
END OptTree;

PROCEDURE PrintTree(i, j, level: CARDINAL);
  VAR k: CARDINAL;
BEGIN
  IF i < j THEN
    PrintTree(i, r[i,j]-1, level+1);
    FOR k := 1 TO level DO WriteString("  ") END ;
    WriteString(key[r[i,j]]); WriteLn;
    PrintTree(r[i,j], j, level+1)
  END
END PrintTree;

BEGIN (*programa principal*)
  n := 0; OpenInput("TEXT");
  LOOP ReadCard(b[n]);
    IF NOT Done THEN HALT END ;
    ReadCard(j);
    IF NOT Done THEN EXIT END ;
    n := n+1; a[n] := j;
    ReadString(key[n])
  END ;

  OpenOutput("TREE");
  (*calcular w a partir de a y b*)
  FOR i := 0 TO n DO
    w[i,i] := b[i];
    FOR j := i+1 TO n DO
      w[i,j] := w[i,j-1] + a[j] + b[j]
    END
  END

```

```

END
END ;
WriteString ("Peso total = "); WriteCard(w[0,n], 6); WriteLn;
WriteString ("Longitud de la trayectoria del arbol balanceado = ");
WriteCard(BalTree(0, n), 6); WriteLn;
PrintTree(0, n, 0); WriteLn;

Read(ch);
OptTree;
WriteString ("Longitud de la trayectoria del arbol optimo = ");
WriteCard(p[0,n], 6); WriteLn;
PrintTree(0, n, 0); WriteLn;

Read(ch);
FOR i := 0 TO n DO
  w[i,i] := 0;
  FOR j := i+1 TO n DO
    w[i,j] := w[i,j-1] + a[j];
  END
END ;
OptTree;
WriteString("arbol optimo sin considerar b "); WriteLn;
PrintTree(0, n, 0); WriteLn;
CloseInput; CloseOutput
END OptTree.

```

Programa 4.6 Obtención de un árbol de búsqueda óptimo.

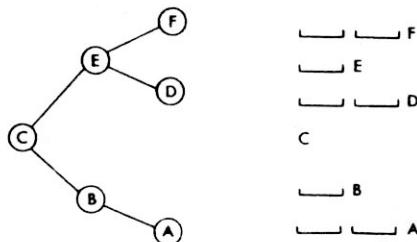


Fig. 4.39 Dibujo de la estructura de un árbol mediante una correcta sangría (escalonamiento) de línea.

b[i-1]	a[i]	k[i]
169	3	AND
25	37	ARRAY
355	125	BEGIN
87	1	BY
264	14	CASE
247	28	CODE
90	9	CONST
118	3	DEFINITION
10	16	DIV
0	55	DO
124	299	ELSE
4	198	ELSIF
10	689	END
281	25	EXIT
35	3	EXPORT
442	19	FROM
0	0	FOR
646	464	IF
5	3	IMPLEMENTATION
13	20	IMPORT
15	2	IN
654	24	LOOP
159	15	MOD
130	16	MODULE
166	79	NIL
16	10	NOT
218	34	OF
31	95	OR
276	11	POINTER
82	171	PROCEDURE
418	1	QUALIFIED
124	6	RECORD
49	9	REPEAT
30	2	RETURN
174	22	SET
505	662	THEN
9	6	TO
385	13	TYPE
37	9	UNTIL
347	203	VAR
84	35	WHILE
0	14	WITH
981		

Tabla 4.5 Llaves y frecuencias de ocurrencia.

1. Peso total = 1126
 Longitud de trayectoria
 del árbol balanceado = 60312

```

graph TD
    DO[DO] --> CASE[CASE]
    DO --> EXPORT[EXPORT]
    CASE --> ARRAY[ARRAY]
    CASE --> BEGIN-BEGIN[BEGIN BY]
    BEGIN-BEGIN --> BY[BY]
    CASE --> CONST[CONST]
    CASE --> CODE[CODE]
    CASE --> DEFINITION[DEFINITION]
    DEFINITION --> DIV[DIV]
    EXPORT --> ELSE[ELSE]
    EXPORT --> ELSIF[ELSIF]
    EXPORT --> END-END[END EXIT]
    END-END --> EXIT[EXIT]
    EXPORT --> IF[IF]
    EXPORT --> FROM-FROM[FROM FOR]
    IF --> FOR[FOR]
    IF --> IMPLEMENTATION[IMPLEMENTATION]
    IMPLEMENTATION --> IMPORT[IMPORT]
    
    N[N]
    RECORD[RECORD] --> NOT[NOT]
    RECORD --> MOD[MOD]
    MOD --> MODULE[MODULE]
    MODULE --> NIL[NIL]
    RECORD --> OF[OF]
    OF --> OR[OR]
    RECORD --> POINTER[POINTER]
    POINTER --> PROCEDURE[PROCEDURE]
    PROCEDURE --> QUALIFIED[QUALIFIED]
    
    RECORD --> TO[TO]
    TO --> REPEAT[REPEAT]
    REPEAT --> RETURN[RETURN]
    RETURN --> SET[SET]
    SET --> THEN[THEN]
    TO --> VAR[VAR]
    VAR --> WHILE[WHILE]
    WHILE --> UNTIL[UNTIL]
    VAR --> TYPE[TYPE]
    TYPE --> UNTIL
  
```

Fig. 4.40 Árbol perfectamente balanceado.

Longitud de la trayectoria del
árbol óptimo = 50371

```

graph TD
    IF[IF] --> BEGIN[BEGIN]
    IF --> END[END]
    IF --> FOR[FOR]
    BEGIN --> CASE[CASE]
    BEGIN --> DO[DO]
    BEGIN --> ELSE[ELSE]
    CASE --> BY[BY]
    DO --> CONST[CONST]
    CONST --> DIV[DIV]
    CONST --> DEFINITION[DEFINITION]
    ELSE --> ELSIF[ELSIF]
    END --> EXIT[EXIT]
    EXIT --> EXPORT[EXPORT]
    EXPORT --> FROM[FROM]
    FOR --> LOOP[LOOP]
    LOOP --> MOD[MOD]
    LOOP --> MODULE[MODULE]
    NIL[NIL] --> NOT[NOT]
    NIL --> OF[OF]
    NIL --> OR[OR]
    NIL --> POINTER[POINTER]
    PROCEDURE[PROCEDURE] --> SET[SET]
    PROCEDURE --> QUALIFIED[QUALIFIED]
    QUALIFIED --> RECORD[RECORD]
    RECORD --> REPEAT[REPEAT]
    SET --> RETURN[RETURN]
    THEN[THEN] --> TYPE[TYPE]
    TYPE --> VAR[VAR]
    TYPE --> TO[TO]
    VAR --> UNTIL[UNTIL]
    VAR --> WHILE[WHILE]
    WITH[WITH] --> TYPE[TYPE]
    TYPE --> VAR[VAR]
    TYPE --> TO[TO]
    
```

Fig. 4.41 Árbol óptimo de búsqueda.

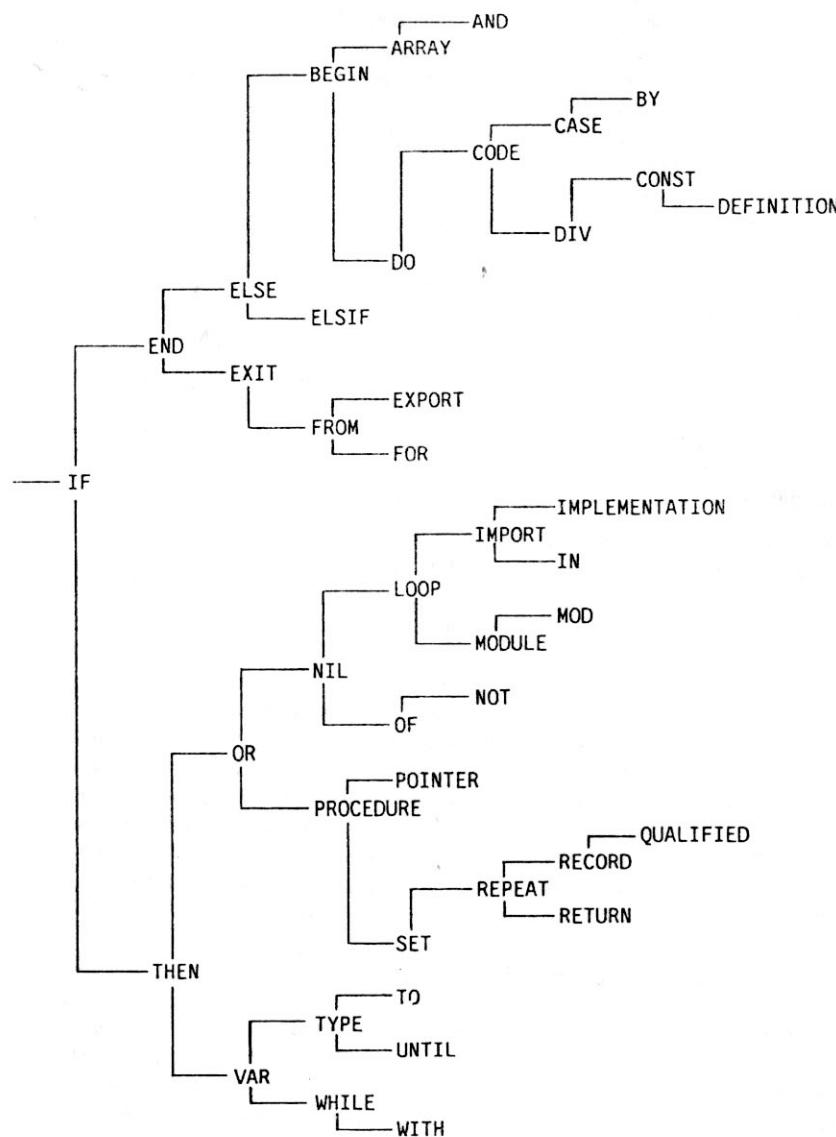


Fig. 4.42 Arbol óptimo que considera solamente llaves

En el algoritmo (4.77) es patente que el intento de determinar la estructura óptima es del orden de n^2 ; por otra parte, la cantidad necesaria de memoria es del orden de n^2 . Ello resulta inaceptable si n es muy grande. De ahí la conveniencia de contar con algoritmos de mayor eficiencia. Uno de ellos es el inventado por Hu y Tucker [4-5] que requiere apenas $O(n)$ elementos de almacenamiento y $O(n \cdot \log(n))$ cálculos. Sin embargo, considera tan sólo el caso en que las frecuencias de llaves son cero, o sea cuando sólo se registran los fracasos de búsqueda. Otro algoritmo, que también requiere $O(n)$ elementos de almacenamiento y $O(n \cdot \log(n))$ cálculos fue descrito por Walker y Gotlieb [4-7]. En vez de tratar de encontrar el óptimo, este algoritmo tan sólo promete generar un árbol casi óptimo. Por consiguiente, puede basarse en principios heurísticos. A continuación se describe la idea básica en que se funda.

Imaginemos que los nodos (genuinos y especiales) están distribuidos sobre una escala lineal, pesados por sus frecuencias (o probabilidades) de acceso. Después se encuentra el nodo que se halla más cerca del centro de gravedad. A ese nodo se le llama *centroide* y su índice es

$$((S_i : 1 \leq i \leq n : i \cdot a_i) + (S_j : 0 \leq j \leq m : j \cdot b_j)) / W \quad (4.78)$$

redondeado al entero más cercano. Si todos los nodos tienen peso igual, la raíz del árbol óptimo deseado coincidirá seguramente con el centroide y, así se razona, casi siempre estará en la vecindad del centroide. Así pues, se recurre a una búsqueda limitada para encontrar el óptimo local y luego el procedimiento se aplica a los dos subárboles resultantes. La probabilidad de que la raíz se halle muy cerca del centroide aumenta con el tamaño n del árbol. Tan pronto los subárboles han alcanzado un tamaño manejable, es posible determinar su óptimo mediante el algoritmo exacto que vimos antes.

4.7. ARBOLES B

Hasta ahora nos hemos limitado en nuestras exposiciones a árboles en los cuales los nodos tienen como máximo dos descendientes, esto es, a los árboles binarios. Ello es enteramente satisfactorio si, por ejemplo, queremos representar las relaciones de familia con una preferencia por un punto de vista genealógico, en el cual toda persona está asociada a sus padres. Después de todo, nadie tiene más de dos progenitores. Pero, ¿qué decir de alguien que prefiera adoptar un punto de vista de su posteridad? Habrá de encarar el hecho de que algunas personas tienen más de dos hijos y su árbol contendrá nodos con muchas ramas. A falta de una designación mejor, a ese tipo de árboles los llamaremos *multicamino* (direcciones múltiples).

Desde luego, tales estructuras no tienen nada de especial; ya hemos visto todos los recursos de programación y de definición de datos que nos permiten resolver esos problemas. Por ejemplo, si se fija un límite superior absoluto al número de hijos (sin duda, se trata de una suposición un poco futurista), podemos representar los hijos como un componente de arreglo del registro que representa a una persona. Si el número de hijos varía mucho entre personas diferentes, esto puede ser una utilización poco eficiente de la memoria disponible. En ese caso será mucho más apropiado arreglar a los hijos como una lista lineal, con un apuntador a los más jóvenes (o a los mayores) asignados al progenitor. Una definición posible de tipo para este caso (4.80) y una estructura también posible de datos aparecen en la figura 4.43.

```
TYPE Ptr =  POINTER TO Person;
TYPE Person = RECORD name: alfa;
               sibling, offspring: Ptr
             END
```

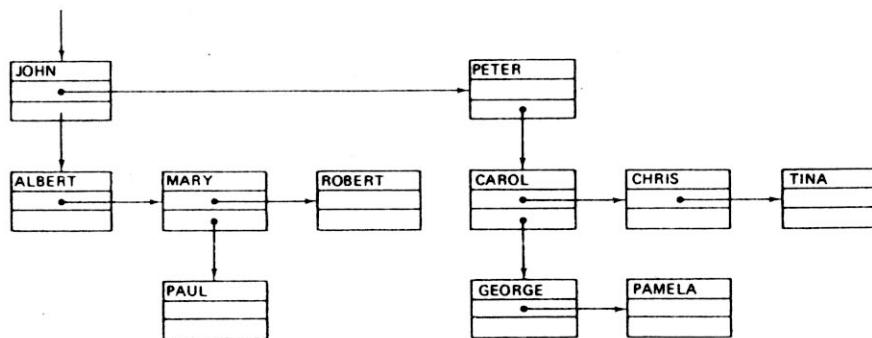
 (4.80)


Fig. 4.43 Árbol multicamino representado como un árbol binario.

Ahora nos damos cuenta que, al inclinar esta figura 45 grados, parecerá un árbol binario perfecto. Pero esta perspectiva es engañosa pues, desde el punto de vista funcional, las dos referencias tienen un significado totalmente distinto. Generalmente no podemos tratar como a un hijo al hermano y si lo hacemos se nos castiga; por tanto, tampoco debemos hacerlo al construir las definiciones de datos. Este ejemplo podría aplicarse fácilmente a una estructura de datos todavía más complicada, introduciendo más componentes en el registro de cada individuo; de ese modo estaríamos en condiciones de representar con mayor detalle las relaciones de familia. Un candidato probable que generalmente no puede derivarse de las referencias de hermanos e hijos es la relación entre cónyuges o incluso la relación inversa de padre y madre. Tal estructura rápidamente se convierte en un complejo banco de datos relaciones, y es posible mapear varios árboles en él. Los algoritmos que operan sobre tales estructuras están íntimamente ligados a sus definiciones de datos; no tiene sentido especificar reglas generales ni técnicas de aplicación amplia.

Sin embargo, hay un área muy práctica de aplicación de los árboles multicamino que tiene un interés general. Se trata de la construcción y mantenimiento de árboles de búsqueda a gran escala, en los cuales se necesitan las inserciones y eliminaciones, pero la memoria primaria de una computadora no es lo suficientemente grande o económica para utilizarse para el almacenamiento a largo plazo.

Así pues, supongamos que los nodos de un árbol deben ser guardados en un medio de memoria secundaria, digamos en un disco. Las estructuras dinámicas de datos introducidas en este capítulo son muy adecuadas para incorporar los medios de memoria secundaria. La principal innovación consiste en que los apuntadores están representados por direcciones de almacenamiento en disco. Usar un árbol binario para un conjunto de datos de, digamos, un millón de elementos requiere en promedio aproximadamente $\log 10^6$ pasos de búsqueda (o sea unos 20). Puesto que cada paso requiere un acceso al disco (con el tiempo inherente de latencia), una organización de memoria que emplee menos accesos será conveniente en extremo. El árbol multicamino es una solución perfecta de este problema. Si se accede a un elemento situado en una memoria secundaria, se puede acceder a un grupo entero de elementos sin mucho costo adicional. Ello significa que un árbol puede subdividirse en subárboles y que éstos se representan como unidades a las cuales se accede simultáneamente. Llamaremos *páginas* a esos subárboles. La figura 4.44 muestra un árbol binario subdividido en páginas, cada una formada de 7 nodos.

Puede ser considerable el ahorro en el número de accesos al disco: cada acceso de página requiere ahora un acceso al disco. Supongamos que decidimos colocar 100 nodos en una página (cifra por lo demás muy razonable); entonces el árbol de búsqueda de un millón de elementos requerirá, en promedio, apenas $\log_{100} 10^6$ (es decir, cerca de 3) accesos de página en vez de 20. Pero, por supuesto, si se deja al árbol crecer aleatoriamente, el peor caso será todavía hasta de 10^4 . Es evidente que un plan del crecimiento controlado resulta casi obligatorio en el caso de árboles multicamino.

4.7.1. Árboles B multicamino

Si buscamos un criterio de crecimiento controlado, de inmediato se eliminará el que exija un equilibrio perfecto pues supone "excesos". Hay que liberalizar un poco las reglas. En

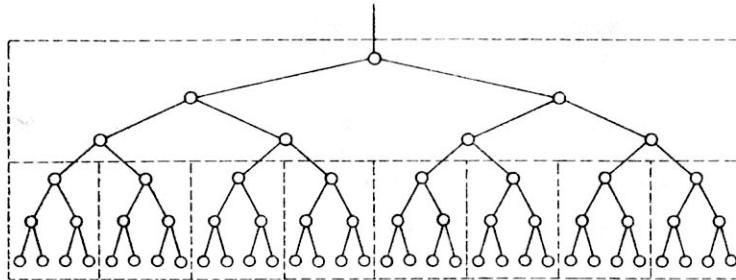


Fig. 4.44 Árbol binario subdividido en páginas.

1970 R. Bayer y E.M. McCreight [4.2] postularon un criterio muy razonable: cada página (salvo una) contiene entre n y $2n$ nodos para determinada constante n . De ahí que, en un árbol con N elementos y un tamaño máximo de página de $2n$ nodos por página, el peor caso requiere $\log_n N$ accesos de página; y los accesos dominan evidentemente la búsqueda entera. No obstante, el factor importante en la utilización de memoria es por lo menos 50% ya que las páginas siempre están llenas a la mitad, por lo menos. Con todas las ventajas señaladas antes, el plan incluye algoritmos relativamente simples de búsqueda, inserción y eliminación. Más adelante los estudiaremos a fondo.

Las estructuras de datos subyacentes reciben el nombre de *árboles B* y tienen las siguientes características; se dice que n es el *orden* del árbol B.

1. Cada página contiene a lo sumo $2n$ elementos (llaves).
2. Cada página, excepto la de la raíz, contiene n elementos por lo menos.
3. Cada página es una página de hoja, o sea que no tiene descendientes o tiene $m + 1$ descendientes, donde m es su número de llaves en esta página.
4. Todas las páginas de hoja aparecen al mismo nivel.

La figura 4.45 muestra un árbol B de orden 2 con 3 niveles. Todas las páginas contienen 2, 3 o 4 elementos; la excepción es la raíz que puede contener un solo elemento únicamente. Todas las páginas de hoja aparecen en el nivel 3. Las llaves aparecen en or-

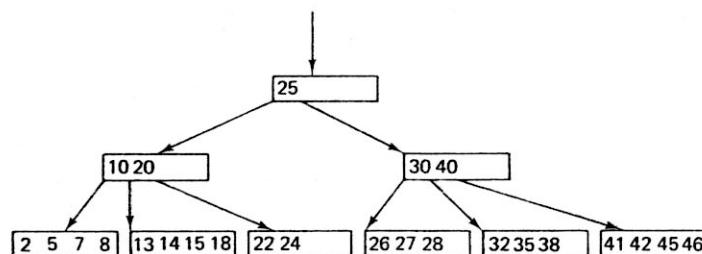


Fig. 4.45 Árbol B de orden 2.

den creciente de la izquierda a la derecha si el árbol B es introducido forzadamente en un solo nivel, insertando los descendientes entre las llaves de su página madre. Este arreglo representa una extensión natural de los árboles binarios y determina el método con que se busca un elemento que tiene una llave determinada. Examinemos una página como la de la figura 4.46 y un argumento de búsqueda x. Suponiendo que la página ha sido metida en la memoria primaria, podemos aplicar los métodos ordinarios de búsqueda entre las llaves $k_1 \dots k_m$. Si m es suficientemente grande, puede recurirse a la investigación binaria; si es bastante pequeña, bastará con una búsqueda secuencial ordinaria. (Nótese que el tiempo requerido para hacer una búsqueda en la memoria principal posiblemente sea despreciable en comparación con el que se tarda en meter en la memoria primaria la página procedente de la memoria secundaria.) Si la búsqueda fracasa, nos encontraremos en una de las siguientes situaciones:

1. $k_j < x < k_{i+1}$, para $1 \leq i < m$. Proseguimos la búsqueda en la página $p_i \uparrow$.
2. $k_m < x$. La búsqueda prosigue en la página $p_m \uparrow$.
3. $x < k_1$. La búsqueda prosigue en la página $p_0 \uparrow$.

Si en algún caso el apuntador designado es NIL, esto es, si no hay página de hijo, entonces tampoco existe un elemento con la llave x en todo el árbol y la búsqueda finaliza.

Es interesante señalar que la inserción en un árbol B es relativamente sencilla. Si hay que insertar un elemento en una página con $m < 2n$ elementos, el proceso de inserción queda limitado a esa página. Es sólo la inserción en una página ya llena la que tiene consecuencias en la estructura del árbol, pudiendo ocasionar la asignación de páginas nuevas. Para entender lo que sucede en ese caso, consultese la figura 4.47 que ilustra la inserción de la llave 22 en un árbol B de orden 2. La acción se realiza en los siguientes pasos:

1. Se descubre que falta la llave 22; la inserción en la página C es imposible porque C ya está llena.
2. La página C se divide en dos páginas (esto es, se asigna una nueva página D).
3. Las $2n + 1$ llaves se distribuyen uniformemente en C y D, y la llave de la mitad se sube un nivel hacia la página madre A.

Este plan tan elegante preserva todas las propiedades típicas de los árboles B. En particular, las páginas divididas contienen exactamente n elementos. Desde luego, la inserción de un elemento en la página madre puede hacer que ésta se desborde, con lo cual ocasiona que la división se propague. En el caso extremo, puede propagarse hasta la raíz. Es decir, la única manera en que el árbol B puede aumentar su altura. Tiene, pues, una manera singular de crecer: crece de las hojas hacia la raíz.

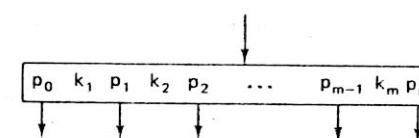


Fig. 4.46 Página de árbol B con m llaves.

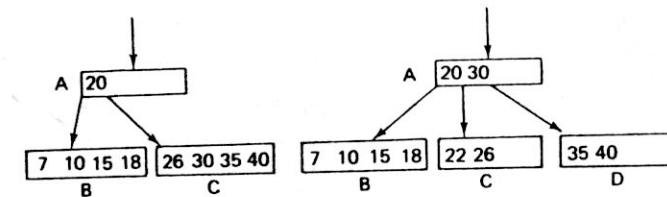


Fig. 4.47 Inserción de la llave 22 en un árbol B.

A continuación elaboraremos un programa detallado a partir de las descripciones sencillas que acabamos de hacer. Ya se habrá advertido que una formulación recursiva será la más idónea, debido a la propiedad que tiene el proceso de división de propagarse hacia atrás a lo largo de la trayectoria de búsqueda. La estructura general del programa será por eso semejante a la inserción de un árbol balanceado, aunque los detalles son diferentes. Ante todo, es preciso formular una definición de la estructura de página. Optamos por representar los elementos en forma de un arreglo.

```
TYPE PPtr = POINTER TO Page;
TYPE index =[0 .. 2*n];
TYPE item = RECORD key: INTEGER;
  p: PPtr;
  count: CARDINAL
END ;
(4.81)
```

```
TYPE page = RECORD m: index;
  p0: PPtr;
  e: ARRAY [1 .. 2*n] OF item
END
```

También en este caso, el *conteo* de componentes de elementos representa toda clase de información adicional que puede estar asociada a cada elemento, pero que no interviene en el proceso de búsqueda. Nótese que cada página ofrece espacio para $2n$ elementos. El campo *m* indica el número real de elementos en la página. Puesto que $m > n$ (salvo por la página de la raíz), una utilización de almacenamiento por lo menos de 50% queda garantizada.

El algoritmo de la búsqueda e inserción en árboles binarios forma parte del programa 4.7, formulado como un procedimiento denominado *search*. Su estructura principal es sencilla y se parece a la búsqueda en un árbol binario balanceado, salvo que la decisión de ramificación no es una opción binaria. Por el contrario, la *búsqueda dentro de página* se representa como una búsqueda binaria en un arreglo de elementos.

El algoritmo de inserción se formula como un procedimiento independiente sólo para mejorar la claridad. Se le activa después de que la búsqueda ha indicado que un elemento debe pasarse al árbol (en dirección de la raíz). Este hecho está indicado por el parámetro booleano del resultado *h*; supone un papel similar al del algoritmo para la inserción en

árboles balanceados, donde *h* indica que ha crecido el subárbol. Si *h* es verdadero, el segundo parámetro del resultado, *u*, representa al elemento que se está desplazando hacia arriba en el árbol.

Nótese que las inserciones empiezan en páginas hipotéticas, a saber, los "nodos especiales" de la figura 4.19; el nuevo elemento se entrega inmediatamente, mediante el parámetro *u*, a la página de la hoja para su inserción. El esquema se describe en (4.83).

PROCEDURE search(x: INTEGER; a: PPtr; VAR h: BOOLEAN; VAR u: item);

```
BEGIN
  IF a = NIL THEN (*x no está en árbol insertar*)
    Asignar x al elemento u, poner h en TRUE, lo cual
    indica que un elemento u se pasa al árbol
  ELSE
    WITH a DO
      búsqueda binaria de x en arreglo e;
      IF se encuentra THEN procesar datos
      ELSE search(x, descendiente u);
      IF h THEN (*un elemento se sube*)
        IF núm. de elemento en página at < 2n THEN
          insertar u en la página y subir at y poner h en FALSE
        ELSE dividir página y subir el elemento de la mitad
        END
      END
    END
  END
END search
```

Si el parámetro *h* es verdadero después de la llamada de *search* en el programa principal, se pide una división de la página de la raíz. Como esta página desempeña un papel excepcional, el proceso debe programarse por separado. Consiste meramente en asignar una nueva página de la raíz y meter el elemento dado por el parámetro *u*. En consecuencia, la nueva página contendrá un solo elemento. Los detalles se deducen del programa 4.7 y la figura 4.48 muestra el resultado de utilizar el programa 4.7 para construir un árbol B con la siguiente secuencia de inserción de llaves:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

Los puntos y coma designan las posiciones de las instantáneas tomadas después de cada asignación de página. La inserción de la última llave produce dos divisiones y la asignación de otras tres páginas.

La cláusula *with* en este programa posee una importancia especial. En primer lugar, indica que los identificadores de los componentes de página se refieren automáticamente a la página *a* dentro de la proposición antecedida por la cláusula. En efecto, si las páginas fueran asignadas en memoria secundaria (como sería necesario en un gran sistema de

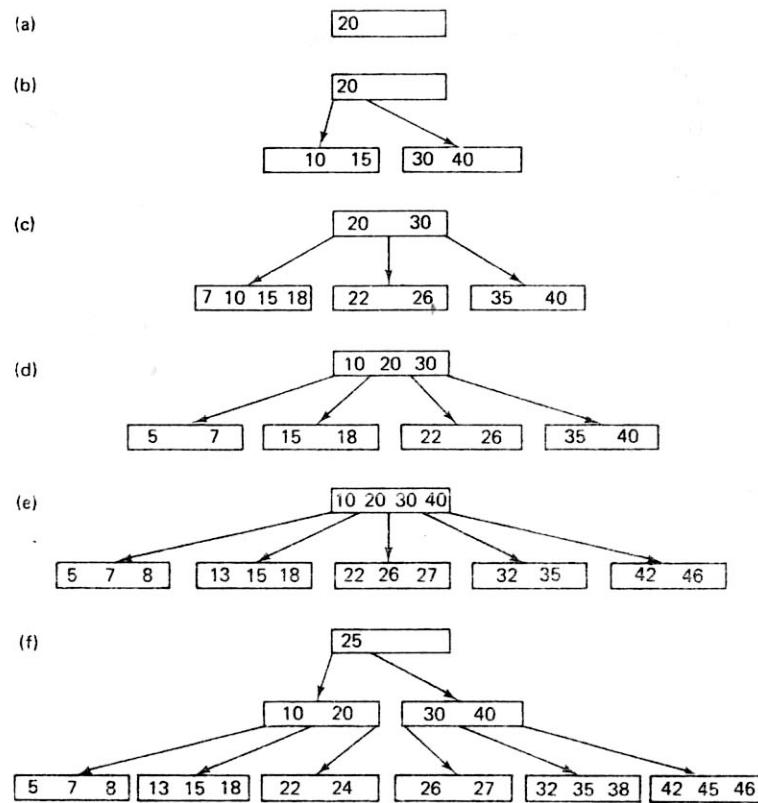


Fig. 4.48 Crecimiento de un árbol B de orden 2.

base de datos), la cláusula *with* pudiera interpretarse además como si indicase la transferencia de la página designada a la memoria primaria. Como cada activación de *search* implica por ello la transferencia de una página a la memoria principal, se requieren a lo sumo $k = \log_n N$ llamadas recursivas, si el árbol contiene N elementos. Por tanto, debemos ser capaces de introducir k páginas en la memoria principal. Ello constituye un factor limitante para el tamaño de página $2n$. En efecto, necesitamos alojar aun más de k páginas, pues la inserción puede ocasionar la división de páginas. Un corolario de lo anterior consiste en que la página de la raíz se asigna mejor permanentemente en la memoria primaria, porque cada búsqueda recorre necesariamente esa página.

Otra cualidad positiva de la organización del árbol B es su conveniencia y economía en el caso de una *actualización puramente secuencial* de la base de datos completa. Cada página se mete en la memoria primaria exactamente una vez.

La eliminación de elementos en un árbol B es en teoría bastante sencilla, pero se complica en sus detalles. Podemos distinguir dos circunstancias:

1. El elemento que debe suprimirse se halla en una página de hoja; entonces su algoritmo de eliminación será fácil y sencillo.
2. El elemento no se encuentra en la página de hoja; hay que sustituirlo por uno de dos elementos lexicográficamente contiguos, que resultan estar en las páginas de hoja y son fáciles de suprimir.

En el caso 2 encontrar la llave contigua es semejante a encontrar la que se usó en la eliminación en árboles binarios. Bajamos por los apuntadores situados al extremo derecho hasta la página de hoja P, reemplazamos el elemento por eliminar con el del extremo derecho en P y luego reducimos en 1 el tamaño de P. En todo caso, la disminución del tamaño debe acompañarse de una verificación del número de elementos en la página reducida, pues se violaría la característica primaria de los árboles B si $m < n$. Hay que tomar medidas complementarias; esta condición de *Subocupación* está indicada por el parámetro de variable booleana *h*.

El único recurso consiste en tomar o anexar un elemento a partir de una de las páginas vecinas, digamos Q. Puesto que para ello se requiere introducir Q en la memoria principal (operación relativamente costosa), sentimos la tentación de salir airoso de esa situación inconveniente y agregar más de un elemento de inmediato. La estrategia habitual consiste en distribuir los elementos en las páginas P y Q de manera uniforme en ambas. A esto se le llama *balanceo de páginas*.

Por supuesto, puede suceder que no queden elementos por agregar, pues Q ya ha alcanzado su tamaño mínimo n . En este caso, el número total de elementos en las páginas P y Q es $2n-1$; podemos *combinar* las dos páginas en una, agregando el elemento intermedio de la página madre de P y Q y luego prescindiendo completamente de la página Q. Este es el proceso inverso de la división de página. El proceso puede visualizarse si se examina la eliminación de la llave 22 en la figura 4.47. También en este caso, la supresión de la llave de la mitad en la página madre puede hacer que su tamaño baje más allá del límite permisible n , con lo cual requeriría que en el siguiente nivel se tomase una medida especial (balanceo o mezcla). En el caso extremo, la combinación de páginas puede probarse hasta llegar a la raíz. Si el tamaño de ésta se reduce a 0, también se suprime ocasionalmente una disminución en la altura del árbol B. De hecho, ésa es la única manera en que un árbol B puede disminuir de tamaño. La figura 4.49 muestra el decaimiento gradual del árbol B de la figura 4.48 después de la eliminación secuencial de las llaves.

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

Los puntos y coma marcan el sitio donde se toman las instantáneas, a saber el sitio donde están eliminándose las páginas. El algoritmo de eliminación se incluye como procedimiento en el programa 4.7. Conviene destacar sobre todo la semejanza de su estructura con la de la eliminación en un árbol balanceado.

Se ha emprendido un análisis exhaustivo del rendimiento de los árboles B y los resultados se mencionan en un artículo que aparece en la bibliografía al final de este capítulo

```

MODULE BTree;
FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
ReadInt, Done, Write, WriteInt, WriteString, WriteLn;
FROM Storage IMPORT ALLOCATE;

CONST n = 2;

TYPE PPtr = POINTER TO Page;

Item = RECORD key: INTEGER;
p: PPtr;
count: CARDINAL
END ;

```

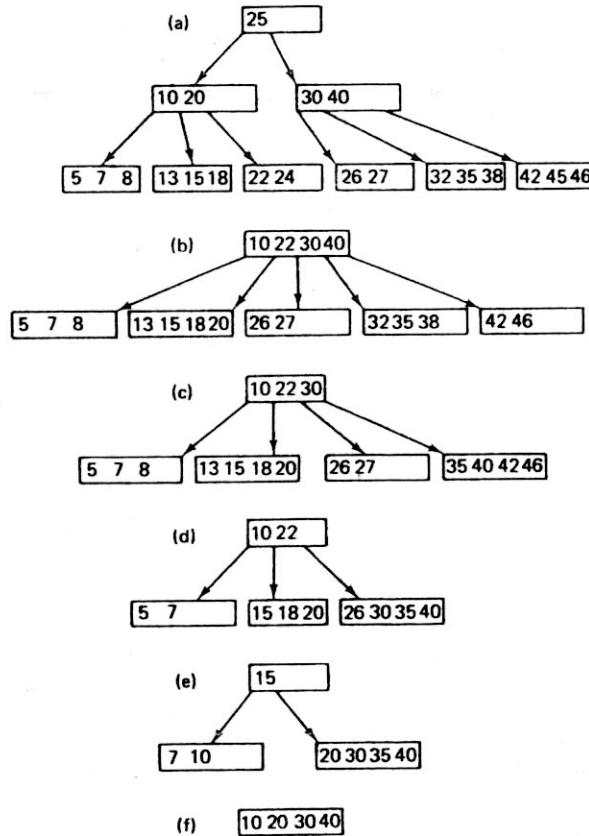


Fig. 4.49 Disminución de un árbol B de orden 2.

```

Page = RECORD m: [0 .. 2*n]; (*num. de elemento en pagina*)
p0: PPtr;
e: ARRAY [1 .. 2*n] OF Item
END ;

VAR root, q: PPtr;
x: INTEGER;
h: BOOLEAN;
u: Item;

PROCEDURE search(x: INTEGER; a: PPtr; VAR h: BOOLEAN; VAR v: Item);
(*buscar llave x en arbol B con raiz a; si se encuentra, aumentar contador.
De lo contrario insertar nuevo elemento con llave x. Si un elemento
se sube, asignarlo a v.h = "el arbol ha aumentado de altura"*)
VAR i, L, R: CARDINAL; b: PPtr; u: Item;
BEGIN (*~h*)
IF a = NIL THEN h := TRUE; (*no esta en el arbol*)
WITH v DO
key := x; count := 1; p := NIL
END
ELSE
WITH a^ DO
L := 1; R := m+1; (*busqueda binaria*)

WHILE L < R DO
i := (L+R) DIV 2;
IF e[i].key <= x THEN L := i+1 ELSE R := i END
END;
R := R-1;
IF (R > 0) & (e[R].key = x) THEN INC(e[R].count)
ELSE (*el elemento no esta en esta pagina*)
IF R = 0 THEN search(x, p0, h, u)
ELSE search(x, e[R].p, h, u)
END;
IF h THEN (*insertar u a la derecha de e[R]*)
IF m < 2*n THEN
h := FALSE; m := m+1;
FOR i := m TO R+2 BY -1 DO e[i] := e[i-1] END;
e[R+1] := u
ELSE ALLOCATE(b, SIZE(Page)); (*desbordamiento*)
(*dividir a en a,b y asignar el elemento de la mitad a v*)
IF R <= n THEN
IF R = n THEN v := u
ELSE v := e[n];
FOR i := n TO R+2 BY -1 DO e[i] := e[i-1] END;
e[R+1] := u
END;

```

```

FOR i := 1 TO n DO bt.e[i] := at.e[i+n] END
ELSE (*insertar en la pagina de la derecha*)
  R := R-n; v := e[n+1];
  FOR i := 1 TO R-1 DO bt.e[i] := at.e[i+n+1] END ;
  bt.e[R] := u;
  FOR i := R+1 TO n DO bt.e[i] := at.e[i+n] END
END ;
m := n; bt.m := n; bt.p0 := v.p; v.p := b
END
END
END
END search;

PROCEDURE underflow(c, a: PPtr; s: CARDINAL; VAR h: BOOLEAN);
(*a = pagina con insuficiencia, c = pagina ancestro,
 s = indice del elemento eliminado en c,h :=*)
VAR b: PPtr; VAR i, k, mb, mc: CARDINAL;
BEGIN mc := ct.m; (*h, at.m = n-1*)
IF s < mc THEN
  (*b := pagina a la derecha de a*) s := s+1;
  b := ct.e[s].p; mb := bt.m; k := (mb-n+1) DIV 2;
  (*k = num. de elementos disponibles en pagina b*)
  at.e[n] := ct.e[s]; at.e[n].p := bt.p0;
  IF k > 0 THEN
    (* mover k elementos de b a a*)
    FOR i := 1 TO k-1 DO at.e[i+n] := bt.e[i] END ;
    ct.e[s] := bt.e[k]; ct.e[s].p := b;
    bt.p0 := bt.e[k].p; mb := mb - k;
    FOR i := 1 TO mb DO bt.e[i] := bt.e[i+k] END ;
    bt.m := mb; at.m := n-1+k; h := FALSE
  ELSE (*mezclar paginas a y b *)
    FOR i := 1 TO n DO at.e[i+n] := bt.e[i] END ;
    FOR i := s TO mc-1 DO ct.e[i] := ct.e[i+1] END ;
    at.m := 2*n; ct.m := mc-1; h := mc <= n;
    (* desasignar (b)*)
  END
ELSE (*b := pagina a la izquierda de a*)
  IF s = 1 THEN b := ct.p0 ELSE b := ct.e[s-1].p END ;
  mb := bt.m + 1; k := (mb-n) DIV 2;
  IF k > 0 THEN
    (*mover k elementos de la pagina b a a*)
    FOR i := n-1 TO 1 BY -1 DO at.e[i+k] := at.e[i] END ;
    at.e[k] := ct.e[s]; at.e[k].p := at.p0; mb := mb-k;
  END
END

```

```

FOR i := k-1 TO 1 BY -1 DO at.e[i] := bt.e[i+mb] END ;
at.p0 := bt.e[mb].p; ct.e[s] := bt.e[mb]; ct.e[s].p := a;
bt.m := mb-1; at.m := n-1+k; h := FALSE
ELSE (*mezclar paginas a y b*)
  bt.e[mb] := ct.e[s]; bt.e[mb].p := at.p0;
  FOR i := 1 TO n-1 DO bt.e[i+mb] := at.e[i] END ;
  bt.m := 2*n; ct.m := mc-1; h := mc <= n;
  (* desasignar (a)*)
END
END
END underflow;
PROCEDURE delete(x: INTEGER; a: PPtr; VAR h: BOOLEAN);
(*buscar y designar llave x en el arbol B con raiz a; si aumenta
 la insuficiencia de una pagina, equilibrar la contigua o mezclarla;
 h := "pagina a no tiene el tamaño adecuado"*)
VAR i, L, R: CARDINAL; q: PPtr;
PROCEDURE del(P: PPtr; VAR h: BOOLEAN);
  VAR q: PPtr; (*global a, R*)
BEGIN
  WITH Pt DO
    q := e[m].p;
    IF q # NIL THEN del(q,h);
    IF h THEN underflow(P, q, m, h) END
  ELSE
    Pt.e[m].p := at.e[R].p; at.e[R] := Pt.e[m];
    m := m - 1; h := m < n
  END
  END
END del;
BEGIN
IF a = NIL THEN (*x no esta en el arbol*) h := FALSE
ELSE
  WITH at DO
    L := i; R := m+1; (*busqueda binaria*)
    WHILE L < R DO
      i := (L+R) DIV 2;
      IF e[i].key < x THEN L := i+1 ELSE R := i END
    END;
    IF R = 1 THEN q := p0 ELSE q := e[R-1].p END ;
    IF (R <= m) & (e[R].key = x) THEN
      (*found, now delete*)
      IF q = NIL THEN (*a es una pagina terminal*)
        m := m-1; h := m < n;
        FOR i := R TO m DO e[i] := e[i+1] END
      ELSE
        del(q,h);
        IF h THEN underflow(at, q, m, h) END
      END
    END
  END
END

```

```

ELSE del(q,h);
  IF h THEN underflow(a, q, R-1, h) END
END
ELSE delete(x, q, h);
  IF h THEN underflow(a, q, R-1, h) END
END
END
END
END delete;

PROCEDURE PrintTree(p: PPtr; level: CARDINAL);
  VAR i: CARDINAL;
BEGIN
  IF p # NIL THEN
    FOR i := 1 TO level DO WriteString("      ") END ;
    FOR i := 1 TO p^.m DO WriteInt(p^.e[i].key, 4) END ;
    WriteLn;
    PrintTree(p^.p0, level+1);
    FOR i := 1 TO p^.m DO PrintTree(p^.e[i].p, level+1) END
  END
END PrintTree;

BEGIN (*programa principal*)
  OpenInput("TEXT"); OpenOutput("TREE");
  root := NIL; Write(">"); ReadInt(x);

  WHILE Done DO
    WriteInt(x, 5); WriteLn;
    IF x >= 0 THEN
      search(x, root, h, u);
      IF h THEN (*insertar nueva pagina base*)
        q := root; ALLOCATE(root, SIZE(Page));
        WITH root^ DO
          m := 1; p0 := q; e[1] := u
        END
      ELSE
        delete(-x, root, h);
        IF h THEN (*tamaño de la pagina base reducido*)
          IF root^.m = 0 THEN
            q := root; root := q^.p0; (*desasignar (q)*)
          END
        END
      END;
      PrintTree(root, 0); WriteLn;
      Write(">"); ReadInt(x)
    END ;
  END;

```

CloseInput; CloseOutput
END BTree.

Programa 4.7 Búsqueda, inserción y eliminación en un árbol B.

(Bayer y McCreigh). En particular, incluye un tratamiento de la cuestión del tamaño óptimo de la página, el cual depende fundamentalmente de las características del almacenamiento y el sistema de cómputo.

Las variaciones del esquema del árbol B se explican en Knuth, vol. 3, pp. 476-479. La observación más relevante es que la división de página debe posponerse de la misma manera que su combinación, tratando antes de balancear las páginas vecinas. Además, los mejoramientos recomendados parecen producir ganancias pequeñas. Un estudio muy completo de los árboles B se encuentra en [4-8].

4.7.2. Árboles B binarios

La especie de árboles B que parece menos interesante es la del árbol binario de primer orden ($n = 1$). Pero en ocasiones vale la pena prestar atención al caso excepcional. Sin embargo, es evidente que los árboles B de primer orden no son útiles para representar conjuntos de datos amplios, ordenados, indexados que requieran memorias secundarias; aproximadamente 50% de las páginas contendrán un elemento únicamente. Por tanto, prescindiremos de las memorias secundarias y abordaremos otra vez el problema de los árboles de búsqueda que incluyen sólo un nivel.

El *árbol B binario* (árbol BB) consta de nodos (páginas) con uno o dos elementos. Por consiguiente, una página contiene dos o tres apuntadores a los descendientes; esto dio origen a la designación *árbol 2-3*. Según la definición de árbol B, todas las páginas de hoja aparecen en el mismo nivel y todas las páginas de no hoja de los árboles BB tienen dos o tres descendientes (incluida la raíz). Puesto que ahora nos ocupamos exclusivamente de la memoria primaria, un ahorro óptimo de espacio de almacenamiento es obligatorio, y la representación de los elementos dentro de un nodo mediante un arreglo no parece adecuada. Una alternativa es la asignación dinámica y ligada; es decir, en el interior de cada nodo existe una lista ligada de elementos de longitud 1 o 2. Puesto que cada nodo tiene a lo máximo tres descendientes y por lo mismo necesita alojar sólo tres apuntadores, estamos tentados a combinar los apuntadores de los descendientes con los apuntadores en la lista de elementos, según se advierte en la figura 4.50. Por tanto, el nodo del árbol B pierde su identidad verdadera y los elementos asumen el papel de nodos en un árbol binario regular. Sin embargo, hay que distinguir todavía entre los apuntadores a los descendientes (verticales) y los apuntadores a los hermanos en la misma página (horizontales). Sólo los apuntadores a la derecha pueden ser horizontales, por lo cual un solo bit es suficiente para anotar esta distinción. De ahí que introduzcamos el campo booleano *h* con el significado *horizontal*. La definición de un nodo de árbol, basada en esta representación, se da en (4.84). La definición fue propuesta e investigada por R. Bayer [4-3] en 1971 y representa una organización de árbol de búsqueda que garantiza una longitud máxima de trayectoria: $p = 2 * \lceil \log N \rceil$.

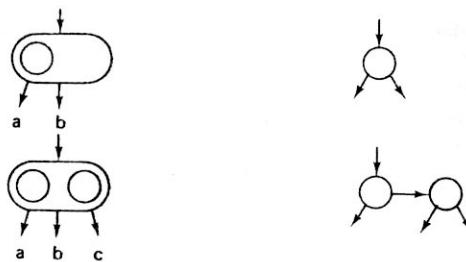


Fig. 4.50 Representación de nodos de árbol B binarios (BB).

```

TYPE Ptr =  POINTER TO Node;
TYPE Node = RECORD key: INTEGER;
.....  

left, right: Ptr;  

h: BOOLEAN (*rama horizontal de la derecha*)
END

```

(4.84)

Al considerar el problema de la inserción de llaves, es preciso distinguir cuatro situaciones posibles que se presentan por el crecimiento de los subárboles de la derecha o izquierda. Los cuatro casos están ejemplificados en la figura 4.51. Recuérdese que los árboles B tienen la característica de crecer del fondo hacia la raíz y que debe conservarse la propiedad de que todas las hojas se hallan en el mismo nivel. El caso más sencillo (1) se presenta cuando crece el subárbol de la derecha de un nodo A y cuando A es la única llave en su página (hipotética). Así pues, el descendiente B sólo se convierte en el hermano de A, esto es, el apuntador vertical se transforma en un apuntador horizontal. Esta simple elevación del brazo derecho no es posible si A ya tiene un hermano. Obtendríamos entonces una página con 3 nodos y habría que dividirla (caso 2). Su nodo dé en medio B se pasa al siguiente nivel superior.

Supongamos ahora que el subárbol de la izquierda de un nodo B ha aumentado de altura. Si B se encuentra otra vez sólo en una página (caso 3), esto es, su apuntador derecho denota un descendiente, el subárbol de la izquierda (A) se deja que se convierta en hermano de B. (Se requiere una simple rotación de apuntadores ya que el apuntador de la izquierda no puede ser horizontal.) Pero si B ya tiene un hermano, la elevación de A produce una página con tres miembros que requieren una división. Esta se realiza en una manera muy sencilla: C se convierte en descendiente de B, que es llevado al siguiente nivel superior (caso 4).

Conviene señalar que, terminada la búsqueda de una llave, es muy importante el hecho de que recorramos un apuntador horizontal o vertical. Parece, pues, artificial preocuparse de que un apuntador de la izquierda en el caso 3 se convierta en horizontal, pese a que su página todavía contiene más de dos miembros. En efecto, el algoritmo de inserción revela una extraña simetría en la administración del crecimiento de los subárboles de la derecha e izquierda, permitiendo que las organizaciones de los árboles B bina-

rios parezca más bien artificial. No hay prueba de la peculiaridad de esta organización; pero una sana intuición nos dice que algo anda mal y que debemos eliminar esta asimetría. Ello nos lleva a la noción de *árbol B binario simétrico* (árbol BBS), que también fue investigada por Bayer [4-4] en 1972. En general conduce a árboles de búsqueda ligeramente más eficientes, pero los algoritmos de inserción y eliminación también son un poco más complejos. Más aún, ahora cada nodo requiere dos bits (variables booleanas lh y rh) para indicar la naturaleza de sus dos apuntadores.

Como limitaremos nuestro examen de los detalles al problema de la inserción, una vez más tenemos que distinguir entre cuatro casos de subárboles crecidos. Se ejemplifican en la figura 4.52, que hace evidente la simetría adquirida. Obsérvese que, cuando crece un subárbol de nodo A sin hermanos, la raíz de él se convierte en hermano de A. No es necesario estudiar más detenidamente este caso.

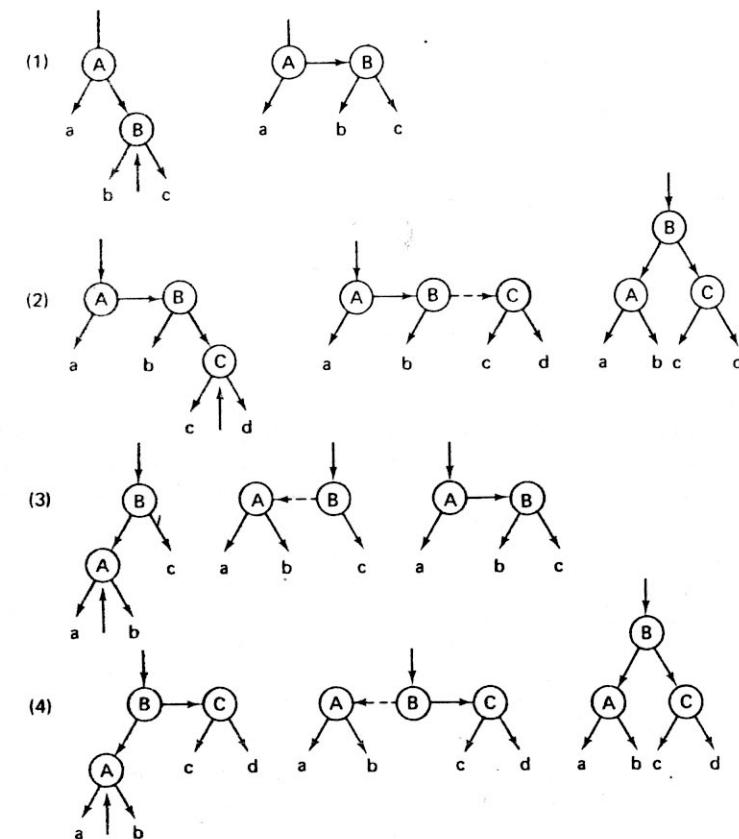


Fig. 4.51 Inserción de nodos en un árbol B binario (BB).

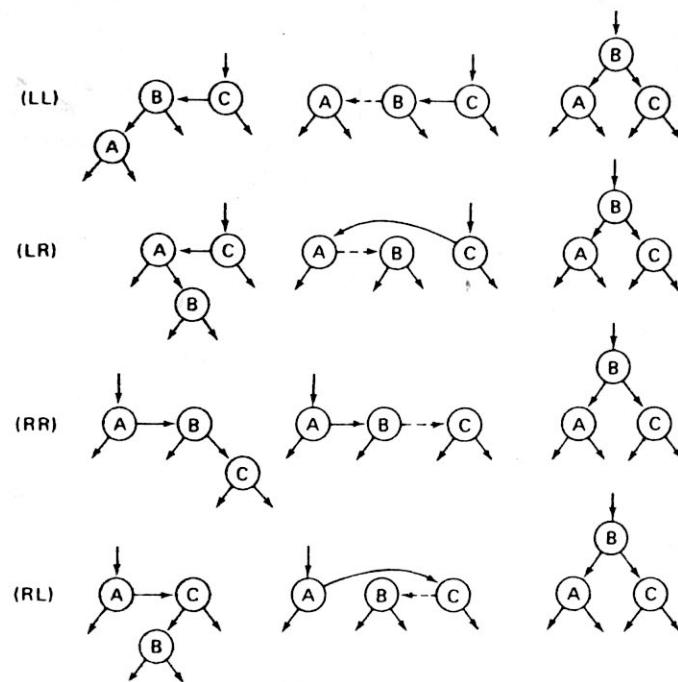


Fig. 4.52 Inserción en árboles B binarios simétricos (SBB).

Los cuatro casos considerados en la figura 4.52 reflejan la ocurrencia de un desbordamiento de página y la consecuente división de página. Se designan conforme a la dirección de los apuntadores horizontales que ligan los tres hermanos en las figuras de la mitad. La situación inicial se muestra en la columna de la izquierda; la columna de la mitad muestra el hecho de que el nodo inferior ha sido elevado conforme crece su subárbol; las cifras de la columna derecha muestran el resultado del rearreglo de los nodos.

Es conveniente olvidar el concepto de páginas del cual había surgido esta organización, pues tan sólo nos interesa unir con $2 \cdot \log N$ la longitud máxima de trayectoria. Para lograrlo sólo necesitamos asegurarnos de que los dos apuntadores horizontales quizás nunca ocurran en sucesión dentro de ninguna trayectoria de búsqueda. Sin embargo, no hay razón para prohibir nodos con apuntadores horizontales a la derecha e izquierda. Por consiguiente, definimos el árbol BBS como aquel que posee las siguientes propiedades.

1. Todo nodo contiene una llave y, al máximo, dos apuntadores a subárboles.
2. Todo apuntador es horizontal o vertical. No existen dos apuntadores consecutivos horizontales en ninguna trayectoria de búsqueda.
3. Todos los nodos terminales (nodos sin descendientes) aparecen en el mismo nivel (terminal).

De esta definición se deduce que la trayectoria más larga de búsqueda no es más larga que el doble de la altura del árbol. Puesto que ningún árbol BBS con N nodos puede tener una altura mayor que $\log N$, se sigue de inmediato que $2 \cdot \log N$ es un límite superior en la longitud de trayectoria de búsqueda. A fin de visualizar cómo crecen esos subárboles, recomendamos al lector consultar la figura 4.53. Las líneas representan instantáneas tomadas durante la inserción de las siguientes secuencias de llaves, donde cada punto y coma indica una instantánea.

$$\begin{array}{l}
 (1) \quad 1 \quad 2; \quad 3; \quad 4 \quad 5 \quad 6; \quad 7; \\
 (2) \quad 5 \quad 4; \quad 3; \quad 1 \quad 2 \quad 7 \quad 6; \\
 (3) \quad 6 \quad 2; \quad 4; \quad 1 \quad 7 \quad 3 \quad 5; \\
 (4) \quad 4 \quad 2 \quad 6; \quad 1 \quad 7; \quad 3 \quad 5;
 \end{array} \tag{4.85}$$

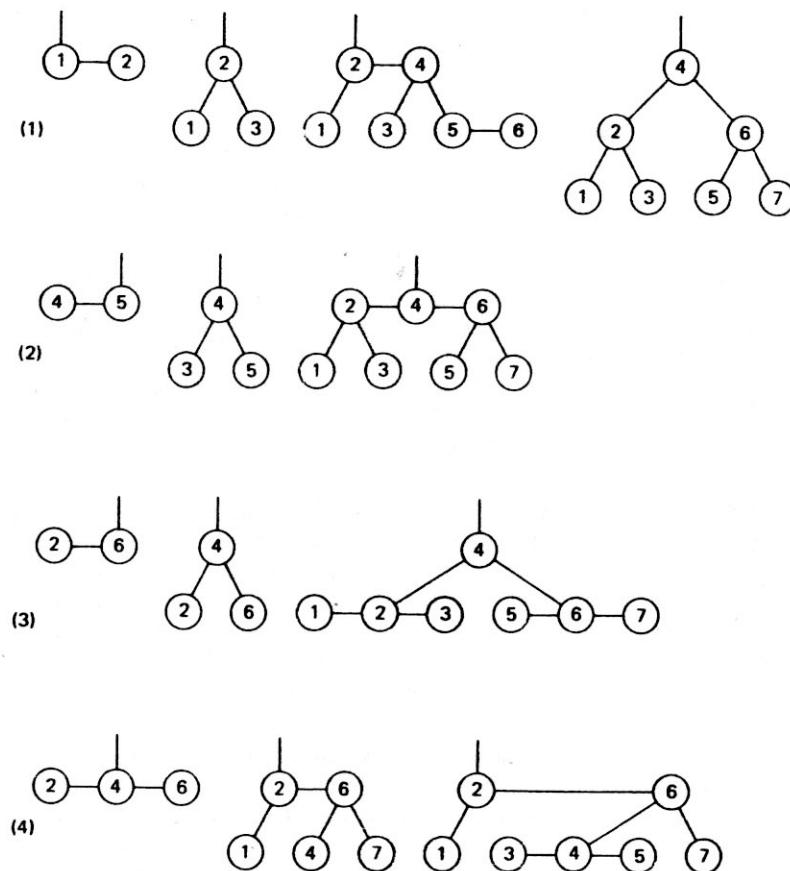


Fig. 4.53 Crecimiento de árboles B binarios simétricos (SBB) con secuencias de inserción (4.85).

Estas descripciones hacen muy obvia la tercera propiedad de los árboles B: todos los nodos terminales aparecen en el mismo nivel. Por tanto, estamos inclinados a comparar estas estructuras con los bordes de un jardín recién cortado con podadora.

El algoritmo para construir los árboles BBS se presenta en (4.87). Se basa en la definición del tipo *Node* (4.86); los dos componentes *lh* y *rh* indican si los apuntadores de la izquierda y derecha son horizontales.

```
TYPE Node = RECORD key: integer;
    count: CARDINAL;
    left, right: Ptr;
    lh, rh: BOOLEAN
END
```

(4.86)

El procedimiento recursivo *search* sigue otra vez el patrón del algoritmo básico de inserción del árbol binario (véase 4.87). Se agrega un tercer parámetro *h*; indica si el subárbol con raíz *p* ha cambiado y corresponde directamente al parámetro *h* del programa de búsqueda en el árbol B. Por tanto, señalamos la consecuencia de representar las páginas como listas ligadas; una página es recorrida por una o dos llamadas del procedimiento de búsqueda. Debemos distinguir entre el caso de un subárbol (indicado por un apuntador vertical) que ha crecido y un nodo hermano (indicado por un apuntador horizontal) que ha obtenido otro hermano y, por lo mismo, requiere una división de página. El problema se resuelve fácil al introducir una *h* valuada por árbol con los siguientes significados:

1. *h* = 0: el subárbol *p* no requiere cambios en la estructura de árbol.
2. *h* = 1: el nodo *p* ha obtenido un hermano.
3. *h* = 2: el subárbol *p* ha aumentado de altura.

```
PROCEDURE search(x: INTEGER; VAR p: Ptr; VAR h: CARDINAL);
    VAR p1, p2: Ptr; (*h = 0*)
BEGIN
    IF p = NIL THEN (*insertar*)
        ALLOCATE(p, SIZE(Node)); h := 2;
        WITH p DO
            key := x; count := 1;
            left := NIL; right := NIL; lh := FALSE; rh := FALSE
    END
    ELSIF p^.key > x THEN
        search(x, p^.left, h);
        IF h > 0 THEN (*la rama de la izquierda ha crecido*)
            IF p^.lh THEN
                p1 := p^.left; h := 2; p^.lh := FALSE;
                IF p1^.lh THEN (*LL*)
                    p^.left := p1^.right; p1^.right := p; p := p1;
                    p^.lh := FALSE
                ELSIF p1^.rh THEN (*LR*)
                    p2 := p1^.right; p1^.right := p2^.left; p2^.left := p1;

```

(4.87)

```
    p^.left := p2^.right; p2^.right := p; p := p2;
    p1^.rh := FALSE
END
ELSE h := h-1;
IF h > 0 THEN p^.lh := TRUE END
END
END
ELSIF p^.key < x THEN
    search(x, p^.right, h);
    IF h > 0 THEN (*la rama de la derecha ha crecido*)
        IF p^.rh THEN
            p1 := p^.right; h := 2; p^.rh := FALSE;
            IF p1^.rh THEN (*RR*)
                p^.right := p1^.left; p1^.left := p; p := p1;
                p1^.rh := FALSE
            ELSIF p1^.lh THEN (*RL*)
                p2 := p1^.left; p1^.left := p2^.right; p2^.right := p1;
                p^.right := p2^.left; p2^.left := p; p := p2;
                p1^.lh := FALSE
            END
            ELSE h := h-1;
            IF h > 0 THEN p^.rh := TRUE END
            END
        ELSE (*encontrado*) p^.count := p^.count + 1
        END
    END search
```

Nótese que las acciones que deben tomarse en el rearreglo de nodos se asemejan mucho a las desarrolladas en el algoritmo de búsqueda en árboles balanceados (4.63). En (4.87) es evidente que los cuatro casos pueden realizarse con las simples rotaciones de los apuntadores: rotaciones individuales en los casos LL y RR, rotaciones dobles en los casos LR y RL. En efecto, el procedimiento (4.87) parece ligeramente más sencillo que (4.63). Sin duda el esquema de árboles BBS se presenta como una alternativa del criterio de balance AVL. Una comparación de rendimiento es posible y conveniente.

Nos abstendremos de realizar el complejo análisis matemático y nos concentraremos en algunas diferencias básicas. Puede demostrarse que los árboles balanceados AVL son un subconjunto de los árboles BBS. Por tanto, la clase de los segundos es más amplia. De ello se deduce que su longitud de trayectoria es, en general, mayor que el caso de los árboles AVL. Nótese dentro de este contexto el árbol del peor caso (4) en la figura 4.53. Por otra parte, se necesita con menor frecuencia el rearreglo de nodos. El árbol balanceado se prefiere por ello en las aplicaciones en que las recuperaciones de llaves son mucho más frecuentes que las inserciones (o eliminaciones); si este cociente es moderado, puede preferirse el esquema del árbol BBS. Es muy difícil decir dónde se encuentra la línea límitrofe. Esta depende mucho no sólo del cociente entre las frecuencias de recuperación y el cambio estructural, sino también de las características de la instrumentación. Ello sucede sobre todo si los registros de los nodos tienen una representación de compactación densa; por tanto, el acceso a los campos requiere una selección de parte-palabra.

4.8. ARBOLES DE BUSQUEDA CON PRIORIDAD

Los árboles, y en especial los binarios, constituyen una organización muy eficaz de datos que pueden ordenarse sobre una escala lineal. En los capítulos anteriores se explicaron los planes más usados en la búsqueda y mantenimiento eficientes (inserción, eliminación). Pero los árboles no parecen ser útiles en problemas en que los datos no están localizados en un espacio unidimensional, sino en un espacio multidimensional. En efecto, la búsqueda eficiente en los espacios multidimensionales constituye todavía uno de los problemas más elusivos en la ciencia de la computación; el caso de dos dimensiones tiene particular importancia en muchas aplicaciones prácticas.

Al estudiar más detenidamente el tema, los árboles pueden seguirse aplicando con buenos resultados por lo menos en el caso bidimensional. Después de todo, trazamos árboles sobre el papel en un espacio bidimensional. En consecuencia, repasaremos brevemente las características de los dos tipos principales de árboles que hemos encontrado hasta ahora.

1. Un *árbol de búsqueda* se rige por las invariantes

$$\begin{aligned} p.\text{left} \neq \text{NIL} &\rightarrow p.\text{left}.x < p.x \\ p.\text{right} \neq \text{NIL} &\rightarrow p.x < p.\text{right}.x \end{aligned} \quad (4.88)$$

y se aplica a todos los nodos p con llave x . Es evidente que sólo la posición *horizontal* de los nodos está limitada por la invariante, y que las posiciones verticales de los nodos pueden elegirse arbitrariamente, de tal forma que se minimice el tiempo de acceso en la búsqueda (o sea, la longitud de las trayectorias).

2. Un *montón*, también llamado *árbol de prioridad*, se rige por las invariantes

$$\begin{aligned} p.\text{left} \neq \text{NIL} &\rightarrow p.y \leq p.\text{left}.y \\ p.\text{right} \neq \text{NIL} &\rightarrow p.y \leq p.\text{right}.y \end{aligned} \quad (4.89)$$

y se aplica a todos los nodos p con llave y . Aquí es evidente que únicamente están restringidas las posiciones *verticales*.

Parece sencillo combinar las dos condiciones en una definición de una organización de árboles dentro de un espacio bidimensional; cada nodo tiene dos llaves x y y , que pueden considerarse como coordenadas del nodo. Ese árbol representa un conjunto de puntos en un plano, o sea en un espacio cartesiano bidimensional; en consecuencia, se llama *árbol cartesiano* [4-9]. Preferimos la designación de *árbol de búsqueda con prioridad*, porque muestra que esta estructura nace de una combinación de árbol de prioridad y el de búsqueda. Se caracteriza por las siguientes invariantes aplicables a cada nodo p :

$$\begin{aligned} p.\text{left} \neq \text{NIL} &\rightarrow (p.\text{left}.x < p.x) \& (p.y \leq p.\text{left}.y) \\ p.\text{right} \neq \text{NIL} &\rightarrow (p.x < p.\text{right}.x) \& (p.y \leq p.\text{right}.y) \end{aligned} \quad (4.90)$$

Pero no debe sorprendernos que las propiedades de búsqueda de esos árboles no sean extraordinarias. Después de todo, se ha quitado un gran grado de libertad en el posicionamiento de los nodos y ya no se cuenta con él al escoger los arreglos que producen longitudes cortas de trayectorias. En efecto, no pueden garantizarse límites logarítmicos en los esfuerzos que exigen la búsqueda, inserción o supresión de elementos. Aunque esto lo vimos con el árbol de búsqueda ordinario y no balanceado, son escasas las probabilidades de una buena conducta promedio. Más aún, pueden resultar muy difíciles las operaciones de mantenimiento. Tomemos, por ejemplo, el caso del árbol de la figura 4.54 (a). La inserción de un nuevo nodo C cuyas coordenadas obligan a insertarlo arriba y entre A y B exige mucho esfuerzo a fin de transformar (a) en (b).

McCreight descubrió una técnica, semejante al balanceo, que garantiza límites logarítmicos de tiempo a estas operaciones pero complica más la operación de inserción y eliminación. McCreight llama a esa estructura *árbol de búsqueda con prioridad* [4-10]; sin embargo, según nuestra clasificación debería denominarse *árbol balanceado de búsqueda con prioridad*. Nos abstenemos de explicar esa estructura, dado que es muy intrincada y rara vez se utiliza en la práctica. Al examinar un problema más restringido pero no menos importante en la práctica, McCreight llegó a otra estructura de árbol, que examinaremos aquí a fondo. En lugar de suponer que el espacio de búsqueda no está acotado, consideraremos que el espacio de datos está delimitado por un rectángulo con dos lados abiertos. Denotamos los valores limitantes de la coordenada x por x_{\min} y x_{\max} .

En la estructura de árbol (no balanceado) de búsqueda con prioridad descrita antes, cada nodo p divide el plano en dos partes a lo largo de la línea $x = p.x$. Todos los nodos del subárbol de la izquierda se hallan a su izquierda, todos los del de la derecha se encuentran a su derecha. Desde el punto de vista de eficiencia de la búsqueda esta elección quizás no sea eficiente. Por fortuna, podemos seleccionar de modo diferente la línea divisoria. Asociemos con cada nodo p un intervalo $[p.L \dots p.R]$, que comprende todos los valores x , incluyendo $x.L$, hasta $x.R$ que no se incluye. Este será el intervalo en que el valor x del nodo p puede encontrarse. Así pues, postulamos que el descendiente de la izquierda (si lo hay) debe hallarse dentro de la mitad de la izquierda, y el descendiente de la derecha dentro de la mitad derecha de este intervalo. Por tanto, la línea divisoria no es $p.x$.

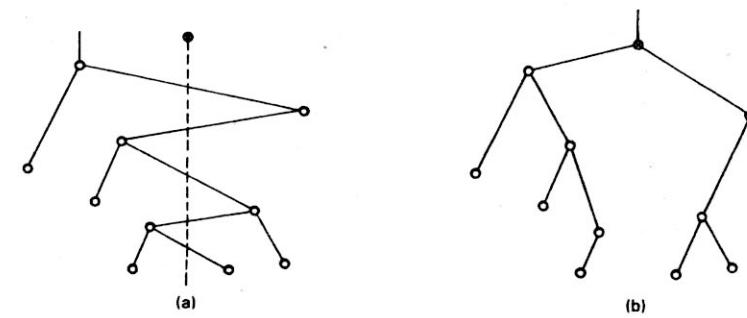


Fig. 4.54 Inserción en un árbol de búsqueda con prioridad.

sino $(p.L + p.R)/2$. Para cada descendiente el intervalo está dividido a la mitad, con lo cual limita la altura del árbol a $\log(x_{\max} - x_{\min})$. Este resultado es válido sólo si no hay dos nodos que tengan el mismo valor x , una condición garantizada por la invariante (4.90). Si trabajamos con coordenadas enteras, este límite es a lo sumo igual a la longitud de palabra de la computadora que se emplea. En efecto, la búsqueda se efectúa como una búsqueda de biseción o raíz; por consiguiente, a esos árboles se les llama *árboles de búsqueda con prioridad de raíz* [4-10]. Presentan *límites logarítmicos* en el número de operaciones requeridas para buscar, insertar y eliminar un elemento; se rigen por las siguientes invariantes de cada nodo p :

$$\begin{aligned} p.left \neq NIL &\rightarrow (p.L \leq p.left.x < p.M) \& (p.y \leq p.left.y) \\ p.right \neq NIL &\rightarrow (p.M \leq p.right.x < p.R) \& (p.y \leq p.right.y) \end{aligned} \quad (4.91)$$

donde

$$\begin{aligned} p.M &= (p.L + p.R) \text{ DIV } 2 \\ p.left.L &= p.L \\ p.left.R &= p.M \\ p.right.L &= p.M \\ p.right.R &= p.R \end{aligned}$$

para todos los nodos p , y $\text{root}.L = x_{\min}$, $\text{root}.R = x_{\max}$.

Una ventaja decisiva del esquema de raíz consiste en que las operaciones de mantenimiento (conservar las invariantes en la inserción y eliminación) se limitan a una sola espina del árbol, dado que las líneas divisorias poseen valores fijos de x , prescindiendo de los valores x de los nodos insertados.

Las operaciones típicas en los árboles de búsqueda con prioridad son la inserción, la eliminación, la obtención de un elemento que tenga el valor mínimo (máximo) de x ($o y$) mayor (más pequeño) que un límite determinado y la enumeración de puntos existentes dentro de un rectángulo. A continuación se dan los procedimientos de inserción y enumeración; se basan en las siguientes declaraciones de tipo:

```
TYPE Ptr = POINTER TO Node;
Node = RECORD
  x: [xmin .. xmax]; y: CARDINAL;
  left, right: Ptr
END
```

(4.92)

Obsérvese que los atributos xL y xR no necesariamente deben registrarse en los nodos. Más bien se calculan durante cada búsqueda. Pero para ello se requieren dos parámetros adicionales del procedimiento recursivo *insert*. Sus valores en la primera llamada (con $p =$ raíz) son x_{\min} y x_{\max} , respectivamente. Además, una búsqueda se efectúa en forma similar a la que se hace con un árbol regular de búsqueda. Si nos encontramos con un nodo vacío, se inserta el elemento. Si el nodo que debe introducirse tiene un valor y más pequeño que el que va a ser examinado, el nuevo nodo se intercambia con aquél. Por último, el nodo se inserta en el subárbol de la izquierda si su valor x es menor que el valor de la mitad del intervalo; de lo contrario se inserta en el subárbol de la derecha.

```
PROCEDURE insert(VAR p: Ptr; X, Y, xL, xR: CARDINAL);
  VAR xm, t: CARDINAL;
BEGIN
  IF p = NIL THEN (*no está en el arbol, insertar*)
    ALLOCATE(p, SIZE(Node));
    WITH p DO
      x := X; y := Y; left := NIL; right := NIL
  END
  ELSIF p.x = X THEN (*encontrado; no insertar*)
  ELSE
    IF p.y > Y THEN
      t := p.x; p.x := X; X := t;
      t := p.y; p.y := Y; Y := t
    END ;
    xm := (xL + xR) DIV 2;
    IF X < xm THEN insert(pt.left, X, Y, xL, xm)
    ELSE insert(pt.right, X, Y, xm, xR)
    END
  END
END insert
```

La tarea de enumerar todos los puntos x, y que se hallan en un rectángulo determinado, o sea satisfacer $x_0 \leq x < x_1$ y $y \leq y_1$, se realiza mediante el procedimiento *enumerate*. Este llama a un procedimiento *report(x, y)* para cada punto encontrado. Nótese que un lado del rectángulo se halla en el eje x , es decir, el límite inferior de y es 0. Con ello se garantiza que la enumeración requiere a lo máximo $O(\log N + s)$ operaciones, donde N es la cardinalidad del espacio de búsqueda en x y s es el número de nodos enumerados.

```
PROCEDURE enumerate(p: Ptr; x0, x1, y, xL, xR: CARDINAL);
  VAR xm: CARDINAL;
BEGIN
  IF p # NIL THEN
    IF (pt.y <= y) & (x0 <= pt.x) & (pt.x < x1) THEN
      report(pt.x, pt.y)
    END ;
    xm := (xL + xR) DIV 2;
    IF x0 < xm THEN enumerate(pt.left, x0, x1, y, xL, xm) END ;
    IF xm < x1 THEN enumerate(pt.right, x0, x1, y, xm, xR) END
  END
END enumerate
```

EJERCICIOS

- 4.1. Introduzcamos la noción de un *tipo recursivo*, que debe declararse así:

```
RECTYPE T = T0
```

y que denota el conjunto de valores definidos por el tipo T0 aumentado por el valor individual NONE.

La definición del tipo *ped* [véase (4.3)], por ejemplo, podría entonces simplificarse

```
RECTYPE ped = RECORD name: alfa;
               father, mother: ped
             END
```

¿Cuál es el patrón de almacenamiento de la estructura recursiva correspondiente a la figura 4.2?

Cabe suponer que una instrumentación de esa característica se basará en una estructura de asignación dinámica de memoria y que los campos llamados *father* y *mother* en el ejemplo precedente contendrán apuntadores generados automáticamente pero ocultos al programador. ¿Cuáles son los problemas que plantea la realización de esa característica?

- 4.2. Defina la estructura de datos descrita en el último párrafo de la sección 4.2 en función de registros y apuntadores. ¿Es también posible representar esta constelación de familias a partir de tipos recursivos como se propuso en el ejercicio anterior?
- 4.3. Suponga que una cola Q de primero en entrar primero en salir (fifo), con elementos de tipo T0, se instrumenta como una lista ligada. Defina un módulo con una estructura adecuada de datos, los procedimientos para insertar y extraer un elemento de Q y una función que comprueba si la cola está vacía. Los procedimientos habrán de tener su propio mecanismo para un reuso económico del almacenamiento.
- 4.4. Suponga que los registros de una lista ligada contienen un campo de llaves de tipo INTEGER. Escriba un programa para clasificar la lista por orden del valor decreciente de las llaves. Después construya un procedimiento para invertir la lista.
- 4.5. Las listas circulares (véase la figura 4.55) suelen establecerse con un *encabezado de lista*. ¿Por qué se usa éste? Escriba procedimientos para insertar, eliminar y buscar un elemento identificado por una llave determinada. Haga eso una vez suponiendo que existe un encabezado y otra vez sin encabezado.
- 4.6. La *lista bidireccional* es una lista de elementos ligados en ambas direcciones. (Véase la figura 4.56.) Ambas ligas nacen de un encabezado. En forma parecida a lo hecho en el ejercicio anterior, construya un módulo con procedimientos de búsqueda, inserción y eliminación de elementos.

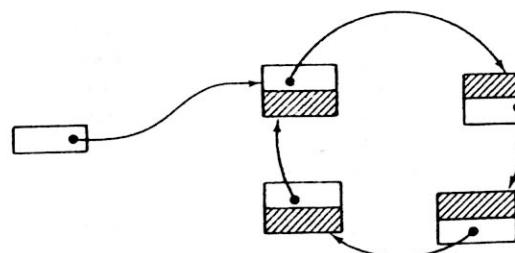


Fig. 4.55 Lista circular.

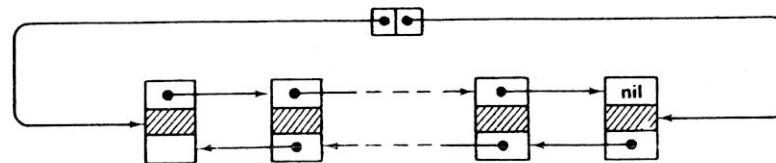


Fig. 4.56 Lista bidireccional

- 4.7. ¿Funciona correctamente el programa 4.2 si cierto par $\langle x, y \rangle$ ocurre más de una vez en la entrada?
- 4.8. El mensaje “Este conjunto no está parcialmente ordenado” en el programa 4.2 no es muy útil en muchas ocasiones. Amplíe el programa de modo que produzca una secuencia de elementos que formen un ciclo si es que hay uno.
- 4.9. Escriba un programa que lea un texto de programa, identifique todas las definiciones de procedimiento y las llamadas; trate de establecer un orden topológico entre las subrutinas. Supongamos que $P < Q$ significa que Q llama a P.
- 4.10. Trace un árbol construido por el programa 4.3, si la entrada consta de los números naturales 1, 2, 3, ..., n.
- 4.11. ¿Cuáles son las secuencias de los nodos encontrados cuando se recorre el árbol de la figura 4.23 en preorden, en orden y en postorden?
- 4.12. Encuentre una regla de composición para la secuencia de n números que, si se aplica al programa 4.4, produce un árbol perfectamente balanceado.
- 4.13. Examine los dos órdenes siguientes para recorrer árboles binarios:
 - a1. Recorrer el subárbol de la derecha.
 - a2. Visitar la raíz.
 - a3. Recorrer el subárbol de la izquierda.
 - b1. Visitar la raíz.
 - b2. Recorrer el subárbol de la derecha.
 - b3. Recorrer el subárbol de la izquierda.

¿Hay relaciones sencillas entre las secuencias de los nodos encontrados después de estos órdenes y los generados por los tres órdenes definidos en el texto?

- 4.14. Defina una estructura de datos para representar árboles n-arios. Luego escriba un procedimiento que atraviese árboles n-arios y que genere un árbol binario que contenga los mismos elementos. Suponga que la llave guardada en un elemento ocupa k palabras y que cada apuntador ocupa una palabra de memoria. ¿Qué ganancia de almacenamiento se obtiene cuando se usa un árbol binario en comparación con el uso de un árbol n-ario?

- 4.15. Suponga que se construye un árbol a partir de la siguiente definición de una estructura recursiva de datos (véase el ejercicio 4.1). Formule un procedimiento para encontrar un elemento con cierta llave y para realizar una operación P en ese elemento.

```
RECTYPE Tree = RECORD x: INTEGER;
    left, right: Tree
END
```

- 4.16. En cierto sistema de archivos un directorio está organizado como un árbol binario ordenado. Cada nodo denota un archivo y especifica el nombre de archivo y, entre otras cosas, la fecha de su último acceso codificada como un entero. Escriba un programa que recorra el árbol y elimine todos los archivos cuyo último acceso fue antes de cierta fecha.

- 4.17. En una estructura de árbol, la frecuencia de acceso de cada elemento se mide empíricamente atribuyéndole un conteo de acceso. En ciertos intervalos de tiempo, la organización del árbol se actualiza recorriendo el árbol y generando otro por medio del programa 4.4 e insertando las llaves por orden de conteo de frecuencia decreciente. Escriba un programa que realice esta reorganización. ¿Es la longitud promedio de trayectoria de este árbol igual, peor o mucho peor que la de un árbol óptimo?

- 4.18. El método de analizar el algoritmo de inserción en árbol, descrito en la sección 4.5 también puede servir para calcular los números esperados C_n de comparaciones y M_n de movimientos (intercambios) que se llevan a cabo en la clasificación rápida (programa 2.10) al clasificar n elementos en un arreglo, suponiendo que todas las $n!$ permutaciones de n llaves 1, 2, ..., n tienen la misma probabilidad. Encuentre la analogía y determine C_n y M_n .

- 4.19. Trace un árbol balanceado con 12 nodos que tenga la altura máxima de todos los árboles balanceados de 12 nodos. ¿En qué secuencia tienen que insertarse los nodos de modo que el procedimiento (4.63) genere este árbol?

- 4.20. Encuentre una secuencia de n llaves de inserción para que el procedimiento (4.63) realice las cuatro acciones de rebalanceo (LL, LR, RR, RL) una vez por lo menos. ¿Cuál es la longitud mínima de n de esa secuencia?

- 4.21. Encuentre un árbol balanceado con las llaves 1 ... n y una permutación de esas llaves de modo que, cuando se aplique al procedimiento de eliminación (4.64), este procedimiento efectúe por lo menos una vez las cuatro rutinas de rebalanceo. ¿Cuál es la secuencia que tiene la longitud mínima n?

- 4.22. ¿Cuál es la longitud promedio de trayectoria del árbol T_n de Fibonacci?

- 4.23. Escriba un programa que genere un árbol casi óptimo según el algoritmo basado en la selección de un centroide como raíz (4.78).

- 4.24. Suponga que las llaves 1, 2, 3, ... se insertan en un árbol B vacío de orden 2 (programa 4.7). ¿Cuáles llaves hacen que se dividan las páginas? ¿Cuáles llaves hacen que aumente la altura del árbol? Si las llaves se suprimen en el mismo orden, ¿cuáles hacen que se mezclen las páginas (y se eliminan) y cuáles aumentan la altura? Conteste la pregunta para (a) una estructura de eliminación que utilice el balanceo (como en el programa 4.7) y (b) para un esquema sin balanceo (cuando se presente la insuficiencia, se extrae un solo elemento de una página vecina).

- 4.25. Escriba un programa para buscar, insertar y eliminar llaves en un árbol B binario. Use la definición de tipo de nodo (4.84). Este esquema de inserción aparece en la figura 4.51.

- 4.26. Encuentre una secuencia de llaves de inserción que, comenzando con el árbol B binario simétrico vacío, haga que el procedimiento (4.87) realice las cuatro acciones de rebalanceo (LL, LR, RR, RL) una vez por lo menos. ¿Cuál es la secuencia más corta?

- 4.27. Escriba un procedimiento que elimine los elementos en un árbol B binario simétrico. Despues encuentre un árbol y una corta secuencia de eliminaciones que haga que las cuatro situaciones de rebalanceo ocurran una vez por lo menos.

- 4.28. Formule una estructura y procedimiento de datos para introducir y suprimir un elemento en un árbol de búsqueda con prioridad. El procedimiento debe conservar las invariantes (4.90). Compare su rendimiento con el del árbol de búsqueda con prioridad de raíz.

- 4.29. Diseñe un módulo con los siguientes procedimientos que operan sobre árboles de búsqueda con prioridad de raíz:

- insertar un punto con las coordenadas x, y.
- enumerar todos los puntos dentro de un rectángulo especificado.
- encontrar el punto con la menor coordenada x en un rectángulo determinado.
- encontrar el punto que tenga la mayor coordenada y dentro de un rectángulo determinado.
- enumerar todos los puntos que se encuentren dentro de dos rectángulos (en intersección).

BIBLIOGRAFIA

- 4-1. G.M. Adelson-Velskii y E.M. Landis. *Doklady Akademia Nauk SSSR*, 146 (1962), 263-66; English Translation in *Soviet Math*, 3, 1259-63.
- 4-2. R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1, No. 3 (1972), 173-89.
- 4-3. ————, Binary B-trees for Virtual memory. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Nov. 1971, pp. 219-35.
- 4-4. ————, Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1, No. 4 (1972), 290-306.
- 4-5. T.C. Hu and A.C. Tucker. *SIAM J. Applied Math*, 21, No. 4 (1971) 514-32.
- 4-6. D.E. KNUTH. Optimum Binary Search Trees. *Acta Informatica*, 1, No. 1 (1971), 14-25.
- 4-7. W.A. Walker and C.C. Gotlieb. A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees. in *Graph Theory and Computing* (New York: Academic Press, 1972), pp. 303-23.
- 4-8. D. Comer. The ubiquitous B-Tree. *ACM Comp. Surveys*, 11, 2 (June 1979), 121-137.
- 4-9. J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23, 4 (April 1980), 229-239.
- 4-10. E.M. McCreight. Priority search trees. *SIAM J. of Comp.* (May 1985)

5. TRANSFORMACIONES DE LLAVES (HASHING)

5.1. INTRODUCCION

La principal cuestión expuesta en el capítulo 4 con mucha amplitud es la siguiente: si tenemos un conjunto de elementos caracterizados por una llave (sobre la cual se define una relación de ordenación), ¿cómo debemos organizar el conjunto para que la recuperación de un elemento con cierta llave se realice con el menor esfuerzo posible? Desde luego, en una memoria de computadora se acude a cada elemento especificando una dirección de la memoria. De ahí que el problema radique fundamentalmente en encontrar un mapeo apropiado H de llaves (K) en las direcciones (A):

$$H: K \rightarrow A$$

En el capítulo 4 ese procedimiento se efectuó mediante varias listas y algoritmos de búsqueda en árbol, basadas en la organización de datos utilizada. Presentamos ahora otro método que, en lo esencial, es sencillo y sumamente eficiente en muchos casos. Más adelante se comenta que también ofrece algunas desventajas.

La organización de datos usada en esta técnica es la estructura de arreglo. H es, por tanto, un mapeo que transforma las llaves en índices de arreglo, y por eso se emplea la designación *transformación de llaves* (cálculo de dirección) que suele usarse para nombrar esta técnica. Conviene mencionar que no necesitamos basarnos en ningún procedimiento de asignación dinámica; el arreglo es una de las estructuras estáticas fundamentales. El método de transformaciones de llaves suele emplearse en áreas problema donde las estructuras de árbol son competidores parecidos.

La dificultad principal al utilizar una transformación de llaves es que el conjunto de posibles valores básicos resulta mucho mayor que el conjunto de direcciones disponibles de memoria (índices de arreglo). Tomemos, por ejemplo, el caso de nombres formados hasta por 16 letras como llaves que identifican a los individuos en un grupo de mil personas. En consecuencia, habrá 26^{16} llaves posibles que deben mapearse en 10^3 índices posi-

bles. La función H es, pues, evidentemente una función de muchos contra uno. Si tenemos una llave k , el primer paso en una operación de recuperación (búsqueda) consiste en calcular su índice asociado $h = H(k)$; el segundo (sin duda necesario) consiste en verificar si el elemento con la llave K es realmente identificado por h en el arreglo (tabla) T , o sea verificar si $T[H(k)].llave = k$. De inmediato nos hallamos ante dos preguntas:

1. ¿Qué tipo de función H debe usarse?
2. ¿Cómo resolvemos el problema de que H no produce la localización del elemento deseado?

La respuesta a la segunda pregunta es que debe aplicarse algún método para obtener una localización alternativa, digamos el índice h' y, si esto no es todavía la localización del elemento deseado, se obtendrá un tercer índice h'' , y así sucesivamente. Se llama *colisión* al caso en que una llave que no sea la deseada se encuentra en la localización identificada; la tarea de generar índices alternos recibe el nombre de *manejo de colisiones*. En el siguiente apartado describiremos la elección de una función de transformación y los métodos para el manejo de las colisiones.

5.2. ELECCION DE UNA FUNCION DE TRANSFORMACION DE LLAVES (HASH)

Un requisito básico de una buena función de transformación es que distribuya las llaves lo más uniformemente posible sobre la gama de valores índice. Además de satisfacer ese requisito, la distribución no está ligada a patrón alguno y, en realidad, es conveniente que dé la impresión de ser enteramente aleatoria. Esta propiedad le da al método el nombre poco científico de *hashing* (picadillo), es decir, desmenuzar el argumento o hacer una mescolanza. H recibe el nombre de *función de transformación*. Sin duda debe ser calculable de modo eficiente, o sea, estar compuesta de un reducido número de operaciones aritméticas básicas.

Supongamos que una función de transferencia $ORD(k)$ está disponible y que denote el número ordinal de la llave k en el conjunto de todas las llaves posibles. Supongamos asimismo que los índices de arreglo i fluctúe entre los enteros $0 \dots N - 1$, donde N es el tamaño del arreglo. Así pues, una elección obvia será

$$H(k) = ORD(k) \text{ MOD } N \quad (5.1)$$

Esta elección tiene la propiedad de que los valores de las llaves están distribuidos uniformemente sobre el intervalo índice, y por lo mismo constituyen la base de la mayor parte de las transformaciones de llaves. También se calcula de manera eficiente si N es una potencia de 2. Pero es precisamente este caso el que hay que evitar si las llaves son secuencias de letras. La suposición de que todas las llaves tienen la misma probabilidad es errónea en este caso. En efecto, las palabras que difieren sólo en unos cuantos caracteres se mapean seguramente en índices idénticos, con lo cual efectivamente producen una distribución muy poco uniforme. De ahí que se recomienda encarecidamente hacer que N sea un número primo [4-7]. Ello tiene la consecuencia de que se requiere una operación de división que no es posible sustituir por un simple enmascaramiento de dígitos binarios, pero eso no es un inconveniente grave en la mayor parte de las computadoras modernas que están provistas de una instrucción incorporada de división.

A menudo se recurre a funciones de transformación consistentes en aplicar operaciones lógicas como la *o exclusiva* para algunas partes de las llaves representadas como una secuencia de dígitos binarios. Estas operaciones pueden ser más rápidas que la división en algunas computadoras; pero en ocasiones no logran en absoluto distribuir las llaves uniformemente sobre el intervalo de índices. Por ello nos abstendremos de describir con más detalles tales métodos.

5.3. MANEJO DE LAS COLISIONES

Si resulta que un elemento de una tabla correspondiente a determinada llave no es el elemento deseado, entonces existe colisión, es decir, dos elementos tienen llaves que los manejan en el mismo índice. Un segundo sondeo es necesario, basado en un índice obtenido en forma determinística a partir de la llave dada. Se cuenta con varios métodos para generar índices secundarios. Uno muy conocido consiste en ligar todos los elementos (entradas) con el índice primario idéntico $H(k)$ en una lista ligada. A esto se le llama *encadenamiento directo*. Los elementos de la lista pueden estar en la tabla primaria o no; en el segundo caso, se da el nombre de *área de desbordamiento* al almacenamiento donde están asignados. Este método tiene la desventaja de que deben conservarse las listas secundarias y de que cada elemento (entrada) ha de reservar espacio para un apuntador (o índice) a su lista de elementos en colisión.

Otra solución del problema de las colisiones estriba en prescindir enteramente de las listas y, en su lugar, limitarse a observar las otras entradas (elementos) en la misma tabla hasta encontrar el elemento o una posición abierta; entonces se supondrá que la llave especificada no se halla en la tabla. Este método recibe el nombre de *dirección abierta* [4-9]. Por supuesto, la secuencia de los índices de las exploraciones secundarias siempre debe ser igual para una llave determinada. A continuación se ofrece un esquema del algoritmo de una búsqueda en tablas:

```

 $h := H(k); i := 0;$ 
REPEAT
  IF  $T[h].key = k$  THEN elemento encontrado
  ELSIF  $T[h].key = \text{free}$  THEN el elemento no está en la tabla
  ELSE (*colisión*)
     $i := i + 1; h := H(k) + G(i)$ 
  END
  UNTIL encontrado o no está en la tabla (o tabla llena)

```

(5.2)

En la literatura sobre el tema se han propuesto varias funciones para resolver las colisiones. Un estudio del tema hecho por Morris en 1968 [4-8] estimuló abundantes actividades en esta área. El método más sencillo consiste en ensayar la siguiente localización (suponiendo que la tabla es circular) hasta encontrar el elemento con la llave especificada o una localización vacía. Por tanto, $G(i) = i$; los índices h_i usados para hacer la exploración en este caso son

$$\begin{aligned}
h_0 &= H(k) \\
h_i &= (h_0 + i) \text{ MOD } N, \quad i = 1 \dots N-1
\end{aligned}$$
(5.3)

Este método se llama *exploración lineal* y tiene la desventaja de que las entradas tienden a agruparse alrededor de las llaves primarias (o sea las que no han sufrido colisión después de la inserción). En teoría, debe elegirse una función G que otra vez distribuya

uniformemente las llaves en el resto del conjunto de localizaciones. Pero, en la práctica, ello suele ser demasiado costoso, por lo cual se prefieren los métodos que representen un compromiso por ser fácil de calcular y, al mismo tiempo, superiores a la función lineal (5.3). Uno de ellos consiste en aplicar una función cuadrática tal que la secuencia de índices de la búsqueda sea

$$\begin{aligned}
h_0 &= H(k) \\
h_i &= (h_0 + i^2) \text{ MOD } N \quad i > 0
\end{aligned}$$
(5.4)

Nótese que el cálculo del siguiente índice no necesariamente requiere la operación de elevar al cuadrado, si nos valemos de las relaciones de recurrencia (5.5) para $h_i = i^2$ y $d_i = +1$.

$$\begin{aligned}
h_{i+1} &= h_i + d_i \\
d_{i+1} &= d_i + 2 \quad (i > 0)
\end{aligned}$$
(5.5)

con h_0 y $d_0 = 1$. A esto se le llama *exploración cuadrática*, y esencialmente evita la agrupación primaria aunque no se necesiten cálculos adicionales. Una pequeña desventaja es que no se realiza la búsqueda en todas las tablas, esto es, después de la inserción quizás no encontremos una ranura libre aunque todavía queden algunos. En efecto, en la exploración por lo menos se visita la mitad de la tabla si su tamaño N es un número primo. Esta aseveración puede derivarse del siguiente razonamiento. Si las búsquedas i -ésima y j -ésima coinciden en la misma entrada de la tabla, esto podemos expresarlo mediante la ecuación

$$\begin{aligned}
i^2 \text{ MOD } N &= j^2 \text{ MOD } N \\
(i^2 - j^2) &\equiv 0 \pmod{N}
\end{aligned}$$

Al dividir las diferencias en dos factores obtenemos

$$(i + j)(i - j) \equiv 0 \pmod{N}$$

y como $i \neq j$, nos damos cuenta de que i o j tienen por lo menos $N/2$ a fin de producir $i + j = cN$, siendo c un entero. En la práctica, ese inconveniente carece de importancia, pues tener que realizar $N/2$ exploraciones secundarias y evasiones de colisión es muy raro y ocurre sólo si la tabla ya casi está llena.

El programa generador de referencias cruzadas 4.5 está reescrito en la forma del programa 5.1, para aplicar la técnica de almacenamiento por dispersión. Las diferencias principales radican el procedimiento *search* y en la sustitución del tipo de apuntador *WPtr* por la tabla de palabras T . La función de transformación H es el módulo del tamaño de la tabla; se escogió la exploración cuadrática para manejar las colisiones. Obsérvese que es indispensable, para lograr un buen rendimiento, que el tamaño de la tabla sea un número primo.

Aunque el método de transformación de llaves es el mejor en este caso (en realidad, más eficiente que las organizaciones de árbol) presenta una desventaja. Luego de analizar el texto y reunir las palabras, queríamos tabularlas por orden alfabetico. Esto es fácil cuando se utiliza una organización de árbol, pues su base misma es la búsqueda de ár-

bol *ordenado*. Pero no lo es cuando se recurre a las transformaciones de llaves. Entonces comprendemos el significado exacto del término *transformación de llaves* (hashing). No sólo deberá la salida impresa de la tabla estar precedida por un proceso de clasificación (que se omite en el programa 5.1), sino que incluso resultó ser útil llevar un control de las llaves insertadas ligándolas explícitamente en una lista. En consecuencia, el rendimiento superior de este método, considerando únicamente el proceso de recuperación, está contrarrestado en parte por las operaciones adicionales que se requieren para terminar la tarea de generar un índice ordenado de referencias cruzadas.

```

MODULE XRef;
FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, Done, EOL, Write, WriteCard, WriteString, WriteLn;
FROM Storage IMPORT ALLOCATE;

CONST P = 997; (*primo, tamaño de tabla*)
  BufLeng = 10000; WordLeng = 16;
  free = 0;

TYPE WordInx = [0 .. P-1];
  ItemPtr = POINTER TO Item;

  Word = RECORD key: CARDINAL;
    first, last: ItemPtr;
  END;

  Item = RECORD lno: CARDINAL;
    next: ItemPtr
  END;

VAR k0, k1, line: CARDINAL;
  ch: CHAR;
  T: ARRAY [0 .. P-1] OF Word; (*tabla de transformacion de llaves*)
  buffer: ARRAY [0 .. BufLeng-1] OF CHAR;

PROCEDURE PrintWord(k: CARDINAL);
  VAR lim: CARDINAL;
BEGIN lim := k + WordLeng;
  WHILE buffer[k] > 0C DO Write(buffer[k]); k := k+1 END ;
  WHILE k < lim DO Write(" "); k := k+1 END
END PrintWord;

PROCEDURE PrintTable;
  VAR i, k, m: CARDINAL; item: ItemPtr;
BEGIN
  FOR k := 0 TO P-1 DO
    IF T[k].key # free THEN
      PrintWord(T[k].key); item := T[k].first; m := 0;
      REPEAT

```

```

        IF m = 8 THEN
          WriteLn; m := 0;
          FOR i := 1 TO WordLeng DO Write(" ") END
        END;
        m := m + 1; WriteCard(item^.lno, 6); item := item^.next
      UNTIL item = NIL;
      WriteLn;
    END
  END
END PrintTable;
PROCEDURE Diff(i, j: CARDINAL): INTEGER;
BEGIN
  LOOP
    IF buffer[i] # buffer[j] THEN
      RETURN INTEGER(ORD(buffer[i])) - INTEGER(ORD(buffer[j]))
    ELSIF buffer[i] = 0C THEN RETURN 0
    END;
    i := i+1; j := j+1
  END
END Diff;
PROCEDURE search;
  VAR i, h, d: CARDINAL; found: BOOLEAN;
  ch: CHAR; x: ItemPtr;
  (*variables globales: T buffer, k0, k1*)
BEGIN (*calcular el indice de transformacion de llaves para palabra
       que empieza en buffer [k0]*)
  i := k0; h := 0; ch := buffer[i];
  WHILE ch > 0C DO
    h := (256*h + ORD(ch)) MOD P; i := i+1; ch := buffer[i]
  END;
  ALLOCATE(x, SIZE(item)); x^.lno := line; x^.next := NIL;
  d := 1; found := FALSE;
  REPEAT
    IF Diff(T[h].key, k0) = 0 THEN (*coincidencia*)
      found := TRUE; T[h].last^.next := x; T[h].last := x
    ELSIF T[h].key = free THEN (*nueva entrada*)
      WITH T[h] DO
        key := k0; first := x; last := x
      END;
      found := TRUE; k0 := k1
    ELSE (*colision*) h := h+d; d := d+2;
      IF h >= P THEN h := h-P END;
      IF d = P THEN WriteString("Desbordamiento de tabla"); HALT END
    END
  UNTIL found
END search;

```

```

PROCEDURE GetWord;
BEGIN k1 := k0;
  REPEAT Write(ch); buffer[k1] := ch; k1 := k1 + 1; Read(ch)
  UNTIL (ch < "0") OR (ch > "9") & (CAP(ch) < "A")
    OR (CAP(ch) > "Z");
  buffer[k1] := 0C; k1 := k1 + 1; (* terminador *)
  search
END GetWord;
BEGIN k0 := 1; line := 0;
  FOR k1 := 0 TO P-1 DO T[k1].key := free END ;
  OpenInput ("TEXTO"); OpenOutput("XREF");
  WriteCard(0, 6); Write(" "); Read(ch);
  WHILE Done DO
    CASE ch OF
      0C .. 35C: Read(ch) |
      36C .. 37C: WriteLn; Read(ch); line := line + 1;
                    WriteCard(line, 6); Write(" ") |
      " " .. "@": Write(ch); Read(ch) |
      "A" .. "Z": GetWord |
      "[" .. "]": Write(ch); Read(ch) |
      "a" .. "z": GetWord |
      "{" .. "~": Write(ch); Read(ch)
    END
  END ;
  WriteLn; WriteLn; CloseInput;
  PrintTable; CloseOutput
END XRef.

```

Programa 5.1 Generador de referencia cruzada que utiliza una tabla de transformación de llaves.

5.4. ANALISIS DE LA TRANSFORMACION DE LLAVES

La inserción y recuperación por transformación de llaves da evidentemente un rendimiento pésimo en el peor caso. Después de todo, es muy posible que un argumento de búsqueda sea tal que las búsquedas encuentren precisamente todas las localizaciones ocupadas, sin hallar las que se desean (o libres). En realidad, todo lo que aplique esta técnica necesita mucha confianza en la corrección de las leyes de la teoría de probabilidad. Debemos tener la seguridad de que, *en promedio*, el número de búsquedas es pequeño. El siguiente argumento probabilístico revela que es incluso sumamente pequeño.

Supongamos otra vez que todas las llaves posibles poseen la misma probabilidad y que la función de transformación H las distribuye uniformemente sobre el intervalo de los índices de tabla. Supongamos, pues, que una llave debe ser insertada en una tabla de tamaño n que ya contiene k elementos. La probabilidad de encontrar una localización libre la primera vez será entonces $(n-k)/n$. Esta es además la probabilidad p_1 de que se necesite una sola comparación. La probabilidad de que se requiera exactamente una segunda exploración es igual a la probabilidad de una colisión en el primer intento, multiplicada por la probabilidad de hallar una localización libre la siguiente vez. En general, obtenemos la probabilidad p_i de una inserción que requiere exactamente i exploraciones como

$$\begin{aligned}
 p_1 &= (n-k)/n \\
 p_2 &= (k/n) * (n-k)/(n-1) \\
 p_3 &= (k/n) * (k-1)/(n-1) * (n-k)/(n-2) \\
 &\dots \\
 p_i &= (k/n) * (k-1)/(n-1) * (k-2)/(n-2) * \dots * (n-k)/(n-i+1)
 \end{aligned} \tag{5.6}$$

El número esperado E de exploraciones requeridas después de insertar la llave $k+1$ -ésima será entonces

$$\begin{aligned}
 E_{k+1} &= \text{Si } 1 \leq i \leq k+1 : i * p_i \\
 &= 1 * (n-k)/n + 2 * (k/n) * (n-k)/(n-1) + \dots + \\
 &\quad (k+1) * (k/n) * (k-1)/(n-1) * (k-2)/(n-2) * \dots * 1/(n-k+1) \\
 &= (n+1)/(n-k+1)
 \end{aligned} \tag{5.7}$$

Puesto que el número de búsquedas requeridas para insertar un elemento es idéntico al número de las que se necesitan para recuperarlo, el resultado (4.94) puede servir para calcular el número promedio E de exploraciones necesarias para acceder a una llave aleatoria de una tabla. Denotemos otra vez con n el tamaño de la tabla y supongamos que m es el número de llaves presentes en ella. Por tanto,

$$\begin{aligned}
 E &= (\sum_{k=1}^m E_k) / m \\
 &= (n+1) * (\sum_{k=1}^m (n-k+1) / (n-k+2)) / m \\
 &= (n+1) * (H_{n+1} - H_{n-m+1}) / m
 \end{aligned} \tag{5.8}$$

donde H es la función armónica. H puede ser aproximada como $H_n = \ln(n) + g$, donde g es la constante de Euler. Más aún, si sustituimos $a = m/(n+1)$, obtenemos

$$\begin{aligned} E &= (\ln(n+1) - \ln(n-m+1))/a = \ln((n+1)/(n-m+1))/a \\ &= -\ln(1-a)/a \end{aligned} \quad (5.9)$$

a es aproximadamente el cociente de las localizaciones ocupadas y disponibles, llamadas *factor de carga*, una $a = 0$ supone una tabla vacía, $a = n/(n+1)$ una tabla llena. El número previsto E de exploraciones para recuperar o insertar una llave elegida de modo aleatorio viene en la tabla 5.1 en función del factor de carga. Los resultados numéricos son realmente sorprendentes y explican el excelente rendimiento del método de transformación de llaves. Aun cuando una tabla esté llena en un 90%, en general apenas se necesitan 2.56 exploraciones para encontrar la llave o una localización vacía. Nótese en especial que esta cifra no depende del número absoluto de llaves presentes, sino sólo del factor de carga.

El análisis anterior se basó en el uso de una colisión que maneja el método que distribuye las llaves uniformemente sobre el resto de las localizaciones. Los métodos que se aplican en la práctica producen un rendimiento un poco peor. El análisis detallado de la búsqueda lineal produce un número previsto de búsquedas como

$$E = (1 - a/2)/(1 - a) \quad (5.10)$$

Algunos valores numéricos de $E(a)$ se dan en la tabla 4.7 [5-4]. Los resultados obtenidos incluso con el método más deficiente del manejo de colisiones son tan buenos, que existe la tentación de considerar la transformación de llaves (hashing) como la panacea para todo. Ello se debe en especial a que su rendimiento es superior aun a la organización de árbol más intrincada, por lo menos si se tienen en cuenta los pasos de comparación indispensables para la recuperación e inserción. De ahí la importancia de señalar explícitamente algunas limitaciones de esa técnica, aun cuando sean obvias si se examinan sin prejuicio alguno.

Sin duda la principal desventaja respecto a las técnicas que se sirven de la asignación dinámica radica en que el tamaño de la tabla es fijo y no puede ajustarse a la exigencia del momento. Una buena estimación *a priori* del número de elementos dato por clasificar será obligatoria por eso, si queremos evitar una mala utilización de la memoria o un rendimiento insatisfactorio (o incluso un desbordamiento de la tabla). Aun cuando se

a	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.9	2.56
0.95	3.15
0.99	4.66

Tabla 5.1 Número esperado de exploraciones en función de factor carga.

a	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Tabla 5.2 Número esperado de exploraciones en la exploración lineal.

conozca con exactitud el número de elementos (caso muy raro), el deseo de lograr un buen rendimiento obliga a dimensionar la tabla ligeramente (digamos un 10%) más grande de lo necesario.

La segunda deficiencia importante de la técnica de almacenamiento por dispersión se vuelve evidente si las llaves no sólo deben insertarse y recuperarse, sino que además han de ser eliminadas. La supresión de entradas en una tabla de transformación de llaves resulta una tarea sumamente difícil a menos que se recurra al encadenamiento (concatenación) directo en un área de desbordamiento independiente. Por tanto, podemos afirmar que las organizaciones de árbol conservan su atractivo, e incluso que han de preferirse, si el volumen de datos se desconoce en gran medida, es variable en extremo y, en ocasiones, hasta llega a disminuir.

EJERCICIOS

- 5.1. Si la cantidad de información asociada a cada llave es bastante grande (en comparación con la llave), esa información no deberá guardarse en la tabla de transformación de llaves. Explique por qué y proponga un esquema para representar ese conjunto de datos.
- 5.2. Examine la propuesta de resolver el problema de agrupación por medio de árboles de desbordamiento en lugar de listas de desbordamiento, esto es, de organizar las llaves que tienen colisiones como estructuras de árbol. De ahí que cada entrada de la tabla de dispersión (hash) deba considerarse como la raíz de un árbol (posiblemente vacío). Compare el rendimiento esperado de este método de transformación de árbol con el de dirección abierta.
- 5.3. Idee un plan que realice las inserciones y eliminaciones en una tabla de transformación de llaves usando los incrementos cuadráticos para la resolución de colisiones. Compare ese plan experimentalmente con la organización directa de árboles binarios, aplicando las secuencias aleatorias de llaves para la inserción y eliminación.
- 5.4. La desventaja primaria de la técnica de tabla de transformación de llaves consiste en que el tamaño de la tabla ha de fijarse en un momento en que se ignoran los números reales de entradas. Suponga que su sistema de cómputo está provisto de un mecanismo de asignación dinámica de memoria, el cual permite obtener almacenamiento en cualquier instante. Por ello, cuando la tabla de transformación de llaves H está llena (o casi llena), se genera una tabla más grande H' y todas las llaves contenidas en H son transferidas a H' , tras lo cual la localización de H puede ser retornada a la administración del almacenamiento. A esto se le llama *retransformación de llaves (rehashing)*. Escriba un programa que realice la retransformación de una tabla H de tamaño n .
- 5.5. Muy a menudo las llaves no son enteros sino secuencias de letras. Son palabras que varían mucho de longitud y, por lo mismo, no es fácil ni económico guardarlas en campos de llaves de tamaño fijo. Escriba un programa que opere con una tabla de transformación y con llaves de longitud variable.

BIBLIOGRAFIA

- 5-1. W.D. Maurer. An Improved Hash Code for Scatter Storage. *Comm. ACM*, 11, No. 1 (1968), 35-38.
- 5-2. R. Morris. Scatter Storage Techniques. *Comm. ACM*, 11, No. 1 (1968), 38-43.
- 5-3. W.W. Peterson. Addressing for Random-access Storage. *IBM J. Res & Dev.*, 1 (1957), 130-46.
- 5-4. G. Schay and W. Spruth. Analysis of a File Addressing Method. *Comm. ACM*, 5, No. 8 (1962), 459-62.

A El conjunto de caracteres ASCII

	0	10	20	30	40	50	60	70
0	nul	dle		0	@	P	'	p
1	soh	dcl	l	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(8	H	X	h	x
9	ht	em)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[k	{
C	ff	fs	,	<	L	\	l	
D	cr	gs	-	=	M]	m	}
E	so	rs	.	>	N	↑	n	~
F	si	us	/	?	O	←	o	del

Caracteres de formato

bs	retroceso
ht	tabulador horizontal
lf	cambio de línea
vt	tabulador vertical
ff	cambio de forma
cr	retorno del carro

Caracteres de separación

fs	separador de archivos
gs	separador de grupos
rs	separador de registros
us	separador de unidades

B Sintaxis de Modula 2

```

1 ident = letra {letra | dígito}.
2 número = entero | real.
3 entero = dígito {dígito} | Dígitoctal {Dígitoctal} ("B" | "C")
4   dígito {DigitoHex} "H".
5 real = dígito {dígito} "." {dígito} [FactordeEscala].
6 FactordeEscala = "E" ["+" | "-"] dígito {dígito}.
7 DigitoHex = dígito | "A" | "B" | "C" | "D" | "E" | "F".
8 dígito = Dígitoctal | "8" | "9".
9 Dígitoctal = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
10 cadena = ""{carácter} ""{carácter}"".
11 identcali = ident {"." ident}.
12 DeclaraciónConstante = ident "=" ExpresiónConst.
13 ExpresiónConst = expresión.
14 DeclaracióndeTipo = ident "=" tipo.
15 tipo = TipoSimple | TipodeArreglo | TipodeRegistro | TipodeConjunto |.
16 TipodeApuntador | TipodeProcedimiento.
17 TipoSimple = identcali | enumeración | TipodeSubintervalo
18 enumeración = "("IdentLista")".
19 IdentLista = ident {"," ident}.
20 TipodeSubintervalo = [identcali] "[" ExpresiónConst ".." Expresión Const "]".
21 TipodeArreglo = ARRAY TipoSimple {"," TipoSimple} OF tipo.
22 TipodeRegistro = RECORD CampoListaSecuencia END.
23 CampoListaSecuencia = CampoLista {"," CampoLista}.
24 CampoLista = [IdentLista ":" tipo |
25   CASE [ident] ":" identcali OF variante {"|" variante}
26   [[ELSE CampoListaSecuencia] END].
27 variante = [ListaEtiquetaCase ":" CampoListaSecuencia].
28 ListaEtiquetaCase = EtiquetasCase {"," EtiquetasCase}.
29 EtiquetasCase = ExpresiónConst [".." ExpresiónConst].
30 TipodeConjunto = SET OF TipoSimple.
31 TipodeApuntador = POINTER TO tipo.
32 TipodeProcedimiento = PROCEDURE [TipodeListaFormal].
33 TipodeListaFormal = "(" [[VAR] TipoFormal
34   {"," [VAR] TipoFormal}] ")" ":" identcali].
35 DeclaraciónVariable = IdentLista ":" tipo.
36 designador = identcali {"." ident "[" ExpLista "]" | "\r").
37 ExpLista = expresión {"," expresión}.
38 expresión = ExpresiónSimple [relación ExpresiónSimple].

```

```

39 relación = "=" | "#" | "<" | "<=" | ">" | ">=" | IN .
40 ExpresiónSimple = ["+" | "-"] término {término OperadordeSuma}.
41 OperadordeSuma = "+" | "-" | OR.
42 término = factor {factor OperadordeMul}.
43 OperadordeMul = "*" | "/" | DIV | REM | MOD | AND.
44 factor = número | cadena | conjunto | designador [ParámetrosReales] |
45   "("expresión")" | factor NOT.
46 conjunto = [identcali] "{" elemento {"," elemento} "}".
47 elemento = ExpresiónConst [".." ExpresiónConst].
48 ParámetrosReales = "(" [ExpLista]"")".
49 Proposición = [asignación | SolicituddeProcedimiento |
50   ProposiciónIf | ProposiciónCase | ProposiciónWhile |
51   ProposiciónRepeat | ProposiciónLoop | ProposiciónFor |
52   ProposiciónWith | EXIT | RETURN [expresión]].
53 asignación = designador ":" expresión.
54 SolicituddeProcedimiento = designador [ParámetrosReales].
55 SecuenciadeProposiciones = proposición {";" proposición}.
56 ProposiciónIf = expresión IF THEN SecuenciadeProposiciones
57   {ELSIF expresión THEN SecuenciadeProposiciones}
58   [ELSE SecuenciadeProposiciones] END.
59 ProposiciónCase = CASE expresión OF caso{" | " caso}
60   [ELSE SecuenciadeProposiciones] END.
61 caso = [ListaEtiquetaCase ":" SecuenciadeProposiciones].
62 ProposiciónWhile = WHILE expresión DO SecuenciadeProposiciones END.
63 ProposiciónRepeat = REPEAT SecuenciadeProposiciones UNTIL expresión.
64 ProposiciónFor = FOR ident ":" expresión TO expresión
65   [BY ExpresiónConst] DO SecuenciadeProposiciones END.
66 ProposiciónLoop = LOOP SecuenciadeProposiciones END.
67 ProposiciónWith = WITH designador DO SecuenciadeProposiciones END.
68 DeclaracióndeProcedimiento = EncabezadodeProcedimiento ";"
69   (ident bloque | FORWARD).
70 EncabezadodeProcedimiento = PROCEDURE ident [ParámetrosFormales].
71 bloque = {declaración} [BEGIN SecuenciadeProposiciones] END.
72 declaración = CONST {DeclaraciónConstante ";"}
73   TYPE {DeclaracióndeTipo ";"}
74   VAR {DeclaraciónVariable ";"}
75   DeclaracióndeProcedimiento ";" | DeclaracióndeMódulo [ ";"].
76 ParámetrosFormales =
77   "(" [SecciónFP {";" SecciónFP}] ")" ":" identcali].
78 SecciónFP = [VAR] IdentLista ":" TipoFormal.
79 TipoFormal = [ARRAY OF] identcali.
80 DeclaracióndeMódulo =
81   MODULE ident [prioridad ";" {importar} [exportar] ident bloque].
82 exportar = EXPORT [QUALIFIED] IdentLista ";".

```

```

83 importar = [FROM ident] IMPORT IdentLista ";;".
84 MódulodeDefinición = DEFINITION MODULE ident ";;"
85 {importar} {definición} END ident ";;".
86 definición = CONST {DeclaraciónConstante ";;"} |
87 TYPE {ident ['=' tipo] ";;"} |
88 VAR {DeclaraciónVariable ";;"} |
89 EncabezadodeProcedimiento ";;".
90 MódulodePrograma = MODULE ident [prioridad] ";;" {impórtar} ident bloque ";;".
91 UnidaddeCompilación = MódulodeDefinición | [IMPLEMENTATION] Módulo dePrograma.

```

ParámetrosReales	54	-48	44
OperadordeSuma	-41	40	
TipodeArreglo	-21	15	
asignación	-53	49	
bloque	90	80	-70 68
caso	-61	59	59
ListaEtiquetaCase	61	-28	27
EtiquetasCase	-29	28	28
ProposiciónCase	-59	50	
carácter	10	10	
UnidaddeCompilación	-91		
DeclaraciónConstante	86	71	-12
ExpresiónConst	81	65	47 47 29 29 20 20 -13 12
declaración	-71	70	
definición	-86	85	
MódulodeDefinición	91	-84	
designador	67	54	53 44 -36
dígito	-8	7	6 6 5 5 5 4 3 3 1
elemento	-47	46	46
enumeración	-18	17	
ExpLista	48	-37	36
exportar	-82	80	
expresión	64	64	63 62 59 57 56 53 52
		45	-38 37 37 13
factor	45	-44	42 42
CampoLista	-24	-23	23
CampoListaSecuencia	27	26	-23 22
ParámetrosFormales	-75	69	
TipoFormal	-78	77	34 33
ListadeTipoFormal	-33	32	
ProposiciónFor	-64	51	
SecciónFP	-77	76	76
Digitohex	-7	4	

ident	90	90	87	85	84	83	80	80	69	68
IdentLista	64	36	25	19	19	14	12	11	11	-1
Proposición	83	82	77	35	24	-19	18			
importar	-56	50								
entero	90	85	-83	80						
letra	-3	2								
ProposiciónLoop	1	\$								
DeclaracióndeMódulo	-66	51								
OperadordeMU1	-79	74								
número	-43	42								
Dígitooctal	44	-2								
TipodeApuntador	-9	8	3	3						
prioridad	-31	16								
SolicituddeProcedimiento	90	-81	80							
DeclaracióndeProcedimiento	-54	49								
EncabezadodeProcedimiento	74	-68								
TipodeProcedimiento	89	-69	68							
MódulodePrograma	-32	16								
identical	91	-90								
real	78	76	46	36	34	25	20	17	-11	
TipodeRegistro	-5	2								
relación	-22	15								
ProposiciónRepeat	-39	38								
FactordeEscala	-63	51								
conjunto	-6	5								
TipodeConjunto	-46	44								
ExpresiónSimple	-30	15								
TipoSimple	-40	38	38							
proposición	30	21	21	-17	15					
SecuenciadeProposiciones	55	55	-49							
	70	67	66	65	63	62	61	60	58	
cadena	57	56	-55							
TipodeSubintervalo	44	-10								
término	-20	17								
tipo	-42	40	40							
DeclaracióndeTipo	87	35	31	24	21	-15	14			
DeclaracióndeVariable	72	-14								
variante	88	73	-35							
ProposiciónWhile	-27	25	25							
ProposiciónWith	-62	50								
	-67	52								

INDICE

Acceso al azar, 32, 39
 Actualización selectiva, 37
 Adelson-Velskii, G. M., 234
 Antepasado, 210
 Apuntador (tipo), 188
 Arbol, 210
 altura de, 210
 balance de, 215, 234
 borrado de, 229, 240, 265
 búsqueda de, 219, 222
 clasificación de, 95
 grado de, 212
 inserción, 222, 235, 261, 272, 276
 nivel de, 210
 página de, 259
 peso de, 247
 profundidad de, 210
 recorrido de, 218
 Arbol-AVL, 234
 Arbol-B, 260
 Arbol-B binario, 271
 Arbol binario, 213
 Arbol-B binario simétrico, 273
 Arbol con vías múltiples, 258
 Arbol de búsqueda con prioridad, 278
 Arbol de búsqueda de prioridad con radical, 280
 Arbol de Fibonacci, 235
 Arbol de prioridad, 278
 Arbol óptimo, 246
 Arbol 2-3, 271
 Area de desbordamiento, 290
 ASCII, 29
 Asignación dinámica, 187

 Bayer, R., 260, 271
 BITSET, 43
 BOOLEAN (booleano), 28
 Boyer, R. S., 73
 Búsqueda
 binaria, 63
 de árbol, 278

 CHAR (de carácter), 29

de cadena, 66
 de tabla, 65
 lineal, 62
 Búsqueda binaria, 63
 Búsqueda de cadena, 66

 Cadena, 65
 Campo (identificador), 37
 Campo objetivo, 40
 Carácter de control, 29
 CARDINAL, 28
 Cardinalidad, 23
 Centinela, 63, 197
 Centroide, 257
 Clasificación en polifase, 126
 Clasificación estable, 83
 Clasificación externa, 82
 Clasificación interna, 82
 Clasificación por burbujas, 89
 Clasificación por inserción, 84
 Clasificación por inserción binaria, 86
 Clasificación por mezcla, 110
 Clasificación por mezcla en cascada, 142
 Clasificación por montones, 96
 Clasificación por selección, 87
 Clasificación por vibración, 91
 Clasificación rápida, 102
 Clasificación topológica, 202
 Colisión o choque, 288
 Compensación (o desplazamiento), 47
 Concordancia, 196
 Concurrencia, 56
 Conjunción, 28
 Corrida, 115
 Corrida ficticia, 129
 Corrutina, 138
 Curva de Hilbert, 151
 Curva de Sierpinski, 153

Declaración, 22
 Desbordamiento, 27
 Descendiente, 210
 Dijkstra, E. W., 9
 Dirección 45
 Direccionamiento abierto, 290
 Discriminador de registros, 40
 Distribución de corridas, 129
 Disyunción, 28
 DIV, 27
 División de páginas, 261

 Empaquetamiento, 46
 Entrada, 59
 Enumeración, 25
 Eslabonamiento directo, 290
 Estructura de conjunto, 43
 Etapa, 111
 Euler, L., 27, 89
 Exclusión mutua, 57

 Factor de carga, 296
 Factorial, 148
 Fase, 111
 Floyd, R. W., 97
 Formateo, 59
 Función armónica, 232
 Función de cálculo de dirección, 289
 Fusión de páginas, 265

 Gauss, C. F., 164
 Gilstad, R. L., 126
 Gotlieb, C. C., 257
 Grupo, 102

 Hilbert, D., 151
 Hoare, C. A. R., 10, 100, 105
 Hu, T. C., 257

 Índice bibliográfico cruzado, 196, 225
 INTEGER, 27

 Knuth, D. E., 68, 83

 Landis, E. M., 234
 Lista, 193
 borrado, 194
 búsqueda, 196
 generación, 193
 inserción, 194
 recorrido, 195
 Lista ordenada, 198
 Longitud de trayectoria, 212, 231
 Longitud de trayectoria con valor asignado, 254
 Longitud de trayectoria externa, 212
 Longitud de trayectoria interna, 212

 Manejo por buffer, 54
 Matriz, 33
 McCreight, E. M., 260, 279
 McVitie, D. G., 174
 Mediana, 105
 Mezcla balanceada, 111, 121
 Mezcla directa, 110
 Mezcla natural, 115
 MOD, 27
 Modula-2, 15, 21
 Módulo de definición, 52
 Módulo de implantación, 52
 Monitor, 57
 Montón, 96, 278
 Moore, J. S., 73
 Morris, J. H., 68

 NIL, 189
 Notación infija, 218
 Notación postfija, 218
 Notación prefija, 218
 Número de Fibonacci, 128
 Número de Leonardo, 235

 Operación de conjunto, 43
 Orden parcial, 202

 Ramificación y límite, 179
 Rastreo inverso, 158

Recálculo de dirección (retransformación de llaves), 298
 Recorrido en orden, 218
 Recorrido postorden, 218
 Recorrido preorden, 218
 Recursión, 102, 145
 Refinamiento por pasos, 12, 112
 Relleno, 45
 REM, 27
 Reorganización de listas, 200
 Salida, 59
 Shell, D. L., 93
 Sincronización, 56
 Sondeo cuadrático, 291
 Sondeo lineal, 290
 Suborden (tipo), 31
 Sucesión o secuencia, 50
 Tamaño de palabra, 45
 Tipo de base, 23, 32
 Tipo (de datos), 20
 Tipo de datos recursivo, 184
 Tipo de índice, 32
 Tipos estándar, 27
 Transformación de llaves, 287
 Tucker, A. C., 257
 Variable suscrita, 32
 Variante de registro, 40
 Walker, W. A., 257
 Wilson, L. B., 174
 Williams, J., 96

Índice de programas:

Árbol óptimo, 250
 Borrado de árbol, 229
 Borrado de árbol balanceado, 241

Búsqueda binaria, 63
 Búsqueda Boyer-Moore, 74
 Búsqueda directa de cadena, 68
 Búsqueda e inserción de árbol, 222
 Búsqueda e inserción de árbol balanceado, 234
 Búsqueda e inserción de árbol-B binario simétrico, 275
 Búsqueda e inserción directa de listas, 147
 Búsqueda en tabla, 65
 Búsqueda, inserción y borrado de árbol-B, 265
 Búsqueda Knuth-Morris-Pratt, 71
 Cálculo de la mediana, 107
 Cernido, 98
 Clasificación de Shell, 94
 Clasificación en polifase, 133
 Clasificación por burbuja, 89
 Clasificación por fusión balanceada, 124
 Clasificación por inserción binaria, 86
 Clasificación por inserción directa, 86
 Clasificación por mezcla directa, 110
 Clasificación por mezcla natural, 121
 Clasificación por montones, 99
 Clasificación por selección directa, 88
 Clasificación por vibración, 91
 Clasificación rápida (no recursiva), 103
 Clasificación rápida (recursiva), 102
 Clasificación topológica, 207
 Construcción de árbol, 216
 Curvas de Hilbert, 152
 Curvas de Sierpinski, 154
 Distribución de corridas, 137
 Entrada/Salida, 60
 Enumeración de árbol de rastreo de prioridad, 281
 Generador de referencias cruzado (árbol), 226
 Generador de referencias cruzado (tabla combinada), 291

Inserción de árbol de rastreo de prioridad, 280
 Matrimonios estables, 172
 Partición, 100
 Potencias de 2, 39
 Recorrido de árbol, 219
 Todas las reinas, 167

Recorrido del caballo (del ajedrez), 160
 Reinas, 164
 Selección, 178
 Separador (buffer), 57, 58
 Sistema de archivo, 52
 Sucesiones o secuencias, 117