

MICHAEL J. FOLK

Oklahoma State University

BILL ZOELLICK

Versión en español de

Luis F. Castro Careaga

Universidad Autónoma Metropolitana,

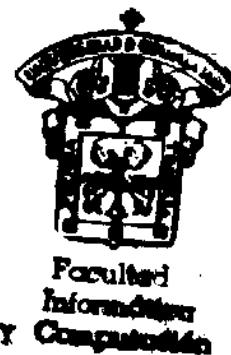
Unidad Iztapalapa, México

Con la colaboración de

Guillermo Levine Gutiérrez

Universidad de Guadalajara

México



ESTRUCTURAS DE ARCHIVOS

UN CONJUNTO DE HERRAMIENTAS
CONCEPTUALES

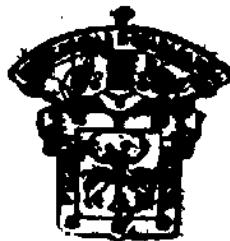


ADDISON-WESLEY IBEROAMERICANA

Argentina • Brasil • Chile • Colombia • Ecuador
España • Estados Unidos • México
Perú • Puerto Rico • Venezuela

Versión en español de la obra titulada *File Structures. A Conceptual Toolkit*, de Michael J. Folk y Bill Zoellick, publicada originalmente en inglés por Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, E.U.A. © por Addison-Wesley Publishing Company, Inc.

Esta edición en español es la única autorizada.



Facultad
Informática
y Computación



UNIVERSIDAD DE GUADALAJARA
UNIDAD DE BIBLIOTECAS

CUCEI

23554

No. ADQUISICION
CLASIFICACION
FACTURA
FECHA
V.

© 1992 por Addison-Wesley Iberoamericana, S.A.
Wilmington, Delaware, E.U.A.

Impreso en E.U.A. Printed in U.S.A.

Bib. 404.

ISBN 0-201-62923-2

1 2 3 4 5 6 7 8 9 10 -AL- 96 95 94 93 92

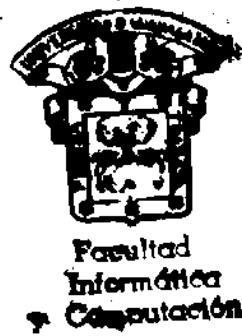
004.32
Fol



Facultad
Informática
y Computación

ESTRUCTURAS DE ARCHIVOS

UN CONJUNTO DE HERRAMIENTAS CONCEPTUALES



Facultad
Informática
y Computación



**A LA MEMORIA DE
MARTHA LIVERIGHT FOLK**



Diseñar estructuras de archivos rápidas y flexibles es una de las actividades más interesantes y satisfactorias en las ciencias de la computación. Este libro instruye al lector sobre los elementos que intervienen en este proceso de diseño. Como en cualquier otro libro sobre el tema, se hace énfasis en los *conceptos* y *restricciones* del diseño.

Consideramos que los *conceptos* de diseño son *herramientas conceptuales*, y un buen número de ellas las deberían poder usar con facilidad todos los diseñadores de estructuras de archivos. Por ejemplo, es necesario manejar bien una amplia gama de opciones para organizar un archivo en registros. Asimismo, hay que contar con un *conjunto de herramientas* de métodos de *extracción* de estos registros. Este conjunto de herramientas debe incluir, por ejemplo, acceso por dispersión (*hashing*), índices simples, índices paginados, como árboles B, y variaciones sobre índices paginados, como árboles B⁺ y árboles B*. El diseñador debe saber no sólo los nombres de estas herramientas y cómo se usan sino también *por qué* y *cuándo* usarlas.

El porqué y cuándo están asociados con el aspecto de las *restricciones*. El diseñador de estructuras de archivos debe comprender el medio de diseño, lo cual significa conocer ampliamente las ventajas y debilidades de los dispositivos de almacenamiento de archivos. El diseñador debe poder relacionar estas restricciones del medio con las posibilidades

PREFACIO

que ofrecen las herramientas y, por supuesto, con los *objetivos* de diseño del sistema que se está construyendo. El objetivo de este libro es ayudar al lector a desarrollar dichos *juicios de diseño*.

Esta obra está estructurada como libro de texto para cursos de licenciatura, pero también está dirigido a aquellos profesionales de las ciencias de la computación que quieran aprender más acerca del diseño de estructuras de archivos. Se ha intentado hacer *legible* este libro; no es un manual de consulta que deba explicarse a través de un conjunto de lecturas complementarias, sino un libro que un estudiante o un programador profesional puede leer de principio a fin.

===== EL USO DE C Y PASCAL

El pseudocódigo es una herramienta de instrucción maravillosa que se centra en los aspectos esenciales de un algoritmo, dejando aparte todos los detalles que deben estar presentes en un programa real. En este libro se usa ampliamente el pseudocódigo, en particular en los últimos capítulos, donde los procedimientos son complejos y, por tanto, donde es importante evitar las distracciones inherentes en el código real compilable. Sin embargo, en algunas ocasiones es necesario fijarse en los detalles, en particular cuando se estudian los primeros capítulos del libro, donde se exploran temas tales como las diversas formas de expresar el concepto de *registro* en una estructura de archivo. Por tal razón es necesario ser capaz de presentar programas reales, además del pseudocódigo, y por lo tanto, elegir un lenguaje de programación para el texto.

En realidad, se usan dos lenguajes: C y Pascal. C es un lenguaje excelente para la realización de estructuras de archivos innovadoras y nuevas, y el número de programadores profesionales que lo usa se está incrementando con rapidez. Por otro lado, Pascal es un lenguaje más familiar para muchos estudiantes universitarios. Entre las motivaciones iniciales que se tuvieron para usar ambos lenguajes se encuentra el hacer un libro de fácil acceso para un número grande de lectores. Sin embargo, a medida que se desarrollaron los programas de ejemplo en ambos lenguajes surgió un segundo beneficio: dado que el enfoque más "natural" para abordar un problema en C difiere algunas veces de la solución "natural" en Pascal, el uso de ambos lenguajes ayuda a recalcar el hecho de que el diseño de estructuras de archivos es una cuestión de *juicio*. La solución "correcta" a un problema de estructuras de archivos depende de muchos factores, entre ellos el lenguaje que se emplea para realizar el diseño.

===== EL USO DE ESTE LIBRO EN LA ENSEÑANZA DE UN CURSO

Durante los dos años pasados se han usado las versiones preliminares de este libro como texto principal en el curso sobre estructuras de archivos, en la Oklahoma State University. Este es un curso de un semestre donde se estudia el material planteado como Curso CS-5 del *Curriculum de 1978 de la ACM*, el cual proporciona un fundamento para el estudio del diseño de bases de datos. La matrícula del curso se compone principalmente de estudiantes del penúltimo y último años de licenciatura; gran porcentaje de estudiantes que entran a la maestría toma también el curso. Este incluye tres lecciones por semana y una sesión de laboratorio de hora y media.

Puesto que el libro es bastante legible, se ha considerado razonable esperar que los estudiantes lean el libro completo en el transcurso del semestre. El texto cubre lo básico; las lecciones en clase amplían y complementan la base del material que se presenta en el texto. Esto proporciona libertad al instructor para investigar cuestiones y posibilidades más complejas. Se ha visto que, dadas las restricciones de tiempo que impone un solo semestre, no se puede profundizar a la vez en dispersión y en árboles B. Aunque siempre se abordan ambos temas, en cualquier semestre determinado, la asignación del problema se concentra ya sea en dispersión o en árboles B.

Una advertencia: con facilidad se puede desperdiciar demasiado tiempo en los aspectos de bajo nivel que se presentan en los primeros siete capítulos. Se debe avanzar rápidamente al estudiar este material. El número relativamente grande de páginas dedicadas a estos temas no refleja el porcentaje de tiempo del curso que debe ocuparse en esto. Por el contrario, la intención es abarcálos totalmente en el texto, de modo que el instructor pueda simplemente asignar la lectura de estos capítulos como estudio en casa, ahorrando valioso tiempo de exposición en clase para temas más importantes.

Se ha considerado importante involucrar a los estudiantes en la escritura de programas de procesamiento de archivos desde el principio del semestre. Se inicia con una tarea sencilla de lectura y escritura de un archivo que se realiza después de la primera semana de clases. La inclusión en el texto de programas de ejemplo tanto en C como en Pascal facilita este estilo de trabajo práctico. Se recomienda que cuando los estudiantes se encuentren en el capítulo de árboles B, dispongan ya de programas escritos para tener acceso a un conjunto de datos a través de una estructura de índices simple. Como los estudiantes ya tendrán experiencia de primera mano con los aspectos fundamentales de organización, será posible que los instructores se aboquen a los aspectos conceptuales involucrados en el diseño de árboles B.

Por último, se sugiere que los instructores se apeguen estrechamente a la secuencia temática del libro, en especial en los primeros siete capítulos. Ya se ha señalado que los autores escribieron el libro de tal forma que pueda leerse de principio a fin. No es un libro de consulta. Por tal razón se *desarrollan* ideas a medida que se avanza, capítulo por capítulo. Brincar de un tema a otro a lo largo del libro dificultaría a los estudiantes el seguimiento de este desarrollo. La única excepción notable de esta regla involucra los capítulos de árboles B y el capítulo de dispersión. El texto está escrito para adecuarse a aquellos instructores que quieran abordar la dispersión antes de examinar los árboles B y los árboles B⁺.

ORGANIZACION DEL LIBRO

Se muestra un resumen esquemático de la secuencia de los temas que se analizan en el texto:

INTRODUCCION

Capítulo 1: Cómo difieren las estructuras de archivos de las estructuras de datos contenidas en la memoria electrónica. Los aspectos clave del diseño de estructuras de archivos.

FUNDAMENTOS

Capítulo 2: Procesos fundamentales en archivos: creación, apertura, lectura, escritura y localización.

Capítulo 3: Dispositivos de almacenamiento secundario: restricciones físicas y posibilidades que conforman el diseño de estructuras de archivos.

Capítulo 4: Conceptos fundamentales: campos, registros, acceso secuencial y directo, y búsqueda secuencial.

Capítulo 5: Mantenimiento de archivos y eliminación de registros: reutilización del espacio en un archivo, fragmentación del almacenamiento y administración del almacenamiento.

ESTRATEGIAS SIMPLES DE EXTRACCION

- Capítulo 6: Clasificación y búsqueda: clasificación en memoria RAM y acceso a través de búsqueda binaria.
- Capítulo 7: Indización: estructuras simples de índices, extracción por llaves secundarias, y estructuras de listas invertidas.

PROCESAMIENTO SECUENCIAL CONCURRENTE

- Capítulo 8: Procesamiento secuencial concurrente y clasificación de archivos grandes: un modelo general para la intercalación y correspondencia en dos formas, intercalación en K formas, clasificación de archivos grandes en disco, selección por reemplazo y clasificación en cinta.

METODOS DE ACCESO POR ARBOLES

- Capítulo 9: Los árboles B y otros tipos de organizaciones de archivos estructuradas con árboles: la historia del desarrollo de los árboles B, otras estructuras de árboles paginadas y estructuras de árboles balanceadas, operaciones fundamentales de los árboles B, árboles B virtuales, árboles B* y otras variantes.
- Capítulo 10: Los árboles B* y el acceso secuencial indizado a archivos: una introducción al acceso secuencial indizado, árboles B* con prefijos simples, carga de árboles y una comparación de los métodos de estructuración con árboles.

DISPERSION (HASHING)

- Capítulo 11. Dispersión: conceptos fundamentales, métodos sencillos para resolver y reducir colisiones, herramientas para analizar y predecir el desempeño, deterioro de archivos, estructuras avanzadas de dispersión y patrones de acceso a registros.

RECONOCIMIENTOS

Es imposible agradecer a todas las personas que nos ayudaron a preparar este libro. Una importante expresión de reconocimiento debe dirigirse a nuestro colega Jim VanDoren, cuyos puntos de vista sobre las estructuras de archivos influyeron decisivamente en nuestro enfoque original sobre el tema. Quisiéramos manifestar especial agradecimiento a Gail Smith, por su ayuda en la realización de muchos de los programas que aparecen al final de los capítulos a lo largo del texto. Quisiéramos agradecer a TMS, Inc., por proporcionarnos las cargas de trabajo flexibles que se requieren para permitir la escritura de un libro.

Los más de 200 estudiantes que usaron los manuscritos que evolucionaban durante los pasados dos años hicieron importantes contribuciones con el cuestionamiento del material, con el señalamiento de los errores que se encontraron en el texto y en los ejercicios y, en general, con el apoyo a nuestros esfuerzos. Gracias, especialmente, a los estudiantes Huey Liu y Marilyn Aiken, y a los colegas Art Crotzer y Don Fisher, por sus útiles comentarios y sugerencias.

Mark Dalton, nuestro editor en Addison-Wesley, brindó un apoyo incondicional desde los primeros capítulos tentativos hasta el final de la publicación. Agradecemos a Sherry Berg y a Sarah Meyer por afinar el formato del libro y pulir la prosa. Gracias también a los revisores de Addison-Wesley, quienes proporcionaron comentarios constructivos y sugerencias durante el desarrollo del libro: Laurian M. Chirica, del California Polytechnic State University; Charles W. Reynolds, de James Madison University; Billy G. Claybrook, de Wang Institute; Henry A. Etlinger, de Rochester Institute of Technology; Paul W. Ross de Millersville University; Sharon Salveter, de Boston University; Ronald L. Lancaster, de Bowling Green State University, y Karen A. Lemone, de Worcester Polytech Institute.

También debemos destacar la contribución de dos restaurantes de Stillwater, el *Hideaway* y el *Mom's Place*, por alimentar este proyecto y proporcionar el sitio de incontables reuniones.

Finalmente, nuestro agradecimiento sincero a aquellos que soportaron días y fines de semana reducidos durante dos años, y devolvieron el favor con estímulo y apoyo: Ruthann, Rachel, Martha, Joshua y Peter.

Stillwater, Oklahoma

M.F.
B.Z.

1 INTRODUCCION: ARCHIVOS Y ESTRUCTURAS DE ARCHIVOS

- 1.1 Almacenamiento primario y secundario 2**
- 1.2 Nada es gratis 3**
- 1.3 Archivos 5**
- 1.4 Estructuras de archivos *versus* estructuras de datos 6**
- 1.5 Un conjunto de herramientas conceptuales 7**
- Resumen 8**
- Términos clave 9**

2 OPERACIONES FUNDAMENTALES PARA EL PROCESAMIENTO DE ARCHIVOS

- 2.1 Archivos físicos y archivos lógicos 12**
- 2.2 Apertura y creación de archivos 13**
- 2.3 Cierre de archivos 17**
- 2.4 Lectura y escritura 18**
- 2.5 Detección del fin del archivo 22**
- 2.6 Localización 22**
 - 2.6.1 Localización en C 23**
 - 2.6.2 Localización en Pascal 25**

INDICE GENERAL

- 2.7 Caracteres inesperados en archivos 27**
- Resumen 28
- Términos clave 29
- Ejercicios 31
- Lecturas adicionales 33
- El programa *agrega* en Pascal 34

3

DISPOSITIVOS DE ALMACENAMIENTO SECUNDARIO Y SOFTWARE DE SISTEMAS: CONSIDERACIONES DE DESEMPEÑO

3.1 Discos 39

- 3.1.1 La organización de los discos 39
- 3.1.2 Estimación de las capacidades y necesidades de espacio 42
- 3.1.3 Organización por sectores 43
- 3.1.4 Organización por bloques 48
- 3.1.5 Sobrecarga por datos usados para control 50
- 3.1.6 El costo de un acceso a disco 52

3.2 Cinta magnética 56

- 3.2.1 Organización de datos en cintas 56
- 3.2.2 Estimación de requerimientos de longitud de cinta 58
- 3.2.3 Estimación de los tiempos de transmisión de datos 60
- 3.2.4 Aplicaciones de las cintas 62

3.3 Otros tipos de almacenamiento 62

3.4 El almacenamiento como una jerarquía 65

3.5 El viaje de un byte 65

- 3.5.1 El administrador de archivos 67
- 3.5.2 El buffer de E/S 69
- 3.5.3 El byte sale de la memoria RAM: el procesador de E/S 71
- 3.5.4 El byte llega al disco: el controlador del disco 73

3.6 Manejo de buffers 74

Resumen 77

Términos clave 79

Ejercicios 83

Lecturas adicionales 87

4**CONCEPTOS FUNDAMENTALES DE
ESTRUCTURAS DE ARCHIVOS**

- 4.1 Un archivo como secuencia de bytes 90**
- 4.2 Estructuras de campos 92**
 - 4.2.1 Método 1: Fijar la longitud de los campos 93**
 - 4.2.2 Método 2: Comenzar cada campo con un indicador de longitud 94**
 - 4.2.3 Método 3: Separar los campos con delimitadores 94**
- 4.3 Lectura de una secuencia de campos 95**
- 4.4 Estructuras de registros 96**
 - 4.4.1 Método 1: Hacer los registros de una longitud predecible 97**
 - 4.4.2 Método 2: Comenzar cada registro con un indicador de longitud 99**
 - 4.4.3 Método 3: Usar un segundo archivo para mantener información sobre las direcciones 99**
 - 4.4.4 Método 4: Colocar un delimitador al final de cada registro 100**
- 4.5 Una estructura de registros que usa un indicador de longitud 100**
- 4.6 Mezcla de números y caracteres: uso de un vaciado hexadecimal 102**
- 4.7 Lectura de registros de longitud variable de un archivo 106**
- 4.8 Extracción de registros por llave: formas canónicas para llaves 106**
- 4.9 Una búsqueda secuencial 109**
- 4.10 Evaluación del desempeño de la búsqueda secuencial 109**
- 4.11 Mejora del desempeño de la búsqueda secuencial: manejo de registros en bloques 111**
- 4.12 Acceso directo 113**
- 4.13 Elección de una estructura y una longitud de registro 115**
- 4.14 Registros de encabezado 119**
- 4.15 Acceso y organización de archivos 119**
- Resumen 121**
- Términos clave 123**
- Ejercicios 125**
- Lecturas adicionales 130**
- Programas en C 131**
- Programas en Pascal 146**

5**MANTENIMIENTO DE ARCHIVOS Y
ELIMINACION DE REGISTROS**

- 5.1 Mantenimiento de archivos 164**
- 5.2 Compactación del almacenamiento 165**
- 5.3 Panorama de la eliminación de registros de longitud fija 167**
- 5.4 Realización de la eliminación de registros de longitud fija 171**
- 5.5 Eliminación de registros de longitud variable 173**
- 5.6 Fragmentación del almacenamiento 177**
- 5.7 Estrategias de colocación 181**
- Resumen 183**
- Términos clave 185**
- Ejercicios 187**
- Lecturas adicionales 189**

6**LOCALIZACION RAPIDA DE DATOS EN UN
ARCHIVO: INTRODUCCION A LA CLASIFICACION
Y A LA BUSQUEDA BINARIA**

- 6.1 Localización de datos en los archivos ya desarrollados 193**
- 6.2 Búsqueda por conjetura: búsqueda binaria 193**
- 6.3 Clasificación de un archivo de disco en memoria RAM 196**
- 6.4 Algoritmo para la clasificación en memoria RAM 201**
- 6.5 La función de clasificación: `clasif_shell()` 202**
- 6.6 Limitaciones de la búsqueda binaria y de la clasificación en memoria RAM 203**
- 6.7 Clasificación por llave 206**
 - 6.7.1 Descripción del método 206**
 - 6.7.2 Limitaciones del método de clasificación por llave 208**
 - 6.7.3 Otra solución: ¿Por qué molestarse en escribir de nuevo el archivo? 209**
- 6.8 Registros fijos 211**
- Resumen 212**
- Términos clave 214**
- Ejercicios 215**
- Lecturas adicionales 217**

- Programas en C** 218
Programas en Pascal 222

7**INDIZACION**

- 7.1 ¿Qué es un índice?** 228
7.2 Indice simple de un archivo con entradas secuenciales 229
7.3 Operaciones básicas en un archivo indizado con entradas secuenciales 233
7.4 Indices demasiado grandes para almacenarse en memoria 237
7.5 Indización para proporcionar acceso mediante varias llaves 239
7.6 Extracción de información mediante combinaciones de llaves secundarias 244
7.7 Mejora de la estructura secundaria de índices: listas invertidas 246
 7.7.1 Primer intento de solución 247
 7.7.2 Una mejor solución: ligar la lista de referencias 249
7.8 Indices selectivos 253
7.9 Enlace (*Binding*) 253
Resumen 255
Términos clave 257
Ejercicios 258
Lecturas adicionales 262

8**PROCESAMIENTO SECUENCIAL COORDINADO Y CLASIFICACION DE ARCHIVOS GRANDES**

- 8.1 Un modelo para la realización de procesos secuenciales coordinados** 265
 8.1.1 Correspondencia de nombres en dos listas 265
 8.1.2 Intercalación de dos listas 270
 8.1.3 Resumen del modelo de procesamiento secuencial coordinado 272
8.2 Aplicación del modelo a un programa de libro mayor 275
 8.2.1 El problema 275
 8.2.2 Aplicación del modelo al programa de libro mayor 278

- 8.3 Extensión del modelo para incluir la intercalación múltiple 284**
- 8.4 La intercalación como forma de clasificación de archivos grandes en disco 287**
 - 8.4.1 Patrones de intercalación de varios pasos 291**
 - 8.4.2 Incremento en las longitudes de las porciones mediante el uso de selección por reemplazo 294**
 - 8.4.3 Longitud promedio de las porciones para la selección por reemplazo 297**
 - 8.4.4 Costo del uso de la selección por reemplazo 299**
 - 8.4.5 Configuraciones de disco 302**
 - 8.4.6 Resumen de la clasificación en disco 302**
- 8.5 Clasificación de archivos en cinta magnética 303**
 - 8.5.1 Intercalación balanceada 304**
 - 8.5.2 Intercalación en varias fases 306**
- 8.6 Paquetes de clasificación e intercalación 309**
 - Resumen 309**
 - Términos clave 312**
 - Ejercicios 315**
 - Lecturas adicionales 318**

9

ARBOLES B Y OTRAS ORGANIZACIONES DE ARCHIVOS ESTRUCTURADAS EN FORMA DE ARBOL

- 9.1 Introducción. La invención de los árboles B 322**
- 9.2 Planteamiento del problema 325**
- 9.3 Arboles binarios de búsqueda como solución 325**
- 9.4 Arboles AVL 329**
- 9.5 Arboles binarios paginados 332**
- 9.6 El problema con la construcción descendente de los árboles paginados 335**
- 9.7 Arboles B: construcción ascendente 337**
- 9.8 División y promoción 337**
- 9.9 Algoritmos para la búsqueda e inserción en árboles B 342**
- 9.10 Nomenclatura de árboles B 353**
- 9.11 Definición formal de las propiedades de los árboles B 354**
- 9.12 Profundidad de la búsqueda en el peor caso 355**
- 9.13 Eliminación, redistribución y concatenación 357**

- 9.13.1 Redistribución 361
- 9.14 Redistribución durante la inserción: una forma de mejorar la utilización del almacenamiento 362**
- 9.15 Árboles B* 363**
- 9.16 Manejo de páginas en buffers: árboles B virtuales 365**
- 9.16.1 Reemplazo LRU 366
- 9.16.2 Reemplazo según la altura de la página
- 9.16.3 Importancia de los árboles B virtuales 369
- 9.17 Colocación de la información asociada con la llave 370**
- 9.18 Registros y llaves de longitud variable 371**
- Resumen 373**
- Términos clave 375**
- Ejercicios 377**
- Lecturas adicionales 380**
- Programa en C para insertar llaves en un árbol B 382**
- Programa en Pascal para insertar llaves en un árbol B 390**

10

LA FAMILIA DE LOS ARBOLES B⁺ Y EL ACCESO A LOS ARCHIVOS SECUENCIALES INDIZADOS

- 10.1 Acceso secuencial indizado 398**
- 10.2 Mantenimiento de un conjunto de secuencias 399**
- 10.2.1 Uso de bloques 400
- 10.2.2 Elección del tamaño del bloque 403
- 10.3 Adición de un índice simple al conjunto de secuencias 404**
- 10.4 El contenido del índice: separadores en lugar de llaves 405**
- 10.5 Árboles B⁺ de prefijos simples 409**
- 10.6 Mantenimiento de árboles B⁺ de prefijos simples 410**
- 10.6.1 Cambios localizados en bloques individuales del conjunto de secuencias 410
- 10.6.2 Cambios que involucran varios bloques en el conjunto de secuencias 412
- 10.7 Tamaño de bloque del conjunto índice 415**
- 10.8 Estructura interna de los bloques del conjunto índice: árbol B de orden variable 416**
- 10.9 Carga de un árbol B⁺ de prefijos simples 420**
- 10.10 Árboles B⁺ 424**

10.11 Los árboles B, B⁺ y B^{*} de prefijos simples en perspectiva 426**Resumen 430****Términos clave 432****Ejercicios 434****Lecturas adicionales 439****11****DISPERSION (*HASHING*)****11.1 Introducción 442****11.1.1 ¿Qué es la dispersión? 444****11.1.2 Colisiones 445****11.2 Un algoritmo simple de dispersión 447****11.3 Funciones de dispersión y distribuciones de registros 451****11.3.1 Distribución de registros entre direcciones 451****11.3.2 Algunos otros métodos de dispersión 453****11.3.3 Predicción de la distribución de los registros 454****11.3.4 Predicción de las colisiones en un archivo lleno 459****11.4 ¿Cuánta memoria adicional debe usarse? 460****11.4.1 Densidad de empaquetamiento 460****11.4.2 Predicción de colisiones para diferentes densidades de empaquetamiento 461****11.5 Resolución de colisiones mediante saturación progresiva' 465****11.5.1 Funcionamiento de la saturación progresiva 465****11.5.2 Longitud de búsqueda 467****11.6 Almacenamiento de más de un registro por dirección: compartimientos 470****11.6.1 Efectos de los compartimientos en el desempeño 471****11.6.2 Aspectos de la realización 476****11.7 La operación de eliminación 479****11.7.1 Marcas de inutilización para eliminaciones 480****11.7.2 Implicaciones de las marcas de inutilización para las inserciones 481****11.7.3 Efectos de las eliminaciones y adiciones en el desempeño 482****11.8 Otras técnicas de resolución de colisiones 483****11.8.1 Dispersión doble 483****11.8.2 Saturación progresiva encadenada 484****11.8.3 Encadenamiento con un área de saturación separada 486****11.8.4 Tablas de dispersión: reconsideración de la indización 488**

11.8.5 Dispersión extensible 488
11.9 Patrones de acceso a los registros 490
Resumen 491
Términos clave 495
Ejercicios 498
Lecturas adicionales 504

Apéndice A Tabla de códigos de caracteres ASCII 508

**Apéndice B Funciones de cadenas en Pascal:
herramientas.prc 509**

Apéndice C Introducción a C 514

Apéndice D Comparación de unidades de disco 570

Bibliografía 573

Vocabulario técnico bilingüe* 587

Indice de materias 579

1

OBJETIVOS

Enumerar las razones que justifican el uso de dispositivos de almacenamiento secundario.

Apreciar el alto costo del uso del almacenamiento secundario.

Definir el término *archivo*.

Describir la diferencia entre el estudio de *estructuras de archivos* y el estudio más general de *estructuras de datos*, y mostrar que para el diseño de sistemas de almacenamiento secundario eficientes se requiere conocer los fundamentos de las estructuras de archivos.

Introducir la noción de un *conjunto de herramientas conceptuales*.

INTRODUCCION: ARCHIVOS Y ESTRUCTURAS DE ARCHIVOS

PLAN GENERAL DEL CAPITULO

- 1.1 Almacenamientos primario y secundario
- 1.2 Nada es gratis
- 1.3 Archivos
- 1.4 Estructuras de archivos *versus* estructuras de datos
- 1.5 Un conjunto de herramientas conceptuales

1.1

ALMACENAMIENTOS PRIMARIO Y SECUNDARIO

En 1986, 64 kilobytes (64K) de memoria de acceso aleatorio (RAM) moderadamente rápida para computador costaba alrededor de diez dólares. Estos 64K de RAM, que son el *almacenamiento primario* del computador, tienen capacidad para almacenar alrededor de 25 páginas de texto mecanografiado; el costo de almacenamiento por página mecanografiada de un texto tal como el que se lee ahora es de alrededor de 40 centavos de dólar.

Al llegar al final de este libro se tendrán alrededor de 700 páginas de manuscrito, de modo que hay, al menos, tres buenas razones para no almacenar el libro completo en la memoria electrónica interna del computador:

- Existe un límite en la cantidad de memoria RAM accesible en un computador determinado. El que se usa para escribir este libro sólo puede direccionar a 640K de RAM, es decir, sólo alrededor de una tercera parte de lo que se necesita para almacenar todo el libro.
- Aunque existen computadores grandes que pueden hacer referencia a mayores cantidades de memoria, la memoria RAM sigue siendo un dispositivo relativamente caro. La memoria RAM que está siendo empleada para almacenar este manuscrito no puede usarse a la vez para otras tareas, tales como la compilación de programas.
- Es conveniente poder apagar el computador. Incluso si no se apaga, las fallas de energía pueden hacerlo, y una vez que el

computador se apaga, aunque sea durante unos cuantos segundos, desaparece todo lo que estaba almacenado en memoria RAM, y esto no es precisamente una ventaja para nuestro manuscrito.

El término *almacenamiento secundario* se refiere a los medios de almacenamiento que están fuera del almacenamiento primario en memoria RAM dentro del computador. Las cintas magnéticas, los paquetes de discos, los discos flexibles y los discos de almacenamiento óptico son ejemplos de medios de almacenamiento secundario. Se puede almacenar más información en estos medios de la que normalmente puede almacenarse en RAM. Además, son más económicos y no requieren el suministro continuo de energía para conservar la información almacenada. Por ejemplo, se usó un disco fijo para almacenar las páginas de este manuscrito en un computador personal. El disco fijo retiene alrededor de 20 000K, o 7800 páginas de manuscrito mecanografiado, y el costo de esta clase de discos es de alrededor de 600 dólares. Por lo tanto, el empleo de un disco fijo en lugar de RAM hace descender el costo de almacenamiento por página de 40 a 8 centavos de dólar. Esto representa una mejora considerable, casi de un orden de magnitud.

También se podría haber usado discos flexibles para almacenar el manuscrito. Los que se utilizan en nuestro computador pueden retener alrededor de 360K, o 150 páginas del manuscrito. Puesto que un disco flexible cuesta alrededor de 1.50 dólares, el costo del almacenamiento descendería hasta menos de un centavo por página. Esto es una mejora casi de otro orden de magnitud en cuanto a costos.

Se podría almacenar y distribuir el manuscrito terminado en discos láser digitales CD-ROM de 5.25 pulgadas. Cada disco puede retener 540 000K, lo cual significa que puede almacenar más de 210 000 páginas de manuscrito, es decir, este libro y otros 300. Si los discos se produjeran en gran número, el costo de un disco individual que contuviera toda esta información sería menor de 20 dólares. Así, el costo por página se reduciría a una centésima de centavo de dólar: un gran progreso a partir de los 40 centavos de costo por página al usar memoria RAM.

1.2

NADA ES GRATIS

No se logran tales economías sin dar algo a cambio. El acceso a la información del almacenamiento secundario es más lento que el acceso

a la de memoria RAM, mucho más. Recuperar un solo carácter de la memoria de un computador personal toma alrededor de 150 nanosegundos, que son 150 milmillonésimas de segundo. Si ese carácter se almacena en un disco fijo, tiene que trasladarse del disco a la memoria RAM antes de que el computador pueda hacer algo con él. El tiempo medio requerido para acceder a un carácter de un disco de computador personal es un poco mayor de 75 milisegundos (mseg), o 75 milésimas de segundo.

Ahora bien, 75 milisegundos no aparentan ser mucho, pero el costo de tiempo por tener acceso a un disco debe verse en relación con el tiempo de acceso al mismo byte si estuviera ya en RAM. Para apreciar este costo desde otra perspectiva se planteará un problema donde sea el lector el que recupere información, en lugar de un computador.

Supóngase que se pide al lector una definición del término *registro*, mediante accesos al libro que tiene en sus manos. El lector puede intentarlo midiendo el tiempo que tardaría. Nuestra suposición es que si trabaja rápidamente y consulta el índice tardará alrededor de 20 segundos en obtener esta definición. Se calcula así el tiempo que una persona tarda en recuperar información en RAM.

Ahora calculemos cuánto tardaría en encontrar esta definición si obtuviera la información del almacenamiento secundario. Para seguir el modelo del computador, es necesario mantener la relación entre el acceso a memoria RAM y almacenamiento secundario igual a la relación que realmente existe en un computador cuando lee de RAM y de disco fijo.

Si se hacen las operaciones aritméticas, se observa que 75 mseg es 500 000 veces mayor que 150 nanosegundos. Así, puesto que el acceso de una persona a RAM tomó 20 segundos, su acceso al almacenamiento secundario tomaría 10 000 000 de segundos; esto es, 2778 horas, o ¡casi 116 días!

Usar un computador grande con una unidad de disco rápida y costosa puede ayudar a reducir la relación entre un acceso a memoria y otro a disco, pero la reducción no es considerable. Un computador grande tiene un tiempo medio de acceso al disco que es de alrededor de la tercera parte del de un computador pequeño, pero el acceso a RAM también es más rápido. Incluso con un computador grande, los accesos al disco son desmedidamente más caros que los accesos a la memoria electrónica.

Como se puede observar, cuando se decide almacenar información en un disco en lugar de hacerlo en RAM, se toma una decisión costosa. Por ello, tales decisiones deben hacerse cuidadosamente.

1.3

ARCHIVOS

Los datos colocados en el almacenamiento secundario se reúnen en *archivos*. Por lo común, un archivo se define como una colección de información relacionada, de acuerdo con las siguientes definiciones:

Una colección de registros que abarca un conjunto de entidades con ciertos aspectos en común y organizados para algún propósito particular

Tremblay y Sorenson [1984]

Una colección de registros semejantes, guardados en dispositivos de almacenamiento secundario del computador

Wiederhold [1983]

Uno de los revisores de la primera versión de este manuscrito cambió esta definición de “colección de información relacionada” para hacer una interesante e importante observación:

Siempre he sostenido que la información está relacionada porque está en el mismo archivo.

Lemone [1984]

Nos agrada este punto de vista porque destaca el hecho de que los datos incluidos en un archivo en realidad no se organizan por sí mismos. Las decisiones acerca de la estructuración de un archivo están entre las más críticas que debe tomar el diseñador de un sistema de archivos.

Volvamos por un momento a las dos fuentes de información, una que es accesible en 20 segundos, y la otra que requiere una espera de 116 días. Supóngase que sí se debe obtener la información del almacenamiento secundario, por lo que estamos resignados a esperar 116 días para recibirla. ¿Qué características se desearía que tuviera nuestra interacción con el almacenamiento secundario, si tenemos el compromiso de usarlo? Lo natural es que se pretenda tener las siguientes condiciones:

- Obtener exactamente lo que se necesita en el primer intento. No queremos esperar 116 días tan sólo para descubrir que recibimos información equivocada.
- Si nos es absolutamente imposible saber lo que necesitamos pedir en el primer intento, esperamos que la información en esa

fuente distante esté organizada de modo tal que se pueda conseguir, al menos, alguna información inicial que reduzca la cantidad de las siguientes solicitudes.

- Queremos obtener todo lo que necesitamos de una sola vez. Si sabemos que se necesitan cuatro partes diferentes de la información, seguramente desearíamos ser capaces de solicitarlas todas juntas, y no a razón de un dato por cada intervalo de 116 días.

Este es precisamente el tipo de consideraciones que se hacen en el diseño de las estructuras de archivos. Es inevitable emplear el almacenamiento secundario, pero se intenta minimizar su uso. Cuando solicitamos datos de un archivo deseamos ser capaces de ir directamente a ellos, si es posible, y en el mejor de los casos, ser capaces de obtener lo necesario en un solo acceso. La posibilidad de lograr estos objetivos está determinada por la estructura del archivo. El estudio formal de las estructuras de archivos está motivado por la necesidad que hay, considerando el costo relativamente alto del acceso al almacenamiento secundario, de que un diseñador de estructuras de archivos parta de un marco conceptual sólido.

1.4

ESTRUCTURAS DE ARCHIVOS VERSUS ESTRUCTURAS DE DATOS

Se supondrá que la mayoría de los lectores de este libro están familiarizados con los fundamentos básicos de las estructuras de datos, tales como listas ligadas y árboles binarios. El estudio de las estructuras de archivos es, en esencia, la aplicación de las técnicas de estructuras de datos a problemas especiales asociados con el almacenamiento y la recuperación de datos en dispositivos de almacenamiento secundario.

Hace pocos meses uno de nosotros entrevistó a un aspirante a un puesto que implicaba abocarse a un problema de manejo de mucha información. De manera informal se le expusieron las dificultades que habíamos tenido en encontrar un paquete de clasificación adecuado para ordenar los cientos de millones de registros utilizados en las estructuras de índices del sistema en que trabajábamos. El solicitante comentó: "Supongo que para ello se usa el algoritmo Quicksort, ¿no?"

Este comentario puede servir para ilustrar la diferencia entre un curso normal de *estructuras de datos* y el estudio de las *estructuras de archivos*. Normalmente el estudio de las estructuras de datos supone

que éstos se encuentran almacenados en memoria RAM. El término *acceso aleatorio*, como en *memoria de acceso aleatorio*, significa que el costo de recuperación de una parte de la información es igual al costo de recuperación de cualquier otra parte. Más aún, el costo del acceso a la información en RAM es en realidad inferior al costo de muchas otras operaciones, tales como la comparación de dos cantidades. Desde esta perspectiva tradicional de las estructuras de datos puras, es posible demostrar que el método Quicksort es, en promedio, más rápido que otros algoritmos de clasificación. Así, cuando el entrevistado preguntó por la clasificación rápida demostró un conocimiento adquirido en un curso de estructuras de datos.

Por desgracia también demostró que no había tenido la oportunidad de tomar un curso de estructuras de archivos. Un archivo constituido por cientos de millones de registros es demasiado grande para cargarlo en memoria RAM a fin de clasificarlo; la clasificación implica el uso de almacenamiento secundario, lo que significa que ya no se está en un ambiente de *acceso aleatorio*; ahora el costo de recuperar algunas partes de la información es desproporcionadamente mayor que el costo de recuperar otras. Un enfoque para la clasificación que se oriente al diseño de estructuras de archivos reconocería esto. Así, minimizaría el número de accesos al disco y además intentaría adecuar el patrón de acceso de modo tal que, cuando se necesite otra parte de la información, ésta se localice donde el acceso resulte relativamente barato.

El algoritmo Quicksort no toma en cuenta esas diferencias. Si se aplicara al archivo completo se tendrían muchos accesos al disco, sin considerar el costo relativo de cada uno y, por tanto, se tendría un desempeño muy lento. En un capítulo posterior se pondrá especial atención en las limitaciones que presentan los dispositivos de almacenamiento secundario en la construcción de un algoritmo de clasificación, el cual resulta muy diferente del de clasificación rápida ya mencionado.

1.5

UN CONJUNTO DE HERRAMIENTAS CONCEPTUALES

El estudio de las estructuras de archivos implica, entonces, obtener ventaja de un medio que proporciona un almacenamiento amplio, barato y no volátil. También supone confrontar las limitaciones del medio, tales como la relación de 20 seg/116 días entre el tiempo de acceso a la memoria primaria y el de acceso al almacenamiento secun-

dario. La solución surge con las estructuras de archivos, que reducen el número de accesos al almacenamiento secundario e incrementan la cantidad de información obtenida en un solo acceso.

Uno de los atractivos que tiene trabajar con estructuras de archivos es que su diseño considera sólo un puñado de conceptos básicos, aunque parece haber un número casi infinito de formas de combinar y variar esos conceptos. Estudiar el problema de almacenar y extraer información siempre brinda la oportunidad de hacer algo nuevo. Pero junto al placer de hacer algo nuevo, está también el placer que proporciona el trabajo con herramientas conceptuales fundamentales que se comprenden bien y que se han usado anteriormente muchas veces.

El propósito de este libro es proporcionar el conjunto de herramientas conceptuales necesario para construir estructuras de archivos útiles que den respuesta a diversos problemas.



RESUMEN

Aun cuando la memoria primaria (RAM) del computador proporciona el tipo de memoria más rápido y simple, existen varias razones por las cuales es necesario guardar la información en dispositivos de almacenamiento secundario en lugar de hacerlo en RAM:

- La capacidad de la memoria RAM normalmente es bastante limitada, en comparación con el espacio disponible en el almacenamiento secundario.
- La memoria RAM es más costosa que el almacenamiento secundario, y
- La memoria RAM es volátil; el almacenamiento secundario no.

El problema con el almacenamiento secundario es que se necesita mucho más tiempo para tener acceso a los datos que en RAM. La decisión de acceder a los datos del almacenamiento secundario es tan costosa que resulta muy importante enviar y recuperar datos con inteligencia. Cuando se busca un dato, se espera encontrarlo en el primer intento, o al menos en pocos intentos, sin necesidad de revisar todo un archivo para ello. O, si se buscan varios datos, se espera obtenerlos todos de una sola vez, en lugar de tener que buscar por separado cada uno.

Otra diferencia importante entre el almacenamiento secundario y la memoria RAM es que, en el almacenamiento secundario, el costo de recuperación varía de unas partes de la información a otras; en RAM todos los accesos tienen el mismo costo. Cuando existe un patrón en la forma en que se tiene acceso a la información, un buen diseño de las estructuras de archivos trata de organizarla para minimizar el costo del acceso.

En los dispositivos secundarios, la información está organizada en archivos. Un archivo es tan sólo una colección de bytes que representa información, pero éstos pueden estar organizados jerárquicamente en estructuras capaces de afectar enormemente la facilidad y eficiencia de las operaciones sobre los archivos.

Conforme se aprende sobre archivos se descubre que la clave para trabajar de manera apropiada la constituyen varios conceptos básicos. Esos conceptos se relacionan con las formas de organizar los archivos para ajustar las diferencias que hay de costos en espacio y tiempo cuando se trabaja con datos en almacenamiento secundario en lugar de en memoria RAM.

TERMINOS CLAVE

Archivo. Una colección de bytes que representa información y que normalmente se guarda en almacenamiento secundario. Para su procesamiento, todo el contenido de un archivo, o parte de él, suele cargarse en memoria RAM.

Estructuras de archivos. Organización impuesta a un archivo para facilitar su procesamiento. Las estructuras de archivos que se encuentran en este libro incluyen campos, registros, bloques, árboles, índices, secuencias y otras construcciones conceptuales.

RAM. Memoria de acceso aleatorio. En este libro el término se emplea como sinónimo de *memoria primaria*. La palabra “aleatorio” se refiere a que proporciona acceso inmediato a cualquier localidad de almacenamiento, sin importar su dirección. En el texto se confrontan la memoria RAM y el almacenamiento secundario, al cual normalmente le toma mucho más tiempo tener acceso a cualquier localidad, y requiere también distintos tiempos de acceso a localidades distintas.

2

OBJETIVOS

Describir el proceso de enlace de un *archivo lógico* dentro de un programa con un *archivo físico* existente o un dispositivo.

Describir los procedimientos usados para crear, abrir y cerrar archivos.

Proporcionar ejemplos de lectura y escritura de archivos.

Introducir el concepto de *posición* dentro de un archivo y describir los procedimientos para *localizar* diferentes posiciones.

OPERACIONES FUNDAMENTALES PARA EL PROCESAMIENTO DE ARCHIVOS

PLAN GENERAL DEL CAPITULO

2.1 Archivos físicos y archivos lógicos

2.2 Apertura y creación de archivos

2.3 Cierre de archivos

2.4 Lectura y escritura

2.5 Detección del fin del archivo

2.6 Localización

2.6.1 Localización en C

2.6.2 Localización en Pascal

2.7 Caracteres inesperados en archivos

El programa *agrega* en Pascal

2.1

ARCHIVOS FISICOS Y ARCHIVOS LOGICOS

Cuando se habla de un archivo en un disco o cinta, se hace referencia a un conjunto en particular de bytes almacenados allí. La palabra "archivo", en este sentido, significa que éste tiene una existencia física. Una unidad de disco puede contener cientos, incluso miles, de estos *archivos físicos*.

Desde el punto de vista de un programa de aplicación, la noción de archivo es distinta. Para el programa, un archivo es algo parecido a una línea telefónica conectada a una red telefónica. El programa puede recibir o enviar bytes a través de esta línea telefónica, pero no sabe de dónde provienen o a dónde van esos bytes; sólo conoce ese extremo de la línea. Además, aunque puedan existir miles de archivos físicos en un disco, un programa normalmente está limitado al uso de sólo alrededor de 20 enlaces telefónicos.

El programa de aplicación se apoya en el sistema operativo para que se encargue de los detalles del sistema de conmutación telefónica, como se ilustra en la figura 2.1. Los bytes que el programa recibe de la línea podrían originarse en un archivo físico real o provenir del teclado, o de algún otro dispositivo de entrada. De igual forma, los bytes que el programa envía por la línea pueden llegar a un archivo o pueden aparecer en la pantalla de la terminal. Aunque el programa puede no saber de dónde vienen o a dónde van los bytes, sí sabe qué línea está

usando. Normalmente esta línea se conoce como *archivo lógico*, para distinguirla de los *archivos físicos* en disco o cinta.

Antes de que el programa pueda abrir un archivo para usarlo, el sistema operativo debe recibir instrucciones para hacer un enlace entre un archivo lógico (p. ej., una línea telefónica) y algún archivo o dispositivo físicos. Cuando se usan sistemas operativos tales como el OS/MVS de IBM, esas instrucciones se proporcionan a través de un lenguaje de control (*JCL*, por sus siglas en inglés). En mini y microcomputadores hay sistemas operativos más modernos, tales como UNIX, MS-DOS y VMS, que proporcionan las instrucciones dentro del programa. Por ejemplo, en Turbo Pascal[†] la asociación entre un archivo lógico llamado *arch_ent* y un archivo físico llamado *archmío.dat* se hace con la siguiente instrucción:

```
assign(arch_ent, 'archmío.dat');
```

Esta proposición solicita al sistema operativo encontrar el archivo físico *archmío.dat*, y hacer después el enlace asignándole un archivo lógico (línea telefónica). El número que identifica la línea telefónica particular que se asigna se devuelve a través de la variable *arch_ent* de tipo FILE, que es el *nombre lógico* del archivo. Este nombre lógico se usa para referirse al archivo dentro del programa. De nuevo se aplica la analogía de la línea telefónica: el teléfono de una oficina está conectado a seis líneas telefónicas. Cuando entra una llamada, se recibe por el interfono un mensaje tipo: "Tiene una llamada por la línea 3". La recepcionista no dice: "Tiene una llamada por el 918-123-4567". Es necesario tener la llamada identificada *lógicamente*, no *físicamente*.

2.2

APERTURA Y CREACION DE ARCHIVOS

Una vez que se tiene un identificador de archivo lógico enlazado con un archivo o dispositivo físico, es necesario declarar lo que se desea hacer con el archivo. En general, se tienen dos opciones:

1. ABRIR (OPEN) un archivo existente, o
2. CREAR (CREATE) un archivo nuevo, eliminando cualquier contenido ya existente en el archivo físico.

[†] Los diferentes compiladores de Pascal varían mucho respecto a los procedimientos de E/S, debido a que el Pascal estándar contiene poco sobre definiciones de E/S. A lo largo de este libro se usará el término *Pascal* cuando se analicen las características que sean comunes a la mayoría de las versiones de Pascal. Cuando se haga referencia a las características de una versión en particular, como Turbo Pascal, se hará la aclaración.

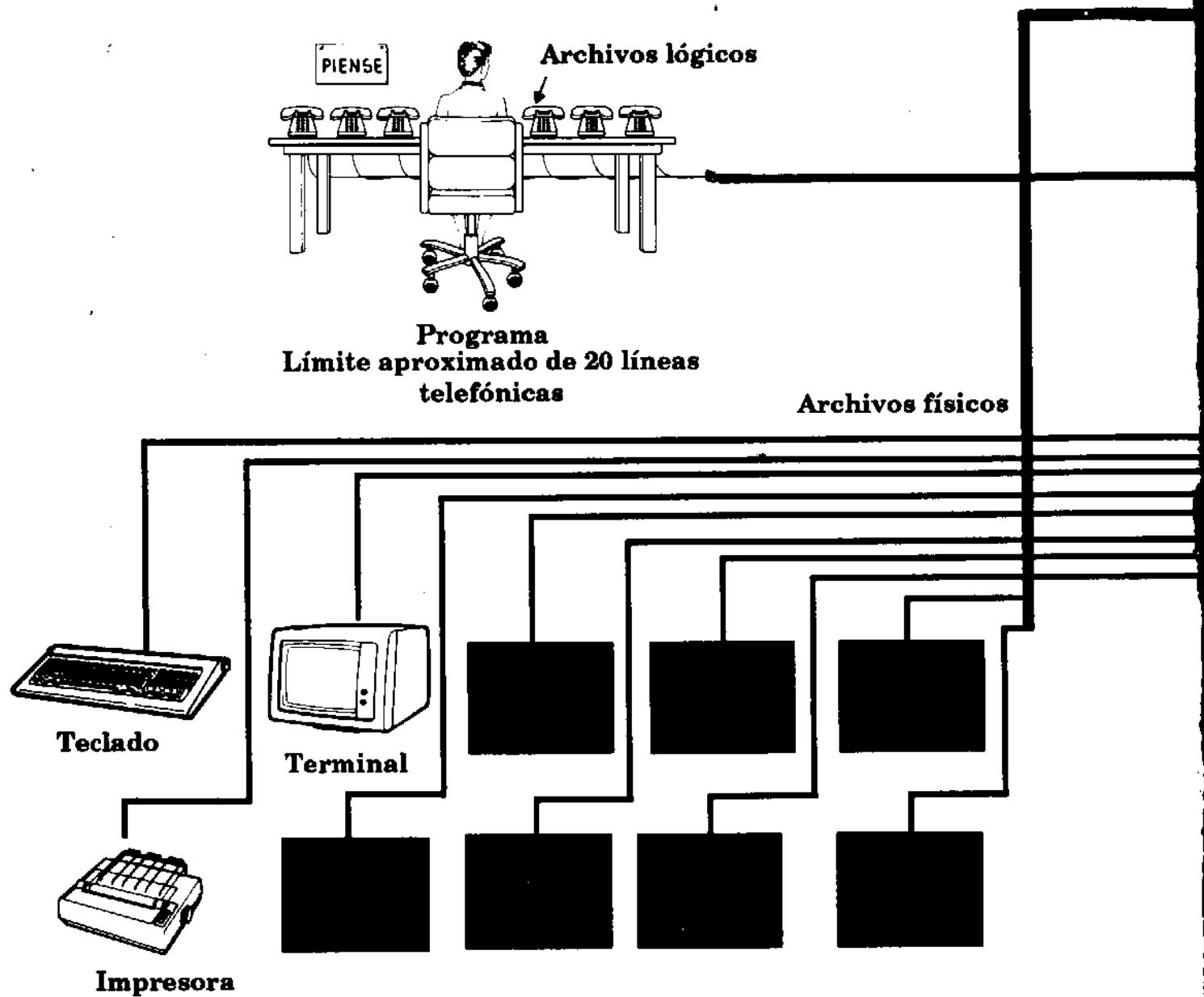
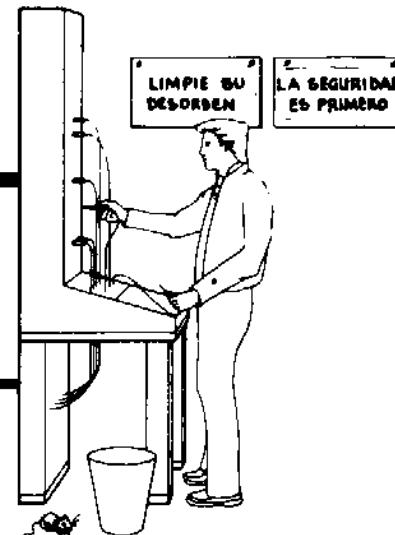
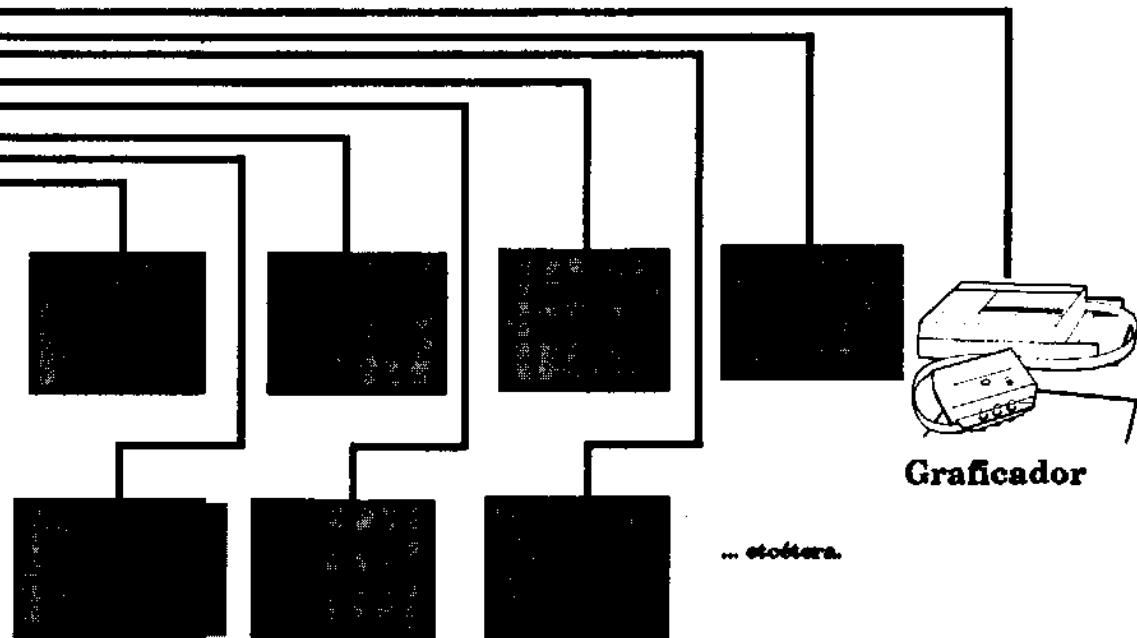


FIGURA 2.1 • El programa se apoya en el sistema operativo para hacer las conexiones entre los archivos lógicos y los archivos y dispositivos físicos.

La apertura de un archivo implica que está listo para que el programa lo use. El usuario estará colocado al principio del archivo y listo para leer o escribir. El contenido del archivo no se altera con la proposición OPEN.



Panel de conmutación del sistema operativo
Puede hacer conexiones con miles de archivos o
dispositivos de E/S



Crear un archivo es también abrirlo, en el sentido de que estará listo para usarse después de creado. Puesto que un archivo recién creado no tiene contenido, el único uso que tiene sentido al inicio es la escritura.

En Pascal, la proposición *reset()* se usa para abrir archivos y la proposición *rewrite()* se usa para crearlos. Por ejemplo, para abrir un archivo en Turbo Pascal puede usarse una secuencia de proposiciones tales como:

```
assign(arch_ent, 'archmio.dat');
reset(arch_ent)
```

Nótese que se usa el nombre lógico, y no el físico, en la proposición *reset()*. Para crear un archivo en Turbo Pascal, las instrucciones deben ser:

```
assign(arch_sal, 'archmio.dat');
rewrite(arch_sal)
```

En Turbo Pascal la separación que se hace de la *asignación* de los nombres lógico y físico para diferenciarla de la apertura del archivo real tiene la cualidad de hacer claros los pasos del proceso, pero no es común. El enfoque usado en el lenguaje C es similar al empleado en la mayoría de los otros lenguajes: la *asignación del nombre* y la apertura del archivo se realizan con una sola llamada al sistema operativo. En C, la función *open()* toma la siguiente forma:

```
fd = open(nombrearch, MODO);
```

donde fd, nombrearch y MODO tienen los siguientes significados:

fd El *descriptor del archivo* (*file descriptor, fd*). Es el número de línea telefónica empleado para referirse al archivo dentro del programa. Como puede observarse, es un valor entero devuelto por *open()*, y no algo establecido mediante una proposición aparte.

nombrearch El nombre del archivo físico.

MODO El modo de acceso al archivo. Se especifica como un entero igual a 0 si el archivo se abre sólo para lectura; a 1 si es sólo para escritura, y a 2 si se abre para lectura y escritura.

Además de la combinación de la asignación del nombre y la apertura del archivo en una sola llamada, se encuentra aquí otra noción, la de *modo de acceso*. Muchas versiones de Pascal han ampliado el Pascal estándar para ofrecer esta posibilidad. El control del acceso a un archivo, restringiéndolo sólo al uso de lectura, puede ser particularmente importante cuando varios programas comparten el acceso a un archivo.

Se puede emplear C para crear un archivo nuevo, de la siguiente forma:

```
fd = creat (nombrearchivo, PMODE);
```

donde *fd* y *nombrearchivo* tienen el mismo significado que en la función *open()*. El nuevo, diferente argumento de *creat()* es:

<i>PMODE</i>	El modo de protección establecido para el archivo, que indica quién puede leerlo, escribir en él y ejecutarlo.
--------------	--

Los diferentes sistemas operativos ofrecen distintas clases de protección de archivos, y algunos no ofrecen ninguna. Por ejemplo, en el sistema operativo UNIX, PMODE es un número octal de tres dígitos que indica cómo deben utilizar el archivo el propietario (primer dígito), los miembros del grupo del propietario (segundo dígito), y todos los demás. El primer bit de cada número octal indica el permiso de lectura; el segundo, el permiso de escritura, y el tercero, el permiso de ejecución. De esta forma, si PMODE es el número octal 0751, el propietario del archivo tendría permiso para leer, escribir y ejecutar el archivo; el grupo del propietario tendría permiso para leer y ejecutar, y las demás personas tendrían permiso sólo para ejecutarlo.

$PMODE = 0751 \Rightarrow$	$\begin{array}{ccc} r & w & e \\ 1 & 1 & 1 \\ \text{prop.} & \text{grupo} & \text{los demás} \end{array}$	$\begin{array}{ccc} r & w & e \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{array}$
----------------------------	---	--

La protección de archivos está más ligada al sistema operativo que a un lenguaje específico. Por ejemplo, las versiones de Pascal en sistemas que manejan protección de archivos, tales como VAX/VMS, frecuentemente incluyen ampliaciones al Pascal estándar, las cuales permiten asociar un estado de protección a un archivo cuando se crea.

2.3

CIERRE DE ARCHIVOS

En términos de la analogía de la línea telefónica, el cierre de un archivo es como colgar el teléfono. Cuando esto se hace, la línea telefónica queda disponible para recibir o hacer otra llamada; cuando se cierra un archivo, el nombre lógico del archivo o el descriptor del archivo queda disponible para usarse con otro archivo. El cierre de un archivo que se

ha usado para salida también asegura que todo se ha escrito en el archivo. Como se estudiará en un capítulo posterior, es más eficaz mover datos en bloques desde o hacia el almacenamiento secundario que mover un byte cada vez. En consecuencia, el sistema operativo no envía inmediatamente los bytes que se escriben, sino que los guarda en un buffer (almacenamiento temporal) para transferirlos en forma de un bloque de datos. El cierre de un archivo asegura que el buffer de ese archivo se vació ya, y que todo lo escrito se envió efectivamente al archivo.

Por lo común, el sistema operativo cierra los archivos automáticamente cuando el programa termina, de modo que el uso explícito de una proposición CLOSE (cerrar) se requiere sólo como protección contra la pérdida de datos en caso de interrupción del programa, y para liberar los nombres lógicos de los archivos, de manera que se puedan volver a usar. Algunos lenguajes, entre ellos algunas versiones de Pascal, ni siquiera proporcionan la proposición CLOSE. Sin embargo, el cierre de archivos explícito es posible en los lenguajes C, Turbo Pascal, Pascal VAX, PL/I, y en la mayoría de los otros lenguajes empleados para trabajo serio de procesamiento de archivos.

Ahora que ya se sabe cómo conectar y desconectar programas con archivos físicos, y cómo abrirlos, se está preparado para empezar a enviar y recibir datos.

2.4

LECTURA Y ESCRITURA

La *lectura* y la *escritura* son fundamentales para el procesamiento de archivos, ya que son acciones que efectúan las operaciones de *entrada* y *salida* (E/S). La forma concreta de las proposiciones de lectura y escritura empleadas en los distintos lenguajes es variable. Algunos lenguajes proporcionan acceso de muy alto nivel a la lectura y la escritura, y se encargan automáticamente de los detalles por el programador. Otros lenguajes proporcionan acceso de un nivel muy inferior. El uso de Pascal y C permite explorar algunas de esas diferencias.[†]

Se comenzará aquí con la lectura y escritura en un nivel relativamente bajo. Es útil tener algún conocimiento sobre programación de sistemas; saber qué sucede cuando se envía información a un archivo o se recibe de él.

[†]Para acentuar esas diferencias y observar las operaciones de E/S en un nivel que se asemeje más al de la programación de sistemas, se usan las llamadas al sistema *read()* y *write()* en C, en lugar de las funciones de nivel superior como *fgetc()*, *fgets()* y otras.

Una llamada de lectura de bajo nivel requiere tres componentes de información, expresados aquí como argumentos de una función genérica READ().

READ(Archivo_fuente, Dir_destino, Tamaño)

Archivo_fuente. La llamada READ() debe saber de dónde leerá. Se especifica la fuente mediante el nombre lógico de un archivo (línea telefónica), a través del cual se recibirán los datos. (Recuérdese que antes de hacer cualquier lectura, el archivo debe estar abierto, de tal forma que exista la conexión entre un archivo lógico y un archivo o dispositivo físico específico.)

Dir_destino. READ() debe saber dónde colocar la información que lee del archivo de entrada. En esta función genérica se especifica el destino proporcionando la dirección de la localidad de memoria donde se desea almacenar los datos.

Tamaño. Por último, READ() necesita saber cuánta información debe extraer del archivo. Aquí el argumento se proporciona como una indicación de la cantidad de bytes.

La proposición WRITE() es similar; la única diferencia es que los datos se mueven en dirección opuesta:

WRITE(Archivo_destino, Dir_fuente, Tamaño)

Archivo_destino. El nombre lógico del archivo (línea telefónica) que se usa para enviar los datos.

Dir_fuente. WRITE() debe saber dónde se localiza la información que enviará. Esta especificación se proporciona como una dirección de memoria.

Tamaño. Se debe proporcionar el número de bytes por escribir.

A continuación se hacen algunas lecturas y escrituras para aprender a usar esas funciones. Este primer programa sencillo de procesamiento de archivos, que llamaremos LISTA, abre un archivo de entrada para leerlo carácter por carácter, y envía cada uno a la pantalla después de leerlo del archivo. LISTA incluye los siguientes pasos.

1. Presentar un mensaje solicitando el nombre del archivo de entrada.
2. Leer la respuesta que el usuario dio a través del teclado en una variable llamada *nombrearchivo*.
3. Abrir (OPEN) el archivo de entrada.
4. MIENTRAS existan caracteres por leer en el archivo de entrada:
 - a) Leer (READ) un carácter del archivo;

- b) Escribir (WRITE) el carácter en la pantalla de la terminal.
 5. Cerrar (CLOSE) el archivo de entrada.

Las figuras 2.2 y 2.3 son realizaciones en C y Pascal, respectivamente, de este programa. Es conveniente comparar las diferencias entre esas dos realizaciones.

Los pasos 1 y 2 del programa comprenden la escritura y la lectura, y cada una se realiza por medio de las funciones usuales de manejo de la pantalla y del teclado. El paso 4a, donde se lee del archivo de entrada, representa la primera petición real de E/S al archivo. Nótese que la llamada a *read()*, en el lenguaje C, es casi idéntica a la proposición genérica de bajo nivel *READ()*, descrita anteriormente. En realidad, se empleó la llamada de sistema *read()* en C como el modelo de la función genérica *READ()*. El primer argumento de la función proporciona el descriptor de archivo (la versión de C del nombre lógico del archivo) como *fuente* para la entrada; el segundo argumento proporciona la *dirección* de una variable de tipo carácter que se emplea como el *destino* de los datos, y el tercer argumento especifica que sólo se leerá un byte.

```
/* lista.c -- Programa que lee caracteres de un archivo y los
   transcribe a la pantalla de la terminal
*/
#define SOLOEC 0

main()  {

    char c;
    int fd; /* Descriptor del archivo */
    char nombrearchivo[20];

    printf("Proporcione el nombre del archivo: "); /* Paso 1 */
    gets(nombrearchivo); /* Paso 2 */
    fd = open(nombrearchivo, SOLOEC); /* Paso 3 */

    while (read(fd, &c, 1) > 0) /* Paso 4a */
        write(1, &c, 1); /* Paso 4b */

    close(fd); /* Paso 5 */
}
```

FIGURA 2.2 • El programa LISTA en C.

Los argumentos para la llamada a *read()* en Pascal comunican la misma información en un nivel más alto. De nuevo, el primer argumento es el nombre lógico del archivo para la fuente de entrada. El segundo

```

PROGRAM lista(INPUT, OUTPUT);
{
  Lee la entrada de un archivo y lo transcribe a la pantalla de la terminal
}
VAR
  c           : char;
  archent     : file of char;                      { Nombre lógico del archivo }
  nombrearchivo : packed array [1..20] of char; { Nombre del archivo físico }

BEGIN {principal}

  write('Proporcione el nombre del archivo: ');      { Paso 1 }
  readln(nombrearchivo);                            { Paso 2 }
  assign(archent,nombrearchivo);                   { Paso 3 }
  reset(archent);

  while not (eof(archent)) DO
  BEGIN
    read(archent,c);                                { Paso 4a }
    write(c);                                       { Paso 4b }
  END;
  close(archent)                                    { Paso 5 }
END.

```

FIGURA 2.3 • El programa LISTA en Pascal.

argumento proporciona el *nombre* de la variable de tipo carácter empleada como destino; mediante ese nombre, Pascal puede encontrar la dirección. Debido a la importancia que se da en Pascal a los tipos de variables, el tercer argumento de la función genérica **READ()** no es necesario. En Pascal se supone que, como se trasladan los datos a una variable de tipo *char*, en realidad se quiere leer sólo un byte.

Después de que se leyó el carácter, se transcribe a la pantalla en el paso 4b. De nuevo las diferencias entre C y Pascal son indicativas de la gama de enfoques de E/S que se aplican en diferentes lenguajes. En C, todo en la llamada a *write()* debe ser explícito; el uso del descriptor de archivo especial 1 para identificar la pantalla de la terminal como el destino de lo escrito,

```
  write(1, &c, 1);
```

significa: "Escribe en la pantalla el contenido de la memoria que empieza en la dirección *&c*. Escribe sólo un byte." Los programadores principiantes en C deben poner especial atención al uso del símbolo *&* en la llamada a *write()*; esta llamada de C en particular, como llamada de muy bajo nivel, requiere que el programador proporcione la *dirección* inicial en memoria RAM de los bytes que se van a transferir.

De nuevo, en Pascal se trabaja en un nivel superior. Cuando en la proposición *write()* no se especifica el nombre lógico del archivo, en Pascal se supone que se está escribiendo en la pantalla de la terminal. Como la variable *c* es de tipo *char*, en Pascal se supone que se está escribiendo un solo byte. La proposición simplemente queda:

```
write(c);
```

Como en la proposición *read()*, Pascal se encarga de encontrar la dirección de los bytes; el programador sólo necesita especificar el nombre de la variable *c* que está asociada con esa dirección.

2.5

DETECCION DEL FIN DEL ARCHIVO

Los programas de las figuras 2.2 y 2.3 deben saber cuándo terminar el ciclo *while* y dejar de leer caracteres. En Pascal y en C la forma de indicar el fin del archivo es diferente. Se ilustran así dos de los enfoques más comunes para la detección del fin del archivo.

En Pascal se proporciona una función booleana, *eof()*, que puede usarse para preguntar por el fin del archivo. Conforme se realiza la lectura del archivo, el sistema operativo mantiene un registro de la posición en el archivo por medio de algo conocido como *apuntador de lectura/escritura*. Esto es necesario para que, al leer el siguiente byte, el sistema sepa de dónde extraerlo. La función *eof()* pregunta al sistema si el apuntador de lectura/escritura ya pasó del último elemento del archivo. Si eso sucede, *eof()* devuelve *true*; si no, devuelve *false*. Como se ilustra en la figura 2.3, se emplea la llamada a *eof()* antes de intentar leer el siguiente byte. En el caso de un archivo vacío, *eof()* devuelve inmediatamente *true* y no se lee ningún byte.

En el lenguaje C, la llamada a *read()* devuelve el número de bytes leídos. Si *read()* devuelve un valor de cero, entonces el programa ha llegado al final del archivo. Así que, en lugar de usar la función *eof()*, se construye el ciclo *while* de tal forma que se ejecute mientras que la llamada a la función *read()* siga encontrando material de lectura.

2.6

LOCALIZACION

En los programas de los ejemplos anteriores la lectura del archivo se realiza *en forma secuencial*, un byte después de otro hasta que se llega al final del archivo. Cada vez que se lee un byte, el sistema operativo

mueve el apuntador de lectura/escritura hacia adelante, y queda listo para leer el siguiente byte.

Algunas veces se desea leer o escribir sin perder tiempo por hacerlo en forma secuencial byte por byte. Quizás la siguiente sección de la información que se necesita está a 10 000 bytes de distancia, de manera que se desea saltar para iniciar ahí la lectura. O quizás es necesario saltar hasta el final del archivo para agregar información nueva. Para satisfacer esas necesidades se debe ser capaz de controlar el movimiento del apuntador de lectura/escritura.

La acción de moverse directamente hasta cierta posición en un archivo suele llamarse *localización (seek)*. Esto requiere al menos dos datos, formulados aquí como argumentos de la función SEEK():

SEEK(Archivo_fuente, Distancia)

donde las variables tienen los siguientes significados:

Archivo_fuente. Es el nombre lógico del archivo donde ocurre la localización.

Distancia. Es el número de posiciones que el apuntador se mueve desde el inicio del archivo.

Ahora bien, si se desea ir directamente desde el origen a la posición 373 del archivo llamado *datos*, no es necesario moverse en forma secuencial a lo largo de las 372 posiciones previas, sino solicitar

SEEK(*datos*, 373)

2.6.1 LOCALIZACION EN C

Una de las características del sistema operativo UNIX que se incorporó en muchas realizaciones del lenguaje C es la capacidad de concebir un archivo como un *arreglo de bytes* potencialmente muy extenso que resulta estar guardado en almacenamiento secundario. En un arreglo de bytes en memoria RAM es posible moverse a cualquier byte en particular mediante el uso de un subíndice. La función de localización en el lenguaje C, llamada *lseek()*, ofrece una posibilidad semejante para archivos, y permite colocar el apuntador de lectura y escritura en cualquier byte del archivo.

La función *lseek()* tiene la siguiente forma:

```
cpos = lseek(fd,distancia_en_bytes,origen);
```

donde las variables tienen los siguientes significados:

<i>cpos</i>	Un valor entero largo devuelto por <i>lseek()</i> , igual a la posición (en bytes) del apuntador de lectura y escritura después de que se ha movido. <i>lseek()</i> devuelve un valor de <i>-1L</i> si intenta colocarse fuera del límite del archivo.
<i>fd</i>	El descriptor del archivo para el cual se está aplicando <i>lseek()</i> .
<i>distancia_en_bytes</i>	El número de bytes que se moverán desde algún <i>origen</i> en el archivo. La distancia en bytes debe especificarse como un entero largo, de ahí el nombre de <i>lseek()</i> para una localización larga. La distancia en bytes puede ser negativa cuando convenga.
<i>origen</i>	Un entero que especifica la posición inicial desde la cual será tomada la <i>distancia en bytes</i> . El <i>origen</i> puede tener el valor de 0, 1 o 2: 0 - <i>lseek()</i> desde el inicio del archivo; 1 - <i>lseek()</i> desde la posición actual; 2 - <i>lseek()</i> desde el final del archivo.

El siguiente fragmento de programa muestra cómo se podría usar *lseek()* para trasladarse hasta una posición que está 373 bytes dentro del archivo.

```

long cpos, lseek();
int fd;

cpos = lseek(fd, 373L, 0);
if(cpos == -1L)
    printf("Intento de colocarse más allá del final
           del archivo\n");

```

La figura 2.4 muestra un programa en C que toma una cadena de entrada desde el teclado y después localiza el final del archivo para escribir la cadena ahí.

El programa también ilustra cómo pueden combinarse las proposiciones *open()* y *creat()* para volver a usar un archivo existente o crear uno nuevo.

```

/* agrega.c -- toma una cadena de entrada y la agrega al final
   de un archivo.
   El archivo se abre o se crea, según se requiera.
*/
#define LEEEESC 2
#define PMODE 0644 /* el propietario puede leer y escribir;
                  los demás sólo leer */
main() {
    char c;
    int fd;                                /* Descriptor del archivo */
    char nombrearchivo[20];
    char cad_nueva[80];                     /* Cadena de entrada      */
    long lseek();

    printf("Proporcione el nombre del archivo: ");
    gets(nombrearchivo);
    if ((fd = open(nombrearchivo, LEEEESC)) < 0)/* Si OPEN falla     */
        fd = creat(nombrearchivo, PMODE);      /* entonces CREAT     */

    printf("Cadena que se agregará: ");
    gets(cad_nueva);
    lseek(fd, 0L, 2);                        /* Salta al final      */
    write(fd, cad_nueva, strlen(cad_nueva)); /* Escribe los datos */
    close(fd);
}

```

FIGURA 2.4 • El programa AGREGA en C.

2.6.2 LOCALIZACION EN PASCAL

La concepción de un archivo como se presenta en Pascal difiere de la concepción en UNIX y C al menos en dos aspectos importantes:

- En C un archivo es una secuencia de bytes, de tal forma que la referencia dentro del archivo se realiza byte por byte. Cuando se localiza una posición, se expresa la dirección en términos de bytes. En Pascal, un archivo es una secuencia de elementos de algún tipo en particular. Un archivo puede ser una secuencia de caracteres, una secuencia de enteros, una secuencia de registros, etc. La referencia a direcciones dentro de un archivo en Pascal se realiza en términos de esos elementos. Por ejemplo, si un archivo está compuesto por registros de 100 bytes y se desea hacer referencia al cuarto registro, en Pascal se lograría mediante una simple referencia al registro número 4. En C, donde el enfoque

está dado siempre y únicamente en términos de bytes, se tendría que hacer referencia al cuarto registro como la dirección del byte 400.

- El Pascal estándar no permite, en realidad, la localización, ya que el modelo de E/S para el Pascal estándar es la cinta magnética, que se debe leer secuencialmente. En Pascal estándar, agregar datos al final del archivo implica leer el archivo completo, transcribir los datos del archivo de entrada a un segundo archivo de salida y agregar entonces nuevos datos al final del archivo de salida. Sin embargo, muchas versiones de Pascal, tales como Pascal VAX y Turbo Pascal, han ampliado el estándar para manejar la localización.

Una ampliación de Pascal, propuesta por la Junta del Comité de Estándares de Pascal ANSI/IEEE [1984], que podrá integrarse en el futuro al estándar de Pascal, incluye los siguientes procedimientos y funciones que permiten la localización.

SeekWrite(f,n). Procedimiento que localiza el elemento con índice *n* dentro del archivo *f* y coloca el archivo en la modalidad de escritura, de tal forma que tanto el elemento seleccionado como los siguientes pueden ser modificados.

SeekRead(f,n). Procedimiento que localiza el elemento con índice *n* dentro del archivo *f* y coloca el archivo en la modalidad de lectura, de tal forma que tanto el elemento seleccionado como los siguientes puedan examinarse. Si se intenta con SeekRead() una posición que rebasa el final del archivo, entonces se coloca al final del archivo.

Position(f). Función que devuelve el valor del índice que señala la posición del elemento actual del archivo.

EndPosition(f). Función que devuelve el valor del índice que señala la posición del último elemento del archivo.

Por desgracia, muchas versiones de Pascal, al reconocer la necesidad de proporcionar capacidades de localización, han desarrollado ya funciones antes de que se establecieran las propuestas. La consecuencia es que los mecanismos que manejan la localización varían mucho entre las versiones. En los capítulos subsecuentes se señalarán algunas de esas variaciones conforme se desarrolleen programas que lo requieran. Al final de este capítulo se menciona un programa llamado *agrega.pas*, cuyo funcionamiento es similar al programa en C de la figura 2.4, que ilustra la manera en que se realiza la localización en Turbo Pascal.

2.7

CARACTERES INESPERADOS EN ARCHIVOS

Conforme se crean las estructuras de archivos descritas en este texto es posible encontrar alguna dificultad con caracteres adicionales inesperados que aparecen en los archivos, con caracteres que desaparecen, o con contadores numéricos que se insertan dentro de los archivos. A continuación se presentan algunos ejemplos de las clases de situaciones que pueden ocurrir:

- En muchos computadores pequeños puede suceder que se agregue un Control-Z (el valor ASCII 26) al final de los archivos. Muchos sistemas lo usan para indicar el final del archivo, aun cuando no se haya colocado ahí. Es muy probable que esto suceda en los sistemas basados en MS-DOS o CP/M.
- Algunos sistemas adoptan la convención de indicar el final de la línea en un archivo de texto con un par de caracteres constituidos por un retorno de carro (CR: el valor ASCII 13) y un salto de línea (LF: el valor ASCII 10). Algunas veces los procedimientos de E/S escritos para tales sistemas amplían automáticamente los caracteres sueltos CR o LF a pares CR-LF. Esta adición de caracteres que no se solicitó puede causar grandes dificultades. De nuevo, lo más probable es que este fenómeno se encuentre en sistemas MS-DOS o CP/M.
- Los usuarios de sistemas grandes, tales como VMS, pueden enfrentarse al problema opuesto. Ciertos formatos de archivos bajo VMS *eliminan* los caracteres de retorno de carro del archivo sin preguntar, y los reemplazan por la cuenta de los caracteres en lo que el sistema percibió como una línea de texto.

Estos son sólo algunos ejemplos de las clases de modificaciones inesperadas que los sistemas administradores de registros o los paquetes de E/S pueden ocasionar a los archivos. Por lo general, están asociados con los conceptos de línea de texto o fin del archivo. Con tales modificaciones en los archivos se pretende facilitar el trabajo al usuario haciendo algunas cosas en forma automática. De hecho, esto puede funcionar para los usuarios que sólo almacenan texto en un archivo. Pero, por desgracia, los programadores que construyen estructuras de archivos complejas a veces dedican mucho tiempo a encontrar la forma de deshabilitar esta ayuda automática, para poder tener el control total de lo que construyen. A los lectores que se encuentren con esa clase

de dificultades al construir las estructuras de archivos descritas en este texto los puede consolar el saber que la experiencia adquirida al deshabilitar las ayudas automáticas les servirá una y otra vez en el futuro.



RESUMEN

En este capítulo se presentó una introducción a las operaciones fundamentales de los sistemas de archivos: OPEN(), CREATE(), CLOSE(), READ(), WRITE() y SEEK(). Cada una de estas operaciones implica la creación o el uso de un enlace entre un *archivo físico* que se almacena en un dispositivo secundario y un *archivo lógico*, el cual representa un punto de vista más abstracto desde la perspectiva del programa sobre el mismo archivo. Cuando el programa describe una operación usando el *nombre lógico del archivo*, la operación física equivalente tiene lugar en el archivo físico correspondiente.

Las cinco operaciones aparecen en los lenguajes de programación en diversas formas. Algunas veces son órdenes internas, otras veces son funciones, otras son llamadas directas a un sistema operativo. No todos los lenguajes proporcionan al usuario las cinco operaciones. Por ejemplo, la operación SEEK() no está disponible en Pascal estándar.

Antes de usar un archivo físico se le debe enlazar con un archivo lógico. En algunos ambientes de programación esto se hace con una proposición (p. ej., *assign* en Turbo Pascal), o con instrucciones fuera del programa (p. ej., instrucciones de lenguaje de control (JCL)). En otros lenguajes, el enlace entre el archivo físico y el lógico se realiza con OPEN() o con CREATE().

Las operaciones CREATE() y OPEN() preparan los archivos para la lectura o escritura. CREATE() permite crear un nuevo archivo físico. OPEN() funciona en un archivo físico ya existente, colocando usualmente el apuntador de lectura/escritura al inicio del archivo.

La operación CLOSE() rompe el enlace entre el archivo lógico y su archivo físico correspondiente. También asegura que el buffer del archivo se vacíe de suerte que todo lo escrito se envíe en efecto al archivo.

Vistas desde un nivel bajo, en el nivel de sistema, las operaciones de E/S READ() y WRITE() requieren tres datos:

- El *nombre lógico* del archivo donde se leerá o se escribirá;
- La *dirección* de un área de memoria que se usará para la parte del intercambio dentro del computador, y
- la indicación de *cuántos datos* se leerán o escribirán.

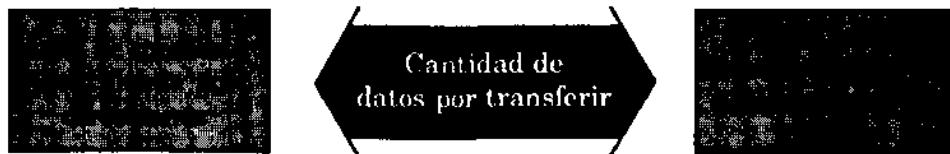


FIGURA 2.5 • Intercambios entre la memoria y un dispositivo externo.

Esos tres elementos fundamentales del intercambio se ilustran en la figura 2.5.

READ() y WRITE() son funciones suficientes para trasladarse secuencialmente a través de un archivo, a cualquier posición deseada, pero esta forma de acceso suele resultar muy ineficiente. Algunos lenguajes proporcionan operaciones de localización que permiten a un programa moverse directamente hacia determinada posición en un archivo. C proporciona el acceso directo por medio de la operación *lseek()*. Esta operación permite considerar un archivo como una especie de arreglo grande, proporcionando así libertad para decidir cómo organizar el archivo. Pascal estándar no dispone del acceso directo a archivos, pero muchos dialectos de Pascal sí lo incluyen.

Otra operación útil es la que permite saber cuándo se llega al final del archivo. La detección del final del archivo se maneja de distintas maneras, según los diferentes lenguajes.

Se invierte mucho esfuerzo para evitar a los programadores la necesidad de tratar con las características físicas de los archivos, pero inevitablemente hay algunos detalles sobre su organización física que necesitan conocer. Cuando se pretende que un programa trabaje con archivos en un nivel muy bajo (como se intenta en gran medida en este texto), se debe estar siempre preparado para las pequeñas sorpresas intercaladas en los archivos por el sistema operativo que los organiza.

TERMINOS CLAVE

Archivo físico. Archivo que en realidad existe en el almacenamiento secundario. Es el archivo tal como lo conoce el sistema operativo y que aparece en su directorio de archivos.

Archivo lógico. El archivo, visto por el programa. El uso de archivos lógicos permite a un programa describir las operaciones

que van a efectuarse en un archivo sin saber cuál archivo físico real se usará. El programa puede entonces usarse para procesar cualquiera de diversos archivos que comparten la misma estructura.

CLOSE(). Función o llamada al sistema que rompe el enlace entre un nombre lógico de archivo y el nombre del archivo físico correspondiente.

CREATE(). Función o llamada al sistema para crear un archivo en almacenamiento secundario, que también puede unir un nombre lógico al nombre físico del archivo (véase OPEN()). Una llamada a CREATE() también genera información que es utilizada por el sistema para administrar el archivo, tal como la fecha de creación, la localización física y los privilegios de acceso a los usuarios previstos del archivo.

Distancia en bytes. Distancia, medida en bytes, a partir del principio del archivo. El primer byte del archivo tiene una distancia de 0, el segundo byte tiene un desplazamiento de 1, y así sucesivamente.

Manejo de buffers. Cuando la entrada y la salida se guardan, en lugar de enviarse de inmediato a su destino, se dice que se hace un *manejo de buffers*. En capítulos posteriores se encontrará que se puede mejorar drásticamente el desempeño de los programas que leen y escriben datos si manejan la E/S con buffers.

Modo de acceso. Tipo de acceso permitido al archivo. La variedad de modos de acceso varía de un sistema operativo a otro.

OPEN(). Función o llamada al sistema para lograr que un archivo esté listo para usarse. También puede enlazar un nombre lógico de archivo con un archivo físico. Sus argumentos incluyen el nombre del archivo lógico y el nombre del archivo físico, y también pueden incluir información sobre cómo realizar el acceso al archivo.

READ(). Función o llamada al sistema que se usa para obtener datos de un archivo o dispositivo. Visto desde un nivel inferior, requiere tres argumentos: 1) un nombre lógico de archivo_fuente que corresponda a un archivo abierto; 2) la dirección_destino de los bytes que se leerán y 3) el tamaño o cantidad de datos que se van a leer.

SEEK(). Función o llamada al sistema que coloca el apuntador de lectura y escritura en una posición específica en el archivo. Los lenguajes que proporcionan estas funciones de localización permiten a los programas tener acceso a elementos específicos de un archivo *directamente*, en lugar de leer todo el archivo desde el principio (en forma secuencial) cada vez que se desea un elemento específico. En C, la llamada al sistema *lseek()*

proporciona esta posibilidad. Pascal estándar no dispone de la capacidad de localización, pero sí la poseen muchos dialectos de Pascal que no son estándar.

WRITE(). Función o llamada al sistema usada para proporcionar capacidades de salida. Considerado desde el nivel inferior, requiere tres argumentos: 1) un nombre de archivo_destino correspondiente a un archivo abierto; 2) la dirección_fuente de los bytes que serán escritos y 3) el tamaño o cantidad de datos que se han de escribir.

EJERCICIOS

1. Investigue las operaciones equivalentes a *OPEN()*, *CLOSE()*, *CREATE()*, *READ()*, *WRITE()* y *SEEK()* en otros lenguajes de alto nivel, tales como PL/I, COBOL y Fortran. Compárelas con las versiones en C y en Pascal.

2. Si se usa C:

- a) Haga una lista de las diferentes formas de efectuar las operaciones *CREATE()*, *OPEN()*, *CLOSE()*, *READ()* y *WRITE()*. ¿Por qué hay más de una forma de hacer cada operación?
- b) ¿Cómo emplearía *lseek()* para encontrar la posición actual en un archivo?
- c) Describa una situación en la que se pueda usar un PMODE igual a 0644. Suponga un sistema operativo UNIX.
- d) ¿Cuál es la diferencia entre PMODE y MODO? ¿Qué tipo de PMODE y MODO existen en su sistema?
- e) En algunos ambientes C comunes, tales como UNIX y MS-DOS, lo que sigue a continuación representa maneras de trasladar los datos de un lugar a otro:

<code>scanf()</code>	<code>fgetc()</code>	<code>read()</code>	<code>cat (o type)</code>
<code>fscanf()</code>	<code>gets()</code>	<code><</code>	<code>main(argc, argv)</code>
<code>getc()</code>	<code>fgets()</code>	<code>:</code>	

Describa tantas como pueda, e indique cómo podrían ser útiles. ¿Cuáles pertenecen al lenguaje C, y cuáles al sistema operativo?

3. Si se usa Pascal:

- a) ¿Qué formas se proporcionan en su versión de Pascal para efectuar las operaciones de archivos CREATE(), OPEN(), CLOSE(), READ() y WRITE()? Si hay más de una forma de hacer cierta operación, diga por qué. Si una operación falta, ¿cómo pueden llevarse a cabo sus funciones?
- b) Realice una función SEEK() en su Pascal, si es que no existe ya una.

4. Hace un par de años una compañía conocida adquirió un nuevo compilador de COBOL. Una diferencia entre el compilador nuevo y el antiguo fue que el nuevo compilador no cerraba los archivos automáticamente cuando terminaba la ejecución de un programa, mientras que el antiguo sí lo hacía. ¿Qué clase de problemas causó esto cuando algo del software antiguo fue ejecutado después de haber sido recompilado por el compilador nuevo?

5. Observe los dos programas LISTA del texto. Cada uno tiene un ciclo *while*. En Pascal la secuencia de pasos en el ciclo es probar, leer y escribir. En C es leer, probar y escribir. ¿A qué se debe la diferencia? ¿Qué sucedería en Pascal si se usara la construcción de ciclo que se emplea en C? ¿Qué sucedería en C si se usara la construcción que se emplea en Pascal?

EJERCICIOS DE PROGRAMACION

6. Haga que el programa LISTA proporcionado en este capítulo trabaje con su compilador en su sistema operativo.

7. Escriba un programa que cree un archivo y almacene una cadena en él. Escriba otro programa que abra el archivo y lea la cadena.

8. Intente colocar el *modo de protección* de un archivo sólo a lectura y después abra el archivo con la modalidad de acceso de lectura y escritura. ¿Qué sucede?

9. Realice el programa *agrega.c* del texto, o escriba un programa en Pascal que lleve a efecto la misma función.

10. Escriba una función *final()* que presente en la pantalla la última parte de un archivo, a partir de una distancia en *n* bytes, donde *n* es relativo al inicio del archivo. Escriba un programa manejador que solicite al usuario el nombre del archivo y un valor para *n*. (En C, se

puede escribir *final()* como un programa principal que tome el nombre del archivo y a *n* como argumentos en la línea de mandatos.)

LECTURAS ADICIONALES

Los libros de texto introductorios sobre C y Pascal tienden a abordar las operaciones fundamentales de archivos en forma breve, cuando lo hacen. Esto sucede sobre todo con respecto a C, ya que en C hay funciones de E/S estándar de alto nivel, tales como las operaciones de lectura *fgets()* y *fgetc()*, que se supone son más fáciles de usar.

Algunos libros de C que dan un tratamiento completo (aunque breve) de las operaciones fundamentales de archivos son Bourne [1984], Kelley y Pohl [1984], Kernighan y Pike [1984], y Kernighan y Ritchie [1978]. Esos libros también proporcionan análisis de las funciones de E/S de alto nivel que se han omitido en este texto.

Como para Pascal esas operaciones varían mucho de una realización a otra, recomendamos consultar manuales para el usuario y literatura relacionada con la versión en particular. Cooper [1983] cubre el estándar ISO de Pascal así como algunas ampliaciones. Jensen y Wirth [1974] proporcionan la definición de Pascal en la cual se han basado todas las demás. Wirth [1975] analiza algunos problemas con el Pascal estándar y las operaciones con archivos en la sección "Un concepto importante y una fuente constante de problemas: los archivos".

EL PROGRAMA agrega EN PASCAL

COMENTARIOS: *agrega.pas*

El programa de Turbo Pascal llamado *agrega.pas* es funcionalmente similar al programa *agrega.c* expuesto en el capítulo 2. Ilustra el uso de la proposición *seek()* en Turbo Pascal. Algunos aspectos que hay que destacar acerca del programa son:

- Turbo Pascal maneja un tipo especial de cadena. Se decidió no usar ese tipo aquí para estar más apegados al Pascal estándar.
- Se emplea un ciclo WHILE para extraer uno a uno los caracteres del contenido de la cadena por agregar. Se necesita hacer esto para contar los caracteres de la cadena. Nótese que la función EOLN() se comporta de la misma forma que la función EOF() descrita en el capítulo 2: EOLN() se vuelve verdadera tan pronto como se lee el último carácter; en consecuencia, la proposición *read()* debe estar al principio del ciclo. Dicho de otra forma, las funciones EOLN() y EOF() se comportan de la misma manera en la entrada por teclado que en los archivos. En futuros programas en Pascal se incorpora este ciclo de lectura por teclado a una función llamada *lee_cad()*.
- El comentario {\$B-} es en realidad una directiva para el compilador de Turbo Pascal que le indica manejar la entrada por teclado como un archivo de Pascal estándar. Sin esta directiva no se manejaría apropiadamente la función EOLN() en el ciclo WHILE, que es el que recibe la entrada del teclado.
- En este capítulo se dijo que la proposición *assign()* no forma parte del Pascal estándar, sino que es una característica de Turbo Pascal. El enlace del archivo lógico con el nombre del archivo físico se realiza de distintas formas en diferentes ambientes de Pascal. De igual modo, la proposición *seek()* y la función *FileSize()* no son estándar; de nuevo, son características de Turbo Pascal. Cuando otras versiones de Pascal proporcionan tales funciones, suelen tener diferentes nombres.

```
PROGRAM agrega (INPUT, OUTPUT);
{
    Lee una cadena de entrada para agregarla al final de un
    archivo, y usa la proposición seek() para saltar al final.
}
{$B-} { Directiva especial para el compilador de Turbo Pascal, para que maneje
        la entrada por teclado en forma estándar; véanse los comentarios
        asociados en el programa }

VAR
    car          : char;
    i, long_dato : integer;
    resp         : char;
    nomarchivo   : packed array [1..20] of char;
    dato_nuevo   : packed array [1..80] of char;
    archsalida   : file of char;

BEGIN {principal}
    write('Proporcione el nombre del archivo: ');
    readln(nomarchivo);
    assign(archsalida,nomarchivo);

    write('¿Este archivo ya existe? (responda S o N): ');
    readln(resp);
    if (resp = 'S') or (resp = 's') then
        reset(archsalida);
    else
        rewrite(archsalida);

    { Toma el dato nuevo del teclado, manteniendo cuenta del número de
      caracteres proporcionados
    }

    write('Cadena que se agregará: ');
    long_dato := 0;
    while (not EOLN) and (long_dato < 80) DO
    BEGIN
        read(car);
        long_dato := long_dato + 1;
        dato_nuevo[long_dato] := car;
    END;

    {Localiza el final del archivo y escribe el dato ahí }
    seek(archsalida, FileSize(archsalida));
    for i := 1 to long_dato DO
        write(archsalida,dato_nuevo[i]);

    close(archsalida)
END.
```

OBJETIVOS

Describir la organización de las unidades de disco típicas, así como las unidades básicas de organización (sectores, bloques, pistas y cilindros) y sus relaciones.

Identificar y describir los factores que afectan el tiempo de acceso al disco, y describir métodos para la estimación de tiempos de acceso y requerimientos de espacio.

Describir las cintas magnéticas, identificar algunas de sus aplicaciones e investigar las implicaciones del uso de distintos tamaños de bloques en los requerimientos de espacio y las velocidades de transmisión.

Identificar otros medios de almacenamiento secundario e indicar los papeles que desempeñan en la jerarquía de los sistemas de memoria.

Describir, en términos generales, los eventos que ocurren cuando los datos son transmitidos entre el área de datos de un programa y un dispositivo secundario.

Introducir conceptos y técnicas de manejo de buffers.

3

DISPOSITIVOS DE ALMACENAMIENTO SECUNDARIO Y SOFTWARE DE SISTEMAS: CONSIDERACIONES DE DESEMPEÑO

PLAN GENERAL DEL CAPITULO

3.1 Discos

- 3.1.1 La organización de los discos
- 3.1.2 Estimación de las capacidades y necesidades de espacio
- 3.1.3 Organización por sectores
- 3.1.4 Organización por bloques
- 3.1.5 Sobrecarga por datos usados para control
- 3.1.6 El costo de un acceso a disco

3.2 Cinta magnética

- 3.2.1 Organización de datos en cintas
- 3.2.2 Estimación de requerimientos de longitud de cinta
- 3.2.3 Estimación de los tiempos de transmisión de datos
- 3.2.4 Aplicaciones de las cintas

3.3 Otros tipos de almacenamiento

3.4 El almacenamiento como una jerarquía

3.5 El viaje de un byte

- 3.5.1 El administrador de archivos
- 3.5.2 El buffer de E/S
- 3.5.3 El byte sale de la memoria RAM: el procesador de E/S
- 3.5.4 El byte llega al disco: el controlador del disco

3.6 Manejo de buffers

El buen diseño responde siempre al ambiente y a las restricciones del medio. Esto es tan cierto para el diseño de estructuras de archivos como para el diseño de construcciones en madera y piedra. Dada la capacidad para crear, abrir y cerrar archivos, y para localizar, leer y escribir, se pueden realizar las operaciones fundamentales de la *construcción* de archivos. Ahora es necesario examinar la naturaleza y limitaciones de los dispositivos y sistemas empleados para almacenar y extraer archivos, con objeto de estar preparados para el *diseño* de archivos.

Si los archivos fueran almacenados sólo en memoria RAM, no habría un campo llamado estructuras de archivos. El estudio general de las estructuras de datos proporcionaría todas las herramientas necesarias para construir aplicaciones con archivos. Pero los dispositivos de almacenamiento secundario son muy diferentes de los de memoria RAM. Una diferencia que ya se ha señalado es que los accesos al

almacenamiento secundario toman mucho más tiempo que los accesos a memoria RAM. Una diferencia de mayor importancia aún, medida en términos del impacto del diseño, es que no todos los accesos son iguales. El buen diseño de estructuras de archivos toma en cuenta el desempeño del disco y de la cinta, para disponer los datos de manera que minimicen los costos de acceso.

En este capítulo se examinan las características de los dispositivos de almacenamiento secundario, y en los capítulos siguientes nos ocuparemos de las restricciones que determinan el trabajo de diseño. Para empezar se hará un repaso de los principales medios usados en el almacenamiento y procesamiento de archivos, los discos magnéticos y las cintas. En seguida se verán los alcances de otros dispositivos y medios empleados para almacenamiento secundario. Posteriormente se recorrerá el trayecto de un byte para conocer los elementos de hardware y software implicados cuando un programa envía un byte a un archivo en disco. Por último, se examinará con mayor detalle uno de los aspectos más importantes del manejo de archivos: el manejo de buffers.

3.1

DISCOS

Comparados con el tiempo que toma acceder a un dato en memoria RAM, los accesos al disco siempre son caros. Sin embargo, no todos los accesos a disco son *igualmente* caros. La diferencia de costos radica en la manera como trabaja una unidad de disco. Las unidades de disco[†] pertenecen a una clase de dispositivos conocidos como *dispositivos de almacenamiento de acceso directo* (DAAD) ya que permiten el acceso *directo* a los datos. Los DAAD contrastan con los *dispositivos de acceso en serie*, el otro tipo principal de dispositivos de almacenamiento secundario. Los dispositivos de acceso en serie usan medios tales como la cinta magnética que sólo permite el acceso en serie; no se puede leer o escribir un dato en particular hasta que todos los datos que lo preceden en la cinta hayan sido leídos o escritos en orden.

3.1.1 LA ORGANIZACION DE LOS DISCOS

La información almacenada en un disco se guarda sobre la superficie de uno o más platos (Fig. 3.1). La disposición es tal que la información es

[†] Cuando se emplean los términos discos o unidades de disco, se está haciendo referencia a medios de disco *magnéticos*. Los medios de disco no magnéticos, especialmente los discos ópticos, cobran cada vez mayor importancia para el almacenamiento secundario.

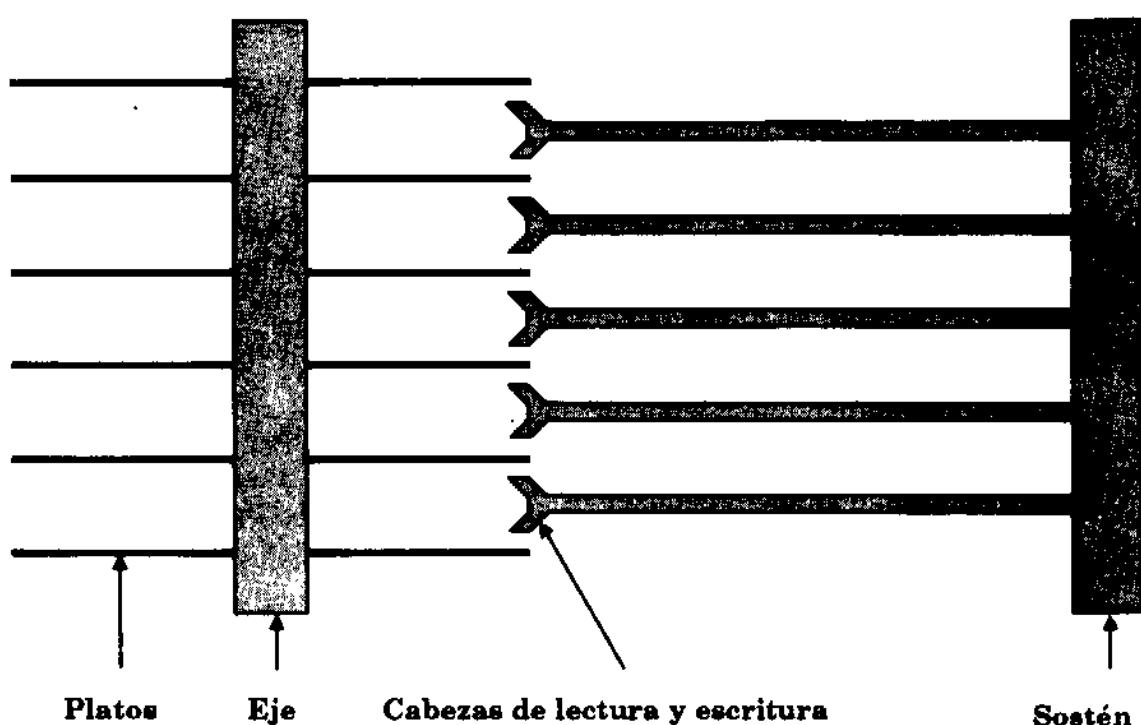


FIGURA 3.1 • Ilustración esquemática de una unidad de disco con seis platos en un eje y cinco brazos de acceso móviles con las cabezas de lectura y escritura. Hay diez cabezas de lectura y escritura, una por superficie, de tal modo que pueden leerse diez pistas diferentes sin tener que mover el brazo de acceso. Estas diez pistas constituyen un cilindro (véase la Fig. 3.3). (Tomado de Harvey Deitel, *Introducción a los sistemas operativos*, ©1987, Addison-Wesley Iberoamericana, Wilmington, Delaware, pág. 305, Fig. 12.1. Reimpresa con permiso.)

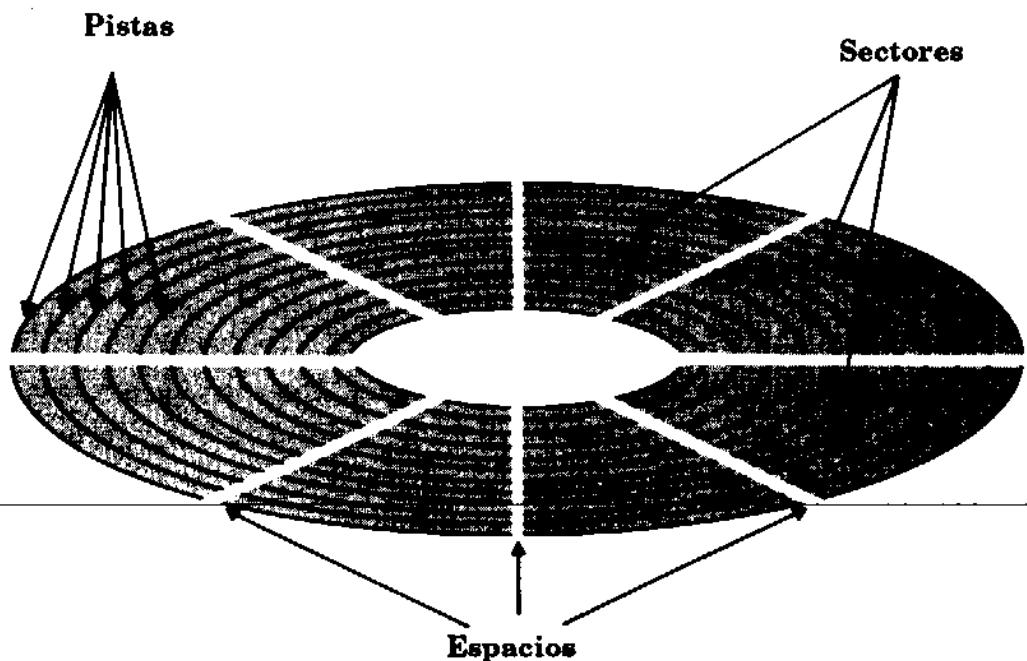


FIGURA 3.2 • Superficie de un disco que muestra pistas y sectores. Este disco tiene doce pistas y ocho sectores por pista. Por lo común cada sector contiene la misma cantidad de datos, de modo que las pistas interiores contienen más bits por pulgada que las exteriores. Los espacios entre sectores ayudan al dispositivo que lee el disco a distinguir los sectores.

almacenada en *pistas* sucesivas en la superficie del disco (Fig. 3.2). Por lo común, cada pista se divide en varios *sectores*. Un sector es la porción referenciable más pequeña de un disco. Cuando una proposición *READY* pide un byte en particular de un archivo en disco, el sistema operativo del computador encuentra la superficie, pista y sector correctos, lee el sector completo y lo pone en un área especial en memoria RAM llamada *buffer*, y después encuentra dentro de ese buffer el byte solicitado.

Si una unidad de disco usa varios platos, puede llamarlese *paquete de discos*. Las pistas que están directamente unas sobre otras forman un *cilindro* (Fig. 3.3). La importancia del cilindro es que se puede tener el acceso a toda la información almacenada en uno sin mover el brazo

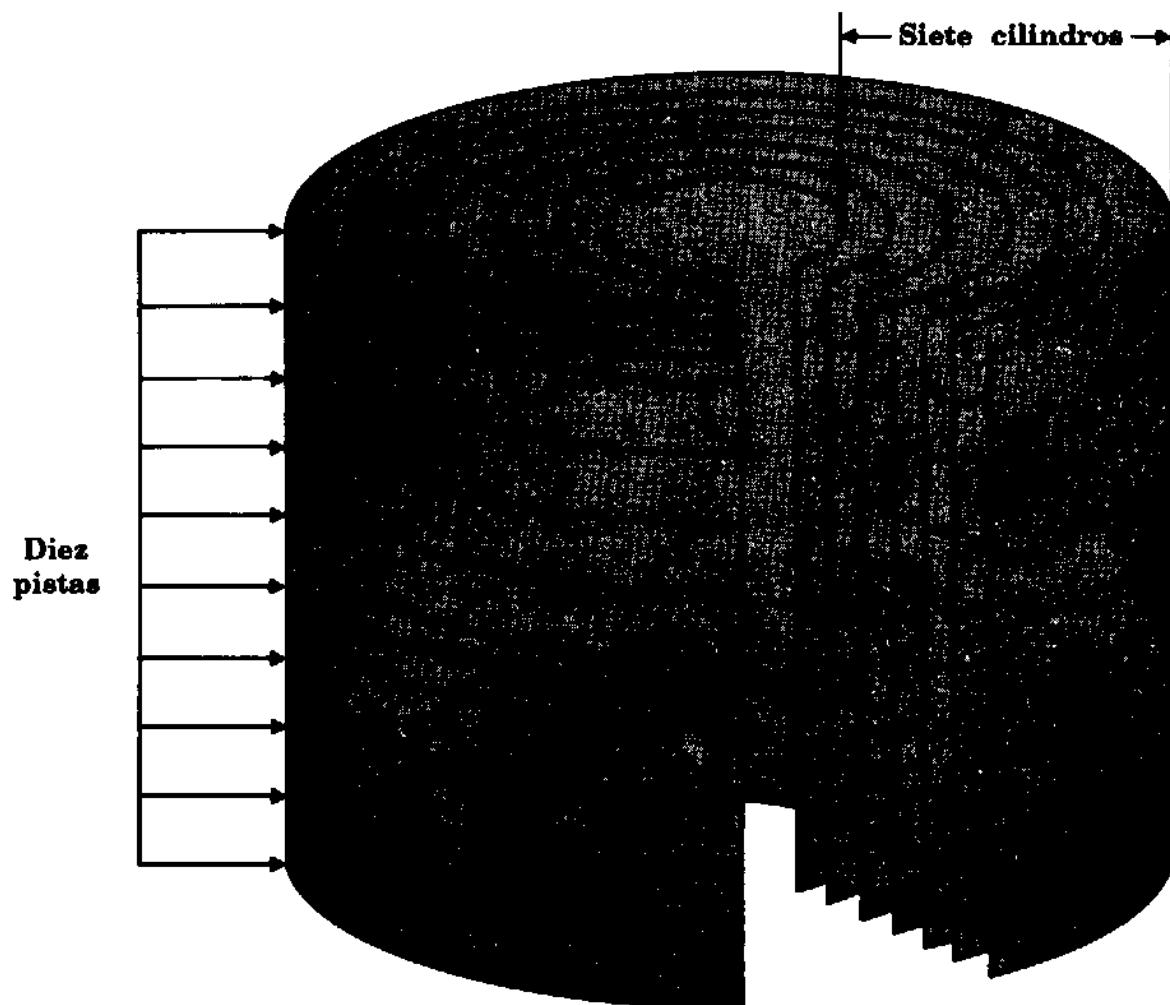


FIGURA 3.3 • Ilustración esquemática de una unidad de disco vista como un conjunto de siete cilindros, con diez pistas por cilindro. Un cilindro es un conjunto de pistas, a las cuales se puede tener acceso sin mover el brazo que contiene las cabezas de lectura y escritura. Físicamente, cada una de las pistas que componen un cilindro está en una superficie diferente.

que sostiene las cabezas de lectura y escritura. El movimiento de este brazo se llama *desplazamiento*. Este movimiento del brazo suele ser la parte más lenta de la lectura de información en un disco.

3.1.2 ESTIMACION DE LAS CAPACIDADES Y NECESIDADES DE ESPACIO

El ancho de los discos varía desde 3 hasta alrededor de 14 pulgadas, y su capacidad de almacenamiento varía de menos de 100 000 bytes a miles de millones de bytes. En un paquete de discos normal, los platos superior e inferior contribuyen con una superficie cada uno para formar el paquete y los demás platos contribuyen con dos superficies, de tal forma que el número de pistas por cilindro está en función del número de platos.

La cantidad de datos que pueden guardarse en una pista depende de la densidad con que puedan almacenarse los bits en la superficie del disco. (Esto, a su vez, depende de la calidad del medio de grabación y del tamaño de las cabezas de lectura y escritura.) Un disco barato, de baja densidad, puede guardar alrededor de 4 kilobytes en una pista y tiene 35 pistas en una superficie, mientras que un disco de la mejor calidad puede almacenar cerca de 50 kilobytes por pista, y más de 1000 pistas en una superficie.

Puesto que un cilindro consiste en un grupo de pistas, una pista consiste en un grupo de sectores, y un sector en un grupo de bytes, es fácil calcular las capacidades de pistas, cilindros y unidades de discos:

$$\text{Capacidad de la pista} = \text{número de sectores por pista} \times \text{bytes por sector}$$

$$\text{Capacidad del cilindro} = \text{número de pistas por cilindro} \times \text{capacidad de la pista}$$

$$\text{Capacidad de la unidad} = \text{número de cilindros} \times \text{capacidad del cilindro}$$

Si se conoce el número de bytes en un archivo, se pueden usar esas relaciones para calcular el espacio del disco que el archivo requiere. Por ejemplo, suponga que se desea almacenar un archivo con 20 000 registros de datos de longitud fija en un disco que tenga las siguientes características:

$$\text{Número de bytes por sector} = 512$$

$$\text{Número de sectores por pista} = 40$$

$$\text{Número de pistas por cilindro} = 11$$

¿Cuántos cilindros requiere el archivo, si cada registro de datos necesita 256 bytes? Si cada sector puede almacenar dos registros, el archivo requiere

$$\frac{20\,000}{2} = 10\,000 \text{ sectores.}$$

Un cilindro puede almacenar

$$40 \times 11 = 440 \text{ sectores}$$

de modo que el número de cilindros necesarios es de alrededor de:

$$\frac{10\,000}{440} = 22.7 \text{ cilindros}$$

Por supuesto, puede ser que en una unidad de disco con 22.7 cilindros de espacio disponible no haya 22.7 cilindros *físicamente contiguos*. En este caso, que es muy común, el archivo puede esparcirse sobre incluso cientos de cilindros.

3.1.3 ORGANIZACION POR SECTORES

Hay dos formas básicas para organizar los datos en un disco: por sector y por bloques definidos por el usuario. Hasta ahora, sólo se han mencionado las organizaciones por sectores; en esta sección se examinan con mayor detalle, y en la siguiente se estudiarán las organizaciones por bloques.

LA DISPOSICION FISICA DE LOS SECTORES. Hay diversos puntos de vista respecto a la organización de los sectores en una pista. El más sencillo, que casi siempre es suficiente para la mayoría de los usuarios, propone que los sectores sean segmentos de pista adyacentes de tamaño fijo, capaces de contener un archivo (Fig. 3.4a). Esta es, con frecuencia, una forma muy adecuada de visualizar un archivo *lógicamente*, pero no es conveniente para almacenar los sectores *físicamente*.

Resulta que cuando se quiere leer una serie de sectores que se encuentran en la misma pista, uno después de otro, por lo general no se puede leer sectores *adyacentes*. Esto se debe a que, después de leer los datos, el computador necesita tiempo para procesar la información recibida antes de poder aceptar más. Entonces, si los sectores que son

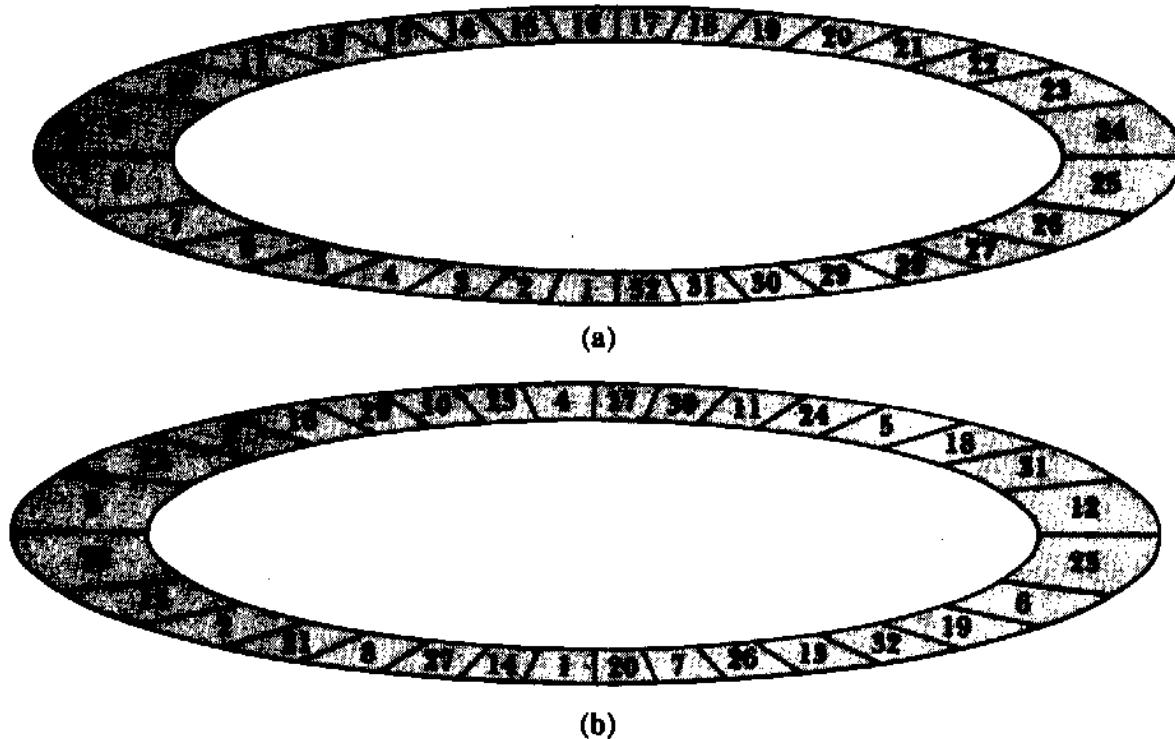


FIGURA 3.4 • Dos aspectos de la organización de sectores en una pista de 32. (a) La forma más simple de la organización de sectores en una pista. Los sectores se ven como segmentos de pista adyacentes, de tamaño fijo, y que resultan capaces de almacenar un archivo. (b) Intercalación de sectores, con un factor de intercalación de 5. Los sectores lógicamente adyacentes están a intervalos de cinco sectores físicos.

adyacentes *lógicamente* estuvieran colocados en el disco de modo que también fueran adyacentes *físicamente*, se perdería el inicio del siguiente sector mientras se procesa el que se acaba de leer. Por lo tanto, sólo se podría leer un sector por cada revolución del disco.

Para solucionar este problema, los diseñadores de los sistemas de E/S suelen *intercalar* los sectores, dejando un intervalo de varios sectores físicos entre los sectores lógicamente adyacentes. Suponga que un disco tiene un *factor de intercalación* de 5. La transferencia del contenido de sectores lógicos a los 32 sectores físicos en una pista se ilustra en la figura 3.4(b). Si se estudia este diagrama, puede observarse que requiere cinco revoluciones para leer completamente los 32 sectores de una pista. Esto representa una gran mejora, si se compara con la necesidad de esperar 32 revoluciones.

CUMULOS. Una tercera forma de entender las organizaciones por sectores, diseñada también para mejorar el desempeño, está determi-

nada por la parte del sistema operativo de un computador que se denomina *administrador de archivos*. Cuando un programa accede a un archivo, el administrador de archivos hace la correspondencia entre las partes lógicas del archivo y sus posiciones físicas. Para ello considera el archivo como una serie de *cúmulos* de sectores. Un cúmulo es un número fijo de sectores contiguos,[†] de suerte que todos los cúmulos de un disco son del mismo tamaño, y una vez que se ha encontrado un cúmulo en particular en el disco, se puede acceder a todos los sectores de ese cúmulo, sin requerir una búsqueda adicional.

Para ver un archivo como una serie de cúmulos y aun así mantener el punto de vista de sectores, el administrador de archivos une los sectores lógicos con los cúmulos físicos a los que pertenecen; para ello emplea una *tabla de asignación de archivos* (FAT, por sus siglas en inglés). La tabla FAT contiene una lista ligada de todos los cúmulos de un archivo, ordenados de acuerdo con el orden lógico de los sectores que contienen. Cada entrada de un cúmulo en la tabla FAT proporciona información sobre la posición física del cúmulo (Fig. 3.5).

**Tabla de asignación de archivos
(FAT)**

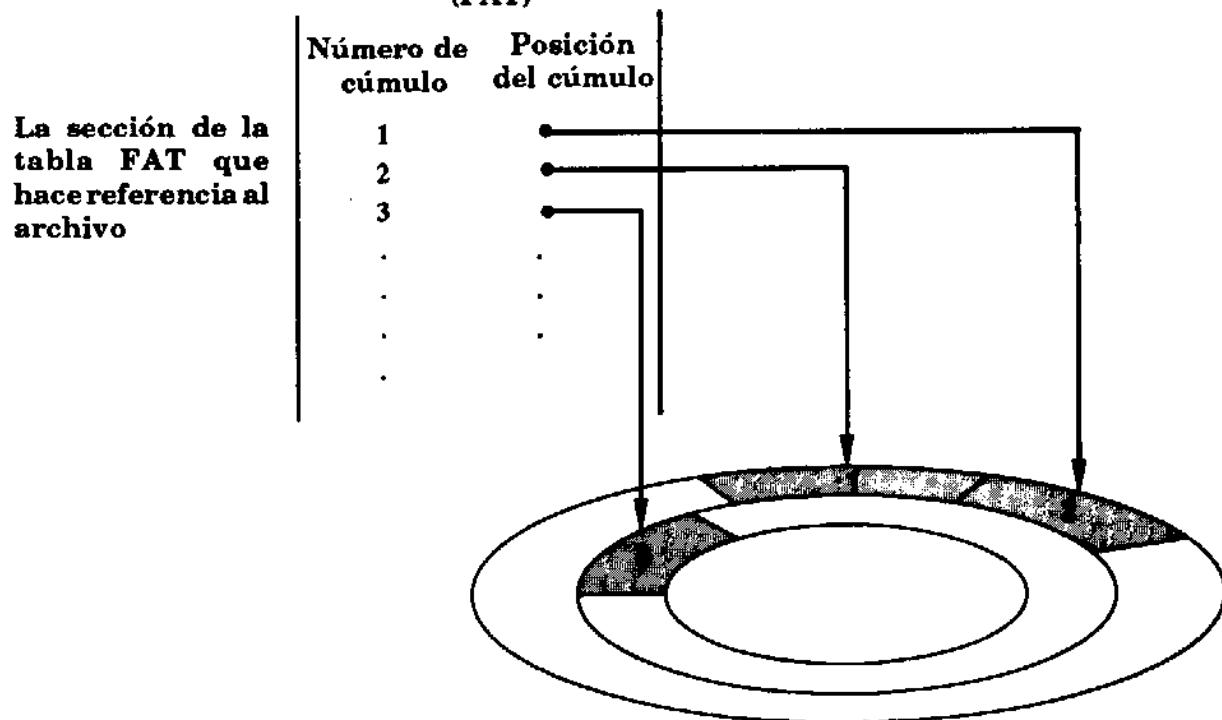


FIGURA 3.5 • El administrador de archivos determina cuál cúmulo del archivo contiene el sector al que se accederá. La tabla FAT dice al administrador dónde está colocado físicamente el cúmulo en el disco. La tabla FAT puede tener datos de muchos archivos, pero aquí sólo se muestran los de uno.

[†]No es del todo *físicamente* contiguo; el grado de contigüidad física está determinado por el factor de intercalación.

En muchos sistemas, el administrador del sistema puede determinar cuántos sectores habrá en un cùmulo. Por ejemplo, en la estructura física del disco estándar empleado por los sistemas VAX, el administrador determina el tamaño del cùmulo por emplear en un disco cuando se le asignan valores iniciales. El valor por omisión es de tres sectores de 512 bytes por cùmulo, pero el tamaño del cùmulo puede ser cualquier valor entre 1 y 65 535 sectores. Como los cùmulos representan grupos de sectores físicamente contiguos, los cùmulos más grandes garantizan la posibilidad de leer más sectores sin mover el brazo del disco, de tal forma que el uso de cùmulos grandes redunda en ganancias considerables en el desempeño cuando un archivo se procesa secuencialmente.

EXTENSIONES. El último punto de vista que se presenta sobre la organización por sectores subraya una vez más la importancia de la contigüidad física de los sectores en un archivo, minimizando por tanto aún más el desplazamiento del brazo del disco. (Si el lector piensa que evitar esto es un aspecto importante del diseño de archivos, está en lo correcto.) Si un disco cuenta con mucho espacio disponible, sería posible que se constituyera un archivo completo de cùmulos contiguos. Cuando esto sucede, se dice que el archivo se compone de una *extensión*: todos sus sectores, pistas y (si es suficientemente grande) cilindros forman un todo contiguo (Fig. 3.6a). Esto es muy conveniente, sobre todo si el archivo se procesa secuencialmente, pues significa que se puede tener acceso al archivo completo con un número mínimo de desplazamientos.

Si no hay suficiente espacio contiguo disponible para contener el archivo completo, éste se divide en dos o más partes separadas. Cada una de las partes es una extensión. Cuando se agregan nuevos cùmulos a un archivo, el administrador de archivos intenta ponerlos físicamente contiguos al final del archivo previo, pero si no hay espacio disponible para esto, deben agregarse una o más extensiones (Fig. 3.6b). Lo que es más importante comprender de las extensiones es que conforme se incrementa el número de ellas en un archivo, éste se disemina más sobre el disco y aumenta el número de desplazamientos requeridos para el procesamiento del archivo.

FAGMENTACION. En general, todos los sectores de una unidad de disco determinada deben contener el mismo número de bytes. Si, por ejemplo, el tamaño de un sector es de 512 bytes y el de todos los registros en un archivo es de 300 bytes, no hay una correspondencia conveniente entre registros y sectores. Existen dos formas de enfrentar esta situación: almacenar sólo un registro por sector, o permitir que los registros se *traslapen entre sectores*, de modo que el principio de un registro pueda encontrarse en un sector y el final en otro (Fig. 3.7).

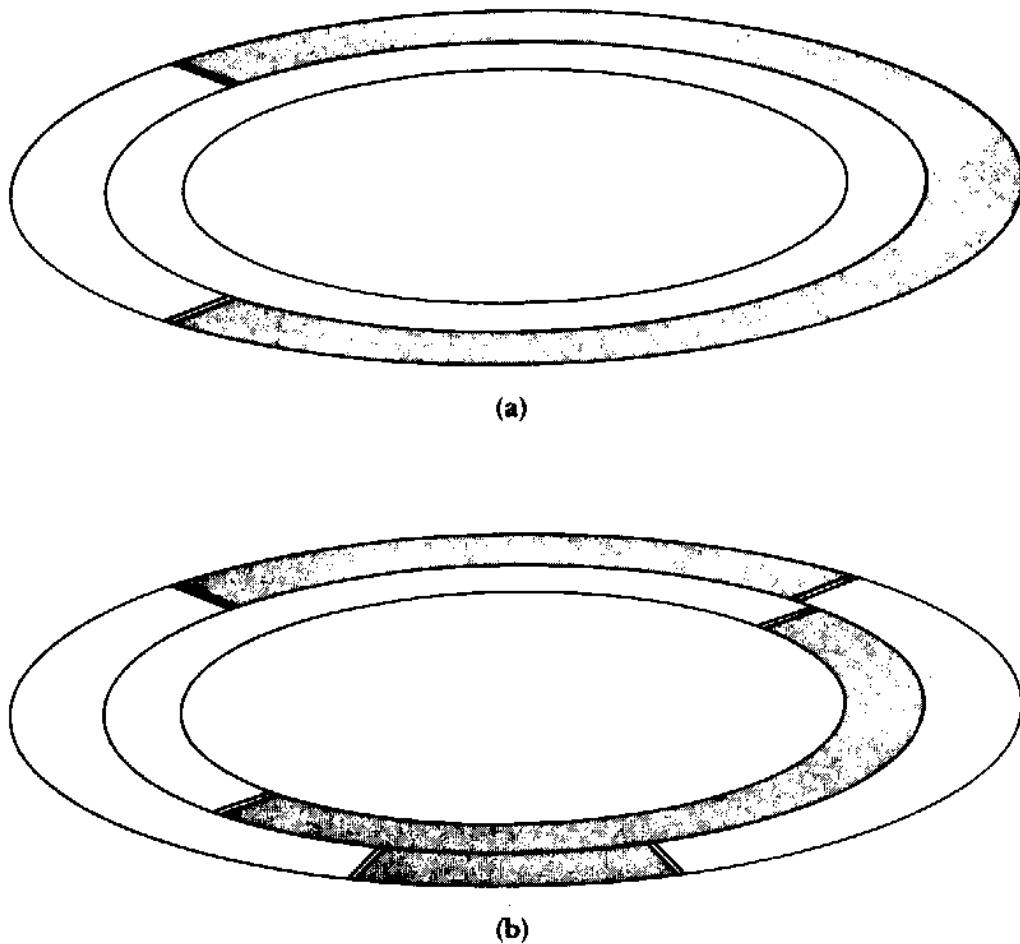


FIGURA 3.6 • Extensiones de archivos (las áreas sombreadas representan el espacio del disco usado por un archivo). (a) El archivo completo almacenado en una extensión. (b) El archivo dividido en varias extensiones.

La primera opción tiene la ventaja de que cualquier registro puede extraerse con sólo recuperar un sector, pero tiene la desventaja de que puede quedar sin uso una cantidad enorme de espacio dentro de cada sector. A esta pérdida de espacio dentro de un sector se le llama *fragmentación interna*. La segunda opción tiene la ventaja de no perder espacio por fragmentación interna, pero la desventaja es que algunos registros sólo pueden extraerse mediante el acceso a dos sectores.

Otra fuente posible de fragmentación interna surge del empleo de cúmulos. Recordemos que un cúmulo es la unidad de espacio más pequeña que puede asignarse para un archivo. Cuando el número de bytes en un archivo no es un múltiplo exacto del tamaño del cúmulo, habrá fragmentación interna en la última extensión del archivo. Por ejemplo, si un cúmulo consiste en tres sectores de 512 bytes, un archivo que contuviera un byte usaría 1536 bytes en el disco, y 1535 bytes se desperdiciarían debido a la fragmentación interna.

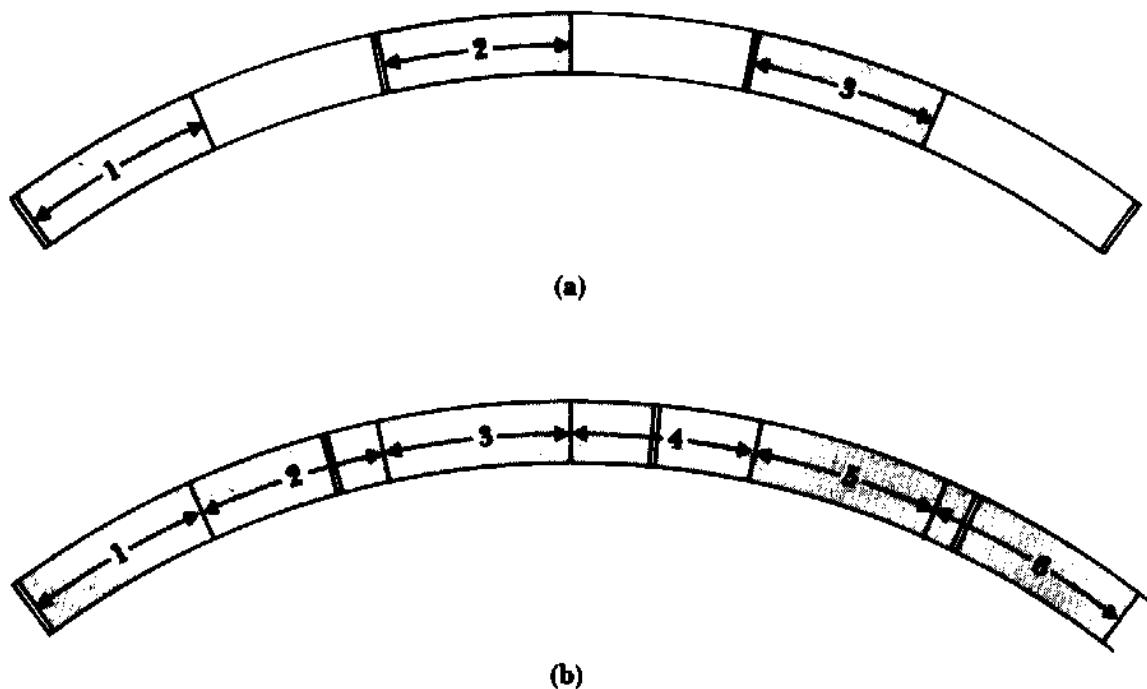


FIGURA 3.7 • Organización alterna de registros dentro de sectores (las áreas sombreadas representan registros de datos, y las áreas en blanco representan espacio libre). (a) Un registro almacenado por sector, lo cual provoca *fragmentación*. (b) Los registros se traslanan entre sectores, eliminando la fragmentación, pero tal vez requieran más de un acceso por registro.

Está claro que hay compromisos importantes en el uso de cúmulos de gran tamaño. Si se espera tener en un disco principalmente archivos grandes que serán procesados en forma secuencial, se le dará normalmente un tamaño grande de cúmulo, ya que la fragmentación interna no sería un gran problema, y la ganancia en el desempeño podrá ser considerable. A un disco que tenga archivos más pequeños, o archivos a los cuales normalmente se accede sólo de manera aleatoria, se le darán cúmulos pequeños.

3.1.4 ORGANIZACION POR BLOQUES

Algunas veces las pistas de discos no están divididas en sectores, sino en números enteros de *bloques* que el usuario define y cuyos tamaños pueden variar. Por lo común, cuando los datos de una pista están organizados por bloques, la cantidad de datos transferida en una

operación de E/S puede variar, dependiendo de las necesidades del diseñador de software, no del hardware. Los bloques pueden tener una longitud fija o variable, dependiendo de los requerimientos del diseñador de archivos. Como sucede con los sectores, los bloques se designan frecuentemente como registros físicos. (Algunas veces la palabra "bloque" se usa como sinónimo de un sector o grupo de sectores. Para evitar confusión, aquí no se emplea en esa forma.) La figura 3.8 ilustra la diferencia entre cómo aparecen los datos de una pista en sectores y los datos de una pista en bloques.

Una organización en *bloques* no presenta los problemas de distribución de sectores y de fragmentación, porque el tamaño de los bloques puede variar para ajustarse a la organización lógica de los datos. Por lo común, un bloque está organizado para almacenar un número entero de registros lógicos. El término *factor de bloque* se emplea para indicar el número de registros que se almacenan en cada bloque en un archivo. Por lo tanto, si se tiene un archivo con registros de 300 bytes, un esquema de referencias a direcciones por bloques permitiría definir un bloque según un múltiplo apropiado de 300 bytes, dependiendo de las necesidades del programa. No se desperdiciaría espacio por fragmentación interna y no habría necesidad de cargar dos bloques para extraer un registro.

En general, los bloques son superiores a los sectores cuando se pretende que la asignación física del espacio para registros corresponda con su organización lógica. (Existen unidades de disco que permiten la referencia a direcciones tanto por sectores como por bloques, pero no se describirán aquí. Véase Bohl[1981].)

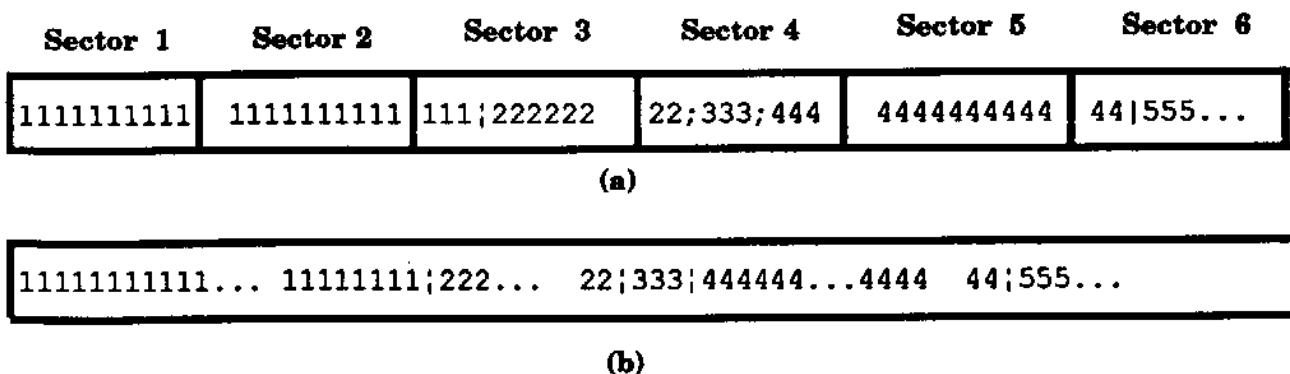


FIGURA 3.8 - Organización por sectores en comparación con la organización por bloques. (a) Organización por sectores: sin importar el tamaño de un registro u otro agrupamiento lógico de los datos, los datos están agrupados físicamente en sectores. Todo acceso implica la transmisión de un número entero de sectores. (b) Organización por bloques: la cantidad de datos transmitidos en un acceso depende del tamaño de los bloques.

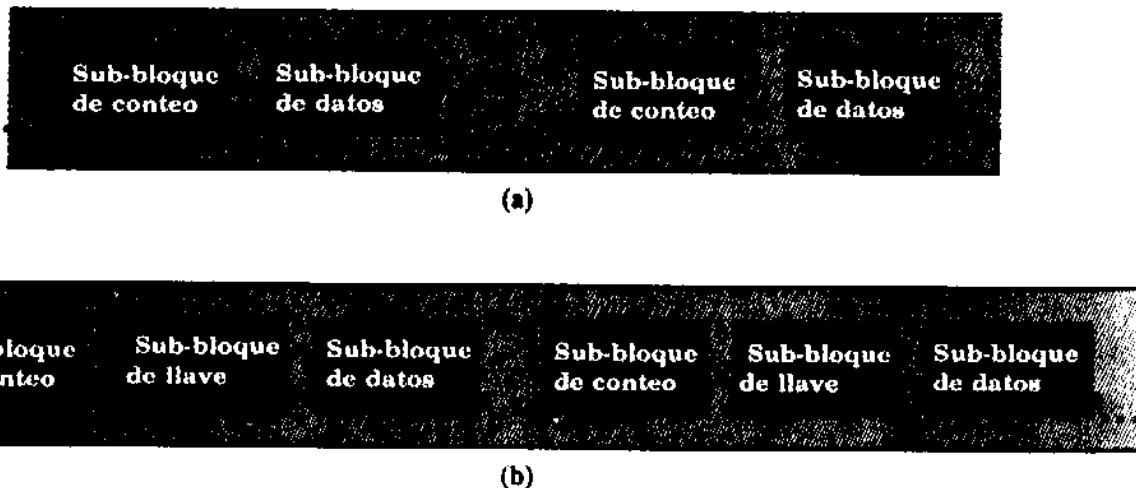


FIGURA 3.9 • La referencia a direcciones por bloques requiere que cada bloque físico de datos sea acompañado por uno o más sub-bloques que contienen información respecto a su contenido. Los *sub-bloques de conteo* dicen cuántos bytes de datos hay en el bloque de datos correspondiente. Los *sub-bloques de llave* indican la llave del último registro del bloque de datos correspondiente. (a) Bloques con sub-bloques de conteo y de datos. (b) Bloques con sub-bloques de conteo, de llave y de datos.

En los esquemas de referencia a direcciones por bloques, a cada bloque de datos lo acompaña normalmente uno o más *sub-bloques* adicionales que contienen información suplementaria acerca del bloque de datos. Por lo general hay un *sub-bloque de conteo* que contiene (entre otras cosas) el número de bytes del bloque de datos al que acompaña (Fig. 3.9a). También puede haber un *sub-bloque de llave* que contiene la llave del último registro del bloque (Fig. 3.9b). Cuando se usan los sub-bloques de *llave*, el controlador del disco puede buscar dentro de una pista un bloque o registro identificado por una llave específica. Esto significa que un programa puede pedir a su unidad de disco que busque, entre todos los bloques de una pista, uno con una determinada llave. Este enfoque permite búsquedas mucho más eficientes de lo que sería posible con esquemas de referencia a direcciones por sectores, donde a menudo las llaves no se pueden interpretar si no se cargan antes en la memoria primaria.

3.1.5 SOBRECARGA POR DATOS USADOS PARA CONTROL

Tanto los bloques como los sectores ocupan una cierta cantidad de espacio en el disco en forma de *sobrecarga por datos de control*. Parte de la sobrecarga consiste en información que se almacena en el disco al *dar formato previo*, el cual se realiza antes de usar el disco.

En discos con referencia a direcciones por sector, el dar formato previo implica almacenar, al principio de cada sector, información respecto a la dirección del sector, dirección de la pista y estado (si el sector es usable o está defectuoso). Dar formato previo también implica la colocación de espacios y marcas de sincronización entre campos de información, para ayudar a los mecanismos de lectura y escritura a distinguirlos. Esta sobrecarga por datos de control normalmente no le preocupa al programador. Cuando el tamaño del sector está dado para cierta unidad de disco, el programador puede suponer que ésta es la cantidad de datos reales que pueden ser almacenados en un sector.

En un disco organizado por bloques, parte de la sobrecarga por datos de control es invisible para el programador, aunque otra parte sí debe tomarla en cuenta. Como los sub-bloques y los espacios entre bloques deben proporcionarse para cada bloque, generalmente es necesaria más información de control para los bloques que para los sectores. Además, como el número y el tamaño de los bloques varía de una aplicación a otra, la cantidad relativa de espacio que ocupa la sobrecarga puede variar cuando se usa la referencia a direcciones por bloques. Esto se ilustra en el siguiente ejemplo.

Supongamos que se tiene una unidad de disco con referencia a direcciones por bloques con 20 000 bytes por pista, y que la cantidad de espacio dedicada a los sub-bloques y los huecos entre bloques es equivalente a 300 bytes por bloque. Si se quiere almacenar en el disco un archivo que contiene registros de 100 bytes, ¿cuántos registros pueden almacenarse por pista si el factor de bloque es de 10, o si es de 60?

1. Si hay 10 registros de 100 bytes por bloque, y cada bloque almacena 1000 bytes de datos y emplea 300 + 1000, o 1300 bytes de espacio de la pista, tomando en cuenta la sobrecarga por datos de control. El número de bloques que pueden entrar en una pista de 20 000 bytes se puede expresar como

$$\left\lfloor \frac{20\,000}{1300} \right\rfloor = \underline{\underline{15.38}} = 15$$

De modo que se pueden almacenar 15 bloques, o 150 registros, por pista. (Nótese que se tiene que tomar *el redondeo hacia abajo* del resultado, debido a que un bloque no puede distribuirse en dos pistas.)

2. Si hay 60 registros de 100 bytes por bloque, cada bloque almacena 6000 bytes de datos y usa 6300 bytes de espacio de la pista. El número de bloques por pista puede expresarse como

$$\left\lfloor \frac{20\,000}{6300} \right\rfloor = 3$$

De manera que pueden almacenarse 3 bloques, o 180 registros, por pista.

Está claro que el factor de bloque mayor puede dar un uso más eficiente al almacenamiento. Cuando los bloques son grandes se requieren menos para contener un archivo, de tal forma que es menor el espacio consumido por los 300 bytes de sobrecarga que acompañan a cada bloque.

¿Se puede concluir de este ejemplo que los factores de bloque grandes siempre conducen a una utilización más eficiente del almacenamiento? No necesariamente. Como sólo se puede poner un número entero de bloques en una pista, y las pistas tienen una longitud fija, casi siempre se pierde espacio al final de una pista. Aquí se presenta de nuevo el problema de la fragmentación interna, pero esta vez se aplica a la fragmentación dentro de una *pista*. A mayor tamaño de bloque, mayor cantidad potencial de fragmentación interna de la pista. ¿Qué sucedería si se eligiera un factor de bloque de 98 en el ejemplo anterior? ¿Qué sucedería si fuera de 97?

La flexibilidad obtenida con el uso de bloques, en lugar de sectores, permite ahorrar tiempo y ganar eficiencia, ya que el programador determina casi por completo la organización física de los datos en un disco. El aspecto negativo de los esquemas por bloques es que *requieren* trabajo adicional del programador, del sistema operativo, o de ambos, para determinar la organización de los datos. Además, la flexibilidad que introduce el uso de esquemas por bloques también impide la sincronización de las operaciones de E/S con el movimiento físico del disco; que el manejo por sectores sí permite. Esto significa que no pueden usarse estrategias tales como la intercalación de sectores para mejorar el desempeño.

3.1.6 EL COSTO DE UN ACCESO A DISCO

Para dar una idea de los factores que intervienen en la suma total del tiempo necesario para acceder a un archivo que se encuentra en un disco fijo, se calcularán algunos tiempos de acceso. Un acceso a disco se puede dividir en tres operaciones físicas distintas, cada una de las cuales tiene un costo propio: *tiempo de desplazamiento*, *retraso por rotación* y *tiempo de transferencia*.

TIEMPO DE DESPLAZAMIENTO. El tiempo de desplazamiento es el tiempo requerido para mover el brazo de acceso hasta el cilindro ade-

cuando. El tiempo utilizado para el desplazamiento depende, por supuesto, de la distancia que tenga que recorrer el brazo. Si se accede en forma secuencial a un archivo y éste está comprimido en varios cilindros consecutivos, el desplazamiento se realiza sólo después de que todas las pistas del cilindro han sido procesadas, y aun así la cabeza de lectura y escritura necesitará moverse a lo ancho de una pista tan sólo. Si, por el contrario, se accede en forma alterna a sectores de dos archivos que están almacenados en los extremos opuestos de un disco (uno en el cilindro más interno, y otro en el más externo), el desplazamiento resulta muy caro.

El desplazamiento suele ser más costoso cuando hay varios usuarios —situación en la cual diversos procesos compiten por el uso del disco— que cuando hay un único usuario, ya que en este caso el uso del disco está dedicado a un solo proceso. Debido al alto costo del desplazamiento, los diseñadores de sistemas con frecuencia toman medidas extremas para reducirlo. Por ejemplo, en una aplicación que intercala tres archivos no es raro encontrar los tres archivos de entrada almacenados en tres unidades de disco distintas y el archivo de salida en una cuarta, de tal forma que no hay necesidad del desplazamiento, porque las operaciones de E/S saltan de un archivo a otro.

El desplazamiento implica varias operaciones un tanto lentas. Entre las más importantes se consideran el tiempo inicial de arranque (s) y el tiempo que toma recorrer los cilindros que se deben cruzar una vez que el brazo de acceso adquiere su velocidad normal. Si n indica el número de cilindros por atravesar, entonces las contribuciones de esos dos valores pueden aproximarse con una función lineal de la forma

$$f(n) = m \times n + s$$

donde m es una constante que depende de la unidad de disco en cuestión. Por ejemplo, el tiempo de desplazamiento en una unidad de disco fijo barata (circa 1985) de 20 megabytes, y usada con computadores personales, puede aproximarse por

$$f(n) = 0.3 \times n + 20 \text{ mseg.}$$

Una unidad de disco grande y más costosa puede tener un tiempo de desplazamiento aproximado por

$$f(n) = 0.1 \times n + 3 \text{ mseg.}$$

La unidad más grande arranca más rápido ($s = 3$ mseg contra $s = 20$ mseg) y atraviesa las pistas más velozmente ($m = 0.1$ contra $m = 0.3$).

Como a menudo resulta imposible saber con exactitud cuántas pistas serán atravesadas en cada desplazamiento, normalmente se

intenta determinar el *tiempo promedio de desplazamiento* requerido para una operación de archivo en particular. Si las posiciones de inicio y fin para cada acceso son aleatorias, resulta que el desplazamiento promedio atraviesa una tercera parte del número total de pistas.[†]

RETRASO POR ROTACION. El retraso por rotación se refiere al tiempo que transcurre para que en el disco que gira el sector que se desea quede bajo la cabeza de lectura y escritura. Por lo general, los discos giran a una velocidad aproximada de 3600 rpm,^{††} lo que significa una revolución cada 16.7 mseg. En promedio, el retraso por rotación es la mitad de una revolución, o cerca de 8.3 mseg.

TIEMPO DE TRANSFERENCIA. Una vez que los datos que se desean están bajo la cabeza de lectura y escritura, se pueden transferir. El tiempo de transferencia está dado por la fórmula

$$\text{Tiempo de transferencia} = \frac{\text{número de bytes por transferir}}{\text{número de bytes en una pista}} \times \text{Tiempo de rotación.}$$

Si la unidad está dividida en sectores, el tiempo de transferencia de un sector depende del número de sectores que haya en una pista. Por ejemplo, si hay 32 sectores por pista, el tiempo requerido para transferir un sector será de 1/32 de revolución, o 0.5 mseg.

ALGUNOS CALCULOS DE TIEMPO. Ahora se analizarán dos situaciones diferentes de procesamiento de archivos que muestran la forma en que los tiempos de acceso se ven afectados por los diferentes tipos de acceso a un archivo. La base de los cálculos es el disco fijo normal de 20 megabytes antes mencionado. Aunque este disco en particular es típico en los discos fijos baratos usados en computadores personales, las observaciones que se hacen conforme se realizan los cálculos son bastante generales. Los discos usados en computadores más grandes y costosos son mayores y más rápidos que este disco, pero la naturaleza y los costos relativos de los factores que intervienen en los tiempos de acceso totales son, en esencia, los mismos.

La unidad de disco en cuestión tiene cuatro superficies (dos platos), con una cabeza de lectura y escritura por superficie. Hay 612 pistas por superficie, 16 sectores por pista y 512 bytes por sector. La unidad usa

[†] La deducción de este resultado, así como modelos más detallados y refinados, se encuentran en Wiederhold [1983], Knuth [1973b], y en Teory y Fry [1982]. Pechora y Schoeffler [1983] demuestran que un conjunto muy diferente de suposiciones se aplica a los discos flexibles.

^{††} Los discos flexibles rotan mucho más lentamente, esto es, entre 300 y 600 rpm.

un tamaño de cúmulo de 16 sectores (8 Kbytes) y un tamaño de extensión de un cúmulo, de tal forma que el espacio asignado para almacenar archivos es en unidades de una pista. Los sectores están intercalados con un factor de intercalación de 5, de manera que una pista puede transferirse en cinco revoluciones. Puesto que una revolución tarda 16.7 mseg, una pista puede transferirse en 83.5 mseg.

Supongamos que, para este disco, se desea conocer el tiempo de lectura de un archivo de 128 Kbytes que está dividido en 256 registros del tamaño de un sector. Lo primero que se necesita saber es cómo está distribuido el archivo en el disco. Como cada cúmulo es una pista, el archivo debe estar almacenado como una secuencia de pistas. Puesto que cada pista del disco almacena 8K, el disco necesita 16 pistas para almacenar los 128 Kbytes completos que se desea leer. Supongamos ahora una situación en la que los 16 cúmulos están diseminados de manera aleatoria sobre la superficie del disco. (Esta es una situación extrema, elegida para poner de relieve un aspecto específico. Sin embargo, no es tan extrema, ya que puede ocurrir fácilmente en un disco común de computador personal saturado que tenga un gran número de pequeños archivos.)

Un desplazamiento promedio en este ambiente atraviesa una tercera parte de los 612 cilindros, de modo que usando la fórmula anterior tenemos un

$$\text{Desplazamiento promedio} = f(612/3) = 0.3 \times (612/3) + 20 \text{ mseg} = 81.2 \text{ mseg.}$$

Ahora se puede calcular el tiempo de lectura de los 128 Kbytes del disco. Primero se estima el tiempo de lectura del archivo sector por sector, *en secuencia*. Este proceso abarca las siguientes operaciones para cada cúmulo.

Desplazamiento promedio	81.2 mseg
Retraso por rotación	8.3 mseg
Lectura de 16 sectores (5 × 16.7 mseg)	83.5 mseg
	172.0 mseg

Si se pretende encontrar y leer 16 pistas, entonces

$$\text{Tiempo total} = 16 \times 173 \text{ mseg} = 2768 \text{ mseg} = 2.8 \text{ segundos.}$$

Ahora se calcula el tiempo de lectura de los mismos datos mediante *acceso aleatorio*, en lugar del acceso secuencial. En otras palabras, en lugar de leer un sector después de otro, se supone que se debe acceder a los sectores en algún orden que requiere múltiples saltos de una pista a otra cada vez que se lee un nuevo sector. Este proceso implica las siguientes operaciones para cada sector.

Desplazamiento promedio	81.2 mseg
Retraso por rotación	8.3 mseg
Lectura de un sector ($1/16 \times 16.7$)	1.0 mseg
	90.5 mseg

Tiempo total = 256×90.5 mseg = 23168 mseg = 23.2 segundos.

Esta diferencia de tiempo entre el acceso secuencial y el aleatorio es muy importante. Está claro que colocarse en el lugar adecuado sobre el disco y leer una gran cantidad de información secuencialmente es mucho mejor que saltar de un lado a otro y realizar el *desplazamiento* cada vez que se necesite un nuevo sector. Recuérdese que el tiempo de desplazamiento es muy caro; por ello, cuando se efectúan operaciones de disco conviene minimizarlo.

3.2

CINTA MAGNETICA

Cuando se accede a un archivo en forma secuencial no hay necesidad de almacenarlo en un disco si se puede usar un dispositivo menos costoso. Por ejemplo, considérese una lista de correo clasificada. Cuando se usa para generar etiquetas de correo, se puede acceder a sus registros en el orden que tienen en el archivo. Se puede intercalar con una segunda lista de correo, clasificándola y procesando luego los dos archivos secuencialmente. La actualización puede hacerse de manera similar. Como resulta raro acceder sólo a un registro en una lista de correo, el gasto adicional para lograrlo no justifica el costo de guardarlo en un disco.

Las unidades de cinta magnética pertenecen a una clase de dispositivos que no brindan la facilidad del acceso directo, pero son muy buenas para el procesamiento secuencial de datos. Cuando se sabe que un archivo grande normalmente no requiere acceso directo, almacenarlo en cinta tiene ventajas. Las cintas son compactas, estables en diferentes condiciones ambientales y fáciles de almacenar y transportar. Además, el espacio de la cinta generalmente es menos costoso que el del disco.

3.2.1 ORGANIZACION DE DATOS EN CINTAS

Puesto que el acceso a las cintas es secuencial, no se requieren direcciones para identificar la ubicación de los datos. En una cinta, la posición lógica de un byte dentro del archivo corresponde directamente

con su posición física relativa al inicio del archivo. Se puede imaginar la superficie de una cinta común como un conjunto de pistas paralelas, cada una de las cuales es una secuencia de bits. Si hay ocho pistas, los ocho bits cuyas posiciones correspondan con las ocho pistas respectivas forman un byte. Así, podemos imaginar un byte como una sección de cinta de un bit de ancho. Dicha sección se llama *marco*.

Una cinta común tiene nueve pistas (Fig. 3.10), y una de ellas se emplea para la *paridad*. El *bit de paridad* no es parte del dato, sino que se usa para revisar la validez de los datos. Si está en vigor la *paridad impar*, este bit se usa para que el número de bits iguales a 1 del marco sea *impar*. La paridad par trabaja en forma similar, pero rara vez se usa en las cintas.

Los marcos (bytes) se agrupan en bloques de datos cuyos tamaños varían entre unos cuantos bytes y muchos kilobytes, dependiendo de las necesidades del usuario. Como la lectura de las cintas con frecuencia se realiza de bloque en bloque, y éstas no se pueden parar y arrancar instantáneamente, los bloques están separados por *huecos entre bloques*, los cuales no contienen información, y son lo bastante largos para permitir la detención y el arranque. Cuando las cintas usan paridad impar, ningún marco válido puede contener todos los bits en 0, de modo que se usa un número grande de marcos consecutivos con 0 para cubrir los huecos entre registros.

Las cintas se presentan en una diversidad de formas, tamaños y velocidades. Las diferencias de desempeño entre las unidades normalmente se miden en términos de tres cantidades:

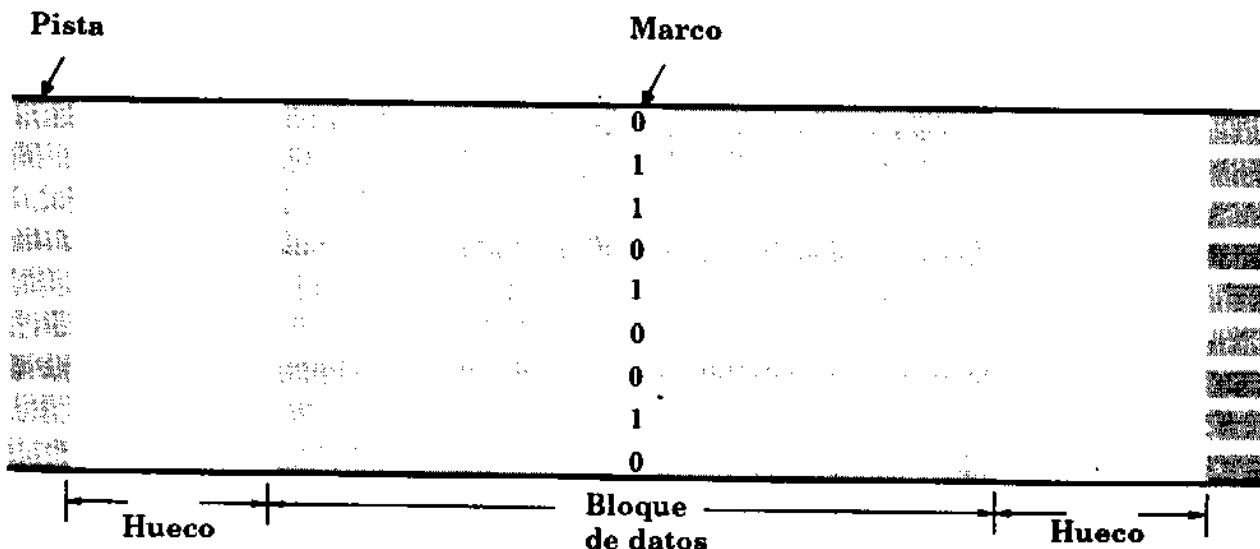


FIGURA 3.10 • Cinta de nueve pistas.

- Densidad de la cinta: las densidades más comunes son de 800, 1600 o 6250 bits por pulgada (*bpi*, por sus siglas en inglés) por pista, pero en la actualidad hay hasta de 30 000 bpi;
- Velocidad de la cinta: suele ser de 30 a 200 pulgadas por segundo (*ips*, por sus siglas en inglés), y
- Tamaño del hueco entre bloques: normalmente fluctúa entre 0.3 y 0.75 pulgadas.

3.2.2 ESTIMACION DE REQUERIMIENTOS DE LONGITUD DE CINTA

Es común respaldar conjuntos de archivos de disco para protegerse contra la pérdida de información, ocasionada por errores de software o hardware; las cintas son un buen recurso para lograrlo. Por ejemplo, suponga que se desea almacenar una copia de respaldo de un archivo de una gran lista de correo, y se necesita saber cuánta cinta se requerirá. Suponga que el archivo tiene un millón de direcciones, y que cada una se almacena en un registro de 100 bytes. Si se desea almacenar el archivo en una cinta de 6250 bpi que tiene un hueco entre bloques de 0.3 pulgadas, ¿cuánta cinta se necesita?

Para contestar esta pregunta primero se debe determinar qué es lo que ocupa espacio en la cinta. Existen dos contribuyentes principales: los huecos entre bloques y los bloques de datos. Para cada bloque de datos hay un hueco entre bloques. Si se define

b = la longitud física de un bloque de datos,
 g = la longitud de un hueco entre bloques, y
 n = el número de bloques de datos,

entonces el requerimiento de espacio s para almacenar el archivo es

$$s = n \times (b + g).$$

Se sabe que g es 0.3 pulgadas, pero se desconocen los valores de b y n . De hecho, b es cualquier valor que se desee, y n depende de la elección de b . Si se determina que cada bloque de datos contenga un registro de 100 bytes, entonces b , la longitud de cada bloque, está dada por

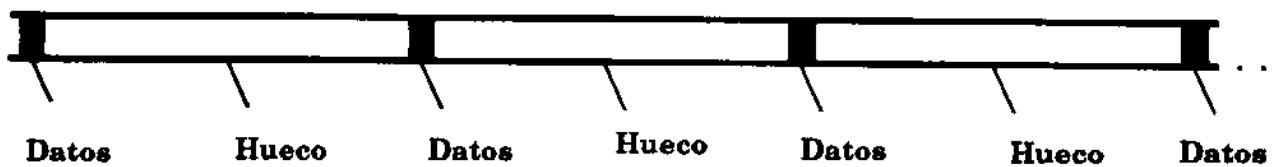
$$b = \frac{\text{Tamaño del bloque (bytes por bloque)}}{\text{Densidad de la cinta (bytes por pulgada)}} = \frac{100}{6250} = 0.016 \text{ pulgadas},$$

y n , el número de bloques, es un millón (uno por registro).

El número de registros almacenados en un bloque físico se llama *factor de bloque*. Este tiene el mismo significado que tenía cuando se aplicaba al uso de bloques para el almacenamiento en disco. Se elige aquí un factor de bloque de 1 porque cada bloque tiene sólo un registro. En consecuencia, el requerimiento de espacio para el archivo es

$$\begin{aligned}s &= 1\ 000\ 000 \times (0.016 + 0.3) \text{ pulgadas} \\&= 1\ 000\ 000 \times 0.316 \text{ pulgadas} \\&= 316\ 000 \text{ pulgadas} \\&= 26\ 333 \text{ pies.}\end{aligned}$$

La longitud de las cintas magnéticas va de 300 a 3600 pies, siendo 2400 pies la longitud más común. Está claro que se necesitan bastantes cintas de 2400 pies para almacenar el archivo. ¿Es así? El lector habrá notado que el tamaño de bloque elegido no es muy conveniente desde el punto de vista de la utilización del espacio. En la representación física del archivo los huecos entre bloques ocupan alrededor de *diecinueve veces* más espacio que los bloques de datos. Si tomara una fotografía de la cinta, se vería algo semejante a esto:



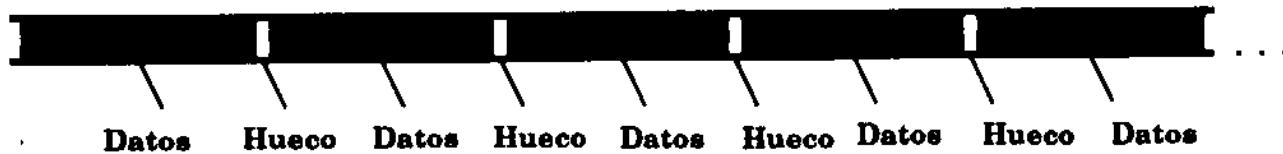
¡La mayor parte del espacio de la cinta está sin uso!

Está claro que se debe incrementar la cantidad relativa del espacio empleado por los datos si se pretende comprimir el archivo en una cinta de 2400 pies. Si se incrementa el factor de bloque, se puede *disminuir* el número de bloques, lo cual reduce el número de huecos entre bloques, y esto, a su vez, hace disminuir la suma de espacio consumido por los huecos entre bloques. Por ejemplo, si se incrementa el factor de bloque de 1 a 50, el número de bloques se convierte en

$$n = \frac{1\ 000\ 000}{50} = 20\ 000$$

y los requerimientos de espacio para los huecos entre bloques decrecen de 300 000 a 6000 pulgadas. Los requerimientos de espacio para los datos son, por supuesto, los mismos que se tenían anteriormente. Lo que ha cambiado es la cantidad *relativa* de espacio ocupado por los

huecos, comparada con los datos. Ahora una fotografía de la cinta se vería muy distinta:



Se le encarga al lector mostrar que el archivo puede caber fácilmente en una cinta de 2400 pies cuando se usa un factor de bloque de 50.

Cuando se calculan los requerimientos de espacio para un archivo, se obtienen números específicos para ese archivo. Una forma más general de medir el efecto que tiene la elección de distintos tamaños de bloque es mediante la *densidad de grabado efectiva*. Se supone que la densidad de grabado efectiva refleja la cantidad de datos reales que pueden almacenarse por pulgada de cinta. Ya que esto depende exclusivamente de los tamaños relativos de los huecos entre bloques y del tamaño del bloque de datos, puede definirse como

$$\frac{\text{Número de bytes por bloque}}{\text{Número de pulgadas necesarias para almacenar un bloque}}$$

En este ejemplo, cuando empleamos un factor de bloque de 1, el número de bytes por bloque es 100, y el número de pulgadas necesarias para almacenar un bloque es de 0.316. Por lo tanto, la densidad de grabado efectiva es

$$\frac{100 \text{ bytes}}{0.316 \text{ pulgadas}} = 316.4 \text{ bpi},$$

lo cual está muy lejos de la densidad de grabado *nominal* de 6250 bpi.

Desde cualquier punto de vista, la utilización del espacio depende de los tamaños relativos de los bloques de datos y de los huecos entre bloques. Ahora se verá cómo estos tamaños afectan el *tiempo* que se empleará en transmitir los datos de la cinta.

3.2.3 ESTIMACION DE LOS TIEMPOS DE TRANSMISION DE DATOS

Si se comprendió el papel que desempeñan los huecos entre bloques y los tamaños de los bloques de datos en la determinación de la densidad

de grabado efectiva, entonces se comprenderá de inmediato que esos dos factores también afectan la tasa de transmisión de datos. Otros dos factores que afectan la tasa de transmisión de datos, desde o hacia la cinta son la densidad de grabado nominal y la velocidad con la que pasa la cinta por la cabeza de lectura y escritura. Si se conocen esos dos valores, se puede calcular la *tasa de transmisión de datos nominal*:

$$\text{Tasa nominal} = \text{densidad de la cinta (bpi)} \times \text{velocidad de la cinta (ips)}.$$

Por lo tanto, la cinta de 200 ips y 6250 bpi tiene una tasa de transmisión nominal de

$$\begin{aligned} 6250 \times 200 &= 1\,250\,000 \text{ bytes/seg} \\ &= 1250 \text{ kilobytes/seg.} \end{aligned}$$

Esta tasa es competitiva con la mayoría de las unidades de disco.

Pero, ¿qué sucede con los huecos entre bloques? Una vez que los datos se han diseminado debido a los huecos entre bloques, ciertamente la *tasa de transmisión efectiva* se ve afectada. Por ejemplo, suponga que se emplea un factor de bloque de 1 con el mismo archivo y la misma cinta que se analizó en la sección anterior (1 000 000 de registros de 100 bytes, 0.3 pulgadas de hueco entre bloques). Se vio que la densidad de grabado efectiva para esta organización de cinta es de 316.4 bpi. Si la cinta se mueve a una velocidad de 200 ips, entonces su tasa de transmisión efectiva es

$$\begin{aligned} 316.4 \times 200 &= 63\,280 \text{ bytes/seg} \\ &= 63.3 \text{ kilobytes/seg,} \end{aligned}$$

juna tasa que representa alrededor de *una vigésima parte* de la tasa nominal!

Por supuesto, un factor de bloque mayor que 1 mejora este resultado, y un factor de bloque sustancialmente mayor lo mejora mucho más.

Aunque aquí hay otros factores que pueden influir en el desempeño, el tamaño del bloque se considera como la variable de mayor influencia en la utilización del espacio y en la tasa de transmisión de datos. Los otros factores que se han incluido son el tamaño del hueco entre bloques, la velocidad de la cinta y la densidad de grabado, pero éstos están fuera del control del usuario. Otro factor que algunas veces puede ser importante es el tiempo que toma arrancar o detener la cinta. En los ejercicios que aparecen al final del capítulo se considera el tiempo de arranque y detención.

3.2.4 APLICACIONES DE LAS CINTAS

La cinta magnética es un medio apropiado para las aplicaciones de procesamiento secuencial cuando no es el caso que los archivos que se procesan se usen también en aplicaciones que requieren acceso directo. Por ejemplo, considere el problema de actualizar una lista de correo para un periódico mensual. ¿Es esencial que la lista se mantenga absolutamente al corriente, o es suficiente con una actualización mensual?

Si la información debe estar al corriente en todo momento, entonces el medio debe permitir el acceso directo, de tal forma que las actualizaciones individuales puedan hacerse de inmediato. Pero si se necesita que la lista de correo esté actualizada sólo cuando se imprimen las etiquetas de correo, todos los cambios que ocurran durante el transcurso de un mes se recolectan en un lote y se colocan dentro de un archivo de transacciones que se clasifica en el mismo orden que la lista de correo. Después se puede ejecutar un programa que lea simultáneamente los dos archivos y realice todos los cambios requeridos al pasar una sola vez por los datos.

Puesto que una cinta es relativamente barata, también es un buen medio para almacenar los datos fuera de línea. A los precios actuales, un paquete de discos removibles que almacene 150 megabytes cuesta alrededor de 30 veces más que un rollo de cinta que, si maneja los bloques adecuadamente, almacena la misma cantidad. La cinta es un buen medio para almacenar archivos y para transportar datos, mientras no se requiera que estén disponibles de inmediato para procesamiento directo.

3.3

OTROS TIPOS DE ALMACENAMIENTO

Cuando los sistemas de computación se vuelven grandes y complejos o muy especializados, las cintas y los discos por sí solos no pueden satisfacer todos los requerimientos del almacenamiento secundario. Existen otros medios y dispositivos menos comunes que los discos y las cintas, los cuales, no obstante, tienen usos importantes para ciertas aplicaciones de archivos. He aquí algunos de los principales medios secundarios.

TARJETAS PERFORADAS. Las tarjetas perforadas son caras, lentas y propensas a errores; sin embargo, a pesar de sus inconvenientes, existen al menos dos razones que siguen justificando su empleo:

- Su tamaño y durabilidad facilitan su manejo individual, sobre todo en ambientes donde no hay control, como en tiendas de departamentos, correo y casetas de peaje, y
- Aún existe mucho hardware y software viejo que trabaja sólo con tarjetas perforadas, y que no es fácilmente reemplazable por hardware y software más nuevo, más barato y más rápido.

RESPALDO Y ALMACENAMIENTO DE ARCHIVOS. Normalmente el respaldo y el almacenamiento de archivos se mantienen fuera de línea, y éstos deben cargarse a memoria RAM o a una unidad de disco antes de acceder a ellos. A continuación se presentan algunos dispositivos y medios de uso común para el respaldo y el almacenamiento de archivos.

Unidades de cinta de grabado continuo (*streamers*). Estas unidades están diseñadas especialmente para la descarga de datos a alta velocidad y sin detención a partir de discos. Aunque suelen ser más baratas que las unidades de cinta normales, son menos apropiadas para un procesamiento que implique muchos arranques y paradas.

Unidades de disco flexible. Las unidades de disco flexible son baratas, pero son lentas y almacenan relativamente pocos datos. Estos discos son buenos para respaldar archivos individuales u otros discos flexibles, y para transportar pequeñas cantidades de datos.

Paquetes de discos removibles. Algunas veces pueden montarse diferentes paquetes de discos en la misma unidad en distintos tiempos. Esto proporciona una forma adecuada de almacenamiento de respaldo que también hace posible el acceso directo a los datos.

Sistemas de almacenamiento masivo. Estos sistemas pueden tener acceso a cualquiera de numerosos cartuchos de cinta especialmente diseñados, procedentes de un acervo de cartuchos, en pocos segundos y sin intervención humana. Los sistemas de almacenamiento masivo tienen mucha capacidad (cientos de miles de millones de bytes) y operan con tiempos de acceso menores de 15 segundos. Son baratos, en términos de costo por bit, pero resultan económicos sólo en instalaciones que usan grandes cantidades de datos fuera de línea. En términos estrictos, los sistemas de almacenamiento masivo no son realmente dispositivos de respaldo, ya que en general almacenan la única copia de un conjunto de datos determinado. Por esta razón algunos sistemas de almacenamiento masivo mantienen dos copias de cada cartucho.

Almacenamiento redundante en línea. Si la pérdida del acceso a los datos aun por corto tiempo, representa un costo extremadamente alto, pueden mantenerse dos copias idénticas de todos los datos en dos dispositivos idénticos. Este enfoque tiene la ventaja adicional de que, si está ocupado sólo uno de los dos dispositivos cuando se necesita cierta parte de los datos, se puede acceder a éstos desde el otro dispositivo, sin demora. Una desventaja del almacenamiento redundante es la dificultad que hay de asegurar que las dos copias sean en realidad idénticas todo el tiempo. Por ejemplo, a menudo sucede que durante una operación de escritura los datos nuevos no llegan al mismo tiempo a los dos dispositivos.

Discos ópticos. El almacenamiento en disco óptico está ya entre los principales medios de almacenamiento secundario, pues ha demostrado su competitividad, en términos de capacidad, velocidad y costo, en relación con los demás medios de respaldo y almacenamiento para archivo.

ALMACENAMIENTO MAS RAPIDO QUE EL DISCO. En la actualidad, conforme decrece el costo de la memoria RAM, aumenta el número de usuarios que hasta hace pocos años tenían que almacenar los datos en disco. Hay dos formas efectivas de usar la memoria RAM para reemplazar el almacenamiento secundario: los *discos en RAM* y el *almacenamiento caché para disco*.

Un *disco en RAM* es una gran sección de memoria RAM configurada para simular en todos aspectos el comportamiento del disco mecánico, excepto en velocidad y volatilidad. Como los datos en memoria RAM pueden colocarse sin desplazamiento o retraso por rotación, estos discos pueden proporcionar un acceso mucho más rápido que los discos mecánicos. Puesto que la memoria RAM normalmente es volátil, el contenido de un disco en RAM se pierde cuando se apaga el computador. Con frecuencia se usan los discos en RAM en lugar de los discos flexibles porque son mucho más rápidos y se necesita relativamente poca memoria RAM para simular un disco flexible normal.

Un *almacenamiento caché para disco*[†] es un gran bloque de memoria RAM configurado para contener páginas de datos de un disco. Un esquema de almacenamiento caché para disco común puede usar un caché de 256 Kbytes con un disco. Cuando se solicitan datos de la memoria secundaria, el administrador de archivos examina primero

[†] El término *caché* (a diferencia del *almacenamiento caché para disco*) se refiere, en general, a un bloque de memoria principal de muy alta velocidad que realiza el mismo tipo de operaciones para mejorar el desempeño de la memoria RAM que los que realiza un *almacenamiento caché para disco* con respecto a la memoria secundaria.

dentro del almacenamiento caché del disco para saber si contiene la página con los datos solicitados. Si es así, los datos pueden procesarse inmediatamente. En caso contrario, el administrador de archivos lee del disco la página que contiene los datos, reemplazando alguna de las páginas presentes en el almacenamiento caché. Cuando el patrón de acceso de datos de un programa muestra un alto grado de referencias locales, la memoria caché puede proporcionar mejoras importantes en el desempeño.

3.4

EL ALMACENAMIENTO COMO UNA JERARQUÍA

Aunque la combinación óptima de dispositivos de un sistema de computo depende mucho de las necesidades de los usuarios del sistema, podemos imaginar cualquier sistema de computación como una jerarquía de dispositivos de almacenamiento de diferentes velocidades, capacidades y costos. La figura 3.11 resume los diferentes tipos de almacenamientos encontrados en los diferentes niveles de tales jerarquías y muestra en forma aproximada cómo se pueden comparar en términos de tiempo de acceso, capacidad y costo.

3.5

EL VIAJE DE UN BYTE

¿Qué sucede cuando un programa escribe un byte en un archivo en disco? Se sabe lo que hace el programa (dice WRITE(...)), y ahora ya se sabe algo acerca de cómo se almacena el byte en un disco, pero aún no se ha examinado lo que sucede *entre* el programa y el disco. La historia completa de lo que sucede a los datos entre el programa y el disco no se puede narrar aquí (es necesario que el lector aprenda sobre sistemas operativos y comunicación de datos para comprenderla completa), pero se puede dar una idea de los numerosos elementos diferentes de hardware y software que intervienen, así como de las tareas que deben realizarse, mediante un ejemplo del viaje de un byte.

Supongamos que se desea agregar un byte que represente el carácter 'P', almacenado en una variable *c* de tipo carácter, a un archivo llamado TEXTO, que se almacena en algún lugar de un disco.

Tipos de memoria	Dispositivos y medios	Tiempos de acceso (seg)	Capacidades (bytes)	Costo (centavos de dólar/bit)
Primaria				
Registros	Núcleo de ferrita y semiconductores	10^{-8} - 10^{-6}	10^6 - 10^7	10^6 - 10^{-2}
Memoria RAM				
Disco en RAM y caché disco				
Secundaria				
Acceso directo	Discos magnéticos	10^{-3} - 10^{-1}	10^4 - 10^9	10^{-2} - 10^{-5}
Serie	Cintas y almacenamiento masivo	10^1 - 10^2	10^6 - 10^{11}	10^{-6} - 10^{-7}
Fuera de línea				
Respaldo y archivo	Discos magnéticos removibles, discos ópticos y cintas	10^0 - 10^2	10^4 - 10^8	10^{-6} - 10^{-7}

FIGURA 3.11 • Comparación aproximada de tipos de almacenamiento, circa 1985.

Desde el punto de vista del programa, el viaje completo que hará el byte puede estar representado por la proposición

WRITE (TEXTO, C, 1)

pero el viaje es mucho más largo de lo que sugiere esta sencilla proposición. Es un viaje que está marcado por obstrucciones, demoras e incluso por posibles accidentes. El byte tiene que usar varios medios diferentes de transportación, algunos lentos, algunos rápidos. Ocasionalmente tiene que sentarse a esperar hasta que un canal u otro vehículo quede disponible. Parte del tiempo viaja en grupo, por momentos solo. Pasa por muchos puntos de revisión y tiene que acreditar sus credenciales en cada ocasión.

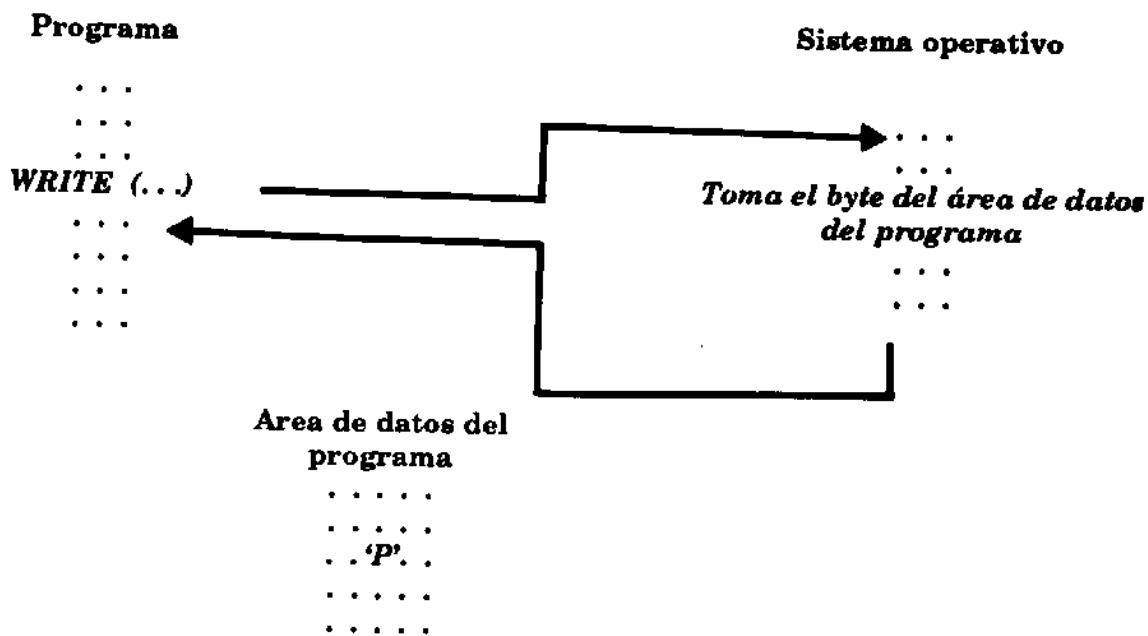


FIGURA 3.12 • La proposición `WRITE()` en un programa pide al sistema operativo que envíe un carácter al disco e indica al sistema operativo la posición del carácter. El sistema operativo se encargará de hacer la escritura real, y después devolverá el control al programa solicitante.

El byte comienza en algún lugar de la memoria RAM, como el contenido de la variable de caracteres `c`. La proposición `WRITE()` da como resultado una llamada al sistema operativo del computador, el cual tiene la tarea de verificar que el resto del viaje se complete con éxito (Fig. 3.12). Con frecuencia, el programa puede proporcionar al sistema operativo información que le ayude a realizar esta tarea con mayor eficiencia, pero una vez que el sistema operativo se encarga, la tarea de vigilar el resto del viaje queda fuera del alcance del programa.

3.5.1 EL ADMINISTRADOR DE ARCHIVOS

Un sistema operativo no es un solo programa, sino un conjunto de programas, cada uno de los cuales está diseñado para manejar una parte diferente de los recursos del computador. Entre esos programas existen algunos que tratan con aspectos relacionados con archivos y dispositivos de entrada y salida. A este subconjunto de programas se le llama *administrador de archivos* del sistema operativo. El administrador de archivos puede concebirse como un conjunto de capas de procedi-

mientos (Fig. 3.13), donde las capas superiores tratan principalmente con los aspectos simbólicos, o lógicos, de la administración de archivos, y las capas inferiores tratan con los aspectos físicos. El viaje del byte en cuestión se inicia con una llamada a la capa más elevada. Cada capa llama a la inmediata inferior, hasta que, en el nivel más bajo, el byte se escribe físicamente en el disco.

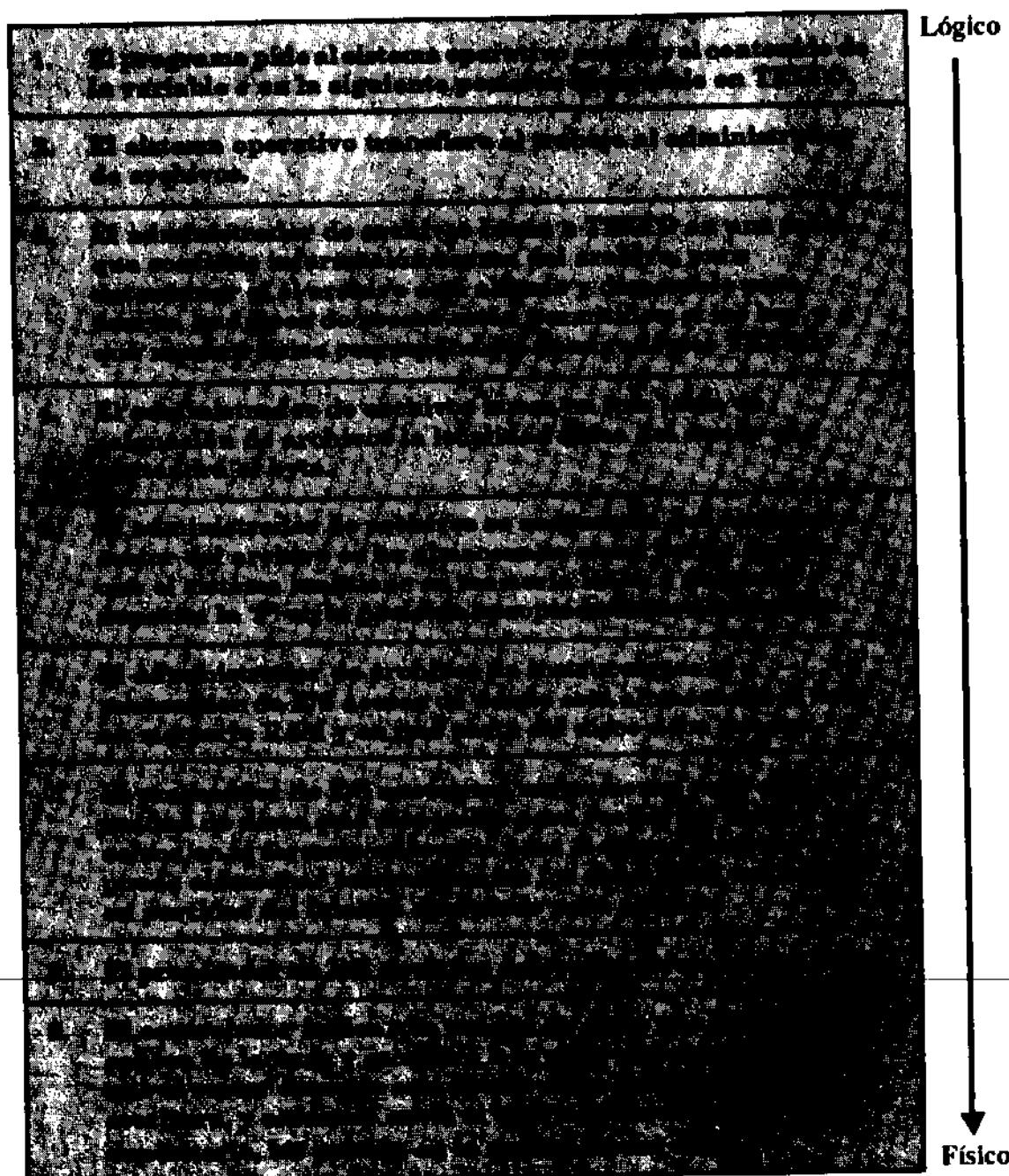


FIGURA 3.13. Capas de procedimientos que intervienen en la transmisión de un byte del área de datos de programa a un archivo en disco llamado TEXTO.

Nombre	Abierto por	Tipo de acceso para el que se abrió	Propietario	Modos de protección	...
UNARCH	Smith	sólo lectura	Smith	Propietario: lectura/escritura Otros: sólo lectura	...
PROGRAM1	Jones	lectura/escritura	Jones	Propietario: lectura/escritura Otros: lectura/escritura	...
TEXTO	Lector	sólo escritura	Lector	Propietario: lectura/escritura Otros: sin acceso	...
.
.
.

FIGURA 3.14 Tabla de información acerca de los aspectos lógicos de un archivo. Los puntos suspensivos (...) indican que puede incluirse otra información, como en cuál parte de la tabla FAT puede encontrarse información acerca de aspectos físicos del archivo.

El administrador de archivos empieza por cerciorarse de que las características lógicas del archivo sean compatibles con lo que se le solicita. Puede buscar el archivo solicitado en una tabla, donde aparecen datos acerca de si el archivo ha sido abierto, a qué tipo de archivo se está enviando el byte (un archivo binario, uno de texto o con alguna otra organización), quién es el propietario del archivo, y si está permitido el acceso del WRITE() para este usuario en particular (Fig. 3.14).

Una vez que se identificó el archivo deseado y se verificó la legalidad del acceso solicitado, el administrador de archivos debe determinar dónde se depositará la 'P' en el archivo TEXTO. Puesto que la 'P' se agregará al archivo, el administrador de archivos necesita saber dónde está el final del archivo, es decir, la posición física de su último sector. Esta información se obtiene de la tabla de asignación de archivos (FAT), descrita anteriormente. A partir de la tabla FAT, el administrador de archivos localiza la unidad, el cilindro, la pista y el sector donde se almacenará el byte.

3.5.2 EL BUFFER DE E/S

En seguida, el administrador de archivos determina si el sector que contendrá la 'P' está en memoria RAM o si es necesario cargarlo. Si el sector necesita cargarse, el administrador de archivos debe encontrar

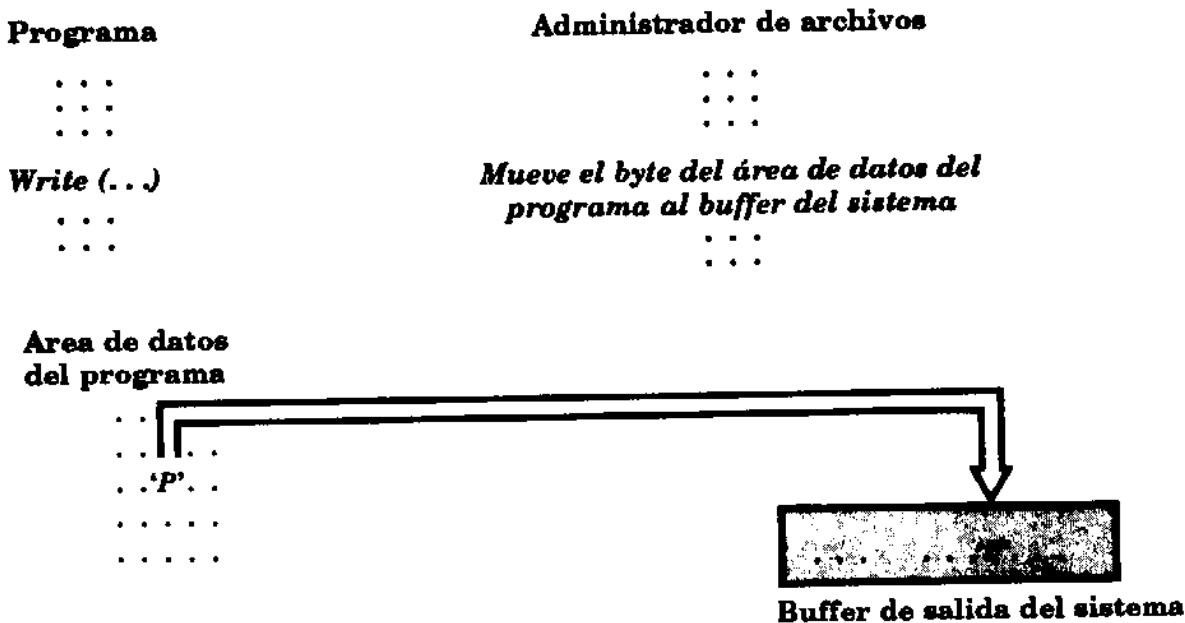


FIGURA 3.15 • El administrador de archivos mueve la 'P' del área de datos del programa al buffer de salida del sistema, donde puede unirse a otros bytes dirigidos al mismo lugar del disco.

un *buffer del sistema para E/S* disponible, y después leer el sector del disco. Una vez que se tiene el sector en un buffer en memoria RAM, el administrador de archivos puede depositar la 'P' en la posición apropiada dentro del buffer (Fig. 3.15).

El buffer que el sistema tiene para E/S permite al administrador de archivos leer y escribir datos en unidades del tamaño del bloque o del sector. En otras palabras, hace posible que el administrador de archivos asegure que la organización de los datos en memoria RAM concuerda con la organización que tendrá en el disco.

En lugar de enviar el sector inmediatamente al disco, el administrador de archivos por lo común espera a que se acumulen más bytes que vayan al mismo sector antes de realmente transmitir cualquier dato. Aun cuando la proposición `WRITE(TEXTO,c,1)` parece implicar que el carácter se está enviando de inmediato al disco, éste puede mantenerse en memoria RAM por algún tiempo antes de enviarse efectivamente. (Hay muchas ocasiones en que el administrador de archivos no puede esperar hasta que un buffer se llene antes de transmitirlo. Por ejemplo, si `TEXTO` se cerrara, se tendrían que *vaciar* todos los buffers de salida que tengan datos en espera de ser transcritos allí, para que los datos no se perdieran.)

3.5.3 EL BYTE SALE DE LA MEMORIA RAM: EL PROCESADOR DE E/S

Hasta aquí, todas las actividades del byte ocurren dentro de la memoria primaria del computador y es probable que las haya llevado a cabo la unidad central de procesamiento del computador (UCP). El byte ha viajado por caminos que han sido diseñados para ser muy rápidos y que son relativamente caros. Ahora es tiempo de que el byte viaje fuera de la memoria RAM, en dirección a la unidad de disco. Tiene que recorrer un camino para datos que normalmente es más lento y más estrecho que el de la memoria principal. (Un computador común puede tener un camino interno para datos de cuatro bytes de ancho, mientras que el camino que lleva al disco puede ser de uno o dos bytes de ancho.)

El embotellamiento originado por las diferencias de velocidad y anchura de los caminos para los datos obliga al byte y a sus compañeros a esperar hasta que un camino externo para datos esté disponible, y aun cuando entren en un camino disponible, viajarán más despacio que en memoria RAM.[†] Esto significa también que la UCP dispone de tiempo adicional mientras maneja información en porciones pequeñas y a velocidades suficientemente bajas para que el mundo exterior pueda manejarlas. De hecho, las diferencias entre las velocidades internas y externas de transmisión de datos con frecuencia son tan grandes que la UCP puede transmitir simultáneamente a varios dispositivos externos.

Los procesos de desensamblaje y ensamblaje de grupos de bytes para transmisión desde o hacia los dispositivos externos son tan especializados que no es razonable pedir a una costosa UCP, diseñada para propósitos generales, que desperdicie su valioso tiempo en hacer E/S, cuando un dispositivo más simple haría el trabajo igualmente bien, dejando a la UCP en libertad de realizar el trabajo para el que fue diseñada. Tal dispositivo de uso especial se llama *procesador de E/S*.

Un procesador de E/S puede ser desde un simple circuito integrado, capaz de tomar un byte y mediante una señal únicamente pasarlo, hasta un poderoso computador pequeño, capaz de ejecutar programas muy complejos y de comunicarse en forma simultánea con muchos dispositivos. El procesador de E/S recibe instrucciones del sistema operativo pero, una vez que empieza el procesamiento de E/S, actúa en forma independiente, relevando al sistema operativo (y a la UCP) de las tareas de comunicación con los dispositivos de almacenamiento secun-

[†] En muchos computadores, los caminos para datos se denominan *canales de datos (data bus, en inglés)*, de tal forma que en inglés se puede entender que el viaje del byte en cuestión puede quedar en suspenso mientras "espera el autobús".

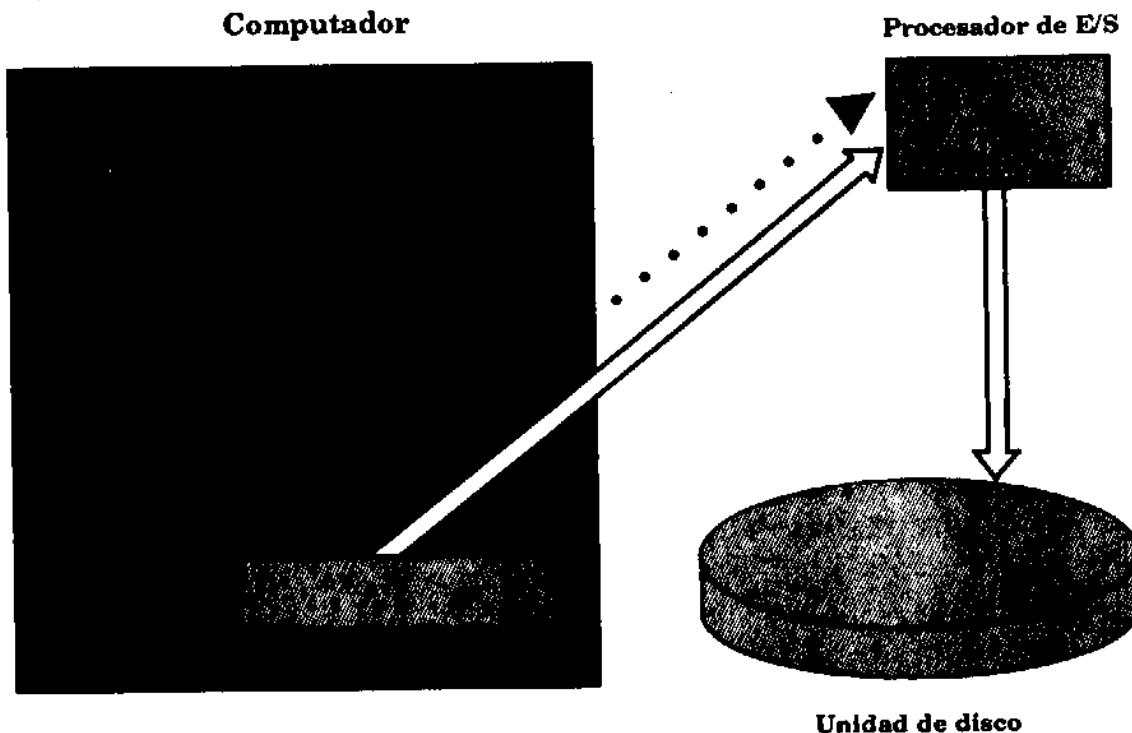


FIGURA 3.16 • El papel del procesador de E/S: el administrador de archivos envía las instrucciones al procesador de E/S en el formato de un programa del procesador de E/S. El procesador de E/S toma los datos del buffer del sistema, los prepara para almacenarlos en el disco, y después los envía al disco.

dario. Esto permite que los procesos de E/S y la computación interna se traslapen.[†]

En un computador común, el administrador de archivos puede informar ahora al procesador de E/S que hay datos en el buffer para transmitir al disco, junto con su cantidad y su destino en el disco. Esta información puede estar en forma de un pequeño programa que el sistema operativo construye y que el procesador de E/S ejecuta (Fig. 3.16).

Como el procesador de E/S debe ser capaz de manejar varias clases de dispositivos diferentes, no suele estar diseñado para controlar dispositivos específicos. En lugar de ello, su trabajo consiste básicamente en hacer que los datos sean aceptables para cualquier dispositivo, y en pasar la información adicional que un dispositivo en particular pueda necesitar para efectuar el resto del proceso.

[†]En muchos sistemas, el procesador de E/S puede tomar los datos directamente de memoria RAM, sin mayor intervención de la UCP. Este proceso se llama *acceso directo a memoria* (DMA, por sus siglas en inglés). En otros sistemas, la UCP debe colocar los datos en registros de E/S especiales antes de que el procesador de E/S pueda tener acceso a ellos.

3.5.4 EL BYTE LLEGA AL DISCO: EL CONTROLADOR DEL DISCO

El trabajo real de controlar la operación del disco lo realiza otro dispositivo, llamado *controlador del disco*. Es probable que, en el caso del byte en cuestión, el procesador de E/S consulte al controlador para saber si la unidad de disco está disponible para la escritura. Si hay mucho procesamiento de E/S, es muy factible que la unidad no esté libre y el byte tenga que esperar en su buffer hasta que la unidad se desocupe.

Lo que sucede después hace que el tiempo usado hasta aquí parezca insignificante en comparación: se da instrucciones a la unidad de disco para que coloque la cabeza de lectura y escritura en la pista y el sector de la unidad donde el byte y sus acompañantes serán almacenados. ¡Por primera vez, se pide a un dispositivo que haga algo mecánico! La cabeza de lectura y escritura debe desplazarse a la pista adecuada (a menos que ya esté ahí) y después esperar a que el disco gire hasta que el sector deseado esté bajo la cabeza. Una vez que la pista y el sector están localizados, el procesador de E/S (o quizás el controlador) pueden enviar los bytes, uno por uno, a la unidad de disco. El byte espera su turno y después viaja solo hacia la unidad donde probablemente será almacenado en un pequeño buffer de un byte, mientras espera ser depositado en el disco.

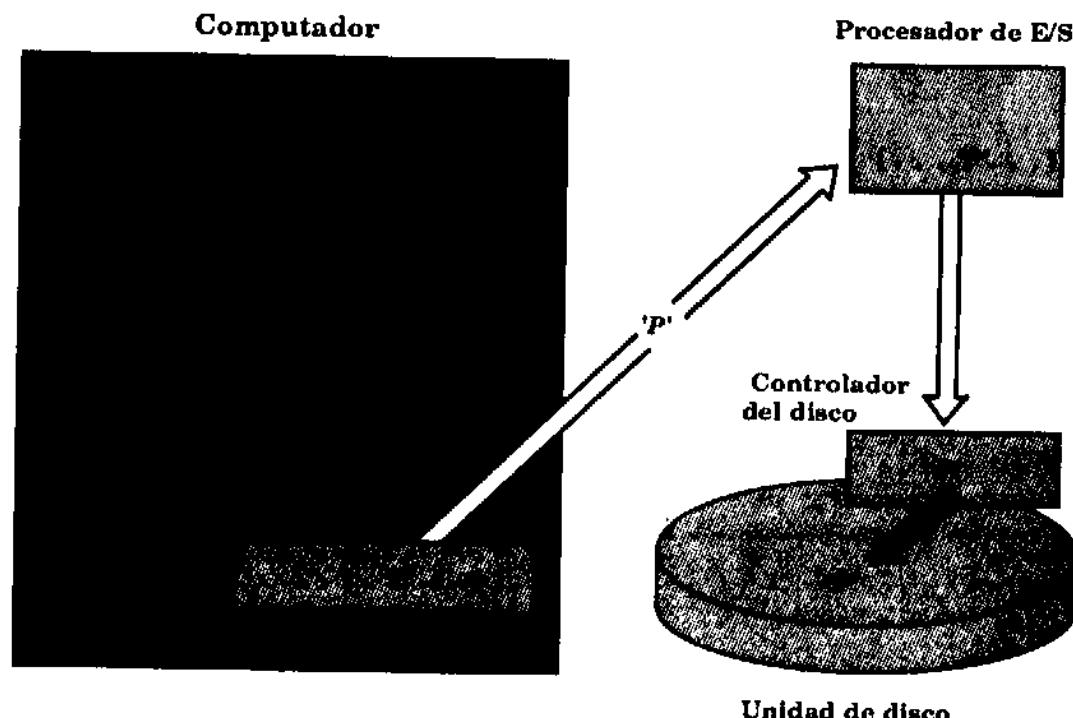


FIGURA 3.17 • Viaje típico de un byte: del área de datos del programa va al buffer del sistema, luego al procesador de E/S, al controlador del disco y, por último, al disco.

Por último, conforme el disco gira bajo la cabeza de lectura y escritura se depositan los ocho bits del byte del ejemplo, uno a la vez, en la superficie del disco (Fig. 3.17). Ahí la 'P' permanecerá, al final de su viaje, girando tranquilamente a 50 o 100 millas por hora.

3.6

MANEJO DE BUFFERS

A cualquier usuario de archivos le puede beneficiar saber lo que sucede con los datos que viajan entre un área de datos de un programa y el almacenamiento secundario. Un aspecto particularmente importante de este proceso es el uso de buffers. Manejar buffers implica trabajar con grandes grupos de datos en memoria RAM para que el número de accesos al almacenamiento secundario se reduzca. Esta sección se centra en la operación de los buffers que el sistema tiene para E/S, pero se debe tener en cuenta que el uso de los buffers en un programa también puede afectar considerablemente el desempeño.

CUELLOS DE BOTELLA CON UN BUFFER. Se sabe que un administrador de archivos asigna buffers de E/S con capacidad suficiente para almacenar los datos que van llegando, pero no se ha dicho nada hasta aquí respecto a cuántos buffers se usan. De hecho, es común que los administradores de archivos asignen varios buffers para realizar E/S.

Para comprender la necesidad de contar con varios buffers del sistema, considere lo que ocurre cuando un programa efectúa al mismo tiempo la entrada y la salida de un carácter, y sólo hay un buffer de E/S disponible. Cuando el programa pide su primer carácter, el buffer de E/S se carga con el sector que contiene el carácter, y éste se transmite al programa. Si en ese momento el programa decide escribir un carácter, el buffer de E/S se llena con el sector a donde debe ir el carácter de salida, y destruye su contenido original. Después, cuando se necesite el siguiente carácter de entrada, el contenido del buffer tendrá que transcribirse al disco para proporcionar lugar al sector (el original) que contenga el segundo carácter de entrada; y así sucesivamente.

Por fortuna existe una solución sencilla y generalmente muy efectiva para esta ridícula situación, y consiste en usar más de un buffer del sistema. Por esta razón, los sistemas de E/S casi siempre usan al menos dos buffers: uno para la entrada y otro para la salida.

Aun cuando un programa transmita datos en una sola dirección, el uso de un solo buffer para E/S del sistema puede frenarlo de manera considerable. Por ejemplo, se sabe que la operación de lectura de un sector de un disco es extremadamente lenta comparada con el tiempo

necesario para mover los datos en memoria RAM, por lo que se puede suponer que un programa que lee muchos sectores de un archivo desperdiciará mucho tiempo esperando a que el sistema de E/S llene su buffer cada vez que se efectúa una operación de lectura antes de poder empezar el procesamiento. Cuando esto sucede, se dice que el programa que se ejecuta está *orientado a E/S*: la UCP dedica mucho tiempo simplemente a esperar que la operación de E/S se realice. La solución a este problema es usar más de un buffer, y hacer que el sistema de E/S llene el siguiente sector o bloque de datos mientras procesa el presente.

ESTRATEGIAS DE MANEJO DE BUFFERS. Suponga que un programa sólo está escribiendo a un disco y que está orientado a E/S. La UCP desea llenar un buffer al mismo tiempo que se efectúa la E/S. Si se usan dos buffers y se permite que se traslapen la E/S y la UCP, esta última puede estar llenando un buffer mientras el contenido del otro se transmite al disco. Cuando ambas tareas terminan, las funciones de los buffers pueden intercambiarse. Esta técnica de intercambio de funciones de dos buffers después de cada operación de salida (o de entrada) se llama *manejo doble de buffers*. Este manejo doble permite al sistema operativo trabajar con un buffer mientras el otro se carga o vacía (Fig. 3.18).

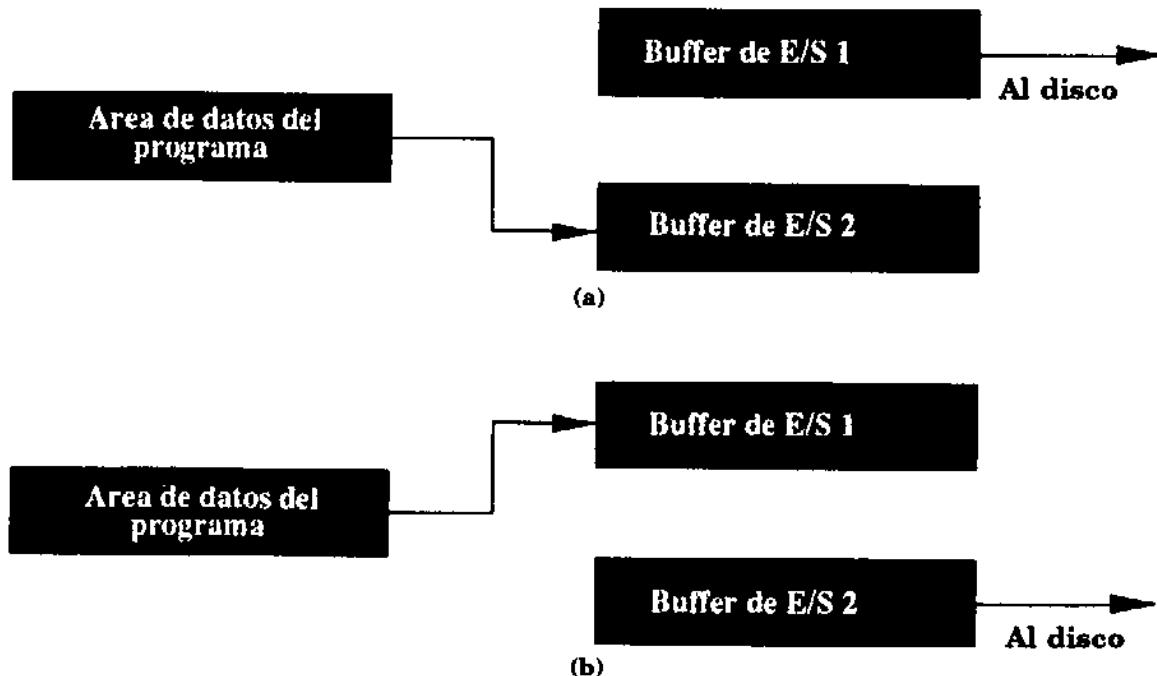


FIGURA 3.18• Manejo doble de buffers. (a) El contenido del buffer 1 de E/S del sistema se envía al disco mientras se llena el buffer 2 de E/S. (b) El contenido del buffer 2 se envía al disco mientras el buffer 1 de E/S se está llenando.

La idea de intercambiar los buffers del sistema para permitir el traslape del procesamiento y la E/S no necesariamente se restringe a dos buffers. En teoría, puede usarse cualquier número de buffers, y organizarlos en diversas formas. Es el sistema operativo el que maneja realmente los buffers del sistema y sólo rara vez pueden ser controlados por los programadores que no trabajen en los niveles de sistemas. Sin embargo, los usuarios son capaces, por lo común, de controlar el número de buffers del sistema asignados a los trabajos.

Algunos sistemas de archivos usan un esquema de manejo de buffers llamado *manejo de buffers en un depósito común*: cuando se necesita un buffer del sistema, se toma de un depósito común de buffers disponibles. Cuando el sistema recibe una solicitud para leer cierto sector o bloque, averigua si alguno de los buffers lo contiene. Si no está en ningún buffer, entonces el sistema busca en su depósito común de buffers alguno que no esté en uso y carga en él el sector o bloque.

Existen varios esquemas diferentes para decidir cuál buffer se tomará de un depósito común. Una estrategia que generalmente resulta efectiva consiste en tomar del depósito el buffer que haya sido menos recientemente usado. Una vez que se tiene acceso a un buffer, se coloca en la cola de los de uso menos reciente, de tal forma que puede retener sus datos hasta que se haya tenido acceso a todos los otros buffers de uso menos reciente. Esta estrategia del menos recientemente usado (LRU, por sus siglas en inglés) para el reemplazo de datos viejos por nuevos tiene muchas aplicaciones en computación. Esta estrategia se basa en la suposición de que en el futuro cercano es más probable que se necesite un bloque de datos de uso reciente que uno de uso menos reciente. El esquema LRU se encontrará de nuevo en capítulos posteriores.

Es difícil predecir en qué momento el agregar buffers adicionales deja de contribuir a mejorar el desempeño. Los buffers ocupan un espacio valioso de memoria RAM y, a mayor número de buffers, más tiempo necesitará el administrador de archivos para manejarlos. Cuando haya duda, considere la propuesta de experimentar con diferentes cantidades de buffers.

MODO DE MOVIMIENTO Y MODO DE DIRECCIONES. Algunas veces no es necesario distinguir entre un área de datos de programa y los buffers del sistema. Cuando los datos deban copiarse siempre de un buffer del sistema a un buffer del programa (o viceversa), el tiempo que toma efectuar este movimiento puede ser importante. Esta forma de manejar los datos en los buffers se llama *modo de movimiento*, ya que implica mover los bloques de datos de un lugar de la memoria RAM a otro antes de poder acceder a ellos.

Hay dos formas de evitar este modo de movimiento. Si el administrador de archivos puede efectuar directamente la E/S entre el

almacenamiento secundario y el área de datos del programa, no es necesario ningún movimiento adicional. Otra alternativa es que el administrador de archivos use los buffers del sistema para manejar toda la E/S, pero proporcione al programa las *localidades* de los buffers del sistema, a través del uso de variables tipo apuntador. Ambas técnicas son ejemplos de un enfoque general para el manejo de buffers, llamado *modo de direcciones*. Cuando se usa, los programas son capaces de operar directamente sobre los datos en el buffer de E/S, sin necesidad de transferirlos de un buffer de E/S a un buffer del programa.

No siempre resulta obvia la disponibilidad que en un sistema operativo haya de aspectos tales como el modo de dirección y el manejo común de buffers, y con frecuencia el usuario tendrá que buscar por sí mismo esta información. Algunas veces puede solicitar esos recursos comunicándose con su sistema operativo, y otras veces puede hacer que sean solicitados si organiza su programa en una forma compatible a la que el sistema operativo usa para la E/S. En el resto del texto se vuelve una y otra vez al tema de cómo mejorar el desempeño, tratando de comprender cómo trabajan los buffers y adaptando, de acuerdo con ello, los programas y las estructuras de archivos.

RESUMEN

En este capítulo se estudió el ambiente de software en el que deben operar los programas que procesan archivos junto con algunos dispositivos de hardware en donde comúnmente se almacenan los archivos, con la intención de comprender cómo deben influir en el diseño y procesamiento de archivos.

Al inicio se examinaron los dos medios de almacenamiento más importantes: discos y cintas magnéticos.

Las unidades de disco consisten en un conjunto de cabezas de lectura y escritura, interpuestas entre uno o más platos. Cada plato contribuye con una o dos superficies, cada superficie contiene un conjunto de pistas concéntricas, y cada pista está dividida en sectores o bloques. Al conjunto de pistas que pueden leerse sin mover las cabezas de lectura y escritura se le llama cilindro.

Hay dos formas básicas de hacer referencia a datos en los discos: por sectores y por bloques. En este contexto, el término *bloque* se refiere a un grupo de registros que se almacenan juntos en un disco y son tratados como una unidad, para propósitos de E/S. Al usar bloques, es

más fácil para el usuario hacer que la organización física de los datos corresponda con su organización lógica y, en consecuencia, se puede llegar a mejorar el desempeño. Algunas veces también la organización de las unidades de disco por bloques permite a la unidad de disco buscar entre los bloques de una pista un registro con cierta llave sin tener que transmitir primero los bloques no deseados a la memoria RAM.

Tres posibles desventajas de los dispositivos organizados por bloques son: el peligro de fragmentación interna de la pista, la molestia que significa para el usuario el trabajo con alguna complejidad adicional, y la pérdida de oportunidades para lograr algunos tipos de sincronización (como la intercalación de sectores) que sí proporciona el hacer referencia por sectores.

El costo de un acceso a disco puede medirse en términos del tiempo que consume el desplazamiento, el retraso por rotación y el tiempo de transferencia. Si se usa la intercalación de sectores, es posible acceder a sectores lógicamente adyacentes separándolos físicamente con uno o más sectores. Aunque el tiempo de acceso a un solo registro en forma directa es mucho menor que en forma secuencial, el tiempo de desplazamiento adicional requerido para efectuar el acceso directo hace que éste sea mucho más lento que el secuencial cuando se accede a una serie de registros.

Aunque las cintas magnéticas no son tan importantes como los discos, tienen un lugar destacado en el procesamiento de archivos. Las cintas son baratas, bastante rápidas para el acceso secuencial, compactas, resistentes y fáciles de guardar y transportar. Los datos almacenados en cinta normalmente se organizan en pistas paralelas de un bit de ancho, y se interpreta una sección transversal con un ancho de un bit como uno o más bytes. Para estimar la velocidad de procesamiento y la utilización del espacio, es fundamental reconocer el papel que desempeña el hueco entre bloques. La densidad de grabación efectiva y la tasa de transmisión efectiva son medidas útiles para calcular el desempeño que se puede esperar de una determinada organización física de archivos.

Se examinó la diversidad de medios adicionales empleados para almacenar archivos. Las distintas aplicaciones requieren dispositivos y medios muy variables, por lo que si se mezclan de acuerdo con las demandas de los problemas específicos, se pueden obtener ganancias en términos de costo y desempeño.

El capítulo describe el viaje que realiza un byte cuando se envía desde la memoria RAM hacia el disco. El viaje implica la participación de diversos programas y dispositivos, como son:

- El programa del usuario, que hace la llamada inicial al sistema operativo;

- El administrador de archivos del sistema operativo, el cual mantiene y maneja tablas cuya información emplea para pasar del punto de vista lógico que el programa tiene con respecto a un archivo hasta el archivo físico en donde se almacenará el byte;
- Un procesador de E/S y su software, que sincronizan la transmisión de un byte entre un buffer de E/S en memoria RAM y el disco;
- El controlador del disco y su software, que dan instrucciones a la unidad acerca de cómo encontrar la pista y el sector apropiados, para después enviar el byte, y
- La unidad de disco, que recibe el byte y lo deposita en la superficie del disco.

Se concluye con un análisis más detallado del manejo de buffers, con especial atención en las técnicas de administración, para mejorar el desempeño. Algunas técnicas comprenden el manejo doble de buffers, el manejo de buffers en un depósito común y el manejo de buffers empleando modo de direcciones.



TERMINOS CLAVE

Acceso directo a memoria (DMA, por sus siglas en inglés).

Transferencia directa de datos entre memoria RAM y dispositivos periféricos, sin intervención significativa de la UCP.

Administrador de archivos. La parte del sistema operativo responsable de la administración de archivos, que incluye un conjunto de programas cuyas responsabilidades van desde seguirle la pista a los archivos hasta llamar a los procesos de E/S que transmiten la información entre los almacenamientos primario y secundario.

Almacenamiento caché para disco. Segmento de memoria RAM configurado para contener páginas de datos de un disco. El almacenamiento caché puede proporcionar considerables mejoras en el tiempo de acceso cuando las solicitudes de acceso muestran un alto grado de referencias locales.

Bloque. Unidad de organización de datos que corresponde a la cantidad de datos transferidos en un solo acceso. Con frecuencia un bloque se refiere a un conjunto de registros, pero puede ser un conjunto de sectores (véase *cúmulos*) cuyo tamaño no tiene correspondencia con la organización de los datos. Algunas veces,

a un bloque se le llama *registro físico*. Otras veces, a un sector se le llama bloque.

Bpi. Bits por pulgada en una pista. En un disco, los datos se graban en forma serial en pistas. En una cinta, los datos se graban en paralelo sobre varias pistas, de tal forma que una cinta de 9 pistas a 6250 bpi contiene 6250 bytes por pulgada cuando se toman en cuenta las nueve pistas (una pista se usa para la paridad).

Cilindro. Conjunto de pistas en un disco que se encuentran directamente una sobre otra. Se puede acceder a todas las pistas de un cilindro sin tener que mover el brazo de acceso, lo que significa que se puede tener acceso a ellas sin gasto de *tiempo de desplazamiento*.

Controlador. Dispositivo que controla directamente la operación de uno o más dispositivos de almacenamiento secundario, como unidades de disco y de cinta magnética.

Cúmulo. Unidad mínima de *asignación de espacio* en un disco organizado por sectores, que consiste en uno o más sectores contiguos. El uso de cúmulos grandes puede mejorar los tiempos del acceso secuencial, garantizando capacidad de leer grandes cantidades de datos sin tener que desplazar el brazo del disco. Los cúmulos pequeños tienden a disminuir la fragmentación interna.

Dar formato. El proceso de preparar un disco para el almacenamiento de datos, que implica funciones como definir sectores, preparar la tabla de asignación de archivos del disco y determinar daños en el medio magnético.

Densidad de grabado nominal. Densidad de grabado en una pista de disco o cinta magnética sin considerar los efectos de los huecos o los sub-bloques con datos de control.

Densidad de grabado efectiva. Densidad de grabado resultante después de tomar en cuenta el espacio usado por los huecos entre bloques, por los bloques de datos de control y por otros detalles que acompañan a los datos y que también ocupan espacio.

Disco en memoria RAM. Bloque de memoria RAM configurado para simular el comportamiento de un disco.

Disco fijo. Unidad de disco con platos que no pueden removese.

Dispositivo de acceso secuencial. Dispositivo, unidad de cinta magnética o lectora de tarjetas, en donde se debe tener acceso al medio (p. ej., la cinta) desde su inicio. Algunas veces se llama dispositivo *serial*.

Dispositivo de almacenamiento de acceso directo (DAAD). Disco, u otro dispositivo de almacenamiento secundario, que

permite el acceso a un sector o bloque de datos específico sin requerir primero la lectura de los bloques que lo preceden.

Extensión. Uno o más cúmulos adyacentes asignados como parte (o total) de un archivo. El número de extensiones de un archivo refleja grado de diseminación sobre el disco. Cuanto más diseminado esté un archivo, más desplazamientos se requerirán para moverse de una parte del archivo a otra.

Factor de bloque. El número de registros almacenados en un bloque.

Factor de intercalación. Puesto que muchas veces no es posible leer sectores físicamente adyacentes de un disco, algunas veces los sectores que deben ser adyacentes *lógicamente* se acomodan de tal forma que no sean adyacentes físicamente. A esto se le llama intercalación. El *factor de intercalación* se refiere al número de sectores físicos que hay entre el siguiente sector lógicamente adyacente y el sector que se está leyendo o escribiendo.

Fragmentación. Espacio que se vuelve inútil dentro de un cúmulo, bloque, pista u otra unidad de almacenamiento físico. Por ejemplo, la fragmentación en pistas ocurre cuando el espacio en una pista se queda sin uso por ser insuficiente para acomodar un bloque completo.

Hueco entre bloques. Intervalo de espacio en blanco que separa sectores, bloques o sub-bloques en cinta o disco. En el caso de la cinta, este espacio proporciona suficiente espacio para que la cinta acelere o desacelere cuando arranca o se detiene. Tanto en cintas como en discos, los huecos permiten que las cabezas de lectura y escritura determinen con precisión cuándo termina un sector (o bloque o sub-bloque) y comienza otro.

Marco. "Rebanada" de cinta de un bit de ancho, que normalmente representa un byte.

Organización por bloques. Organización de la unidad de disco que permite al usuario definir el tamaño y la organización de los bloques, y después acceder a un bloque dada su dirección o la llave de uno de sus registros. Véase *Organización por sectores*.

Organización por sectores. Organización de unidades de disco que usa sectores.

Paquete de discos. Empalme de discos magnéticos montados en el mismo eje. Un paquete de discos se trata como una unidad que consiste en un número de cilindros equivalente al número de pistas por superficie. Si los paquetes de discos son *removibles*, pueden montarse distintos paquetes en la misma unidad, en diferentes momentos, para proporcionar así una forma conveniente de almacenamiento de datos fuera de línea con acceso directo.

Paridad. Técnica de revisión de errores en la que un bit adicional de paridad acompaña a cada byte, y se activa de tal forma que el número total de bits en 1 sea par (paridad par) o impar (paridad impar).

Pista. El conjunto de bytes en la superficie de un disco al cual puede accederse sin mover el brazo de acceso. Se puede imaginar la superficie de un disco como una serie de círculos concéntricos, donde cada círculo corresponde a una posición particular del brazo de acceso y de las cabezas de lectura y escritura. Cada uno de estos círculos es una pista.

Plato. Disco de la pila en una unidad de disco.

Procesador de E/S. Dispositivo que lleva a cabo tareas de E/S, para que la UCP realice tareas que no sean de E/S.

Retraso por rotación. Tiempo que tarda en girar el disco hasta que el sector deseado queda bajo la cabeza de lectura y escritura.

Sector. Los bloques de datos de tamaño fijo que componen en conjunto las pistas, en ciertas unidades de disco. Los sectores son las unidades referenciables más pequeñas en un disco cuyas pistas están hechas de sectores.

Sistema de almacenamiento masivo. Término general que se aplica a las unidades de almacenamiento de gran capacidad. También se aplica a los sistemas de almacenamiento secundario de gran capacidad que pueden transmitir en unos cuantos segundos datos entre un disco y uno cualquiera de varios miles de cartuchos de cinta.

Sub-bloque. Cuando se emplea el manejo por bloques, casi siempre existen agrupamientos separados de información correspondientes a cada bloque en particular. Por ejemplo, pueden presentarse juntos un sub-bloque de conteo, uno de llave y uno de datos.

Sub-bloque de conteo. En unidades de disco organizadas por bloques, es un pequeño bloque que precede a cada bloque de datos y contiene información acerca de él, como su tamaño en bytes y su dirección.

Sub-bloque de llave. En unidades de disco que hacen referencias por bloque, es un bloque que contiene la llave del último registro en el bloque de datos que le sigue, que permite que la unidad busque, entre los bloques de una pista, el que contenga una determinada llave, sin tener que cargar los bloques en la memoria primaria.

Tabla de asignación de archivos (FAT, por sus siglas en inglés). Tabla que contiene un esquema de las posiciones físicas de todos los cúmulos de todos los archivos que contiene el almacenamiento en disco.

Tasa de transmisión efectiva. Razón de transmisión resultante después de considerar el tiempo empleado en detectar y transmitir un bloque de datos en donde se encuentra un registro deseado.

Tasa de transmisión nominal. Razón de transmisión de una unidad de disco o cinta que no considera los efectos de operaciones adicionales, como el tiempo de desplazamiento, para discos, o el tiempo de recorrido de los huecos entre bloques, para cintas.

Tiempo de desplazamiento. El tiempo requerido para mover el brazo de acceso al cilindro correcto en una unidad de disco.

Tiempo de transferencia. Cuando los datos que se desean están bajo la cabeza de lectura y escritura, se debe esperar a que vayan pasando bajo la cabeza mientras se leen. El tiempo de transferencia es la suma del tiempo requerido para realizar este movimiento y la lectura.

Unidad de cinta de grabado continuo (*streamer*). Unidad de cinta cuyo propósito principal es vaciar grandes cantidades de datos de disco a cinta o viceversa.

EJERCICIOS

1. Intenta describir de la mejor manera posible el viaje de un byte en su sistema. Se pueden consultar los manuales de referencia técnica que describen el sistema de administración de archivos, el sistema operativo y los dispositivos periféricos de su computador. También se puede consultar a un especialista o gurú que tenga experiencia con el sistema.

2. Averigüe cuáles rutinas de utilería están disponibles en su sistema de cómputo para la supervisión del desempeño de E/S y la utilización del disco. Si se tiene un sistema de cómputo grande, existen distintas rutinas para diferentes clases de usuarios, dependiendo de los privilegios y las responsabilidades que tengan.

3. Cuando se crea o se abre un archivo en C o en Pascal, debe proporcionarse cierta información al administrador de archivos del computador, de tal forma que pueda manejar el archivo apropiadamente. Comparada con ciertos lenguajes, como PL/I o COBOL, la cantidad de información que debe proporcionarse en C o en Pascal es muy poca. Consulte un texto o manual de PL/I o COBOL, y examine el atributo de descripción de archivos ENVIRONMENT, el cual puede servir para indicar al administrador de archivos muchos aspectos sobre cómo se espera que

un archivo se organice y se use. Compare PL/I y COBOL con C o Pascal en lo referente a los tipos de especificaciones de archivo disponibles para el programador.

4. Una unidad de disco utiliza sectores de 512 bytes. Si un programa solicita que un registro de 128 bytes se escriba en el disco, es probable que el administrador de archivos tenga que *leer* un sector del disco antes de escribir el registro. ¿Por qué? ¿Qué haría usted para disminuir el número de veces que pueda ocurrir dicha lectura adicional?

5. Se sabe que algunos sistemas operativos distribuyen el espacio de almacenamiento en disco en cúmulos y/o en extensiones, en lugar de hacerlo en sectores, de manera que el tamaño de cualquier archivo debe ser un múltiplo de un cúmulo o una extensión.

- a) ¿Cuáles son algunas ventajas y desventajas potenciales de este método de asignación de espacio en el disco?
- b) ¿Qué tan apropiado sería el uso de extensiones grandes para una aplicación que implica principalmente acceso secuencial de archivos muy grandes?
- c) ¿Qué tan apropiado sería el uso de extensiones grandes para un sistema de cómputo que da servicio a un gran número de programadores de C? (Los programas en C tienden a ser pequeños, de modo que es probable que haya muchos archivos pequeños que contengan programas en C.)
- d) El sistema administrador de registros de VAX por omisión usa un tamaño de cúmulo de tres sectores de 512 bytes, pero permite que el usuario vuelva a dar formato a la unidad con cualquier tamaño de cúmulo desde 1 hasta 65 535 sectores. ¿Cuándo es deseable un tamaño de cúmulo mayor a tres sectores? ¿Cuándo es deseable un tamaño de cúmulo más pequeño?

6. La unidad de disco IBM 3350 hace referencias por bloque. Se dispone de las dos organizaciones de sub-bloques descritas en el texto:

Datos de conteo, donde el espacio adicional empleado por el sub-bloque de conteo y los huecos entre bloques es equivalente a 185 bytes, y

Datos de conteo y de llave, donde el espacio adicional empleado por los sub-bloques de conteo y de llave, y por los espacios que los acompañan, es equivalente a 267 bytes, más el tamaño de la llave.

Una unidad IBM 3350 tiene 19 069 bytes útiles disponibles por pista, 30 pistas por cilindro y 555 cilindros por unidad de disco. Suponga que

se desea almacenar un archivo de 350 000 registros de 80 bytes en una unidad 3350. Conteste las siguientes preguntas. A menos que se indique otra cosa, suponga que el factor de bloque es de 10 y que se usa la organización de sub-bloques de conteo y de datos.

- a) ¿Cuántos bloques pueden almacenarse en una pista? ¿Cuántos registros?
- b) ¿Cuántos bloques pueden almacenarse en una pista si se usa la organización de sub-bloques de datos de conteo y llave?
- c) Haga una gráfica que muestre el efecto del tamaño del bloque en la utilización del almacenamiento, suponiendo sub-bloques de datos de conteo. Use la gráfica para predecir el mejor y el peor factor de bloque posible, en términos de utilización del almacenamiento.
- d) Suponiendo que el acceso al archivo siempre es secuencial, use la gráfica de la pregunta anterior para predecir el mejor y el peor factor de bloque. Justifique su respuesta en términos de eficiencia en la utilización del almacenamiento y el tiempo de procesamiento.
- e) ¿Cuántos cilindros se requieren para almacenar el archivo (factor de bloque de 10 y formato de datos de conteo)? ¿Cuánto espacio resultará inútil debido a la fragmentación interna de la pista?
- f) Si el archivo fuera almacenado en cilindros contiguos y no hubiera interferencia de otros procesos al usar la unidad de disco, el tiempo promedio de desplazamiento para un acceso aleatorio al archivo sería de alrededor de 12 mseg. Con esto calcule el tiempo promedio necesario para acceder a un registro en forma aleatoria.
- g) Explique cómo influye el incremento del tamaño del bloque en el tiempo de extracción de información para accesos aleatorios a registros. Analice los compromisos entre eficiencia y extracción de información cuando se emplean diferentes tamaños de bloque. Haga una tabla de los distintos tamaños de bloque para ilustrar sus explicaciones.
- h) Suponga que el archivo será ordenado según una clasificación Shell. Como el archivo es demasiado grande para trasladarlo a la memoria, se clasificará en el lugar donde reside, en el disco. Se estima (Knuth, 1973b, p. 380) que esto requiere alrededor de $15N^{1.25}$ movimientos de registros, donde N representa el número total de registros del archivo. Cada movimiento requiere un acceso aleatorio. Si todo esto es cierto, ¿cuánto tiempo tomará clasificar el archivo? (Como se observará, ésta

no es una buena solución. Se proporcionan soluciones mucho mejores en el capítulo de procesamiento secuencial concurrente.)

7. Una unidad de disco sectorizada se diferencia de una organizada por bloques en que hay menos correspondencia entre las organizaciones lógica y física de los registros o bloques de datos.

Por ejemplo, considere la unidad de disco Digital RM05, que usa referencia por sectores. Tiene 32 sectores de 512 bytes por pista, 19 pistas por cilindro y 823 cilindros en la unidad. Desde el punto de vista de la unidad (y de su controlador), un archivo sólo es un vector de bytes dividido en sectores de 512. Como la unidad no sabe dónde termina un registro y empieza otro, un registro puede distribuirse en dos o más sectores, pistas o cilindros.

Una forma común de dar formato a los registros en la RM05 consiste en colocar un campo de dos bytes al principio de cada bloque, que proporciona el número de bytes de datos, seguido de los datos. No hay huecos adicionales, ni ninguna otra cosa. Suponiendo que se usa esta organización, y que se desea almacenar un archivo con 350 000 registros de 80 bytes, conteste las siguientes preguntas.

- a) ¿Cuántos registros pueden almacenarse en una pista si se almacena un registro por bloque?
- b) ¿Cuántos cilindros se requieren para contener el archivo?
- c) ¿Cómo se pueden poner los registros en bloques de suerte que por cada acceso a un registro físico se acceda a diez registros reales? ¿Qué beneficios se obtiene con esto?

8. Considere la lista de correo de 1 000 000 de registros que se analizó con anterioridad. El archivo se respaldará en cintas de 2400 pies de 6250 bpi, con un hueco entre bloques de 0.3 pulgadas. La velocidad de la cinta es de 200 pulgadas por segundo.

- a) Muestre que sólo se requiere una cinta para respaldar el archivo, si se usa un factor de bloque de 50.
- b) Si se usa un factor de bloque de 50, ¿cuántos registros adicionales podrían acomodarse en una cinta de 2400 pies?
- c) ¿Cuál es la densidad de grabado efectiva cuando se emplea un factor de bloque de 50?
- d) ¿Qué tan grande debe ser el factor de bloque para alcanzar la máxima densidad de grabado efectiva? ¿Qué resultados negativos puede tener el incremento del factor de bloque?
(Nota: debe asignarse un buffer de E/S lo suficientemente grande para guardar un bloque.)

- e) ¿Cuál sería el factor de bloque *mínimo* requerido para que el archivo quepa en la cinta?
- f) Si se usa un factor de bloque de 50, ¿cuánto tiempo tomaría leer un bloque, incluido el hueco? ¿Cuál sería la tasa de transmisión efectiva? ¿Cuánto tiempo tomaría leer el archivo completo?
- g) ¿Cuánto tiempo tomaría efectuar la búsqueda binaria de un registro en el archivo, suponiendo que no es posible leer hacia atrás en la cinta? (Suponga que toma 60 segundos rebobinar la cinta desde el final hasta el principio.) Compare esto con el tiempo medio esperado para la búsqueda secuencial de un registro.
- h) Cuando se analizó el desempeño de la cinta, implícitamente se hizo la suposición de que la unidad de cinta siempre está leyendo o escribiendo a toda velocidad, así que no se pierde tiempo por el arranque o la detención. Pero no siempre ocurre así. Por ejemplo, algunas unidades se detienen automáticamente después de escribir cada bloque.

Suponga que el tiempo adicional que implica arrancar antes de iniciar la lectura de un bloque, y detenerse después de leerlo, suma 1 mseg, y que la unidad debe arrancar antes y detenerse después de la lectura de cada bloque. ¿Cuánto disminuirá la tasa de transmisión efectiva debido al arranque y la detención, si el factor de bloque es de 1? ¿Cuánto si es de 50?

- 9.** El uso de bloques grandes puede conducir a una severa fragmentación interna de pistas en los discos. ¿Ocurre lo mismo cuando se usan cintas? Explique.

LECTURAS ADICIONALES

Muchos libros de texto contienen información más detallada sobre el material que cubre este capítulo. En el área de los sistemas operativos y sistemas administradores de archivos, son útiles los textos de sistemas operativos de Deitel [1974], Peterson y Silberschatz [1985] y Madnick y Donovan [1974]. Hanson [1982] tiene abundante material sobre el manejo de bloques y buffers, dispositivos de almacenamiento secundario, y desempeño. El libro de Flores [1973] sobre dispositivos periféricos puede resultar un tanto anticuado, pero ofrece un tratamiento completo del tema.

Bohl [1981] proporciona un tratamiento completo de DAAD de IBM orientados a máquinas grandes. Chaney y Johnson [1984] es un buen artículo sobre la maximización del desempeño de los discos fijos en computadores pequeños. Ritchie y Thompson [1974], Kernighan y Ritchie [1978], Deitel [1984] y McKusick *et al.* [1974] proporcionan información sobre el manejo de la E/S de archivos en el sistema operativo UNIX. Este último, además, presenta un buen estudio sobre las formas en que un sistema de archivos puede ser alterado para proporcionar un desempeño considerablemente más rápido en ciertas aplicaciones.

La información sobre sistemas específicos y dispositivos suele encontrarse en los manuales y documentos publicados por los fabricantes. (Por desgracia, la información sobre cómo trabaja en realidad el software por lo común está patentada y, por tanto, no está disponible.) Si se usa una VAX, son recomendables los manuales *Introduction to the VAX Record Management Services* (Digital, 1978), *VAX Software Handbook* (Digital, 1982) y *Peripherals Handbook* (Digital, 1981). Para los usuarios de UNIX será útil la monografía publicada por los Laboratorios Bell, *The UNIX I/O System*, de Dennis Ritchie [1978]. Los usuarios de las PC de IBM encontrarán útiles los manuales de IBM *Disk Operating System* (Microsoft, 1983 o posteriores) y *Technical Reference* (IBM, 1983 o posteriores).

4

CONCEPTOS FUNDAMENTALES DE ESTRUCTURAS DE ARCHIVOS

OBJETIVOS

Introducir los conceptos de estructuras de archivos que tratan con:

- Archivos como secuencia de bytes;
- Límites de campos y registros;
- Campos y registros de longitudes fijas y variables;
- Llaves de búsqueda y formas canónicas;
- Búsqueda secuencial;
- Acceso directo, y
- Acceso y organización de archivos.

PLAN GENERAL DEL CAPITULO

4.1 Un archivo como secuencia de bytes

4.2 Estructuras de campos

4.2.1 Método 1: Fijar la longitud de los campos

4.2.2 Método 2: Comenzar cada campo con un indicador de longitud

4.2.3 Método 3: Separar los campos con delimitadores

4.3 Lectura de una secuencia de campos

4.4 Estructuras de registros

4.4.1 Método 1: Hacer los registros de una longitud predecible

4.4.2 Método 2: Comenzar cada registro con un indicador de longitud

4.4.3 Método 3: Usar un segundo archivo para mantener información sobre las direcciones

4.4.4 Método 4: Colocar un delimitador al final de cada registro

4.5 Una estructura de registros que usa un indicador de longitud

4.6 Mezcla de números y caracteres: Uso de un vaciado hexadecimal

4.7 Lectura de registros de longitud variable de un archivo

4.8 Extracción de registros por llave: formas canónicas para llaves

4.9 Una búsqueda secuencial

4.10 Evaluación del desempeño de la búsqueda secuencial

4.11 Mejora del desempeño de la búsqueda secuencial: manejo de registros en bloques

4.12 Acceso directo

4.13 Elección de una estructura y una longitud de registro

4.14 Registros de encabezado

4.15 Acceso y organización de archivos

Programas en C

Programas en Pascal

4.1

UN ARCHIVO COMO SECUENCIA DE BYTES

Cuando se construyen estructuras de archivos se impone un orden sobre los datos. En este capítulo se investigarán las múltiples formas que puede tomar este orden. Se comienza examinando el caso desde la base: un archivo organizado como una secuencia de bytes.

Suponga que el archivo que se construye contiene información sobre nombres y direcciones. En la figura 4.1 se muestra el pseudocódigo que describe un programa que recibe nombres y direcciones del teclado, y los transcribe como una secuencia de bytes consecutivos a un archivo con nombre lógico SALIDA.

Las realizaciones de este programa, tanto en C como en Pascal, llamadas *escribesec.c* y *escribesec.pas*, se presentan al final de este capítulo. El lector debe digitar este programa, ya sea en C o en Pascal, compilarlo y ejecutarlo. Se usa como base de numerosos experimentos, y se podrá comprender mejor las diferencias entre las estructuras de archivos que se están analizando si se llevan a cabo los experimentos.

Los siguientes nombres y direcciones se usan como entrada al programa:

John Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater, OK 74075	Ada, OK 74820

Cuando se lista el archivo de salida en la pantalla de la terminal, se ve lo siguiente:

```
AmesJohn123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
```

PROGRAMA: escribesec

Lee el nombre del archivo de salida y lo abre con el nombre lógico SALIDA

Lee APELLIDO

mientras (APELIDO tenga longitud > 0)

 lee NOMBRE, DIRECCION, CIUDAD, ESTADO y CP

 escribe APELLIDO en el archivo SALIDA

 escribe NOMBRE en el archivo SALIDA

 escribe DIRECCION en el archivo SALIDA

 escribe CIUDAD en el archivo SALIDA

 escribe ESTADO en el archivo SALIDA

 escribe CP en el archivo SALIDA

 Lee APELLIDO

fin mientras

cierra SALIDA

FIGURA 4.1 • Programa para escribir un archivo de nombres y direcciones como una secuencia de bytes.

El programa transcribe la información al archivo tal como se especificó: como una secuencia de bytes que no contiene información adicional. Pero al cumplir con las especificaciones, el programa crea un problema de tipo "Humpty-Dumpty al revés". Una vez que se pone toda la información junta como una secuencia de bytes, no hay forma de separarla de nuevo.

Lo que sucede es que se ha perdido la integridad de las unidades organizacionales fundamentales de los datos de entrada; estas unidades fundamentales no son los caracteres individuales, sino los agregados significativos de caracteres, tales como "John Ames" o "123 Maple". Cuando se trabaja con archivos, se denomina *campos* a esos agregados fundamentales. Un campo es *la unidad de información lógicamente significativa más pequeña en un archivo*.[†]

Un campo es una idea lógica, una *herramienta conceptual*. Un campo no necesariamente existe en algún sentido físico, pero aun así es importante para la estructura del archivo. Cuando la información sobre el nombre y la dirección se transcribe como una secuencia de bytes no diferenciables, se pierde el rastro de los campos que le dan significado a la información. Es necesario organizar el archivo de manera que la información se mantenga dividida en campos.

4.2

ESTRUCTURAS DE CAMPOS

Hay muchas formas de añadir estructura a los archivos para mantener la identidad de los campos. Los tres métodos más comunes son:

- Forzar que los campos tengan una longitud predecible;
- Comenzar cada campo con un indicador de longitud, y
- Colocar un *delimitador* al final de cada campo para separarlo del siguiente.

[†] No se deben confundir los términos *campo* y *registro* con los significados que algunos lenguajes de programación les han dado, incluso Pascal. En Pascal, un registro es una estructura de datos agregada que puede contener miembros de tipos diferentes, donde a cada miembro se le denomina campo. Como se verá, con frecuencia hay una correspondencia directa entre esas definiciones de los términos y los campos y registros que se usan en los archivos. Sin embargo, los términos *campo* y *registro*, como se emplean en el texto, tienen significados mucho más generales que los que tienen en Pascal.

4.2.1 METODO 1: FIJAR LA LONGITUD DE LOS CAMPOS

La longitud de los campos varía en nuestro archivo de ejemplo. Si se fuerza que los campos tengan longitudes predecibles, entonces se pueden recuperar del archivo con sólo contar hasta el final del campo. Se puede definir una estructura en C o un registro en Pascal para que tenga esos campos de longitud fija, como se muestra en la figura 4.2.

Con esta clase de estructura de campos de longitud fija la salida cambia, de modo que se ve como se muestra en la figura 4.3(a). Basta con aritmética simple para poder recuperar los datos en términos de los campos originales.

Una desventaja obvia de este enfoque es que al agregar el relleno requerido para llevar los campos a una longitud fija, el archivo crece mucho. En lugar de usar cuatro bytes para almacenar el apellido Ames, se usan diez. También se pueden presentar problemas con datos que sean tan grandes que no quepan en el espacio asignado. Este segundo problema se resolvería dando a los campos longitudes lo suficientemente grandes como para cubrir todos los casos, pero esto agravaría aún más el problema de espacio desperdiciado en el archivo.

Debido a estas dificultades, el enfoque de campos fijos para la estructuración de datos con frecuencia resulta inapropiado para datos que inherentemente tienen campos de longitud muy variable, como nombres y direcciones. Pero hay tipos de datos para los cuales los campos de longitud fija son muy apropiados. Si cada campo tiene una longitud fija, o con poca variación, una buena solución es emplear una estructura que consista en una secuencia continua de bytes organizados en campos de longitud fija.

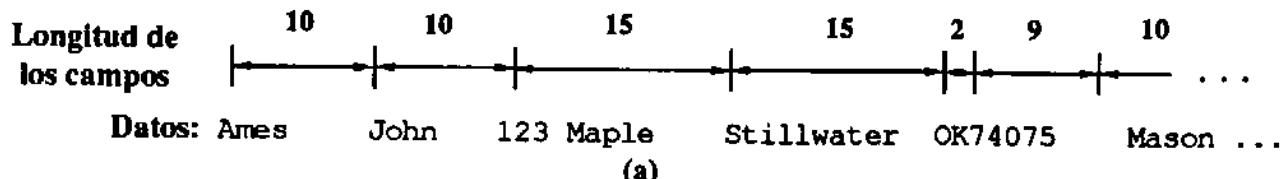
En C:

```
struct {
    char apellido[10];
    char nombre[10];
    char dirección[15];
    char ciudad[15];
    char estado[2];
    char cp[9];
} conj_de_campos;
```

En Pascal:

```
TYPE
conj_de_campos = RECORD
    apellido : packed array [1..10] of char;
    nombre   : packed array [1..10] of char;
    dirección : packed array [1..15] of char;
    ciudad   : packed array [1..15] of char;
    estado   : packed array [1..2] of char;
    cp       : packed array [1..9] of char;
END;
```

FIGURA 4.2 • Registros de longitud fija.



04Ames04John09123 Maple 10 Stillwater 020K057407509Mason04Alan1190 Eastgate ...
(b)

Ames;John;123 Maple;Stillwater;OK;74075;Mason;Alan;90Eastgate;Ada;OK;74820
(c)

FIGURA 4.3 • Tres métodos para la organización de estructuras de campos. (a) Campos de longitud fija. (b) Cada campo se inicia con un indicador de longitud. (c) Los campos se separan con delimitadores.

4.2.2 METODO 2: COMENZAR CADA CAMPO CON UN INDICADOR DE LONGITUD

Otra forma de contar hasta el final de un campo es almacenar su longitud delante del campo, como se ilustra en la figura 4.3(b). Si los campos no son demasiado largos (longitudes de menos de 256 bytes), es posible almacenar la longitud en un solo byte al inicio de cada campo.

4.2.3 METODO 3: SEPARAR LOS CAMPOS CON DELIMITADORES

Para preservar la identidad de los campos, éstos también se pueden separar con delimitadores. Todo lo que se necesita hacer es elegir algún carácter especial que no aparezca como un carácter legítimo dentro de un campo e *insertar* ese carácter dentro del archivo luego de escribir cada campo.

La elección del carácter delimitador es muy importante, ya que debe ser un carácter que no se confunda con lo que se está procesando. Por ejemplo, el carácter coma sería una mala elección para este archivo, porque las comas aparecen con frecuencia como caracteres legales dentro de un campo de dirección. En esta sección se usará el carácter de barra vertical como delimitador, de modo que el archivo aparece como en la figura 4.3(c). Se deberán modificar los programas originales para secuencias de bytes, *scribesec.c* y *scribesec.pas* (que se encuentran al final del capítulo), de forma que coloquen un delimitador después de cada campo. Se usa este formato de campo delimitado en los siguientes ejemplos de programas.

4.3

LECTURA DE UNA SECUENCIA DE CAMPOS

Con las versiones modificadas de *escribese.c* y *escribese.pas*, que usan delimitadores para separar los campos, se puede escribir un programa llamado *leesec*, que lee la secuencia de bytes y la divide en campos. Es conveniente concebir el programa en dos niveles, como se muestra en la descripción en pseudocódigo de la figura 4.4. El nivel más externo del

```
Definición de constante: DELIMITADOR = ' '|'

PROGRAMA: leesec
    Lee el nombre del archivo de entrada y lo abre como ENTRADA
    inicia CONT_CAMPOS
    LONG_CAMPO := leecampo(ENTRADA, CONTENIDO_CAMPO);
    mientras ( LONG_CAMPO > 0 )
        incrementa el CONT_CAMPOS
        escribe CONT_CAMPOS y CONTENIDO_CAMPO en la pantalla
        LONG_CAMPO := leecampo(ENTRADA, CONTENIDO_CAMPO);

    fin mientras
    cierra ENTRADA
fin PROGRAMA

FUNCION: leecampo(ENTRADA, CONTENIDO_CAMPO)
    inicia I
    inicia CH
    mientras (no EOF (ENTRADA) y CAR distinto de DELIMITADOR)

        lee un carácter de ENTRADA en CAR
        incrementa I
        CONTENIDO_CAMPO[I] := CAR

    fin mientras
    devuelve (longitud del campo que se leyó)

fin FUNCION
```

FIGURA 4.4 • Programa que lee campos de un archivo y los despliega en la pantalla.

programa abre el archivo y después llama a la función *leecampo()*, hasta que ésta devuelve un campo con longitud cero, lo cual indica que no existen más campos por leer. La función *leecampo()*, a su vez, trabaja a lo largo del archivo, carácter por carácter, recolectando caracteres dentro de un campo hasta que encuentra un delimitador o el fin del archivo. La función devuelve el número de caracteres que se encontraron en el campo. Las realizaciones de *leesec*, tanto en C como en Pascal, se incluyen con los programas al final del capítulo.

Cuando este programa se ejecuta, usando la versión del archivo con campos delimitados que contiene los datos de John Ames y Alan Mason, la salida se ve así:

```
Campo # 1: Ames
Campo # 2: John
Campo # 3: 123 Maple
Campo # 4: Stillwater
Campo # 5: OK
Campo # 6: 74075
Campo # 7: Mason
Campo # 8: Alan
Campo # 9: 90 Eastgate
Campo # 10: Ada
Campo # 11: OK
Campo # 12: 74820
```

Es evidente que ahora se preserva la noción del campo conforme se almacenan y extraen esos datos. Pero todavía falta algo. En realidad este archivo no se concibió como una secuencia de campos; de hecho los campos necesitan agruparse en conjuntos. Los primeros seis campos constituyen un conjunto asociado con alguien llamado John Ames; los siguientes seis son un conjunto asociado con Alan Mason. A estos conjuntos de campos se les llama *registros*.

4.4

ESTRUCTURAS DE REGISTROS

Un *registro* puede definirse como *un conjunto de campos agrupados bajo la perspectiva de un archivo de nivel más alto de organización en un archivo*. Al igual que la noción de campo, un registro es otra herramienta conceptual. Es otro nivel de organización que se impone sobre los datos para preservar su significado. Los registros no necesariamente

existen en el archivo en un sentido físico; sin embargo, constituyen una noción lógica importante incluida en la estructura del archivo.

Presentamos a continuación algunos de los métodos que se usan con mayor frecuencia para organizar un archivo en registros:

- Exigir que los registros sean de longitud predecible. Esta longitud puede medirse en términos de bytes o en términos del número de campos.
- Comenzar cada registro con un indicador de longitud que señale el número de bytes que contiene.
- Usar un segundo archivo para mantener información de la dirección del byte de inicio de cada registro.
- Colocar un delimitador al final de cada registro, para separarlo del siguiente.

4.4.1 METODO 1: HACER LOS REGISTROS DE UNA LONGITUD PREDECIBLE

Hacer un registro de longitud predecible permite mantener la cuenta dentro del registro. Cuando el conteo alcanza una cantidad predeterminada, se sabe que el registro se ha leído por completo; cualquier lectura adicional conduciría al siguiente registro. Este método para reconocer registros es análogo al primer método analizado para hacer reconocibles los campos, que implicaba fijar la longitud del campo. La diferencia importante radica en qué, cuando se cuenta a lo largo de un registro se eligen las unidades de conteo; se pueden contar bytes o campos.

CONTEO DE BYTES: REGISTROS DE LONGITUD FIJA. Un *archivo con registros de longitud fija* es aquel cuyos registros contienen todos el mismo número de bytes. Como se verá en los capítulos siguientes, las estructuras de registros de longitud fija son uno de los métodos más usados para organizar archivos .

La estructura de C *conjunto_de_campos* (o el RECORD de Pascal del mismo nombre) que se definió en el análisis de los campos de longitud fija en realidad es un ejemplo de un *registro de longitud fija*, así como un ejemplo de campos de longitud fija. Se tiene un número fijo de campos, cada uno con una longitud predeterminada, los cuales se

combinan para hacer un registro de longitud fija. Esta clase de estructura de campos y registros se ilustra en la figura 4.5(a).

Sin embargo, es importante comprender que fijar el número de bytes en un registro no implica de ninguna manera que los tamaños o el número de campos deban ser fijos. Los registros de longitud fija se usan con mucha frecuencia como recipientes para guardar un número variable de campos de longitud variable. También es posible mezclar campos de longitud fija y variable dentro de un registro. La figura 4.5(b) muestra cómo pueden colocarse campos de longitud variable en un registro de longitud fija.

CAMPOS DE CONTEO. En lugar de especificar que cada registro en un archivo contiene un número fijo de bytes, se puede especificar que contendrá un número fijo de campos. Esta es la forma más simple de organizar los registros en el archivo de nombres y direcciones que se analizó con anterioridad. El programa *escribesec* pide seis partes de información para cada persona, de tal forma que en el archivo hay seis campos contiguos por cada registro (Fig. 4.5c). Se podría modificar *leesec* para reconocer los campos por un simple conteo de campos *módulo seis*, sacando la información delimitada del registro a la pantalla cada vez que el conteo empieza de nuevo.

Ames	John	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

(a)

Ames;John;123 Maple;Stillwater;OK;74075;	Unused space
Mason;Alan;90 Eastgate;Ada;OK;74820;	Unused space

(b)

Ames;John;123 Maple;Stillwater;OK;74075;Mason;Alan;90 Eastgate;Ada;OK . . .

(c)

FIGURA 4.5 • Tres formas de volver constantes y predecibles las longitudes de los registros. (a) Conteo de bytes: registros de longitud fija con campos de longitud fija. (b) Conteo de bytes: registros de longitud fija con campos de longitud variable. (c) Conteo de campos: seis campos por registro.

4.4.2 METODO 2: COMENZAR CADA REGISTRO CON UN INDICADOR DE LONGITUD

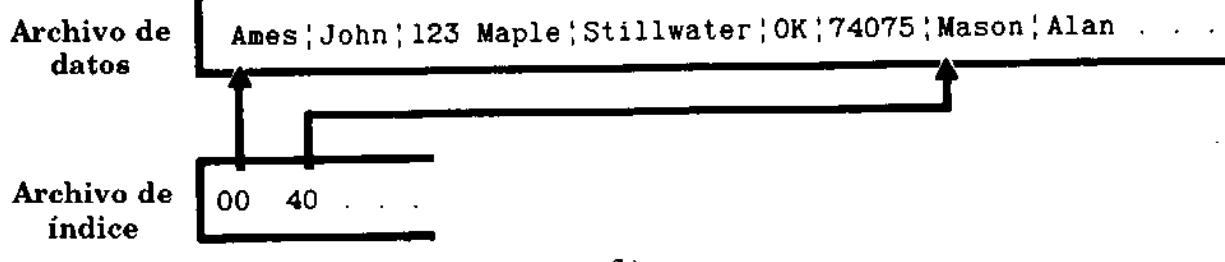
Se sabe cuántos bytes hay en los registros de longitud fija porque la longitud del registro es una constante predeterminada a lo largo del archivo. También se puede transmitir la longitud de los registros, y hacerlos reconocibles, comenzando cada registro con un campo que contenga un número entero que indique cuántos bytes hay en el resto del registro (Fig. 4.6a). Éste es un método muy común en el manejo de registros de longitud variable. Se examinan con mayor detalle en la siguiente sección.

4.4.3 METODO 3: USAR UN SEGUNDO ARCHIVO PARA MANTENER INFORMACION SOBRE LAS DIRECCIONES

Se puede emplear un segundo archivo de *índice* para mantener información sobre la distancia en bytes de cada registro en el archivo original. La distancia en bytes permite encontrar el comienzo de cada registro sucesivo y calcular su longitud. Se busca la posición de un registro en el índice y después se alcanza el registro dentro del archivo de datos. La figura 4.6(b) ilustra este mecanismo de dos archivos.

40Ames;John;123 Maple;Stillwater;OK;74075;36 Mason;Alan;90 Eastgate . . .

(a)



(b)

Ames;John;123 Maple;Stillwater;OK;74075;#Mason;Alan;90 Eastgate;Ada;OK . . .

(c)

FIGURA 4.6 • Estructuras de registros para registros de longitud variable. (a) Se inicia cada registro con un indicador de longitud. (b) Se usa un archivo de índice para mantener la información de las direcciones de los registros. (c) Se coloca el delimitador '#' al final de cada registro.

4.4.4 METODO 4: COLOCAR UN DELIMITADOR AL FINAL DE CADA REGISTRO

Esta es la opción que, en el nivel de registro, es por completo similar a la solución que se dio para distinguir los campos en el programa del ejemplo que se desarrolló. La marca de fin de línea se usa frecuentemente como un delimitador de registro (la pareja de retorno de carro y avance de línea o, en sistemas UNIX, sólo un carácter de avance de línea ('\n')). En la figura 4.6(c) se usa un carácter '#' como delimitador de registro.

4.5

UNA ESTRUCTURA DE REGISTROS QUE USA UN INDICADOR DE LONGITUD

Ninguno de estos enfoques para preservar la idea de un *registro* dentro de un archivo es apropiado en todas las situaciones. La selección de un método de organización del registro depende de la naturaleza de los datos y de lo que se necesite hacer con ellos. Se comenzará examinando una estructura de registro que usa un campo de longitud al principio del registro. Este enfoque permite preservar la *variabilidad* de la longitud de los registros, inherente al archivo de secuencia de bytes inicial.

Se llamará *escribereg* al programa que construye esta nueva estructura de registros de longitud variable. El conjunto de programas que se presenta al final del capítulo contiene las versiones de este programa en C y en Turbo Pascal. La realización de este programa requiere, en parte, modificar el programa *escribesec* que se hizo con anterioridad en este capítulo, aunque también implica resolver algunos problemas nuevos:

- Si se quiere colocar un indicador de longitud al principio de cada registro (antes de cualquier campo) se debe conocer la suma de las longitudes de los campos de cada registro antes de empezar a transcribir el registro al archivo. Se necesita acumular el contenido completo de un registro en un *buffer* antes de escribirlo.
- ¿En qué forma debe transcribirse el campo de longitud del registro al archivo? ¿Como un entero binario? ¿Como una serie de caracteres ASCII?

Conforme se trabaja con archivos, se recurre una y otra vez al concepto de manejo de buffers. En el caso de *escribereg*, el buffer puede

```

lee APELLIDO
mientras ( APELLIDO tenga longitud > 0 )
    asigna la longitud de la cadena en BUFFER a cero
    concatena: BUFFER + APELLIDO + DELIMITADOR

    mientras ( existan campos por leer para el registro )
        lee el CAMPO
        concatena: BUFFER + CAMPO + DELIMITADOR
    fin mientras

    escribe la longitud de la cadena de BUFFER al archivo
    escribe la cadena de BUFFER al archivo

    lee APELLIDO
fin mientras

```

FIGURA 4.7 • Lógica del programa principal de *escribereg*.

ser un simple arreglo de caracteres dentro del cual se colocan los campos y sus delimitadores, conforme se recolectan. Para reiniciar la longitud del buffer en cero y añadir información al buffer se puede emplear el ciclo lógico que se proporciona en la figura 4.7.

La pregunta sobre cómo representar la longitud del registro es un poco más difícil. Una opción sería escribir la longitud en forma de un entero binario antes de cada registro. Esta es una solución natural en C, ya que no existe el problema de conversión de la longitud del registro a forma de caracteres. También desde el punto de vista conceptual es interesante, ya que ilustra el uso de un campo binario de longitud fija en combinación con campos de caracteres de longitud variable.

Aunque se podría dar esta misma solución en una realización en Pascal, es preferible explicar algunas diferencias importantes entre C y Pascal:

- A diferencia de C, Pascal convierte automáticamente los enteros binarios en representaciones de caracteres de esos enteros, si es que se escriben en un archivo de texto. En consecuencia, no existe el problema de convertir la longitud del registro a forma de caracteres: esto sucede automáticamente.
- En Pascal, un archivo se define como una secuencia de elementos de un solo tipo. Ya que se tiene un archivo de cadenas de caracteres de longitud variable, el tipo natural del archivo es el de carácter. Se *pueden* almacenar enteros de dos bytes en el archivo, pero entonces se deben dividir en sus valores de bytes constituyentes y convertirlos con la función CHR(). Sin embargo,

40 Ames | John | 123 Maple | Stillwater | OK | 74075 | 36 Mason | Alan | 90
Eastgate | Ada | OK | 74820:

FIGURA 4.8 • Registros precedidos por campos de longitud de registro en forma de caracteres.

es mucho más sencillo dejar que Pascal efectúe la conversión automática de la longitud a la forma de carácter.

En síntesis, lo más fácil en C es almacenar los enteros en el archivo como campos de dos bytes de longitud fija que contengan enteros. En Pascal es más fácil usar la conversión automática de enteros a caracteres para los archivos de texto. El diseño de la estructura de archivos siempre es un ejercicio de flexibilidad. Ninguno de estos enfoques es correcto; el buen diseño consiste en elegir el enfoque más *apropiado* para el lenguaje y ambiente determinados de computación. En los programas incluidos al final del capítulo se desarrolla la estructura de registros de las dos formas estudiadas, usando campos de longitud entera en C y representaciones de caracteres en Pascal. La salida de la realización en Pascal se muestra en la figura 4.8. (La salida pasa a la siguiente línea después de 64 caracteres.) En cada registro hay ahora un campo de longitud del registro que precede a los campos de datos. Este campo está delimitado por un espacio. Por ejemplo, el primer registro (para John Ames) contiene 40 caracteres, contando desde la primera 'A' en "Ames" hasta el delimitador final después de "74075", así que los caracteres '4' y '0' se colocan antes del registro, seguidos por un espacio.

Antes de examinar la salida de la versión de *escribereg* en C, donde se usan enteros binarios para las longitudes de registro, es necesario revisar el uso del vaciado hexadecimal para poder interpretar la parte que no tiene caracteres.

4.6

MEZCLA DE NUMEROS Y CARACTERES: USO DE UN VACIADO HEXADECIMAL

Los vaciados hexadecimales permiten examinar dentro de un archivo los bytes reales almacenados. Por ejemplo, considérese la información referida a la longitud de registro en la salida del programa en Pascal

Valor decimal del número	Valores hexade- cimales almacena- dos en bytes	Forma en caracteres ASCII
40	34 30	'4' '0'

FIGURA 4.9 • El número 40, almacenado como caracteres ASCII.

que se examinó anteriormente. La longitud del registro Ames, que es el primero del archivo, es de 40 caracteres, incluyendo los delimitadores. En la versión Pascal de *escribereg*, donde se almacenó la representación de caracteres ASCII para este número decimal, los bytes reales almacenados *dentro del archivo* se ven como en la representación de la figura 4.9.

Como se puede observar, no es lo mismo el *número* 40 que el conjunto de caracteres '4' y '0'. El valor hexadecimal del *entero binario* 40 es 0x28; los valores hexadecimales de los *caracteres* '4' y '0' son 0x34 y 0x30. (Se está empleando la convención del lenguaje C para identificar números hexadecimales, mediante el uso del prefijo 0x.) Así, cuando se almacena un número en forma ASCII, son los valores hexadecimales de los *caracteres ASCII* los que van dentro del archivo, y no el valor hexadecimal del número mismo.

En la realización en C se elige representar la longitud del campo para cada registro como un entero corto, en lugar de hacerlo con caracteres ASCII. Esto tiene dos ventajas:

1. Se puede representar números más grandes con un entero de dos bytes que con dos bytes ASCII (32767 contra 99).
2. No se necesita traducir la longitud del registro de ASCII a entero, cuando se pretende *usarla* como un entero.

La figura 4.10 muestra la representación en bytes del número 40 almacenado como un entero (a esto se le llama *almacenamiento del*

Valor decimal del número	Valores hexade- cimales almacena- dos en bytes	Forma en caracteres ASCII
40	00 28	"\0" "(

FIGURA 4.10 • El número 40, almacenado como un entero corto.

número en forma *binaria*, aun cuando usualmente se considera a la salida como un número hexadecimal). Ahora el valor hexadecimal almacenado en el archivo es el del número mismo. Los caracteres ASCII que se asocian al valor hexadecimal real del número no guardan una relación obvia con él.

Tomando en cuenta las explicaciones respecto a la diferencia entre almacenar números en forma de caracteres ASCII y almacenarlos como cantidades binarias, y considerando que los dos registros tienen longitudes de 40 (valor hexadecimal 0x28) y 36 (valor hexadecimal 0x24), la versión del archivo que usa enteros binarios para las longitudes de registro, en la pantalla de una terminal, se vería así:

```
( Ames ; John ; 123 Maple ; Stillwater ; OK ; 74075 ;$ Mason ; Alan ; 90 Eastgate...
```

Espacio, ya que '0' no es imprimible
0x28 es el código ASCII de ''

Espacio, por la misma razón
0x24 es el código ASCII de '\$'

Las representaciones ASCII de los caracteres y números en el registro real aparecen claras, pero las representaciones binarias de los campos de longitud se despliegan en forma rara. Nótese también que el orden de los bytes de los dígitos binarios está al revés. Se ve "(" en lugar de "(" para el número 40. Antes de analizar esta anomalía, se examinará el archivo en forma distinta, esta vez usando el vaciado hexadecimal.

Intervalo	Valores hexadecimales	Valores ASCII
0 - F 2800416D 65737C4A	6F686E7C 31323320	(.Ames ; John ; 123
10 - 1F 4D61706C 657C5374	696C6C77 61746572	Maple ; Stillwater
20 - 2F 7C4F4B7C 37343037	347C2400 4D61736F	:OK ; 74075 ;\$.Maso
30 - 3F 6E7C416C 616E7C39	30204561 73746761	n ; Alan ; 90 Eastga
40 - 4F 74657C41 64617C4F	4B7C3734 3832307C	te ; Ada ; OK ; 74820;

Como se puede observar, el despliegue creado por este programa en particular de vaciado hexadecimal se divide en tres clases diferentes de datos. El área del despliegue con el rótulo *Intervalo* indica las distancias de los bytes, que están dadas en forma hexadecimal; como cada línea contiene 16 (decimal) bytes, el movimiento de una línea a la siguiente agrega 0x10 al intervalo.

La parte del vaciado hexadecimal rotulada *Valores hexadecimales* contiene el valor hexadecimal de cada byte que se encuentra en el archivo. El tercer byte tiene el valor 0x41, que es el código ASCII del carácter 'A'. Cada espacio insertado en el archivo resulta en un byte con el valor 0x20, que es el código ASCII para el espacio en blanco.

Examinemos ahora los valores hexadecimales de los bytes que representan números. En algunos casos, como en 123, las representaciones son códigos ASCII (0x31, 0x32, 0x33); en otros, como en la longitud de registro 40 que comienza el archivo, son, en realidad, valores hexadecimales de dos bytes (0x28). Pero el número hexadecimal correspondiente a 40 parece ser 0x2800 en lugar de 0x0028. ¿Por qué están invertidos los bytes?

La respuesta es que este vaciado hexadecimal es el tipo de salida producida por un computador que almacena los valores de los números según un orden de bytes invertido. Para enteros de dos bytes, el byte menos significativo siempre se guarda en una dirección más baja que la del más significativo. De esta forma, el byte con 0x28 se coloca en una dirección de memoria que precede a la que se usó para 0x00. Este orden invertido también se aplica a los enteros largos de cuatro bytes en esas máquinas. Por ejemplo, el valor hexadecimal del número 500 000 000 es 0x1DCD6500. Si se transcribe este valor a un archivo en una máquina VAX, IBM PC, o alguna otra de orden invertido, un vaciado hexadecimal del archivo creado se verá así:

0 - F 0065CD1D 00000000 00000000 00000000 .e.....

No todos los computadores invierten el orden de los bytes en esta forma. Por ejemplo, las grandes máquinas IBM no lo hacen. Este es uno de los aspectos de archivos que requieren mucha atención, si se pretende que los vaciados hexadecimales tengan sentido. También es importante tener esto en cuenta cuando se intente transferir archivos que contengan datos binarios entre dos máquinas que ordenen los bytes de diferente manera. Nótese que el problema no aparece si se están empleando archivos ASCII, donde todos los números son representados como secuencias de caracteres.

Regresemos al vaciado hexadecimal del archivo de nombres y direcciones para observar la columna de la extrema derecha. Esta es la lista de los valores ASCII representados por los bytes del archivo. Como se esperaba, los datos colocados en el archivo en forma ASCII aparecen en esta columna en forma legible, pero existen valores hexadecimales para los que no hay representación ASCII imprimible. El único de tales valores que aparece en este archivo es 0x00, pero podría haber muchos otros. Si se examina el vaciado que contiene el número 500 000 000, se observará que el único byte imprimible es el que tiene el valor 0x65 ('e'). Este programa de vaciado hexadecimal en particular maneja todos los demás valores colocando un punto ('.') en la representación ASCII, para apartar el lugar para el valor del byte.

El vaciado hexadecimal de esta salida en la versión en C de *escriberreg* muestra cómo esta estructura de archivo representa una mezcla

interesante de varias de las herramientas organizacionales que se encuentran a lo largo del libro. En un solo registro se tienen datos tanto binarios como ASCII. Cada registro consiste en un campo de longitud fija (el número de bytes) y en varios campos de longitud variable delimitados. Esta mezcla de diferentes tipos de datos y métodos de organización es normal en las estructuras de archivos reales.

4.7

LECTURA DE REGISTROS DE LONGITUD VARIABLE DE UN ARCHIVO

Con la estructura del archivo de registros de longitud variable precedidos por campos de longitud de registro, es fácil escribir un programa que lea todo el archivo, registro por registro, desplegando los campos de cada uno de los registros en la pantalla. La lógica del programa se muestra en la figura 4.11. El programa principal llama a la función *toma_reg()* para que traslade los registros a un buffer; este llamado continúa hasta que *toma_reg()* devuelve un valor de 0. Una vez que *toma_reg()* coloca el contenido de un registro dentro de un buffer, se pasa el buffer a una función llamada *toma_campo()*. La llamada a *toma_campo()* implica una posición de búsqueda (POS_BUS) en la lista de argumentos. A partir de POS_BUS, *toma_campo()* traslada los caracteres del buffer hacia un campo hasta que encuentra un delimitador o se llega al final del registro. La función *toma_campo()* devuelve la POS_BUS para que se use en la siguiente llamada. Las realizaciones de *leereg*, tanto en C como en Pascal, se incluyen junto con los demás programas al final del capítulo.

4.8

EXTRACCION DE REGISTROS POR LLAVE: FORMAS CANONICAS PARA LLAVES

Como está claro que la nueva estructura de archivo analizada concibe un registro como la cantidad de información que se lee o escribe, tiene sentido pensar en extraer sólo un registro específico en lugar de leer el archivo completo, desplegándolo todo. Cuando se examina un registro individual, es conveniente identificarlo con una *llave* que se base en el contenido del registro. Por ejemplo, en el archivo de nombres y direcciones se puede desear tener acceso al “registro Ames”, o al “registro Mason”,

PROGRAMA: leereg

```
abre el archivo de entrada como ARCH_ENT
inicia POS_BUS con 0
LONG_REGISTRO := toma_reg(ARCH_ENT, BUFFER)
mientras (LONG_REGISTRO > 0)
    POS_BUS := toma_campo(CAMPO, BUFFER, POS_BUS, LONG_REGISTRO)
    mientras (POS_BUS > 0)
        escribe CAMPO en la pantalla
        POS_BUS := toma_campo(CAMPO, BUFFER, POS_BUS, LONG_REGISTRO)
    fin mientras
    LONG_REGISTRO := toma_reg(ARCH_ENT, BUFFER)
fin mientras
fin PROGRAMA
```

FUNCION: toma_reg(ARCH_ENT, BUFFER)

```
si EOF (ARCH_ENT) entonces devuelve 0
lee LONG_REGISTRO
lee el contenido del registro en BUFFER
devuelve LONG_REGISTRO

fin FUNCION
```

FUNCION: toma_campo(CAMPO, BUFFER, POS_BUS, LONG_REGISTRO)

```
si POS_BUS == LONG_REGISTRO entonces devuelve 0
toma un carácter CAR en la POS_BUS del BUFFER
mientras (POS_BUS < LONG_REGISTRO y CAR distinto de DELIMITADOR)
    coloca CAR en CAMPO
    incrementa POS_BUS
    toma un carácter CAR en la POS_BUS del BUFFER
fin while

devuelve POS_BUS

fin FUNCION
```

FIGURA 4.11 • Lógica del programa principal para *leereg*, junto con las funciones *toma_reg()* y *toma_campo()*.

en lugar de pensar en términos del “primer registro”, o el “segundo registro”. (¿Puede recordar cuál registro está primero?) Esta idea de una *llave* basada en el contenido es otra herramienta conceptual fundamental, por ello es necesario desarrollar una idea más precisa de lo que es una llave.

Cuando se busca un registro que contenga el apellido Ames, el usuario quiere reconocerlo, aunque introduzca la llave en forma de “AMES”, “ames”, o “Ames”. Para ello, se debe definir una forma estándar para llaves, paralelamente con las reglas y procedimientos asociados para convertirlas a esta forma. Una forma estándar de este tipo suele llamarse *forma canónica* de la llave. Un significado de la palabra canon es regla; la palabra *canónico* significa “en conformidad con la regla”. Una forma canónica para una llave de búsqueda es la representación única para esa llave que se ajusta a la regla.

A manera de ejemplo, se podría establecer que la forma canónica para una llave requiere que la llave consista sólo en letras mayúsculas y no tenga espacios adicionales al final. De modo que si un usuario introduce “Ames”, la llave se convertiría a la forma canónica “AMES” antes de buscarla.

Una llave no tiene que corresponder a un solo campo en un registro; es posible construir llaves que combinen información de más de un campo del archivo, por ejemplo, cuando se quiere usar el nombre y el apellido de una persona para buscar un registro. Para ello es necesario desarrollar una regla que combine los campos del apellido y del nombre en una llave en forma canónica. Una regla muy sencilla puede ser:

**Ajustar cada campo, concatenar un espacio al final del campo del apellido, después concatenar el campo del nombre.
Convertir la cadena completa a mayúsculas.**

Esta regla convierte el nombre John Ames en la forma canónica “AMES JOHN”. Si se compara esta llave con la del apellido solo, resulta mucho más adecuada para identificar un registro *únivamente*.

A menudo se necesita tener *llaves distintas* o llaves que identifiquen *únivamente* un solo registro. Si no hay una relación de uno a uno entre la llave y un registro, entonces el programa tendría que proporcionar mecanismos adicionales que permitan al usuario resolver la confusión que puede haber cuando existan más registros que concuerden con una llave en particular. Por ejemplo, suponga que se busca la dirección de John Ames. Si hay varios registros en el archivo para varias personas llamadas John Ames, ¿cómo debería responder el programa? Ciertamente, no debería proporcionar sólo la dirección del primer John Ames que encuentre. ¿Proporcionaría todas las direcciones

inmediatamente? ¿Proporcionaría una forma de moverse a través de los registros?

La solución más sencilla es *prevenir* tal confusión. La prevención se lleva a cabo conforme se agregan nuevos registros al archivo. Cuando el usuario introduce un registro nuevo, se forma una llave canónica para ese registro y después se la busca en el archivo. Si ya existe, se indica al usuario que modifique de alguna forma los campos de la llave, de manera que sea única.

Este requisito de unicidad se aplica sólo a las *llaves primarias*. Una llave primaria es, por definición, la llave que se usa para identificar únicamente un registro. También es posible, como se verá posteriormente, buscar con *llaves secundarias*. Un ejemplo de llave secundaria puede ser el campo de ciudad en el archivo de nombres y direcciones. Si se pretende encontrar en el archivo todos los registros de las personas que viven en ciudades llamadas Stillwater, se usaría alguna forma canónica de "Stillwater" como llave secundaria. Por lo regular, las llaves secundarias no identifican un único registro.

4.9

UNA BUSQUEDA SECUENCIAL

Ahora que se tiene acceso al concepto de llave canónica, estamos preparados para escribir un programa llamado *encuentra*, que lea el archivo, registro por registro, buscando uno con una llave en particular. Dicha búsqueda secuencial es sólo una simple ampliación del programa *leereg*, con una operación adicional de comparación en el ciclo principal para ver si la llave del registro corresponde con la llave que se busca. La lógica de la parte principal de *encuentra* se ilustra en pseudocódigo en la figura 4.12. Las funciones *toma_reg()* y *toma_campo()* son las mismas que se usaron en *leereg*. Los detalles de realización se proporcionan en las versiones del programa *encuentra*, escritas en C y en Pascal, al final del capítulo.

4.10

EVALUACION DEL DESEMPEÑO DE LA BUSQUEDA SECUENCIAL

En los capítulos que siguen se encuentran formas de búsqueda de registros más rápidas que el mecanismo de búsqueda secuencial. Se puede usar la búsqueda secuencial como una especie de punto de

PROGRAMA: encuentra

```

abre el archivo de entrada como ARCH_ENT
lee APELLIDO y NOMBRE que se buscará
construye la forma canónica LLAVE_BUS a partir de APELLIDO y NOMBRE

asigna FALSO a ENCONTRO
LONG_REGISTRO := toma_reg(ARCH_ENT, BUFFER)
mientras (no ENCONTRO y LONG_REGISTRO > 0)
    inicia POS_BUS con 0
    POS_BUS := toma_campo(APELLIDO, BUFFER, POS_BUS, LONG_REGISTRO)
    POS_BUS := toma_campo(NOMBRE, BUFFER, POS_BUS, LONG_REGISTRO)
    forma la LLAVE_REGISTRO canónica a partir de APELLIDO y NOMBRE

    si (LLAVE_REGISTRO == LLAVE_BUS)
        asigna VERDADERO a ENCONTRO
    otro
        LONG_REGISTRO := toma_reg(ARCH_ENT, BUFFER)
    fin mientras

    si (ENCONTRO)
        llama a toma_campo() para leer y desplegar los campos del registro

```

FIGURA 4.12 • Lógica del ciclo principal para *encuentra*.

comparación para medir las mejoras que se hagan. Por lo tanto, es importante encontrar alguna forma de expresar el tiempo y el trabajo invertidos en una búsqueda secuencial.

Para desarrollar una medida de desempeño se requiere elegir una unidad de trabajo que represente de manera útil las restricciones del desempeño del proceso total. Cuando se describe la eficiencia de las búsquedas que se efectúan en memoria RAM electrónica, donde las operaciones de comparación son más costosas que las operaciones de extracción de datos de la memoria, por lo regular se emplea el *número de comparaciones* requeridas para la búsqueda como medida del trabajo. Sin embargo, dado que el costo de una comparación en RAM es muy pequeño en relación con el costo de un acceso a disco, las comparaciones no representan claramente las restricciones del desempeño de una búsqueda en un archivo en almacenamiento secundario. En lugar de esto, se cuentan las llamadas a la función READ() de bajo nivel. Se supone que cada llamada a READ() requiere un desplazamiento del brazo del disco, y que todas las llamadas a READ() tiene el mismo costo. Gracias al análisis de temas como el manejo de buffers del sistema, en el capítulo 3, se sabe que esas suposiciones no son precisas; sin embargo, en un ambiente multiusuario, donde muchos procesos requieren el disco a un mismo tiempo, resultan ser casi correctas y, por tanto, de utilidad.

Supongamos que en un archivo con 1000 registros se pretende usar una búsqueda secuencial para encontrar el registro de Al Smith. ¿Cuántas llamadas a READ() se requieren? Si el registro de Al Smith es el primero del archivo, el programa tiene que leer sólo un registro; pero si es el último, el programa hace 1000 llamadas a READ() antes de concluir la búsqueda. Para realizar una búsqueda se necesitan, en promedio, 500 llamadas.

Si se duplica el número de registros del archivo, también se duplican el número promedio y el número máximo de llamadas a READ() requeridas. Usar una búsqueda secuencial para encontrar el registro de Al Smith en un archivo de 2000 registros requiere, en promedio, 1000 llamadas. En otras palabras, la cantidad de trabajo requerido por una búsqueda secuencial es directamente proporcional al número de registros del archivo.

En general, el trabajo requerido para buscar en forma secuencial un registro en un archivo con n registros es proporcional a n ; implica, a lo sumo, n comparaciones y, en promedio, alrededor de $n/2$ comparaciones. Se dice que una búsqueda secuencial es de orden $O(n)$ porque el tiempo que tarda es proporcional a n .[†]

Gran parte del resto de este libro está dedicado a identificar mejores caminos para tener acceso a los registros individuales; la búsqueda secuencial es demasiado costosa para la mayoría de las situaciones de extracción de información. Hay, sin embargo, algunas aplicaciones en las que la búsqueda secuencial *puede ser razonable*, tal como en la búsqueda en archivos con muy pocos registros (p. ej., 10 registros), la búsqueda en archivos en los que casi nunca se necesita buscar (p. ej., los archivos en cintas que normalmente se usan para otra clase de procesamiento) y la búsqueda en archivos de registros con cierto valor de llave secundaria, donde se espera que haya un gran número de correspondencias.

4.11

MEJORA DEL DESEMPEÑO DE LA BUSQUEDA SECUENCIAL: MANEJO DE REGISTROS EN BLOQUES

Es interesante y útil aplicar la información que proporciona el capítulo 3 sobre desempeño del disco cuando se aborda el problema de mejorar la eficiencia de la búsqueda secuencial. En dicho capítulo se vio que el principal costo asociado con un acceso al disco es el tiempo requerido

[†]Si no está familiarizado con esta notación, el lector debe informarse al respecto. Knuth [1973a] es una buena fuente.

para efectuar un desplazamiento al lugar adecuado del disco. Una vez que comienza, la transferencia de los datos es bastante rápida, aunque aún mucho más lenta que una transferencia de datos dentro de memoria RAM. Por tanto, el costo que implica el desplazamiento y la lectura de un registro y después el desplazamiento y lectura de otro registro es mucho mayor que el costo de un solo desplazamiento y luego la lectura de dos registros sucesivos. (De nuevo, se supone un ambiente multiusuario donde se requiere un desplazamiento por cada llamada a READ().) Se concluye que existe la posibilidad de mejorar la eficiencia de la búsqueda secuencial leyendo un *bloque* de varios registros a la vez y después procesar ese bloque de registros en memoria RAM.

Se inició este capítulo con una secuencia de bytes. Después se agruparon los bytes en campos y, por último, se agruparon los campos en registros. Ahora se está considerando un nivel de organización todavía superior, el agrupamiento de registros en bloques. Sin embargo, este nuevo nivel de agrupamiento se diferencia de los otros. Mientras que los campos y registros son formas de mantener la organización lógica dentro del archivo, el manejo de bloques es únicamente una medida para mejorar el desempeño. Como tal, el tamaño del bloque tiene más relación con las propiedades físicas de la unidad de disco que con el contenido de los datos. Por ejemplo, en discos orientados por sectores, el tamaño del bloque casi siempre es algún múltiplo del tamaño del sector.

Supongamos que se tiene un archivo con 1000 registros, y que la longitud media de un registro es de 64 bytes. Una búsqueda secuencial sin manejo de bloques requiere, en promedio, 500 llamadas a READ() antes de poder extraer un registro en particular. Si los registros se manejan en bloques y cada bloque contiene grupos de 32, de modo que cada llamada a READ() proporcione 2 kilobytes de registros, el número de READ requeridos para una búsqueda media decrece a 15. Cada READ() requiere un poco más de tiempo, ya que se transfieren más datos desde el disco, pero por lo regular es un gasto que vale la pena hacer a cambio de la drástica reducción en el número de lecturas.

Nótese que, aunque el manejo de registros en bloques puede brindar mejoras importantes en el desempeño, no cambia el orden de la operación de búsqueda secuencial. El costo de la búsqueda aún es de $O(n)$, y se incrementa en proporción directa al incremento del tamaño del archivo. Obsérvese también que esta nueva herramienta refleja con claridad las diferencias entre la velocidad de acceso a memoria RAM y el costo del acceso al almacenamiento secundario. El manejo de registros en bloques no cambia el número de comparaciones que deben hacerse en memoria RAM, y es muy probable que incremente la cantidad de datos transferidos entre el disco y la memoria RAM. (Siempre se lee un bloque completo, aun cuando el registro que se busca sea el primero)

del bloque.) El manejo de bloques ahorra tiempo, ya que disminuye el número de desplazamientos del brazo del disco. Se encuentra, de nuevo, que esta diferencia entre el costo del desplazamiento y el costo de otras operaciones, como las transferencias o el acceso a memoria RAM, es la fuerza que impulsa el diseño de estructuras de archivos.

4.12

ACCESO DIRECTO

La alternativa más radical para la búsqueda secuencial de un registro en un archivo es un mecanismo de extracción de información conocido como *acceso directo*. Se tiene acceso directo a un registro cuando es posible colocarse directamente en el inicio del registro y leerlo. Mientras que la búsqueda secuencial es una operación $O(n)$, el acceso directo es $O(1)$; no importa cuán grande sea el archivo, con un solo desplazamiento es posible extraer el registro que se quiera.

El acceso directo implica saber dónde está el comienzo del registro requerido. Algunas veces esta información acerca de la posición del registro se guarda en un archivo separado de índices. Sin embargo, por el momento se supone que no existe un índice, y que, en lugar de esto, se sabe el *número relativo de registro* (NRR) del que se desea. La idea de un NRR es un concepto importante que surge de la noción de un archivo como un conjunto de registros, y no como un conjunto de bytes. Si un archivo es una secuencia de registros, entonces el NRR de un registro proporciona su posición relativa con respecto al principio del archivo. El primer registro de un archivo tiene un NRR 0, el siguiente tiene un NRR 1, y así sucesivamente. †

En el archivo de nombres y direcciones es posible unir un registro a su NRR asignándole números de membresía relacionados con el orden en que se introducen los registros al archivo. La persona con el primer registro puede tener el número de membresía 1001, la segunda, el número 1002, y así sucesivamente. Dado un número de membresía, puede restarse 1001 para tener el NRR del registro.

¿Qué se puede hacer con este NRR? No mucho, ya que las estructuras de archivos que se han usado hasta ahora consisten en registros de longitud variable. El NRR indica la posición relativa del registro que se quiere en la secuencia de registros, pero aún se debe leer secuencialmente a lo largo del archivo, contando los registros que se llevan, para tener

† Respetando las convenciones de C y Turbo Pascal, se supone que el NRR es un conteo con *base cero*. En algunos sistemas de archivos, el conteo empieza en 1, en lugar de 0.

el registro que se quiere. El ejercicio 20, al final de este capítulo, explora un método de movimiento a lo largo de un archivo llamado procesamiento secuencial con saltos, que puede mejorar un poco el desempeño, pero la búsqueda de un NRR en particular aún es un proceso $O(n)$.

Para que sea posible el acceso directo por NRR se necesita trabajar con registros de longitud fija y conocida. Si todos los registros son de la misma longitud, entonces se puede usar el NRR de un registro para calcular la *distancia en bytes* del inicio del registro en relación con el inicio del archivo. Por ejemplo, si se está interesado en el registro con NRR 546 y el archivo tiene registros de longitud fija de 128 bytes, se puede calcular la distancia en bytes de la siguiente forma:

$$\text{Distancia en bytes} = 546 \times 128 = 69\,888.$$

En general, dado un archivo de registros de longitud fija de tamaño r , la distancia en bytes de un registro con un NRR de n es

$$\text{Distancia en bytes} = n \times r.$$

Los lenguajes de programación y los sistemas operativos difieren con respecto a dónde se hace este cálculo de la distancia en bytes, e incluso con respecto al uso de las distancias en bytes para hacer referencia a direcciones dentro de los archivos. En C (y los sistemas operativos UNIX y MS-DOS), donde un archivo se considera como una mera secuencia de bytes, el programa de aplicación hace el cálculo y usa la orden *lseek()* para saltar al byte que inicia el registro. Todos los movimientos dentro de un archivo están en términos de bytes. Éste es un punto de vista desde un nivel muy bajo; la responsabilidad de la traducción de un NRR a una distancia en bytes pertenece por completo al programa de aplicación.

El lenguaje PL/I y los ambientes operativos en los que se usa con frecuencia (OS/MVS, VMS) son ejemplos de un punto de vista diferente y de nivel superior sobre los archivos. La idea de una secuencia de bytes sencillamente no existe cuando se trabaja con archivos orientados a registros en este ambiente. Los archivos se conciben, en cambio, como un conjunto de registros a los cuales se tiene acceso mediante llaves. El sistema operativo se hace cargo de la traducción entre una llave y la posición de un registro. En el caso más sencillo, la llave es, de hecho, sólo el NRR del registro, pero la determinación de la posición real dentro del archivo no es asunto del programador.

Si uno se restringe a usar Pascal estándar, la cuestión de la localización por bytes o la localización por registros no es ningún problema: no hay localización alguna en Pascal estándar. Pero, como se mencionó anteriormente, muchas realizaciones de Pascal amplían la definición

estándar del lenguaje para permitir el acceso directo a diferentes localidades de un archivo. La naturaleza de esas ampliaciones varía según las diferencias de los sistemas operativos alrededor de los cuales se desarrollan. Por ello, una característica persistente en las diferentes realizaciones es que un archivo en Pascal siempre comprende elementos de un solo tipo. Un archivo es una secuencia de enteros, o caracteres, o arreglos, o registros, y así sucesivamente. La referencia a direcciones siempre está en términos del tamaño del elemento fundamental. Por ejemplo, se puede tener un *archivo de regdato*, donde *regdato* se define como:

```
TYPE regdato = packed array [0..64] of char;
```

La localización dentro de este archivo está en términos de múltiplos de la unidad elemental *regdato*, esto es, en términos de múltiplos de una entidad de 65 bytes. Si se pide saltar al *regdato* número 3 (conteo con base cero), esto implica saltar 195 bytes ($3 \times 65 = 195$) dentro del archivo.

4.13

ELECCION DE UNA ESTRUCTURA Y UNA LONGITUD DE REGISTRO

Una vez que se decide fijar la longitud de los registros para poder usar el NRR y tener acceso directo a un registro, se debe determinar su longitud. Está claro que esta decisión está relacionada con el tamaño de los campos que se desea guardar en el registro. Algunas veces la decisión es sencilla. Supongamos que se está construyendo un archivo de operaciones de venta que contiene la siguiente información acerca de cada operación:

- Un número de cuenta de 6 dígitos para el comprador;
- Seis dígitos para el campo de fecha;
- Un número de inventario de 5 caracteres para el artículo comprado;
- Un campo de 3 dígitos para la cantidad, y
- Un campo de 10 posiciones para el costo total.

Todos esos campos tienen una longitud fija; la suma de sus longitudes es de 30 bytes. Si se intenta almacenar los registros en un disco normal dividido en sectores (véase el cap. 3) de 512 bytes de tamaño, o de alguna otra potencia de 2, puede convenir redondear el registro a 32 bytes para poder colocar un número entero de registros en un sector.

La elección de la longitud de registro es más complicada cuando la longitud de los campos puede variar, como en el archivo de nombres y direcciones. Si se elige una longitud de registro equivalente a la suma de los valores que, según las estimaciones, sea lo más grande posible para todos los campos, es seguro que habrá suficiente espacio para todo, pero también bastante espacio desperdiciado. Por otra parte, si se es moderado en el uso de espacio y se fijan las longitudes de los campos conforme a los valores más pequeños, tal vez habría que dejar información fuera de un campo. Por fortuna, se puede evitar hasta cierto grado este problema con un diseño apropiado de la estructura de campos *dentro* de un registro.

En el análisis de las estructuras de la sección 4.4 se presentan dos enfoques generales que se pueden considerar aquí, referidos a la organización de campos dentro de un registro de longitud fija. El primero, ilustrado en la figura 4.13(a), emplea campos de longitud fija dentro del registro de longitud fija. Este enfoque se tomó en cuenta para el archivo de operaciones de ventas descrito con anterioridad. El segundo enfoque, ilustrado en la figura 4.13(b), usa el registro de longitud fija como una especie de recipiente de tamaño estándar, para guardar algo que se parece a un registro de longitud variable. El primer enfoque tiene la virtud de la sencillez: es muy fácil "separar" los campos de longitud fija dentro de un registro de longitud fija. El segundo enfoque permite aprovechar un efecto normalizador que suele ocurrir: lo más probable es que los nombres más largos no aparezcan en el mismo registro que

Ames	John	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

(a)

Ames John 123 Maple Stillwater OK 74075	← Espacio sin usar →
Mason Alan 90 Eastgate Ada OK 74820	← Espacio sin usar →

(b)

FIGURA 4.13• Dos enfoques fundamentales para la estructura de campos dentro de un registro de longitud fija. (a) Registros de longitud fija con campos de longitud fija. (b) Registros de longitud fija con campos de longitud variable.

el campo de dirección más largo. Al permitir que los límites de los campos varíen se puede hacer un uso más eficiente de la cantidad fija de espacio. También obsérvese que los dos enfoques no son excluyentes entre sí. Si tenemos un registro que contenga un número de campos de longitud fija verdadera junto con algunos campos que tengan información de longitud variable, es posible diseñar una estructura de registro que combine estos dos enfoques.

Los programas *actualiza.c* y *actualiza.pas*, que se incluyen en el conjunto de programas que aparecen al final del capítulo, emplean el acceso directo para permitir al usuario extraer un registro, cambiarlo, y después escribirlo de nuevo; dichos programas crean una estructura de archivo que usa campos de longitud variable dentro de registros de longitud fija. Ésta es una elección apropiada, considerando la variabilidad de longitud de los campos en el archivo de nombres y direcciones.

Una de las cuestiones más interesantes que debe resolver el diseño de esta clase de estructuras es distinguir la porción de datos reales del registro de la porción de espacio no usado. La gama de soluciones posibles es paralela a la de las soluciones para el reconocimiento de registros de longitud variable en cualquier otro contexto: se puede colocar un contador de longitud del registro al inicio, se puede usar un delimitador especial al final del registro, se puede contar los campos, y así sucesivamente. Como ambas, *actualiza.c* y *actualiza.pas*, usan un buffer de cadenas de caracteres para recolectar los campos, y como el manejo de cadenas es diferente en C y en Pascal (en C las cadenas terminan con un carácter nulo; en Pascal se mantiene el número de bytes de la longitud de la cadena al inicio de éstas), es conveniente usar una estructura de archivo ligeramente diferente para las dos realizaciones. En la versión en C se ocupa la porción no usada del registro con caracteres nulos; en la versión en Pascal se coloca un campo de longitud fija (un entero) al inicio del registro, para indicar cuántos bytes del registro son válidos. Como es habitual, no existe una forma correcta única de realizar esta estructura de archivo; en todos los casos, se debe buscar la solución más apropiada, según las necesidades y la situación particulares.

La figura 4.14 muestra el vaciado hexadecimal de cada uno de estos programas, en donde se introducen varias ideas más, tales como el uso de *registros de encabezado*, que se analizan en la siguiente sección. Sin embargo, por ahora sólo se examina la estructura de los registros de datos. En la salida del programa en Pascal, los campos de longitud al inicio de los registros aparecen en cursivas. Aunque se llenaron los registros creados por el programa en Pascal con espacios, para hacer la salida más legible, esto es innecesario. El campo de longitud al principio del registro garantiza que no se leerá más allá del final de los datos que contiene.

0 - F	02000000	00000000	00000000	00000000	Registro de encabezado
10 - 1F	00000000	00000000	00000000	00000000	Contador de registro en los primeros 2 bytes
20 - 2F	416D6573	7C4A6F68	637C3132	33204D61	Ames ; John ; 123 ; Ma	Primer registro
30 - 3F	706C657C	5374696C	6C776174	65727C4F	ple ; Stillwater ; 0	
40 - 4F	4B7C3734	3037357C	00000000	00000000	K ; 75075 ;	
50 - 5F	00000000	00000000	00000000	00000000	

60 - 6F	4D61736F	6E7C416C	616E7C39	30204561	Mason ; Alan ; 90 Ea	Segundo registro
70 - 7F	73746761	74657C41	64617C4F	4B7C3734	stgate ; Ada ; OK ; 74	
80 - 8F	3832307C	00000000	00000000	00000000	820 ;	
90 - 9F	00000000	00000000	00000000	00000000	

(a)

0 - F	02000000	00000000	00000000	00000000	Registro de encabezado
10 - 1F	00000000	00000000	00000000	00000000	Contador de registro en los primeros 2 bytes
20 - 2F	00000000	00000000	00000000	00000000	
30 - 3F	00000000	00000000	00000000	00000000	
40 - 41	0000				..	
42 - 4F	2800	416D6573	7C4A6F68	637C3132	(.Ames ; John ; 12	Primer registro
50 - 5F	33204D61	706C657C	5374696C	6C776174	3 Maple ; Stillwat	
60 - 6F	65727C4F	4B7C3734	3037357C	20202020	er;OK ; 74075 ;	
70 - 7F	20202020	20202020	20202020	20202020	20202020	
80 - 83	202020					
84 - 8F		24004D61	736F6E7C	416C616E	\$.Mason ; Alan	Segundo registro
90 - 9F	7C393020	45617374	67617465	7C416461	; 90 Eastgate ; Ada	
A0 - AF	7C4F4B7C	37343832	307C2020	20202020	; OK ; 74820 ;	
B0 - BF	20202020	20202020	20202020	20202020	20202020	
C0 - C5	202020	2020				

(b)

FIGURA 4.14• Dos estructuras de registro distintas que llevan campos de longitud variable en un registro de longitud fija. (a) Estructura de registro creada por *actualiza.c*: registros de longitud fija que contienen campos de longitud variable terminados con un carácter nulo. (b) Estructura de registro creada por *actualiza.pas*: los registros de longitud fija comienzan con un campo (entero) de longitud fija que indica el número de bytes utilizables en los campos de longitud variable del registro.

4.14

REGISTROS DE ENCABEZADO

A menudo es necesario, o útil, mantener información general sobre un archivo que se usará en el futuro. Por ejemplo, en algunas versiones de Pascal no es fácil saltar hasta el final del archivo, aunque la realización permita el acceso directo. Una solución simple a este problema es mantener el contador del número de registros del archivo y almacenarlo en alguna parte. Muchas veces se coloca un *registro de encabezado* al principio del archivo para guardar esta clase de información.

El registro de encabezado tiene una estructura diferente de la que tienen los registros de datos del archivo. Por ejemplo, la salida de *actualiza.c* usa un registro de encabezado de 32 bytes, mientras que cada uno de los registros de datos contiene 64 bytes. Asimismo, los registros de datos creados por *actualiza.c* contienen sólo caracteres como datos, mientras que el registro de encabezado contiene un entero que indica cuántos registros de datos hay en el archivo. La diferencia entre los registros de encabezado y los de datos se acentúa al incluir información como la longitud de registro de los registros de datos, la fecha y la hora de la actualización más reciente del archivo, etcétera.

Al realizar los registros de encabezado en Pascal, el programador suele adecuar las diferencias entre los registros de datos y los de encabezado mediante el uso de un registro *variante*. En *actualiza.pas* se muestra una solución más sencilla aún: se usa el campo entero inicial del registro para un propósito diferente en el registro de encabezado. En los registros de datos este campo guarda el número de bytes de datos válidos dentro del registro; en el registro de encabezado guarda el número de los registros de datos del archivo.

Los registros de encabezado son una herramienta de diseño de archivos muy difundida e importante. Por ejemplo, cuando se llegue al punto en donde se hace el análisis de la construcción de índices para archivos, se observará que los registros de encabezado con frecuencia se colocan al principio del índice para mantener información sobre datos tales como el NRR del registro que es la base del índice.

4.15

ACCESO Y ORGANIZACION DE ARCHIVOS

Durante el curso de este capítulo se examinaron:

- Los registros de longitud variable;
- Los registros de longitud fija;

- El acceso secuencial, y
- El acceso directo.

Los primeros dos términos se relacionan con aspectos de *organización de archivos*. El segundo par de términos se refiere al *acceso al archivo*. La distinción entre la organización y el acceso a los archivos es de utilidad, por lo que resulta necesario examinarla con más detalle antes de concluir el capítulo.

La mayor parte de lo considerado hasta aquí entra en la categoría de organización de archivos. ¿Puede un archivo estar dividido en campos? ¿Hay un nivel más alto de organización para el archivo, que combine los campos en registros? ¿Tienen todos los registros el mismo número de bytes o de campos? ¿Cómo se distingue un registro de otro? ¿Cómo se organiza la estructura interna de un registro de longitud fija, de tal forma que se pueda distinguir entre los datos y el espacio adicional? Se ha visto que hay muchas respuestas posibles a esas preguntas y que la elección de una organización de archivos en particular depende de muchos factores; entre ellos, las facilidades que brinde el lenguaje que se utilice y el *uso que se quiera dar al archivo*.

Usar un archivo implica tener acceso a él. Se examinó primero el acceso secuencial, para llegar a desarrollar una *búsqueda secuencial* en los programas *encuentra.c* y *encuentra.pas*. Mientras no se sabía dónde empezaban los registros individuales, el acceso secuencial era la única opción abierta. Cuando se escribió el programa *actualiza*, se pretendió el *acceso directo*, de tal forma que se fijó la longitud de los registros para calcular con precisión dónde principiaba cada registro y poder hacer la localización directamente allí.

En otras palabras, la necesidad del acceso directo motivó la elección de la *organización* del archivo en registros de longitud fija. ¿Significa esto que se pueden equiparar los registros de longitud fija con el acceso directo? Definitivamente no. No hay nada, al fijar la longitud de los registros de un archivo, que excluya el acceso secuencial; sin duda se podría escribir un programa que lea secuencialmente un archivo con registros de longitud fija.

No sólo se puede leer a lo largo de registros de longitud fija en forma secuencial, sino que también se puede proporcionar el acceso directo a registros de *longitud variable* con sólo mantener una lista de las distancias en bytes desde el inicio del archivo hasta la localidad de cada registro. Se eligió la estructura de registro que se usa en *actualiza.c* y *actualiza.pas* porque es sencilla y adecuada para los datos que se quieren almacenar. Aunque varían las longitudes de los nombres y direcciones, la variación no es tan grande como para que no se pueda acomodar en un registro de longitud fija. Sin embargo, considérense los efectos del uso de una organización de registros de longitud fija para

proporcionar acceso directo a registros que sean documentos con un margen de longitud que vaya desde unas pocas centenas de bytes hasta más de cien kilobytes. Los registros de longitud fija representarían un desperdicio de espacio escandaloso, de modo que habría que encontrar alguna forma de estructurar registros de longitud variable. A fin de desarrollar estructuras de archivos adecuadas para tales situaciones se necesita distinguir con claridad entre el *acceso* y las opciones de *organización*.

Las restricciones impuestas por el lenguaje y el sistema de archivos empleados para desarrollar aplicaciones establecen, ciertamente, límites a la capacidad del diseñador para aprovechar esta distinción entre el método de acceso y la organización. Por ejemplo, el lenguaje C proporciona al programador la posibilidad de realizar el acceso directo a registros de longitud variable, puesto que permite el acceso a cualquier byte del archivo. Por otro lado, Pascal, aun cuando disponga de localización, impone limitaciones relacionadas con la definición que tiene de un archivo como un conjunto de elementos del mismo *tipo* y, por consiguiente, del mismo tamaño. La localización se efectúa, por lo regular, sólo al inicio de un elemento. Puesto que en Pascal todos los elementos deben ser del mismo tamaño, el acceso directo a registros de longitud variable es, en el mejor de los casos, difícil.

Cada uno de los dos modos de acceso básicos, secuencial y directo, tiene usos importantes. Está claro que el acceso directo es preferible cuando sólo se necesitan unos pocos registros específicos; pero el acceso directo no siempre es el método más apropiado para la extracción de información. Por ejemplo, si se están preparando cheques de pago usando un archivo de registros de empleados, normalmente no será necesario para hacer el cálculo de las posiciones de los registros. Puesto que todos los registros del archivo deben procesarse, es más rápido y sencillo empezar por el principio y trabajar a lo largo del archivo, registro por registro, hasta llegar al final.

RESUMEN

El nivel más bajo de organización que por lo regular se impone a un archivo es el de una *secuencia de bytes*. Por desgracia, al almacenar los datos en un archivo sólo como una secuencia de bytes, se pierde la capacidad de distinguir entre las unidades fundamentales de información de los datos. A estas piezas fundamentales de información se les llama *campos*. Los campos se agrupan para formar *registros*. Para reconocer

los campos y los registros se debe imponer una estructura a los datos del archivo.

Existen varias formas de separar un campo de otro y un registro de otro:

1. Fijar la longitud de cada campo o registro.
2. Empezar cada campo o registro con el número de bytes que contenga.
3. Usar delimitadores para marcar las divisiones entre entidades.

Otra técnica útil, en el caso de los registros, es usar un segundo archivo de índices que informe dónde empieza cada registro.

A menudo también se impone a los archivos una organización de nivel superior, donde los registros se agrupan en *bloques*. Este nivel se impone para mejorar la eficiencia de la E/S en lugar de sólo mejorar el punto de vista lógico del archivo.

En este capítulo se usa la estructura de registro que emplea un indicador de longitud al inicio de cada uno para desarrollar los programas que escriben y leen un archivo de registros de longitud variable con nombres y direcciones de personas. Se usa el manejo de buffers para acumular el dato en un registro individual antes de conocer su longitud para poderlo transcribir al archivo. Los buffers también son útiles para realizar la lectura de un registro completo a la vez. Se representa el campo de longitud de cada registro como un número binario o como una secuencia de dígitos ASCII. En el primer caso es útil usar un *vaciado hexadecimal* para examinar el contenido del archivo.

Algunas veces se identifican los registros individuales por su *número relativo de registro* (NRR) en un archivo. Sin embargo, a menudo se necesita también identificar un registro mediante una *llave*, cuyo valor está basado en algo del contenido del registro. Los valores llave deben presentarse en alguna *forma canónica* predeterminada, o convertirse en ésta, para que los programas las reconozcan en forma precisa y sin ambigüedades. Si cada valor llave de registro es distinto de los demás, la llave puede emplearse para identificar y ubicar un registro único en el archivo. A las llaves que se usan en esta forma se les llama *llaves primarias*.

En este capítulo se examina un programa llamado *encuentra*, que busca secuencialmente a lo largo de un archivo un registro con una llave en particular. La búsqueda secuencial puede tener un desempeño deficiente en archivos grandes, pero hay ocasiones en que la búsqueda secuencial es adecuada. El manejo de registros en bloques puede usarse para mejorar considerablemente el tiempo de E/S en una búsqueda secuencial.

En el análisis anterior sobre las formas de separar registros se vio con claridad que algunos de los métodos brindan un mecanismo para averiguar o calcular la *distancia en bytes* desde el inicio de un registro. Esto, a su vez, abre la posibilidad de acceder a los registros *directamente*, por NRR, en lugar de hacerlo en forma secuencial.

El formato más simple de registro que permite el acceso directo por NRR implica el uso de registros de longitud fija. Cuando los datos por sí mismos tienen tamaño fijo (p. ej., códigos postales), los registros de longitud fija pueden tener un buen desempeño y utilizar bien el espacio. Sin embargo, cuando la cantidad y tamaño de los datos en los registros es muy variable, el uso de registros de longitud fija puede ocasionar un costoso desperdicio de espacio. En tales casos, el diseñador debe examinar con cuidado la posibilidad de usar registros de longitud variable.

Algunas veces es útil mantener información general acerca de los archivos, como el número de registros que contienen. Un *registro de encabezado*, almacenado al inicio del archivo al que pertenece, es una herramienta útil para almacenar este tipo de información.

Es importante ser cuidadoso con las diferencias entre el *acceso al archivo* y la *organización del archivo*. Se intenta organizar los archivos de manera que permitan los tipos de acceso necesarios para una aplicación en particular. Por ejemplo, una de las ventajas de la *organización* de registros de longitud fija es que permite tanto el *acceso secuencial* como el *directo*.

TERMINOS CLAVE

Acceso directo. Modo de acceder a un archivo que implica saltar al lugar preciso de un registro. El acceso directo a un registro de longitud fija por lo regular se lleva a cabo mediante su *número relativo de registro* (NRR), después calculando su distancia en bytes y, por último, colocándose en el primer byte del registro.

Acceso secuencial. El acceso secuencial a un archivo significa leer el archivo desde el principio y continuar hasta que se haya leído todo lo que se necesita. La alternativa es el acceso directo.

Bloque. Conjunto de registros almacenados como una unidad físicamente contigua en el almacenamiento secundario. En este capítulo se manejaron los registros por bloques para mejorar la eficiencia de la E/S durante la búsqueda secuencial.

Búsqueda secuencial. Método de búsqueda en un archivo, que implica leer el archivo desde el principio y continuar haciéndolo hasta que se haya encontrado el registro deseado.

Campo. La más pequeña unidad lógicamente significativa de un archivo. Un registro de un archivo por lo regular está compuesto de varios campos

Campo de conteo de bytes. Campo al inicio de un registro de longitud variable que indica el número de bytes usados para almacenar el registro. El uso de un campo de conteo de bytes permite al programa transmitir (o saltar) un registro de longitud variable sin necesidad de alterar la estructura interna del registro.

Delimitador. Uno o más caracteres usados para separar campos y registros en un archivo.

Forma canónica. Forma estándar para una llave que puede derivarse, con la aplicación de reglas bien definidas, a partir de la forma no estándar particular de los datos encontrados en un campo de llave del registro, o que puede ser proporcionada en una solicitud de búsqueda por parte del usuario.

Llave. Expresión derivada de uno o más campos dentro de un registro, que puede usarse para ubicar ese registro. A los campos usados para construir una llave se les denomina *campos de llave*. El acceso por llave proporciona una forma de recuperar información que se basa en el contenido de los registros, y no en su posición.

Llave primaria. Llave que identifica únicamente cada registro y que se usa como el método primario de acceso a los registros.

Método de acceso a un archivo. El enfoque proporcionado para localizar información en un archivo. En general, las dos alternativas que hay son *acceso secuencial* y *acceso directo*.

Métodos de organización de archivos. La combinación de estructuras conceptuales y físicas empleadas para distinguir un registro de otro y un campo de otro. Ejemplos de un tipo de organización de archivos son los registros de longitud fija, que contienen varios campos delimitados de longitud variable.

Número relativo de registro (NRR). Índice que da la posición de un registro en relación con el inicio de su archivo. Si un archivo tiene registros de longitud fija, el NRR puede usarse para calcular la *distancia en bytes* a un registro de tal forma que se pueda acceder directamente a él.

Registro. Conjunto de campos relacionados. Por ejemplo, el nombre, la dirección, y demás datos de una persona en un archivo de lista de correos, con probabilidad conformaría un registro.

Registro de encabezado. Registro colocado al inicio de un archivo, que se usa para guardar información acerca de su contenido y organización.

Registros de longitud fija. Organización de archivos en la que todos los registros tienen la misma longitud. Los registros se completan con espacios, nulos, u otros caracteres, de tal forma que se extiendan a la longitud fijada. Puesto que todos los registros tienen la misma longitud, es posible calcular la posición de inicio de cada registro, haciendo posible el *acceso directo*.

Registros de longitud variable. Organización de archivos en donde los registros no tienen una longitud predeterminada; son tan largos como sea necesario y, por tanto, permiten un mejor uso del espacio que los registros de longitud fija.

Desafortunadamente, no es posible calcular la distancia en bytes a un registro de longitud variable únicamente con su número relativo de registro.

Secuencia de bytes. Término que describe el punto de vista de más bajo nivel de un archivo. Si se empieza con el punto de vista básico de un archivo, como *secuencia de bytes*, entonces se puede imponer niveles superiores de orden en el archivo, como estructuras de campos, registros y bloques.

EJERCICIOS

1. Encuentre situaciones donde sean apropiadas cada una de las tres estructuras de campos descritas en el texto. Haga lo mismo para cada una de las estructuras de registro descritas.
2. Analice cuán apropiado es el uso de los siguientes caracteres para delimitar campos en registros: retorno de carro, salto de línea, espacio, coma, punto, dos puntos y escape. ¿Puede imaginar situaciones en las que se plantee usar delimitadores diferentes para campos diferentes?
3. Suponga que se quiere cambiar los programas para incluir un campo de número telefónico en cada registro. ¿Qué cambios necesitan hacerse?
4. Suponga que se necesita guardar un archivo en el cual cada registro tiene campos de longitud fija y campos de longitud variable. Por ejemplo, se quiere crear un archivo de registros de empleados, usando campos de longitud fija para cada identificador de empleado (llave primaria), sexo,

fecha de nacimiento y departamento, y campos de longitud variable para cada nombre y dirección. ¿Qué ventajas pueden justificar el uso de dicha estructura? ¿Pondría la parte de longitud variable al principio o al final? Cualquiera de los dos enfoques es posible; ¿cómo puede realizarse cada uno?

5. Una estructura de registro no descrita en este capítulo se llama *rotulada*. En una estructura de registros rotulados, cada campo que se representa es precedido por un rótulo que describe su contenido. Por ejemplo, si se usan los rótulos AP, NO, DI, CD, ED y CP para describir los seis campos de longitud fija para un registro de nombre y dirección, éste podría aparecer así:

APAmesbbbbbbNOJohnbbbbbbDI123 MaplebbbbbbCDStillwaterEDOKCP74075bbbb

¿En qué condiciones puede ser razonable, e incluso deseable, esta estructura de registro?

6. Proporcione los significados de los términos *secuencia de bytes*, *secuencia de campos* y *secuencia de registros*.

7. Encuentre las estructuras de archivos básicas disponibles en el lenguaje de programación que actualmente usa. Por ejemplo, ¿reconoce su lenguaje un tipo de estructura de secuencia de bytes? ¿Reconoce líneas de texto? ¿Maneja bloques de registros? Explique cómo puede desarrollar los tipos de estructuras que su lenguaje no reconoce, usando estructuras que sí reconozca.

8. Indique cuáles son las estructuras de campos y registros básicas disponibles en PL/I o en COBOL.

9. Compare el uso de caracteres ASCII para representar *todo* en un archivo contra el uso de datos binarios y ASCII mezclados.

10. Si se lista el contenido de un archivo con caracteres tanto ASCII como binarios en la pantalla de la terminal, ¿qué resultados se esperarían? ¿Qué sucede cuando se lista un archivo completamente binario en la pantalla? (Cuidado. Si realmente intenta esto, hágalo con un archivo muy pequeño; podría bloquear o reconfigurar la terminal, o incluso salirse del sistema).

11. Si una llave en un registro está ya en forma canónica y es su primer campo, es posible buscar un registro por llave sin separar el campo llave del resto de los campos. Explique.

12. Se ha planteado (Sweet, 1985) que las llaves primarias deben ser "sin datos, invariables, precisas y únicas." Analice la importancia de cada criterio y muestre, por ejemplo, cómo es que su ausencia podría provocar problemas. ¿La llave primaria usada en el archivo del ejemplo viola alguno de los criterios?
13. ¿Cuántas comparaciones se requerirían en promedio para encontrar un registro en un archivo con 10 000 almacenado en disco, usando la búsqueda secuencial? Si el registro no está en el archivo, ¿cuántas comparaciones se requieren? Si el archivo se maneja por bloques, de tal forma que se guardan 20 registros por bloque, ¿cuántos accesos se requieren en promedio? ¿Cuántos, si sólo se guarda un registro por bloque?
14. En la evaluación del desempeño de la búsqueda secuencial, se supone que toda lectura implica un desplazamiento del brazo del disco. ¿Cómo cambian las suposiciones en una máquina con un solo usuario? ¿Cómo afectan el análisis de la búsqueda estas nuevas suposiciones?
15. Proporcione una fórmula para encontrar el desplazamiento en bytes de un registro de longitud fija donde el NRR del primer registro es 1 en lugar de 0.
16. ¿Por qué no funciona la estructura de registros de longitud variable en el programa *actualiza*? ¿Ayudaría tener un índice que apunte al inicio de cada registro de longitud variable?
17. El programa *actualiza* permite al usuario modificar registros, pero no eliminarlos. ¿Cómo se deben modificar la estructura de archivo y los procedimientos de acceso para que permitan la eliminación, si no tiene importancia la reutilización del espacio de los registros eliminados? ¿Cómo cambian las estructuras de archivos y los procedimientos si se quiere reutilizar el espacio?
18. En el análisis de los usos de los números relativos de registro (NRR), se dice que puede crearse un archivo en el que exista una correspondencia directa entre una llave primaria, tal como un número de membresía, y el NRR, de tal forma que pueda encontrarse el registro de una persona conociendo sólo el nombre o el número de membresía. ¿Qué clase de problemas pueden encontrarse con esta correspondencia simple entre el número de membresía y el NRR? ¿Qué sucede si se quiere eliminar un nombre? ¿Qué sucede si se cambia la información que hay en un registro de un archivo con registros de longitud variable y la nueva longitud es mayor?

19. El siguiente vaciado hexadecimal describe los primeros bytes de un archivo del tipo producido por la versión C de *escribereg*, pero la columna de la derecha está vacía. ¿De qué tamaño es el primer registro? ¿Cuál es su contenido?

```
0 - F 26004475 6D707C46 7265647C 38323120 .....
10 - 1F 4B6C7567 657C4861 636B6572 7C50417C .....
20 - 2F 36353533 357C2E2E 48657861 64656369 .....
```

20. Suponga que se tiene un archivo de registros de longitud variable con registros largos (de más de 1000 bytes cada uno, en promedio). Suponga que se está buscando un registro con un NRR en particular. Describa los beneficios de usar el contenido de un campo de conteo de bytes para saltar secuencialmente, de registro en registro, hasta encontrar el que se desea. A esto se le llama procesamiento en *saltos secuenciales*. Emplee los conocimientos adquiridos sobre manejo de buffers del sistema para describir por qué esto sólo es útil para registros largos. Si los registros se clasifican según un orden por llave, y se manejan por bloques, ¿qué información se tiene que colocar al inicio de cada bloque para lograr un procesamiento en saltos secuenciales aún más rápido?

21. Suponga que se tiene un registro de longitud fija con campos de longitud fija, y que la suma de las longitudes de los campos es de 30 bytes. Un registro con una longitud de 30 bytes podría guardarlos todos. Si se intenta almacenar los registros en un disco manejado por sectores de 512 bytes (véase Cap. 3), puede decidirse llenar el registro a 32 bytes de tal forma que pueda colocarse un número entero de registros en un sector. ¿Por qué se querría hacer esto?

22. ¿Por qué es importante distinguir entre el acceso a archivos y la organización de archivos?

EJERCICIOS DE PROGRAMACION

23. Reescriba *escribecad*, de tal forma que use delimitadores como separadores de campos. La salida de la nueva versión de *escribecad* deberá ser legible para *leecad.c* o para *leecad.pas*.

24. Desarrolle versiones de *escribereg* y *leereg* que usen las siguientes longitudes fijas de campos en vez de delimitadores.

Apellido:	15 caracteres
Nombre:	15 caracteres
Dirección:	30 caracteres
Ciudad:	20 caracteres
Estado:	2 caracteres
Código postal:	5 caracteres

25. Escriba el programa descrito en el problema anterior de tal forma que use bloques. Almacene cinco registros por bloque.

26. Haga el programa *encuentra*.

27. Reescriba el programa *encuentra* de modo que pueda encontrar un registro por su posición en el archivo. Por ejemplo, si se pide encontrar el registro 547 de un archivo, leería los primeros 546 registros y después imprimiría el contenido del registro 547. Use búsqueda con saltos secuenciales (véase el ejercicio 20) para evitar leer el contenido de los registros que no se desean.

28. Escriba un programa parecido a *encuentra*, pero con las siguientes diferencias: en lugar de recibir las llaves de los registros mediante el teclado, las lee de un archivo de transacciones separado que contiene sólo las llaves de los registros por extraer. En vez de mostrar los registros en la pantalla, los escribe en un archivo de salida separado. Primero, suponga que los registros no están en un orden en particular. Después, suponga que tanto el archivo principal como el de transacciones están clasificados por llave. En el último caso, ¿cómo puede lograrse que el programa sea más eficiente que el programa *encuentra*?

29. Haga cualquiera de las siguientes modificaciones, o todas, a *actualiza.pas* o a *actualiza.c*.

- Permite que el usuario identifique por nombre el registro que será modificado, y no por NRR;
- Permite al usuario modificar campos individuales sin tener que cambiar un registro completo, y
- Proporcione al usuario la opción de revisar el archivo completo.

30. Modifique *actualiza.c* y *actualiza.pas* para señalar al usuario cuando un registro excede su longitud fija. La modificación permitirá al usuario reducir el registro a un tamaño aceptable e introducirlo de nuevo. ¿Qué otras modificaciones harían más sólido el programa?

31. Cambie *actualiza.c* y *actualiza.pas* a un programa de procesamiento por lotes que lea un archivo de transacciones, donde cada registro de transacción contenga el NRR de un registro por actualizar, seguido por su nuevo contenido, y que después haga los cambios en una ejecución por lotes. Aunque no es necesario, es deseable clasificar el archivo de transacciones por NRR. ¿Por qué?

32. Escriba un programa que lea un archivo y escriba su contenido en forma de un vaciado hexadecimal. El vaciado hexadecimal debe tener un formato parecido al que se usó en los ejemplos de este capítulo. El programa debe leer el nombre del archivo de entrada de la línea de órdenes. La salida debe presentarse en la salida estándar (la pantalla de la terminal).

33. Desarrolle un conjunto de reglas para traducir las fechas: agosto 7, 1949; Ag. 7, 1949; 8-7-49; 08-07-49; 8/7/49, y otras variaciones parecidas, a una forma canónica común. Escriba una función que lea una cadena que contenga una fecha en una de esas formas y devuelva la forma canónica de acuerdo con sus propias reglas. Asegúrese de documentar las limitaciones de su función y de sus reglas.

LECTURAS ADICIONALES

Muchos libros de texto cubren el material básico del diseño de estructuras de campos y registros, pero sólo unos pocos abordan las consideraciones y opciones de diseño con mucho detalle. Dos posibles fuentes son Teorey y Fry [1982] y Wiederhold [1983]. El capítulo “*Choice of File Organization*”, de Hanson [1982], es excelente, pero resulta más provechoso después de leer el material de los últimos capítulos de este texto. Se puede aprender mucho sobre los tipos de alternativas de organización y acceso a archivos estudiando las descripciones de las opciones disponibles en ciertos lenguajes y sistemas administradores de archivos. PL/I ofrece un conjunto de alternativas particularmente rico, y Pollack y Sterling [1980] las describen bien y ampliamente.

Sweet [1985] es un artículo corto pero estimulante sobre el diseño de campos de llave. Se describen varios algoritmos interesantes para mejorar la eficiencia de las búsquedas secuenciales en Gonnet [1984] y, por supuesto, en Knuth [1973b].

PROGRAMAS EN C

Los programas en C que se listan en las siguientes páginas corresponden a los programas analizados en el texto. Los programas están contenidos en los siguientes archivos.

· <i>escribesec.c</i>	Escribe la información de nombres y direcciones como una secuencia de bytes consecutivos.
· <i>leesec.c</i>	Lee un archivo de secuencia de bytes como entrada y lo muestra en la pantalla.
· <i>escribereg.c</i>	Escribe un archivo de registros de longitud variable, que usa un contador de bytes al inicio de cada registro para proporcionar su longitud.
· <i>leereg.c</i>	Lee un archivo, registro por registro, mostrando los campos de cada registro en la pantalla.
<i>tomare.c</i>	Contiene funciones de apoyo para la lectura de registros o campos individuales. Estas funciones son necesarias para los programas <i>leereg.c</i> y <i>encuentra.c</i> .
<i>encuentra.c</i>	Busca un registro con una llave en particular en forma secuencial a lo largo de un archivo.
<i>hazllave.c</i>	Combina el nombre y el apellido y los convierte en una llave en forma canónica. Llama a <i>cadblanc()</i> y <i>mayúsculas()</i> , que se encuentran en <i>funscads.c</i> .
<i>funscads.c</i>	Contiene dos funciones de apoyo para cadenas: <i>cadespac()</i> elimina los espacios del final de las cadenas; <i>mayúsculas()</i> convierte los caracteres alfabéticos en mayúsculas.
<i>actualiza.c</i>	Permite agregar registros nuevos a un archivo, o modificar registros existentes.

ARCHES.H

Todos los programas incluyen un archivo de encabezado llamado *arches.h*, el cual contiene definiciones útiles, algunas de ellas dependientes del sistema. Si los programas fueran ejecutados en un sistema UNIX, *arches.h* podría ser así:

```
/*
arches.h — archivo de encabezado que contiene las
definiciones de E/S para archivos
*/
#define PMODE      0755
#define SOLOLECT   0
#define SOLOESCRIT 1
#define LECTESCRIT 2
#define DELIM_CAD  "|"
#define DELIM_CAR  '|'

#define TAM_MAX_REG 512
```

===== ESCRIBESEC.C

```
/* escribesecc.c
```

Crea el archivo de nombres y direcciones que es, en forma estricta, una secuencia de bytes (sin delimitadores, contadores, u otra información que ayude a distinguir campos y registros).

Una modificación simple a la macro `saca_cad`:

```
#define  saca_cad(fd,cad)  write((fd),(cad),strlen(cad)); \
                           write((fd), DELIM_CAD,1);
```

cambia el programa de tal forma que crea campos delimitados.

```
*/
```

```
#include "arches.h"
```

```
#define  saca_cad(fd,cad)  write((fd),(cad),strlen(cad))
```

```
main()
```

```
char nombre[30], apellido[30], dirección[30], ciudad[20];
char estado[15], cp[9];
char nomarch[15];
int fd;
```

```
printf("Proporcione el nombre del archivo que quiera crear: ");
gets(nomarch);
```

```
if ((fd = creat(nomarch,PMODE)) < 0) {
```

```

        printf("Error en la apertura del archivo --- Fin de
        programa\n"); exit(1);
    }
    printf("\n\nDigite un apellido, o <CR> para salir\n>>>");
    gets(apellido);
    while (strlen(apellido) > 0)
    {
        printf("\n Nombre:");
        gets(nombre);
        printf(" Dirección:");
        gets(direccion);
        printf(" Ciudad:");
        gets(ciudad);
        printf(" Estado:");
        gets(estado);
        printf("Cod. Post.:");
        gets(cp);

        /* Traslada las cadenas al buffer y después al archivo */

        saca_cad(fd,apellido);
        saca_cad(fd,nombre);
        saca_cad(fd,dirección);
        saca_cad(fd,ciudad);
        saca_cad(fd,estado);
        saca_cad(fd,cp);

        /* se prepara para los siguientes datos */

        printf("\n\nDigite un apellido, o <CR> para salir\n>>>");
        gets(apellido);
    }

    /* Cierra el archivo antes de terminar */
    close(fd);
}

```

===== LEESEC.C

```

/* leesec.c

   lee una secuencia de campos delimitados
*/
#include "arches.h"

```

```

main() {

    int fd ,n;
    char cad[30];
    char nomarch[15];
    int cont_campos;

    printf("Proporcione el nombre del archivo a leer: ");
    gets(nomarch);
    if ((fd = creat(nomarch, SOLOLECT)) < 0) {
        printf("Error en la apertura del archivo - fin de programa\n");
        exit(1);
    }
    /* Ciclo del programa principal - llama a leecampo() hasta que
       se lean todos los campos */
    cont_campos = 0;
    while ((n = leecampo(fd,cad)) > 0)
        printf("\tCampo # %3d: %s\n",++cont_campos,cad);

    close(fd);
}

leecampo(fd,cad)
    int fd;
    char cad[];
{
    int i;
    char c;

    i = 0;
    while ( read(fd,&c,1) > 0 && c != DELIM_CAR)
        cad[i++] = c;

    cad[i] = '\0'; /* Agrega un carácter nulo al final de la cadena
*/
    return(i);
}

```

===== ESCRIBEREG.C

```

/* escribereg.c

Crea el archivo de nombres y direcciones empleando un campo de
longitud fija adelante de cada registro
*/
#include "arches.h"
#define campo_a_buffreg(br,cad) strcat(br,cad); strcat(br,DELIM_CAD);

```

```
char buffreg[TAM_MAX_REG + 1];
char *solicitud[] = {
    "Digite apellido, o <CR> para salir: ",
    "                Nombre: ",
    "                Dirección: ",
    "                Ciudad: ",
    "                Estado: ",
    "                Cód. Post.: ",
    " "           /* Cadena nula para terminar el ciclo de solicitud */
};

main() {

    char respuesta[50];
    char nomarch[15];
    int fd,i;
    int long_reg;

    printf("Proporcione el nombre del archivo que quiera crear: ");
    gets(nomarch);

    if ((fd = creat(nomarch,PMODE)) < 0) {
        printf("Error en la apertura del archivo - Fin de programa\n");
        exit(1);
    }
    printf("\n\n%s", solicitud[0]);
    gets(respuesta);
    while (strlen(respuesta) > 0)
    {
        buffreg[0] = '\0';
        campo_a_buffreg(buffreg,respuesta);
        for (i=1; *solicitud[i] != '\0' ; i++)
        {
            printf("%s",solicitud[i]);
            gets(respuesta);
            campo_a_buffreg(buffreg,respuesta);
        }

        /* Escribe la longitud del registro y el contenido del buffer */
        long_reg = strlen(buffreg);
        write(fd,&long_reg,2);
        write(fd,buffreg,long_reg);

        /* Se prepara para los siguientes datos */
    }
}
```

```

printf("\n\n%s", solicitud[0]);
gets(respuesta);
}

/* Cierra el archivo antes de terminar */
close(fd);
}

/* Preguntas:

¿Cómo funciona la condición de terminación en el ciclo for:

for (i=1; *solicitud[i] != '\0' ; i++) ?

¿A qué se refiere la "i"? ¿Por qué se necesita el "*"?
*/

```

===== LEEREG.C

```

/* leereg.c

Lee un archivo, registro por registro, desplegando
los campos de cada uno de los registros en la pantalla
*/
#include "arches.h"

main() {

    int fd, cont_reg, cont_campos;
    int pos_bus, long_reg;
    char nomarch[15];
    char buffreg[TAM_MAX_REG + 1];
    char campo[TAM_MAX_REG + 1];

    printf("Proporcione el nombre del archivo a leer: ");
    gets(nomarch);
    if ((fd = creat(nomarch, SOLOLECT)) < 0) {
        printf("Error en la apertura del archivo - Fin de programa\n");
        exit(1);
    }

    cont_reg = 0;
    pos_bus = 0;
}

```

```

while ((long_reg = toma_reg(fd, buffreg)) > 0)
{
    printf ("Registr %d\n", ++cont_reg);
    cont_campos > 0;
    while ((pos_bus=toma_campo(campo,buffreg,pos_bus,long_reg)) > 0)
        printf ("\tCampo %d: %s\n", ++cont_campos, campo);
}

close(fd);

/*
 * Pregunta: ¿Por qué se puede asignar 0 a pos_bus sólo una vez, fuera
 * del ciclo while para los registros? */

```

===== TOMARC.C

```
/* tomarc.c ...
```

Dos funciones que son empleadas en leereg.c y encuentra.c

toma_reg lee un registro de longitud variable del archivo fd
y lo coloca en el arreglo de caracteres buffreg.

toma_campo mueve un campo de buffreg al arreglo de caracteres
campo, insertando un '\0' para convertirlo en cadena.

```

*/
#include "arches.h"

toma_reg(fd, buffreg)
    int fd;
    char buffreg[];
{
    int long_reg;

    if (read(fd, &long_reg, 2) == 0) /* Toma la longitud del registro */
        return(0); /* Devuelve 0 si EOF */
    long_reg = read(fd, buffreg, long_reg); /* Lee el registro */
    return(long_reg);
}

toma_campo(campo,buffreg,pos_bus,long_reg)
    char campo[], buffreg[];
    short pos_bus, long_reg;
{
```

```

short cpos = 0;           /* Posición en el arreglo "campo" */

if (pos_bus == long_reg) /* Si no hay más campos que leer, */
    return(0);           /*     devuelve pos_bus de 0. */

/* Ciclo de recolección */
while (pos_bus < long_reg &&
(campo[cpos++] = buffreg[pos_bus++]) != DELIM_CAR)
;

if (campo[cpos - 1] == DELIM_CAR)      /* Si el último
                                         carácter es un campo */
    campo[-cpos] = '\0'; /* delimitador, reemplácese con nulo. */
else
    campo[cpos] = '\0'; /* En otro caso, sólo asegúrese de que
                           el campo no termina con nulo. */

return(pos_bus); /* Devuelve la posición de inicio
                  del siguiente campo. */
}

```

===== ENCUENTRA.C

```

/* tomarc.c

Busca un registro con una llave en particular, en forma
secuencial a lo largo de un archivo.

*/
#include "arches.h"
#define EXITO 1
#define FRACASO 0

main() {
    int fd, long_reg, pos_bus;
    int encontró;
    char llave_bus[30], llave_enc[30], apellido[30], nombre[30];
    char nomarch[15];
    char buffreg[TAM_MAX_REG+1];
    char campo[TAM_MAX_REG+1];

    printf("Proporcione el nombre del archivo en donde buscar: ");
    gets(nomarch);
    if ((fd = creat(nomarch, SOLOLECT)) < 0) {
        printf("Error en la apertura del archivo - Fin de programa\n");
    }
}
```

```
    exit(1);
}

printf("\n\nDigite el apellido: "); /* Toma la llave de búsqueda*/
gets(apellido);
printf("\n\nDigite el nombre: ");
gets(nombre);
hazllave(apellido, nombre, llave_bus);

encontro = FRACASO;
while (!encontró && (long_reg = toma_reg(fd,buffreg)) > 0 )
{
    pos_bus = 0;
    pos_bus = toma_campo(apellido, buffreg, pos_bus, long_reg);
    pos_bus = toma_campo(nombre, buffreg, pos_bus, long_reg);
    hazllave(apellido, nombre, llave_enc);
    if (strcmp(llave_enc, llave_bus) == 0)
        encontró = EXITO;
}
if (encontró)
{
    printf("\n\nSe encontró el registro:\n\n");
    pos_bus = 0;

    /* Divide los campos */
    while((pos_bus=toma_campo(campo,buffreg,pos_bus,long_reg)) > 0)
        printf("\t%s\n",campo);
    } else
    printf("\n\nNo se encontró el registro.\n");
}

/* Preguntas:
```

¿Por qué pos_bus se pone en cero dentro del ciclo while?

¿Qué sucedería si se escribiese el ciclo que lee los registros así:

```
while ((long_reg = toma_reg(fd,buffreg)) > 0 && !encontró )?
```

```
*/
```

===== HAZLLAVE.C

```
/* hazllave(apellido,nombre,cad) ...
```

Función que hace una llave a partir del nombre y del apellido pasados por los argumentos de la función. Devuelve la llave en forma canónica a través de la dirección pasada por el argumento cad. La rutina que llama es responsable de asegurarse de que cad sea lo suficientemente grande para guardar la cadena que se devuelve.

El valor devuelto por medio del nombre de la función es la longitud de la cadena devuelta a través de cad.

```
*/
```

```
hazllave(apellido,nombre,cad)
```

```
char apellido[], nombre[], cad[];
{
    int longap, longnom;

    longap = cadespac(apellido); /* Quita los espacios del apellido */
    strcpy(cad,apellido);        /* lo coloca en la cadena de salida */
    cad[longap++] = ' ';         /* le agrega un espacio al final */
    cad[longap] = '\0';
    longnom = cadespac(nombre); /* Quita los espacios del nombre */
    strcat(cad,nombre);         /* lo agrega a la cadena */
    mayúsculas(cad,cad);        /* convierte todo a mayúsculas */
    return(longap + longnom);
}
```

===== FUNSCADS.C

```
/* funscads.c ...
```

Módulo que contiene las siguientes funciones:

`cadespac(cad)` elimina los espacios del final de la cadena (terminada con nulo) referida por la dirección `cad`. La función trabaja de derecha a izquierda, eliminando los espacios hasta que encuentra un carácter distinto de espacio. Cuando se termina la función, el parámetro `cad` apunta a la cadena sin espacios. Mediante su nombre, la función devuelve la longitud de la cadena sin espacios.

```

mayúsculas(cadent,cadsal) convierte todos los caracteres
alfabéticos que están en minúsculas en la cadena con
dirección cadent a mayúsculas, y devuelve la cadena convertida
por medio de la dirección cadsal.

*/
cadespac(cad)
    char cad[];
{
    int i;

    for ( i = strlen(cad) - 1; i >= 0 && cad[i] == ' '; i-- )
        ;

    /* Ahora que los espacios se han eliminado, se recoloca el nulo
    al final para formar una cadena */

    cad[++i] = '\0';
    return(i);
}

mayúsculas(cadent,cadsal)
    char cadent[],cadsal[];
{
    while (*cadsal++ = (*cadent >= 'a' && *cadent <= 'z') ?
                           *cadent & 0x5F : *cadent )
        cadent++;
}

```

===== ACTUALIZA.C

```

/* actualiza.c ...

Programa que abre o crea un archivo de registros de longitud
fija para hacer la actualización. Se debe tener acceso a los
registros que serán agregados o modificados por medio del nú-
mero relativo de registro

*/
#include "arches.h"
#define LONG_REG 64
#define campo_a_buffreg(br,cad) strcat(br,cad); strcat(br,DELIM_CAD);

static char *solicitud[] = { "    Apellido:",
                            "    Nombre:",
                            "    Dirección:",

```

```

        "      Ciudad: ",
        "      Estado: ",
        " Cód. Post.: ",
        " "
    });

static int fd;
static struct {
    short  cont_reg;
    char   relleno[30];
} encabezado;

main() {
    int i,menu_elec, nrr;
    int byte_pos;
    char nomarch[15];
    long lseek();
    char buffreg[TAM_MAX_REG + 1]; /* buffer para guardar un registro */

    printf("Proporcione el nombre del archivo: ");
    gets(nomarch);
    if((fd = open(nomarch, LECTESCRIT)) < 0) /* Si falla OPEN */
    {
        fd = creat(nomarch, PMODE);           /* entonces CREAT */
        encabezado.cont_reg = 0;             /* Inicia encabezado */
        write(fd,&encabezado,sizeof(encabezado)); /* Escribe el reg. de
                                                encabezado */
    }
    else      /* Se abre el archivo existente - se lee el encabezado */
        read(fd,&encabezado,sizeof(encabezado));

    /* Ciclo del programa principal - llama al menú y después salta a
       las opciones */
    while((menú_elec = menu()) < 3)
    {
        switch(menú_elec)
        {
            case 1:          /* Agrega un registro nuevo */
                printf("Proporcione la información del registro nuevo -\n\n");
                pide_info(buffreg);
                byte_pos = encabezado.cont_reg * LONG_REG +
                           sizeof(encabezado);
                lseek(fd, (long) byte_pos, 0);
                write(fd,buffreg,LONG_REG);
                encabezado.cont_reg++;
                break;

            case 2:          /* Actualiza un registro existente */

```



```
elección = atoi(respuesta);
return(elección);
}

/* pide_info() ...
   Función local que lee los campos de nombre y dirección,
   transcribiéndolos al buffer que se pasa como parámetro
*/
static pide_info(buffreg)
    char buffreg[];
{
    int cont_campos, i;
    char respuesta[50];

    /* Limpia el buffer del registro */
    for (i = 0; i < LONG_REG; buffreg[i++] = '\0')

    /* Toma los campos */
    for (i=0; *solicitud[i] != '\0'; i++)
    {
        printf("%s", solicitud[i]);
        gets(respuesta);
        campo_a_buffreg(buffreg, respuesta);
    }
}

/* pide_nrr() ...
   Función local que pide el número relativo del registro
   que se actualizará.
*/
static pide_nrr()
{
    int nrr;
    char respuesta[10];

    printf("\n\nDigite el número relativo del registro\n");
    printf("\tque desee actualizar: ");
    gets(respuesta);
    nrr = atoi(respuesta);
    return(nrr);
}

/* lee_y_muestra() ...
   Función local que lee y despliega un registro. Nótese que esta
   función no incluye localización: la lectura empieza en
   la posición actual del archivo
*/
static lee_y_muestra()
```

```
{  
    char buffreg[TAM_MAX_REG + 1], campo[TAM_MAX_REG + 1];  
    int pos_bus, long_datos;  
  
    pos_bus = 0;  
    read(fd, buffreg, LONG_REG);  
  
    printf("\n\n\n\nContenido del registro existente\n");  
  
    buffreg[LONG_REG] = '\0'; /* Se asegura que el registro termine con  
                           nulo */  
    long_datos = strlen(buffreg);  
    while((pos_bus=toma_campo(campo, buffreg, pos_bus, long_datos)) > 0)  
        printf("\t%s\n", campo);  
}  
  
/* cambio() ...  
   Función local que pregunta al usuario si quiere cambiar el  
   registro. Devuelve 1 si la respuesta es sí y 0 en caso contrario  
*/  
static cambio() {  
    char respuesta[10];  
  
    printf("\n\n¿Desea modificar este registro?\n");  
    printf(" Conteste S o N, seguido por <ENTER> ==>");  
    gets(respuesta);  
    mayúsculas(respuesta, respuesta);  
    return((respuesta[0] == 'S') ? 1 : 0);  
}
```

PROGRAMAS EN PASCAL

Los programas en Pascal listados en las siguientes páginas corresponden a los programas analizados en el texto. Cada programa está organizado en uno o más archivos, como sigue.

<i>escribesec.pas</i>	Escribe la información de nombres y direcciones como una secuencia de bytes consecutivos.
<i>leesec.pas</i>	Lee un archivo de secuencia de bytes como entrada y lo muestra en la pantalla.
<i>escribereg.pas</i>	Escribe un archivo de registros de longitud variable que usa un contador de bytes al inicio de cada registro para proporcionar su longitud.
<i>leereg.pas</i>	Lee un archivo, registro por registro, mostrando los campos de cada uno de los registros en la pantalla.
<i>toma.prc</i>	Funciones de apoyo para la lectura de registros o campos individuales. Estas funciones son necesarias para el programa <i>leereg.pas</i> .
<i>encuentra.pas</i>	Busca un registro con una llave en particular en forma secuencial a lo largo de un archivo.
<i>actualiza.pas</i>	Permite agregar registros nuevos a un archivo, o modificar registros existentes.
<i>caddat.prc</i>	Función de apoyo para <i>actualiza.pas</i> , que convierte una variable de tipo <i>cadena</i> en una variable de tipo <i>regdato</i> .

Además de estos archivos, existe un archivo llamado *herramientas.prc*, que contiene las herramientas para trabajar con variables de tipo *cadena*. Un listado de *herramientas.prc* aparece en el Apéndice B, al final del libro.

Se han agregado los números de línea a algunos de estos listados en Pascal, para ayudar al lector a encontrar proposiciones de programa específicas.

Los archivos que contienen funciones o procedimientos de Pascal, pero que no contienen programas principales, están dados por la extensión *.prc*, como en *toma.prc* y en *caddat.prc*.

===== ESCRIBESEC.PAS

Algunos aspectos de *escribesec.pas* que se deben considerar son:

- El comentario {\$B-} en la línea 6 es una directiva para el compilador de Turbo Pascal, que indica manejar la entrada del teclado como un archivo estándar de Pascal. Sin esta directiva no se podría manejar apropiadamente la función *long_cad()* en el ciclo WHILE de la línea 36.
- El comentario {\$I herramientas.prc}, en la línea 24, es también una directiva para el compilador de Turbo Pascal, para que incluya el archivo *herramientas.prc* en la compilación. Los procedimientos *lee_cad*, *long_cad*, y *escribe_cad* están en el archivo *herramientas.prc*.
- Aunque Turbo Pascal maneja un tipo especial de cadenas, aquí se ha preferido no usarlo, para tener mayor similitud con Pascal estándar. En su lugar, se creó un tipo *cadena* propio, que es un *packed array [0..TAM_MAX_REG] of char*. La longitud de la *cadena* se almacena en el byte cero del arreglo, como un valor de tipo carácter. Si X es el carácter almacenado en el byte cero del arreglo, entonces la longitud de la cadena es *ORD(X)*.
- La proposición de asignación de la línea 31 no es estándar. Es un procedimiento de Turbo Pascal que, en este caso, asigna un *nomarch a archivosal*, de manera que todas las operaciones siguientes sobre *archivosal* operen sobre el archivo del disco.

```

1: PROGRAM escribesec (INPUT,OUTPUT);
2:
3: { Escribe la información de nombres y direcciones como una secuencia
4:   de bytes consecutivos }
5:
6: {$B-}{ Directiva para el compilador de Turbo Pascal, indicándole que
7:   maneje el teclado como un archivo estándar de Pascal }
8:
9: CONST
10:    DELIM_CAR = '|';
11:    TAM_MAX_REG = 255;
12:
13: TYPE
14:    cadena      = packed array [0..TAM_MAX_REG] of char;
15:    lista_ent   = (apellido,nombre,dirección,ciudad,estado,codpost);
16:    tipArchivo = packed array[1..40] of char;
17:
18: VAR
19:    respuesta : array [lista_ent] of cadena;

```

(continuación)

```
20:     tipo_resp : lista_ent;
21:     nomarch   : tipoarchivo;
22:     archivosal : text;
23:
24: {$I herramientas.prc}
25: { Otra directiva que indica al compilador incluir el archivo
26:   herramientas.prc }
27:
28: BEGIN {principal}
29:   write('Proporcione el nombre del archivo: ');
30:   readln(nomarch);
31:   assign(archivosal,nomarch);
32:   rewrite(archivosal);
33:
34:   write('Digite un apellido, o <CR> para salir: ');
35:   lee_cad(respuesta[apellido]);
36:   while (long_cad(respuesta[apellido]) > 0) DO
37:     BEGIN
38:       { Toma todos los datos de una persona }
39:       write('    Nombre:');
40:       lee_cad(respuesta[nombre]);
41:       write('    Dirección:');
42:       lee_cad(respuesta[dirección]);
43:       write('    Ciudad:');
44:       lee_cad(respuesta[ciudad]);
45:       write('    Estado:');
46:       lee_cad(respuesta[estado]);
47:       write('Cod. Post.:');
48:       lee_cad(respuesta[codpost]);
49:
50:       { escribe las respuestas al archivo }
51:       for tipo_resp := apellido TO codpost DO
52:         escribe_arch_cad(archivosal,respuesta[tipo_resp]);
53:
54:         { inicia la siguiente vuelta de lectura }
55:         write('Digite un apellido, o <CR> para salir\n>>>');
56:         lee_cad(respuesta[apellido]);
57:     END;
58:   close(archivosal)
59: END.
```

===== LEESEC.PAS

```

PROGRAM leesec (INPUT,OUTPUT);

{ Programa que lee un archivo de secuencias de bytes (campos separados
por delimitadores) como entrada y lo despliega en la pantalla }

CONST
  DELIM_CAR = '|';
  TAM_MAX_REG = 255;

TYPE
  cadena      = packed array [0..TAM_MAX_REG] of char;
  tipoarchivo = packed array[1..40] of char;

VAR
  nomarch    : tipoarchivo;
  archivoent : text;
  cont_campos: integer;
  long_campo : integer;
  cad        : cadena;

{$I herramientas.prc}

FUNCTION leecampo(VAR archivoent: text; VAR cad : cadena): integer;

{ Función que lee caracteres del archivo archivoent hasta que alcanza
el final del archivo o una '|'. Leecampo pone los caracteres en cad
y devuelve la longitud de cad }

VAR
  i : integer;
  car : char;
BEGIN
  i := 0;
  car := ' ';
  while (not EOF(archivoent)) and (car <> DELIM_CAR) DO
    BEGIN
      read (archivoent,car);
      i := i + 1;
      cad[i] := car
    END;
  i := i - 1;
  cad[0] := CHR(i);
  leecampo := i
END;

```

(continuación)

```

BEGIN { principal }
  write('Proporcione el nombre del archivo que quiera abrir: ');
  readln(nomarch);
  assign(archivoent,nomarch);
  reset(archivoent);

  cont_campos := 0;

  long_campo := leecampo(archivoent,cad);
  while (long_campo > 0) DO
    BEGIN
      cont_campos := cont_campos + 1;
      write(' campo #':10,cont_campos:1,'':2);
      escribe_cad(cad);           { escribe_cad está en herramientas.prc }
      long_campo := leecampo(archivoent,cad)
    END;
  close(archivoent)
END.

```

===== ESCRIBEREG.PAS

Nota acerca de *escribereg.pas*: después de escribir la *long_reg* al archivo de salida en la línea 69, se escribe un espacio en el archivo. Esto se debe a que en Pascal los valores que se van a leer en variables enteras deben estar separados por espacios, tabuladores, o marcas de fin de línea.

```

1: PROGRAM escribereg (INPUT,OUTPUT);
2:
3: {$B-}
4:
5: CONST
6:   DELIM_CAR = '|';
7:   TAM_MAX_REG = 255;
8:
9: TYPE
10:  cadena      = packed array [0..TAM_MAX_REG] of char;
11:  tipoarchivo = packed array[1..40] of char;
12:
13: VAR
14:  nomarch     : tipoarchivo;
15:  archivosal  : text;
16:  respuesta   : cadena;
17:  buffer      : cadena;
18:  long_reg    : integer;
19:

```

```
20: {$I herramientas.prc}
21:
22:
23: PROCEDURE campo_a_buffer(buffer: cadena; cad: cadena);
24:
25: { Este procedimiento concatena cad y un delimitador al final de
26:   buffer }
27:
28: VAR
29:   cad_dato : cadena;
30: BEGIN
31:   conc_cad(buffer,cad);
32:   cad_dato[0] := CHR(1);
33:   cad_dato[1] := DELIM_CAR;
34:   conc_cad(buffer,cad_dato)
35: END;
36:
37:
38: BEGIN (principal)
39:   write('Proporcione el nombre del archivo que desee crear: ');
40:   readln(nomarch);
41:   assign(archivosal,nomarch);
42:   rewrite(archivosal);
43:
44:   write('Digite el apellido - o <CR> para salir: ');
45:   lee_cad(respuesta);
46:   while (long_cad(respuesta) > 0) DO
47:     BEGIN
48:       buffer[0] := CHR(0);           { Hace la longitud de la
49:                                         cadena en el buffer igual a 0 }
50:       campo_a_buffer(buffer,respuesta);
51:       write('                           Nombre:');
52:       lee_cad(respuesta);
53:       campo_a_buffer(buffer,respuesta);
54:       write('                           Dirección:');
55:       lee_cad(respuesta);
56:       campo_a_buffer(buffer,respuesta);
57:       write('                           Ciudad:');
58:       lee_cad(respuesta);
59:       campo_a_buffer(buffer,respuesta);
60:       write('                           Estado:');
61:       lee_cad(respuesta);
62:       campo_a_buffer(buffer,respuesta);
63:       write('                           Cód. Post.:');
64:       lee_cad(respuesta);
65:       campo_a_buffer(buffer,respuesta);
66:
```

(continuación)

```

67:      { Escribe la longitud del registro y el contenido del buffer }
68:      long_reg := long_cad(buffer);
69:      write(archivosal,long_reg);
70:      write(archivosal,' ');
71:      escribe_arch_cad(archivosal,buffer);
72:
73:      { Se prepara para el siguiente registro }
74:      write('Digite el apellido - o <CR> para salir: ');
75:      lee_cad(respuesta)
76:      END;
77:      close(archivosal)
78: END.

```

===== LEEREG.PAS

```

PROGRAM leereg (INPUT,OUTPUT);

{ Este programa lee un archivo, registro por registro,
desplegando los campos de cada uno de los registros en la pantalla. }

{$B-}

CONST
  tamaño_ent = 255;
  DELIM_CAR = '|';
  TAM_MAX_REG = 255;

TYPE
  cadena      = packed array [0..TAM_MAX_REG] of char;
  tipoarchivo = packed array[1..40] of char;

VAR
  nomarch    : tipoarchivo;
  archivosal : text;
  cont_regs  : integer;
  pos_bus    : integer;
  long_reg   : integer;
  cont_campos: integer;
  buffer     : cadena;
  campo      : cadena;

{$I herramientas.prc}
{$I toma.prc}

```

```

BEGIN {principal}
  write('Proporcione el nombre del archivo por leer: ');
  readln(nomarch);
  assign(archivosal,nomarch);
  rewrite(archivosal);

  cont_regs := 1;
  pos_bus := 0;
  long_reg := toma_reg(archivosal,buffer);
  while long_reg > 0 DO
    BEGIN
      writeln('Registro ',cont_regs);
      cont_regs := cont_regs + 1;
      cont_campos := 1;
      pos_bus := toma_campo(campo,buffer,pos_bus,long_reg);
      while pos_bus > 0 DO
        BEGIN
          write('      Campo ',cont_campos,' : ');
          escribe_cad(campo);
          cont_campos := cont_campos + 1;
          pos_bus := toma_campo(campo,buffer,pos_bus,long_reg)
        END;
      long_reg := toma_reg(archivosal,buffer)
    END;

  close(archivosal)
END.

```

===== TOMA.PRC

```

FUNCTION toma_reg(VAR fd: text; VAR buffer: cadena):integer;

{ Función que lee un registro y su longitud del archivo fd. La
función devuelve la longitud del registro. Toma_reg() devuelve 0
si se encuentra EOF }

VAR
  long_reg : integer;
  espacio : char;
BEGIN
  if EOF(fd) then
    toma_reg := 0
  else
    BEGIN

```

```

read(fd,long_reg);
read(fd,espacio);
lee_arch_cad(fd,buffer,long_reg);
toma_reg := long_reg
END
END;

FUNCTION toma_campo(VAR campo: cadena;buffer: cadena;Var posbus: integer;
                     long_reg: integer):integer;

{ Función que empieza leyendo en posbus y lee caracteres del buffer
hasta que alcanza un delimitador o el final del registro. Devuelve
pos_bus para que se use en la siguiente llamada. }

VAR
  posf : integer;
BEGIN
  if posbus = long_reg then
    toma_campo := 0
  else
    BEGIN
      posf := 1;
      posbus := posbus + 1;
      campo[posf] := buffer[posbus];
      while (campo[posf] <> DELIM_CAR) and (posbus < long_reg) DO
        BEGIN
          posf := posf + 1;
          posbus := posbus + 1;
          campo [posf] := buffer[posbus]
        END;
      if campo[posf] = DELIM_CAR then
        campo[0] := CHR(posf - 1)
      else
        campo[0] := CHR(posf);
      toma_campo := posbus
    END
  END;
END;

```

ENCUENTRA.PAS

PROGRAM encuentra (INPUT,OUTPUT);

{ Este programa lee un archivo, registro por registro,
buscando un registro con una llave en particular. Si hay corres-
pondencia, entonces se despliegan todos los campos del registro.
De lo contrario, se despliega un mensaje de error indicando que
no se encontró el registro. }

```

{$B-}

CONST
  TAM_MAX_REG = 255;
  DELIM_CAR = '|';

TYPE
  cadena      = packed array [0..TAM_MAX_REG] of char;
  tipoarchivo = packed array[1..40] of char;

VAR
  nomarch      : tipoarchivo;
  archivosal   : text;
  apellido     : cadena;
  nombre       : cadena;
  llave_bus    : cadena;
  longitud     : integer;
  encontró     : boolean;
  long_reg     : integer;
  buffer       : cadena;
  pos_bus      : integer;
  llave_encont : cadena;
  campo        : cadena;

{$I herramientas.prc}
{$I toma.prc}
BEGIN {principal}
  write('Proporcione el nombre del archivo para búsqueda: ');
  readln(nomarch);
  assign(archivosal,nomarch);
  rewrite(archivosal);

  write('Digite el apellido: ');
  lee_cad(apellido);
  write('Digite el nombre: ');
  lee_cad(nombre);
  hazllave(apellido,nombre,llave_bus);

  encontró := FALSE;
  long_reg := toma_reg(archivosal,buffer);
  while ((not encontró) and (long_reg > 0)) DO
    BEGIN
      pos_bus := 0;
      pos_bus := toma_campo(apellido,buffer,pos_bus,long_reg);
      pos_bus := toma_campo(nombre,buffer,pos_bus,long_reg);
      hazllave(apellido,nombre,llave_encont);

```

```

if comp_cad(llave_encont,llave_bus) = 0 then
    encontró := TRUE
else
    long_reg := toma_reg(archivosal,buffer);
END;
close(archivosal);

{ Si se encontró el registro, se imprimen los campos }
if encontró then
BEGIN
writeln('Se encontró el registro:');
writeln;
pos_bus := 0;

{ Divide los campos }
pos_bus := toma_campo(campo,buffer,pos_bus,long_reg);
while pos_bus > 0 DO
BEGIN
    escribe_cad(campo);
    pos_bus := toma_campo(campo,buffer,pos_bus,long_reg)
END;
END
else
writeln(' El registro no se encontró.');
END.

```

===== ACTUALIZA.PAS

Algunos aspectos acerca de *actualiza.pas* que se deben considerar son:

- En el procedimiento *pide_info()*, los campos de nombre y dirección se leen como *cadenas*, y el procedimiento *campo_a_buffer()* escribe los campos a *buffcad* (también de tipo *cadena*). La escritura de *buffcad* a *archivosal* daría como resultado una falta de correspondencia en los tipos, ya que *archivosal* es un archivo de tipo *regdato*. Sin embargo, el procedimiento *caddat*, localizado en *caddat.prc*, convierte una variable del tipo *cadena* en una variable del tipo *regdato* para transcribir el buffer al archivo. Las llamadas a *caddat()* se localizan en las líneas 210 y 237.
- Las proposiciones *seek()* de las líneas 212, 229, 239 y 250 no son estándar, son características de Turbo Pascal.

```
1: PROGRAM actualiza (INPUT,OUTPUT);
2:
3: {$B-}
4:
5: { Programa que abre o crea un archivo de registros de longitud
6:   fija para hacer la actualización. Los registros pueden agregarse
7:   o modificarse. A los registros por agregar o modificar
8:   se debe tener acceso por medio del número relativo de registro }
9: CONST
10:    TAM_MAX_REG = 255;
11:    LONG_REG = 64;
12:    DELIM_CAR = '|';
13:
14: TYPE
15:    cadena      = packed array [0..TAM_MAX_REG] of char;
16:    tipoarchivo = packed array[1..40] of char;
17:    regdato     = RECORD
18:        longitud : integer;
19:        datos      : packed array [1..LONG_REG] of char
20:    END;
21:
22: VAR
23:    nomarch     : tipoarchivo;
24:    archivosal : file of regdato;
25:    respuesta   : char;
26:    menu_elec   : integer;
27:    buffcad    : cadena;
28:    pos_byte    : integer;
29:    encabezado : regdato;
30:    nrr         : integer;
31:    regbuffdat : regdato;
32:    i           : integer;
33:    cont_regs  : integer;
34: {$I herramientas.prc}
35: {$I caddat.prc}
36: {$I toma.prc}
37:
38:
39: PROCEDURE campo_a_buffer(buffer: cadena; cad: cadena);
40:
41: { Este procedimiento concatena cad y un delimitador al final del
42:   buffer }
43:
44: VAR
45:    cad_dato   : cadena;
46: BEGIN
```

```
47:     conc_cad(buffer,cad);
48:     cad_dato[0] := CHR(1);
49:     cad_dato[1] := DELIM_CAR;
50:     conc_cad(buffer,cad_dato)
51: END;
52:
53:
54: FUNCTION menú:integer;
55:
56: { Función local que pide al usuario la siguiente operación.
57: Devuelve el valor numérico de la respuesta del usuario }
58:
59: VAR
60:     elección : integer;
61: BEGIN
62:     writeln;
63:     writeln('                                PROGRAMA DE ACTUALIZACION DEL ARCHIVO');
64:     writeln;
65:     writeln('Ud. Puede elegir:');
66:     writeln;
67:     writeln('      1. Agregar un registro al final del archivo');
68:     writeln('      2. Extraer un registro para actualizarlo');
69:     writeln('      3. Salir del programa');
70:     writeln;
71:     write('Proporcione el número de su elección: ');
72:     readln(elección);
73:     writeln;
74:     menú := elección
75: END;
76: PROCEDURE pide_info(VAR buffcad: cadena);
77:
78: {Procedimiento local que lee los campos de nombre y dirección,
79: escribiéndolos al buffer que se pasa como parámetro }
80:
81: VAR
82:     respuesta : cadena;
83: BEGIN
84:     { Limpia el buffer del registro }
85:     limpia_cad(buffcad);
86:
87:     { Toma los campos }
88:     write('                            Apellido:');-
89:     lee_cad(respuesta);
90:     campo_a_buffer(buffcad,respuesta);
91:     write('                            Nombre:');-
92:     lee_cad(respuesta);
93:     campo_a_buffer(buffcad,respuesta);
94:     write('                            Dirección:');
```

```
95:    lee_cad(respuesta);
96:    campo_a_buffer(buffcad, respuesta);
97:    write('                                Ciudad:');
98:    lee_cad(respuesta);
99:    campo_a_buffer(buffcad, respuesta);
100:   write('                               Estado:');
101:   lee_cad(respuesta);
102:   campo_a_buffer(buffcad, respuesta);
103:   write('                                Cód. Post.:');
104:   lee_cad(respuesta);
105:   campo_a_buffer(buffcad, respuesta);
106:   writeln
107: END;
108:
109:
110: FUNCTION pide_nrr: integer;
111:
112: { Función que pide el número relativo del registro
113:   que se actualizará. }
114:
115: VAR
116:   nrr      : integer;
117: BEGIN
118:   writeln('Proporcione el número relativo del');
119:   writeln('registro que desea actualizar: ');
120:   readln(nrr);
121:   writeln;
122:   pide_nrr := nrr
123: END;
124: PROCEDURE lee_y_muestra;
125:
126: { Procedimiento para leer y desplegar un registro. Este
127:   procedimiento no incluye localización: la lectura empieza
128:   en la posición actual del archivo }
129: VAR
130:   pos_bus    : integer;
131:   regbuffdat: regdato;
132:   i          : integer;
133:   long_datos: integer;
134:   campo      : cadena;
135:   buffcad    : cadena;
136: BEGIN
137:   pos_bus := 0;
138:   read(archivosal, regbuffdat);
139:
140:   { Convierte regbuffdat en tipo cadena }
```

```
141:     buffcad [0] := CHR(regbuffdat.longitud);
142:     for i := 1 to regbuffdat.longitud DO
143:         buffcad[i] := regbuffdat.datos[i];
144:
145:     writeln('Contenido del registro existente');
146:     writeln;
147:
148:     long_datos := long_cad(buffcad);
149:     pos_bus := toma_campo(campo,buffcad,pos_bus,long_datos);
150:     while pos_bus > 0 DO
151:         BEGIN
152:             escribe_Cad(campo);
153:             pos_bus := toma_campo(campo,buffcad,pos_bus,long_datos)
154:         END
155:     END;
156:
157:
158: FUNCTION cambio: integer;
159:
160: { Función que pregunta al usuario si quiere cambiar el registro.
161:   Devuelve 1 si la respuesta es sí y 0 en caso contrario. }
162:
163: VAR
164:     respuesta : char;
165: BEGIN
166:     writeln('¿Desea modificar este registro?');
167:     write('        Conteste S o N, seguido por <ENTER> ==>');
168:     readln(respuesta);
169:     writeln;
170:     if (respuesta = 'S') or (respuesta = 's') then
171:         cambio := 1
172:     else
173:         cambio := 0
174: END;
175: BEGIN {principal}
176:     write('Proporcione el nombre del archivo: ');
177:     readln(nomarch);
178:     assign(archivosal,nomarch);
179:
180:     write(' ¿El archivo ya existe? (responda S o N): ');
181:     readln(respuesta);
182:     writeln;
183:     if (respuesta = 'S') OR (respuesta = 's') then
184:         BEGIN
185:             reset(archivosal);           { Abre archivosal      }
186:             read(archivosal,encabezado); { Toma el encabezado  }
187:             cont_regs := encabezado.longitud { Lee el n m. de regs. }
188:         END
```

```

189: else
190:   BEGIN
191:     rewrite(archivosal);           { Crea archivosal      }
192:     cont_regs := 0;              { Inicia el núm. de regs.  }
193:     encabezado.longitud := cont_regs; { lo coloca en el reg.  }
194:     for i := 1 to LONG_REG DO      { de encabezado      }
195:       encabezado.datos[i] := CHR(0); { pone los datos del enc. }
196:     write(archivosal,encabezado)    { a nulo; escribe el reg. }
197:   END;                         { de encabezado      }
198:
199:{ Ciclo del prog. princ.-llama al menú y después salta a las ops. }
200:   menú_elec := menú;
201:   while menú_elec < 3 DO
202:     BEGIN
203:       CASE menú_elec of
204:         1 :          { Agrega un registro nuevo }
205:           BEGIN
206:             writeln('Proporcione la inf. del registro nuevo --');
207:             writeln;
208:             writeln;
209:             pide_info(buffcad);
210:             caddat(regbuffdat,buffcad); { Convierte cadbuff }
211:             nrr := cont_regs + 1;       { al tipo regdatos  }
212:             seek(archivosal,nrr);
213:             write(archivosal,regbuffdat);
214:             cont_regs := cont_regs + 1
215:           END;
216:         2:          { Actualiza un registro existente }
217:           BEGIN
218:             nrr := pide_nrr;
219:
220:             { Si el NRR es muy grande, imprime mensaje de error ... }
221:             if (nrr > cont_regs) or (nrr < 1) then
222:               BEGIN
223:                 writeln('El número de reg. está fuera de margen');
224:                 writeln('... se regresa al menú ...')
225:               END
226:
227:             else { en caso contrario, se coloca en el registro ...}
228:               BEGIN
229:                 seek(archivosal, nrr);
230:
231:                 { Lo despliega y pide los cambios }
232:                 lee_y_muestra;
233:                 if cambio = 1 then
234:                   BEGIN

```

```

235:           writeln('Proporcione los valores revisados: ');
236:           pide_info(buffcad);
237:           caddat(regbuffdat,buffcad); { Convierte buffcad
238:                                     al tipo regdatos }
239:           seek(archivosal, nrr);
240:           write(archivosal,regbuffdat)
241:           END
242:           END
243:           END .
244:       END; { CASE }
245:       menu_elec := menu
246:   END; { while }
247:
248: { Reescribe el cont. de regs. correcto en el encab. antes de ... }
249:     encabezado.longitud := cont_regs;      { ... terminar }
250:     seek(archivosal,0);
251:     write(archivosal,encabezado);
252:     close(archivosal)
253: END.

```

===== CADDAT.PRC

```

PROCEDURE caddat(VAR regbuffdat: regdato; buffcad: cadena);

{ Procedimiento que convierte una variable de tipo cadena en una
  variable de tipo regdatos }

VAR
  i : integer;
BEGIN
  regbuffdat.longitud := min(LONG_REG,long_cad(buffcad));
  for i := 1 to regbuffdat.longitud DO
    regbuffdat.datos[i] := buffcad[i];

  { Limpia el resto del buffer }
  while i < LONG_REG DO
    BEGIN
      i := i + 1;
      regbuffdat.datos[i] := ' '
    END
END;

```

OBJETIVOS

Estudiar la *compactación del almacenamiento* como una forma sencilla de reutilización del espacio en un archivo.

Desarrollar un procedimiento para la eliminación de registros de longitud fija que permita la reutilización dinámica del espacio vacante del archivo.

Ilustrar el uso de *listas ligadas* y *pilas* para manejar una *lista de disponibles*.

Considerar diversos enfoques en el problema de la eliminación de registros de longitud variable.

Introducir los conceptos asociados con los términos *fragmentación interna* y *fragmentación externa*.

Plantear algunas *estrategias de colocación* asociadas con la reutilización del espacio en un archivo de registros de longitud variable.

5

MANTENIMIENTO DE ARCHIVOS Y ELIMINACION DE REGISTROS

PLAN GENERAL DEL CAPITULO

- 5.1 Mantenimiento de archivos**
- 5.2 Compactación del almacenamiento**
- 5.3 Panorama de la eliminación de registros de longitud fija**
- 5.4 Realización de la eliminación de registros de longitud fija**
- 5.5 Eliminación de registros de longitud variable**
- 5.6 Fragmentación del almacenamiento**
- 5.7 Estrategias de colocación**

5.1

MANTENIMIENTO DE ARCHIVOS

Ya se ha visto lo importante que es para el diseñador de sistemas de archivos considerar la forma en que se accede a ellos cuando se organiza el archivo. En este capítulo se verá que el diseñador también debe considerar los tipos de *cambios* que probablemente tendrán lugar en la vida de un archivo. Si un archivo es muy *volátil* (sometido a inserciones o eliminaciones frecuentes) y se usa en un ambiente de tiempo real, la organización del archivo debe facilitar cambios rápidos en registros individuales en tiempo real, sin interferir el acceso del usuario al archivo. Un ejemplo de un archivo volátil usado en tiempo real es un archivo de reservaciones en un sistema de reservaciones en línea.

En el otro extremo está un archivo fuera de línea, el cual se somete a relativamente pocos cambios y no necesita mantenerse absolutamente actualizado; puede actualizarse en procesamiento por lotes y no precisa que se incluyan estructuras adicionales para facilitar cambios rápidos. Un ejemplo de este tipo de archivos puede ser un archivo de lista de correos.

El mantenimiento de archivos es importante, pues el desempeño se deteriora conforme se hacen cambios al archivo. Por ejemplo, supongamos que se modifica un registro en un archivo de registros de longitud variable, de tal forma que el nuevo registro resulta ser más grande que el original. ¿Qué se hace con los datos adicionales? Se podrían agregar

al final del archivo y colocar un apuntador desde el espacio original del registro hasta la ampliación del registro. Se podría reescribir el registro completo al final del archivo (a menos que el archivo necesite clasificarse), dejando un hueco en la localidad original del registro. Cada una de estas soluciones tiene un inconveniente: en el primer caso, el procesamiento del registro es más difícil de lo que era originalmente; en el último caso, el archivo contiene espacio desperdiciado.

En este capítulo se estudiará con mayor detalle la forma en que se deteriora la organización del archivo conforme se modifica. En general, las modificaciones pueden ser de una de estas tres formas:

- Agregar un registro;
- Actualizar un registro, y
- Eliminar un registro.

Si la única clase de cambios posible en un archivo es agregar registros, no sucede la clase de deterioro que se abarca en este capítulo. (Los capítulos anteriores describen cómo realizar la adición simple de registros.) Sólo cuando se actualizan registros de longitud variable, o cuando se eliminan registros de longitud fija o variable es que los aspectos del mantenimiento se vuelven complejos e interesantes. Como la actualización de registros siempre se puede tratar como una eliminación de registro seguida por la inserción de otro, este capítulo se centrará en los efectos de la eliminación de registros. Cuando se elimina un registro se desea reutilizar el espacio. Se comienza examinando la *compactación del almacenamiento*, que es el más simple y más utilizado de los métodos de recuperación de almacenamiento que se analizan aquí.

5.2

COMPACTACION DEL ALMACENAMIENTO

Cualquier estrategia de eliminación de registros debe proporcionar alguna forma de reconocer los registros que se han eliminado. Un enfoque sencillo y que normalmente funciona, es colocar una marca especial en cada registro eliminado. Por ejemplo, en el archivo de nombres y direcciones desarrollado en los capítulos anteriores, se puede colocar simplemente un asterisco como primer campo en un registro eliminado. Las figuras 5.1(a) y (b) muestran un archivo de nombres y direcciones semejante al del capítulo 4 antes y después de que el segundo registro se marque como eliminado. (Los puntos al final de los

Ames|John|123 Maple|Stillwater|OK|74075.....
 Morrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
 Brown|Martha|625 Kimbark|Des Moines|IA|50311|

(a)

Ames|John|123 Maple|Stillwater|OK|74075|.....
 *|rrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
 Brown|Martha|625 Kimbark|Des Moines|IA|50311|

(b)

Ames|John|123Maple|Stillwater|OK|74075|.....
 Brown|Martha|625 Kimbark|Des Moines|IA|50311|.....

(c)

FIGURA 5.1 • Requerimientos de almacenamiento para el archivo de ejemplo empleando registros de longitud fija de 64 bytes. (a) Antes de eliminar el segundo registro. (b) Despues de eliminar el segundo registro. (c) Despues de la compactación: el segundo registro desaparece.

registros 0 y 2 representan el relleno entre el último campo y el final de cada registro.)

Una vez que se puede reconocer un registro como eliminado, la siguiente cuestión es cómo reutilizar el espacio del registro. Los enfoques dados a este problema que se basan en la compactación del almacenamiento no hacen nada para reutilizar de inmediato el espacio. Los registros simplemente se marcan como eliminados y se dejan en el archivo durante un lapso. Los programas que usan el archivo deben incluir una cierta lógica para que se ignoren los registros que están marcados como eliminados. Un efecto colateral positivo de este enfoque es que normalmente permite al usuario anular la eliminación de un registro con muy poco esfuerzo. Esto es particularmente fácil si se guarda la marca de eliminado en un campo especial, en lugar de destruir algunos de los datos originales como en el ejemplo.

El espacio de todos los registros eliminados se recupera al mismo tiempo. Después de que los registros eliminados se han acumulado por algún lapso, se emplea un programa especial de compactación de almacenamiento que reconstruye el archivo ya sin los registros eliminados (Fig. 5.1c). Si se cuenta con espacio suficiente, la forma más simple de hacer la compactación es mediante un programa de copia de archivos que ignora los registros eliminados. También es posible hacer la compactación en el lugar mismo, aunque es más complicado y tardado. Cualquiera de estos enfoques puede utilizarse con registros de longitud fija y variable.

La decisión respecto a la frecuencia con que se ejecuta el programa de compactación de almacenamiento puede basarse bien en el número de registros eliminados o en el calendario. Por ejemplo, en programas de contabilidad tiene sentido ejecutar un procedimiento de compactación en ciertos archivos al final del año fiscal o en algún punto asociado con el cierre de los libros.

La compactación del almacenamiento es una forma aceptable de recuperación del espacio para muchas aplicaciones. Sin embargo, hay otras demasiado volátiles e interactivas para las cuales no es de utilidad. En estas situaciones se necesita reutilizar el espacio de los registros eliminados tan pronto como sea posible. Comenzamos el análisis de la recuperación dinámica del almacenamiento con los archivos de registros de longitud fija, ya que éstos hacen mucho más sencillo el problema de la recuperación.

5.3

PANORAMA DE LA ELIMINACION DE REGISTROS DE LONGITUD FIJA

En general, para proporcionar un mecanismo de eliminación de registros con la subsecuente reutilización del espacio liberado, es necesario garantizar dos cosas:

- Que los registros eliminados se marquen de alguna forma especial, y
- Que se pueda encontrar el espacio que los registros eliminados ocupaban, para reutilizarlo cuando se agreguen registros.

Ya se ha identificado un método que cumple con el primer requisito: marcar los registros como eliminados, colocando un campo que contenga un '*' al inicio de los registros eliminados.

Si se trabaja con registros de longitud fija y se está dispuesto a hacer una búsqueda secuencial a través del archivo antes de agregar un registro, siempre se podrá proporcionar la segunda garantía, si se tiene la primera. Cuando el programa puede distinguir un registro eliminado, la reutilización del espacio puede darse en forma de búsqueda en el archivo, registro por registro, hasta que se encuentra un registro eliminado. Si el programa llega al final del archivo sin encontrar un registro eliminado, entonces el nuevo registro puede agregarse allí, al final.

Desafortunadamente, este método para agregar registros es un proceso intolerablemente lento cuando el programa es interactivo, y el

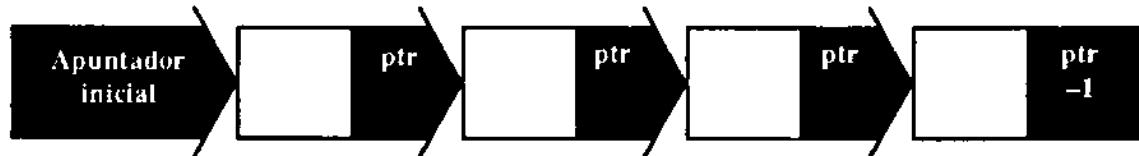


FIGURA 5.2 • Una lista ligada.

usuario se ve obligado a esperar a que ocurra la inserción del registro. Para que esto se realice más rápido se necesita:

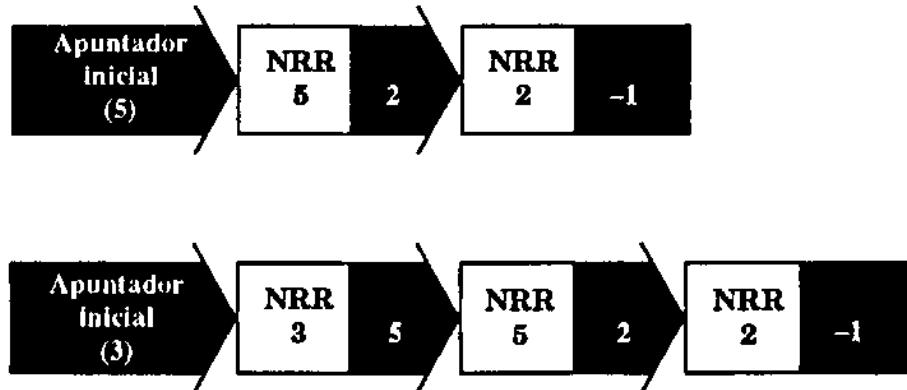
- Una forma de saber de inmediato si hay lugares vacíos en el archivo, y
- Una forma de saltar directamente a uno de esos lugares, en caso de existir.

LISTAS LIGADAS. El uso de una *lista ligada* para unir todos los registros disponibles puede cubrir estas necesidades. Una lista ligada es una estructura de datos en la que cada elemento o *nodo* contiene algún tipo de referencia sobre su sucesor en la lista.

Si se tiene una referencia inicial al primer nodo de la lista, ésta puede recorrerse examinando cada nodo y luego el campo apuntador del nodo, de manera que se sepa dónde está localizado el siguiente. Cuando por fin se encuentra un campo apuntador con algún valor especial predeterminado de fin de lista, se detiene el recorrido. En la figura 5.2 se emplea un -1 en el campo apuntador para señalar el final de la lista.

Cuando una lista está compuesta de registros eliminados que se han convertido en *espacio disponible* dentro del archivo, se llama, por lo común, *lista de disponibles*. Al insertar un registro nuevo en un archivo de registros de longitud fija, cualquier registro disponible es bueno. No hay razón para preferir un espacio libre en lugar de otro, ya que todos son del mismo tamaño; de ahí que no hay razón para ordenar la lista de disponibles en alguna forma en particular. (Como se verá más adelante, esta situación cambia en los registros de longitud variable.)

PILAS. El modo más sencillo de manejar una lista es como si fuera una pila. Una pila es una lista en donde todas las inserciones y eliminaciones de nodos se realizan en uno de los extremos. Así, si se maneja una lista de disponibles en forma de pila compuesta por los números relativos de registro (NRR) 5 y 2, y después se agrega el NRR 3, veríamos, antes y después de la adición del nuevo nodo, lo siguiente:



Cuando se agrega un nodo nuevo al tope o al frente de una pila, se dice que se *mete* en la pila. Si lo que sigue es una solicitud de espacio disponible, se toma el número relativo de registro 3 de la lista de elementos disponibles. A esto se le llama *extracción* de la pila. La lista vuelve al estado anterior, cuando contenía sólo los registros 5 y 2.

LIGAR Y APILAR REGISTROS ELIMINADOS. Ahora se pueden cumplir los dos criterios para acceso rápido al espacio reutilizable que dejaron los registros eliminados. Se necesita:

- Una forma de saber de inmediato si existen espacios vacíos en el archivo, y
- Una forma de ir directamente a uno de esos lugares, si es que existen.

Al colocar los registros eliminados en una pila se cumple con ambos criterios. Si el apuntador que está al tope de la pila contiene el valor de fin de lista, entonces se sabe que no hay ningún lugar vacío en el archivo y se debe insertar los registros nuevos agregándolos al final. Si el apuntador de la pila contiene una referencia de nodo válida, entonces no sólo se sabe que hay un espacio reutilizable, sino también el lugar preciso dónde encontrarlo.

¿Dónde se guarda la pila? ¿Es una lista separada, quizás mantenida en un archivo por separado, o quizás encajada dentro de los datos del archivo? Una vez más es necesario ser cuidadoso para distinguir entre las estructuras *físicas* y las *conceptuales*. Los espacios que dejan disponibles los registros eliminados en realidad no se mueven de lugar cuando se meten en la pila. Están en donde se les necesita, colocados en el archivo. El apilamiento y el ligamiento se realizan organizando y reorganizando las ligas usadas, de modo que un espacio de registro disponible apunte al siguiente. Dado que se trabaja con registros de longitud fija en un archivo de disco, en lugar de direcciones de memoria,

el señalamiento no se realiza con variables tipo *apuntador*, en el sentido formal, sino a través de números relativos de registros (NRR).

Supóngase que se trabaja con un archivo de registros de longitud fija que contenía siete registros (NRR 0–6); que se han eliminado los registros 3 y 5, *en ese orden*, y que los registros eliminados se marcan reemplazando el primer campo con un asterisco. Entonces puede usarse el segundo campo de un registro eliminado para mantener la liga con el siguiente registro en la lista de disponibles. Sin tomar en cuenta los detalles de los registros válidos que se están usando, la figura 5.3(a) muestra cómo se ve el archivo.

El registro número 5 es el primero en la lista de disponibles (tope de la pila), puesto que es el último que se eliminó. Siguiendo la lista ligada, se observa que el registro 5 apunta al registro 3. Como el *campo de liga* del registro 3 contiene –1, que es la marca de fin de lista, se sabe que el registro 3 es el último espacio disponible para reutilizar.

La figura 5.3(b) muestra el mismo archivo después de que se eliminó también el registro número 1. Nótese que el contenido de todos los otros registros de la lista de disponibles permanece sin cambio.

Cabeza de lista (primer registro disponible) → 5

0	1	2	3	4	5	6
Edwards...	Bates...	Wills...	* -1	Masters...	*3	Chavez...

(a)

Cabeza de lista (primer registro disponible) → 1

0	1	2	3	4	5	6
Edwards...	*5	Wills...	* -1	Masters...	*3	Chavez...

(b)

Cabeza de lista (primer registro disponible) → –1

0	1	2	3	4	5	6
Edwards...	Primer registro nuevo	Wills...	Tercer registro nuevo	Masters...	Segundo registro nuevo	Chavez...

(c)

FIGURA 5.3 • Archivo de ejemplo que muestra listas ligadas de registros eliminados. (a) Despues de la eliminación de los registros 3 y 5, en ese orden. (b) Despues de la eliminación de los registros 3, 5 y 1, en ese orden. (c) Despues de la inserción de tres registros nuevos.

Considerar la lista como una pila implica una mínima cantidad de reorganización cuando se meten y sacan registros de ella.

Si ahora se agrega un nombre nuevo al archivo, se colocará en el registro número 1, ya que NRR 1 es el primer espacio disponible. La lista de disponibles regresaría a la configuración que se muestra en la figura 5.3(a). Puesto que aún hay espacio para dos registros en la lista de disponibles, se podrían agregar dos nombres más al archivo sin que se incrementara su tamaño. Sin embargo, después de esto, la lista de disponibles estaría vacía (Fig. 5.3c). Si se agregara otro nombre al archivo, el programa sabría que la lista de disponibles está vacía y que el nombre requiere un registro nuevo al final del archivo.

5.4

REALIZACION DE LA ELIMINACION DE REGISTROS DE LONGITUD FIJA

Los mecanismos para colocar los registros eliminados en una lista ligada de disponibles y para tratarla como una pila se realizan de manera relativamente directa. Se debe considerar algunos detalles importantes, como los siguientes.

- Se requiere un lugar adecuado para el apuntador al primer registro disponible de la lista de disponibles.
- Cuando se elimina un registro, se debe poder marcarlo en alguna forma especial y después colocarlo en la lista de disponibles.
- Cuando se agregan registros, se debe poder reutilizar el espacio de la lista de disponibles, si es que no está vacía.
- Se debe poder verificar si un registro se ha eliminado antes de intentar usarlo.

APUNTADOR AL TOPE DE LA LISTA DE DISPONIBLES. Se comenzará con el primer aspecto: encontrar un lugar para mantener el NRR del primer registro de la lista de disponibles. Como esta información es específica para el archivo de datos, se puede llevar en un registro de encabezado al principio del archivo. Supongamos que hubiera ya un campo en el registro de encabezado llamado ENCAB.PRIMER_DISP,[†] el cual almacena el número relativo de registro del primer disponible. Si este elemento contuviera un -1, significaría que la lista de disponibles está vacía.

[†] Suponemos que el encabezado sería leído dentro de una estructura en C y de un registro en Pascal.

FUNCION: `elim_reg(NRR)`

move el apuntador del archivo a la posición NRR en el archivo
 escribe la bandera de eliminación ('*') en la posición actual del archivo
 escribe ENCAB.PRIMER_DISP en la nueva posición actual del archivo
 asigna a ENCAB.PRIMER_DISP el valor de NRR

fin FUNCION

FIGURA 5.4 • La función `elim_reg(NRR)` elimina el registro número NRR. El espacio abierto por la eliminación se mete a la lista de disponibles.

COLOCAR EL REGISTRO EN LA LISTA DE DISPONIBLES. El siguiente punto en la lista de aspectos importantes concierne a la colocación del registro eliminado en la lista de disponibles. El registro debe marcarse como eliminado y, como miembro de la lista de disponibles, se le debe dar un campo de liga en lugar de los campos usuales de nombre y dirección. Colocar en el registro un '*' (o algún otro carácter especial) como marca de eliminación, es fácil: simplemente se escribe el carácter '*' al principio del registro.

Manejar el campo de liga es un poco más complicado que manejar el '*', porque se tiene que escoger: ¿se debe almacenar la liga en formato ASCII o en binario? Esta decisión es análoga a la que enfrentamos en el capítulo 4, para expresar la longitud del registro en los registros de longitud variable. De nuevo, en C se optaría por un formato binario, pues no requiere traducción y ahorra un poco de espacio; en Pascal sería mejor escribirlo en forma de cadena de caracteres.

La figura 5.4 contiene la descripción en pseudocódigo de una función que elimina registros del archivo de registros de longitud fija .

AGREGAR REGISTROS Y REUTILIZAR EL ESPACIO. Hasta este momento, agregar registros a un archivo siempre significó colocarlos al final. Ahora que se tiene una lista de registros disponibles dentro de un archivo, se puede reutilizar el espacio ocupado previamente por registros eliminados. Lo que se necesita es una función que devuelva

- El NRR de una entrada de un registro reutilizable, o
- El NRR del siguiente registro que se agregará, si es que no hay segmentos reutilizables disponibles.

La función `saca_disp()` (Fig. 5.5) hace esto. La lógica de esta función se esclarece al leer el código y los comentarios.

FUNCION: saca_disp()

```

si ENCAPB.PRIMER_DISP == -1 entonces /* lista de disponibles vacía */
    devuelve el NRR del siguiente registro por agregar

otro /* saca de la lista de disponibles */
    asigna a VAL_DEV el valor de ENCAPB.PRIMER_DISP
    mueve el apuntador del archivo a la posición ENCAPB.PRIMER_DISP
    en el archivo
    ignora el campo '**'
    lee el campo de liga del archivo y lo coloca en NRR
    asigna a ENCAPB.PRIMER_DISP el valor de NRR
    devuelve VAL_DEV

fin FUNCION

```

FIGURA 5.5 • La función *saca_disp()* devuelve el NRR de la primera entrada disponible en el archivo. Si la lista de espacios disponibles está vacía, la función devuelve el NRR del siguiente registro por agregarse al final del archivo.

RECONOCER LOS REGISTROS ELIMINADOS. Como último detalle del desarrollo de este mecanismo hay que asegurarse de que sea posible revisar si un registro se ha eliminado antes de intentar recuperarlo. Ya que el primer carácter en un registro eliminado es un asterisco, es posible revisar la eliminación usando una prueba simple, como ésta:

Si la primera posición en BUFFER es '*' entonces
notificar a quien llamó que el registro está eliminado

5.5

ELIMINACION DE REGISTROS DE LONGITUD VARIABLE

Ahora que se tiene un mecanismo para el manejo de una lista de espacios disponibles cuando se eliminan los registros, se intentará aplicarlo a un problema más complejo, el de la reutilización del espacio de registros de longitud variable. Se ha visto que para reutilizar los registros mediante una lista de disponibles se necesita:

- una forma de ligar los registros eliminados en una lista (p. ej., un lugar para campo de liga);

- un algoritmo para agregar nuevos registros eliminados a la lista de disponibles; y
- un algoritmo para encontrar y eliminar registros de la lista de disponibles cuando estén listos para usarse.

LISTA DE DISPONIBLES DE REGISTROS DE LONGITUD VARIABLE.

BlE. ¿Qué clase de estructura de archivo se necesita para manejar una lista de disponibles de registros de longitud variable? Como se quiere eliminar registros completos y después colocarlos en una lista de disponibles, se necesita una estructura donde el registro sea una entidad definida claramente. Para ello, será de utilidad una estructura de archivo donde se define la longitud de cada registro, colocando un contador de bytes del contenido del registro al inicio de cada uno.

Como esta estructura no considera el número de campos de un registro, se puede manejar el contenido de un registro de longitud variable, como se hizo con los de longitud fija; esto es, se puede colocar un asterisco en el primer campo, seguido por un campo binario de liga que apunte al siguiente registro eliminado en la lista de disponibles. Esta puede organizarse igual que los registros de longitud fija, pero con una pequeña aunque importante diferencia. Sin los registros de longitud fija, no se pueden utilizar los números relativos de registro (NRR) para *ligas*. Como no se puede calcular la distancia en bytes para los registros de longitud variable a partir de sus NRR, son las ligas las que deben contenerla.

Para ilustrar esto, suponga que se empieza con un archivo de registros de longitud variable que contiene los tres registros de Ames, Morrison y Brown que se presentaron con anterioridad. La figura 5.6(a) muestra cómo se ve el archivo (sin el encabezado) antes de cualquier eliminación, y la figura 5.6(b) muestra cómo se ve después de la eliminación del segundo registro. Los puntos en el registro eliminado significan caracteres eliminados.

AGREGAR Y ELIMINAR REGISTROS. Las cuestiones de agregar y eliminar registros de la lista se analizarán juntas porque están muy relacionadas. Con los registros de longitud fija se podía acceder a la lista de disponibles como si fuera una pila, porque todos los miembros de la lista son igualmente utilizables. Esto no es así cuando los segmentos de los registros en la lista de disponibles difieren en tamaño, como sucede en un archivo de registros de longitud variable. En este momento deben cumplirse dos condiciones para reutilizar un registro:

1. El registro debe haber sido eliminado. Esta es la única circunstancia requerida para la reutilización de registros de longitud fija.
2. El registro debe tener el tamaño correcto. Por el momento se define "tamaño correcto" como "suficientemente grande"; y después se

ENCAB.PRIMER_DISP: -1

40 Ames; John; 123 Maple; Stillwater; OK; 74075; 64 Morrison; Sebastian; 9035 South Hillcrest; Forest Village; OK; 74820; 45 Brown; Martha; 62 5 Kimbark; Des Moines; IA; 50311;

(a)

ENCAB.PRIMER_DISP:43

40 Ames; John; 123 Maple; Stillwater; OK; 7407; 64 *; -1
 45 Brown; Martha; 62
 5 Kimbark; Des Moines; IA 50311;

(b)

FIGURA 5.6 • Archivo de ejemplo para ilustrar la eliminación de registros de longitud variable. (a) Archivo original almacenado en formato de longitud variable con el contador de bytes (no se incluye el registro de encabezado). (b) El archivo después de la eliminación del segundo registro (los puntos suspensivos muestran los caracteres eliminados).

verá que algunas veces se requiere ser más explícito acerca del significado de "tamaño correcto".

Se sabe que todos los registros de la lista de disponibles cumplen con la primera condición, pero no todos los miembros cumplen necesariamente con la segunda. Es posible, e incluso común, que se necesite *buscar* en la lista de disponibles un segmento de registro que sea del tamaño correcto. Como se necesita buscar, no se puede organizar la lista de disponibles como una pila.

La incapacidad de usar la extracción de pila como esquema de acceso no significa que no se pueda *organizar* la lista de disponibles como lista ligada, sino que se *accede* en forma diferente. Puesto que está organizada como una lista ligada, encontrar una entrada apropiada significa simplemente recorrer la lista hasta encontrar una que sea lo bastante grande para almacenar el nuevo registro por insertar.

Por ejemplo, suponga que la lista de disponibles contiene las entradas para registros eliminados que muestra la figura 5.7(a), y que se agregará un registro que requiere 55 bytes. Como la lista de disponibles no está vacía, se recorren los registros cuyos tamaños son 47 (demasiado pequeño), 38 (demasiado pequeño) y 72 (suficientemente grande). Al encontrar una entrada lo bastante grande para almacenar el registro, se elimina de la lista de disponibles y se crea una nueva liga que salta el registro (Fig. 5.7b). Si se hubiera llegado al final de la lista

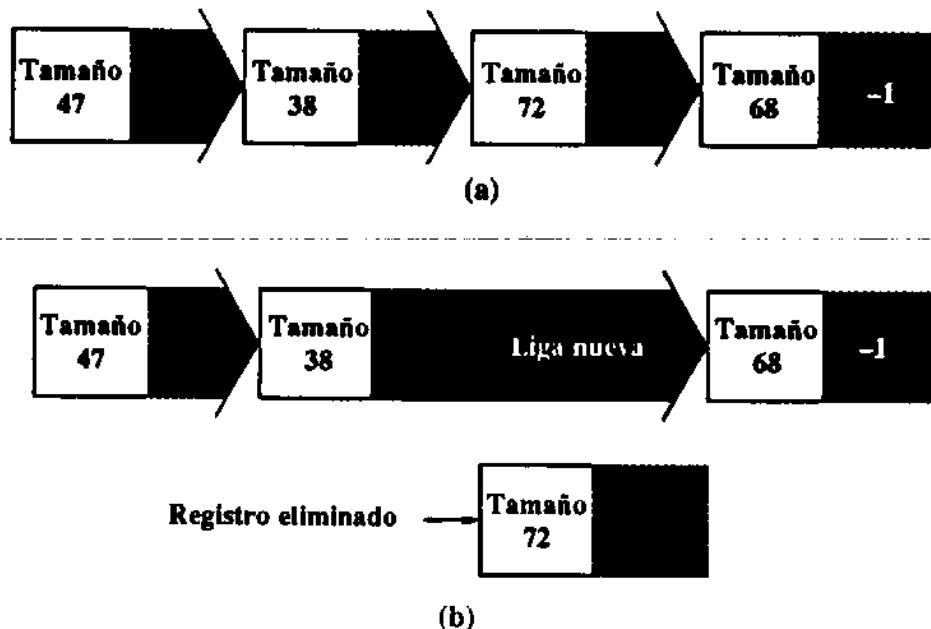


FIGURA 5.7 • Eliminación de un registro de una lista de disponibles con registros de longitud variable: (a) antes de eliminarlo; y (b) después de eliminarlo.

de disponibles sin encontrar un registro suficientemente grande, se habría tenido que agregar el nuevo registro al final del archivo.

La figura 5.8 contiene la descripción de un algoritmo para encontrar una entrada en la lista de disponibles.

FUNCION: toma_disp()

```

encuentra el primer registro de la lista de disponibles

mientras (el registro no sea lo suficientemente grande Y no se
llegue al fin de la lista)
salta al siguiente espacio disponible

si el registro es lo suficientemente grande
reacomoda la lista ligada para eliminar el registro
devuelve la distancia en bytes de la entrada

otro /* se alcanzó el final de la lista antes de encontrar
una entrada suficientemente grande */ 
devuelve la distancia en bytes del final del archivo
  
```

fin FUNCION

FIGURA 5.8 • Función para tomar una entrada de la lista de disponibles para la inserción de registros de longitud variable.

FUNCION: `elimina_reg(NRR)`

```
lee en forma secuencial el archivo hasta encontrar el registro  
NRR  
  
asigna a POS_BYTE la distancia en bytes del registro  
que va a ser eliminado  
coloca la marca de registro eliminado ("*") en el primer campo  
coloca el valor de ENCAB.PRIMER_DISP en el siguiente campo  
como liga  
asigna a ENCAB.PRIMER_DISP el valor de POS_BYTE  
  
fin FUNCION
```

FIGURA 5.9 • Procedimiento para colocar los registros eliminados en la lista de disponibles.

Como este procedimiento para encontrar un registro reutilizable revisa la lista completa de disponibles, si es necesario, no se requiere un método complicado para colocar los registros recién eliminados en la lista. Si existe un registro del tamaño correcto en algún lugar de la lista, el procedimiento para obtención de un registro disponible lo encontrará finalmente. De ahí que se puede continuar introduciendo nuevos miembros al frente de la lista, como se hace con los registros de longitud fija. La figura 5.9 muestra el esbozo de un procedimiento que elimina un registro dado su número relativo de registro (NRR), y que después lo introduce en la lista de disponibles.

5.6

FRAGMENTACION DEL ALMACENAMIENTO

Se examinará de nuevo la versión con registros de longitud fija del archivo de tres registros (Fig. 5.10). Los puntos al final de los registros representan caracteres que se usan como relleno entre el último campo

Ames ; John ; 123 Maple ; Stillwater ; OK ; 74075;.....
Morriso ; Sebastian;9035 South Hillcrest;Forest Village;OK;74820;
Brown ; Martha ; 625 Kimbark ; Des Moines ; IA ; 50311;.....

FIGURA 5.10 • Requerimientos de almacenamiento del archivo de ejemplo empleando registros de longitud fija de 64 bytes.

40 Ames ; John ; 123 Maple ; Stillwater ; OK ; 74075 ; 64 Morrison ; Sebastian
; 9035 South Hillcrest ; Forest Village ; OK ; 74820 ; 45 Brown ; Martha ; 62
5 Kimbark ; Des Moines ; IA ; 50311 ;

FIGURA 5.11• Requerimientos de almacenamiento del archivo de ejemplo empleando registros de longitud variable con campos de conteo.

y el final de los registros. El relleno es espacio desperdiciado; es parte del costo que implica usar registros de longitud fija. El espacio desperdiciado *dentro* de un registro se llama *fragmentación interna*.

Por supuesto, es deseable minimizar la fragmentación interna. Cuando se trabaja con registros de longitud fija, se procura dicha minimización eligiendo una longitud que sea lo más aproximada posible a la que se necesita para cada registro. Pero a menos que los datos reales sean de una longitud fija, debe admitirse una cierta cantidad de fragmentación interna en un archivo con registros de longitud fija.

Uno de los atractivos de los registros de longitud variable es que minimizan el espacio desperdiciado al eliminar la fragmentación interna cuando se escribe el archivo por primera vez. El espacio definido para cada registro es exactamente del tamaño necesario. Compárese el ejemplo de longitud fija con el de la figura 5.11, que usa la estructura de registros de longitud variable: un contador de bytes seguido por campos delimitados. El único espacio (además del de los delimitadores) que no se usa para guardar datos en cada registro es el campo de conteo. Si se supone que este campo usa dos bytes, esto asciende a sólo seis bytes para el archivo de tres registros. El archivo de registros de longitud fija desperdicia 24 bytes tan sólo en el primer registro.

Pero antes de felicitarnos por resolver el problema del espacio desperdiciado por la fragmentación interna, debe considerarse lo que sucede en un archivo de registros de longitud variable después de que se elimina un registro y se reemplaza por otro más corto. Si el registro más corto ocupa menos espacio que el original, ocurre fragmentación interna. La figura 5.12 muestra cómo ocurriría el problema en el archivo del ejemplo cuando se elimina el segundo registro del archivo y se agrega el siguiente:

Ham ; Al ; 28 Elm ; Ada ; OK ; 70332 ;

Al parecer, escapar de la fragmentación interna no es tan fácil. El espacio liberado por el registro eliminado es 37 bytes más largo que lo necesario para el registro nuevo. Como los 37 bytes adicionales se consideran parte del registro nuevo, no están en la lista de disponibles y, por lo tanto, no son usables. ¿Qué sucedería si en lugar de guardar

ENCAB.PRIMER_DISP:43 _____
 ↓
 40 Ames | John | 123 Maple | Stillwater | OK | 74075 | 64 * | -1
 45 Brown | Martha | 62
 5 Kimbark | Des Moines | IA 50311 |

(a)

ENCAB.PRIMER_DISP:-1
 40 Ames | John | 123 Maple | Stillwater | OK | 74075 | 64 Ham | Al | 28 Elm | Ada | OK
 | 70332 |..... 45 Brown | Martha | 625 Kimbark | Des
 Moines | IA 50311 |

(b)

FIGURA 5.12• Ilustración de la fragmentación con registros de longitud variable. (a) Despues de la eliminación del segundo registro (los caracteres que no se usan en el registro eliminado se reemplazan con puntos). (b) Despues de la subsecuente adición del registro para Al Ham.

intacto el espacio de 64 bytes, se dividiera en dos partes: una para guardar el nuevo registro Ham, y la otra para colocar de nuevo en la lista de disponibles? Como sólo ocuparía el espacio necesario para el registro Ham, no habría fragmentación interna.

La figura 5.13 muestra cómo se ve el archivo cuando se usa este enfoque para insertar el registro para Al Ham. Se toma el espacio del registro Ham *del final* del espacio de 64 bytes y se dejan los primeros 35 bytes del espacio en la lista de disponibles. (El espacio disponible es 35, en lugar de 37, porque se necesitan dos bytes para formar un nuevo campo de tamaño para el registro Ham.)

Los 35 bytes que aún están en la lista de disponibles pueden usarse para almacenar incluso otro registro. La figura 5.14 muestra el efecto de la inserción del siguiente registro de 25 bytes:

Lee | Ed | Rt 2 | Ada | OK | 74820 |

Como se esperaría, el nuevo registro se extrajo del de 35 bytes que está en la lista de disponibles. La porción de datos del registro nuevo

ENCAB.PRIMER_DISP:43 _____
 ↓
 40 Ames | John | 123 Maple | Stillwater | OK | 74075 | 35 * | -1
 26 Ham | Al | 28 Elm | Ada | OK | 70332 | 45 Brown | Martha | 625
 Kimbark | Des Moines | IA | 50311 |

FIGURA 5.13• La fragmentación interna se combate devolviendo a la lista de disponibles la parte sin usar de las entradas eliminadas.

ENCAB.PRIMER_DISP:43

40 Ames ; John ; 123 Maple ; Stillwater ; OK ; 74075 ; 8 * ; -1... 25 Lee ; Ed ; Rt
2 ; Ada ; OK ; 74820 ; 26 Ham ; Al ; 28 Elm ; Ada ; OK ; 70332 ; 45 Brown ; Martha ; 625
Kimbark ; Des Moines ; IA ; 50311 ;

FIGURA 5.14- Adición del segundo registro en la entrada que originalmente ocupaba un registro eliminado.

requiere 25 bytes, y son necesarios dos bytes más para otro campo de tamaño; esto deja todavía ocho bytes del registro en la lista de disponibles.

¿Cuál es la probabilidad de encontrar un registro que pueda usar estos ocho bytes? Suponemos que es casi cero. Estos ocho bytes no son utilizables, aunque no estén atrapados en otro registro. Este es un ejemplo de *fragmentación externa*. El espacio está en realidad en la lista de disponibles, y no encerrado en algún otro registro, pero está demasiado fragmentado como para reutilizarse.

Existen algunas formas interesantes de combatir la fragmentación externa. Una de ellas, que se analiza al principio de este capítulo, es la *compactación del almacenamiento*. Cuando se vuelve intolerable la fragmentación externa se puede simplemente regenerar el archivo. Otros dos enfoques son:

- Cuando es posible, se combinan fragmentos pequeños para crear fragmentos más grandes y útiles. Si el programa advierte que dos entradas en la lista de disponibles son adyacentes, puede combinarlas para hacer una entrada mayor. Esto se llama *unir los huecos* en el espacio de almacenamiento.
- Intentar minimizar la fragmentación antes de que suceda, adoptando una estrategia de colocación que el programa pueda usar cuando selecciona una entrada de la lista de disponibles.

Unir los huecos presenta algunos problemas interesantes. La lista de disponibles no se mantiene en un orden *físico* de registros; si hay dos registros eliminados físicamente adyacentes, no hay razón para pensar que se ligarán en forma adyacente en la lista de disponibles. El ejercicio 15, al final del capítulo, proporciona un análisis del problema, junto con ideas para desarrollar una solución.

Sin embargo, el desarrollo de mejores *estrategias de colocación* es una cuestión distinta. Es un tema que justifica un análisis por separado, ya que la elección entre estrategias alternativas no es tan obvia como parece a primera vista.

5.7

ESTRATEGIAS DE COLOCACION

Anteriormente se presentaron dos funciones para la eliminación de registros de longitud variable, *elimina_registro()* y *toma_disp()*. La función *elimina_registro()* maneja la lista de disponibles como una pila, colocando los registros recién eliminados al frente. Puesto que los registros se colocan en la lista de disponibles según un orden donde el recién eliminado es el primero, no están ordenados por tamaño. Cuando se necesita una entrada para guardar un registro nuevo, *toma_disp()* busca desde el inicio de la lista hasta que encuentra una entrada lo suficientemente grande para guardarlo o bien hasta que llega al final.

Esta estrategia de colocación se llama de *primer ajuste*. Se utiliza la mínima cantidad posible de trabajo cuando se coloca un nuevo espacio disponible en la lista, y no se preocupa mucho por la exactitud del ajuste mientras se busca una entrada para almacenar un registro nuevo. Se acepta la primera entrada disponible que funcione, sin importar si el segmento es diez veces más grande de lo que se necesita, o si ajusta a la perfección.

Por supuesto, se podría desarrollar un enfoque más ordenado para la colocación de registros en la lista de disponibles, manteniéndolos en secuencia, ascendente o descendente, por tamaño. En lugar de colocar siempre el nuevo registro eliminado al frente de la lista, estos enfoques implican recorrer la lista buscando un lugar donde insertar el registro para mantener la secuencia deseada.

¿Qué efecto tendría clasificar la lista de disponibles en orden ascendente, por tamaño, en la precisión del ajuste de los registros que *toma_disp()* extraería de la lista? Como *toma_disp()* busca en la lista de disponibles en forma secuencial hasta encontrar un registro suficientemente grande para almacenar el nuevo, el primer registro encontrado por *toma_disp()* sería el *más pequeño* que funcionaría. El ajuste entre la entrada disponible y las necesidades del registro nuevo sería lo más preciso posible. A esto se le llama estrategia de colocación del *mejor ajuste*.

La estrategia del mejor ajuste resulta atractiva de inmediato, pero por supuesto, tiene un precio: al final siempre es necesario buscar, al menos, a través de una parte de la lista, no sólo cuando se toman registros de ella, sino también cuando se colocan registros recién eliminados. En un ambiente de tiempo real, el tiempo adicional de procesamiento podría ser significativo.

Una desventaja menos obvia de la estrategia del mejor ajuste está relacionada con la idea misma de encontrar el mejor ajuste posible: el área libre que queda después de insertar un nuevo registro en una

entrada es lo más pequeña posible. A menudo este espacio restante es demasiado pequeño para ser de utilidad, lo que ocasiona fragmentación externa. Además, los segmentos que tienen menos probabilidad de ser útiles son aquellos que estarán colocados al principio de la lista, lo que significa que las búsquedas de mejor ajuste se incrementan conforme pasa el tiempo.

Estos problemas sugieren una estrategia alternativa: ¿Qué sucede si se acomoda la lista de disponibles en orden *descendente* por tamaño? Entonces la entrada más grande de la lista de disponibles siempre estaría al principio. Puesto que *toma_disp()* inicia su búsqueda al principio de la lista de disponibles, siempre devuelve la entrada disponible más grande, si es que devuelve alguna. Esto se conoce como estrategia de colocación del *peor ajuste*. La cantidad de espacio de la entrada por encima de lo que en realidad se necesita es lo más grande posible.

La estrategia del *peor ajuste*, a primera vista, no parece muy atractiva, pero considérese que:

- El procedimiento *toma_disp()* se puede simplificar de modo que busque sólo en el primer elemento de la lista de disponibles. Si la primera entrada no es lo suficientemente grande para funcionar, ninguna de las otras lo será.
- Al extraer el espacio necesario de la entrada *más grande* disponible, se asegura que la porción que no se usa sea lo más grande posible. Esto es importante, porque la fragmentación externa empieza a desarrollarse cuando el espacio no usado en el archivo está compuesto por piezas demasiado pequeñas para ser de utilidad.

¿Qué es lo que se puede concluir de todo esto? Debe quedar claro que ninguna estrategia de colocación es superior en todas las situaciones. Se debe formular una serie de observaciones generales y después, dada una situación de diseño en particular, seleccionar la estrategia que parezca más apropiada. A continuación se presentan algunas observaciones generales: la decisión es del lector.

- Las estrategias de colocación tienen sentido sólo para archivos de registros de longitud variable. Con registros de longitud fija, la colocación no importa.
- Si el uso del espacio debido a la eliminación y reutilización de registros no es un factor importante, porque hay bastante espacio o porque hay muy poca eliminación, probablemente el primer ajuste es la estrategia apropiada, ya que requiere el mínimo de tiempo.
- Si se pierde espacio debido a *fragmentación interna*, entonces la

decisión está entre el primer ajuste y el mejor ajuste. Una estrategia del peor ajuste empeora efectivamente la fragmentación interna.

- Si se pierde espacio debido a *fragmentación externa* y no se utiliza el primer ajuste, entonces se debe considerar cuidadosamente la estrategia del peor ajuste, la cual requiere menos tiempo que el mejor ajuste y puede hacer que el grado de fragmentación externa tienda a decrecer.

RESUMEN

La organización original de un archivo influye en cómo puede ser alterado, y también cómo puede responder al deterioro. Un archivo volátil, es decir, que esté sometido a muchos cambios, puede deteriorarse muy rápido, a menos que se tomen medidas para ajustar su organización a los cambios. Una de las consecuencias de los cambios en los archivos es la fragmentación del almacenamiento.

La *fragmentación interna* ocurre cuando existe espacio desperdigado dentro de un registro. En un archivo de registros de longitud fija, la fragmentación interna puede ocurrir cuando se almacenan registros de longitud variable en entradas de tamaño fijo. También puede ocurrir en un archivo de registros de longitud variable, cuando un registro se reemplaza por otro de menor tamaño. La *fragmentación externa* ocurre cuando se crean espacios sin utilizar entre registros, normalmente debido a eliminaciones.

Existen varias formas de combatir la fragmentación. La más sencilla es la *compactación del almacenamiento*, la cual elimina el espacio no utilizado que ocasionó la fragmentación externa, agrupando todos los registros no eliminados. Por lo general, la compactación se realiza mediante procesamiento por lotes.

La fragmentación puede tratarse *dinámicamente* reutilizando el espacio eliminado cuando se agregan registros. Debido a la necesidad de mantener información respecto al espacio que puede reutilizarse, este enfoque resulta más complejo que la compactación. Además, si un archivo tiene registros de longitud variable, no basta mantener una lista de entradas o posiciones de registros disponibles. Se necesita también desarrollar una *estrategia de colocación* para elegir un espacio del tamaño correcto para el registro nuevo. La mayor parte del capítulo está dedicada al desarrollo de técnicas y estrategias para la reutilización dinámica del espacio obtenido por eliminación de registros.

Se empieza con el problema de eliminación de registros de longitud fija. Puesto que es muy fácil encontrar el primer campo de un registro de longitud fija, la eliminación de un registro puede llevarse a cabo colocando una marca especial en el primer campo. Se usa un asterisco ("*") para marcar que un registro ha sido eliminado.

Como todos los registros de un archivo de registros de longitud fija son del mismo tamaño, no es difícil reutilizar los registros eliminados. La solución consiste en agrupar todos los espacios disponibles en una *lista de disponibles*, que se crea reuniendo todos los registros eliminados en forma de *lista ligada*. Ésta se forma acomodando *campos de liga* en los espacios de registros eliminados (justo después del campo "*") de tal forma que cada registro de la lista contenga un campo que proporcione el número relativo de registro del que sea el siguiente en la lista.

En un archivo de registros de longitud fija, todas las entradas son igualmente útiles; son intercambiables. Por consiguiente, la forma más simple de mantener la lista ligada de disponibles es tratarla como una *pila*. Los nuevos espacios disponibles se agregan a la lista de disponibles *metiéndolos* al frente de la lista; las entradas se eliminan de la lista de disponibles *sacándolas* del frente de la lista.

Después de analizar los detalles de realización asociados con la eliminación y reutilización de registros de las estructuras de archivos desarrolladas en el capítulo 4, se considera la cuestión un tanto más difícil de la eliminación de registros de *longitud variable*. Se encuentra que se pueden colocar campos de "*" y ligar los campos en los registros de longitud variable tal como se hace en los de longitud fija, aunque existe una diferencia importante. Con los registros de longitud fija no es necesario asegurarse de que una entrada sea del tamaño correcto para almacenar el nuevo registro. La definición inicial de tamaño correcto se refiere simplemente a que debe ser lo suficientemente grande. Por lo tanto, se desarrolla una función, *toma_disp()*, para buscar en la lista de disponibles hasta encontrar una entrada lo bastante grande para almacenar el nuevo registro. Dadas dicha función y una función complementaria, que coloca los registros recién eliminados al frente de la lista de disponibles, puede desarrollarse un sistema para eliminar y reutilizar registros de longitud variable.

Posteriormente se considera la cantidad y naturaleza de la fragmentación que se desarrolla dentro del archivo, debida a la eliminación y reutilización de registros. La fragmentación puede ocurrir *internamente* cuando el espacio se pierde por quedar encerrado dentro de un registro. Se desarrolla un procedimiento que divide una entrada grande de longitud variable en dos o más, menores, usando exactamente el espacio necesario para un registro nuevo, y dejando el resto en la lista de disponibles. Se observa que aunque esto puede reducir la cantidad de espacio desperdiciado, a la larga los fragmentos restantes son

demasiado pequeños para ser de utilidad. Cuando esto sucede, el espacio se pierde por *fragmentación externa*.

Existen varios caminos para minimizar la fragmentación externa; entre otros:

- *Compactar* el archivo mediante procesamiento por lotes, cuando el nivel de fragmentación se vuelve excesivo ;
- *Unir* las entradas adyacentes (huecos) en la lista de disponibles para formar segmentos más grandes y más útiles en general, y
- Adoptar una *estrategia de colocación* para seleccionar las entradas que se reutilizarán, de modo que se minimice la fragmentación.

El desarrollo de algoritmos de unión de huecos forma parte de los ejercicios que se encuentran al final del capítulo. Por otro lado, las estrategias de colocación ameritan un análisis más cuidadoso.

La estrategia de colocación usada hasta ahora por los procedimientos de eliminación y reutilización de registros de longitud variable es la de *primer ajuste*. Esta estrategia es sencilla: "si la entrada es lo suficientemente grande, úsese". Si se mantiene la lista de disponibles clasificada, resulta fácil usar cualquiera de las otras dos estrategias de colocación:

- *Mejor ajuste*: un registro nuevo se coloca en la entrada más pequeña que sea lo suficientemente grande para almacenarlo. Esta es una estrategia atractiva para archivos con registros de longitud variable donde la fragmentación es interna, pero implica más gasto que otras estrategias de colocación.
- *Peor ajuste*: un registro nuevo se coloca en la entrada más grande disponible. La idea es que la porción sobrante de la entrada sea lo más grande posible.

No hay una regla para seleccionar la estrategia de colocación; lo mejor es informarse y considerar las sugerencias antes de decidir.

TERMINOS CLAVE

Compactación. Forma de liberarse por completo de la *fragmentación externa*, recorriendo todos los registros para juntarlos de modo que no existan segmentos desperdiciados entre ellos.

Estrategia de colocación. En este capítulo, las estrategias de colocación se definieron como un mecanismo de selección del espacio en la lista de disponibles, a fin de usarlo para almacenar un nuevo registro que se agregue al archivo.

Fragmentación. El espacio no utilizado dentro de un archivo. El espacio puede estar encerrado en los registros individuales (*fragmentación interna*), o bien fuera o entre ellos (*fragmentación externa*).

Fragmentación externa. Tipo de fragmentación que ocurre en un archivo cuando el espacio que no se usa está fuera del archivo o entre registros individuales.

Fragmentación interna. Forma de fragmentación que ocurre cuando se desperdicia el espacio en un archivo porque está encerrado, sin utilizar, dentro de los registros. A menudo las estructuras de registros de longitud fija producen fragmentación interna.

Lista de disponibles. Lista de los espacios liberados al eliminar registros, los cuales quedan disponibles para almacenar registros nuevos. En los ejemplos estudiados en este capítulo, esta lista de espacios disponibles tomó la forma de una lista ligada de registros eliminados.

Lista ligada. Conjunto de nodos organizados con una secuencia específica por medio de referencias colocadas en cada nodo, que apuntan a un nodo sucesor único. Con frecuencia el orden *lógico* de una lista ligada es diferente del orden físico de los nodos en la memoria del computador.

Mejor ajuste. Estrategia de colocación para seleccionar el espacio de la lista de disponibles y almacenar un registro nuevo. La colocación de mejor ajuste encuentra la entrada disponible cuyo tamaño se aproxime más al que se necesita para almacenar el nuevo registro.

Peor ajuste. Tipo de estrategia de colocación para seleccionar un espacio de la lista de disponibles. La estrategia de colocación del peor ajuste selecciona la entrada más grande, sin importar cuán pequeño sea el nuevo registro. Como esto deja la mayor entrada posible para reutilización, en algunos casos, el peor ajuste ayuda a minimizar la *fragmentación externa*.

Pila. Tipo de lista donde todas las inserciones y eliminaciones tienen lugar en el mismo extremo.

Primer ajuste. Estrategia de colocación para la selección de un espacio de la lista de disponibles. La colocación de primer ajuste selecciona la primera entrada disponible que sea lo suficientemente grande para almacenar el nuevo registro.

Unión de huecos. Si dos registros eliminados dejan espacios

disponibles que son físicamente adyacentes, pueden juntarse para formar un solo espacio mayor. Este proceso de combinación de espacios disponibles pequeños para formar uno mayor se conoce como *unión de huecos*. Esta es una manera de contrarrestar el problema de la fragmentación externa.

EJERCICIOS

1. ¿Cuál es la diferencia entre la fragmentación interna y la externa? ¿Cómo afecta la compactación a la cantidad de fragmentación interna en un archivo? ¿Qué pasa con la fragmentación externa?
2. La compactación en el mismo sitio quita los registros eliminados de un archivo sin crear uno nuevo por separado. ¿Cuáles son las ventajas y desventajas de la compactación en el mismo sitio, comparada con la compactación en la que se crea un archivo compacto separado?
3. ¿Por qué la estrategia del peor ajuste es una mala elección cuando hay una pérdida significativa de espacio ocasionada por fragmentación interna?
4. Imagine una forma económica de llevar un registro continuo de la cantidad de fragmentación que hay en un archivo. Esta medida de fragmentación se usaría para activar el procesamiento por lotes empleado para reducir la fragmentación.
5. Suponga que un archivo debe permanecer clasificado. ¿Cómo afecta esto en la gama de estrategias de colocación disponibles?
6. Desarrolle una descripción en pseudocódigo de un procedimiento que realice compactación en el mismo sitio, en un archivo de registros de longitud variable que contenga campos de tamaño al inicio de cada registro.
7. Considere el proceso de actualización en lugar del de eliminación de un registro de longitud variable. Plantee un procedimiento para manejar dicha actualización, tomando en cuenta que de la actualización puede resultar un registro más largo o más corto.
8. En la sección 5.4 se pregunta dónde hay que mantener la pila que contiene la lista de registros disponibles. ¿Debe ser una lista separada, que quizás se mantiene en un archivo separado, o debe estar inmersa

dentro del archivo de datos? En la realización se eligió la última organización. ¿Qué ventajas y desventajas tiene el segundo enfoque? ¿Cuáles otros tipos de estructuras de archivos pueden facilitar las varias clases de eliminación de registros?

9. En algunos archivos, cada registro tiene un bit de eliminación que se pone en 1 cuando se elimina el registro. Este bit también se puede utilizar para indicar que un registro está inactivo y no eliminado. ¿Qué se necesita para reactivar un registro inactivo? ¿Podría hacerse la reactivación con los procedimientos de eliminación que se han usado?

10. En este capítulo se esbozaron tres enfoques generales de minimización de la fragmentación del almacenamiento:

- desarrollo de una estrategia de almacenamiento;
- unión de huecos, y
- compactación.

Suponiendo un ambiente de programación interactivo, ¿cuál de estas estrategias podría aplicarse al tiempo que se agregan y eliminan registros? ¿Qué estrategias podrían aplicarse en forma de procesamiento por lotes que pudiesen ejecutarse periódicamente?

11. ¿Por qué las estrategias de colocación tienen sentido sólo con archivos de registros de longitud variable?

EJERCICIOS DE PROGRAMACION

12. Reescriba el programa *actualiza.c* o *actualiza.pas* de modo que pueda eliminar y agregar registros a un archivo de registros de longitud fija, usando uno de los procedimientos de reemplazo analizados en este capítulo

13. Escriba un programa similar al descrito en el ejercicio anterior, pero que trabaje con archivos de registros de longitud variable.

14. En la sección 5.6 se muestra que la función *toma_disp()*, empleada para reutilizar el espacio de registros de longitud variable eliminados, puede provocar gran cantidad de fragmentación interna, y se sugiere un enfoque alterno que permite que las entradas se dividan para formar registros nuevos. Modifique *toma_disp()* de manera que realice esta división cuando la entrada original sea lo suficientemente grande para permitirlo.

15. Desarrolle en pseudocódigo la descripción de un procedimiento para eliminar registros de longitud variable, con el fin de revisar si el registro recién eliminado es contiguo a algún otro registro eliminado. Si existe contigüidad, una los registros para obtener una entrada disponible mayor. Algunos aspectos que deben considerarse al abordar este problema son los siguientes:

- La lista de disponibles no mantiene los registros acomodados en orden físico; el siguiente registro en la lista de disponibles no necesariamente es el siguiente registro físico eliminado en el archivo. ¿Es posible combinar estos dos puntos de vista sobre la lista de disponibles, el orden físico y el orden lógico, dentro de una sola lista? Si se puede, ¿qué estrategia de colocación se aplicaría?
- La adyacencia física puede incluir tanto registros que preceden como registros que siguen al registro recién eliminado. ¿Cómo buscaría un registro eliminado que preceda al registro recién eliminado?
- Mantener dos puntos de vista respecto a la lista de registros eliminados implica que, conforme se descubran registros físicamente adyacentes, tienen que reacomodarse las ligas para actualizar la lista, no física, de disponibles. ¿Qué complicaciones adicionales se presentarían si se combinan la unión de huecos y una estrategia de mejor o de peor ajuste?

LECTURAS ADICIONALES

No deja de ser sorprendente que, en la literatura sobre fragmentación del almacenamiento y reutilización, no suelen considerarse estos aspectos desde el punto de vista del almacenamiento secundario. Normalmente, la fragmentación del almacenamiento, las estrategias de colocación, la unión de huecos y la recolección de basura se tratan en el contexto de reutilización de espacio dentro de la memoria electrónica de acceso aleatorio (RAM). Conforme se lee, con la idea de aplicar los conceptos al almacenamiento secundario, se hace necesario evaluar cada estrategia en términos del costo del acceso al almacenamiento secundario. Algunas estrategias que son atractivas cuando se usan en memoria RAM son demasiado costosas en el almacenamiento secundario.

A menudo se encuentran análisis acerca del manejo del espacio en memoria RAM bajo el título de "asignación dinámica de almacenamiento". Knuth

[1973a] proporciona un buen repaso, aunque técnico, de los aspectos fundamentales asociados con la asignación dinámica del almacenamiento, incluyendo estrategias de colocación. Tremblay y Sorenson [1984] reelaboraron gran parte del trabajo de Knuth, haciéndolo más accesible. Standish [1980] proporciona un panorama más completo del tema gracias a la revisión de muchas obras importantes sobre la materia.

6

OBJETIVOS

Presentar la idea que fundamenta la búsqueda binaria.

Desarrollar un procedimiento de ordenamiento basado en la clasificación Shell en memoria interna.

Introducir, mediante un ejemplo, el concepto de *indirección* y demostrar el poder de esta herramienta conceptual.

Examinar las limitaciones de la búsqueda binaria.

Desarrollar un procedimiento de *clasificación por llave* para ordenamiento de archivos grandes; investigar los costos asociados con la clasificación por llave.

Introducir el concepto de *registro fijo*.

LOCALIZACION RAPIDA DE DATOS EN UN ARCHIVO: INTRODUCCION A LA CLASIFICACION Y A LA BUSQUEDA BINARIA

PLAN GENERAL DEL CAPITULO

Este libro comienza con un análisis sobre el costo del acceso al almacenamiento secundario. Se recordará que el grado de diferencia entre el acceso a memoria RAM y la búsqueda de información en un disco fijo es tal que, si se amplificaran los tiempos, un acceso a memoria RAM representaría 20 segundos, y un acceso a disco representaría 116 días.

Hasta aquí no se ha prestado mucha atención a este costo. Este capítulo marca, pues, un cambio de enfoque. Tan pronto como uno se mueve de los aspectos organizacionales fundamentales al tema de la búsqueda de información particular dentro de un archivo, el costo de los desplazamientos se convierte en un factor importante para la determinación del enfoque. Lo que es cierto para la búsqueda lo es más para la clasificación. Si el lector ha estudiado algoritmos de clasificación, sabe que aun una buena clasificación implica muchas comparaciones. Si cada una de esas comparaciones implica un desplazamiento, la clasificación es terriblemente lenta.

Por tanto, el análisis de la clasificación y la búsqueda va más allá del simple trabajo de hacerlo. Se desarrollan enfoques que minimicen el número de accesos a disco y que minimicen también la cantidad de tiempo invertido. Esta inquietud de minimizar el número de despla-

zamientos sigue siendo uno de los aspectos más importantes del resto del libro. Este capítulo sólo es el principio de una investigación sobre las formas de clasificar y encontrar la información rápidamente.

6.1

LOCALIZACION DE DATOS EN LOS ARCHIVOS YA DESARROLLADOS

Se han realizado ya algunas estructuras de archivos que permiten al usuario hacer procesos un tanto complejos. Pero todos los programas que se han presentado en este libro adolecen, a pesar de cualesquiera otras ventajas que proporcionen, de una gran falla: la única forma de extraer o encontrar un registro con algún grado de rapidez es buscar por su número relativo de registro (NRR). Si el archivo tiene registros de longitud fija, conocer el NRR permite saltar al registro usando *acceso directo*.

Pero ¿qué pasa si no se conoce el NRR del registro que se desea? Los ejemplos de los capítulos anteriores se asocian con un simple archivo de nombres y direcciones. ¿Qué tan probable es que una pregunta acerca de este archivo sea de la forma, "¿cuál es la dirección almacenada en el NRR 23?" Por supuesto, no muy probable. Es mucho más factible que se conozca la identidad de un registro por su llave.

Dados los métodos de organización desarrollados hasta aquí, el acceso por llave implica una búsqueda secuencial, examinando un registro tras otro hasta que se encuentra el que contiene la llave. ¿Qué pasa si no existe ningún registro que contenga la llave solicitada? Entonces tendría que revisarse todo el archivo. ¿Qué pasa si se sospecha que puede haber más de un registro que contenga la llave y se pretende encontrarlos todos? De nuevo se estaría obligado a examinar en todos los registros del archivo. Está claro que se necesita encontrar una mejor forma de manejar los accesos mediante llave. Por fortuna, hay muchas formas apropiadas.

Se comenzará la investigación de organizaciones alternas considerando las formas de encontrar información mediante conjeturas.

6.2

BUSQUEDA POR CONJETURA: BUSQUEDA BINARIA

Supongamos que se busca el registro de Bill Kelly en un archivo de 1000 registros de longitud fija, y que el archivo está clasificado de tal forma que los registros aparecen en orden ascendente por llave. Los números

relativos de registro del archivo van de 0 a 999. Se empieza comparando KELLY BILL (la forma canónica de la llave de búsqueda) con la llave que está a la mitad del archivo, cuyo NRR es 500. El resultado de la comparación, indica cuál mitad del archivo contiene el registro de Bill Kelly. Por ejemplo, si se encuentra que el registro 500 es de Sam Miller, se sabe que el resto de la búsqueda puede concentrarse en la primera mitad del archivo, en los registros que van desde 0 hasta 499. En seguida se compara KELLY BILL con la llave de la mitad, entre los registros 0 y 499, para saber en cuál cuarto del archivo está el registro de Bill Kelly. Este proceso se repite hasta que se encuentre el registro de Bill Kelly o se haya reducido el número de registros potenciales a cero.

Esta clase de búsqueda, donde la mitad de los objetos de informa-

```
/* Función para efectuar una búsqueda binaria en el archivo asociado con el
nombre lógico ENTRADA. Supone que ENTRADA contiene CONT_REG registros.
Busca la llave LLAVE. Devuelve el NRR del registro que contiene la llave,
si se encuentra; de lo contrario devuelve -1
```

```
*/
```

```
FUNCION: búsqueda_bin (ENTRADA, LLAVE, CONT_REG)
```

```
MENOR : = 0           /* Asigna el límite inferior de la búsqueda */
MAYOR : = CONT_REG -1 /* Asigna el límite superior restando 1 al
                      contador, porque los NRR inician en 0 */

mientras (MENOR <= MAYOR)
    INTENTO : = (MAYOR + MENOR) / 2      /* encuentra el punto
                                             medio */
```

lee el registro con NRR de INTENTO

coloca la forma canónica de la llave del registro

INTENTO en LLAVE_ENC

```
si (LLAVE < LLAVE_ENC)          /* INTENTO es muy alto, */
    MAYOR : = INTENTO - 1        /* así que se reduce el límite
                                  superior */
```

otro si (LLAVE > LLAVE_ENC) /* INTENTO es muy bajo, */
 MENOR : = INTENTO + 1 /* se aumenta el límite inferior */

otro

devuelve (INTENTO) /* Son iguales; se devuelve el NRR */

fin mientras

devuelve (-1) /* Si el ciclo termina, entonces no se encontró la */
fin FUNCION llave

FIGURA 6.1.0 La función *busca_bin()* en pseudocódigo.

ción restantes se eliminan con cada comparación, se llama *búsqueda binaria*. En la figura 6.1 se muestra un algoritmo de búsqueda binaria. La búsqueda binaria realiza *a lo sumo* diez comparaciones para encontrar el registro de Bill Kelly, si es que están en el archivo, o bien para determinar que no está en él. Compárese esto con la búsqueda secuencial de un registro: si existen 1000 registros, entonces realiza *a lo sumo* 1000 comparaciones para encontrar uno dado (o establecer que no está presente); en promedio, se necesitan 500 comparaciones.

En general, una búsqueda binaria en un archivo con n registros realiza *a lo sumo*

$$\lfloor \log n \rfloor + 1 \text{ comparaciones}^{\dagger}$$

y en promedio

$$\lfloor \log n \rfloor + 1/2 \text{ comparaciones.}$$

Por lo tanto, se dice que una búsqueda binaria es $O(\log n)$. En contraste, se recordará que una búsqueda secuencial para el mismo archivo requiere *a lo sumo* n comparaciones y, en promedio, $1/2 n$, lo cual quiere decir que una búsqueda secuencial es $O(n)$.

La diferencia entre una búsqueda binaria y una secuencial se hace más drástica conforme se incrementa el tamaño del archivo en donde se busca. Si se duplica el número de registros del archivo, se duplica el número de comparaciones necesarias para la búsqueda secuencial; en cambio, cuando se usa la búsqueda binaria, duplicar el tamaño del archivo implica hacer sólo una conjetura más, para el peor caso. Esto tiene sentido, porque se sabe que por cada conjetura se elimina la mitad de las posibles elecciones, de tal forma que si se intenta encontrar el registro de Bill Kelly en un archivo de 2000 registros esto implicaría *a lo sumo*

$$1 + \lfloor \log 2000 \rfloor = 11 \text{ comparaciones}$$

mientras que una búsqueda secuencial implicaría en promedio

$$1/2 n = 1000 \text{ comparaciones,}$$

y podría efectuar hasta 2000.

Sin duda la búsqueda binaria es una forma más atractiva de encontrar lo que se pretende que la búsqueda secuencial. Pero, como es de esperarse, también tiene un precio, y se paga antes de usarla: la

[†]En este texto, $\log x$ se refiere a la función de logaritmo con base 2. Cuando se utilice cualquier otra base, se hará la indicación.

búsqueda binaria funciona sólo cuando la lista de registro está ordenada en términos de la llave que se está usando en la búsqueda. De este modo, para hacer uso de la búsqueda binaria se debe poder *clasificar* una lista con base en una llave.

Como la clasificación ejemplifica de manera excelente los problemas que presenta el diseño de estructuras de archivos, se hará un estudio detenido de los diversos enfoques de clasificación, comenzando en este capítulo con las clasificaciones que tienen lugar en memoria RAM.

6.3

CLASIFICACION DE UN ARCHIVO DE DISCO EN MEMORIA RAM

Considérese la operación de cualquier algoritmo de clasificación interna con el que se esté familiarizado. El algoritmo requiere pasar varias veces por la lista que se clasifica, comparando y reorganizando los elementos. Algunos objetos de información de la lista se transportan grandes distancias desde su posición inicial. Si tal algoritmo fuese aplicado en forma directa a los datos almacenados en un disco, está claro que habría bastantes saltos por todos lados, desplazamientos, y relectura de datos. Esto sería una operación muy lenta, increíblemente lenta.

Si el contenido completo del archivo puede almacenarse en memoria RAM, una alternativa muy atrayente es trasladar el archivo completo del disco a la memoria, y después clasificarlo ahí. Todavía hay que leer los datos del disco, pero así se puede acceder a ellos en forma secuencial sector por sector, sin tener que hacer tantos desplazamientos y recorridos en el disco.

Este es un ejemplo de una clase más general de soluciones al problema de la minimización del uso del disco: forzar el acceso al disco a la forma secuencial efectuando en memoria RAM los accesos más complejos que no son secuenciales. Por desgracia, no siempre es posible emplear este tipo tan simple de solución pero, cuando se pueda, se debe aprovechar.

El primer paso en este enfoque particular de clasificación consiste en trasladar todos los registros del archivo a la memoria. Para hacer las cosas sencillas, se supone que se trata de un archivo de registros, de longitud fija del tipo de los creados por los programas *actualiza.c* y *actualiza.pas*, desarrollados en el capítulo 4. Los archivos creados por esos programas contienen registros de encabezado que indican cuántos registros hay en el archivo. Una vez que se ha leído el número correcto de registros de datos en memoria RAM, se tiene un arreglo de vectores

REGISTROS

Harrison Susan 387 Eastern ...
Kellogg Bill 17 Maple ...
Harris Margaret 4343 West ...
...
:
Bell Robert 8912 Hill ...

FIGURA 6.2 • Registros de datos después de cargar el archivo en el arreglo REGISTROS en memoria RAM (sin encabezado).

de caracteres del mismo tamaño, y cada renglón del arreglo está compuesto del contenido de un registro (Fig. 6.2). Llámese a este arreglo REGISTROS[].

Ya que se han trasladado todos los registros a la memoria, es necesario clasificarlos. Por desgracia, no es posible tratar el conjunto de registros como un conjunto de cadenas de caracteres, clasificando literalmente los elementos del arreglo. El problema es que la clasificación debe hacerse sobre la forma canónica de la llave. Por ejemplo, si se clasifican los registros de la figura 6.2 sin convertir las llaves a su forma canónica, el registro de Harrison precedería al de Harris. Se necesita extraer un segundo arreglo que contenga sólo las llaves en forma canónica y después hacer la clasificación sobre esas llaves.

Por supuesto, debe haber alguna forma de relacionar nuevamente las llaves con los registros de donde fueron extraídas. En consecuencia, cada nodo del arreglo nuevo tiene un segundo campo que contiene el NRR del registro asociado con la llave. Este arreglo se llamará NODOSLLAVE[]. Esta disposición se ilustra en forma esquemática en la figura 6.3(a). Se comienza con el NRR1, reservando el NRR 0 para el registro de encabezado. Este mecanismo de construcción de una cadena de referencia (en este caso es la referencia de una llave hacia la copia del registro original) se conoce como *indirección*.

Desde el punto de vista *conceptual*, la disposición que queda después de la clasificación es como se muestra en la figura 6.3(b). Se subraya la palabra *conceptual* porque en realidad no se desea reorganizar el arreglo de llaves canónicas de manera física. Cada elemento de NODOSLLAVE[] contiene una cadena de caracteres. Si se clasifica el arreglo de llaves moviendo físicamente los elementos, nodo llave, habría que hacer muchas *copias* de cadenas. La clasificación puede

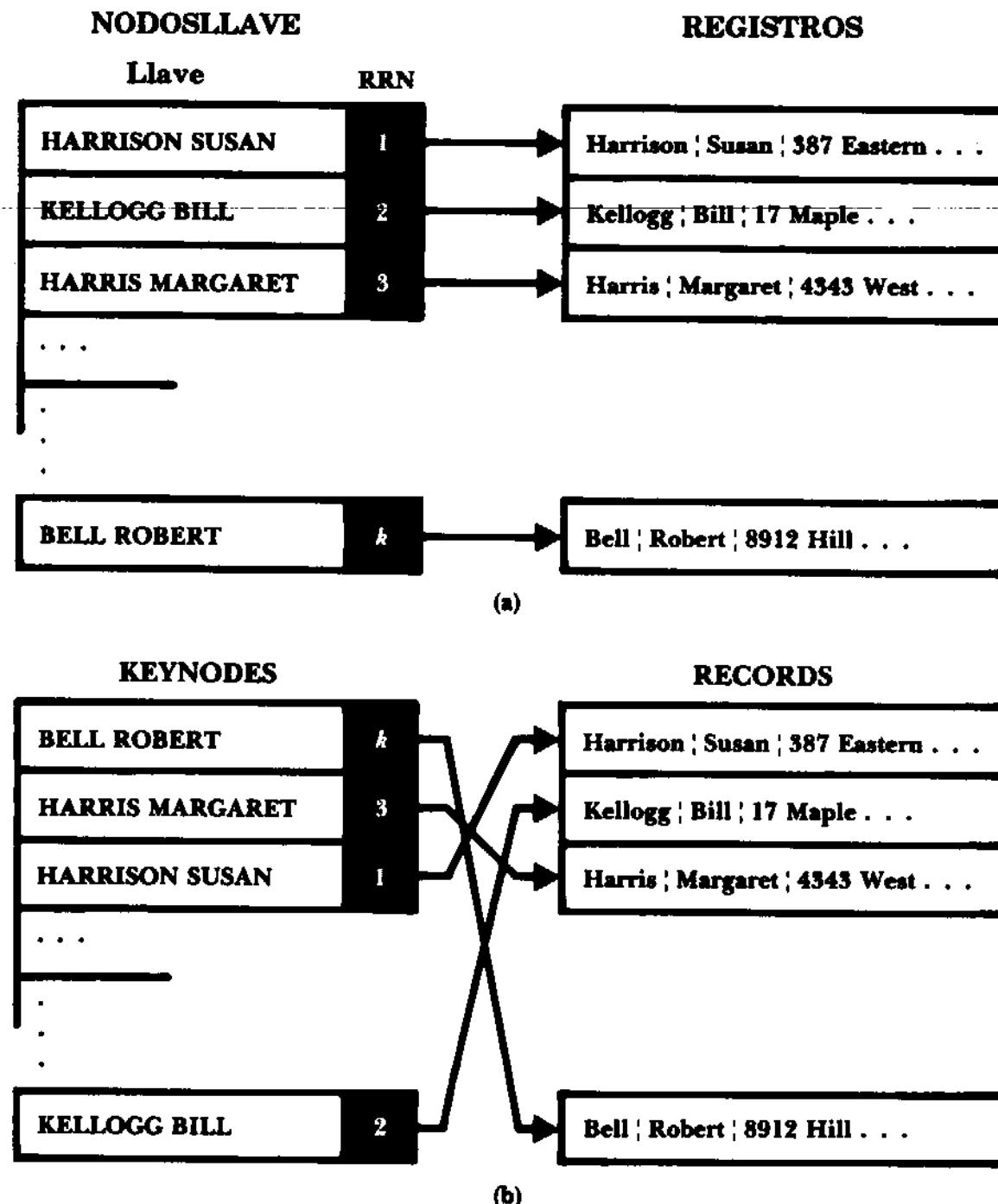


FIGURA 6.3 • Vista conceptual del arreglo de llaves (NODOSLLAVE) y del arreglo de registros (a) antes de la clasificación, y (b) *conceptualmente*, después de la clasificación. (El arreglo de la izquierda en realidad no está reacomodado. Véase el texto.)

requerir un gran número de movimientos; si cada uno implica copiar una cadena con 30 o más caracteres para pasarlo de un lugar a otro, se tendría un desempeño innecesariamente lento.

Se dice innecesariamente lento porque en realidad no se requiere mover las llaves. Cada llave es un elemento del arreglo NODOSLLAVE[] y, por lo tanto, puede hacerse referencia a ella a través de un

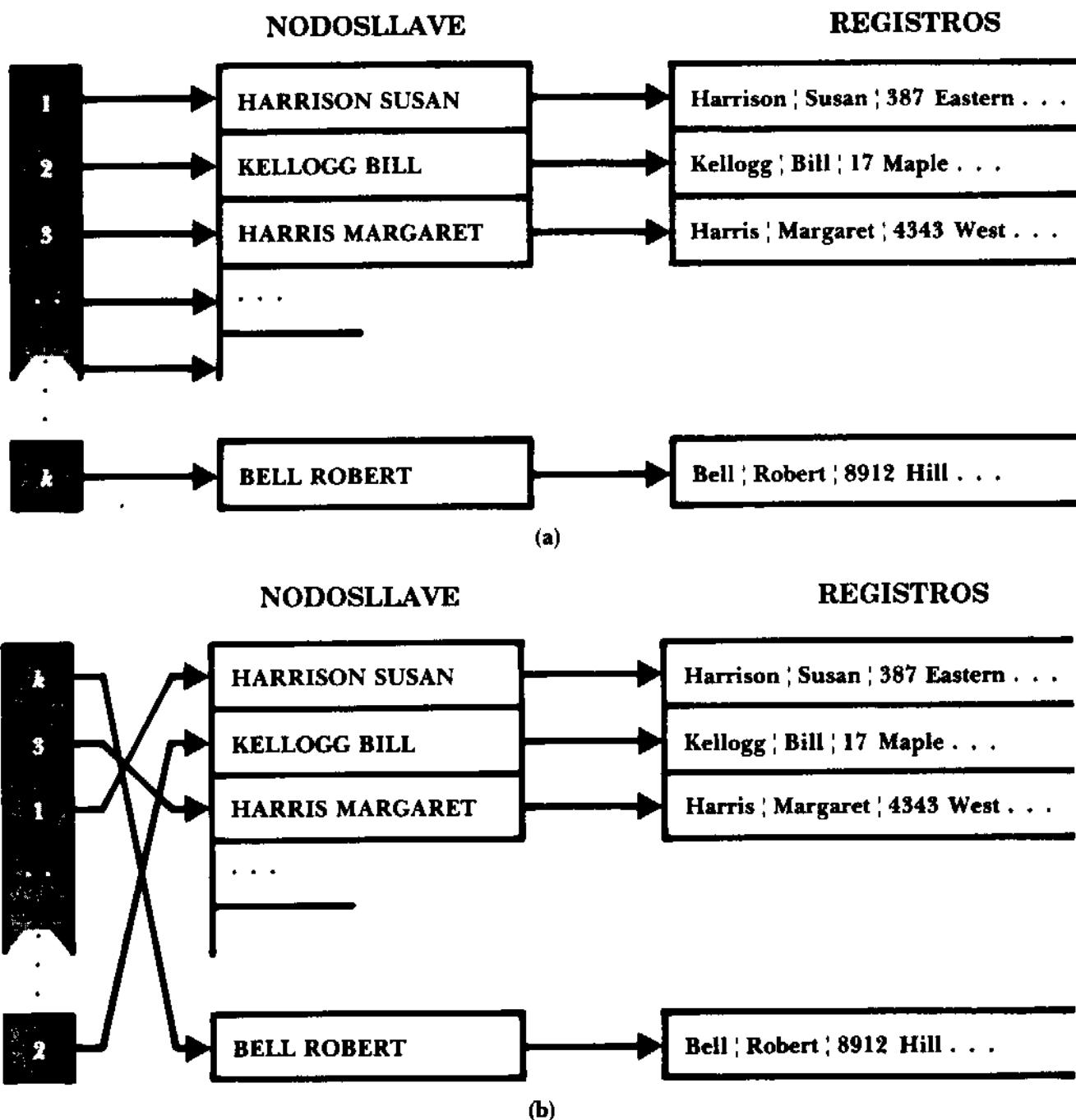


FIGURA 6.4 • Vista conceptual del arreglo de apuntadores (APUNTS), el arreglo de llaves y el arreglo de registros (a) antes de la clasificación, y (b) después de la clasificación. Nótese que el arreglo de llaves canónicas no se ha reacomodado, pero el arreglo de apuntadores sí.

subíndice entero. Por ejemplo, si se quiere hacer referencia a la llave de Susan Harrison en la figura 6.3(a), todo lo que se necesita para mantenerse informado es el entero 1, de tal forma que se tiene el subíndice necesario en la expresión NODOSLLAVE[1]. En consecuencia, se pueden reordenar las llaves sin tener que moverlas si se forma un tercer arreglo, compuesto simplemente por los subíndices del arreglo NODOS-

Dados los siguientes arreglos, cada uno de los cuales inicia con un subíndice 1 y contiene CONT_REG elementos:

- Arreglo de llaves, NODOSLLAVE[]
- Arreglo de enteros, INDICE[], que contiene subíndices del arreglo NODOSLLAVE[] clasificado de acuerdo a la secuencia de llaves
- Arreglo de imágenes de los registros, REGISTROS[]. Los subíndices de este arreglo de registros son tales que NODOSLLAVE[i] corresponde a REGISTROS[i]

para i := 1 hasta CONT_REG
 escribe REGISTRO [INDICE[i]] en el archivo de salida
 siguiente i

FIGURA 6.5 • Escritura de los registros en el orden de la llave.

LLAVE. A este tercer arreglo se le llamará INDICE[], porque después de organizarlo forma un índice a la secuencia del arreglo NODOSLLAVE, correctamente ordenada. Esto parece más complejo de lo que en realidad es. Se comprenderá la idea examinando la figura 6.4 que ilustra las disposiciones, antes y después, de los tres arreglos que se han analizado hasta aquí.

El arreglo INDICE resulta atractivo porque se puede reordenar el arreglo NODOSLLAVE simplemente reubicando los valores enteros dentro de INDICE[], lo cual es mucho más rápido y menos costoso que reubicar los propios elementos NODOSLLAVE. Aunque esta mejora en el desempeño es la consecuencia verdaderamente importante de la introducción de un nivel más de indirección, existe, en forma del arreglo INDICE, un segundo resultado interesante: como no se necesita mover los elementos del arreglo NODOSLLAVE, tampoco es necesaria la referencia de los NRR dentro de NODOSLLAVE[]. Como se puede observar en la figura 6.4(b), existe ahora una relación fija entre el número de subíndice del arreglo NODOSLLAVE y los NRR de los registros de datos almacenados en el arreglo REGISTROS. Ahora los elementos dentro de NODOSLLAVE pueden ser sólo vectores de caracteres; y no es necesaria una cadena de referencia de NODOSLLAVE[] a REGISTROS[].

Una vez que a los elementos de INDICE[] se les vuelve a dar una secuencia de acuerdo con los valores de las llaves en NODOSLLAVE[], se está preparado para escribir todos los REGISTROS, clasificados conforme al orden de la llave. Ese proceso se describe en la figura 6.5. En otras palabras, el subíndice del siguiente REGISTRO por transcribirse en el orden clasificado está contenido en el elemento INDICE[1] de INDICE. Se puede observar que esta lógica de salida de registros funciona al aplicarla a la figura 6.4(b).

6.4**ALGORITMO PARA LA CLASIFICACION
EN MEMORIA RAM**

Es un problema relativamente sencillo trasladar el análisis y los diagramas anteriores a pseudocódigo, para que después se realicen en C, Pascal o algún otro lenguaje. Al examinar el proceso desde un nivel superior, los pasos básicos son los siguientes:

1. Leer los registros del archivo de entrada en el arreglo REGISTROS;
2. Extraer las llaves, construyendo el arreglo NODOSLLAVE;
3. Construir un arreglo INDICE de subíndices de NODOSLLAVE[] y REGISTROS[];
4. Ordenar INDICE[] con base en los valores NODOSLLAVE[], y
5. Usar el nuevo INDICE[] ordenado para transcribir REGISTROS[] al nuevo archivo, conforme al orden clasificado.

PROGRAMA: *clasifram*

Abrir el archivo de entrada como ARCH_ENT
Crear el archivo de salida ARCH_SAL

Ler el registro de encabezado en ARCH_ENT
Escribir una copia del registro de encabezado en ARCH_SAL
CONT_REG := contador de registros del registro de encabezado

```
/* Leer los registros, llenar los arreglos */
para i:= 1 hasta CONT_REG
    Ler un registro de ARCH_ENT en REGISTROS [i]
    Extraer la llave canónica y colocarla en NODOSLLAVE [i]
    INDICE [i] := i
siguiente i

/* Ordenar INDICE[] de acuerdo a los valores de
   NODOSLLAVE[] */
clasif_shell (INDICE, NODOSLLAVE, CONT_REG)

/* Escribir los registros en el orden de clasificación */
para i:= 1 hasta CONT_REG
    Escribir REGISTRO [ INDICE [i] ] a ARCH_SAL
siguiente i

cerrar ARCH_ENT y ARCH_SAL
fin PROGRAMA
```

FIGURA 6.6 • Pseudocódigo para *clasifram*;

El pseudocódigo de la figura 6.6 completa este esbozo. Los arreglos de nombre REGISTROS[], NODOSLLAVE[], e INDICE[] se refieren a las estructuras descritas en la sección anterior. El proceso real de clasificación lo hace una función llamada *clasif-shell()*, la cual se describe en la siguiente sección. Las realizaciones de *clasifram*, tanto en C como en Pascal, se proporcionan al final de este capítulo.

6.5

LA FUNCION DE CLASIFICACION: *clasif-shell()*

Una vez que el programa principal de *clasifram* ha extraído las llaves canónicas y formado de índices de subíndices a este arreglo de llaves, pasa el control a una rutina de clasificación que es la responsable de reacomodar el índice. Hay varios diferentes algoritmos de clasificación que podrían usarse para esta tarea; aquí se usa la clasificación de Shell, llamada así porque fue D.L. Shell quien sugirió este método por primera vez. A menudo se designa este enfoque simplemente como clasificación Shell. Es un algoritmo moderadamente rápido que es fácil de realizar. Baase [1978] proporciona una descripción excelente del algoritmo; el enfoque dado en el pseudocódigo de la figura 6.7 se asemeja mucho a la descripción de Baase, ya que se apoya en una clasificación de inserción. Knuth [1973b] realiza un análisis detallado de la eficiencia del algoritmo.

Para el diseño de cualquier versión de la clasificación Shell es fundamental la elección de una secuencia de los intervalos que se usan para dividir y después subdividir la lista original. Para el programa en pseudocódigo (Fig. 6.7) se emplea el método simple, de usar un intervalo inicial de la mitad de la longitud de la lista, para dividir después el intervalo entre dos, en pasos sucesivos sobre la lista.

Los nombres de los arreglos usados en la figura 6.7 (NODOSLLAVE, INDICE) tienen el mismo significado que en secciones anteriores. Nótese que la rutina de clasificación debe recibir ambos arreglos, INDICE y NODOSLLAVE, ya que clasifica con valores de NODOSLLAVE, pero en realidad reacomoda en el arreglo INDICE. Esta indirección se refleja en el uso de la expresión

NODOSLLAVE [INDICE[i]]

para acceder a los elementos del arreglo NODOSLLAVE. Las realizaciones de la función *clasif-shell()*, tanto en C como en Pascal se incluyen al final del capítulo.

FUNCION: `clasif_shell (INDICE, NODOSLLAVE, CONT_REG)`

variables:

`ESPACIO`: intervalo actual entre elementos que se comparan.

Inicia en `CONT_REG/2` y se reduce a 1.

`LLAVE_INS`: llave a insertarse en un ciclo dado de la clasificación.

(La llave no se inserta en realidad; sino que se mueve el subíndice a la llave dentro de `INDICE[]`.)

`SUB_INS`: subíndice (dentro de `NODOSLLAVE`) de la `LLAVE_INS`. Este subíndice será el elemento que en realidad se mueva en el arreglo `INDICE[]`.

```

ESPACIO := CONT_REG / 2
mientras ( ESPACIO > 0 )
    /* Comienza la inserción para este valor de ESPACIO */
    para j := ( ESPACIO + 1 ) hasta CONT_REG
        LLAVE_INS := NODOSLLAVE [INDICE [j]]
        SUB_INS := INDICE [j]

        /* Revisar hacia atrás buscando el lugar de inserción */
        i := j - ESPACIO
        mientras ((i > 0) y (NODOSLLAVE [INDICE [i]] > LLAVE_INS))
            INDICE [i + ESPACIO] := INDICE [i] /* Deslizar para insertar */
            i := i - ESPACIO           /* Se mueve un intervalo */
        fin mientras

        INDICE [i + espacio]: = SUB_INS /*Se hace la inserción */
        siguiente j

        ESPACIO := ESPACIO / 2 /* Reducir el ESPACIO, inicia
                               el nuevo ciclo */
    fin mientras
fin FUNCION

```

FIGURA 6.7. • Pseudocódigo de `clasif_shell()`.

6.6

LIMITACIONES DE LA BUSQUEDA BINARIA Y DE LA CLASIFICACION EN MEMORIA RAM

Gracias a la capacidad de clasificar archivos, es posible ahora buscar un registro en particular con base en una llave sin tener que realizar la búsqueda secuencial. Se ha dado ya una primera solución al problema con el que comienza el capítulo, el problema de encontrar información rápidamente mediante el uso de una llave. Pero esta solución sólo es el primer paso. Se estudiarán algunos problemas asociados con este

enfoque de "clasificación y después búsqueda binaria", para encontrar información.

PROBLEMA 1: LA BUSQUEDA BINARIA REQUIERE MAS DE UNO O DOS ACCESOS. En este capítulo se afirmó anteriormente que, en el caso promedio, una búsqueda binaria requiere aproximadamente $\lfloor \log n \rfloor + 1/2$ comparaciones. Si cada comparación requiere un acceso a disco, una serie de búsquedas binarias en una lista de 1000 registros requiere, en promedio, 9.5 accesos por solicitud. Si la lista se amplía a 100 000 registros, la longitud de búsqueda promedio aumenta a 16.5 accesos.

Aunque esto significa una mejora *enorme* respecto al costo de una búsqueda secuencial de la llave, también es cierto que 16 accesos, o incluso nueve o diez accesos, no es un costo insignificante. El costo de los desplazamientos en particular es notable, y objetable, si se realizan repetidos accesos por llave.

Cuando se accede a los registros por número relativo de registro (NRR) en lugar de hacerlo por llave, se puede extraer un registro con un solo acceso. Esta mejora es de un orden de magnitud con respecto a los diez o más accesos que la búsqueda binaria requiere con un archivo moderadamente grande. Lo ideal sería acercarse a la eficiencia de la extracción por NRR y mantener, mientras tanto, las ventajas del acceso por llave. En el siguiente capítulo, que versa sobre el uso de las estructuras de índices, se comienza a estudiar las formas de alcanzar este ideal.

PROBLEMA 2: MANTENER UN ARCHIVO ORDENADO ES MUY COSTOSO. La posibilidad de emplear la búsqueda binaria tiene un precio: se debe mantener el archivo clasificado por llaves. Supongamos que se trabaja con un archivo al cual se agregan registros con la misma frecuencia con que se buscan los que ya existen. Si se deja el archivo sin clasificar y se hacen búsquedas secuenciales de los registros, entonces, en promedio, cada búsqueda requiere leer la mitad del archivo. Sin embargo, cada inserción de registro es muy rápida, ya que únicamente implica ir al final del archivo y escribir un registro.

Si, como alternativa, se mantiene el archivo en forma clasificada, puede abatirse considerablemente el costo de la búsqueda, reduciéndola a unos pocos accesos. Sin embargo, se encuentran dificultades cuando se agrega un registro, porque se desea mantener todos los registros ordenados. Insertar un registro nuevo en el archivo implica, en general, no sólo leer la mitad de los registros, sino también *reacomodar* los registros para abrir el espacio requerido para la inserción. En realidad, esto representa *más* trabajo que las simples búsquedas secuenciales en un archivo sin clasificar.

Está claro que los costos de mantener un archivo al que se puede tener acceso con búsqueda binaria no siempre son tan altos como en este ejemplo, que implica la inserción frecuente de registros. Por ejemplo, a menudo ocurre que la búsqueda se requiere con mucho mayor frecuencia que la inserción de registros. En tales circunstancias, los beneficios de la extracción rápida de información pesan más que los costos por guardar el archivo clasificado. Como otro ejemplo, existen muchas aplicaciones donde las inserciones de registros pueden acumularse en un archivo de transacciones y realizarse mediante procesamiento por lotes. Al clasificar la lista de los nuevos registros, antes de agregarlos al archivo principal, es posible *intercalarlos* con los registros existentes. Como se analizará en el capítulo 8, la intercalación es un proceso secuencial en el que se pasa sólo una vez cada registro del archivo. Este puede ser un enfoque eficiente y atractivo para mantener el archivo.

Así, pese a estos problemas, existen situaciones en las que la búsqueda binaria es una estrategia útil. Sin embargo, conocer los costos de la búsqueda binaria también permite ver los requerimientos para obtener mejores soluciones al problema de encontrar información por llave. Las mejores soluciones tendrán que cumplir, al menos, con una de las siguientes dos condiciones.

- No implicarán reordenación de los registros del archivo cuando se agregue un registro nuevo, y
- Estarán asociadas con estructuras de datos que permitan una reordenación del archivo considerablemente más rápida y eficiente.

En los capítulos siguientes se desarrollan enfoques que entran en cada una de estas categorías. Las soluciones del primer tipo pueden implicar el uso de índices simples. También pueden implicar *dispersión (hashing)*. Las soluciones del segundo tipo pueden implicar el uso de *estructuras de árboles* tales como un árbol B, para mantener el archivo en orden.

PROBLEMA 3: UNA CLASIFICACION EN MEMORIA RAM FUNCIONA SOLO CON ARCHIVOS PEQUEÑOS. La posibilidad de la búsqueda binaria está limitada por la posibilidad de clasificar el archivo. El programa *clasifram* funciona sólo si puede leerse el contenido completo de un archivo dentro de la memoria electrónica del computador. Si el archivo es tan grande que no se puede, entonces es necesario un tipo distinto de clasificación.

En lo que resta de este capítulo se desarrolla una variación del programa *clasifram* llamada *clasifllave*. Como *clasifram*, *clasifllave* tiene límites en cuanto al tamaño del archivo que puede clasificar, pero

su límite es más amplio. Lo importante es que al trabajar con *clasifllave* se empezará a vislumbrar un nuevo enfoque para solucionar el problema de encontrar la información, el cual permitirá evitar por completo la clasificación de registros en un archivo.

6.7

CLASIFICACION POR LLAVE

6.7.1 DESCRIPCION DEL METODO

Algunas veces la clasificación por llave se denomina *clasificación por marca* y simplemente es una modificación de la clasificación en memoria RAM. La diferencia es que no se leen los registros reales en memoria; se leen sólo las llaves canónicas.

La figura 6.8 muestra un diagrama muy parecido a los anteriores que ilustran la relación entre las copias de los registros del archivo, el arreglo de llaves y el arreglo índice de subíndices. La diferencia es que el diagrama de la figura 6.8 está dividido a la mitad por una línea punteada. El área de la izquierda se titula *en memoria RAM* y el área de la derecha, *en almacenamiento secundario*.

Clasifllave mantiene los mismos componentes fundamentales que *clasifram*. El proceso de clasificación reacomoda el arreglo de subíndices de las llaves canónicas, exactamente como el *clasifram*. Pero como *clasifllave* nunca pone todo el conjunto de registros en la memoria, no necesita usar tanta como *clasifram*. Esto significa que *clasifllave* puede

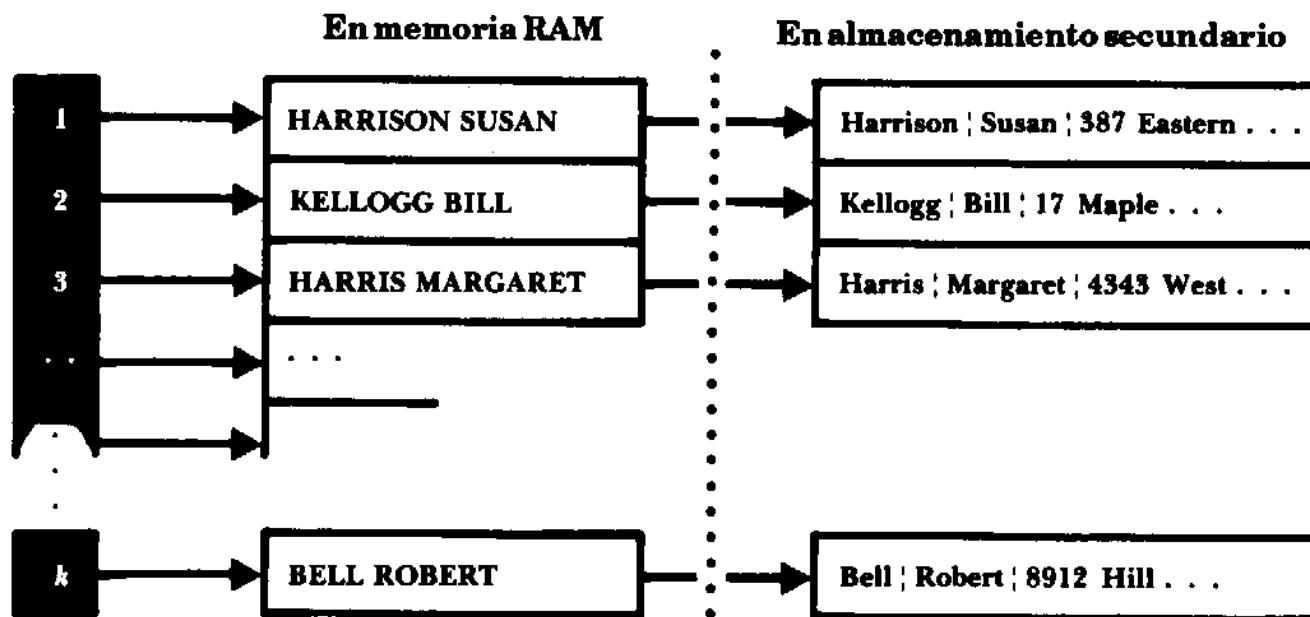


FIGURA 6.8 • Disposición de arreglos para *clasifllave*.

PROGRAMA: *clasifllave*

```

Abrir el archivo de entrada como ARCH_ENT
Crear el archivo de salida ARCH_SAL

Leer el registro de encabezado de ARCH_ENT
Escribir una copia del registro de encabezado en ARCH_SAL
CONT_REG : = contador de registros del registro de encabezado

/* Leer los registros, llena los arreglos */
para i : = 1 hasta CONT_REG
    [REDACTED]
        Extraer la llave canónica, y colocarla en NODOSLLAVE [i]
        INDICE [i] : = i
    siguiente i

/* Ordenar INDICE [] de acuerdo a los valores de NODOSLLAVE [] */
clasif_shell (INDICE, NODOSLLAVE, CONT_REG)

/* Escribir los registros ordenados */
para i : = 1 hasta CONT_REG
    [REDACTED]
siguiente i

cerrar ARCH_ENT y ARCH_SAL
fin PROGRAMA

```

FIGURA 6.9 • Pseudocódigo para *clasifllave* (las líneas que difieren de *clasifram* aparecen en negritas).

clasificar archivos más grandes, dada la misma cantidad de memoria RAM.

Tomando en cuenta la similitud entre los dos enfoques, sólo se necesitan modificaciones menores para que el código de *clasifram* se convierta en un nuevo programa llamado *clasifllave*. La figura 6.9 esboza el procedimiento *clasifllave* en pseudocódigo. Las pocas líneas que difieren de *clasifram* se han puesto en cursivas. Las diferencias son:

- En lugar de leer todos los registros en un arreglo en memoria RAM, simplemente se lee cada registro en un BUFFER temporal y luego se desecha, y
- Cuando se escriben los registros ordenados tienen que leerse por

segunda vez, porque no se tienen todos almacenados en memoria RAM.

6.7.2 LIMITACIONES DEL METODO DE CLASIFICACION POR LLAVE

A primera vista, la clasificación por llave representa una mejora obvia sobre *clasifram*; incluso podría parecer un caso donde se recibe algo a cambio de nada. Se sabe que la clasificación es una operación costosa y que se pretende hacerla en memoria RAM. La clasificación por llave permite llevar a cabo este objetivo sin tener que almacenar a la vez el archivo completo en memoria RAM. Después de todo, se está clasificando con base en una *llave*; entonces ¿por qué hay que molestarse en llevar todo el resto del registro a la memoria? Con la clasificación por llave parece que se puede efectuar la parte costosa de la operación de clasificación en memoria RAM, mientras que se deja la parte más voluminosa en el almacenamiento secundario. Muy elegante.

Pero, al leer acerca de la operación de escritura de los registros ya clasificados, incluso un lector casual percibirá probablemente una nube en este horizonte aparentemente brillante. En la clasificación por llave es necesario leer los registros una segunda vez antes de escribir el nuevo archivo clasificado, y realizar algo dos veces nunca es deseable. Pero el problema es peor aún.

Examíñese cuidadosamente el ciclo *for* que lee los registros antes de transcribirlos al nuevo archivo. Se observará que *no* se está leyendo el archivo secuencialmente, sino que se trabaja en el orden de clasificación, moviéndose del [INDICE] clasificado a los NRR de los registros. (El índice se clasifica como un conjunto de subíndices al arreglo NODOSLLAVE, pero en la relación que se establece entre [NODOS-LLAVE] y el archivo, NODOSLLAVE [*i*] corresponde al registro con NRR = *i*) Puesto que hay que desplazarse a cada registro y leerlo antes de escribirlo, la creación del archivo clasificado requiere tantos *desplazamientos aleatorios* dentro del archivo de entrada como registros existen. Como se ha señalado varias veces, hay una diferencia enorme entre el tiempo requerido para leer todos los registros en un archivo en forma secuencial y el tiempo requerido para leer estos mismos registros cuando se debe localizar cada registro en forma separada. Lo peor es que se efectúan todos estos accesos en forma alternada con proposiciones de escritura al archivo de salida, de modo que aun la escritura del archivo de salida, que parecería secuencial, en la mayoría de los casos implica desplazamientos. La unidad de disco debe mover alternadamente el brazo entre los dos archivos, conforme se lee y se escribe.

El recibir algo a cambio de nada, en la clasificación por llave, se ha evaporado súbitamente. Aun cuando la clasificación por llave hace el

trabajo pesado de la clasificación en memoria RAM, resulta que la creación de una versión clasificada del archivo, a partir del mapa proporcionado por el arreglo INDICE, no es una cuestión insignificante cuando sólo se mantienen copias de los registros en el almacenamiento secundario. En el ejercicio 5 al final del capítulo se hace uso de buffers para reducir las desventajas de la clasificación por llave, manteniendo la ventaja fundamental de requerir menos memoria que *clasifram*.

6.7.3 OTRA SOLUCION: ¿POR QUE MOLESTARSE EN ESCRIBIR DE NUEVO EL ARCHIVO?

La idea fundamental en *clasifllave* es muy atractiva: ¿por qué trabajar con el archivo completo cuando las únicas partes de interés, con respecto a la búsqueda y clasificación, son los campos que se usan para formar la llave? Hay una buena dosis de moderación detrás de esta idea, que da a la clasificación por llave una apariencia promisoria. La promesa se desvanece cuando se presenta el problema de reacomodar todos los registros del archivo para que reflejen el nuevo orden de clasificación.

Es interesante preguntar si se puede evitar este problema con sólo despreocuparse de la tarea que implica: ¿Qué sucede si simplemente, se pasa por alto el tardado asunto de escribir la versión clasificada del archivo? ¿Qué pasa si, en lugar de eso, simplemente se escribe una copia del vector de nodos de llaves canónicas? Supóngase, por ejemplo, que NODOSLLAVE es un arreglo de estructuras de C, o de registros de Pascal, en los que cada NODOSLLAVE individual está definido como sigue, de tal forma que pueda almacenar el NRR del registro de datos original, además de la llave canónica.[†]

En C:

```
typedef struct {           NODOLLAVE = record
    char llave [30];       llave : packed array [0...29] of char;
    short nrr;             nrr : integer
} NODOLLAVE;               END;
```

En Pascal:

Si se hace la escritura del contenido NODOSLLAVE[], lugar de escribir los registros clasificados se tendrá un programa que producirá un índice del archivo original. La relación entre los dos archivos se ilustra en la figura 6.10.

Este es un ejemplo de nuestro tipo favorito de soluciones para

[†]Se emplea un arreglo que comienza con 0 en Pascal, de modo que permita colocar la longitud de la cadena llave en la posición cero del arreglo. En C, la cadena se termina con el carácter nulo, como es de esperarse.

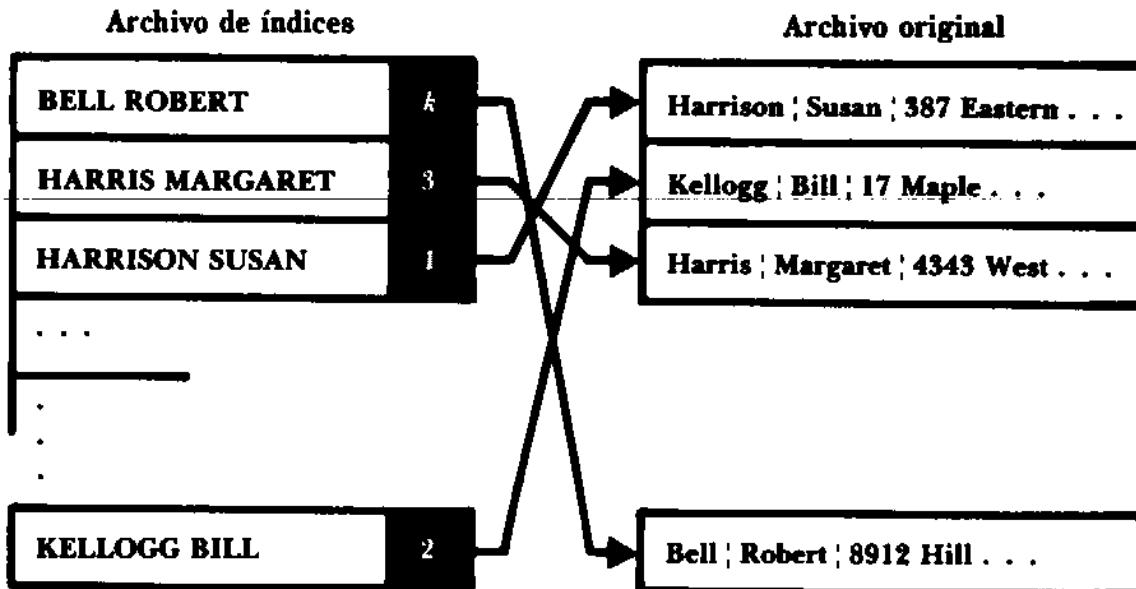


Figura 6.10- Relación entre el archivo índice y el archivo de datos.

problemas de las ciencias de la computación: si alguna parte de un proceso comienza a parecerse a un cuello de botella, conviene pensar saltársela por completo. ¿Se puede continuar sin eso? En vez de crear una copia nueva y clasificada del archivo para la búsqueda, se ha creado un segundo tipo de archivo, un archivo de índices, que se usa en conjunción con el archivo original. Si está buscando un registro en particular, se hace la búsqueda binaria en el archivo de índices y después se usa el NRR almacenado en el registro del archivo de índices, para encontrar el registro correspondiente del archivo original.

Hay mucho que decir acerca del uso de archivos de índices, suficiente para llenar varios capítulos. El siguiente capítulo aborda diversas formas en que se pueden usar índices sencillos, que es el tipo de índices que se ilustra aquí. En capítulos posteriores se habla de las diferentes formas de organizar los índices para proporcionar accesos más flexibles y un mantenimiento más sencillo.

Todas estas estructuras de índices pueden verse como una especie de ampliación lógica de las ideas analizadas en este capítulo. Como los programas de clasificación, las distintas estructuras de índices implican extraer llaves de los registros de un archivo. Estas estructuras de índices por lo regular también implican *indirección* que es el acceso a los registros por medio de una cadena de uno o más apuntadores, o referencias a números relativos de registro. Y, como en la solución al problema de la clasificación por llave, se centran en mantener y organizar un archivo de *índices*, en lugar de reorganizar simplemente los registros originales. Los problemas que se comentan en este capítulo son, de

hecho, los problemas básicos de cualquier sistema de almacenamiento y extracción de información. El problema permanece igual, sólo son las soluciones las que cambian, se estructuran unas con otras y se vuelven más complicadas, flexibles y poderosas.

6.8

REGISTROS FIJOS

En el capítulo 5 se analizó el problema de actualizar y mantener archivos. Gran parte de ese capítulo gira alrededor de los problemas de eliminación de registros y mantenimiento de la información del espacio liberado por los registros eliminados, de tal forma que pueda reutilizarse. En el capítulo 5 se creó una lista de las entradas disponibles de registros eliminados, ligando todas las entradas disponibles. En el caso de la eliminación de registros de longitud fija, el ligamento se hace escribiendo un campo en cada registro eliminado, que apunte al número relativo de registro (NRR) del siguiente registro eliminado. Con los registros de longitud variable, el campo de liga consiste en la *distancia en bytes*, real, del siguiente registro en la lista de disponibles. En ambos casos los campos de liga proporcionan información muy específica acerca de la *posición física* del siguiente registro disponible.

Cuando un archivo contiene esas referencias a las localidades físicas de los registros, se dice que tales registros están *fijos*. Se puede comprender mejor esta terminología si se consideran los efectos de clasificar uno de esos archivos que contenga una lista de disponibles para los registros eliminados. Es decir, un *registro fijo* es uno que no puede moverse. Otros registros del mismo archivo, o de algún otro (como un archivo de índices), contienen referencias a la localidad física del registro. Si el registro se mueve, estas referencias no conducen ya al registro y se convierten en lo que se llama *apuntadores suspendidos*, que llevan a localidades dentro del archivo que son incorrectas y sin significado.

Sin duda, el uso de registros fijos en un archivo puede hacer la clasificación más difícil y algunas veces imposible. Pero ¿qué pasa si se quiere manejar el acceso rápido por llave, reutilizando a la vez el espacio disponible por la eliminación de registros? Una solución es usar un archivo de índices para mantener el orden de los registros, y mantener el archivo de datos real en su orden original. De nuevo, el problema de encontrar las cosas nos remite a la sugerencia de que se profundice más en el uso de los índices, lo cual, a su vez, conduce al siguiente capítulo.

RESUMEN

El principal problema estudiado en este capítulo es el de encontrar las cosas rápidamente en un archivo por medio del uso de una llave. En los capítulos anteriores no era posible acceder a un registro rápidamente sin conocer su posición física o su número relativo de registro. El presente capítulo explora algunos de los problemas y oportunidades asociados con el acceso directo por llave.

Este capítulo en realidad desarrolla sólo un método para encontrar registros por llave: la *búsqueda binaria*. La búsqueda binaria requiere $O(\log n)$ comparaciones para encontrar un registro en un archivo con n registros y, por tanto, es muy superior a la búsqueda secuencial. La parte más importante de este capítulo está asociada con las funciones y operaciones requeridas para manejar la búsqueda binaria en un archivo.

Como la búsqueda binaria funciona sólo en un archivo clasificado, es absolutamente necesario un procedimiento de clasificación. El problema de la clasificación se dificulta debido a tres factores:

- Se desea clasificar un archivo con base en llaves formadas canónicamente, que corresponden a cada registro, y no con base en el contenido de los registros mismos.
- Se desea que la clasificación se haga rápidamente, lo que implica *no* efectuar mucho movimiento físico de los datos. Más específicamente, no se quiere estar moviendo y copiando campos completos de la llave.
- Se está clasificando archivos en almacenamiento secundario en lugar de hacerlo en vectores en memoria RAM. Se necesita desarrollar un procedimiento de clasificación que no requiera múltiples desplazamientos sobre el archivo.

El programa de clasificación en memoria RAM responde a todos estos requisitos y lo hace empleando *indirección*. Se leen los registros del archivo secuencialmente en un arreglo, formando al mismo tiempo un vector de llaves, que están relacionadas por su posición con el registro del que son imagen. También se forma un vector de subíndices relacionados con las posiciones dentro del vector de llaves. Aunque la clasificación es por llave, el vector que se reordena es el de subíndices. Se emplea la clasificación Shell para efectuar el trabajo de ordenamiento.

En este punto del capítulo se ha resuelto ya el problema inicial de organizar y buscar en un archivo para encontrar registros con una llave primaria específica sin tener que hacer búsqueda secuencial. El si-

guiente paso es evaluar lo hecho. Hasta este punto, se asocian tres desventajas con la clasificación y búsqueda binaria que se han desarrollado.

- Aunque la búsqueda binaria representa un adelanto enorme respecto a la búsqueda secuencial, el hecho es que, aun con archivos que contengan sólo unos pocos miles de registros, se accedería al disco docenas de veces o más. Sería mucho mejor tener un método que requiriese sólo uno o, a lo sumo, tres o cuatro accesos al disco por registro. Esta necesidad de reducir el número de accesos al disco por registro es mayor en aplicaciones en donde se requieren accesos por llave a un gran número de registros.
- El requisito de que el archivo esté ordenado es muy costoso. Para archivos activos donde los registros se agregan con la misma frecuencia con la que se accede a ellos, el costo por mantener el archivo clasificado puede ser mayor que los beneficios de la búsqueda binaria.
- La clasificación en memoria RAM puede emplearse sólo en archivos relativamente pequeños, ya que se debe almacenar el archivo completo en la memoria. Esto limita el tamaño de los archivos que se podrían organizar para búsqueda binaria, teniendo en cuenta las herramientas de clasificación estudiadas hasta aquí.

El tercer problema puede resolverse parcialmente desarrollando procedimientos de clasificación más poderosos, como el método que se conoce como *clasificación por llave*. Esto se parece a la clasificación en memoria RAM en la mayoría de los aspectos, pero no usa la memoria para almacenar todo el archivo. En lugar de esto, lee sólo las *llaves* de los registros y después clasifica las llaves. Es decir, la clasificación por llave usa la lista de llaves clasificada para reacomodar los registros del almacenamiento secundario, de modo que estén en orden.

La desventaja de la clasificación por llave es que el reacomodo de un archivo con n registros requiere n desplazamientos aleatorios en el archivo original. Esto puede tomar mucho más tiempo que hacer una búsqueda secuencial del mismo número de registros. Sin embargo, el estudio de la clasificación por llave no es un esfuerzo desperdiciado, porque es una forma de realizar sencilla y fácilmente la clasificación de muchos archivos demasiado grandes para entrar en la memoria. Quizás sea más importante aún el hecho de que conduce, en forma natural, a la sugerencia de simplemente escribir la lista clasificada de llaves en el almacenamiento secundario, dejando a un lado el costoso reacomodo del archivo. Esta lista de llaves, acoplada con las marcas NRR que apuntan a los registros originales, es un ejemplo de un índice. La indización es un tema que se examina con mucho mayor detalle en el capítulo 7.

El capítulo 6 termina con el análisis de otro costo de la clasificación y búsqueda potencialmente oculto. Los registros fijos son registros a los que se hace referencia desde algún otro lugar (del mismo archivo o de algún otro) de acuerdo con su posición física. La clasificación y la búsqueda binaria no pueden aplicarse a un archivo que contenga registros fijos ya que, por definición, la clasificación cambia la posición física del registro. Dicho cambio hace que las otras referencias a este registro sean inexactas, creando el problema de los apuntadores suspendidos.

TERMINOS CLAVE

Búsqueda binaria. El algoritmo de búsqueda binaria localiza una llave en una lista ordenada seleccionando repetidamente el elemento medio de la lista, dividiéndola a la mitad, y formando una lista nueva y más pequeña a partir de la mitad que contiene la llave. Este proceso continúa hasta que el elemento seleccionado sea la llave que se buscaba.

Clasificación por llave. Método de clasificación de un archivo que no requiere almacenarlo por completo en la memoria. Sólo las llaves se mantienen en memoria, junto con apuntadores que las asocian con los registros del archivo de donde se trajeron. Las llaves se clasifican, y la lista de llaves clasificada se usa para construir una nueva versión del archivo que tiene los registros ya ordenados. La ventaja principal de la clasificación por llave es que requiere menos memoria que la clasificación en memoria RAM. La desventaja es que el proceso de construcción de un archivo nuevo requiere bastantes desplazamientos.

Clasificación Shell. Algoritmo interno de clasificación relativamente rápido, que clasifica una lista ordenando primero elementos muy separados para después, mediante recorridos sucesivos por la lista, ordenar elementos separados por intervalos cada vez más pequeños. Se empleó para efectuar la porción en memoria RAM de las clasificaciones desarrolladas en este capítulo.

Indirección. Término que describe el proceso de referir información de manera indirecta, mediante el uso de una serie de apuntadores u otros tipos de localizadores. El método empleado en *clasifram*, en este capítulo, proporciona un pequeño estudio sobre la indirección, ya que el reacomodo real se hace en un vector de subíndices dentro de un arreglo que contiene las llaves canónicas. Los elementos de este arreglo de llaves, a su vez,

están relacionados por su posición con los registros que, finalmente, son los objetos de la clasificación. El poder de la indirección como herramienta general para abordar mejor los problemas se hará más evidente conforme se estudia la indización, en capítulos posteriores.

Registro fijo. Se dice que un registro está fijo cuando existen otros registros o estructuras de archivos referidas a él mediante su posición física. Está *fijo* en el sentido de que no se tiene la libertad de alterar la posición física del registro, ya que al hacerlo se destruiría la validez de las referencias físicas al registro. Estas referencias se convierten en apuntadores suspendidos inútiles.

EJERCICIOS

1. Compare la eficiencia del caso promedio de la búsqueda binaria de registros con la búsqueda secuencial, suponiendo:

- a) que los registros que se están buscando efectivamente están en el archivo;
- b) que la mitad de las veces los registros que se buscan no están en el archivo;
- c) que la mitad de las veces los registros que se buscan no están en el archivo y que los registros que faltan se deben insertar.

Haga una tabla que muestre las comparaciones de desempeño de archivos de 1000, 2000, 4000, 8000, y 16000 registros.

2. Si los registros del problema 1 se manejan por bloques con 20 registros por bloque, ¿cómo afecta esto al desempeño de las búsquedas binarias y secuencial?

3. La clasificación empleada en *clasifram* no ordena el archivo de datos; lo que ordena es un arreglo de llaves en forma canónica. Suponga que las llaves de los registros de datos ya están en forma canónica y almacenadas en el primer campo del arreglo REGISTROS[] ¿Cómo es que esto podría eliminar la necesidad del arreglo de nodos llave? ¿Cómo tendría que cambiarse *clasifram* para prescindir del uso del arreglo de nodos llave? ¿Qué ahorro se obtendría si se pudiera usar este enfoque?

4. El programa *clasifram* funciona sólo con archivos lo suficientemente pequeños para caber en la memoria RAM. Algunos sistemas de cómputo proporcionan a los usuarios una cantidad de memoria RAM casi ilimitada, con una técnica de administración de memoria llamada

almacenamiento virtual. Evalúe el uso de *clasifram* para clasificar grandes archivos en sistemas que usen almacenamiento virtual.

5. En el análisis de la clasificación por llaves se considera el alto gasto asociado con el proceso de crear realmente el archivo de salida clasificado, dado el vector clasificado de apuntadores a los nodos de llaves canónicas. El gasto gira alrededor de dos áreas principales de dificultad:

- Tener que saltar de un lado a otro en el archivo de entrada, efectuando muchos desplazamientos para extraer los registros recién clasificados, y
- Escribir el archivo de salida al mismo tiempo que se lee el archivo de entrada; saltar entre archivos puede implicar desplazamientos.

Elabore un planteamiento de este problema que use buffers para almacenar varios registros, mitigando así estas dificultades. Si la solución propuesta es viable, obviamente los buffers deben emplear menos memoria RAM que una clasificación que tenga lugar enteramente dentro de la memoria electrónica.

EJERCICIOS DE PROGRAMACION

6. Realice la función *búsqueda_bin()*, ya sea en C o en Pascal. Escriba un programa llamado *busca*, para probar la función *búsqueda_bin()*. Suponga que los archivos se crean con el programa *actualiza* desarrollado en el capítulo 4, y después se clasifican mediante el programa *clasifram* que se proporciona al final de este capítulo. Incluya suficiente información para depuración en el programa *busca* y en la función *búsqueda_bin()* para observar la lógica de búsqueda binaria conforme se hacen las conjeturas sucesivas acerca de dónde se debe colocar el registro nuevo.

7. Modifique la función *búsqueda_bin()* de modo que, si la llave no está en el archivo, devuelva el número relativo de registro que la llave ocuparía si estuviera en el archivo. La función también debe continuar indicando si la llave se encontró o no.

8. Reescriba el programa *busca* del problema 6 para que use la nueva función *búsqueda_bin()* desarrollada en el problema 7. Si la llave buscada está en el archivo, el programa debe desplegar el contenido del registro. Si la llave no se encuentra, el programa debe desplegar una lista de las llaves que rodean la posición que la llave debería haber ocupado. Debe ser posible moverse a voluntad hacia atrás o hacia

adelante a lo largo de esta lista. Dada esta modificación, no se tiene que recordar una llave completa para extraerla. Si, por ejemplo, se sabe que se busca a alguien llamado Smith, pero no se puede recordar el nombre de la persona, este nuevo programa permite ir al área donde están almacenados todos los registros Smith. Entonces se puede mover hacia atrás o adelante, a lo largo de las llaves, hasta reconocer el nombre correcto.

9. Escriba una versión de *clasifram* que pueda clasificar un archivo de *registros de longitud variable* del tipo producido por los programas *escribereg* del capítulo 4. Como *escribereg* no crea un registro de encabezado, el programa *clasifram* debe contar los registros conforme los lee. Un efecto colateral que tiene esto es que la versión en C de *escribereg* ya no puede seguir asignando todo el espacio de los arreglos en forma dinámica con una serie de proposiciones *calloc()*. Cambie la versión en C de modo que, como en la versión en Pascal, fije el tamaño de los arreglos en algún número máximo de registros.

LECTURAS ADICIONALES

Este capítulo aborda superficialmente los aspectos relacionados con la búsqueda y clasificación de archivos. Gran parte del resto del libro se dedica al análisis detallado de estos aspectos, así que una de las fuentes de lecturas adicionales es el texto mismo. Es muchísimo lo que se ha escrito, incluso sobre los aspectos relativamente sencillos que aparecen en este capítulo.

La referencia clásica sobre clasificación y búsqueda es Knuth [1973b], donde se proporciona un análisis excelente de las limitaciones de los métodos de clasificación por llave. También desarrolla un estudio muy completo sobre la búsqueda binaria, donde muestra claramente la analogía entre la búsqueda binaria y el uso de árboles binarios.

Baase [1978] ofrece un análisis comprensible sobre el desempeño de la búsqueda binaria. También explica el funcionamiento de la clasificación Shell. La versión de la clasificación Shell que se usó en este capítulo emula a la desarrollada por Baase.

C.A.R. Hoare pronunció un discurso al recibir el premio Turing de la Association for Computing Machinery (ACM), donde hizo un recuento del desarrollo de la clasificación *Quicksort*, que comenzó cuando trataba de realizar una clasificación Shell. Sus comentarios son muy accesibles e interesantes, con perspectivas que van más allá de la clasificación y la búsqueda. Aparecen en *Communications of the ACM*, febrero de 1981.

PROGRAMAS EN C

En este capítulo se describe la estructura del programa *clasifram* con detalle. Lo que sigue es una realización de *clasifram* en C; a continuación se hacen algunos comentarios:

- Las funciones *toma_archent()*, *toma_archsal()* y *extrae_llave()* son funciones estáticas y locales que deben incluirse en el mismo archivo que la rutina principal de *clasifram*. La función *clasif_shell()* está escrita de manera que se pueda colocar en un archivo independiente para compilarse por separado.
- Las funciones *hazllave()* y *toma_campo()* se listan al final del capítulo 4. La función *toma_campo()* es parte de un módulo llamado *tomarc.c*. Como antes, *hazllave()* debe ligarse con las funciones *cadespac()* y *mayúsculas()*, que se incluyen en el archivo *cadsfuncs.c*.
- Debido a que el lenguaje C permite al programador elegir la forma de trabajar, con apuntadores o con subíndices, es bastante fácil asignar *dinámicamente* los arreglos *REGISTROS[]*, *NODOS-LLAVE[]* e *INDICE[]*. Lo único que se necesita hacer para pedir la cantidad adecuada de espacio es usar la función *calloc()*, y después hacer una *conversión forzosa* de la dirección resultante al tipo correcto del apuntador. El uso de un operador de *conversión forzosa* es absolutamente necesario, para que la transición entre *REGISTRO[0]* y *REGISTRO[1]*, por ejemplo, sea un salto de un registro entero, en lugar de un salto de un solo byte.
- La función *clasif_shell()* hace uso de una macro titulada *CAMPO_LLAVE()* para simplificar la referencia al arreglo *NODOS-LLAVE[]*, por medio de los subíndices contenidos en *INDICE[]*.

```
/* clasifram.c ...
```

```
Solicita un archivo de entrada en formato de registros de longitud
fija creado por actualiza.c y solicita el nombre del archivo de salida.
Clasifica los registros del archivo de entrada y los escribe en el
archivo de salida. Preserva los registros de encabezado al principio
del archivo.
```

Esta clasificación tiene lugar en memoria RAM. En esta versión en C, el espacio en memoria RAM se asigna dinámicamente. El tamaño del archivo que puede clasificarse está limitado por la cantidad de memoria RAM disponible.

```

#include "arches.h"
#define LONG_REG 64
typedef char NODOLLAVE[30];
typedef char REGDATOS [LONG_REG + 1];
static struct {
    short cont_reg;
    char relleno[30];
} encabezado;

main()
{
    int fd_ent, fd_sal, i, cont_reg;
    char *calloc();
    REGDATOS *registros; /* Dirección del arreglo de registros */
    NODOLLAVE *nodosllave; /* Dirección del arreglo de nodosllave */
    short *índice; /* Dirección del arreglo de índice */

    fd_ent = toma_archent(); /* Abre el archivo de entrada */
    fd_sal = toma_archsal(); /* Abre el archivo de salida */

    if (fd_ent < 0 || fd_sal < 0) /* Termina si la apertura o creación
                                    fallaron */
    {
        printf ("Programa terminado\n");
        exit (1);
    }

    cont_reg = encabezado. cont_reg; /* Convierte el contador de registros
                                    a int para usarlo en calloc() */

    /* asigna memoria para el archivo completo, para el arreglo de nodosllave
    y para el arreglo índice de subíndices de los nodos llave */
    registros = (REGDATOS *) calloc(cont_reg, sizeof (REGDATOS));
    nodosllave = (NODOLLAVE *) calloc(cont_reg, sizeof (NODOLLAVE));
    índice = (short *) calloc(cont_reg, sizeof (short));

    /* Revisa que todas las asignaciones hayan sido exitosas */
    if (registros == OL || nodosllave == OL || índice == OL) {
        printf ("No se pudo asignar el espacio requerido\n");
        printf ("El archivo es demasiado grande para\n");
        printf ("clasificarse en memoria\n");
        exit (1);
    }

    /* Lee los registros secuencialmente, colocándolos en el arreglo
    registros[]. Extrae las llaves y asigna valores a índice[]
    conforme se avanza */
    for (i = 0; i < cont_reg; i++) {

```

```
    read (fd_ent, registros [i], LONG_REG);
    extrae_llave (nodosllave [i], registros [i]);
    indice [i] = i;
}

clasif_shell (indice, nodosllave, cont_reg);
/* Escribe los registros en el orden de los subíndices del
arreglo indice[] ya clasificado */

for (i = 0; i < cont_reg; i++)
    write (fd_sal, registros [indice [i]], LONG_REG);

close (fd_ent);
close (fd_sal);
}

/* toma_archent() . . .
Toma el nombre del archivo de entrada, intenta abrirlo, y si hay éxito,
lee el registro de encabezado en el encabezado
*/
static toma_archent () {
    char nomarch [30];
    int fd;

    printf ("Proporcione el nombre del archivo por clasificar: ");
    gets (nomarch);
    if ((fd = open(nomarch, LECTESCRIT)) < 0) /* ¿OPEN falla? */
        printf ("El archivo %s no puede abrirse\n", nomarch);
    else
        read (fd, &encabezado, sizeof (encabezado));
    return (fd);
}

/* toma_archsal() . . .
Toma el nombre del archivo de entrada, lo crea y escribe el registro de
encabezado que se leyó durante toma_archent().
*/
static toma_archsal() {
    char nomarch [30];
    int fd;

    printf ("Proporcione el nombre del archivo de salida: ");
    gets (nomarch);
    if ((fd = creat (nomarch, MODOP)) < 0) /* ¿CREAT falla? */
        printf ("El archivo %s no puede crearse\n", nomarch);
    else
```

```

        write (fd, &encabezado, sizeof (encabezado));
        return (fd);
    }

/* extrae_llave (llave, registro) . . .
   Extrae el apellido y el nombre de los primeros dos campos en "registro"
   y los almacena en "llave" en forma canónica.
*/
static extrae_llave (llave, registro)
REGDATOS registro;
char llave [];
{
    int pos_bus;
    char nombre [30], apellido [30];

    pos_bus = 0;
    pos_bus = toma_campo (apellido, registro, pos_bus, LONG_REG);
    pos_bus = toma_campo (nombre, registro, pos_bus, LONG_REG);
    hazllave (apellido, nombre, llave);
}

/* clasif_shell (índice, nodosllave, n) . . .
   índice[] es un arreglo de subíndices del arreglo nodosllave[], que
   contiene las llaves en forma canónica. Ambos arreglos contienen n
   elementos válidos. Esta función emplea la clasificación Shell para
   ordenar los subíndices del arreglo índice[] de modo que hagan referencia
   a las llaves en nodosllave[] en el orden de clasificación por llave
*/
#define CAMPO_LLAVE (i) nodosllave [índice [(i)]]
typedef char NODOLLAVE [30];

clasif_shell (índice, nodosllave, n)
    short índice [];
    NODOLLAVE nodosllave [];
    int n;
{
    int espacio, i, j;
    char *llave_ins;
    short subíndice_ins;

    for (espacio = n >> 1; espacio > 0; espacio >>= 1)
    {
        /* Empieza la clasificación por inserción para este espacio */
        for (j = espacio; j < n; j++)

```

```

    llave_ins = CAMPO_LLAVE (j);
    subindice_ins = indice [j];

    /* Trabaja hacia atrás buscando la posición de inserción */
    for (i = j - espacio; i >= 0; i -= espacio)
    {
        if (strcmp (CAMPO_LLAVE (i), llave_ins) <= 0)
            break;
        indice [i + espacio] = indice [i];
    }
    indice [i + espacio] = subindice_ins;
}
}

```

PROGRAMAS EN PASCAL

En este capítulo se describe la estructura del programa *clasifram* con detalle. Lo que sigue es una realización de *clasifram* en Pascal. A continuación se hacen algunos comentarios:

- Como antes, ésta es una realización en Turbo Pascal. Se emplea la opción {\$B-} del compilador y nuestro propio tipo *cadena* para que esté más apegado al Pascal estándar. Véanse los comentarios asociados con los programas de Pascal anteriores, al final de los capítulos 2 y 4, para más detalles acerca de las variaciones entre estos programas, el Pascal estándar y el uso normal de Turbo Pascal.
 - Se usan las instrucciones para el compilador {\$I herramientas.prc}, {\$I toma.prc} y {\$I caddat.prc} para incluir varios procedimientos que se definen en otros lugares. El archivo *herramientas.prc* contiene las funciones requeridas para efectuar operaciones básicas sobre variables de tipo *cadena* (véase el Apéndice B); *toma.prc* contiene la función *toma_campos()* (véanse los programas del final del Cap. 4); *caddat.prc* contiene la función *caddat()*, que transfiere caracteres de una variable *cadena* a un registro REGDATO (véanse los programas al final del Cap. 4).
 - En la realización en C es fácil asignar dinámicamente los arreglos requeridos, pidiendo con exactitud tanta memoria RAM como sea necesaria, mediante el número de registros a clasificar. Aunque Pascal maneja asignación dinámica, lo hace registro por registro. Por ello, no se puede trabajar en términos de un arreglo contiguo de

registros asignado dinámicamente. En consecuencia, se asignan arreglos de tamaño fijo en esta realización.

```
PROGRAM clasifram (INPUT, OUTPUT);
```

```
{ Solicita un archivo de entrada en el formato de registros de longitud fija
  creado por actualiza.pas y solicita el nombre del archivo de salida.
  Clasifica los registros del archivo de entrada y los escribe en el archivo
  de salida. Preserva los registros de encabezado al principio del archivo.
```

Esta clasificación tiene lugar en memoria RAM. En esta versión en Pascal se asigna una cantidad fija y predeterminada de memoria RAM para los arreglos. El número de registros que pueden clasificarse se indica con la constante MAX_CONT_REG.

```
}
```

```
{$B-}
```

```
CONST
```

```
  LONG_REG = 64;
  MAX_CONT_REG = 100;
  TAM_MAX_REG = 255;
  DELIM_CAR = ':';
```

```
TYPE
```

```
  REGDATO = RECORD
    longitud : integer;
    datos : packed array [1 .. LONG_REG] of char
  END;
  arch_reg_fijo = file of REGDATO;
  nombre_a = packed array [1 .. 40] of char;
  cadena = packed array [0 .. LONG_REG] of char;
  REGLLAVE = cadena;
  ARREGLO_INDICE = packed array [1 .. MAX_CONT_REG] of integer;
  ARREGLO_NODOSLLAVE = packed array [1 .. MAX_CONT_REG] of REGLLAVE;
  ARREGLO_REGISTROS = packed array [1 .. MAX_CONT_REG] of REGDATO;
```

```
VAR
```

```
  fd_ent : arch_reg_fijo;
  fd_sal : arch_reg_fijo;
  encabezado : REGDATO;
  cont_reg : integer;
  i : integer;
  indice : ARREGLO_INDICE;
  nodosllave : ARREGLO_NODOSLLAVE;
  registros : ARREGLO_REGISTROS;
```

```
buffcad      : cadena;
buffregdat   : REGDATO;
llave        : cadena;

{$I herramientas.prc    }
{$I toma.prc          }
{$I caddat.prc         }

PROCEDURE toma_archent (VAR ar_chent: arch_reg_fijo);

{ toma_archent() toma el nombre del archivo de entrada, lo abre y lee el
registro de encabezado en el encabezado }

VAR
  nomarch : nombre a;
BEGIN
  write ('Proporcione el nombre del archivo por clasificar: ');
  readln (nomarch);
  assign (arch_ent, nomarch);
  reset (arch_ent);
  read (arch_ent, encabezado)
END; { toma_archent }

PROCEDURE toma_archsal (VAR arch_sal: arch_reg_fijo);

{toma_archsalent() toma el nombre del archivo de salida, lo coloca en modo
de creación y escribe el registro de encabezado que se leyó durante
toma_archent. }

VAR
  nomarch : nombre a;
  i       : integer;
BEGIN
  write ('Proporcione el nombre del archivo de salida: ');
  readln (nomarch);
  assign (arch_sal, nomarch);
  rewrite (arch_sal);
  write (arch_sal, encabezado)
END; { toma_archsal }

PROCEDURE lee_reg ( VAR fd : arch_reg_fijo; VAR buffcad: cadena);

{lee_reg lee un registro del archivo fd en el registro buffer y convierte el
contenido del buffer en una variable del tipo cadena}
```

```

VAR
  buffregdat : regdato;
  i           : integer;
BEGIN
  readln (fd, buffregdat);

  buffcad [0] := CHAR (buffregdat.en);
  for i:= 1 to buffregdat.longitud DO
    buffcad [i] := buffregdat.datos [i]
END; { lee_reg}

PROCEDURE extrae_llave ( VAR llave : cadena; VAR buffcad : cadena);

{ Extrae el apellido y el nombre de los primeros dos campos en el buffer de
cadena y los almacena en la llave en forma canónica. }

VAR
  pos_bus   : integer;
  nombre     : cadena;
  apellido   : cadena;
BEGIN
  pos_bus := 0;
  pos_bus := toma_campo (apellido, buffcad, pos_bus, LONG_REG);
  pos_bus := toma_campo (nombre, buffcad, pos_bus, LONG_REG);
  hazllave (apellido, nombre, llave)
END; { extrae_llave }

PROCEDURE clasif_shell (VAR indice: ARREGLO_INDICES; nodosllave:
                        ARREGLO_NODOSLLAVE; n: integer);

{ indice [] es un arreglo de subíndices del arreglo nodosllave [], que contiene
las llaves en forma canónica. Ambos arreglos contienen n elementos válidos.
Esta función emplea la clasificación Shell para ordenar los subíndices del
arreglo indice[] de modo que hagan referencia a las llaves en nodosllave[]
en el orden de clasificación por llave. }

VAR
  espacio      : integer;
  i, j          : integer;
  subindice_ins : integer;
  llave_ins    : cadena;
BEGIN
  espacio := n div 2;
  while espacio > 0 DO
    BEGIN { Empieza la clasificación por inserción para
            este espacio }

```

```

for j := (espacio + 1) to n DO
BEGIN
llave_ins := nodosllave [indice [j]];
subindice_ins := indice [j];

(Trabaja hacia atrás buscando la posición
de inserción )
i := j - espacio;
while (i > 0) and (comp_cad (nodosllave [indice [i]],
    llave_ins) > 0) DO
BEGIN
indice [i + espacio] := indice [i];
i := i - espacio
END;
indice [i + espacio] := subindice_ins
END;
espacio := espacio div 2
END
END; { clasif_shell }

BEGIN { principal }
toma_archent (fd_ent); { Abre el archivo de entrada }
toma_archsal (fd_sal); { Abre el archivo de salida }
cont_reg := encabezado. longitud; { Lee el contador de registros
                                    del encabezado }
{ Lee los registros secuencialmente, colocándolos en el arreglo
registros []. Extrae las llaves y asigna los valores a índice [] conforme
se avanza }
for i := 1 to cont_reg DO
BEGIN
lee_reg (fd_ent, buffcad);
caddat (registros [i], buffcad);
extrae_llave (nodosllave [i], buffcad);
indice [i] := i
END;
clasif_shell (indice, nodosllave, cont_reg);

{ Escribe los registros en el orden de los subíndices en el arreglo
índice[] ya clasificado }
for i := 1 to cont_reg DO
write (fd_sal, registros [indice [i]]);

close (fd_ent);
close (fd_sal)
END.

```

OBJETIVOS

Introducir conceptos de *indización* que tienen muchas aplicaciones en el diseño de sistemas de archivos.

Introducir el uso de un *índice lineal simple* para proporcionar acceso rápido a registros en un archivo de registros de longitud variable, con entradas secuenciales.

Investigar las implicaciones del uso de índices para el mantenimiento de archivos.

Describir el uso de índices para proporcionar acceso a registros mediante más de una llave.

Introducir la idea de *listas invertidas*, ilustrando *operaciones booleanas* en listas.

Analizar el problema de *cuándo enlazar* una llave índice a una dirección del archivo de datos.

Introducir e investigar las implicaciones de los archivos *autoindizados*.

INDIZACION

7

PLAN GENERAL DEL CAPITULO

- ■ ■ 7.1 ¿Qué es un índice?
- ■ ■ 7.2 Índice simple de un archivo con entradas secuenciales
- ■ ■ 7.3 Operaciones básicas en un archivo indizado con entradas secuenciales
- ■ ■ 7.4 Índices demasiado grandes para almacenarse en memoria
- ■ ■ 7.5 Indización para proporcionar acceso mediante varias llaves
- ■ ■ 7.6 Extracción de información mediante combinaciones de llaves secundarias
- ■ ■ 7.7 Mejora de la estructura secundaria de índices: listas invertidas
 - 7.7.1 Primer intento de solución
 - 7.7.2 Una solución mejor: ligar la lista de referencias
- ■ ■ 7.8 Índices selectivos
- ■ ■ 7.9 Enlace (*binding*)

7.1

¿QUE ES UN INDICE?

Las últimas páginas de un libro suelen contener un índice, es decir, una tabla que contiene una lista de temas (llaves) y números de página en donde pueden encontrarse esos temas (campos de referencia).

Todos los índices están basados en el mismo concepto básico de llaves y campos de referencia. Los tipos de índices que se examinan en este capítulo se llaman *índices simples*, porque se representan mediante *arreglos simples* de estructuras que contienen las llaves y los campos de referencia. En capítulos posteriores se examinan esquemas de indización que usan estructuras de datos más complejas, especialmente árboles. En este capítulo se hace hincapié en que los índices pueden ser muy simples, pero aun así proporcionan herramientas poderosas de procesamiento de archivos.

El índice de un libro ofrece una forma de encontrar un tema rápidamente. Si el lector alguna vez ha consultado un libro que no tiene un buen índice, sabe que un índice es una mejor alternativa mejor que tener que buscar un tema a lo largo del libro en forma secuencial. En general, la indización es otra forma de manejo del problema que se exploró en el capítulo 6; un índice es un recurso para encontrar información.

Considérese lo que sucedería si se intentara aplicar los métodos del capítulo anterior, clasificación y búsqueda binaria, al problema de encontrar información en un libro. Reacomodar todas las palabras del libro de modo que estuviesen en orden alfabético con certeza facilitaría la búsqueda de cualquier término, pero obviamente tendría efectos desastrosos en el significado del libro. En cierto sentido, los términos del libro son registros fijos. Este es un ejemplo absurdo, pero permite entender claramente el poder y la importancia del índice como herramienta conceptual. Como trabaja por indirección, *un índice permite imponer un orden en un archivo sin que realmente se reacomode*. Así no sólo se evita desacomodar los registros fijos, sino que también los costos de aspectos tales como la adición de registros son mucho menores que en un archivo clasificado.

También a manera de ejemplo, considérese el problema de encontrar libros en una biblioteca. Se espera poder localizar los libros por autor específico, por título o por materia. Una forma de lograrlo es tener tres copias de cada libro y tres edificios de biblioteca separados. Todos los libros de un edificio estarían clasificados por el nombre del autor; otro edificio contendría libros acomodados por título, y el tercero los tendría ordenados por materia. De nuevo, éste es un ejemplo absurdo, pero señala otra ventaja importante de la indización. En vez de usar distribuciones múltiples, una biblioteca usa un catálogo de tarjetas. El catálogo de tarjetas en realidad es un conjunto de tres índices; cada uno tiene un distinto pero todos tienen el mismo número de catálogo como . Por lo tanto, otro uso de la indización consiste en proporcionar a un archivo.

También se encuentra que la indización proporciona *acceso por llave a archivos de registros de longitud variable*. El análisis de la indización se iniciará con el examen del problema del acceso a registros de longitud variable, y con la solución sencilla que la indización proporciona.

7.2

INDICE SIMPLE DE UN ARCHIVO CON ENTRADAS SECUENCIALES

Supóngase que se posee un amplio conjunto de discos de música y se quiere mantener información sobre la colección por medio del uso de archivos de computador. Para cada disco, se mantiene la información que se muestra en la figura 7.1. Los registros del archivo de datos son de longitud variable. La figura 7.2 ilustra dicha colección de registros de datos. Se designa este archivo como *Archdatos*.

Número de identificación
 Título
 Compositor o compositores
 Artista o artistas
 Marca (compañía disquera)

FIGURA 7.1 • Contenido de un registro de datos.

Existen varios métodos que podrían emplearse para crear un archivo de registros de longitud variable que almacene estos registros; las direcciones de registros empleadas en la figura 7.2 sugieren que cada registro sea precedido por un campo de tamaño que permita acceso mediante saltos secuenciales y un fácil mantenimiento del archivo. Esta es la estructura que se emplea aquí.

Supóngase que se formó la *llave primaria* de estos registros considerando las iniciales de la marca de la compañía grabadora, combinadas

Reg. Dir.	Etiqueta	número ID	Título	Compositor(es)	Artista(s)
32*	LON	2312	Romeo y Julieta	Prokofiev	Maazel
77	RCA	2626	Cuarteto en do sostenido menor	Beethoven	Julliard
132	WAR	23699	Touchstone	Corea	Corea
167	ANG	3795	Sinfonía núm. 9	Beethoven	Giulini
211	COL	38358	Nebraska	Springsteen	Springsteen
256	DG	18807	Sinfonía núm. 9	Beethoven	Karajan
300	MER	75016	Suite El gallo de oro	Rimsky-Korsakov	Leinsdorf
353	COL	31809	Sinfonía núm. 9	Dvorak	Bernstein
396	DG	139201	Concierto para violín	Beethoven	Ferras
422	FF	245	Good News	Sweet Honey in the Rock	Sweet Honey in the Rock

* Supóngase que existe un registro de encabezado que usa los primeros 32 bits.

FIGURA 7.2 • Muestra del contenido de Archdatos.

con el número de identificación de compañía del disco. Esto hará una buena llave primaria, ya que debe darse una llave única para cada registro del archivo. A esta llave se le llama *ID de marca*. La forma canónica del ID de marca está compuesta por el campo de marca en mayúsculas, seguido inmediatamente por la representación ASCII del número de identificación. Por ejemplo,

LON2312

¿Cómo podría organizarse el archivo para que permita acceso rápido por llave a los registros individuales? ¿Podría clasificarse el archivo y después usar búsqueda binaria? Por desgracia, la búsqueda binaria depende de la posibilidad de saltar al registro situado en medio del archivo. Esto no es posible en un archivo de registros de longitud variable, porque no es posible el acceso directo mediante el número relativo de registro; no hay forma de saber dónde está el registro medio en algún grupo de registros.

Una alternativa a la clasificación es la construcción de un índice para el archivo; la figura 7.3 ilustra uno. A la derecha está el archivo de

Archíndices		Archdatos	
<i>Llave</i>	<i>Campo de referencia</i>	<i>Dirección de registro</i>	<i>Registro de datos actual</i>
ANG3795	167	32	LON 2312 Romeo y Julieta Prokofiev...
COL31809	353	77	RCA 2626 Cuarteto en do sostenido menor...
COL38358	211	132	WAR 23699 Touchstone Corea...
DG139201	396	167	ANG 3795 Sinfonía núm. 9 Beethoven ...
DG18807	256	211	COL 38358 Nebraska Springsteen ...
FF245	442	256	DG 18807 Sinfonía núm. 9 Beethoven ...
LON2312	32	300	MER 76016 Suite El gallo de oro Rimsky ...
MER75016	300	353	COL 31809 Sinfonía núm. 9 Dvorak ...
RCA2626	77	396	DG 139201 Concierto para violín Beethoven...
WAR23699	132	422	FF 245 Good News Sweet Honey In The...

FIGURA 7.3 • Muestra del índice con el archivo de datos correspondiente.

Se observa también que el índice esta clasificado, a diferencia del archivo de datos. En consecuencia, aunque el ID de marca ANG3795 es la primera entrada del índice, no necesariamente es la primera entrada del archivo de datos. De hecho, el archivo de datos tiene entradas secundarias, lo cual significa que los registros están en el orden en que se introducen en el archivo. Como pronto se verá, el archivo de entradas secundarias puede hacer que la adición de registros y el mantenimiento del archivo sean más sencillos que en un archivo de datos que mantienen clasificado mediante alguna llave.

Emplear el índice para proporcionar acceso a los archivos de datos mediante el ID de marca es una cuestión sencilla. Los pasos necesarios para extraer un registro con la llave LLAVE de Archdatos se muestran

La estructura del archivo de índices es muy simple; es un archivo de registros de longitud fija: un campo llave y un campo de distancia en bytes. Para cada registro del archivo de datos hay un registro en el archivo de índices.

Al lado izquierdo esta el archivo de indice, en donde cada registro contiene una llave de 12 caracteres (alineada a la izquierda) completada con espacios), que corresponde a algun ID de marca del archivo de datos. Cada llave se asocia con un campo de referencia que proporciona la dirección del primer byte del registro de datos correspondiente. Por ejemplo, ANG3795 corresponde al campo de referencia que contiene el numero 167, lo que significa que el registro que contiene la informacion completa del disco con el ID de marca ANG3795 se encuentra al inicio del byte numero 167 del archivo de registros.

Los datos que contiene la información de la colección de discos, con un registro de datos de longitud variable por disco. Se lo se muestran cuatro campos (Marca, Número de identificación, Titulo y Compositor), pero es fácil imaginar la información que completa cada registro.

FIGURA 7.4 - Recupera_registro(); procedimiento que recupera un registro de Archivos por medio de ArchIndice.

Encontrar la posición de Llave en Archíndices /* Usando probabilmente
Toma la DIST_BYTTS del registro correspondiente en Archíndices */
busquedas binaria /*
Usar SEEK() Y dist_byttes para trastadarise al registro de datos
Leer el registro en Archíndices

PROCEDIMENTO recupera_registro (LVAE)

en el procedimiento *recupera_registro()*, en la figura 7.4. Aunque esta estrategia de extracción de información es relativamente directa, tiene algunas características que ameritan comentarse:

- Ahora se trabaja con dos archivos, el archivo de datos y el archivo de índices. El archivo de índices es considerablemente más fácil de manejar que el archivo de datos, porque emplea registros de longitud fija (ésta es la razón por la cual puede usarse en él búsqueda binaria), y porque suele ser mucho más pequeño que el archivo de datos.
- Al requerir que el archivo de índices tenga registros de longitud fija, se impone un límite en los tamaños de las llaves. En este ejemplo se supone que el campo de la llave primaria es lo suficientemente grande para mantener la identidad única de cada llave. El empleo de un campo de llave pequeño y fijo en el índice puede causar problemas si la identidad única de una llave se pierde cuando se coloca en el campo fijo de índice.
- En el ejemplo, el índice no contiene más información que las llaves y los campos de referencia, pero no siempre es así. Por ejemplo, puede mantenerse la longitud de cada registro de *Archdatos* en *Archíndices*.

7.3

OPERACIONES BASICAS EN UN ARCHIVO INDIZADO CON ENTRADAS SECUENCIALES

Se ha observado que el proceso de mantener los archivos clasificados para permitir búsqueda binaria de registros puede ser muy costoso. Una de las grandes ventajas de un índice simple con un archivo de datos con entradas secuenciales es que la adición de registros es mucho más rápida que con un archivo de datos clasificado, siempre y cuando el índice sea lo suficientemente pequeño para almacenarse completo en la memoria. Cuando la longitud del registro de índices es corta, no es difícil cumplir esta condición para archivos pequeños, conformados por unos pocos miles de registros. Por el momento, en estos análisis se supone que la condición se cumple y que se lee el índice del almacenamiento secundario hacia un arreglo de estructuras llamado *INDICE[]*. Posteriormente se considera lo que debe hacerse cuando el índice es demasiado grande para caber en la memoria.

Conforme se ejecuta el programa, mantener el índice en la memoria también permite encontrar registros por llave más rápido con un archivo indizado que con uno clasificado, porque la búsqueda binaria

puede efectuarse por completo en la memoria. Una vez que se encuentra la distancia en bytes del registro de datos, todo lo que se requiere es un solo desplazamiento para recuperar el registro. Por otro lado, el uso de un archivo clasificado requiere un desplazamiento por cada paso de la búsqueda binaria.

El manejo y mantenimiento de un archivo con entradas secuenciales acoplado a un índice simple requiere el desarrollo de procedimientos para el manejo de varias tareas diferentes. Junto al algoritmo *recupera_registro()*, descrito con anterioridad, se usan otros procedimientos para encontrar datos por medio del índice, que incluyen lo siguiente:

- Crear los archivos vacíos originales de índices y datos;
- Cargar el archivo de índices en la memoria antes de usarlo;
- Reescribir el archivo de índices de la memoria después de usarlo;
- Agregar registros al archivo de datos y al índice;
- Eliminar registros del archivo de datos, y
- Actualizar registros del archivo de datos.

CREACION DE LOS ARCHIVOS. Tanto el archivo de índice como el archivo de datos se crean como archivos vacíos, con registros de encabezado y nada más. Esto puede llevarse a cabo con bastante facilidad creando los archivos y escribiendo los encabezados en ambos archivos.

CARGA DEL INDICE EN LA MEMORIA. Se supone que el archivo de índice es lo suficientemente pequeño para caber en la memoria primaria, de tal suerte que se define un arreglo *INDICE[]* para almacenar los registros de índice. Cada elemento del arreglo tiene la estructura de un registro de índice. Por consiguiente, la carga del archivo de índices en la memoria es sólo cuestión de leer y guardar el registro de encabezado del índice, y después leer los registros del archivo de índice en el arreglo *INDICE[]*. Como esto será una lectura secuencial, y puesto que los registros son pequeños, el procedimiento debe escribirse de tal forma que se lea un número grande de registros de índice a la vez, y no uno por uno. ¿Se entiende por qué?

REESCRITURA DEL ARCHIVO DE INDICES DE LA MEMORIA. Cuando se termina el procesamiento de un archivo indizado es necesario reescribir *INDICE[]* de vuelta en el archivo de índices, si el arreglo ha cambiado en alguna forma. En la figura 7.5, el procedimiento *reescribe_index()* describe los pasos que se deben seguir.

Es importante considerar lo que sucede si esta reescritura del índice no se realiza o se efectúa en forma incompleta. El hecho es que

PROCEDIMIENTO reescribe_index()

 Revisar la bandera de status que dice si el arreglo INDICE[]
 ha cambiado en alguna forma.

 si hubo cambios, entonces

 Abrir el archivo de índice como un archivo nuevo vacío

 Actualizar el registro de encabezado y reescribir el encabezado

 Escribir el índice al archivo recién creado

 Cerrar el archivo de índice

fin PROCEDIMIENTO

FIGURA 7.5 • El procedimiento *reescribe_index()*.

los programas no siempre se ejecutan hasta que terminan. El diseñador de programas necesita protegerse de las fallas de energía, de que el operador apague la máquina en el momento equivocado y de otros desastres similares. Uno de los peligros serios asociados con la lectura de un índice en la memoria y la escritura posterior, cuando el programa termina, es que si el programa se interrumpe, la copia del índice en el disco estará desactualizada e incorrecta. Es imperativo que un programa contenga al menos las siguientes dos defensas contra esta clase de error:

- Debe existir un mecanismo que permita al programa saber cuándo el índice no está actualizado. Una posibilidad consiste en activar una bandera de estado tan pronto como la copia del índice en la memoria cambie. Esta bandera de estado se puede escribir en el registro de encabezado del archivo de índices del disco en cuanto se haya leído el índice en la memoria, y se puede desactivar después cuando el índice se haya reescrito. Todos los programas podrían revisar la bandera de estado antes de usar un índice. Si se encuentra que la bandera está activa, entonces el programa sabrá que el índice no está actualizado.
- Si un programa detecta que un índice no está actualizado, debe acceder a un procedimiento que reconstruya el índice a partir del archivo de datos. Esto debe suceder automáticamente, antes de cualquier intento de usar el índice.

ADICION DE REGISTROS. Agregar un registro nuevo al archivo de datos requiere que también se agregue un registro al archivo de índices.

La adición al archivo de datos por sí misma es fácil. Por supuesto, el procedimiento preciso depende del tipo de organización de registros de longitud variable que se use. En cualquier caso, cuando se agrega un registro de datos se debe conocer la distancia en bytes inicial de la posición física del archivo donde se escribió el registro. Esta información debe colocarse en el arreglo INDICE[], junto con la forma canónica de la llave del registro.

Como el arreglo INDICE[] se mantiene en orden clasificado por llaves, la inserción de índices nuevos probablemente requerirá algún reacomodo del índice. En cierta forma, la situación es similar a la de agregar registros a un archivo de datos clasificado: tienen que recorrerse o moverse todos los registros cuyas llaves están en orden después de la llave del registro que se inserta. Esto abre un espacio al registro nuevo. La gran diferencia entre el trabajo requerido por los registros del índice y el requerido por un archivo de datos clasificado es que el arreglo INDICE[] está contenido *por completo en la memoria*. Todo el reacomodo de los índices puede hacerse sin ejecutar ningún acceso al archivo.

Un aspecto problemático de este procedimiento directo es el tiempo que toma recorrer los registros de los índices. En el capítulo 6 se presenta el mismo problema cuando se reacomoda el vector de llaves y apuntadores en el programa *clasifram*. Se recordará que se evita mover realmente las estructuras que contienen las llaves formando un vector de apuntadores a las estructuras que las contienen. Después se reacomoda el vector de apuntadores en lugar de las estructuras mismas. Puede desarrollarse un mecanismo similar para el mantenimiento del vector del archivo de índices.

ELIMINACION DE REGISTROS. En el capítulo 5 se describen varios métodos para la eliminación de registros en archivos con registros de longitud variable, para reutilizar el espacio ocupado por ellos. Estos métodos son completamente viables para nuestro archivo de datos, ya que, a diferencia de un archivo de datos clasificado, los registros de este archivo no necesitan moverse para mantener el orden. Esta es una de las grandes ventajas de una organización de archivo indizada: se accede rápido a los registros individuales por llave sin la molestia de tener registros fijos. De hecho, la indización por sí misma fija todos los registros.

Por supuesto, cuando se elimina un registro del archivo de datos, también debe eliminarse el registro correspondiente del archivo de índices. Como el índice está contenido en un arreglo durante la ejecución del programa, la eliminación del registro de índice y el desplazamiento de los demás registros para agrupar el espacio puede ser una operación no muy costosa. Una vez más, existen oportunidades de hacer este reacomodo aún más rápido, empleando alguna indirección adicional por medio del uso de un vector de apuntadores a estructuras. Una

alternativa es simplemente marcar como eliminado el registro de índice, al igual que puede marcarse el registro de datos correspondiente.

ACTUALIZACION DE REGISTROS. La actualización de registros entra en dos categorías:

- La actualización cambia el valor del campo de llave.* Esta clase de actualización puede traer consigo un reacomodo del archivo de índices, así como del de datos. Conceptualmente, la forma más fácil de concebir esta clase de cambio es como una eliminación seguida de una adición. Puede implantarse este método de eliminar y agregar, y dar al usuario del programa la impresión de que simplemente está cambiando un registro.
- La actualización no afecta el campo de la llave.* Este segundo tipo de actualización no requiere reacomodo del archivo de índices, pero bien puede implicar el reacomodo del archivo de datos. Si el tamaño del registro no cambia, o si disminuye por la actualización, el registro puede escribirse directamente en el espacio que tenía, pero si aumenta por la actualización, se tendrá que encontrar una nueva entrada para el registro. En el último caso, la dirección de inicio del registro reescrito debe reemplazar la dirección antigua en el campo *dist_bytes*, del registro de índice correspondiente.

7.4

INDICES DEMASIADO GRANDES PARA ALMACENARSE EN MEMORIA

Los métodos analizados y, por desgracia, muchas de las ventajas que presentan, están sujetos a la suposición de que el archivo de índices es lo suficientemente pequeño para cargarse por completo en memoria. Si el índice es demasiado grande como para que este método resulte práctico, entonces el acceso y el mantenimiento del índice deben hacerse en el almacenamiento secundario. Con índices simples del tipo que se ha analizado, el acceso al disco tiene las siguientes desventajas:

- La búsqueda binaria del índice requiere varios desplazamientos en lugar de realizarse a las velocidades electrónicas de la memoria. La búsqueda binaria de un índice en el almacenamiento secundario no es sustancialmente más rápida que la búsqueda binaria en un archivo clasificado.

- El reacomodo de índices debido a la adición o eliminación de registros requiere mover o clasificar registros en el almacenamiento secundario. Estas mismas operaciones son, literalmente, millones de veces más costosas que si se realizan en la memoria electrónica.

Aunque estos problemas no son peores que los asociados con el uso de cualquier archivo que se clasifica por llave, son lo suficientemente severos para justificar la consideración de alternativas. Cuando un índice simple es demasiado grande para almacenarse en memoria, debe considerarse el uso de:

- una organización por *dispersión*, cuando la velocidad de acceso tiene la máxima prioridad, o
- un índice *estructurado en forma de árbol*, como un árbol B, cuando se necesita flexibilidad tanto en el acceso por llave como en el acceso ordenado y secuencial.

Estas alternativas de organización de archivos se analizan con amplitud en los siguientes capítulos. Pero, antes de descartar el uso de índices simples en el almacenamiento secundario, debe observarse que proporcionan algunas ventajas importantes que no tiene el uso de archivos clasificados por llave, aun si el índice no puede almacenarse en memoria:

- Un índice simple hace posible la búsqueda binaria para obtener acceso por llave a un registro en un archivo de registros de longitud variable. El índice proporciona el servicio de asociar un registro de longitud fija, donde por consiguiente, se pueda usar la búsqueda binaria con cada registro de datos de longitud variable.
- Si los registros de índices son considerablemente más pequeños que los registros de datos del archivo, la clasificación y el mantenimiento del índice pueden ser menos costosos de lo que sería la clasificación y el mantenimiento del archivo de datos. Esto se debe simplemente a que existe menos información que mover en el archivo de índices.
- Si hay registros fijos en el archivo de datos, el uso de un índice permite reacomodar las llaves sin tener que mover los registros de datos.

Existe otra ventaja asociada con el uso de índices simples, que no se ha analizado aún. Esta, por sí misma, puede ser una razón suficiente para emplear índices simples, aunque no entren por completo en la

memoria. Se recordará la analogía entre un índice y un catálogo de tarjetas de biblioteca: el catálogo de tarjetas proporciona varias disposiciones o formas de enfocar la colección de la biblioteca, aunque sólo existe un conjunto de libros acomodados en orden. De manera similar, pueden emplearse varios índices para proporcionar diversas formas de ver un archivo de datos.

7.5

INDIZACION PARA PROPORCIONAR ACCESO MEDIANTE VARIAS LLAVES

Una pregunta razonable en este punto es: "Todo este asunto de la indización es muy interesante, pero ¿quién querría encontrar un disco mediante la llave COL38358, si lo que se quiere es el disco *Nebraska*, de Bruce Springsteen?"

Volvamos a la analogía entre un índice y un catálogo de tarjetas de biblioteca. Supóngase que se concibe la llave primaria, ID de marca, como una especie de número de catálogo. Como con el número de catálogo asignado a un libro, hay que cuidarse de hacer el ID de marca único. Ahora bien, en una biblioteca es poco usual comenzar buscando un libro por un número de catálogo en particular (p. ej., "Busco un libro con el número de catálogo QA331T5 1959."). Por lo general se empieza buscando un libro por una materia en particular, por un título en particular, o por un autor dado (p. ej., "Busco un libro sobre funciones"; o "Busco *The Theory of Functions* de Titchmarsh."). Dada la materia, el autor o el título, se busca en el catálogo de tarjetas para encontrar la *llave primaria*, el número de catálogo.

En forma similar, se podría construir, para la colección de discos que se exemplifica, un catálogo que contenga el título del álbum, compositor y artista. Estos campos son *campos de llave secundaria*. Así como el catálogo de la biblioteca relaciona un autor (llave secundaria) con un número del catálogo de tarjetas (llave primaria), se puede construir un archivo de índices que relacione al compositor con el ID de marca, como se ilustra en la figura 7.6.

Además de las similitudes, existe una diferencia importante entre este tipo de índices de llave secundaria y el catálogo de tarjetas en una biblioteca. En una biblioteca, una vez que se tiene el número de catálogo, por lo regular puede irse directamente a los estantes para encontrar el libro, ya que los libros están acomodados en el orden del número de catálogo. En otras palabras, los libros están clasificados por la llave primaria. En cambio, los registros reales del archivo del ejemplo están en orden de *entrada secuencial*. Por tanto, después de consultar

Indice de compositores

Llave secundaria Llave primaria

BEETHOVEN	ANG3795
BEETHOVEN	DG139201
BEETHOVEN	DG18807
BEETHOVEN	RCA2626
COREA	WAR23699
DVORAK	COL31809
PROKOFIEV	LON2312
RIMSKY-KORSAKOV	MER75016
SPRINGSTEEN	COL38358
SWEET HONEY IN THE R	FF245

FIGURA 7.6 • Indice de llaves secundarias organizado por compositor.

el índice de compositores para encontrar el ID de marca, debe consultarse un índice adicional, el índice de la llave primaria, para encontrar la distancia en bytes real del registro que tiene este particular ID de marca. El procedimiento se resume en la figura 7.7.

Está claro que se pueden relacionar las referencias de llave secundaria (p. ej., Springsteen) directamente con una distancia en bytes (211), en lugar de hacerlo a una llave primaria (COL38358). Sin embargo, existen buenas razones para posponer tanto como sea posible este enlazamiento de una llave secundaria con una dirección específica. Estas razones se aclararán conforme se analice la forma en que el uso de índices secundarios afecta a las operaciones fundamentales de archivos, como la eliminación y modificación de registros.

ADICION DE REGISTROS. Cuando un índice secundario está presente, agregar un registro al archivo significa agregar un registro al índice secundario. El costo que implica esto es similar al que representa agregar un registro al índice primario: se necesita mover los regis-

PROCEDIMIENTO búsqueda_en_secundario(LLAVE)

Buscar LLAVE en el índice secundario

Cuando se ha encontrado el índice secundario correcto, asignar a ID_MARCA el valor de la llave primaria en el campo de referencia del registro

Llamar a recupera_registro(ID_MARCA) para obtener el registro de datos

fin PROCEDIMIENTO

FIGURA 7.7 • *Búsqueda_en_secundario()*: algoritmo para extraer un registro individual de Archdatos por medio de un índice de llave secundaria.

tros o reacomodar un vector de apuntadores a las estructuras. Como con los índices primarios, el costo disminuye mucho si el índice secundario puede trasladarse a la memoria electrónica y modificarse allí.

Nótese que el campo llave en el archivo de índices secundario se almacena en forma canónica (todos los nombres de los compositores están en mayúsculas), ya que ésta es la forma en que se desea consultar el índice secundario. Si se quiere imprimir el nombre en forma normal, combinando mayúsculas y minúsculas, puede tomarse esa forma del archivo de datos original. También obsérvese que las llaves secundarias se almacenan a una longitud fija, lo cual significa que algunas veces se truncan. La definición de la forma canónica debe tomar en cuenta esta restricción de longitud para que la búsqueda del índice trabaje adecuadamente.

Una diferencia importante entre un índice secundario y uno primario es que un índice secundario puede contener llaves duplicadas. En el índice del ejemplo ilustrado en la figura 7.6 hay cuatro registros con la llave BEETHOVEN. Por supuesto, las llaves duplicadas se agrupan. Dentro de este grupo deben estar ordenadas de acuerdo con los valores de los campos de referencia. En el ejemplo, esto significa colocarlos en orden de ID de marca. Las razones de este segundo nivel de clasificación se esclarecerán posteriormente, conforme se analice la extracción de información con base en combinaciones de dos o más llaves secundarias.

ELIMINACION DE REGISTROS. Por lo general, la eliminación de un registro implica la eliminación de todas las referencias a ese registro en el sistema de archivos, de modo que eliminar un registro del archivo de datos significa la eliminación no sólo del registro correspondiente en el

índice primario, sino también de todos los registros de los índices secundarios que hacen referencia a este registro del índice primario. El problema es que los índices secundarios, como los primarios, se mantienen clasificados en el orden por llave. Por ello, la eliminación de un registro puede implicar el reacomodo de los registros restantes, para cerrar el espacio abierto dejado por la eliminación.

Este método de eliminación de todas las referencias ciertamente puede ser aconsejable si el índice secundario hace referencia directamente al archivo de datos. Si no se eliminaran las referencias de las llaves secundarias, y si éstas estuvieran asociadas con las distancias en bytes reales del archivo de datos, sería difícil decir cuándo dejan de ser válidas las referencias. Este es otro ejemplo del problema de los registros fijos. Los campos de referencia asociados con las llaves secundarias apuntarían a una distancia en bytes que podría estar asociada con registros de datos diferentes, después de la eliminación y la subsecuente reutilización del espacio en el archivo de datos.

Pero en este ejemplo se ha evitado hacer referencia a las direcciones reales en el índice de llaves secundarias. Después de encontrar la llave secundaria, se hace otra búsqueda, esta vez de la llave primaria. Dado que el índice primario *refleja el cambio* ocasionado por la eliminación de registros, la búsqueda de la llave primaria de un registro que se ha eliminado fallará, devolviendo un estado de registro no encontrado. En cierto sentido, el índice actualizado de la llave primaria actúa como una especie de revisión final, evitando la extracción de registros que no existen.

En consecuencia, una opción que queda abierta cuando se borra un registro del archivo de datos es modificar y reacomodar el índice de llaves primarias. Con seguridad podrían dejarse intactas las referencias al registro eliminado que existen en los índices de llaves secundarias. Las búsquedas que se inician desde un índice de llaves secundarias y que conducen a un registro eliminado se captan cuando se consulta el índice de llaves primarias.

Si existen varios índices de llaves secundarias, se pueden lograr considerables ahorros al no tener que reacomodarlos todos cuando se elimina un registro. Esto es importante en especial cuando los índices de llaves secundarias se mantienen en almacenamiento secundario, y lo es también en un sistema interactivo, donde el usuario espera en una terminal hasta que la operación de eliminación termina.

Por supuesto, existe un costo asociado con este ahorro: los registros eliminados ocupan lugar en los archivos de índices secundarios. En un sistema de archivos sujeto a pocas eliminaciones, en general esto no es un problema. Con una estructura de archivos un poco más volátil es posible manejar el problema eliminando periódicamente de los archivos de índices secundarios todos los registros cuyas referencias ya no

existen en el índice primario. Si un sistema de archivos es tan volátil que incluso la eliminación periódica resulta inadecuada, tal vez sea el momento de considerar otra estructura de índices, como el árbol B, que permite eliminaciones sin tener que reacomodar una gran cantidad de registros.

ACTUALIZACION DE REGISTROS. En el análisis sobre la eliminación de registros se encontró que el índice de llaves primarias sirve como una especie de buffer de protección que aísla los índices secundarios de los cambios en el archivo de datos. Este aislamiento se extiende también a la actualización de archivos. Si los índices secundarios contienen referencias directas a las distancias en bytes del archivo de datos, entonces las actualizaciones que ocasionen un cambio de la posición física del registro en el archivo, también implican actualizar los índices secundarios. Pero, ya que se confina dicha información detallada al índice primario, las actualizaciones al archivo de datos afectan el índice secundario sólo cuando cambian la llave primaria o la secundaria. Existen tres posibles situaciones:

- La actualización cambia la llave secundaria:* Si se cambia la llave secundaria, probablemente habrá que reacomodar el índice secundario de tal forma que permanezca en el orden de clasificación. Esto puede ser una operación relativamente costosa.
- La actualización cambia la llave primaria:* Este tipo de cambio tiene gran repercusión en el índice de la llave primaria, pero con frecuencia sólo requiere que se actualice el campo de referencia afectado (*ID_marca* en el ejemplo) en todos los índices secundarios. Esto implica buscar los índices secundarios (en las llaves secundarias que no se cambiaron) y reescribir los campos de longitud fija afectados. No se requiere la reclasificación de los índices secundarios, a menos que la llave secundaria correspondiente aparezca más de una vez en el índice. Si una llave secundaria aparece más de una vez, puede haber una reclasificación local, ya que los registros que tienen la misma llave secundaria están ordenados según el campo de referencia (llave primaria).
- La actualización se restringe a los otros campos.* Las actualizaciones que no afectan los campos de la llave primaria o secundaria no afectan el índice de llaves secundarias, aunque la actualización sea considerable. Nótese que si existen varios índices de llaves secundarias asociados con un archivo, la actualización de registros suele afectar sólo un subconjunto de los índices secundarios.

7.6**EXTRACCION DE INFORMACION MEDIANTE COMBINACIONES DE LLAVES SECUNDARIAS**

Una de las aplicaciones más importantes de las llaves secundarias se relaciona con el uso de dos o más de ellas combinadas para la extracción de subconjuntos especiales de registros del archivo de datos. Para dar un ejemplo de cómo puede hacerse esto, se extraerá otro índice de llave secundaria del archivo de discos. Este índice usa el *título* del disco como llave, según se ilustra en la figura 7.8. Así puede responderse a solicitudes tales como:

- Encontrar el disco con el ID de marca COL38358 (acceso por llave primaria);

477105

Indice de compositores

<i>Llave secundaria</i>	<i>Llave primaria</i>
SUITE EL GALLO DE ORO	MER75016
GOOD NEWS	FF245
NEBRASKA	COL38358
CUARTETO EN DO SOSTENIDO MENOR	RCA2626
ROMEO Y JULIETA	LON2312
SINFONIA núm. 9	ANG3795
SINFONIA núm. 9	COL31809
SINFONIA núm. 9	DG18807
TOUCHSTONE	WAR23699
CONCIERTO PARA VIOLIN	DG139201

FIGURA 7.8 • Indice de llaves secundarias organizado por título de disco.

- Encontrar todos los discos de composiciones de Beethoven (llave secundaria-compositor); y
- Encontrar todos los discos intitulados "Concierto para violín" (llave secundaria-título).

Sin embargo, lo más interesante es que también puede responderse a una solicitud que *combine* la extracción del índice del compositor con la extracción del índice del título, como: encontrar todos los discos de la Sinfonía núm. 9 de Beethoven. Si no se utilizaran índices secundarios, este tipo de solicitud requeriría una búsqueda secuencial en el archivo completo. En un archivo que contenga miles, o incluso sólo cientos de registros, éste es un proceso muy costoso. Pero con la ayuda de índices secundarios, la respuesta a esta solicitud es sencilla y rápida.

Se empezará reconociendo que esta solicitud puede reescribirse como una operación de *conjunción booleana*, especificando la intersección de dos subconjuntos del archivo de datos:

Encontrar todos los registros con:

compositor = "BEETHOVEN" Y título = "SINFONIA NUM. 9"

Se empieza a responder esta solicitud buscando en el índice del compositor la lista de los ID de marca, que identifican los discos donde Beethoven aparece como compositor. (Un ejercicio al final de este capítulo describe un procedimiento de búsqueda binaria que puede usarse para este tipo de consulta de información.) Esto da como resultado la siguiente lista de ID de marca:

ANG3795
DG139201
DG18807
RCA2626

En seguida se buscan en el índice de títulos los ID de marca asociados con los registros que tienen como llave de título SINFONIA NUM. 9:

ANG3795
COL31809
DG18807

Ahora se efectúa la conjunción booleana, que es una operación de pareamiento, combinando las listas de tal forma que sólo los miembros que aparecen en *ambas* listas se colocan en la lista de salida.

Compositores	Títulos	Lista de pares
ANG3795	ANG3795	ANG3795
DG139201	COL31809	DG18807
DG18807	DG18807	RCA2626
RCA2626		

En el capítulo 8 se pone especial atención a los algoritmos que efectúan este tipo de operación de correspondencia. Nótese que este tipo de correspondencia es mucho más fácil si las listas que se combinan están en algún orden de clasificación. Esta es la razón por la cual, cuando se tiene más de una entrada para una determinada llave secundaria, los registros se ordenan según los campos de referencia de la llave primaria.

Para terminar, una vez que se tiene la lista de llaves primarias que existen en ambas listas, se procede a obtener del índice de las llaves primarias las direcciones de los registros del archivo de datos. Entonces pueden extraerse los registros:

ANG ; 3795 ; Sinfonía núm. 9 ; Beethoven ; Giulini
 DG ; 18807 ; Sinfonía núm. 9 ; Beethoven ; Karajan

Este es el tipo de operaciones en las que la utilidad de los sistemas de archivos indizados en computador rebasa por mucho las posibilidades de los sistemas manuales. Se tiene sólo una copia de cada registro del archivo de datos, y aun así, al trabajar con índices secundarios, existen múltiples vistas de estos registros: se pueden buscar en orden por título, por compositor, o por cualquier otro campo de interés. Al emplear la capacidad del computador para combinar rápidamente las listas clasificadas, se pueden combinar incluso otras formas de enfocar los registros, extrayendo *intersecciones* (Beethoven Y Sinfonía núm. 9) o *uniones* (Beethoven O Prokofiev O Sinfonía núm. 9) de esos enfoques. Y puesto que el archivo de datos tiene entradas secuenciales, puede hacerse todo esto sin tener que clasificar los registros del archivo de datos, restringiendo la clasificación a los registros de índice, que son más pequeños y suelen almacenarse en la memoria electrónica.

Ahora que se tiene una idea general del diseño y usos de los índices secundarios, se pueden considerar las formas de mejorarlo, de modo que ocupen menos espacio y requieran menos clasificación.

7.7

MEJORA DE LA ESTRUCTURA SECUNDARIA DE INDICES: LISTAS INVERTIDAS

Las estructuras de índices secundarios que se han desarrollado hasta aquí ocasionan dos problemas:

- El archivo de índices tiene que reacomodarse *cada vez* que se agrega un registro nuevo al archivo, aun cuando el registro nuevo sea de una llave secundaria que ya existe. Por ejemplo, si se agrega otro disco de la Sinfonía No. 9 de Beethoven a la colección, tanto el índice de compositor como el de título tendrían que reacomodarse, aunque ambos ya contengan entradas para las llaves secundarias (pero no los ID de marca) que se están agregando.
- Si hay llaves secundarias duplicadas, el campo de llave secundaria se repite para cada entrada. Esto desperdicia espacio, ya que los archivos se agrandan más de lo necesario. Los archivos de índices grandes tienen menos posibilidades de entrar en la memoria electrónica.

7.7.1 PRIMER INTENTO DE SOLUCION

Una solución simple a estos problemas es cambiar la estructura del índice secundario, de tal forma que se asocie un *arreglo* de referencias con cada llave secundaria. Por ejemplo, puede usarse una estructura de registro que permita asociar hasta cuatro campos de referencias ID de marca con una llave secundaria, como en:

BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
-----------	---------	----------	---------	---------

La figura 7.9 proporciona un ejemplo esquemático de cómo se vería dicho índice si se usara con el archivo de datos de ejemplo.

La principal contribución de esta estructura de índices se orienta a la solución del primer problema: la necesidad de reacomodar el índice cada vez que se agrega un registro nuevo al archivo de datos. Al examinar la figura 7.9 puede observarse que la adición de otro disco de una obra de Prokofiev no requiere la adición de otro registro al índice. Por ejemplo, si se agrega el disco

ANG 36193 Conciertos para piano 3 y 5 Prokofiev Francois

sólo necesita modificarse el registro correspondiente en el índice secundario insertando un segundo ID de marca:

PROKOFIEV	ANG36193	LON2312
-----------	----------	---------

Indice revisado de compositores

<i>Llave secundaria</i>	<i>Conjunto de referencias a llaves primarias</i>			
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
PROKOFIEV	LON2312			
RIMSKY-KORSAKOV	MER75016			
SPRINGSTEEN	COL38358			
SWEET HONEY IN THE R	FF245			

FIGURA 7.9 • Indice de llaves secundarias que contiene espacio para referencias múltiples para cada llave secundaria.

Como no se agrega otro registro al índice secundario, no hay necesidad de reacomodar nada. Todo lo que se requiere es reacomodar los campos en el registro existente para Prokofiev.

Aunque esta nueva estructura ayuda a evitar la necesidad de reacomodar el archivo de índices secundarios con tanta frecuencia, tiene algunos problemas. Por un lado, sólo proporciona espacio para cuatro ID de marca asociados con una llave. En el probable caso de que hubiera más de cuatro ID de marca con alguna llave, se necesitaría un mecanismo que mantuviera la información de los ID de marca adicionales.

Un segundo problema se refiere al uso del espacio. Aunque la estructura ayuda a evitar el desperdicio de espacio ocasionado por la repetición de llaves idénticas, estos ahorros de espacio tienen un costo potencialmente alto. Al ampliar la longitud fija de cada uno de los registros de índice secundario para almacenar más campos de referencias, se puede perder fácilmente más espacio por fragmentación interna que el que se ganó por no repetir las llaves idénticas.

Ya que no se quiere desperdiciar más espacio de lo necesario, es preciso preguntarse cómo se podría mejorar esta estructura de registros. Lo ideal sería desarrollar un diseño nuevo, una revisión de la revisión, que

- retenga la atractiva característica de no requerir la reorganización de los índices secundarios por cada nuevo registro que se introduzca en el archivo de datos.

- permita que más de cuatro llaves de ID de marca se asocien con cada llave secundaria, y
- elimine el desperdicio de espacio ocasionado por fragmentación interna.

7.7.2 UNA MEJOR SOLUCION: LIGAR LA LISTA DE REFERENCIAS

A los archivos que son como los índices secundarios, en los que una llave secundaria lleva a un conjunto de una o más llaves primarias, se les llama *listas invertidas*. El sentido en el que se invierte una lista debe quedar claro si se considera que se trabaja retrocediendo de una llave secundaria a la llave primaria y al registro mismo.

Como el nombre lista invertida indica, se trata de una *lista* de referencias de llaves primarias. El índice secundario revisado, que reúne varias ID de marca para cada llave secundaria, refleja este aspecto de la lista de datos de manera más directa que el índice secundario inicial. Otra forma de concebir esta característica de lista en la lista invertida se ilustra en la figura 7.10.

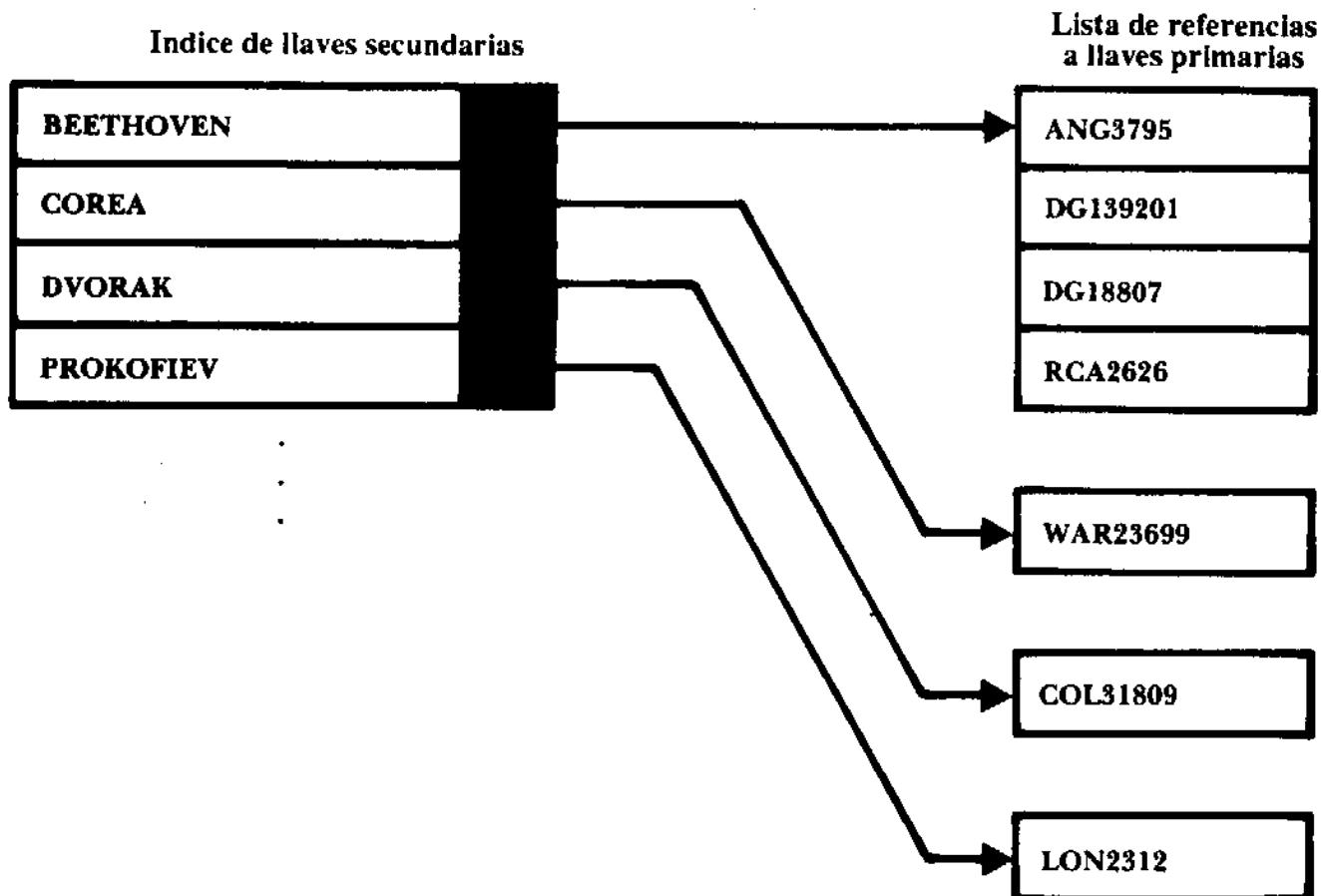


FIGURA 7.10 • Forma conceptual de los campos de referencia a llaves primarias como una serie de listas.

Como se muestra en dicha figura, una situación ideal sería hacer que cada llave secundaria apuntara a una lista diferente de referencias de llaves primarias. Cada una de estas listas podría crecer hasta tener la longitud que se necesite. Si se añade el nuevo registro de Prokofiev, la lista de referencias a Prokofiev se convierte en



Asimismo, al añadir dos nuevos registros para Beethoven se suman únicamente dos elementos adicionales a la lista de referencias asociadas con la llave Beethoven. A diferencia de la estructura de registros que asigna suficiente espacio para cuatro ID de marca para cada llave secundaria, las listas podrían contener cientos de referencias, si fuera necesario, y seguir requiriendo sólo una instancia de llave secundaria. Por otra parte, si una lista requiere sólo un elemento, entonces no se pierde espacio por fragmentación interna. Lo más importante es que sólo se necesita reacomodar el archivo de llaves secundarias cuando se añade un compositor nuevo al archivo.

¿Cómo puede establecerse un número ilimitado de listas diferentes, cada una de longitud variable, sin crear un gran número de archivos pequeños? La manera más sencilla es a través de listas ligadas. Podría redefinirse el índice secundario de modo que conste de registros con dos campos: un campo de llave secundaria y un campo que contenga el número relativo de registro de la primera referencia a la llave primaria correspondiente (ID de marca) en la lista invertida. Las referencias reales de las llaves primarias asociadas con cada llave secundaria podrían almacenarse en un archivo secuencial separado.

Con los datos de muestra con que se ha trabajado, este nuevo diseño daría lugar a un archivo de llaves secundarias para compositores y a un archivo asociado de ID de marca, que se organiza como se ilustra en la figura 7.11. Seguir las ligas para la lista de referencias asociadas con Beethoven ayuda a entender cómo está organizado el archivo de listas de ID de marca. Por supuesto, se inicia buscando Beethoven en el índice de llaves secundarias de compositores. El registro que se encuentra apunta hacia el número relativo de registro (NRR) 3 en el archivo de listas de ID de marca. Como éste es un archivo de longitud fija, es fácil saltar al NRR 3 y leer ahí la ID de marca (ANG3795). A este ID de marca se asocia una liga al registro con NRR 8. Se lee el ID de marca para ese registro y se añade a la lista (ANG3795 DG139201). Se continúa siguiendo las ligas y reuniendo los ID de marca hasta que la lista se ve así:

Revisión mejorada del índice de compositores

<i>Archivo del índice secundario</i>		<i>Archivo de listas de ID de marca</i>
0	BEETHOVEN	3
1	COREA	2
2	DVORAK	7
3	PROKOFIEV	10
4	RIMSKY-KORSAKOV	6
5	SPRINGSTEEN	4
6	SWEET HONEY IN THE R	9
0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	8
4	COL38358	-1
5	DG18807	1
6	MER75016	-1
7	COL31809	-1
8	DG139201	5
9	FF245	-1
10	ANG36193	0

FIGURA 7.11- Índice de llaves secundarias que hacen referencia a listas ligadas de referencias a llaves primarias.

ANG3795

DG139201

DG18807

RCA2626

El campo de liga en el último registro leído del archivo de listas de ID de marca contiene el valor -1. Como en los primeros programas, esto indica el final de la lista, por lo que se sabe que se tienen todas las referencias al ID de marca de los registros para Beethoven.

Para ilustrar cómo la adición de registros afecta el índice secundario y los archivos de listas de ID de marca, se añade la grabación de Prokofiev antes mencionada:

ANG 36193 Conciertos para piano 3 y 5 Prokofiev Francois

Obsérvese (Fig. 7.11) que el ID de marca para esta nueva grabación es el último en el archivo de listas de ID de marca, ya que este archivo es de entradas secuenciales. Antes de añadir este registro, existe sólo una grabación de Prokofiev, que tiene un ID de marca LON2312. Como

se desea mantener las listas de ID de marca en orden mediante los valores de los caracteres ASCII, la nueva grabación se inserta en la lista para Prokofiev de forma que preceda lógicamente al disco LON2312.

Asociar el archivo de índice secundario a un nuevo archivo que contenga listas ligadas de referencias proporciona algunas ventajas sobre las estructuras consideradas hasta este punto:

- El único momento en que se requiere reacomodar el archivo de índices secundarios es cuando se añade un nuevo nombre de compositor o se cambia el nombre de alguno ya existente (p. ej., si estuvo mal escrito). Borrar o añadir grabaciones para un compositor que ya se encuentra en el índice implica cambiar sólo el archivo de listas de ID de marca. Se podría borrar *todas* las grabaciones de un compositor modificando el archivo de listas de ID de marca, dejando el registro en el archivo de índice secundario en su lugar y utilizando el valor -1 en su campo de referencia para indicar que la lista de entradas para este compositor está vacía.
- En caso de que sea necesario reacomodar el archivo de índices secundarios, la tarea es ahora más rápida, ya que existen menos registros y cada registro es más pequeño.
- Puesto que hay menos necesidad de clasificaciones, es menor el precio que hay que pagar por mantener los archivos de índices secundarios en el almacenamiento secundario, por lo que queda más lugar en la memoria RAM para otras estructuras de datos.
- El archivo de listas de ID de marca es de entradas secuenciales. Esto significa que *nunca* necesitará clasificarse.
- Como el archivo de listas de ID de marca es de registros de longitud fija, sería muy fácil desarrollar un mecanismo para reutilizar el espacio de los registros eliminados, como se describe en el capítulo 5.

También existe, al menos, una desventaja potencialmente significativa para este tipo de organizaciones de archivo: ya no se garantiza que los ID de marca asociados con un compositor dado estén agrupados físicamente. El término técnico para dicho agrupamiento es *localidad*. En una estructura ligada y secuencial como ésta, es poco probable que haya localidad en los agrupamientos lógicos de los campos de referencia para una determinada llave secundaria. Por ejemplo, nótese que la lista de ID de marca para Prokofiev se compone de los últimos y los primeros registros del archivo. Esta falta de localidad significa que recuperar las referencias de un compositor que tenga una lista larga de referencias implicaría una gran cantidad de *desplazamientos* en el disco. Nótese que esto no sería necesario para la estructura original del archivo de índices secundarios.

Un antídoto obvio para este problema de desplazamientos es mantener el archivo de listas de ID de marca en la memoria. Esto podría ser costoso y poco práctico, considerando la cantidad de índices secundarios, excepto por la interesante posibilidad de usar el mismo archivo de listas de ID de marca para almacenar las listas de varios archivos de índices secundarios. Aunque el archivo de listas de referencias fuese demasiado grande para almacenarse en la memoria, puede obtenerse una mejora en el desempeño si se tiene sólo una parte del archivo en la memoria a la vez, manejando por páginas las secciones del archivo dentro y fuera de la memoria, conforme se necesiten.

Varios ejercicios al final del capítulo exploran estas posibilidades en forma más completa. Estos problemas son muy importantes, porque la noción de dividir el índice en páginas es fundamental para el diseño de árboles B y otros métodos para el manejo de índices grandes en almacenamiento secundario.

7.8

INDICES SELECTIVOS

Otra característica interesante de los índices secundarios es que pueden usarse para dividir un archivo en partes, proporcionando un enfoque selectivo. Por ejemplo, es posible construir un *índice selectivo* que contenga sólo los títulos de los discos de música clásica en la colección. Si se tiene información adicional acerca de los discos en el archivo de datos, como la fecha en que apareció el disco, podrían construirse índices selectivos como "discos que aparecieron antes de 1970" y "discos posteriores a 1970". Dicha información de índices selectivos podría combinarse en operaciones de conjunción booleana para responder a solicitudes tales como "la lista de todos los discos de la Sinfonía núm. 9 de Beethoven que han aparecido desde 1970". Algunas veces, los índices selectivos son útiles cuando el contenido del archivo de datos entra, de manera natural y lógica, en varias categorías generales.

7.9

ENLACE (BINDING)

Una cuestión recurrente y muy importante en el diseño de sistemas de archivos que utilizan índices es: *¿En qué momento se liga la llave a la dirección física de su registro asociado?*

En el sistema de archivos que se está diseñando en este capítulo, el *enlace de la llave primaria con una dirección se llevó a cabo en el momento en que se construyen los archivos*. Por otro lado, las llaves secundarias se ligan a una dirección *en el momento en que se usan*.

El enlace que se realiza en el momento de la construcción del archivo permite un acceso más rápido. Una vez que se ha encontrado el registro de índice correcto, se tiene la distancia en bytes del registro de datos que se está buscando. Si se decide enlazar las llaves secundarias a sus registros asociados en el momento de la construcción del archivo, de modo que cuando se encuentre el registro DVORAK en el índice de compositores se pueda saber inmediatamente que comienza en el byte 353 del archivo de datos, la extracción por llave secundaria será más simple y rápida. La mejora en el desempeño es particularmente notable cuando los archivos de llaves primarias y secundarias se usan en el almacenamiento secundario y no en la memoria. Tomando en cuenta la disposición que se diseñó, tendría que efectuarse una búsqueda binaria del índice del compositor y después una búsqueda binaria del índice de la llave primaria antes de poder ir al registro de datos. Si se enlaza antes, en el momento de la construcción del archivo, se elimina por completo la necesidad de buscar mediante la llave primaria.

La desventaja del enlace directo en el archivo, el *enlace fuertemente acoplado*, es que las reorganizaciones del archivo de datos deben provocar modificaciones en todos los archivos de índices enlazados. Este costo de reorganización puede ser muy alto, en particular con archivos de índices simples, donde la modificación suele significar un movimiento de registros. Si se posterga el enlace hasta el tiempo de ejecución, cuando los registros se usan realmente, puede desarrollarse un sistema de llaves secundarias que implique una cantidad mínima de reorganización cuando se agregan o eliminan registros.

Otra ventaja importante de postergar el enlace hasta que realmente se extraiga el registro es que este enfoque es más seguro. Como se observa en el sistema que se desarrolla, asociar las llaves secundarias con los campos de referencia que consisten en llaves primarias permite que el índice de llaves primarias actúe como una especie de revisión final, que asegura que el registro esté en realidad en el archivo. Los índices secundarios pueden darse el lujo de estar equivocados. La situación es muy distinta cuando las llaves secundarias están fuertemente acopladas y contienen direcciones. Se estaría saltando entonces de la llave secundaria al archivo de datos; las direcciones requerirían ser correctas.

Esto implica un aspecto relacionado con la seguridad: siempre es deseable hacer los cambios importantes en un lugar, y no en muchos. Con un esquema de enlace al momento de la extracción de información, como el que se ha desarrollado, se necesita recordar hacer el cambio en un solo lugar, el índice de llaves primarias, en caso de mover un registro

de datos. Con un sistema de enlace fuertemente acoplado, hay muchos cambios que tienen que realizarse en forma exitosa para que el sistema permanezca consistente internamente, desafiando fallas de energía, interrupciones del usuario y otros problemas.

Cuando se diseña un nuevo sistema de archivos, es mejor considerar este aspecto del enlace *voluntario y temprano* en el proceso de diseño, en lugar de dejar que el enlace simplemente suceda. En general, el enlace fuerte en los datos es más atractivo cuando:

- el archivo de datos es estático o casi estático, de tal forma que requiere poca o nula adición, eliminación y actualización de registros, y
- el desempeño rápido durante la extracción tiene prioridad.

Por ejemplo, el enlace fuertemente acoplado es deseable para la organización de archivos en un disco óptico de lectura exclusiva producido en serie. Las direcciones nunca cambian, ya que no puede agregarse ningún registro; en consecuencia, no hay razón para no obtener la eficiencia adicional asociada con el enlace fuertemente acoplado.

Sin embargo, para aplicaciones en las que hay adición, eliminación y actualización de registros, el enlace al momento de la extracción de información es, por lo regular, la opción más deseable. En general, posponer el enlace tanto como sea posible facilita y asegura las operaciones. Si se diseñan con cuidado las estructuras de archivos y, en particular, si los índices tienen organizaciones más complejas, como árboles B, el desempeño de la extracción de información suele ser bastante aceptable, incluso considerando el trabajo adicional requerido por un sistema en que el enlace se realiza durante la extracción de información.

RESUMEN

Se inició este capítulo afirmando que la indización es una alternativa para la clasificación, ya que es una forma de estructurar un archivo que permite encontrar los registros *por llave*. A diferencia de la clasificación, la indización permite efectuar *búsquedas binarias* de llaves en archivos con registros de longitud variable. Si el índice puede almacenarse en la memoria, la adición, eliminación y extracción de registros pueden realizarse mucho más rápidamente con un archivo indizado de entradas secuenciales que con un archivo clasificado.

Los índices no sólo mejoran el tiempo de acceso: también proporcionan nuevas posibilidades que son inconcebibles con los métodos de acceso basados en registros de datos clasificados. La posibilidad más estimulante está en el uso de índices secundarios múltiples. Al igual que un catálogo de tarjetas de biblioteca permite contemplar una colección de libros en orden de autor, de título o por materia, los archivos de índice permiten visualizar de diversas formas los registros en el archivo de datos. Resulta que no sólo se pueden usar los índices secundarios para obtener diferentes formas de visualizar el archivo, sino que también se pueden combinar las listas asociadas de referencias de llaves primarias para así combinar ciertos enfoques particulares.

En este capítulo se aborda el aspecto de cómo liberar los índices secundarios de dos problemas:

- La necesidad de repetir llaves secundarias duplicadas, y
- La necesidad de reacomodar los índices secundarios cada vez que se agrega un registro al archivo de datos.

Una primera solución a estos problemas implica asociar un *vector* de tamaño fijo de los campos de referencia con cada llave secundaria. Esta solución provoca una cantidad excesiva de fragmentación interna, pero sirve para ilustrar las posibilidades que ofrece manejar campos de referencia asociados con una llave secundaria en particular como un grupo o *lista*.

La siguiente solución a los problemas de índices secundarios tiene más éxito y es mucho más interesante. Pueden tratarse las mismas referencias a llaves primarias como un archivo de entradas secuenciales, formando las listas necesarias por medio del uso de *campos de liga* asociados con cada entrada de registros primarios. Esto permite crear un archivo de índices secundarios que, en el caso del índice de compositores, necesita reacomodarse sólo cuando se agregan nuevos compositores al archivo de datos. El archivo secuencial de listas de referencias ligadas nunca requiere clasificarse. A esta clase de estructura de índice secundario se le llama *lista invertida*.

Por supuesto, también existen desventajas asociadas con la nueva solución. La desventaja más seria es que el archivo demuestra menos localidad: es menor la probabilidad de que las listas de registros asociados estén físicamente adyacentes. Un buen antídoto para este problema es almacenar el archivo de listas ligadas en la memoria. Esto resulta más plausible porque un único archivo de referencias primarias puede ligar las listas para varios índices secundarios.

Como demuestra la extensión y amplitud de este capítulo sobre indización secundaria, llaves múltiples y listas invertidas, estos temas son algunos de los aspectos más interesantes del acceso indizado a

archivos. Los conceptos de índices secundarios y listas invertidas adquieren aun mayor importancia posteriormente, conforme se desarrollan estructuras de índices más poderosas que los índices simples que se consideran aquí. Pero, aun así, se comprende que para archivos pequeños que contengan no más de algunos miles de registros, las listas invertidas que se basan en índices simples pueden dar al usuario gran capacidad y flexibilidad.

TERMINOS CLAVE

Archivo de entradas secuenciales. Archivo en el que los registros ocurren en el orden en que se introdujeron.

Campo de referencia. El campo de referencia es la porción de un registro de índice que contiene información acerca de dónde encontrar el registro de datos que contiene la información listada en el campo de llave asociado del índice.

Campo llave. El campo llave es la porción de un registro de índice que contiene la forma canónica de la llave que se busca.

Enlace. El enlace se realiza cuando una llave se asocia con un determinado registro físico del archivo de datos; en general, puede tener lugar ya sea durante la preparación del archivo de datos e índices o durante la ejecución del programa. En el primer caso, que es llamado *enlace fuertemente acoplado*, los índices contienen referencias explícitas a los registros físicos de datos asociados. En el último caso, la conexión entre una llave y un registro físico en particular se posterga hasta que realmente se extrae el registro, durante la ejecución del programa.

Índice. Un índice es una herramienta para encontrar registros en un archivo. Consiste en un *campo de llave* mediante el cual se busca el índice, y un *campo de referencia* que indica dónde encontrar el registro del archivo de datos asociado con una llave en particular.

Índice selectivo. Un índice selectivo contiene llaves sólo para una porción de los registros del archivo de datos. Dicho índice permite al usuario ver un subconjunto específico de los registros del archivo.

Índice simple. Todas las estructuras de índices analizadas en este capítulo son índices simples, porque están construidas con la idea de una secuencia lineal y ordenada de registros de índice. Todos estos índices simples tienen una debilidad en común:

agregar registros al índice es costoso. Como se estudiará posteriormente, los índices estructurados en árboles proporcionan una solución alterna más eficiente.

Lista invertida. El término *lista invertida* se refiere a los índices en donde una llave puede asociarse con una *lista* de campos de referencia que apuntan a objetos que contienen la llave. Los índices secundarios desarrollados al final de este capítulo son ejemplos de listas invertidas.

Localidad. La localidad existe en un archivo cuando los registros a los cuales se accederá en una secuencia temporal dada, se encuentran en proximidad física uno con otro en el disco. Incrementar la localidad por lo regular mejora la eficiencia, ya que los registros que están en la misma área física con frecuencia pueden llevarse a la memoria con una sola solicitud *read* al disco.

EJERCICIOS

1. Hasta ahora no era posible efectuar una búsqueda binaria en un archivo de registros de longitud variable. ¿Cómo es que la indización hace posible la búsqueda binaria? Con un archivo de registros de longitud fija es posible efectuar una búsqueda binaria. ¿Esto significa que no es necesario usar la indización en archivos de registros de longitud fija?
2. ¿Por qué no se usa el *título* como una llave primaria en el archivo de datos descrito en este capítulo? Si fuese usado como una llave secundaria, ¿qué problemas tendrían que considerarse al optar por una forma canónica para títulos?
3. ¿Cuál es el propósito de mantener una bandera de no actualizado en el registro de encabezamiento de un índice? En un ambiente de multiprogramación puede encontrarse esta bandera puesta por un programa porque otro está en el proceso de reorganización del índice. ¿Cómo debe responder a esta situación el primer programa?
4. Explique cómo es que el uso de un índice fija los registros en un archivo.
5. Cuando se actualiza un registro en un archivo de datos, los índices correspondientes de las llaves primarias y secundarias deben o no

alterarse, dependiendo de si el archivo tiene registros de longitud fija y del tipo de cambio hecho al registro de datos. Haga una lista de las distintas situaciones de actualización que puedan ocurrir, y explique cómo afecta cada una los índices.

6. Analice el problema que ocurre cuando se agrega el siguiente disco al archivo de discos, suponiendo que se emplea el índice de compositor que se muestra en la figura 7.9. ¿Cómo puede resolverse el problema sin cambiar sustancialmente la estructura de índices de llaves secundarias?

LON 1259 Fidelio Beethoven Maazel

7. ¿Qué es una lista invertida, y cuándo es de utilidad?

8. Diga cómo se modifican las estructuras de la figura 7.11 al agregar el siguiente disco

LON 1259 Fidelio Beethoven Maazel

9. Suponga que el archivo de datos descrito en este capítulo está muy expandido, con un índice de llave primaria y con índices de llave secundaria organizados por compositor, artista y título. Suponga también que se usa una estructura de lista invertida para organizar los índices de llave secundaria. Describa paso a paso cómo debe responder un programa a las siguientes consultas:

- Liste todos los discos de Bach o Beethoven, y
- Liste todos los discos con Perleman de piezas de Mozart o Joplin.

10. Un antídoto posible contra el problema de disminución de la localidad cuando se usan listas invertidas es emplear el mismo archivo de listas de ID de marca para almacenar las listas de varios de los archivos de índices secundarios. Esto incrementa la probabilidad de que los índices secundarios puedan almacenarse en la memoria principal. Dibuje un diagrama de un archivo de listas de ID de marca que pueda usarse para almacenar las referencias tanto de los índices secundarios de compositores como de títulos. ¿Cómo resolvería el lector las dificultades que esta disposición presenta con respecto al mantenimiento del archivo de listas de ID de marca?

11. Analice las siguientes estructuras como antídotos contra la posible pérdida de localidad en un índice de llaves secundarias.

- Dejar espacio para referencias múltiples para cada llave secundaria (Fig. 7.9).
- Asignar registros de longitud variable para cada valor de llave secundaria, donde cada registro contenga el valor de la llave secundaria, seguido por los ID de marca, y por el espacio libre para la adición posterior de ID de marca nuevos. La cantidad de espacio libre que se deja podría ser fija, o podría ser una función del tamaño de la lista original de ID de marca.

12. El método y tiempo del enlace afectan dos atributos importantes del sistema de archivos: la velocidad y la flexibilidad. Analice la relevancia de estos atributos y el efecto que tiene el tiempo del enlace en ellos, en un sistema de información de los pacientes de un hospital diseñado para proporcionar información acerca de los pacientes actuales mediante su nombre, el ID del paciente, medicamentos, doctor o doctores y enfermedades.

EJERCICIOS DE PROGRAMACION Y DISEÑO

13. Realice el procedimiento *recupera_registro()* planteado en la figura 7.4.

14. En la solución del problema anterior, el lector tiene que crear un mecanismo para decidir cuántos bytes debe leer de *Archdatos* para cada registro. Al menos se presentan cuatro opciones:

- a) Saltar a la *dist_bytes*, leer el campo de tamaño, y después usar esta información para leer el registro.
- b) Construir un archivo de índices que contenga un campo de tamaño del registro que refleje el tamaño *verdadero* del registro de datos, que incluya el campo de tamaño que se lleva en *Archdatos*. Usar el campo de tamaño que se lleva en el archivo de índices para decidir cuántos bytes se tienen que leer.
- c) Seguir la misma estrategia que en la opción (b), pero usando un *Archdatos* que no contenga campos internos de tamaño.
- d) Saltar a la *dist_bytes* y leer un número fijo y muy grande de bytes (p. ej., 512). Una vez que estos bytes se han trasladado a un buffer de memoria, usar el campo de tamaño al inicio del buffer para decidir cuántos bytes hay que tomar del buffer.

Evalúe cada una de estas opciones, listando sus ventajas y desventajas.

15. Realice los procedimientos para leer y para escribir de vuelta el arreglo INDICE[] al archivo de índices.

16. Cuando se buscan índices secundarios que contienen varios registros para alguna de las llaves, no quiere encontrarse sólo *cualquier* registro para una llave secundaria dada; se quiere encontrar el *primer* registro que contenga esa llave. Esto permite leer hacia delante, secuencialmente, extrayendo todos los registros para la llave considerada. Escriba una variación de una función de búsqueda binaria que devuelva el número relativo de registro del *primer* registro que contenga la llave considerada. La función debe devolver un valor negativo si no se encuentra la llave.

17. Si una lista de ID de marca, como la que se muestra en la figura 7.11, es demasiado grande para almacenarse por completo en la memoria, aún es posible mejorar el desempeño reteniendo varios bloques del archivo en memoria. A estos bloques se les llama *páginas*. Puesto que los registros del archivo de listas de ID de marca son de 16 bytes de largo cada uno, una página puede consistir en 32 registros (512 bytes). Escriba una función que pudiera almacenar en la memoria las ocho páginas usadas recientemente. Las llamadas a un registro específico del archivo de listas de ID de marca se procesarían por medio de esta función. Se revisaría si el registro existe en una de las páginas que de momento están en la memoria. Si es así, la función devolvería inmediatamente los valores de los campos del registro. Si no es así, la función leería la página que contiene el registro deseado, escribiendo o eliminando la página que se ha usado menos recientemente. Claramente, si una página se ha modificado, necesita escribirse en lugar de eliminarse. Cuando el programa termina, todas las páginas que están aún en la memoria deben revisarse para ver si deben escribirse.

18. Suponiendo el uso de un índice manejado por páginas como el descrito en el problema anterior, y tomando en cuenta que el archivo de listas de ID de marca es de entradas secuenciales, ¿existe algún orden particular de entrada de datos (carga inicial del archivo) que tienda a ser más eficiente que otros métodos? ¿Cómo modificaría el lector su respuesta acerca del desempeño, considerando el uso de un método de organización como el que se describe en el problema 10 y que combina las listas ligadas de varios índices secundarios en un solo archivo?

19. El archivo de listas de ID de marca es de entradas secuenciales. Desarrollar esquemas de paginación es más sencillo con archivos secuenciales que con archivos que se mantienen en algún orden de clasificación. Encuentre las dificultades adicionales implicadas en el

diseño de un sistema de paginación para un índice clasificado, como el índice de llaves primarias. Considerando la posibilidad de que existan varias páginas que sólo estén parcialmente completas, diseñe ese sistema de paginación. ¿Cómo realizará la inserción de una llave nueva cuando la página a la que pertenece esté completa?

LECTURAS ADICIONALES

Hay mucho más que decir acerca de la indización en los capítulos posteriores, donde se estudian los temas de índices estructurados en árboles y de organizaciones de archivo secuenciales indizadas. Los temas desarrollados en el presente capítulo, en particular los relacionados con índices secundarios y archivos invertidos, también se abordan en muchos otros textos sobre archivos y estructuras de datos. Los pocos textos que se listan aquí son interesantes debido a que desarrollan ciertos temas con mayor detalle o presentan el material desde un punto de vista distinto.

Wiederhold [1983] proporciona una investigación sobre muchas de las estructuras de índices que se analizan en el capítulo, junto con otras varias. El tratamiento que les da es más matemático que el que se proporciona aquí. Para los usuarios interesados en revisar archivos indizados en el contexto de PL/I y de máquinas grandes IBM, se recomienda Bradley [1981]. J.D. Ullman [1980] proporciona un estudio breve y comprensible de varias organizaciones de archivos diferentes.

Tremblay y Sorenson [1984] comparan las estructuras de listas invertidas con una organización alternativa llamada archivos *multilistas*. M.E.S. Loomis [1983] desarrolla un análisis similar, junto con algunos ejemplos orientados hacia los usuarios de COBOL. Salton y McGill [1983] analizan las listas invertidas en el contexto de su aplicación en sistemas de extracción de información.

OBJETIVOS

Describir el tipo de actividades de procesamiento de uso frecuente que se conoce como *procesamiento secuencial coordinado*

Proporcionar un modelo general para la implantación de todas las variedades de procesos.

Ilustrar el uso del modelo para resolver diversos tipos de problemas de procesamiento secuencial, además de los que incluyen intercalaciones simples y correspondencias.

Mostrar cómo la intercalación secuencial coordinada proporciona las bases para la clasificación de archivos grandes.

Examinar los costos de las intercalaciones de K-formas en disco y encontrar formas de reducir ese costo.

Introducir la noción de selección por *reemplazo*.

Examinar algunas cuestiones fundamentales asociadas con la clasificación de archivos grandes empleando cintas en lugar de discos.

8

PROCESAMIENTO SECUENCIAL COORDINADO Y CLASIFICACION DE ARCHIVOS GRANDES

PLAN GENERAL DEL CAPITULO

8.1 Un modelo para la realización de procesos secuenciales coordinados

8.1.1 Correspondencia de nombres en dos listas

8.1.2 Intercalación de dos listas

8.1.3 Resumen del modelo de procesamiento secuencial coordinado

8.2 Aplicación del modelo a un programa de libro mayor

8.2.1 El problema

8.2.2 Aplicación del modelo al programa de libro mayor

8.3 Extensión del modelo para incluir la intercalación múltiple

8.4 La intercalación como forma de clasificación de archivos grandes en disco

8.4.1 Patrones de intercalación de varios pasos

8.4.2 Incremento en las longitudes de las porciones mediante el uso de selección por reemplazo

8.4.3 Longitud promedio de las porciones para la selección por reemplazo

8.4.4 Costo del uso de la selección por reemplazo

8.4.5 Configuraciones de disco

8.4.6 Resumen de la clasificación en disco

8.5 Clasificación de archivos en cinta magnética

8.5.1 Intercalación balanceada

8.5.2 Intercalación en varias fases

8.6 Paquetes de clasificación e intercalación

Las operaciones secuenciales coordinadas implican *el procesamiento coordinado de dos o más listas secuenciales para producir una única lista de salida*. Algunas veces el procesamiento produce una *intercalación o unión* de las listas de entrada; algunas veces el objetivo es una correspondencia o una *intersección* de las listas, y otras veces la operación es una combinación de correspondencia e intercalación. Estos tipos de operaciones sobre listas secuenciales son la base de gran parte de procesamiento de archivos.

En la primera mitad del capítulo se desarrolla un modelo general para efectuar operaciones secuenciales coordinadas, se ilustra la manera de usarlo con operaciones simples de correspondencia e intercalación, y después se aplica al desarrollo de un programa de libro mayor más

complejo. Enseguida se aplica el modelo a la intercalación en varias formas, que es un componente esencial de las operaciones externas de clasificación e intercalación. Se concluye el capítulo con un estudio sobre los procedimientos, estrategias y compromisos de la clasificación e intercalación poniendo especial atención en las consideraciones de desempeño.

8.1

UN MODELO PARA LA REALIZACION DE PROCESOS SECUENCIALES COORDINADOS

Normalmente parece sencillo construir las operaciones secuenciales coordinadas. Dicha apariencia de simplicidad puede volverse cierta con ayuda de la información que se proporciona en este capítulo. Sin embargo, también es cierto que los métodos para el procesamiento secuencial coordinado con frecuencia son confusos, pobemente organizados e incorrectos. Estos ejemplos de una mala práctica de ninguna manera significan que tales métodos estén limitados a programas de estudiantes; estos problemas también aparecen en programas comerciales y en libros de texto. El problema con esos programas incorrectos es que, por lo común, no se organizan alrededor de un único modelo comprensible de procesamiento secuencial coordinado. Por el contrario, parece que abordan los problemas y condiciones de excepción de un proceso secuencial coordinado en una forma *ad hoc*, en lugar de sistemática.

Esta sección señala tal falta de organización. Se presenta un modelo simple que puede ser la base para la construcción de cualquier proceso secuencial coordinado. Cuando el lector comprenda y siga los principios de diseño inherentes al modelo podrá escribir procedimientos secuenciales coordinados sencillos, pequeños y robustos.

8.1.1 CORRESPONDENCIA DE NOMBRES EN DOS LISTAS

Supóngase que se necesita obtener los nombres comunes de las dos listas mostradas en la figura 8.1. A esta operación se le llama usualmente una operación de correspondencia o *intersección*. Se supone por el momento que no están permitidas llaves duplicadas dentro de una lista, y que las listas están clasificadas en orden ascendente.

Se comienza con la lectura del nombre inicial de cada lista, y se encuentra que corresponden. Se saca este primer nombre como un miembro del *conjunto de correspondencia* o de *intersección*. Despues se lee el siguiente nombre de cada lista. Esta vez el nombre de la lista 2 es

Lista 1	Lista 2
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREW
DAVIS	BECH
FOSTER	BURNS
GARWICK	CARTER
JAMES	DAVIS
JOHNSON	DEMPSEY
KARNS	GRAY
LAMBERT	JAMES
MILLER	JOHNSON
PETERS	KATZ
RESTON	PETERS
ROSEWALD	ROSEWALD
TURNER	SCHMIDT
	THAYER
	WALKER
	WILLIS

FIGURA 8.1 • Ejemplo de listas de entrada para operaciones secuenciales coordinadas.

menor que el nombre de la lista 1. Cuando se están procesando estas listas visualmente, como ahora, hay que recordar que se intenta hacer corresponder el nombre CARTER de la lista 1, y se busca en la lista 2 hasta encontrarlo, o pasarlo. En este caso se encuentra finalmente una correspondencia para CARTER, de tal forma que se le envía a la salida, se lee el siguiente nombre de cada lista y se continúa el proceso. A la larga, se llega al final de una de las listas. Como se buscan nombres comunes en ambas listas, se sabe que es posible detenerse en este punto.

Aunque el procedimiento de correspondencia parece bastante simple, hay varios aspectos que deben estudiarse para hacer que funcione razonablemente bien.

- *Asignación de valores iniciales.* Es necesario acomodar la información de modo que el procedimiento funcione en forma apropiada.
- *Sincronización.* Debe asegurarse que el nombre actual de una de las listas nunca esté tan adelante del nombre actual de la otra lista que se pierda una correspondencia posible. Algunas veces esto implica leer el siguiente nombre de la lista 1, otras veces de la lista 2, y otras, de ambas listas.
- *Manejo del estado de fin de archivo.* Cuando se llega al final del archivo 1 o del 2, es necesario detener el programa.

- Reconocimiento de errores.* Cuando hay un error en los datos (p. ej., nombres duplicados o nombres fuera de secuencia) se debe detectar y efectuar alguna acción.

Para finalizar, sería bueno que el algoritmo fuera razonablemente eficiente, simple y fácil de alterar, para que se ajustara a distintos tipos de datos. La clave para lograr dichos objetivos en el modelo que se estudia consiste en la manera de abordar el segundo aspecto de la lista: la sincronización.

En cada paso del procesamiento de las dos listas se puede suponer que se tienen dos nombres para comparar: un *nombre actual* de la lista 1 y un *nombre actual* de la lista 2. A estos nombres actuales se les llamará NOMBRE_1 y NOMBRE_2. Pueden compararse los dos nombres para determinar si NOMBRE_1 es menor, igual, o mayor que NOMBRE_2:

- Si NOMBRE_1 es *menor que* NOMBRE_2, se lee el siguiente nombre de la lista 1;
- Si NOMBRE_1 es *mayor que* NOMBRE_2, se lee el siguiente nombre de la lista 2;
- Si los nombres son iguales, se escribe el nombre y se leen los siguientes de las dos listas.

PROGRAMA: correspondencia

```

Llama al procedimiento inicio ( ) para:
- abrir los archivos de entrada LISTA_1 y LISTA_2
- crear el archivo de salida ARCH_SAL
- asignar verdadero a EXISTEN_MAS_NOMBRES
- iniciar las variables de revisión de secuencia

llama entrada ( ) para tomar NOMBRE_1 de LISTA_1
llama entrada ( ) para tomar NOMBRE_2 de LISTA_2

mientras (EXISTEN_MAS_NOMBRES)

    si (NOMBRE_1 < NOMBRE_2)
        llama entrada ( ) para tomar NOMBRE_1 de LISTA_1
    otro si (NOMBRE_1 > NOMBRE_2)
        llama entrada ( ) para tomar NOMBRE_2 de LISTA_2
    otro /* Correspondencia; los nombres son los mismos */
        escribir NOMBRE_1 a ARCH_SAL
        llama entrada ( ) para tomar NOMBRE_1 de LISTA_1
        llama entrada ( ) para tomar NOMBRE_2 de LISTA_2
    fin si
fin mientras
finish-up()

fin PROGRAMA

```

FIGURA 8.2 • Procedimiento de correspondencia *secuencial coordinada* basado en un ciclo.

Se puede hacer un manejo muy limpio, con un solo ciclo que contenga una proposición condicional con tres ramas, como se ilustra en el algoritmo de la figura 8.2. La característica clave de este algoritmo es que *el control vuelve siempre al inicio del ciclo principal después de cada paso de la operación*. Esto significa que no se requiere lógica adicional dentro del ciclo para manejar el caso cuando la lista 1 se adelanta a la lista 2, o la 2 se adelanta a la 1, o se ha llegado al fin de archivo en una de las listas antes que en la otra.

Puesto que cada recorrido del ciclo principal busca la siguiente pareja de nombres, el hecho de que una de las listas pueda ser más grande que la otra no requiere ninguna lógica especial, ni tampoco se requiere al llegar al fin del archivo; la proposición **mientras** revisa la bandera EXISTEN_MAS_NOMBRES en cada ciclo.

La lógica dentro del ciclo es igual de simple. Sólo pueden ocurrir tres situaciones posibles después de leer un nombre; la lógica del *si.. otro* las maneja todas. Como aquí se realiza un proceso de correspondencia, la salida ocurre sólo cuando los nombres son iguales.

```
PROCEDIMIENTO: entrada ( ) /* Rutina de entrada para el procedimiento de
correspondencia */
```

Argumentos de entrada:

```
ARCH_ENT : descripción de archivo para el archivo de entrada a usar
(puede ser LISTA_1 o LISTA_2)
NOMBRE_ANT: último nombre leído de esta lista
```

Argumentos usados para devolver valores

```
NOMBRE : nombre que va a ser devuelto
EXISTEN_MAS_NOMBRES :bandera usada por el ciclo principal para
detener el procesamiento
```

Leer el siguiente NOMBRE de ARCH_ENT

```
/* Detectar el estado de fin de archivo, nombres duplicados y nombres
```

fuera de orden */

si (EOF)

```
    EXISTEN_MAS_NOMBRES : = FALSO /* Colocar la bandera para
terminar el procesamiento */
```

otro si (NOMBRE <= NOMBRE_ANT)

Emitir error de secuencia

Abortar el procesamiento

fin si

```
    NOMBRE_ANT : = NOMBRE
```

fin PROCEDIMIENTO

FIGURA 8.3 • Rutina de entrada para el procedimiento de correspondencia.

```

PROCEDIMIENTO : inicializar ( )
    Argumentos usados para devolver valores:
        PREV_1 PREV_2 : variables de nombre previo para las dos listas
        LISTA_1 LISTA_2 : descriptores de archivo para los archivos de entrada
        que se usarán
        EXISTEN_MAS_NOMBRES : bandera utilizada por el ciclo principal para
        detener el procesamiento

            /* Asignar a las variables de nombre previo (una para cada lista)
            un valor que se asegure es menor que cualquier valor de entrada
            */
            PREV_1 = VALOR_PEQUEÑO
            PREV_2 = VALOR_PEQUEÑO

            abrir el archivo para Lista 1 como LISTA_1
            abrir el archivo para Lista 2 como LISTA_2

            if (ambas instrucciones para abrir archivos se realizan exitosamente)
                EXISTEN_MAS_NOMBRES : = VERDADERO

fin PROCEDIMIENTO

```

FIGURA 8.4 • Procedimiento de inicialización para procesamiento secuencial coordinado.

Obsérvese que el programa principal no considera aspectos tales como revisión de secuencia y detección del fin del archivo. Puesto que su presencia en el ciclo principal sólo podría oscurecer la lógica principal de sincronización, se han relegado a subprocedimientos.

Como la situación de fin de archivo se detecta durante la lectura, la activación de la bandera EXISTEN_MAS_NOMBRES se realiza en el procedimiento *lectura()*. El procedimiento *lectura()* también puede usarse para cerciorarse de que las listas están en estricto orden ascendente (sin registros duplicados dentro de la lista).

El algoritmo de la figura 8.3 ilustra un método para el manejo de estas tareas. Este "llenado" del procedimiento *lectura()* también señala los argumentos que usaría el procedimiento.

Todo lo que se necesita ahora para terminar la lógica es una descripción del procedimiento *inicial()* que da principio al procedimiento principal de correspondencia secuencial coordinado. El procedimiento *inicial()*, que se muestra en la figura 8.4, efectúa tres tareas:

1. Abre los archivos de entrada y salida.
2. Asigna VERDADERO a la bandera EXISTEN_MAS_NOMBRES.
3. Asigna a las variables *nombre_anterior* (una por cada lista) un valor que se garantice sea menor que cualquier valor de entrada. La

razón de asignar a PREV_1 y PREV_2 VALOR_BAJO es que el procedimiento *lectura()* no necesita realizar la lectura de los dos primeros registros en forma especial.

Tomando en cuenta estos fragmentos de programa, el lector debe poder trabajar con las dos listas proporcionadas en la figura 8.1, siguiendo el pseudocódigo, y demostrarse a sí mismo que estos procedimientos sencillos puedan solucionar los distintos problemas de resincronización que presentan estas listas de ejemplo.

8.1.2 INTERCALACION DE DOS LISTAS

El modelo de prueba de tres formas y ciclo sencillo para el procesamiento secuencial coordinado puede modificarse con facilidad para manejar la *intercalación* de las listas, así como la correspondencia, como se ilustra en la figura 8.5. Nótese que ahora se produce una salida para todos los

```

PROGRAMA : intercala

    llama al procesamiento inicia ( ) para:
        - abrir los archivos de entrada LISTA_1 y LISTA_2
        - crear el archivo de salida ARCH_SAL
        - asignar verdadero a EXISTEN_MAS_NOMBRES
        - iniciar las variables de revisión de secuencia

    llama entrada ( ) para tomar NOMBRE_1 de LISTA_1
    llama entrada ( ) para tomar NOMBRE_2 de LISTA_2

    mientras (EXISTEN_MAS_NOMBRES)

        si (NOMBRE_1 < NOMBRE_2)
            escribir NOMBRE_1 a ARCH_SAL
            llama entrada ( ) para tomar NOMBRE_1 de LISTA_1,
        otro si (NOMBRE_1 > NOMBRE_2)
            escribir NOMBRE_1 a ARCH_SAL
            llama entrada ( ) para tomar NOMBRE_2 de LISTA_2

        otro /* Correspondencia; los nombres son los mismos */
            escribir NOMBRE_1 a ARCH_SAL
            llama entrada ( ) para tomar NOMBRE_1 de LISTA_1
            llama entrada ( ) para tomar NOMBRE_2 de LISTA_2
        fin si
    fin mientras
    finish_up()

fin PROGRAMA

```

FIGURA 8.5 • Procedimiento de intercalación secuencial coordinada basado en un ciclo.

casos de la construcción *si... otro*, puesto que una intercalación es una *unión* del contenido de las listas.

Una diferencia importante entre la correspondencia y la intercalación es que con la intercalación debe leerse *totalmente* cada una de las listas. Esto precisa un cambio en el procedimiento *lectura()*, ya que la versión empleada para la correspondencia asigna FALSO a la bandera EXISTEN_MAS_NOMBRES tan pronto como detecta el final del archivo de una de las listas. Es necesario mantener esta bandera en VERDADERO en tanto existan registros en *cualquiera* de las listas. Al mismo tiempo, se debe reconocer que una de las listas se ha leído por completo, y evitar leerla de nuevo. Ambos objetivos pueden lograrse simplemente asignando a la variable NOMBRE, para la lista que se agotó, algún valor que:

```
PROCEDIMIENTO: entrada ( ) /* Rutina de entrada para el procedimiento de
INTERCALACION */
```

Argumentos de entrada

ARCH_ENT	: descriptor de archivo para el archivo de entrada por usarse (puede ser LISTA_1 o LISTA_2)
NOMBRE_ANT	: último nombre leído de esta lista
NOMBRE_OTRA_LISTA	: el nombre más reciente leído de la otra lista

Argumentos usados para devolver valores

NOMBRE	: nombre que va a ser devuelto a partir del procedimiento de entrada
EXISTEN_MAS_NOMBRES	bandera usada por el ciclo principal para detener el procesamiento

Ler el siguiente NOMBRE de ARCH_ENT

```
/* Detectar el estado de fin de archivo, nombres duplicados y nombres
fuera de orden
si (EOF) y (NOMBRE_OTRA_LISTA = = VALOR_ALTO)
    EXISTEN_MAS_MENOS : = FALSO /* Fin de ambas listas */
otro si (EOF)
    NOMBRE : = VALOR_ALTO /* Sólo terminó esta lista */

otro si (NOMBRE < = NOMBRE_ANT) /* Revisión de secuencia
    emitir error de secuencia
    abortar el procedimiento
fin si

NOMBRE_ANT = NOMBRE

fin PROCEDIMIENTO
```

FIGURA 8.6 • Rutina de entrada para el procedimiento de intercalación.

- no tenga posibilidad de ocurrir como un valor de entrada legal, y
- tenga un valor de secuencia *más alto* que cualquier valor de entrada legal. En otras palabras, este valor especial vendría después de todos los valores de entrada legales en la secuencia ordenada del archivo.

A este valor se le denomina VALOR_ALTO. El pseudocódigo de la figura 8.6 muestra cómo puede usarse para asegurar que ambos archivos se lean por completo. Nótese que tiene que agregarse el argumento NOMBRE_OTRA_LISTA a la lista de argumentos para que la función sepa si la otra lista de entrada ha llegado a su fin.

De nuevo, debe aplicarse esta lógica, paso a paso, en las listas de la figura 8.1, para entender cómo se maneja la resincronización y cómo el uso de VALOR_ALTO fuerza al procedimiento a terminar ambas listas antes de concluir. Obsérvese que la versión de *lectura()* que incorpora la lógica de VALOR_ALTO también puede usarse para los procedimientos de correspondencia, produciendo resultados correctos. La única desventaja de hacerlo así es que el procedimiento de correspondencia no terminaría en cuanto una de las listas se procese totalmente, sino que se haría el trabajo adicional de leer todas las entradas sin correspondencia al final de la otra lista.

Con estos dos ejemplos se han cubierto todas las partes del modelo. En seguida se resume el modelo antes de adaptarlo a un problema más complejo.

8.1.3 RESUMEN DEL MODELO DE PROCESAMIENTO SECUENCIAL COORDINADO

Hablando en general, el modelo puede aplicarse a problemas que impliquen el uso de operaciones de conjuntos (unión, intersección y procesos más complejos) en dos o más archivos de entrada clasificados para producir uno o más archivos de salida. En este resumen del modelo de procesamiento secuencial coordinado se supone que sólo existen dos archivos de entrada y uno de salida. Es importante comprender que el modelo hace ciertas suposiciones generales acerca de la naturaleza de los datos y del tipo de problema que va a resolverse. Aquí se presenta una lista de suposiciones, junto con comentarios aclaratorios.

Suposiciones

Dos o más archivos de entrada se procesan en forma paralela para producir uno o más archivos de salida.

Comentarios

En algunos casos, un archivo de salida puede ser el mismo que uno de los de entrada.

Suposiciones	Comentarios
Cada archivo se clasifica por uno o más campos llave, y todos los archivos están ordenados en la misma forma sobre los mismos campos.	No es necesario que todos los archivos tengan la misma estructura de registro.
En algunos casos debe existir un valor de llave alto que sea más grande que cualquier llave legítima de registro, y un valor bajo que sea menor que cualquier llave legítima de registro.	El uso de un valor de llave alto y de uno bajo no es absolutamente necesario, pero puede ayudar a evitar la necesidad de tratar como casos especiales los estados de inicio y final de archivo y, por lo tanto, disminuir la complejidad.
Los registros se procesan en el orden lógico de clasificación.	El orden <i>físico</i> de los registros es irrelevante para el modelo, pero en la práctica puede ser muy importante para la manera de implantarlo. El orden físico puede influir en la eficiencia del procesamiento.
Para cada archivo existe sólo un registro actual. Este es el registro cuya llave es accesible dentro del ciclo de sincronización	El modelo no prohíbe buscar registros hacia delante o hacia atrás, pero tales operaciones deben restringirse a subprocedimientos, y no debe permitirse que se afecte la estructura del ciclo principal de sincronización.
Los registros pueden manipularse sólo en memoria interna.	Un programa no puede alterar un registro en el almacenamiento secundario.

Considerando estas suposiciones, se presentan a continuación los componentes esenciales del modelo.

1. **Asignación de valores iniciales.** Los registros actuales para todos los archivos se leen de los primeros registros lógicos en los archivos respectivos. Los valores de *Llave_anterior* para todos los archivos se asignan con el valor bajo.
2. Se emplea un ciclo principal de sincronización, que continúa mientras haya registros relevantes.

3. Dentro del cuerpo del ciclo principal de sincronización hay una selección basada en la comparación de las llaves de registro de los registros respectivos de los archivos de entrada. Si hay dos archivos de entrada, la selección se ve así:

```
si (llave_actual_archivos > llave_actual_archivo2) entonces
    .
    .
otro si (llave_actual_archivo1<llave_actual_archivo2) entonces
    .
    .
otro /* Las llaves actuales son iguales */
    .
    .
fin si
```

4. Se revisa que los archivos de entrada y salida estén en secuencia, comparando el valor *llave_anterior*, con el valor *llave_actual*, cuando se lee un registro. Después de revisar con éxito la secuencia, se asigna *llave_anterior* a *llave_actual*, para preparar la siguiente operación de entrada en el archivo correspondiente.
5. Los valores altos se sustituyen en los valores de llave actual cuando aparece un final de archivo. El ciclo principal de procesamiento termina cuando han ocurrido los valores altos para todos los archivos de entrada relevantes. El uso de valores altos evita la necesidad de agregar un código especial para el estado de final de archivo. (Este paso no es necesario en un procedimiento de correspondencia pura, porque el procedimiento de correspondencia se detiene cuando se encuentra el primer estado de final de archivo.)
6. Todas las actividades posibles de E/S y detección de errores se relegan a subprocessos, de tal forma que los detalles de estas actividades no obscurezcan la lógica del procesamiento principal.

Este modelo de prueba de tres formas y ciclo sencillo para la creación de procesos secuenciales coordinados es simple y sólido. Se encontrarán muy pocas aplicaciones que requieran el procesamiento secuencial coordinado de dos archivos que no puedan manejarse limpia y eficientemente con este modelo. Ahora se estudiará un problema mucho más completo que el de una simple correspondencia o intercalación, pero que, no obstante, se presta muy bien para ser resuelto por medio del modelo.

8.2

APLICACION DEL MODELO A UN PROGRAMA DE LIBRO MAYOR

8.2.1 EL PROBLEMA

Supóngase que se presenta el problema de diseñar un programa de libro mayor como parte de un sistema de contabilidad. El sistema incluye un archivo de diario y un archivo de libro mayor. El libro mayor contiene el resumen mensual de los valores asociados con cada una de las cuentas de la contabilidad. Un ejemplo de una parte del libro mayor, que contiene sólo las cuentas de cheques y gastos, se ilustra en la figura 8.7.

El archivo de diario contiene las transacciones mensuales que finalmente se asientan en el archivo de libro mayor. La figura 8.8 muestra cómo se ven estas transacciones. Nótese que los renglones en el archivo diario están en parejas. Ello se debe a que todo cheque implica tanto restar una cantidad de balance de la cuenta de cheques como agregar al menos una cantidad a una cuenta de gastos. El paquete de contabilidad requiere procedimientos para la creación interactiva de

Cuenta núm.	Título de la cuenta	Enero	Febrero	Marzo	Abril
101	Cuenta de cheques núm. 1	1032.57	2114.56	5219.23	
102	Cuenta de cheques núm. 2	543.78	3094.17	1321.20	
505	Gastos de publicidad	25.00	25.00	25.00	
510	Gastos de autos	195.40	307.92	501.12	
515	Cargos de bancos	0.00	5.00	5.00	
520	Libros y publicaciones	27.95	27.95	87.40	
525	Gastos de interés	103.50	255.20	380.27	
530	Gastos legales				
535	Gastos varios	12.45	17.87	23.87	
540	Gastos de oficina	57.50	105.25	138.37	
545	Correo y envío	21.00	27.63	57.45	
550	Renta	500.00	1000.00	1500.00	
550	Provisiones	112.00	167.50	241.80	
560	Viáticos	62.76	198.12	307.74	
565	Utilidades	84.89	190.60	278.48	

FIGURA 8.7 • Ejemplo de un fragmento de libro mayor que contiene cuentas de cheques y gastos.

Cuenta Cheque núm.	núm.	Fecha	Descripción	Cargo o Abono
101	1271	04/02/86	Gastos de auto	- 78.70
510	1271	04/02/86	Afinación y reparación menor	78.70
101	1272	04/02/86	Renta	- 500.00
550	1272	04/02/86	Renta de abril	500.00
101	1273	04/04/86	Publicidad	- 87.50
505	1273	04/04/86	Anuncio en un diario: producto nuevo	87.50
102	670	04/07/86	Gastos de oficina	- 32.78
540	670	04/07/86	Cintas de impresora (6)	32.78
101	1274	04/09/86	Gastos de auto	- 12.50
510	1274	04/09/86	Cambio de aceite	12.50

FIGURA 8.8 • Muestra de entradas del diario.

este archivo de diario, probablemente transcribiendo los registros al archivo mientras se capturan y se imprimen los cheques.

Una vez que el archivo de diario se ha completado para un determinado mes, lo cual significa que contiene todas las transacciones de ese mes, debe asentarse en el libro mayor. *Asentar* implica asociar cada transacción con su cuenta en el libro mayor. Por ejemplo, la salida impresa para las cuentas 101, 102, 505 y 510, durante la operación de asentamiento, para los registros de la figura 8.8, se ve como en la figura 8.9.

¿Cómo se realiza el proceso de asentamiento? Claramente usa el número de cuenta como una *llave* para relacionar las transacciones del diario con los registros del libro mayor. Una solución posible implica la construcción de un índice para el libro mayor, de tal forma que puedan revisarse las transacciones diarias usando el número de cuenta en cada entrada para buscar el registro correcto del libro mayor. Pero esta solución implica desplazamientos en el archivo del libro mayor mientras se trabaja con el diario. Además, esta solución realmente no se encarga de la creación de la lista de salida, en la que se agrupan todos los registros diarios relacionados con una cuenta. Antes de poder imprimir los balances del libro mayor y agrupar los registros diarios de por lo menos la primera cuenta, la 101, se tendría que proceder a lo largo de toda la lista del diario. ¿Dónde se guardarían las transacciones para la cuenta 101 conforme se leen, durante este recorrido completo por el diario?

Una solución mucho mejor es comenzar a agrupar todas las transacciones que estén relacionadas con una cuenta determinada. Esto implica la clasificación de las transacciones diarias por número de cuenta, produciendo una lista ordenada, como en la figura 8.10

101 Cuenta de cheques núm. 1

1271	04/02/86	Gastos de auto	- 78.70
1272	04/02/86	Renta	- 500.00
1273	04/04/86	Publicidad	- 87.50
1274	04/09/86	Gastos de auto	- 12.50
		Bal. anterior: 5219.23	Bal. nuevo: 4540.53

102 Cuenta de cheques núm. 2

670	04/07/86	Gastos de oficina	-32.78
		Bal. anterior: 1321.20	Bal. nuevo: 1288.42

505 Gastos de publicidad

1273	04/04/86	Anuncio en un diario: producto nuevo	87.50
		Bal. anterior: 25.00	Bal. nuevo: 112.50

510 Gastos de auto

1271	04/02/86	Afinación y reparación menor	78.70
1274	04/09/86	Cambio de aceite	12.50
		Bal. anterior: 501.12	Bal. nuevo: 592.32

FIGURA 8.9 • Ejemplo de un listado de libro mayor que muestra el efecto de la aplicación del diario.

Ahora se puede crear la lista de salida trabajando en la forma secuencial coordinada a través del libro mayor y del diario clasificado, lo que significa que se procesan las dos listas secuencialmente y en paralelo. Este concepto se ilustra en la figura 8.11. En cuanto se empieza a trabajar con las dos listas, se observa que se tiene una correspondencia inicial en el número de cuenta. Se sabe que se per-

Cuenta núm.	Cheque núm.	Fecha	Descripción	Cargo o Abono
101	1271	04/02/86	Gastos de auto	- 78.70
101	1272	04/02/86	Renta	-500.00
101	1273	04/04/86	Publicidad	- 87.50
101	1274	04/09/86	Gastos de auto	- 12.50
102	670	04/07/86	Gastos de oficina	- 32.78
505	1273	04/04/86	Anuncio en un diario: producto nuevo	87.50
510	1271	04/02/86	Afinación y reparación menor	78.70
510	1274	04/09/86	Cambio de aceite	12.50
540	670	04/07/86	Cintas de impresora (6)	32.78
550	1272	04/02/86	Renta de abril	500.00

FIGURA 8.10 • Lista de las transacciones diarias clasificadas por número de cuenta.

Lista de libro mayor				Lista de diario			
101	Cuenta de cheques núm. 1	101	1271	Gastos de auto			
		101	1272	Renta			
		101	1273	Publicidad			
		101	1274	<u>Gastos de auto</u>			
102	Cuenta de cheques núm. 2	102	670	Gastos de oficina			
505	Gastos de publicidad	505	1273	Anuncio en un diario: producto nuevo			
510	Gastos de auto	510	1271	Afinación y reparación menor			
		510	1274	Cambio de aceite			

FIGURA 8.11• Vista conceptual de correspondencia secuencial coordinada de los archivos de libro mayor y de diario.

miten entradas múltiples en el archivo de diario, pero no en el libro mayor, de tal forma que hay que moverse al siguiente registro del diario. El número de cuenta aún corresponde. Se continúa así hasta que el número de cuenta ya no corresponde. Entonces se resincroniza la acción secuencial coordinada moviéndose hacia delante en la lista de libro mayor.

El proceso de correspondencia parece sencillo, y de hecho lo es, mientras las cuentas de un archivo aparezcan en el otro. Pero existen cuentas en el libro mayor para las que no existe un registro de diario, y pueden existir errores tipográficos que crean números de cuenta de diario que no existan en realidad en el libro mayor. Tales situaciones pueden hacer más compleja la resincronización y ocasionar salidas erróneas o ciclos infinitos cuando la programación se hace en forma *ad hoc*. Empleando el modelo de procesamiento secuencial coordinado es posible protegerse contra estos problemas. Ahora se aplicará el modelo al problema del libro mayor.

8.2.2 APLICACION DEL MODELO AL PROGRAMA DE LIBRO MAYOR

El programa de libro debe efectuar dos tareas:

- Necesita actuar el archivo de libro mayor con el balance correcto para cada cuenta del mes en curso.
- Debe producir una versión impresa del libro mayor que no sólo muestre el balance inicial y actual de cada cuenta, sino que también liste todas las transacciones diarias del mes.

101 Cuenta de cheques núm. 1

1271	04/02/86	Gastos de auto	- 78.70
1272	04/02/86	Renta	-500.00
1273	04/04/86	Publicidad	- 87.50
1274	04/09/86	Gastos de auto	- 12.50
Bal. anterior: 5219.23 Bal. nuevo: 4540.53			

102 Cuenta de cheques núm. 2

670	04/07/86	Gastos de oficina	-32.78
Bal. anterior: 1321.20 Bal. nuevo: 1288.42			

505 Gastos de publicidad

1273	04/04/86	Anuncio en un diario: producto nuevo	87.50
Bal. anterior: 25.00 Bal. nuevo: 112.50			

510 Gastos de auto

1271	04/02/86	Afinación y reparación menor	78.70
1274	04/09/86	Cambio de aceite	12.50
Bal. anterior: 501.132 Bal. nuevo: 592.32			

515 Cargos de bancos

Bal. anterior. 5.00 Bal. nuevo: 5.00			
--------------------------------------	--	--	--

520 Libros y publicaciones

Bal. anterior: 87.40 Bal. nuevo: 87.40			
--	--	--	--

FIGURA 8.12• Ejemplo de un listado de libro mayor para las primeras seis cuentas.

Se centrará la atención en la segunda tarea, ya que es la más difícil. Véase de nuevo la forma de salida impresa, esta vez ampliéndola para que incluya algunas cuentas, como se muestra en la figura 8.12. Como puede observarse, la salida impresa del programa de libro mayor muestra los balances de todas las cuentas del libro mayor, existan o no transacciones para la cuenta. Desde el punto de vista de las cuentas del libro mayor, el proceso es una *intercalación*, puesto que aparecen en la salida incluso las cuentas de libro mayor que no tuvieron correspondencia.

¿Qué sucede con las cuentas de diario que no tuvieron correspondencia? Las cuentas de libro mayor y del diario no son iguales en autoridad. El archivo de libro mayor *define* el conjunto legal de cuentas; el archivo de diario contiene registros que se *asentarán* en las cuentas listadas en el libro mayor. La existencia de una cuenta de diario que no coincida con una cuenta del libro mayor indica un error. Desde el punto de vista de las cuentas de diario, el proceso de asentamiento es estrictamente un proceso de correspondencia: El procedimiento necesita

realizar una especie de algoritmo combinado de intercalación y correspondencia, mientras maneja simultáneamente las tareas de impresión de las líneas de título de las cuentas, de transacciones individuales y de balances acumulados.

Otra diferencia importante entre la operación de asentamiento en el libro mayor y los algoritmos directos de correspondencia e intercalación radica en que el procedimiento de libro mayor debe aceptar registros duplicados para los números de cuenta del diario, mientras que sigue considerando un registro duplicado en el libro mayor como un error. Recuérdese que las rutinas anteriores de correspondencia intercalación acepta llaves sólo en un orden ascendente estricto, rechazando todos los duplicados.

PROCEDIMIENTO: entrada_mayor ()

Argumentos de entrada:

ARCH_M : descriptor de archivo para el archivo de libro mayor
CUENTA_D : valor actual del número de cuenta diario

Argumentos usados para devolver valores:

CUENTA_M : número de cuenta del nuevo registro de libro mayor
BAL_M : balance para este registro de libro mayor
EXISTEN_MAS_NOMBRES: bandera usada por el ciclo principal para detener el procesamiento

Variable local estática que retiene su valor entre llamadas

CUENTA_M_ANT: último número de cuenta leído del archivo de libro mayor

Ler el siguiente registro de ARCH_M, asignando valores a CUENTA_M y BAL_M

```
si (EOF) y (CUENTA_D == VALOR_ALTO)
    EXISTEN_MAS_NOMBRES : = FALSO /* Fin de ambos archivos */
```

otro si (EOF)

```
    CUENTA_M : = VALOR_ALTO /* Sólo terminó el libro mayor */
```

```
otro si (CUENTA_M < = CUENTA_M_ANT) /* Revisión de secuencia */
    emitir error de secuencia      /* (No se permiten duplicados) */
    abortar el procesamiento
```

fin si

```
CUENTA_M_ANT : = CUENTA_M
```

fin PROCEDIMIENTO

FIGURA 8.13- Rutina de entrada para el archivo de libro mayor.

La simplicidad inherente del modelo de la prueba de tres formas y ciclo sencillo trabaja en nuestro favor conforme se hacen estas modificaciones. Primero, véanse las funciones de entrada que se usan para los archivos de libro mayor y de diario para identificar las variables requeridas para el ciclo principal. La figura 8.13 presenta el pseudocódigo del procedimiento que lee la entrada del libro mayor. Se han tratado las variables individuales dentro del registro de libro mayor como valores de retorno para atraer la atención a estas variables; en la práctica, es probable que el procedimiento devuelva el registro de libro

PROCEDIMIENTO: entrada_diario ()

Argumentos de entrada:

ARCH_ : descriptor de archivo para el archivo de diario
CUENTA_M : valor actual de número de cuenta de libro mayor

Argumentos usados para devolver valores:

CUENTA_D : número de cuenta del nuevo registro de diario
CANT_TRANS : cantidad de esta transacción del diario
EXISTEN_MAS_NOMBRES : bandera usada por el ciclo principal para detener el procesamiento

Variable local estática que retiene su valor entre llamadas
CUENTA_D_ANT : último número de cuenta leído del archivo de diario

Leer el siguiente registro de ARCH_D, asignando valores a CUENTA_D y CANT_TRANS

si (EOF) y (CUENTA_N == VALOR_ALTO)
EXISTEN_MAS_NOMBRES : = falso /* Fin de ambos archivos */

otro si (EOF)
CUENTA_D : = VALOR_ALTO /* Sólo terminó el libro mayor */

otro si (CUENTA_D <= CUENTA_D_ANT) /* Revisión de secuencia */
emitir error de secuencia /* (permite duplicados) */
abortar el procesamiento

fin si

CUENTA_D_ANT : = CUENTA_D

fin PROCEDIMIENTO

FIGURA 8.14- Rutina de entrada para el archivo de diario.

mayor completo a la rutina que lo llamó, de modo que otros procedimientos tuvieran acceso a datos tales como el título de la cuenta a medida que se imprime el libro mayor. Tales aspectos se pasan por alto aquí; más bien se enfocan las variables que están involucradas en la lógica secuencial coordinada. Nótese que, puesto que esta función es para usarse estrictamente con registros de libro mayor, puede mantenerse la información del número de localidad previa de la cuenta de

PROGRAMA: mayor

```

llama al procedimiento inicia ( ) para:
  - abrir los archivos de entrada ARCH_M y ARCH_D
  - asignar VERDADERO a EXISTEN_MAS_REGISTROS

llamada entrada_mayor ( )
BAL_M_ANT : = MAL_M      /* Asignar el balance inicial de libro
                           mayor para esta primera cuenta de mayor */
llama entrada_diario ( )

mientras (EXISTEN_MAS_REGISTROS)
  si (CUENTA_M < CUENTA_D)      /* Se han leído todos los datos
                                     del diario para esta cuenta      */
    escribir BAL_M_ANT y BAL_N
    llama entrada_mayor ( )
    si (CUENTA_M < VALOR_ALTO)
      escribir el número y título de la cuenta
      de libro mayor
      BAL_M_MAYOR : = BAL_M
    fin si

  otro si (CUENTA_M > CUENTA_D)      /* Número de cuenta de
                                         diario incorrecto      */
    imprimir mensaje de error
    llame entrada_diario ( )
  otro      /* Si corresponde: agregar la cantidad de la
             transacción del diario al balance de libro mayor
             para esta cuenta      */
    BAL_M : = BAL_M + CANT_TRANS
    escribir la transacción al libro mayor impreso
    llamada entrada_diario ( )
  fin si
fin mientras

fin PROGRAMA

```

FIGURA 8.15. Procedimiento secuencial coordinado para procesar los archivos de libro mayor y de diario para producir el listado del libro mayor.

libro mayor dentro del procedimiento, en lugar de pasar este valor como argumento.

La figura 8.14 describe la lógica del procedimiento usado para leer la entrada del archivo de diario. Se asemeja al procedimiento *lectura_mayor()* en la mayoría de los aspectos, entre ellos el hecho de que devuelve valores de variables individuales, aun cuando una versión funcional probablemente devolvería el registro de diario completo.

Sin embargo, nótese que la lógica de revisión de secuencia es diferente en *lectura_diario()*. En este procedimiento es necesario aceptar los registros que tengan el mismo número de cuenta que los anteriores. Considerando estos procedimientos de entrada, el procesamiento secuencial coordinado y la salida pueden manejarse como se muestra en la figura 8.15.

El razonamiento que sustenta la prueba de tres formas es el siguiente:

1. Si la cuenta de libro mayor es menor que la cuenta de diario ya no hay más transacciones que agregar a esta cuenta de libro mayor (quizás no hubo ninguna), así se imprimen los balances de la cuenta de libro mayor y se lee la siguiente cuenta. Si la cuenta existe (valor < VALOR_ALTO), se imprime la línea de título para la nueva cuenta y se actualiza la variable *BAL_PREV*.
2. Si la cuenta de diario es menor que la cuenta de libro mayor, entonces se trata de una cuenta que no tuvo correspondencia, quizás debido a un error en la captura. Se imprime un mensaje de error y se continúa.
3. Si los números de cuenta corresponden, entonces se tiene que asentar una transacción del diario en la cuenta actual de libro mayor. Se agrega la cantidad de la transacción al balance de la cuenta, se imprime la descripción de la transacción, y después se lee el siguiente registro del diario. Nótese que, a diferencia del caso de la correspondencia en los algoritmos de correspondencia o intercalación, no se lee un nuevo registro de ambas cuentas. Esto revela la aceptación de más de un registro de diario para una sola cuenta de libro mayor.

El desarrollo de este procesamiento de asentamiento al libro mayor, a partir del modelo básico de procesamiento secuencial coordinado, ilustra cómo la simplicidad del modelo contribuye a su adaptabilidad. También puede generalizarse el modelo en una dirección completamente distinta, ampliéndolo para habilitar el procesamiento secuencial coordinado de más de dos archivos de entrada al mismo tiempo. Para ilustrar esto ahora se amplía el modelo de modo que incluya intercalación de varias formas.

8.3**EXTENSION DEL MODELO PARA INCLUIR LA INTERCALACION MULTIPLE**

La aplicación más común de procesos secuenciales coordinados que requieren más de dos archivos de entrada es una *intercalación de K formas*, en la que se pretende intercalar *K* listas de entrada para crear una sola lista de salida ordenada secuencialmente.

Se recordará el ciclo de sincronización usado para manejar una intercalación de dos formas de dos listas de nombres (Fig. 8.5). Esta operación de intercalación puede verse como un proceso de decisión con respecto a cuál de los dos nombres tiene el valor *mínimo*, para llevar a la salida ese nombre y después moverse hacia delante en la lista de donde se tomó. En el caso de registros duplicados, el movimiento es hacia delante en cada lista.

Dada una función *min()* que devuelva el nombre con el valor de secuencia más bajo, no hay razón para restringir el número de nombres de entrada a dos. El procedimiento podría ampliarse, para que maneje tres (o más) listas de entrada, como se muestra en la figura 8.16.

Está claro que la parte costosa de este procedimiento es la serie de pruebas que se hacen para saber en qué listas ocurre el nombre, y cuáles

mientras (EXISTEN_MAS_NOMBRES)

NOMBRE_SAL = min (NOMBRE_1, NOMBRE_2, NOMBRE_3, ... NOMBRE_K)
escribe NOMBRE_SAL a ARCH_SAL

si (NOMBRE_1 == NOMBRE_SAL)
 llama entrada () para tomar NOMBRE_1 de LISTA_1

si (NOMBRE_2 == NOMBRE_SAL)
 llama entrada () para tomar NOMBRE_2 de LISTA_2

si (NOMBRE_3 == NOMBRE_SAL)
 llama entrada () para tomar NOMBRE_3 de LISTA_3
:

si (NOMBRE_K == NOMBRE_SAL)
 llama entrada () para tomar NOMBRE_K de LISTA_K

fin mientras

FIGURA 8.16• Ciclo de intercalación de *K* formas, que considera nombres duplicados.

archivos, por tanto, se necesita leer. Obsérvese que, puesto que el nombre puede ocurrir en varias listas, cada una de estas pruebas sí debe ejecutarse en cada iteración del ciclo. Sin embargo, con frecuencia es posible garantizar que un solo nombre, o llave, ocurra sólo en una lista. En este caso, el procedimiento se vuelve mucho más simple y eficiente. Supóngase que se hace referencia a las listas por medio de un vector de nombres de listas:

lista[1], lista[2], lista[3], ...lista[K]

y supóngase que, por medio de otro vector, se hace referencia a los nombres (o llaves) que se usan a partir de estas listas en cualquier punto del proceso secuencial coordinado:

nombre[1], nombre[2], nombre[3], ...nombre(K)

Entonces puede usarse el procedimiento que se muestra en la figura 8.17, suponiendo, una vez más, que el procedimiento *lectura()* atiende a la bandera EXISTEN_MAS_NOMBRES.

```

/* Inicia el proceso leyendo un nombre para cada lista */  

para i : = 1 hasta K  

    llama entrada ( ) para tomar nombre [i] de lista [i]  

siguiente i  
  

/* Ahora inicia la intercalación de K formas */  

mientras (EXISTEN_MAS_NOMBRES)  
  

    /* Encontrar el subíndice del nombre como el menor valor de  

       secuencia entre los nombres disponibles en las K letras */  

    MENOR : = 1  

    para i : = 2 hasta K  

        si (nombre[i] < nombre [MENOR])  

            MENOR : = 1  

    siguiente i  
  

    escribir el nombre [MENOR] a ARCH_SAL  
  

    /* Se reemplaza el nombre que se escribió */  

    llama entrada ( ) para tomar [MENOR] de lista [MENOR]
fin mientras

```

FIGURA 8.17 Ciclo de intercalación K, suponiendo que no hay nombres duplicados.

Este procedimiento difiere claramente en muchos aspectos del procedimiento inicial de prueba de tres formas y ciclo sencillo que intercala dos listas. Pero, incluso así, el parecido con el ciclo sencillo es aún evidente: no hay iteraciones dentro de una lista. Se determina cuál tiene la llave con el valor más bajo, se envía esa llave a la salida se avanza una llave en esa lista y se itera de nuevo. Este procedimiento es tan simple como poderoso.

La intercalación de K formas descrita en la figura 8.17 funciona muy bien cuando K no es mayor de 8, aproximadamente. Cuando se comienza a intercalar un mayor número de listas, el conjunto de comparaciones secuenciales para encontrar la llave con el valor mínimo comienza a ser notablemente costoso. Se verá más adelante que, por razones prácticas, es raro querer intercalar más de ocho archivos a la vez, de modo que el uso de comparaciones secuenciales por lo regular es una buena estrategia. Si hubiera necesidad de intercalar mucho más de ocho listas, se reemplazaría el ciclo de comparaciones por un *árbol de selección*.

El uso de un árbol de selección es un ejemplo del clásico compromiso de tiempo contra espacio que con tanta frecuencia se encuentra en las ciencias de la computación. Se reduce el tiempo requerido para encontrar la llave con el valor más bajo empleando una estructura de datos para guardar la información acerca de los valores de llave relativos a lo largo de las iteraciones del ciclo principal del procedimiento. El concepto en el que se basa un árbol de selección se puede exponer fácilmente por medio de un diagrama como el de la figura 8.18. Se han usado listas donde las llaves son números en lugar de nombres.

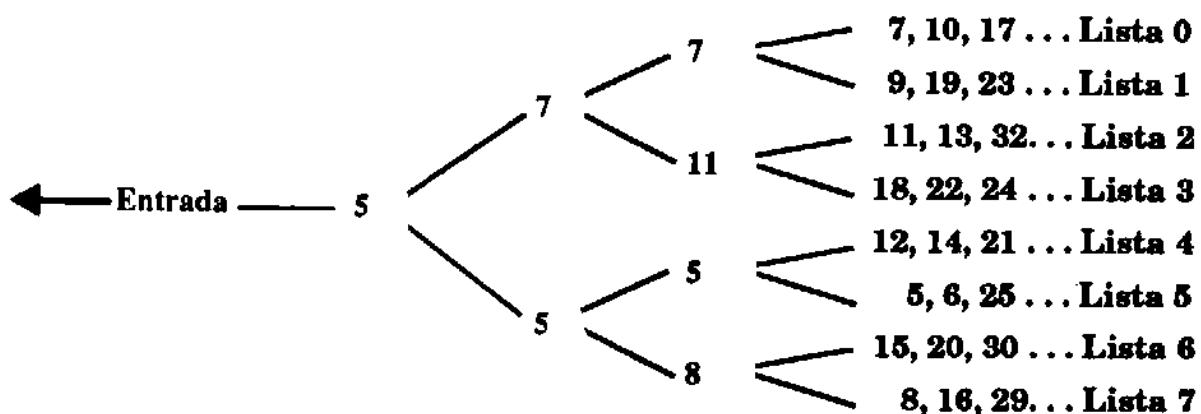


FIGURA 8.18• Uso de un árbol de selección para auxiliarse en la selección de una llave con valor mínimo en una intercalación de K formas.

El árbol de selección es una especie de *árbol de torneo*, donde cada nodo de nivel superior representa al "ganador" (en este caso el valor de llave *mínimo*) de la comparación entre las dos llaves descendientes. El valor mínimo siempre está en el nodo raíz del árbol. Si cada llave tiene una referencia asociada con la lista de donde proviene, es sencillo tomar la llave de la raíz y leer el siguiente elemento de la lista asociada para después efectuar el torneo de nuevo. Como el árbol de torneo es un árbol binario, su profundidad es de

$$\lceil \log_2 K \rceil$$

para una intercalación de K listas. Por supuesto, el número de comparaciones requeridas para establecer un nuevo ganador del torneo se relaciona con esta profundidad, en vez de ser una función lineal de K .

8.4

LA INTERCALACION COMO FORMA DE CLASIFICACION DE ARCHIVOS GRANDES EN DISCO

En el capítulo 6 se presentaron problemas cuando se requería clasificar archivos que eran demasiado grandes para caber por completo en la memoria electrónica de acceso aleatorio (RAM). El capítulo ofreció una solución parcial, pero al final de cuentas insatisfactoria, en forma de una *clasificación por llave*, donde se necesitaba almacenar sólo las llaves en memoria RAM, junto con los apuntadores a cada registro correspondiente de la llave. La clasificación por llave tenía dos defectos:

- Cuando las llaves se habían clasificado, hubo que tolerar el importante costo de los desplazamientos para cada registro clasificado, leyendo cada uno y luego transcribiéndolo en el nuevo archivo clasificado.
- Con la clasificación por llave, el tamaño del archivo que puede clasificarse está limitado por el número de parejas de llave y apuntador que pueden caber en la memoria RAM. En consecuencia, todavía no se pueden clasificar archivos realmente grandes.

A manera de ejemplo de un tipo de archivo que no puede ordenarse mediante una clasificación en memoria RAM ni mediante una clasificación por llave, supóngase que se tiene un archivo con 400 000 registros, cada uno de 100 bytes de longitud y con un campo de llave de 10 bytes de largo. La longitud total de este archivo es de 40 megabytes. Supóngase además que se tiene un megabyte de memoria RAM dis-

ponible, como área de trabajo, sin contar la memoria utilizada para almacenar el programa, el sistema operativo y otros datos. Claramente, no puede clasificarse el archivo completo en memoria RAM. No pueden, siquiera, clasificarse todas las llaves en la memoria.

El algoritmo de intercalación múltiple de la sección 8.3 proporciona el principio de una solución atractiva al problema de clasificación de archivos grandes como éste. Puesto que los algoritmos de clasificación en memoria RAM, como (quicksort), pueden trabajar en su propio espacio, empleando sólo una cantidad pequeña adicional para el mantenimiento de una pila y de algunas variables temporales, se puede crear un subconjunto clasificado del archivo completo, transfiriendo los registros a memoria RAM hasta que el área de trabajo esté casi llena; clasificando los registros en esta área de trabajo, y luego transcribiendo los registros clasificados de regreso al disco como un subarchivo clasificado. A tales subarchivos clasificados se les llama porciones. Considerando las restricciones de memoria y el tamaño del registro del ejemplo, una porción tendría aproximadamente

$$\frac{1\ 000\ 000 \text{ bytes de memoria RAM}}{100 \text{ bytes por registro}} = 10\ 000 \text{ registros.}$$

Cuando se crea la primera porción se lee un nuevo conjunto de registros, de nuevo llenando la memoria RAM, y creando otra porción de 10 000 registros. En el ejemplo se repite este proceso hasta que se han creado 40 porciones, donde cada una contiene 10 000 registros clasificados.

Una vez que se tienen las 40 porciones en 40 archivos separados en disco, puede efectuarse una intercalación de 40 formas, empleando la lógica de intercalación múltiple planteada en la sección 8.3, para crear un archivo completamente clasificado que contenga todos los registros originales. Un diagrama esquemático de esta creación de porciones y del proceso de intercalación aparece en la figura 8.19.

Esta solución al problema de clasificación tiene las siguientes características:

- De hecho, puede clasificar archivos grandes, y puede ampliarse a archivos de cualquier tamaño.
- La lectura del archivo de entrada durante el paso de creación de la porción es completamente secuencial, y por lo tanto es mucho más rápida que en la entrada que requiere desplazamientos (como en una clasificación por llave).
- La lectura de cada porción durante la intercalación y la escritura de los registros clasificados también es secuencial. Se requieren desplazamientos sólo cuando se cambia de una porción a otra durante la operación de intercalación.

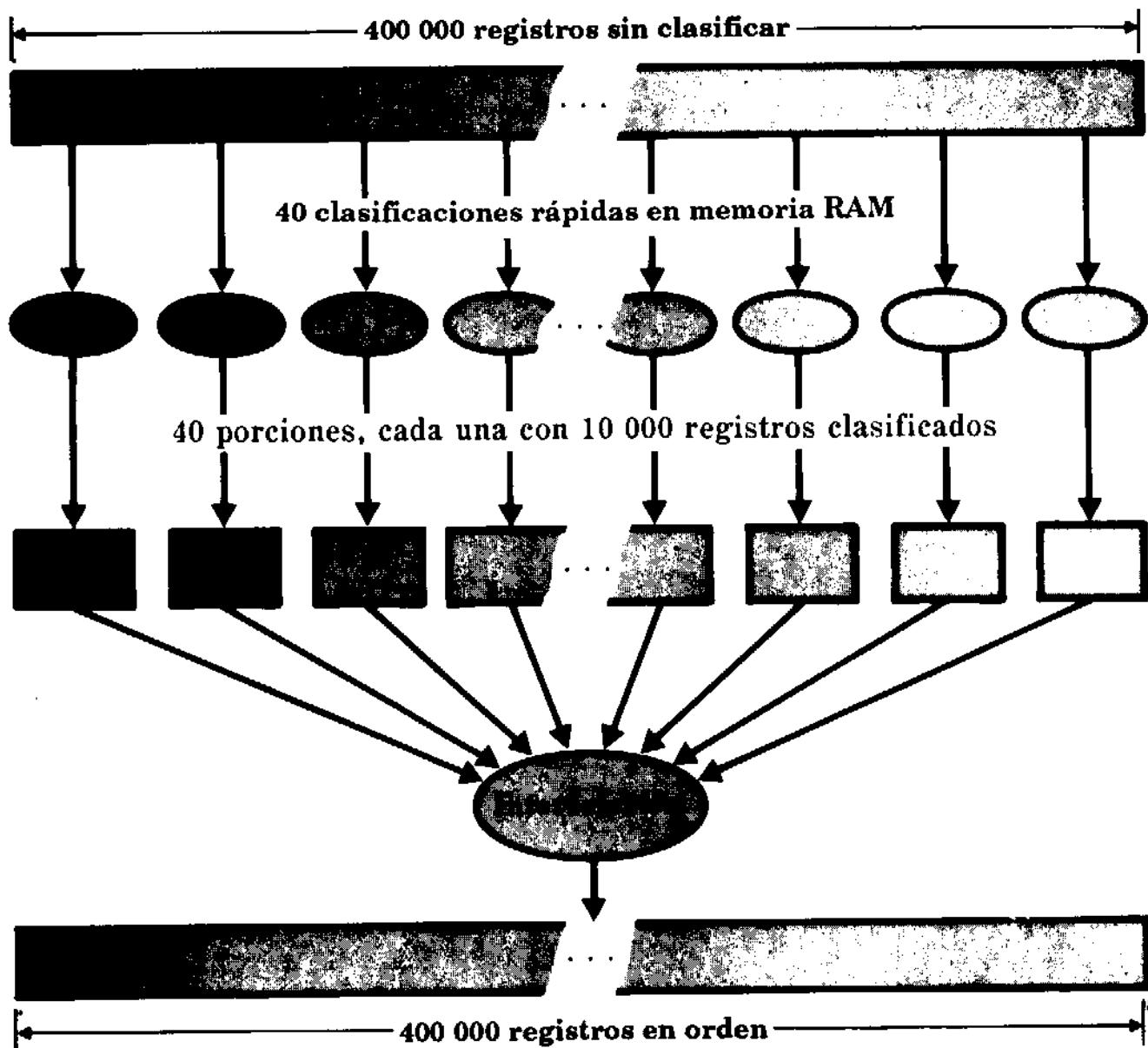


FIGURA 8.19 Clasificación por medio de la creación de porciones (subarchivos clasificados) y su subsecuente intercalación.

- Puesto que la E/S es, en gran medida, secuencial, pueden usarse cintas si es necesario, tanto para las operaciones de entrada como para las de salida.
- Si la entrada y la salida es a una cinta, entonces puede clasificarse un archivo tan grande como el espacio disponible del disco. (El disco se usa para almacenar sólo las porciones.) Para muchos sistemas instalados esto significa que pueden clasificarse archivos de tamaño superior a los cientos de megabytes, y quizás aun hasta de varios gigabytes.

Este enfoque general al problema de la clasificación de archivos grandes parece prometedor. Por desgracia, posee una característica notablemente indeseable: el proceso se retarda debido a los desplazamientos que tienen lugar en el disco durante la fase de intercalación. Para ilustrar esto, considérense los desplazamientos asociados sólo con la *entrada* al paso de intercalación. (Supóngase que la salida de la intercalación va a la cinta magnética, de tal forma que no se requieren desplazamientos para la salida.)

Para los datos del ejemplo que se considera, el desplazamiento se realiza entre las 40 porciones que se están intercalando. Por supuesto, la entrada de cada una se almacena en un buffer, de modo que no es necesario efectuar un desplazamiento por cada dato. ¿Cuán grande puede ser cada buffer? Puesto que cada una de las 40 porciones es del tamaño del área de trabajo disponible en memoria RAM, se sigue que cuando se divide esa área de trabajo en buffers para las porciones, cada buffer sólo almacenará 1/40 de una porción. En consecuencia, es preciso desplazarse 40 veces por cada una para leerla completa. Como hay 40, para completar la operación de intercalación (Fig. 8.20) se tienen que hacer

$$40 \text{ porciones} \times 40 \text{ desplazamientos/porción} = 1600 \text{ desplazamientos}^{\dagger}$$

Ya se sabe que los desplazamientos son costosos, así que cualquier operación que requiera 1600 desplazamientos para terminarse es una operación que conviene revisar cuidadosamente para saber si hay alguna forma de reducir la cantidad que se requiere.

[†]En general, para la intercalación de K formas de K porciones, donde cada porción es del tamaño del área de trabajo disponible en memoria RAM, el tamaño del buffer para cada una de las porciones es

$$(1/K) \times \text{tamaño del espacio de memoria RAM} = (1/K) \times \text{tamaño de cada porción.}$$

de tal forma que se requieren K desplazamientos para leer todos los registros en cada porción individual. Puesto que hay un total de K porción, la operación de intercalación requiere K^2 desplazamientos. Por lo tanto, la clasificación e intercalación medida en términos de desplazamiento es una operación $O(K^2)$. Como K es directamente proporcional a N (si se duplica el número de registros de 400 000 a 800 000, K se duplica de 40 a 80), se concluye que la clasificación e intercalación es una operación $O(N^2)$, medida en términos de desplazamiento. En otras palabras, el desempeño puede deteriorarse rápidamente conforme N se incrementa.

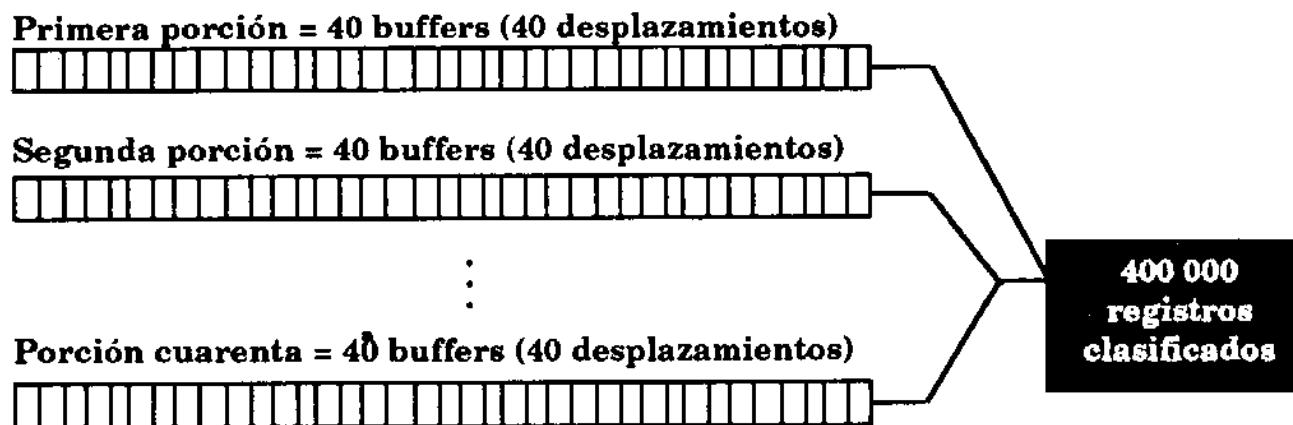


FIGURA 8.20 • Efecto del manejo de buffers en el número de desplazamientos requerido, donde cada porción es tan grande como el área de trabajo disponible en memoria RAM.

Hay dos formas de reducir el número de desplazamientos requeridos para el paso de intercalación de la clasificación:

- Efectuar la intercalación en más de un paso, reduciendo el orden de cada intercalación e incrementando el tamaño del buffer, e
- Incrementar el tamaño de las porciones iniciales clasificadas.

Se estudiará por separado cada una de estas dos mejoras. Sin embargo, no son mutuamente exclusivas: un buen programa de clasificación puede hacer uso de ambas.

8.4.1 PATRONES DE INTERCALACION DE VARIOS PASOS

Uno de los rasgos distintivos de la solución a un problema de estructuras de archivos, en contraste con la solución de un problema meramente de estructuras de datos, es la atención puesta en la enorme diferencia de costos entre el desplazamiento en disco y el acceso a la información en memoria RAM. Si el problema de intercalación implicara sólo operaciones en memoria RAM, la medida importante de trabajo, o de costo, sería el número de *comparaciones* requeridas para terminar la intercalación. *El patrón de intercalación* que minimiza el número de comparaciones en el problema del ejemplo, en el que se pretende intercalar 40 porciones, sería la intercalación de 40 formas estudiada en la sección anterior. Si se considera desde un punto de vista que ignore el costo de los desplazamientos, esta intercalación de K formas tiene las siguientes características deseables:

- Cada registro se lee sólo una vez.
- Si se usa un árbol de selección para las comparaciones efectuadas en la operación de intercalación, como se describe en la sección 8.3, entonces el número de comparaciones requeridas para una intercalación de K formas de N registros (totales) es función de

$$N \times \log K$$

- Puesto que K es directamente proporcional a N , ésta es una operación $O(N \times \log N)$ (medida en número de comparaciones), la cual es razonablemente eficiente aún cuando N crezca mucho.

Todo esto sería bueno si se trabajara exclusivamente en memoria RAM, pero el verdadero propósito de este procedimiento de *clasificación por intercalación* es poder clasificar archivos que son demasiado grandes para caber en la memoria RAM. Para el caso por resolver, los costos asociados con los desplazamientos en el disco son órdenes de magnitud mayores que los costos de las operaciones en memoria RAM. En consecuencia, si pueden sacrificarse las ventajas de una intercalación de 40 formas y cambiarlas por ahorro de tiempo de desplazamientos, puede obtenerse una ganancia neta en el desempeño.

En vez de intercalar las 40 porciones a la vez, se podrían intercalar como cinco conjuntos de ocho porciones cada uno, seguidos de una intercalación de cinco formas de las porciones *intermedias*. Usando la misma notación de Knuth [1973b], esto se designa como una intercalación 8:8:8:8:8 (cinco conjuntos de intercalaciones de ocho formas). Este esquema se ilustra en la figura 8.21.

Cuando se compara con la intercalación original de 40 formas, este método tiene la desventaja de requerir que se lea dos veces cada registro: una para formar las porciones intermedias, y otra, para

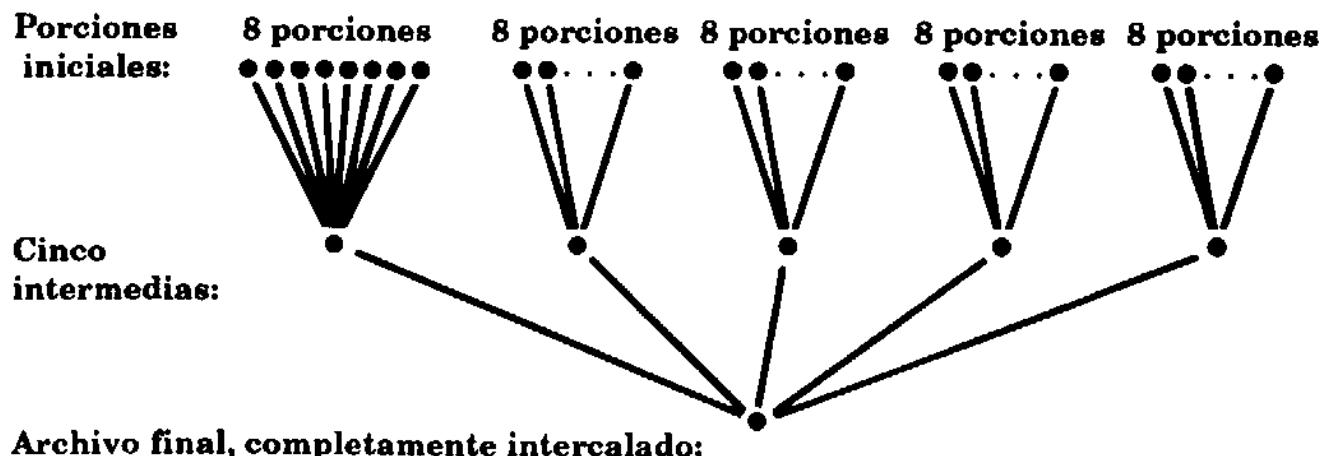


FIGURA 8.21 • Intercalación 8:8:8:8:8 (dos pasos) de 40 porciones.

formar el archivo clasificado final. Pero, puesto que en cada paso de la intercalación se lee, a lo sumo, de ocho archivos de entrada, es posible emplear buffers grandes y evitar una gran cantidad de desplazamientos en disco. Cuando se analizó el desplazamiento requerido para la intercalación de 40 formas, sin considerar el desplazamiento para el archivo de salida, se calculó que la intercalación de 40 formas implicaba 1600 desplazamientos entre los archivos de entrada. Se efectuará un cálculo parecido para la intercalación 8:8:8:8:8.

1. *Para cada una de las intercalaciones de 8 formas de las porciones iniciales:*
 - Cada uno de los ocho buffers de entrada puede almacenar 1/8 de porción;
 - Por lo tanto: 8 desplazamientos/porción × 8 porciones = 64 desplazamiento por cada una de estas intercalaciones;
 - Total de las cinco intercalaciones de ocho formas: $5 \times 64 = 320$ desplazamientos.
2. *Intercalación de cinco formas de las porciones intermedias:*
 - Cada una de las porciones intermedias es ocho veces más grande que una inicial;
 - Cada uno de los cinco buffers de entrada puede almacenar 1/40 de una porción intermedia;
 - Por lo tanto: 40 desplazamientos/porción × 5 porciones = 200 desplazamientos.
3. *Total de desplazamientos para la intercalación completa 8:8:8:8:8:*
 - $320 + 200 = 520$ desplazamientos.

Así, al aceptar el costo que significa procesar cada registro dos veces, se reduce el número de desplazamientos en dos tercios. Es interesante ver lo que sucede si se duplica el número de registros por clasificar, de 400 000 a 800 000, y por lo tanto el número de porciones iniciales de 40 a 80. Ignorando una vez más el manejo de buffers para la salida, una intercalación de 80 formas requiere *6400 desplazamientos!* Reemplazando la intercalación de 80 formas por una intercalación 10:10:10:10:10:10:10 de dos pasos (ocho intercalaciones separadas de 10 formas, seguidas de una intercalación final de ocho formas), puede reducirse el número de desplazamientos a 1440, lo cual es ciertamente un número mucho más manejable que 6400.

Un análisis cuidadoso de los compromisos entre el tiempo de desplazamiento y el tiempo de transmisión (que está relacionado con el número de veces que se lee cada registro), tomando en cuenta los buffers de salida y los de entrada, es más complejo de lo que estos cálculos indican. De hecho, es necesario tomar en cuenta las características específicas del hardware involucrado. Por ejemplo, conside-

rando ciertas configuraciones de hardware, puede tener sentido hacer una intercalación de tres pasos en vez de dos.

Por ejemplo, ¿cuál sería el efecto de intercalar las 40 porciones originales, que contienen 400 000 registros, efectuando dos intercalaciones de dos pasos 5:5:5:5:5 de 200 000 registros cada una, y después un tercer paso donde las dos porciones intermedias de 200 000 registros se combinan por medio de una intercalación de dos formas? ¿Cómo cambia esto si la intercalación puede hacerse en cuatro paquetes de disco con brazos de desplazamiento separados? Las referencias citadas al final de este capítulo, en especial Knuth [1973b], contienen análisis detallados de este tipo de aspectos.

Se debe tener cuidado, a la luz de estas dificultades y opciones, de no perder de vista el punto principal: al clasificar archivos grandes en disco mediante la intercalación de un gran número de porciones, casi siempre resulta ventajoso pagar el precio de leer los registros más de una vez para reducir el número necesario de desplazamiento en disco.

8.4.2 INCREMENTO EN LAS LONGITUDES DE LAS PORCIONES MEDIANTE EL USO DE SELECCIÓN POR REEMPLAZO

¿Qué sucedería si se pudiese incrementar de algún modo el tamaño de las porciones iniciales? Por ejemplo, considérese de nuevo la clasificación de 400 000 registros, donde cada registro era de 100 bytes. Las porciones iniciales se limitaron a 10 000 registros aproximadamente, porque el área de trabajo en memoria RAM estaba limitado a un megabyte. Supóngase que de algún modo se pueden crear porciones del doble de longitud, con 20 000 registros cada una. Entonces, en lugar de requerir una intercalación de 40 formas, se necesita sólo una de 20. La memoria RAM disponible está dividida en 20 buffers, cada uno de los cuales puede almacenar 1/40 de una porción (¿Por qué?) Por lo tanto, el número de desplazamientos requeridos por porción es 40, y el número total de desplazamientos es

$$40 \text{ desplazamientos/porción} \times 20 \text{ porciones} = 800 \text{ desplazamientos}$$

la mitad del número requerido para la intercalación de 40 formas con porciones de 10 000 bytes.

En general, si puede incrementarse de algún modo el tamaño de las porciones iniciales, decrece la cantidad de trabajo requerida durante el paso de intercalación en el proceso de clasificación. Una porción inicial más grande significa menor número de porciones totales, esto es, una intercalación de menor orden lo que implica buffers más grandes y, por

tanto, menor número de desplazamientos. Pero ¿cómo pueden crearse porciones iniciales que sean dos veces más grandes que el número de registros que pueden almacenarse en memoria RAM sin comprar dos veces más memoria para el computador? La respuesta, una vez más, implica prescindir de cierta eficiencia dentro de las operaciones en memoria RAM a cambio de la disminución del trabajo que se hará en el disco. En particular, la respuesta implica el uso de un algoritmo conocido como *selección por reemplazo*.

La selección por reemplazo se basa en la sencilla idea de seleccionar siempre de la memoria la llave con valor más bajo, enviarla a la salida y después *reemplazarla* con una nueva llave de la lista de entrada. La selección de la llave puede realizarse por medio de un árbol de selección del tipo descrito con anterioridad. Algunas veces la nueva llave que se introduce como reemplazo tiene un valor superior al de la que se ha enviado a la salida. Cuando esto sucede, es posible incluir la nueva llave en la porción junto con las demás llaves que se seleccionan para la salida. Esto permite formar porciones que en realidad son más grandes que el número de llaves que pueden almacenarse en memoria a la vez.

Para entender cómo funciona esto se comenzará con un ejemplo sencillo, empleando una lista de entrada de sólo seis llaves, y una memoria de área de trabajo que puede almacenar sólo tres llaves. Como se ilustra en la figura 8.22, se empieza transfiriendo a memoria RAM las tres llaves que caben. Se selecciona la llave en la memoria RAM con el valor mínimo, que es 5 en este ejemplo, y se saca esa llave. Ahora se tiene espacio en memoria para otra llave, así que se lee de la lista de entrada. La nueva llave, que tiene un valor de 12, es ahora un miembro del conjunto de llaves que se clasificarán dentro de la porción de salida. De hecho, puesto que es más pequeña que las otras llaves en memoria

Entrada:

21, 67, 12, 5, 47, 16

↑ Frente de la cadena de entrada

Resto de la entrada	Memoria ($P = 3$)	Porción de salida
21, 67, 12	5 47 16	-
21, 67,	12 47 16	5
21,	67 47 16	12, 5
-	67 47 21	16, 12, 5
-	67 47 -	21, 16, 12, 5
-	67 - -	47, 21, 16, 12, 5
-	- - -	67, 47, 21, 16, 12, 5

FIGURA 8.22 - Ejemplo del principio que sustenta la selección por reemplazo.

RAM, 12 es la siguiente llave que se manda a la salida. Se lee una llave nueva en su lugar, y el proceso continúa. Cuando termina, se produce una lista clasificada de seis llaves, siendo que sólo se usaron tres localidades de memoria.

Al analizar este ejemplo surgen naturalmente preguntas tales como "¿Qué sucede si la cuarta llave de la lista de entrada es 2 en vez de 12?". Es obvio que esta llave llegaría a la memoria demasiado tarde para ser enviada a la salida en su posición adecuada en relación con las otras: el 5 ya se ha escrito en la lista de salida. La solución es simplemente almacenar el 2 hasta que se termine la escritura de la primera porción, incorporándola en la siguiente. En general, cuando una llave leída tiene un valor menor que el de la llave que se ha enviado a la salida más reciente, se señala como miembro de la siguiente porción.

La figura 8.23 ilustra cómo funciona este método. Durante la primera porción, cuando las llaves que son demasiado pequeñas para incluirse allí se llevan a la memoria, se las señala con un paréntesis, indicando que deben guardarse para la segunda porción.

Entrada:

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16

↑
**Frente de la cadena de
entrada**

Resto de la entrada	Memoria ($P = 3$)	Porción de salida
33, 18, 24, 58, 14, 17, 7, 21, 67, 12,	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67,	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21,	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7,	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17,	67 47 (7)	21, 16, 12, 5
33, 18, 24, 58, 14,	67 (17) (7)	47, 21, 16, 12, 5
33, 18, 24, 58,	(14) (17) (7)	67, 47, 21, 16, 12, 5

Primera porción completa; se inicia la construcción de la segunda

33, 18, 24, 58,	14 17 7	-
33, 18, 24,	14 17 58	7
33, 18,	24 17 58	14, 7
33,	24 18 58	17, 14, 7
-	24 33 58	18, 17, 14, 7
-	33 58	24, 18, 17, 14, 7
-	58	33 24, 18, 17, 14, 7
-	58, 33	24, 18, 17, 14, 7

FIGURA 8.23 • Operación paso a paso de la selección por reemplazo que funciona para formar dos porciones clasificadas.

Es interesante usar este ejemplo para comprobar la acción de la selección por reemplazo con el procedimiento que se ha usado hasta este punto, de transferir las llaves a memoria RAM, clasificarlas y enviarlas a una porción que es del tamaño del espacio de memoria RAM. En este ejemplo, la lista de entrada contiene 13 llaves. Una serie de clasificaciones sucesivas en memoria RAM, considerando sólo tres localidades de memoria, dan como resultado cinco porciones. El procedimiento de selección por reemplazo da como resultado sólo dos porciones. Puesto que los desplazamientos en disco durante una intercalación múltiple pueden significar un fuerte gasto, la capacidad de la selección por reemplazo de crear menos y más grandes porciones puede ser una ventaja importante.

En este punto surgen dos cuestiones:

1. Dadas P localidades en memoria, ¿qué longitud media puede esperarse que tenga una porción producida por la selección por reemplazo?
2. Hasta el momento se ha visto que pocas cosas son gratis cuando se trabaja con estructuras de archivos. ¿Cuáles son los costos de la selección por reemplazo?

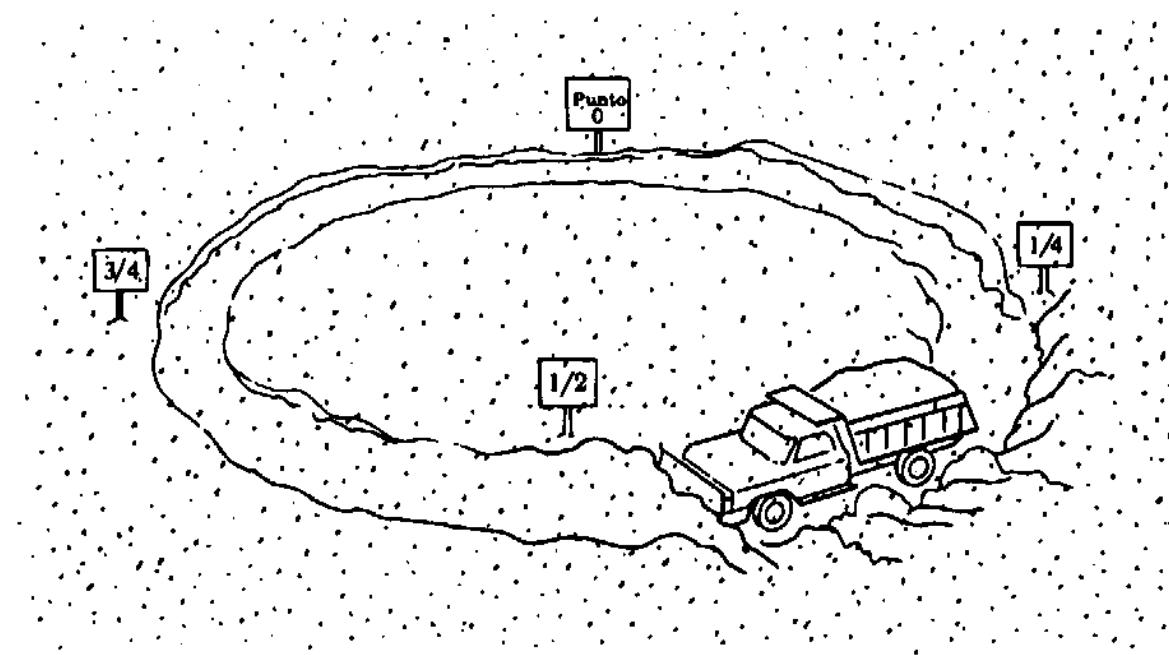
8.4.3 LONGITUD PROMEDIO DE LAS PORCIONES PARA LA SELECCION POR REEMPLAZO

La respuesta a la primera pregunta es que puede esperarse una longitud media de porción de $2P$, dadas P localidades de memoria. Knuth [1973b][†] proporciona una excelente descripción de un argumento intuitivo para explicar esto:

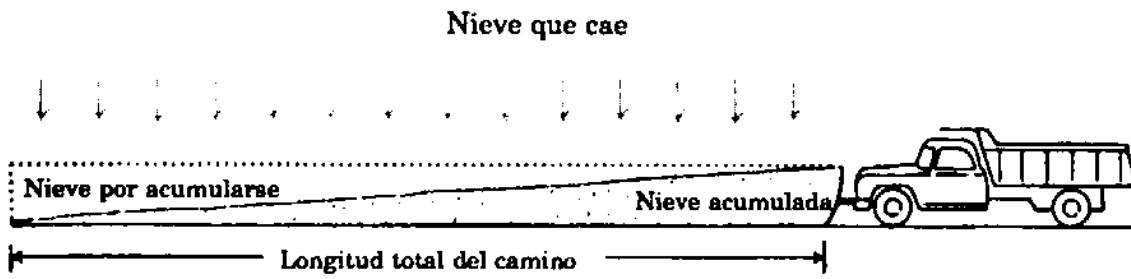
Una forma inteligente de demostrar que $2P$ es en realidad la longitud de porción prevista fue descubierta por E.F Moore, quien comparó la situación con una máquina para quitar la nieve en una pista circular [*Patente U.S. 2983904* (1961), Cols. 3-4]. Considere la situación que se ilustra [página siguiente]; los copos de nieve caen uniformemente en un camino circular, y una sola máquina está limpiando continuamente la nieve. Cuando la nieve ha sido retirada del camino desaparece del sistema. Los puntos del camino pueden designarse por números reales $x, 0 \leq x < 1$; un copo de nieve que cae en la posición x representa un

[†] De Donald Knuth, *The Art of Computer Programming*, ©1973, Addison-Wesley, Reading, Mass. Páginas 254-255 y figuras 64 y 65. Reimpreso con permiso.

registro de entrada cuya llave es x , y la máquina representa la salida de la selección por reemplazo. La velocidad de la máquina es inversamente proporcional a la altura de la nieve que encuentra, y la situación está perfectamente balanceada, de tal forma que la cantidad total de nieve en el camino es siempre exactamente P . Se forma una nueva porción en la salida cada vez que la máquina pasa por el punto 0.



Después de que este sistema ha estado en operación por un periodo, se intuye claramente que llegará a una situación estable, donde la máquina se mueva a una velocidad constante (debido a la simetría circular de la pista). Esto significa que la nieve está a una altura constante cuando se encuentra con la pala de la máquina, y la altura desciende linealmente en el frente de la pala de la máquina como se muestra [abajo]. De esto se infiere que el volumen de nieve removido en una revolución (es decir, la longitud de la porción) es dos veces la cantidad presente en cualquier otro momento (o sea P).



Así, tomando en cuenta un orden aleatorio de llaves, puede esperarse que la selección por reemplazo forme porciones que contengan alrededor del doble de los registros que se pueden almacenar en memoria a un mismo tiempo. Se deduce que la selección por reemplazo crea la mitad de las porciones que lo que logra una serie de clasificaciones en memoria RAM del contenido de la memoria, suponiendo que la selección por reemplazo y la clasificación en memoria RAM tienen acceso a la misma cantidad de memoria. (Como se verá en un momento, la selección por reemplazo, de hecho, debe arreglárselas con menos memoria que la clasificación en memoria RAM.)

En realidad, con frecuencia es posible crear porciones que sean considerablemente más largas de $2P$. En muchas aplicaciones, el orden de los registros *no* es del todo aleatorio; las llaves con frecuencia ya están parcialmente en orden ascendente. En estos casos la selección por reemplazo puede producir porciones que, en promedio, exceden de $2P$. (Considérese lo que sucedería si la lista de entrada ya estuviera clasificada.) La selección por reemplazo es una herramienta especialmente valiosa para tales archivos de entrada parcialmente ordenados.

8.4.4 COSTO DEL USO DE LA SELECCION POR REEMPLAZO

Por desgracia, la regla de "nada es gratis" se aplica a la selección por reemplazo, como sucede en tantas otras áreas del diseño de estructuras de archivos. En los ejemplos hechos a mano examinados hasta ahora, se han introducido los registros en la memoria uno por uno. Se sabe, de hecho, que el costo del desplazamiento en disco para cada registro de entrada es prohibitivo. En lugar de esto se pretende introducir la entrada en un buffer, lo que significa, a su vez, que no es posible usar toda la memoria para la operación de selección por reemplazo; parte de ésta tiene que usarse para el manejo de buffers de entrada y salida. (Como la selección por reemplazo siempre necesita introducir exactamente un registro después de que se envía uno a la salida, puede emplearse un solo buffer para ambas, la entrada y la salida.)

Para estudiar los efectos que tiene esta necesidad de manejar buffers durante el paso de selección por reemplazo, se recurre una vez más al ejemplo en donde se clasifican 400 000 registros, considerando un área de memoria que puede almacenar sólo 10 000. En el análisis anterior de este ejemplo se contaron sólo los desplazamientos requeridos durante el paso de intercalación, una vez que se habían formado las porciones. Ahora también se tomará en cuenta el desplazamiento requerido para transferir los registros del disco a la memoria, para la formación inicial de las porciones.

Para los métodos de clasificación en memoria RAM, en donde simplemente se transfieren registros a la memoria hasta que se llene, pueden efectuarse lecturas secuenciales de 10 000 registros a la vez. Esto significa que la lectura de todo el archivo requiere sólo 40 desplazamientos. (Se supone aquí que se trabaja en un sistema multiusuario, y que cuando la lectura termina, la solicitud de algún otro usuario ocasiona que haya un desplazamiento en el disco hacia alguna otra localidad.)

Para la selección por reemplazo puede usarse un buffer de E/S que permita almacenar, por ejemplo 2500 registros, dejando suficiente espacio para almacenar 7500 para el proceso en sí de selección por reemplazo. En consecuencia, se necesitan 160 desplazamientos para leer completo el archivo de 400 000 registros. Si los registros ocurren con una secuencia de llave aleatoria, la longitud media de la porción será de 15 000 registros.

La tabla 8.1 compara el número de desplazamientos requeridos para ordenar los 400 000 registros empleando una clasificación en memoria RAM, como *quicksort*, y la selección por reemplazo. La tabla incluye la intercalación inicial de 40 formas, la intercalación múltiple mejorada y dos ejemplos de selección por reemplazo. El segundo ejemplo de selección por reemplazo, que produce porciones de 40 000 registros usando sólo 7500 localidades de almacenamiento de registros en memoria, supone que existe ya un buen grado de ordenamiento secuencial dentro de los registros de entrada.

Es claro que, aun tomando en cuenta una distribución aleatoria de los datos de entrada, la selección por reemplazo puede reducir considerablemente el número de porciones formadas; en este caso la reducción es de 40 a 27. Pero la consecuente reducción del esfuerzo por los desplazamientos requeridos para intercalar las porciones casi se pierde por la cantidad de desplazamientos requeridos para formar las porciones. Los 160 desplazamientos requeridos para formar las porciones llevan el número total de desplazamientos para toda la clasificación de selección por reemplazo a 583, mientras que el proceso basado en clasificaciones sucesivas en memoria RAM requiere un total de 560. Sólo cuando se supone que los datos originales poseen suficiente orden como para hacer las porciones cuatro veces más grandes que las producidas por la clasificación en memoria RAM sucede que la selección por reemplazo requiere considerablemente menos desplazamientos.

En la tabla 8.1 también se considera lo que sucede si el archivo de entrada original sin clasificar proviene de cinta en lugar de disco. Si la entrada es de cinta, entonces no se requieren desplazamientos para formar las porciones iniciales; la entrada es verdaderamente secuencial de la cinta al buffer. La quinta columna de la tabla, Número total de desplazamientos requeridos para formar porciones, es cero en cada

TABLA 8.1
Comparación de formación de porciones por
medio de clasificaciones sucesivas en memoria
RAM y por medio de la selección por reemplazo.
(Considera sólo los desplazamientos requeridos para la entrada.)

Método	Número de registros leídos a la vez para formar porciones	Tamaño de las porciones para formar porciones	Número de las porciones formadas	Número total de desplazamientos requeridos para formar porciones	Patrón de intercalación empleado	Número total de desplazamientos requeridos para la intercalación	Total de desplazamientos (formación de porciones más intercalación)
40 clasificaciones en memoria RAM seguidas por una intercalación en 40 formas	10 000	10 000	40	40	40	1600	1640
40 clasificaciones en memoria RAM seguidas por una intercalación de varios pasos	10 000	10 000	40	40	8:8:8:8:8	520	560
Selección por reemplazo seguida por una intercalación de varios pasos (los registros están en orden aleatorio)	2 500	15 000	27	160	5:5:5:6:6	423	583
Selección por reemplazo seguida por una intercalación de varios pasos (los registros están parcialmente ordenados)	2 500	40 000	10	160	3:3:4	256	416

caso. Está claro que, para la entrada por cinta, la selección por reemplazo es mejor que el método de clasificación en memoria RAM. Esta asociación positiva entre la selección por reemplazo y las operaciones en cinta se analiza en la sección 8.5, cuando se examinan las clasificaciones que no usan el disco para nada.

8.4.5 CONFIGURACIONES DE DISCO

Puesto que el tiempo de desplazamiento parece ser la principal fuente del retraso, conviene preguntar si existen situaciones en las que no intervenga. Se ha visto que una forma de reducir el número de desplazamientos es reducir el número de porciones. La razón de que esto funcione es que la intercalación de menos porciones requiere que la cabeza de lectura y escritura se mueva de un archivo a otro con menor frecuencia. Esto sugiere otro método para reducir el número de desplazamientos, que consiste en incrementar el número de cabezas independientes de lectura y escritura.

Si hubiera una cabeza de lectura y escritura para cada archivo y ningún otro archivo compitiera por su uso, no habría retraso ocasionado por el tiempo de desplazamiento una vez que se hubieran generado las porciones originales. La principal fuente del retraso serían ahora los retrasos por rotación, que ocurrirían cada vez que se leyera un bloque nuevo. Al igual que el número de desplazamientos, el número de retraso por rotación puede reducirse disminuyendo el orden de intercalación (incrementando el tamaño del buffer interno), así que continua generalmente siendo ventajoso mantener bajo el orden de intercalaciones.

8.4.6 RESUMEN DE LA CLASIFICACION EN DISCO

Antes de pasar al problema de la clasificación en cinta se listan algunos comentarios sumarios sobre la clasificación en disco.

- Si la clasificación en disco implica una intercalación de más de cinco porciones, probablemente sea ventajoso efectuar la intercalación en dos o más pasos. Por ejemplo, una intercalación de seis formas se maneja probablemente mejor como una intercalación 3:3: un par de intercalaciones de tres formas seguida de una intercalación final de dos formas de las porciones intermedias resultantes.
- Cuando los datos de entrada provienen de un disco, es probable que no convenga el uso de la selección por reemplazo para formar las porciones iniciales, al contrario de lo que sucede con varias clasifica-

ciones en memoria RAM, a menos que ya exista un grado significativo de orden en los datos de entrada.

- / Si los datos pueden distribuirse entre varios discos y cada disco tiene un brazo de acceso separado, el desempeño puede mejorarse considerablemente, sobre todo cuando no hay otros usuarios del sistema. (En muchas instalaciones es posible procurar un sistema con poco uso cuando es necesario efectuar clasificaciones muy grandes.)
- Si el archivo de entrada proviene de cinta, es posible emplear ventajosamente la selección por reemplazo.

8.5

CLASIFICACION DE ARCHIVOS EN CINTA MAGNETICA

La mayoría de las instalaciones de computadores de tamaño pequeño a mediano pueden hacer más clasificaciones en disco que en cinta por la simple razón de que la mayoría de las instalaciones de este tamaño tienen sólo una o dos unidades de cinta. Sin embargo, si se tienen varias unidades de cinta, con frecuencia es posible clasificar archivos grandes más rápido en cinta que en disco, particularmente cuando hay seis o más unidades de cinta. Por supuesto, la mejora está relacionada con el costo de los desplazamientos en que se incurre cuando se trabaja en disco.

Existen muchos métodos para clasificar archivo en cinta. Después de aproximadamente 100 páginas de minuciosos análisis de las distintas alternativas de clasificación en cinta, Knuth [1973b] concluye su estudio en la siguiente forma:

Teorema A. Es difícil decidir qué patrón de intercalación es el mejor en una determinada situación.

Considerando el grado de complejidad y el número de métodos alternativos que existen, así como la forma en que las alternativas dependen de las características del hardware en una instalación en particular, el objetivo aquí es meramente comentar algunos de los aspectos fundamentales asociados con la clasificación e intercalación en cinta. Para un análisis más amplio (y más matemático) de alternativas específicas, se recomienda el trabajo de Knuth [1973b] como punto de partida.

Vistos desde una perspectiva general, los pasos de la clasificación en cinta se asemejan a los que se analizaron al estudiar la clasificación en disco:

1. Distribuir el archivo sin clasificar en *porciones* clasificadas, e
2. Intercalar las porciones dentro de un solo archivo clasificado.

La selección por reemplazo es casi siempre una buena elección como método para crear las porciones iniciales durante la clasificación en cinta. Debe recordarse que el problema de la selección por reemplazo, cuando se trabaja en disco, es que el número de desplazamientos requeridos durante la creación de la porción contrarresta la ventaja que tiene la creación de porciones más largas. Como se señaló con anterioridad, este problema de desplazamiento desaparece cuando la entrada proviene de cinta, de modo que, para una clasificación de cinta a cinta, casi siempre es aconsejable aprovechar la ventaja que ofrecen las porciones grandes que se crean con la selección por reemplazo.

8.5.1 INTERCALACION BALANCEADA

Dado que la pregunta sobre cómo crear las porciones iniciales tiene una respuesta tan directa, está claro que es durante el proceso de intercalación cuando se enfrentan todas las decisiones y complicaciones que entraña el práctico teorema de Knuth. Las decisiones empiezan con la pregunta sobre cómo *distribuir* las porciones iniciales en cinta y continuar con las dudas sobre el proceso de intercalación a partir de esta distribución inicial. Para explicar mejor esto, de nuevo se recurre al ejemplo de un archivo que contiene 400 000 registros.

En la tabla 8.1 se resumen algunos análisis de la clasificación en disco; en el último ejemplo se usó la selección por reemplazo en un conjunto de registros que ya estaban parcialmente ordenados. Debido a la ordenación, la selección por reemplazo es capaz de meter 400 000 registros en diez porciones de 40 000 registros. Se examinan varios métodos distintos para intercalar estas porciones en cinta, suponiendo que el sistema de cómputo tiene cuatro unidades de cinta. Puesto que el propio proceso de selección por reemplazo usa una de las unidades, se decidió distribuir inicialmente las diez porciones en dos o en tres de las demás unidades. Se comienza con un método llamado *intercalación balanceada*, el cual requiere una distribución inicial en dos unidades. La intercalación balanceada es el algoritmo más simple de intercalación en cinta que se considera; también, como se verá, es el más lento.

La intercalación balanceada procede de acuerdo con el patrón ilustrado en la figura 8.24.

Paso 1. Las porciones iniciales están distribuidas uniformemente en las unidades de cinta 1 y 2, así que hay cinco en cada cinta.

Paso 2. Al trabajar en forma secuencial coordinada, se intercala cada

una de las cinco parejas de porciones conforme se recorren las cintas, escribiendo las porciones más grandes resultantes en forma alterna en las cintas 3 y 4.

Paso 3. Se intercalan las porciones de las cintas 3 y 4, transcribiendo el resultado nuevamente a las cintas 1 y 2. Nótese que hay una porción más en la cinta 3 que en la 4. Cuando esto ocurre, la porción adicional (P9-P10) se intercala con una porción vacía, copiándola por tanto en la cinta destino.

Paso 4. Se intercalan las porciones de las cintas 1 y 2, transcribiendo los resultados a las cintas 3 y 4. De nuevo, la porción adicional (P9-P10) se intercala con una porción vacía para terminar con una porción grande en la cinta 3 y una corta en la cinta 4.

Paso 5. Se hace una intercalación final para producir una sola porción constituida por el archivo completo.

Este proceso de intercalación balanceada se expresa en una forma más compacta en la figura 8.25. Los números dentro de la tabla son las

	Cinta	Contiene las porciones					
Paso 1	C1	P1	P3	P5	P7	P9	
	C2	P2	P4	P6	P8	P10	
	C3	—					
	C4	—					
Paso 2	C1	—					
	C2	—					
	C3	P1-P2	P5-P6	P9-P10			
	C4	P3-P4	P7-P8				
Paso 3	C1	P1-P4	P9-P10				
	C2	P5-P8					
	C3	—					
	C4	—					
Paso 4	C1	—					
	C2	—					
	C3	P1-P8					
	C4	P9-P10					
Paso 5	C1	P1-P10					
	C2	—					
	C3	—					
	C4	—					

FIGURA 8.24 • Intercalación balanceada de cuatro cintas de diez porciones.

	C1	C2	C3	C4	
Paso 1	1 1 1 1 1	1 1 1 1 1	—	—	
Paso 2	—	—	2 2 2	2 2	Intercalar diez porciones
Paso 3	4 2	4	—	—	Intercalar diez porciones
Paso 4	—	—	8	2	Intercalar diez porciones
Paso 5	10	—	—	—	Intercalar diez porciones

FIGURA 8.25 • Intercalación balanceada de cuatro cintas de diez porciones expresadas en una tabla con notación más compacta.

longitudes de porción medidas en términos del número de porciones iniciales incluidas en cada porción intercalada. Por ejemplo, en el paso 1 todas las porciones consisten en una sola porción inicial. Para el paso 2, cada una está constituida por un par de porciones iniciales. Al inicio del paso 3, la unidad de cinta C1 contiene una porción compuesta por cuatro iniciales, seguida de una compuesta por dos iniciales. Este método de ilustración muestra más claramente la forma en que algunas de las porciones intermedias se combinan y crecen en porciones de longitudes 2, 4, y 8, mientras que la porción individual que se copia una y otra vez permanece con longitud 2 hasta el final. La forma empleada en esta ilustración es la que se usa aquí al analizar los métodos alternos de intercalación.

8.5.2 INTERCALACION EN VARIAS FASES

El algoritmo de intercalación balanceada tiene la ventaja de ser muy simple; es fácil escribir un programa para que lo efectúe. Desgraciadamente, una de las razones de su sencillez es que es "tonto" y no puede aprovechar oportunidades para ahorrar trabajo. Como puede observarse, la intercalación balanceada debe leer y escribir 40 porciones del tamaño inicial de 40 000 registros. En seguida se verá cómo puede mejorarse esto.

Para principiar, obsérvese que cuando se intercala la porción adicional (P9-P10) con las porciones vacías en los pasos 3 y 4, en realidad no se obtiene nada. La figura 8.26 muestra cómo puede reducirse drásticamente la cantidad de trabajo requerido simplemente por no copiar la porción adicional entre los pasos 2 y 3. En lugar de intercalarlo con

	C1	C2	C3	C4	
Paso 1	11111	11111	—	—	
Paso 2	—	—	2 2 2	2 2	Intercalar diez porciones
Paso 3	4	4	. . 2	—	Intercalar ocho porciones
Paso 4	—	—	—	10	Intercalar diez porciones

FIGURA 8.26 • Modificación de la intercalación balanceada de cuatro cintas que no rebobina la cinta 3 entre los pasos 2 y 3 para evitar la copia de porciones.

una porción simulada, simplemente se detiene la cinta 3 donde esté. Las cintas 1 y 2 contienen ambas ahora una única y larga porción (160 000 registros) compuesta por cuatro iniciales. Se rebobinan todas las cintas, excepto C3, y después se efectúa una intercalación de tres formas de las porciones en las cintas C1, C2 y C3, y se escribe el resultado final en C4. Al agregar este razonamiento al procedimiento de intercalación se reduce de 40 a 28 el número de porciones iniciales que deben leerse y escribirse.

El ejemplo de la figura 8.26 indica claramente que existen formas de mejorar el desempeño de la intercalación balanceada. Es importante poder establecer, en términos generales, qué es lo que ahorra trabajo en este segundo patrón de intercalación:

- Se usa una intercalación de orden más alto. En lugar de intercalaciones de dos formas, se usa una intercalación de tres formas.
- Se extiende la intercalación de porción de una cinta a varios pasos. Específicamente, se intercalan algunas de las porciones de C3 en el paso 3 y algunas en el paso 4. Se diría que se intercalan las porciones de C3 en dos fases.

Nótese que estos dos aspectos están relacionados: sin la posibilidad de hacer una intercalación de tres formas, no tendría sentido guardar la última parte de C3 para usarla en una segunda fase. Por el contrario, si no se deja parte alguna de C3 sin intercalar, no hay oportunidad de hacer una intercalación de tres formas.

Estas ideas, el uso de patrones de intercalación de orden más alto y la intercalación de porción de una cinta en *fases*, son la base de dos métodos bien conocidos para la intercalación llamados *intercalación polifásica* e *intercalación en cascada*. En general, estas intercalaciones comparten las siguientes características:

	C1	C2	C3	C4	
Paso 1	1 1 1 1 1	1 1 1	1 1	—	
Paso 2	.. 1 1 1	.. 1	—	3 3	Intercalar seis porciones
Paso 3	... 1 1	—	5	. 3	Intercalar cinco porciones
Paso 4 1	4	5	—	Intercalar cuatro porciones
Paso 5	—	—	—	10	Intercalar diez porciones

FIGURA 8.27 • Intercalación polifásica de cuatro cintas de diez porciones.

- La distribución inicial de porción es tal que al menos la intercalación inicial es de $J-1$ formas, donde J es el número de unidades de cinta disponibles.
- La distribución de las porciones a lo largo de las cintas es tal que las cintas con frecuencia contienen diferentes números de porciones.

La figura 8.27 ilustra cómo puede utilizarse una intercalación polifásica para intercalar las diez porciones distribuidas en cuatro unidades de cinta. Este patrón de intercalación reduce a 25 el número de porciones iniciales que deben leerse y escribirse. Es fácil observar que esta reducción es consecuencia del uso de varias intercalaciones de tres formas en lugar de las de dos formas de los ejemplos anteriores. También debe quedar clara que la posibilidad de hacer estas operaciones como intercalaciones de tres formas está relacionada con la naturaleza no uniforme de la distribución inicial. Por ejemplo, considérese lo que sucede si la distribución inicial de porción es 4-3-3 en vez de 5-3-2. Pueden efectuarse tres intercalaciones de tres formas para abrir espacio en C3, pero esto también elimina todas las porciones de C2 y deja sólo una en C1. Obviamente, no puede efectuarse otra intercalación de tres formas como segundo paso.

En este punto aparecen varias cuestiones:

1. ¿Cómo se elige una distribución inicial que conduzca fácilmente un patrón de intercalación eficiente?
2. Dada una distribución inicial, ¿existen descripciones algorítmicas de los patrones de intercalación?
3. Considerando N tiradas y J unidades de cinta, ¿existe alguna

forma de calcular el desempeño óptimo de la intercalación, de tal forma que se tenga un patrón para comparar la eficiencia de cualquier algoritmo específico?

Las respuestas precisas a estas preguntas rebasan el alcance de este texto; en particular, la respuesta a la pregunta 3 implica enfocar el problema desde un punto de vista matemático. A los lectores que no se conformen con un razonamiento intuitivo para definir las distribuciones iniciales, se les recomienda consultar Knuth [1973b].

8.6

PAQUETES DE CLASIFICACION E INTERCALACION

Existen muchos buenos programas de servicio para usuarios que necesiten clasificar archivos grandes. Con frecuencia los programas tienen suficiente inteligencia para seleccionar alguna estrategia, dependiendo de la naturaleza de los datos que van a clasificarse y de la configuración disponible del sistema. Con frecuencia también permiten a los usuarios ejercer algún control (si así lo desean) en la organización de los datos y las estrategias usadas. En consecuencia, aun cuando el lector esté empleando un paquete comercial de clasificación en lugar de diseñar su propio procedimiento, resulta útil familiarizarse con la variedad de formas distintas para diseñar intercalaciones y clasificaciones. Es muy recomendable tener un conocimiento general de los factores más importantes y de las limitaciones que efectúan el desempeño.

RESUMEN

En la primera mitad del capítulo se desarrolla un modelo de procesamiento secuencial coordinado y se aplica a dos problemas comunes: la actualización de un libro mayor y la clasificación por intercalación. En la segunda mitad del capítulo se identifican los factores más importantes que influyen en la eficiencia de las operaciones de intercalación y clasificación, y se sugieren algunas estrategias para lograr un buen desempeño.

El modelo de procesamiento secuencial coordinado puede aplicarse a problemas que impliquen operaciones tales como la correspondencia y la intercalación (y combinaciones de éstos) en dos o más archivos de entrada clasificados. Se comienza este capítulo ilustrando el uso del modelo para efectuar una correspondencia simple de los elementos comunes de dos listas, y una intercalación de dos listas. Los procedimientos que se desarrollan para efectuar estas dos operaciones incorporan todos los elementos básicos del modelo.

En su forma más acabada, el modelo depende de algunas suposiciones sobre los datos en los archivos de entrada. Se enumeran estas suposiciones en la descripción formal del modelo. Considerando estas suposiciones, se describen los componentes de procesamiento del modelo.

El valor real del modelo secuencial coordinado es que puede adaptarse sin demasiadas modificaciones a problemas más importantes que simples correspondencias o intercalaciones. Esto se ilustra usando el modelo para diseñar un programa de contabilidad de libro mayor.

Las primeras aplicaciones del modelo manejan sólo dos archivos de entrada. Luego se adapta el modelo a una intercalación de varias formas para mostrar cómo puede ampliarse a fin de operar con más de dos listas de entrada. El problema de encontrar el valor de la llave mínimo durante cada paso del ciclo principal se vuelve más complejo a medida que se incrementa el número de archivos de entrada. La solución implica reemplazar la proposición de selección de tres formas con una selección múltiple o con un procedimiento que guarde las llaves actuales en una estructura de listas que pueda procesarse en forma más conveniente.

Se observa que la aplicación del modelo a la intercalación de K formas funciona bien para valores pequeños de K , pero para valores de K mayores de ocho resulta más eficiente encontrar el valor de llave mínimo por medio de un árbol de selección.

Después de analizar la intercalación múltiple se aborda el problema que se presentó en el capítulo anterior. Cómo clasificar archivos grandes. La solución que generalmente se acepta cuando un archivo es demasiado grande para su clasificación en memoria RAM consiste en alguna forma de *clasificación por intercalación*. Una clasificación por intercalación implica dos pasos:

1. Dividir el archivo en dos o más subarchivos clasificados, o porciones, empleando métodos de clasificación internos, y
2. Intercalar las porciones.

Lo ideal sería almacenar cada porción en un archivo separado, de tal forma que pueda efectuarse el paso de intercalación de un recorrido por las porciones. Por desgracia, algunas consideraciones prácticas

dificultan la realización de esto. Dichas consideraciones pueden diferir, según se usen cintas o discos (o una combinación) para almacenar los archivos. Se principia examinando los factores que influyen en el desempeño del disco.

El elemento crítico cuando se intercalan muchos archivos en disco es el tiempo de desplazamiento. El número de desplazamientos depende en gran medida de dos factores interrelacionados: el número de porciones distintas que se intercalan, y la cantidad de espacio disponible de buffer interno para almacenar partes de las porciones. Puede reducirse el número de desplazamientos en dos formas:

- Efectuando la intercalación en más de un paso; e
- Incrementando los tamaños de las porciones iniciales clasificadas.

En ambos casos, el orden de cada paso de intercalación puede reducirse incrementando los tamaños de los buffers internos y permitiendo que procesen más datos por cada desplazamiento.

Elexaminar en primer lugar la alternativa inicial permite observar cómo hacer la intercalación en varios pasos puede reducir drásticamente el número de desplazamientos, aunque ello significa también que se necesitan leer los datos más de una vez (incrementando así el tiempo total de transmisión de los datos).

La segunda alternativa se realiza mediante un algoritmo llamado *selección por reemplazo*. La selección por reemplazo, que puede efectuarse usando el árbol de selección mencionado anteriormente, implica seleccionar de la memoria la llave cuyo valor sea el más bajo, enviar a la salida esa llave y reemplazarla con una nueva llave de la lista de entrada.

Con archivos organizados en forma aleatoria, puede esperarse que la selección por reemplazo produzca porciones dos veces más grandes que el número de localidades de almacenamiento interno disponibles para efectuar los algoritmos. Aunque esto representa un paso más hacia la reducción del número de porciones por intercalar, lleva consigo un costo adicional. La necesidad de un buffer grande para efectuar la operación de selección por reemplazo deja relativamente poco espacio para el buffer de E/S, lo que significa que se requieren muchos más desplazamientos para la formación de porciones que los que se necesitan cuando el paso de clasificación usa una clasificación en memoria RAM. Si se compara el número total de desplazamientos requeridos por los dos métodos, se observa que la selección por reemplazo en realidad puede requerir más; funciona considerablemente mejor sólo cuando hay un alto grado de orden en el archivo inicial.

Después se centra la atención en la clasificación en cintas magnéticas. Puesto que la E/S de archivos con cintas no implica desplaza-

mientos, los problemas y soluciones asociados con la clasificación en cinta pueden diferir de los asociados con clasificación en disco, aunque permanece el objetivo fundamental de trabajar con menos y mayores porciones. Con la clasificación en cinta, la medida principal de desempeño es el número de veces que cada registro debe transmitirse. (Otros factores, como el tiempo de rebobinado de la cinta, también pueden ser importantes, pero no se consideran aquí.)

Puesto que las cintas no requieren desplazamiento, la selección por reemplazo es casi siempre una buena elección para la creación de porciones iniciales. Como el número de unidades disponibles para almacenar los archivos de porciones está limitado, la siguiente cuestión es cómo distribuir los archivos en las cintas. En la mayoría de los casos, es necesario poner varias porciones en cada una de varias cintas, reservando una o más para los resultados. Esto, por lo general, lleva a intercalaciones de varios pasos, disminuyendo así el número total de porciones después de cada paso de intercalación. Dos métodos para lograr esto son las *intercalaciones balanceadas* y las *intercalaciones de varias fases*. En una intercalación balanceada de K formas, todas las cintas de entrada contienen aproximadamente el mismo número de porción, existe el mismo número de cintas de salida que de entrada, y las cintas de entrada se leen por completo durante cada paso. El número de porción disminuye en un factor de K después de cada paso.

Una intercalación de varias fases (tal como la *intercalación polifásica* o la *intercalación en cascada*) requiere que las porciones se distribuyan inicialmente en forma no uniforme entre todas las cintas disponibles, excepto en una. Esto incrementa el orden de la intercalación y, como resultado, puede reducir el número de veces que debe leerse cada registro. Resulta que la distribución inicial de porciones entre el primer conjunto de cintas de entrada tiene un efecto importante en el número de veces que debe leerse cada registro.

Para concluir el capítulo se mencionan brevemente las utilerías de clasificación e intercalación, disponibles en la mayoría de los sistemas grandes, que pueden ser muy flexibles y efectivas.

TERMINOS CLAVE

Arbol de selección. Arbol binario donde cada nodo de nivel superior representa al ganador en la comparación entre las dos llaves descendientes. El valor mínimo (o máximo) en el árbol de selección está siempre en el nodo raíz, lo que lo convierte en una

buenas estructuras de datos para la intercalación de varias listas. También es una estructura central en los algoritmos de selección por reemplazo, los cuales pueden emplearse para producir porciones grandes para clasificaciones por intercalación. (La *clasificación por torneo*, una clase de clasificación interna, también está basada en el uso de un árbol de selección.)

Ciclo de sincronización. Ciclo principal en el modelo de procesamiento secuencial coordinado. Una característica esencial del modelo es que realiza toda la sincronización dentro de un solo ciclo, en lugar de hacerlo en varios ciclos anidados. Un segundo objetivo es mantener el ciclo principal de sincronización tan simple como sea posible. Esto se logra restringiendo las operaciones que ocurren dentro del ciclo a las que involucran a las llaves actuales, y dejando, en subprocedimiento tanta lógica especial como sea posible (como la revisión de errores y la revisión del final de archivo).

Correspondencia. Proceso de formar un archivo de salida clasificado compuesto por todos los elementos comunes en dos o más archivos de entrada clasificados.

Intercalación. Proceso de formar un archivo de salida clasificado compuesto por una unión de los elementos de dos o más archivos de entrada clasificados.

Intercalación balanceada. Técnica de intercalación de varios pasos que usa el mismo número de dispositivos de entrada que de salida. Una intercalación de dos formas emplea dos cintas, cada una con aproximadamente el mismo número de porciones y produce dos cintas de salida, cada una con cerca de la mitad de porción que el número de cintas de entrada. La intercalación balanceada es adecuada para la clasificación por intercalación con cintas, aunque generalmente no es el mejor método (veáse la intercalación de varias fases).

Intercalación de varias fases. Intercalación de varios pasos en cinta en la que la distribución de las porciones es tal que al menos la intercalación inicial es de $J-1$ formas (J es el número de unidades de cinta disponibles), y donde la distribución de las porciones en las cintas es tal que la intercalación funciona eficientemente en cada paso. (Véase intercalación polifásica.)

Intercalación de varios pasos. Intercalación en la que no todas las porciones se intercalan en un paso. En lugar de esto, se intercalan separadamente varios conjuntos de porciones, produciendo cada conjunto una porción grande constituida por los registros de todas sus porciones. Estos conjuntos nuevos y mayores son intercalados entonces, todos unidos o en varios conjuntos. Después de cada paso, el número de porciones decrece

y su longitud aumenta. La salida del paso final es una sola porción constituida por el archivo completo. (Hay que tener cuidado de no confundir el uso del término *intercalación de varios pasos* con *intercalación de varias fases*.)

Aunque una intercalación de varios pasos en teoría consume más tiempo que una de un solo paso, puede implicar un número mucho menor de desplazamientos cuando se efectúa en disco, y puede ser la única manera razonable de efectuar una intercalación en cinta si el número de unidades es limitado.

Intercalación en K formas. Intercalación en la que K archivos de entrada se intercalan para producir un archivo de salida.

Intercalación polifásica. Intercalación de varias fases en la que, en forma ideal, el orden de la intercalación se maximiza en cada paso.

Operaciones secuenciales coordinadas. Operaciones aplicadas a problemas que implican la realización de uniones, intersecciones y algunas operaciones de conjuntos más complejas en dos o más archivos de entrada clasificados, para producir uno o más archivos de salida construidos a partir de alguna combinación de los elementos de los archivos de entrada. Las operaciones secuenciales coordinadas comúnmente ocurren en problemas de correspondencia, intercalación y actualización de archivos.

Porción. Subconjunto clasificado de un archivo, que resulta de un paso de clasificación de una clasificación por intercalación o de uno de los pasos de una intercalación de varios pasos.

Revisión de secuencia. Revisión de que el orden de los registros en un archivo sea el correcto. Se recomienda que en todos los archivos usados en una operación secuencial coordinada se revise la secuencia.

Selección por reemplazo. Método de creación de porciones iniciales basado en la idea de *seleccionar* siempre de la memoria el registro cuya llave tenga el valor más bajo, enviar a la salida ese registro, y después *reemplazarlo* en memoria con un nuevo registro de la lista de entrada. Cuando se obtienen registros nuevos cuyas llaves son mayores que las de los registros enviados más recientemente a la salida, finalmente forman parte de la porción que se está creando. Cuando los registros nuevos tienen llaves menores que las de los enviados más recientemente a la salida, se guardan para la siguiente porción.

La selección por reemplazo produce por lo general porciones considerablemente más grandes que las que pueden crearse mediante clasificaciones en memoria RAM, y por tanto pueden ayudar a mejorar el desempeño en la clasificación por intercalación. Sin embargo, al usar la selección por reemplazo

con clasificaciones e intercalaciones en disco, se debe cuidar que los desplazamientos adicionales requeridos no eliminen los beneficios que aportan el tener porciones más grandes por intercalar.

VALOR_ALTO. Valor usado en el modelo secuencial coordinado, que es mayor que cualquier valor posible de llave. Al asignar VALOR_ALTO como el valor de la llave actual para archivos en los que ha llegado al estado de final de archivo, la lógica adicional para enfrentar este estado puede simplificarse.

VALOR_BAJO. Valor usado en el modelo secuencial coordinado, que es menor que cualquier valor posible de llave. Al asignar VALOR_BAJO como el valor de la llave anterior durante la asignación de valores iniciales, se elimina la necesidad de algún otro código especial de arranque.



EJERCICIOS

1. Escriba un procedimiento de salida compatible con los descritos en la sección 8.1 para realizar la correspondencia secuencial coordinada. Como medida defensiva, es buena idea hacer que el procedimiento de salida haga una revisión de secuencia de la misma manera en que lo hace el de entrada.

2. Considere la rutina secuencial coordinada de asignación de valores iniciales de la figura 8.4. Si PREV_1 y PREV_2 no se igualan a VALOR_BAJO en esta rutina, ¿cómo tendría que modificarse *entrada()*? ¿Cómo afectaría esto la adaptabilidad de *entrada()* para su utilización en otros algoritmos de procesamiento secuencial coordinado?

3. Considere los procedimientos de intercalación secuencial coordinada descritos en la sección 8.1. Comente como manejan las siguientes situaciones. Si no manejan en forma adecuada una situación, indique cómo pueden modificarse para que lo hagan.

- La lista 1 está vacía y la lista 2 no.
- La lista 1 no está vacía y la lista 2 sí.
- La lista 1 está vacía y la lista 2 también.

4. En el ejemplo de procedimiento de libro mayor de la sección 8.2, modifíquelo de tal forma que también actualice el archivo de libro mayor con los nuevos balances de cuenta mensuales.

5. Use el ejemplo de intercalación en K formas como base de un procedimiento que realice correspondencias en K formas.
6. La figura 8.17 muestra un ciclo para intercalación en K formas, suponiendo que no hay nombres duplicados. Si se permiten nombres duplicados, podría agregarse al procedimiento un recurso para mantener una lista de subíndices de los nombres duplicados más bajos. Modifique el procedimiento para que lo haga.
7. En la sección 8.3 se presentaron dos métodos para elegir la más baja de K llaves en cada paso de una intercalación en K formas: una búsqueda lineal y el uso de un árbol de selección. Compare la eficiencia de los dos métodos en términos del número de comparaciones para $K = 2, 4, 8, 16, 32$, y 100. ¿Por qué se recomienda el método lineal para valores de K menores de 8?
8. ¿Cuánto tiempo de desplazamiento se requiere para efectuar una intercalación de un paso en K formas como la que se describió en la sección 8.4, si el tiempo de un desplazamiento en promedio es de 50 msec y la cantidad de espacio disponible para buffers internos es de 500K y de 100K?
9. El desempeño en la clasificación con frecuencia se mide en términos del número de comparaciones. Explique por qué el número de comparaciones no es una medida de desempeño adecuada en la clasificación de archivos grandes.
10. Derive dos fórmulas para el número de desplazamientos requeridos para efectuar el paso de intercalación de una clasificación por intercalación de un paso en K formas de un archivo con r registros divididos en K porciones, donde la cantidad de memoria RAM disponible es equivalente a M registros. Si se usa una clasificación interna, como Quicksort, para la fase de clasificación, puede suponerse que la longitud de cada porción es M , pero si se usa la selección por reemplazo puede suponerse que la longitud de cada porción es de alrededor de $2M$. ¿Por qué?
11. Suponga un sistema de poca demanda con cuatro unidades de disco separadas cada una capaz de almacenar varios cientos de megabytes. Suponga que el archivo de 40 megabytes de 400 000 registros descrito en la sección 8.4 se encuentra en una de las unidades. Diseñe un procedimiento de clasificación para este archivo de ejemplo que use las unidades individuales para disminuir la cantidad de desplazamientos requeridos. Suponga que el archivo clasificado final se escribe en cinta

y que el manejo de buffers para la cinta es invisible debido al sistema operativo. ¿Se obtiene alguna ventaja usando la selección por reemplazo?

12. Use la selección por reemplazo para producir porciones a partir de los siguientes archivos, suponiendo que $P = 4$.

- a) 23 29 5 17 9 55 41 3 51 33 18 24 11 47
- b) 3 5 9 11 17 18 23 24 29 33 41 47 51 55
- c) 55 51 47 41 33 29 24 23 18 17 11 9 5 3

13. Suponga que se tiene una unidad de disco que posee diez cabezas de lectura/escritura por superficie, de modo que puede accederse a diez cilindros en cualquier momento sin tener que mover los brazos que las sostienen. Si pudiese controlar la organización física de las porciones almacenadas en disco, ¿cómo podría explotar esta disposición para la clasificación por intercalación?

14. Suponga que se necesita intercalar 14 porciones en cuatro unidades de cinta. Desarrolle patrones de intercalación comenzando a partir de cada una de estas distribuciones iniciales:

8-4-2
7-4-3
6-5-3
5-5-4

15. Debe efectuarse una intercalación polifásica de cuatro cintas para clasificar la lista 24 36 13 25 16 45 29 38 23 50 22 19 43 30 11 27. La lista original está en la cinta 4. Las porciones iniciales son de longitud 1. Después de la clasificación inicial, las cintas 1, 2 y 3 contienen las siguientes porciones (la diagonal separa las porciones).

Cinta 1 : 24 / 36 / 13 / 35
Cinta 2 : 16 / 45 / 29 / 38 / 23 / 50
Cinta 3 : 22 / 19 / 43 / 30 / 11 / 27 / 47

- a) Muestre el contenido de la cinta 4 después de una fase de intercalación.
- b) Muestre el contenido de las cuatro cintas después de la sexta y cuarta fases.
- c) Comente si la distribución original 4-6-7 es apropiada para efectuar una intercalación polifásica.

16. Obtenga una copia del manual de uno o más paquetes comerciales de clasificación e intercalación. Identifique los diferentes tipos de op-

ciones que se proporcionan a los usuarios. Relacione las opciones con los aspectos de desempeño analizados en este capítulo.

EJERCICIOS DE PROGRAMACION

17. Realice los procedimientos de correspondencia secuencial coordinada descritos en la sección 8.1, en C o en Pascal.

18. Realice los procedimientos de intercalación secuencial coordinada descritos en la sección 8.1, en C o en Pascal.

19. Realice un programa completo que corresponda a la solución del problema del libro mayor presentado en la sección 8.2.

20. Diseñe y realice un programa que haga lo siguiente:

- a) Examinar los contenidos de dos archivos clasificados, M1 y M2.
- b) Producir un tercer archivo, COMUN, que contenga una copia de los registros de los dos archivos originales que sean idénticos.
- c) Producir un cuarto archivo, DIFEREN, que contenga una copia de los registros de ambos archivos que sean diferentes.

LECTURAS ADICIONALES

El tema tratado en este capítulo puede dividirse en dos aspectos separados: la presentación de un modelo para el procesamiento secuencial coordinado, y el estudio de procesamientos externos de intercalación en cinta y en disco. Aunque la mayoría de los textos de procesamiento de archivos analizan el procesamiento secuencial coordinado, por lo común lo hacen en el contexto de aplicaciones específicas, en lugar de presentar un modelo general que pueda adaptarse a una diversidad de aplicaciones. Encontramos este modelo útil y flexible gracias al Dr. James Van Doren, quien lo desarrolló para los cursos de estructuras de archivos que dicta. No tenemos noticias de ningún estudio del modelo secuencial coordinado en la literatura disponible.

Es mucho lo que se ha trabajado con miras al desarrollo de algoritmos simples y efectivos para realizar la actualización secuencial de archivos, que es un aspecto importante del procesamiento secuencial coordinado. Los resultados enfrentan algunos de los mismos problemas que el modelo secuencial coordinado, y algunas soluciones son parecidas. Véase Levy [1982] y Dwyer [1981] para más detalles.

A diferencia del procesamiento secuencial coordinado, la clasificación externa es un tema que la literatura cubre ampliamente. El estudio más completo del tema, por mucho, está en Knuth [1973b]. Los estudiantes interesados en el tema de la clasificación externa deben familiarizarse, tarde o

temprano, con el resumen definitivo del tema hecho por Knuth. Knuth también describe la selección por reemplazo, como es evidente al haber citado su libro en este capítulo.

Lorin [1975] dedica varios capítulos a las técnicas de clasificación por intercalación. Bradley [1981] proporciona un buen tratamiento de la selección por reemplazo y la intercalación de varias fases, incluidas algunas comparaciones de tiempo de procesamiento en distintos dispositivos. Tremblay y Sorenson [1984], Loomis [1983] y Claybrook [1983] también tienen capítulos acerca de la clasificación externa.

Puesto que la clasificación de grandes archivos representa un gran porcentaje del tiempo de procesamiento, la mayoría de los sistemas disponen de utilería de clasificación. El DFSORT de IBM (descrito en IBM, 1985) es un paquete flexible para manejo de aplicaciones de clasificación e intercalación. Una utilería de clasificación para VAX se describe en Digital [1984].

OBJETIVOS

Ubicar el desarrollo de los árboles B en el contexto histórico de los problemas para cuya solución fueron diseñados.

Examinar brevemente otras estructuras de árboles que pueden usarse en almacenamiento secundario, tales como los árboles AVL paginados.

Proporcionar una explicación de las propiedades importantes de los árboles B, y demostrar que tales propiedades son especialmente convenientes para aplicaciones de almacenamiento secundario.

Describir las operaciones fundamentales en árboles B.

Introducir la idea de manejo de páginas en buffers y de árboles B virtuales.

Describir variaciones de los algoritmos fundamentales de árboles B, como los que se usan para construir B* y árboles B con registros de longitud variable.

9

ARBOLES B Y OTRAS ORGANIZACIONES DE ARCHIVOS ESTRUCTURADAS EN FORMA DE ARBOL

PLAN GENERAL DEL CAPITULO

- 9.1 Introducción. La invención de los árboles B**
- 9.2 Planteamiento del problema**
- 9.3 Los árboles binarios de búsqueda como solución**
- 9.4 Árboles AVL**
- 9.5 Árboles binarios paginados**
- 9.6 El problema con la construcción descendente de los árboles paginados**
- 9.7 Árboles B: construcción ascendente**
- 9.8 División y promoción**
- 9.9 Algoritmos para la búsqueda e inserción en árboles B**
- 9.10 Nomenclatura de árboles B**
- 9.11 Definición formal de las propiedades de los árboles B**
- 9.12 Profundidad de la búsqueda en el peor caso**
- 9.13 Eliminación, redistribución y concatenación**
 - 9.13.1 Redistribución**
- 9.14 Redistribución durante la inserción: una forma de mejorar la utilización del almacenamiento**
- 9.15 Árboles B***
- 9.16 Manejo de páginas en buffers: árboles B virtuales**
 - 9.16.1 Reemplazo LRU**
 - 9.16.2 Reemplazo según la altura de la página**
 - 9.16.3 Importancia de los árboles B virtuales**
- 9.17 Colocación de la información asociada con la llave**
- 9.18 Registros y llaves de longitud variable**
- Programa en C para inserción de llaves en un árbol B**
- Programa en Pascal para inserción de llaves en un árbol B**

9.1

INTRODUCCION. LA INVENCION DE LOS ARBOLES B

La ciencia de la computación es una disciplina joven. Como prueba de esta juventud, considérese que al inicio de 1970, después de que los astronautas había viajado dos veces a la Luna, los árboles B todavía no

existían. Hoy en día, sólo quince años después, es difícil imaginar un gran sistema de archivos de propósito general que no esté construido alrededor de un diseño de árboles B.

Douglas Comer, en su excelente artículo de divulgación "The Ubiquitous B-Tree" [1979], reseña la competencia entre los fabricantes de computadores y los grupos independientes de investigación que se desarrollaba a finales de los años 60. El objetivo era el descubrimiento de un método general para el almacenamiento y extracción de datos en grandes sistemas de archivos que proporcionara acceso rápido a los datos con costos mínimos. Entre los competidores estaban R. Bayer y E. McCreight, quienes trabajaban entonces para la corporación Boeing. En 1972 publicaron un artículo, "Organization and Maintenance of Large Ordered Indexes", donde se anuncian al mundo los árboles B. En 1979, cuando Comer publicó su artículo de divulgación, los árboles B ya eran tan usados que fue capaz de afirmar que "los árboles B, son, *de facto*, la organización estándar para índices en un sistema de base de datos."

Se reproducen aquí los primeros párrafos del artículo de 1972 de Bayer y McCreight[†] por la manera tan concisa en que se describen las facetas del problema que motivó el diseño de los árboles B: como mantener y acceder eficientemente a un índice que es demasiado grande para almacenarse en la memoria. El lector recordará que éste es el mismo problema que quedó sin resolver en el capítulo 7, sobre estructuras simples de índices. Quedará claro, conforme se lea la introducción de Bayer y McCreight, que su trabajo se dirige al núcleo de los aspectos que aparecieron en el capítulo de indización.

En este artículo se considera el problema de la organización y el mantenimiento de un índice de un archivo de acceso aleatorio que cambia dinámicamente. *Índice* se refiere a un conjunto de elementos de índice dispuestos en pares (x, a) de ítems de datos de tamaño fijo físicamente adyacentes, es decir, una llave x y alguna información asociada a . La llave x identifica un elemento único del índice, y la información asociada por lo regular es un apuntador a un registro o conjunto de registros en un archivo de acceso aleatorio. En este artículo la información asociada no es de mayor importancia.

Se supone aquí que el índice por sí mismo es tan voluminoso que sólo algunas pequeñas partes de él pueden mantenerse en el almacenamiento principal al mismo tiempo. De esta forma, el grueso del

[†]De *Acta Informatica*, 1:173-189, © 1972, Springer Verlag, Nueva York. Reimpreso con autorización.

índice debe mantenerse en algún almacenamiento de respaldo. Los tipos de almacenamientos de respaldo considerados son *dispositivos de acceso pseudoaleatorio*, con un tiempo de acceso o de espera más bien largo —a diferencia de un dispositivo con acceso aleatorio verdadero, como el almacenamiento principal— y una tasa de transmisión relativamente alta una vez que se ha iniciado la transmisión de los datos físicamente contiguos. Los dispositivos de acceso pseudoaleatorio usuales son discos con cabezas móviles y fijas, tambores y celdas de datos.

Puesto que el propio archivo de datos cambia, debe ser posible no sólo buscar el índice y extraer elementos, sino también eliminar e insertar llaves —o más precisamente elementos índice— en forma económica. La organización de índices descrita en este artículo permite la extracción, inserción y eliminación de llaves en un tiempo proporcional a $\log_k I$, o mejor, donde I es el tamaño del índice y k es un número natural que depende del dispositivo y que describe el tamaño de la página, de tal forma que el desempeño del esquema de mantenimiento y extracción sea casi óptimo.

Los ejercicios 17, 18 y 19, al final del capítulo 7, introducen la idea de un índice paginado. La indicación de Bayer y McCreight, de que desarrollaron un esquema con un tiempo de extracción proporcional a $\log_k I$, donde k está relacionada con el tamaño de la página, es muy significativa. Como se verá, el uso de un árbol B con un tamaño de página de 64, para indizar un archivo con un millón de registros, permite encontrar la llave de cualquier registro con no más de cuatro desplazamientos en disco. Una búsqueda binaria en el mismo archivo puede requerir más de 20. Además, se pretende obtener esta clase de desempeño a partir de un sistema que requiere sólo un mínimo de gastos cuando se insertan y eliminan llaves.

Antes de examinar en detalles la solución de Bayer y McCreight se hará un repaso más cuidadoso del problema, retomándolo como se dejó en el capítulo 7. También se analizan algunas estructuras de datos y de archivos que se usaban cotidianamente para atacar el problema antes de la invención de los árboles B. Tomando en consideración estos antecedentes será más fácil apreciar la contribución del trabajo de Bayer y McCreight.

Una última cuestión antes de comenzar: ¿Por qué el nombre de *árbol B*? Comer[1979] anota a pie de página lo siguiente:

El origen de “árbol B” nunca ha sido explicado por [Bayer y McCreight]. Como se verá, podría aplicarse “balanceado”, “amplio” (*broad*), o arborecente (*bushy*). Algunos sugieren que la “B” se refiere a Boeing. Sin embargo, parece apropiado pensar en los árboles B como árboles “Bayer”.

9.2

PLANTEAMIENTO DEL PROBLEMA

El problema fundamental de mantener un índice en almacenamiento secundario es, por supuesto, que el acceso al almacenamiento secundario es lento. Este problema fundamental puede dividirse en dos más específicos:

- *La búsqueda binaria requiere demasiados desplazamientos.* La búsqueda de una llave en disco con frecuencia implica desplazamiento a diferentes pistas del disco. Como los desplazamientos son costosos, una búsqueda que tiene que examinar más de tres o cuatro lugares antes de encontrara la llave con frecuencia requiere más tiempo de lo deseable. Si se emplea la búsqueda binaria, cuatro desplazamientos son apenas suficientes para diferenciar entre 15 registros. En un índice de 1000 ítems se requieren alrededor de 9.5 desplazamientos en promedio usando la búsqueda binaria. Es necesario encontrar la forma de encontrar una llave usando menos desplazamientos.
- *Puede ser muy costoso mantener el índice ordenado para que sea posible efectuar una búsqueda binaria.* Como se estudió en el capítulo 7, si insertar una llave implica mover muchas otras llaves del índice, el mantenimiento del índice resulta impráctico en almacenamiento secundario para índices que contienen sólo unas pocas centenas de llaves, y mucho más si se trata de miles. Es necesario encontrar una forma de hacer inserciones y eliminaciones cuyos efectos en el índice sean sólo locales, y que no impliquen una reorganización masiva.

Estos fueron los dos problemas críticos que afrontaron Bayer y McCreight en 1970, y sirven como guía para estudiar aquí el uso de las estructuras de árbol para la extracción del almacenamiento secundario.

9.3

LOS ARBOLES BINARIOS DE BUSQUEDA COMO SOLUCION

Se comenzará con el segundo de estos dos problemas, examinando el costo que representa mantener una lista en el orden requerido para efectuar búsquedas binarias. Para la lista clasificada de la figura 9.1 la búsqueda binaria puede expresarse como un árbol binario de búsqueda, como se muestra en la figura 9.2.

AX CL DE FB FT HN JD KF NR PA RF SD TK WA YJ

FIGURA 9.1 • Lista clasificada de llaves.

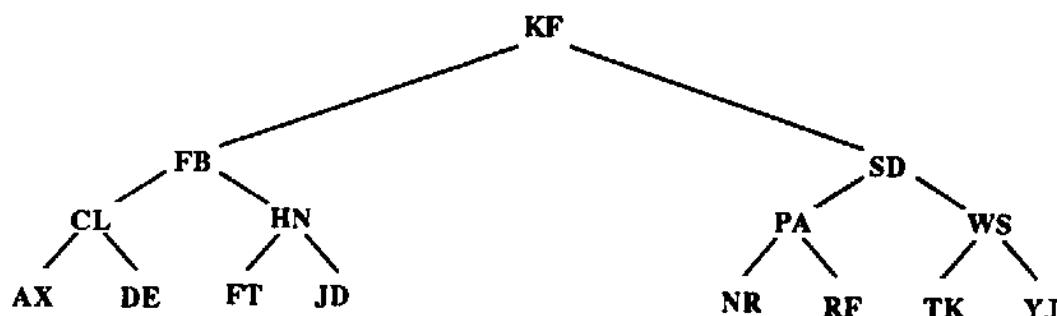


FIGURA 9.2 • Representación del árbol binario de búsqueda de la lista de llaves.

Usando técnicas de estructuras de datos elementales, es sencillo crear nodos que contengan campos de liga a la derecha y a la izquierda de tal forma que el árbol binario de búsqueda pueda construirse como una estructura ligada. La figura 9.3 ilustra una representación ligada de los dos primeros niveles del árbol binario de búsqueda que se muestra en la figura 9.2. En cada nodo, las ligas izquierda y derecha apuntan a los hijos izquierdo y derecho del nodo.

Si cada nodo se trata como un registro de longitud fija en el que los campos de liga contienen números relativos de registro (NRR) que apuntan a otros nodos, entonces es posible colocar dicha estructura de árbol en almacenamientos secundario. La figura 9.4 ilustra el contenido de los 15 registros que se requerirían para formar el árbol binario de la figura 9.2.

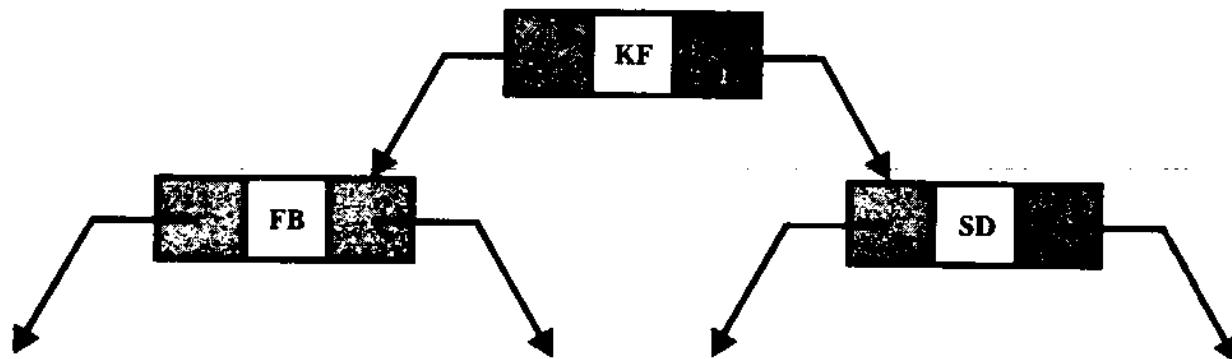


FIGURA 9.3 • Representación ligada de una parte de un árbol binario de búsqueda.

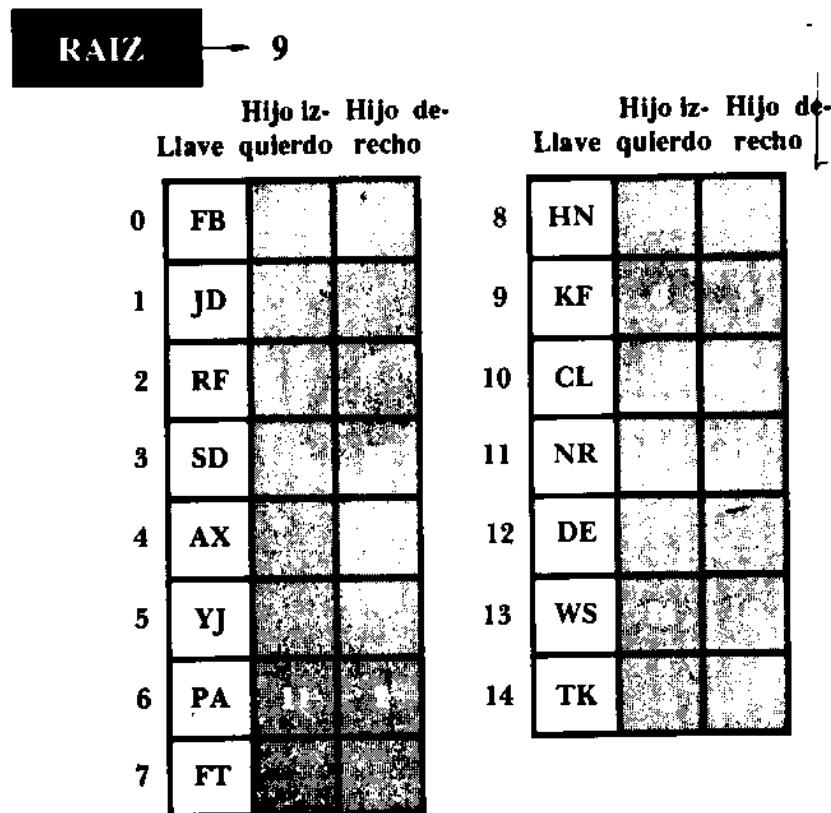


FIGURA 9.4 • Contenido de los registros en una representación ligada del árbol binario de la figura 9.2.

Obsérvese que más de la mitad de los campos de liga del archivo están vacíos porque son nodos hoja, sin hijos. En la práctica, los nodos hoja deben contener algún carácter especial, tal como -1, para indicar que la búsqueda en el árbol ha alcanzado el nivel de hoja y que ya no hay nodos en la trayectoria de búsqueda. Para hacerlos más distinguibles, se dejan los campos en blanco en esta figura, que ilustra el considerable costo potencial en términos de utilización de espacio ocasionada por este tipo de representación ligada de un árbol.

Pero fijarse en los costos y no en las ventajas es perder la nueva e importante capacidad que esta estructura de árbol proporciona: ya no se tiene que clasificar el archivo para poder efectuar una búsqueda binaria. Nótese que los registros del archivo ilustrados en la figura 9.4 aparecen de forma aleatoria y no en un orden de clasificación. La secuencia de los registros en el archivo no necesariamente tiene relación con la estructura del árbol; toda la información acerca de la estructura lógica se lleva en los campos de liga. La consecuencia positiva de esto es que si se agrega una nueva llave al archivo, tal como *LV*, sólo se necesita ligarla con el nodo hoja apropiado para crear un árbol en el que la eficiencia de la búsqueda es tan buena como la que se

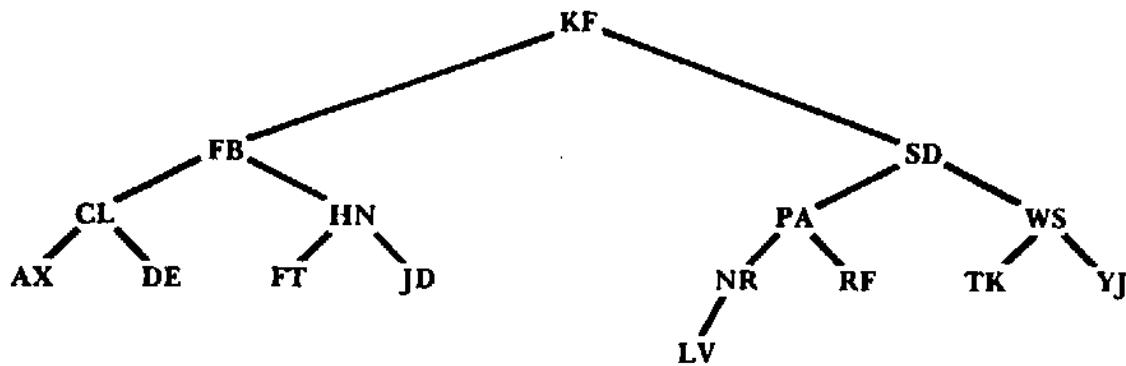


FIGURA 9.5 • Árbol binario de búsqueda con LV agregado.

tiene con una búsqueda binaria en una lista clasificada. El árbol con la LV agregada se ilustra en la figura 9.5.

El desempeño de la búsqueda en este árbol aún es bueno porque está en un estado *balanceado*. Por balanceado se entiende que la altura de la trayectoria más corta hacia una hoja no difiere de la altura de la trayectoria más larga en más de un nivel. Para el árbol de la figura 9.5, esta diferencia de uno es la más cercana posible al *balance completo*, en donde todas las trayectorias de la raíz a las hojas son exactamente de la misma longitud.

Considérese lo que sucede si se introducen las siguientes ocho llaves en el árbol, con la secuencia en que aparecen.

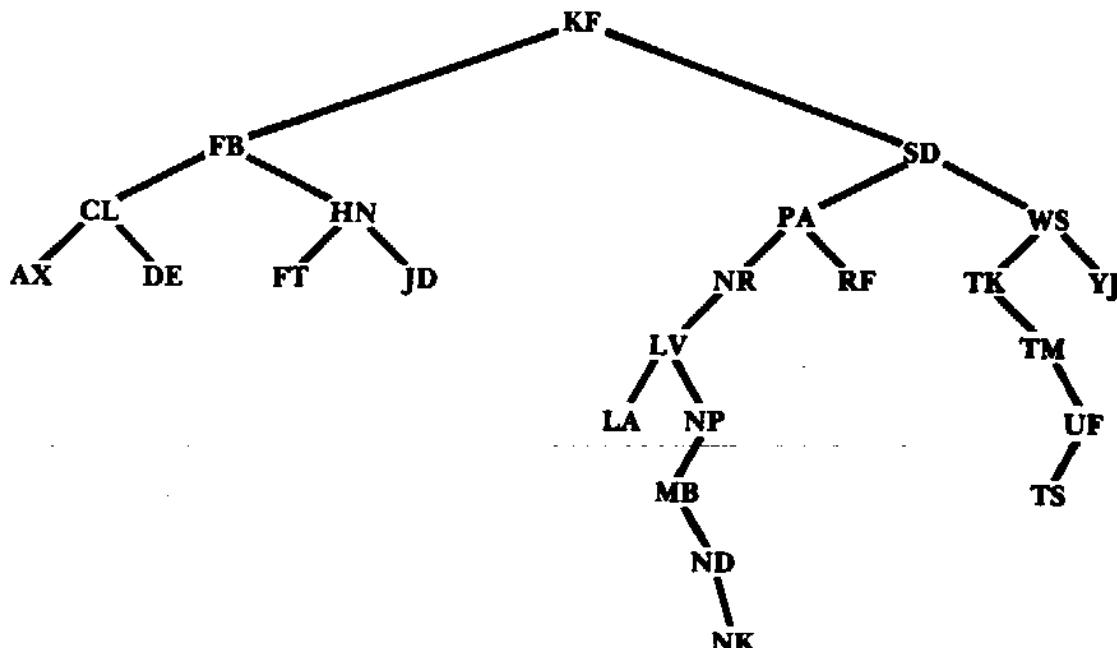


FIGURA 9.6 • Árbol binario de búsqueda que muestra el efecto de las llaves agregadas.

NP MB TM LA UF ND TS NK

Buscando a lo largo del árbol y agregando cada llave en su posición correcta en el árbol de búsqueda se obtiene el árbol que se muestra en la figura 9.6.

El árbol ahora está desbalanceado. Este resultado es normal cuando los árboles se construyen colocando las llaves conforme ocurren, sin reacomodo. La disparidad resultante entre la longitud de varias trayectorias de búsqueda es indeseable en los árboles binarios de búsqueda, pero es especialmente problemática si los nodos del árbol se están colocando en almacenamiento secundario. Ahora hay llaves que requieren siete, ocho o nueve desplazamientos para la extracción. Una búsqueda binaria en una lista clasificada de estas 24 llaves requiere sólo cinco desplazamientos en el peor de los casos. Aunque el uso de un árbol permite evitar la clasificación, se está pagando esta ventaja con desplazamientos adicionales al momento de la extracción. Para árboles con cientos de llaves, en los que una trayectoria fuera de balance puede extenderse a 30, 40 o más desplazamientos, este precio es demasiado alto.

9.4

ARBOLES AVL

Con anterioridad se dijo que no *necesariamente* existe relación entre el orden en el que se introducen las llaves y la estructura del árbol. Se subraya la palabra *necesariamente* porque es claro que el orden de entrada es importante en la determinación de la estructura del árbol del ejemplo de la figura 9.6. La razón de esta susceptibilidad al orden de entrada es que hasta ahora simplemente se han ligado los nodos más nuevos en los niveles hoja del árbol. Con este enfoque se puede obtener algunas organizaciones sumamente indeseables.

Supóngase, por ejemplo, que las llaves consisten en las letras A a G, y que se reciben estas llaves en orden alfabético. Si se ligan los nodos conforme se reciben, se produce un árbol degenerado, que de hecho es sólo una lista ligada, como se ilustra en la figura 9.7.

La solución al problema consiste, de algún modo, en reorganizar los nodos del árbol conforme se reciben llaves nuevas, manteniendo una estructura casi óptima. Un método elegante para manejar tales situaciones da por resultado una clase de árboles llamados *árboles AVL*, en honor de los matemáticos rusos G.M. Adel'son-Vel'skii y E.M. Landis, quienes los definieron por vez primera. Un árbol AVL es un árbol *balanceado en altura*. Esto significa que existe un límite en la mag-

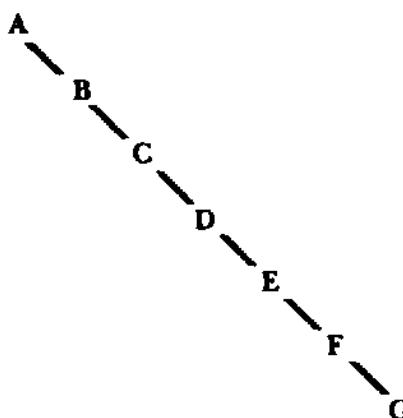


FIGURA 9.7 • Arbol degenerado.

nitud de la diferencia que se permite entre las alturas de cualesquiera dos subárboles que comparten una raíz común. En un árbol AVL la diferencia máxima permitida es 1. Por lo tanto, a un árbol AVL se le llama *árbol-1 balanceado en altura, o árbol BA(1)*, y es miembro de una clase más general de árboles balanceados en altura conocidos como árboles *BA(k)*, a los cuales se permite estar k niveles fuera de balance.

Los árboles de la figura 9.8 tienen la propiedad AVL, o BA(1). Nótese que no hay subárboles de cualquier raíz que difieran en más de un nivel.

Los árboles de la figura 9.9 no son AVL. En cada uno de ellos la raíz del subárbol que no está balanceada se marca con una X.

Las dos características en las que radica la importancia de los árboles AVL son:

- Al establecer una diferencia máxima permitida en la altura de cualesquiera dos subárboles, los árboles AVL garantizan un cierto nivel mínimo de desempeño en la búsqueda, y
- El mantener un árbol en forma AVL conforme se insertan nodos nuevos implica el uso de una de cuatro posibles rotaciones. Cada una de ellas se restringe a una única área local del árbol. Las rotaciones más complejas requieren sólo cinco reasignaciones de apuntadores.

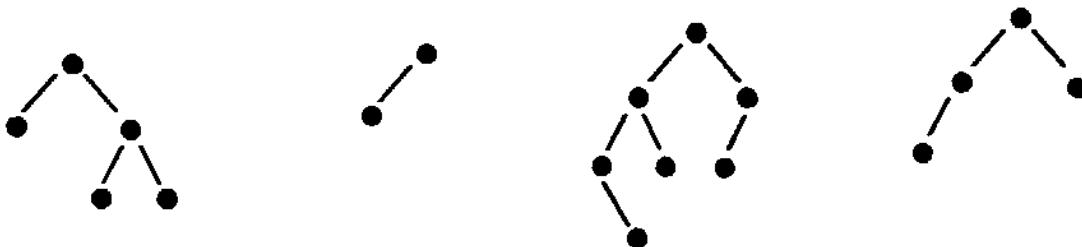


FIGURA 9.8 • Arboles AVL.

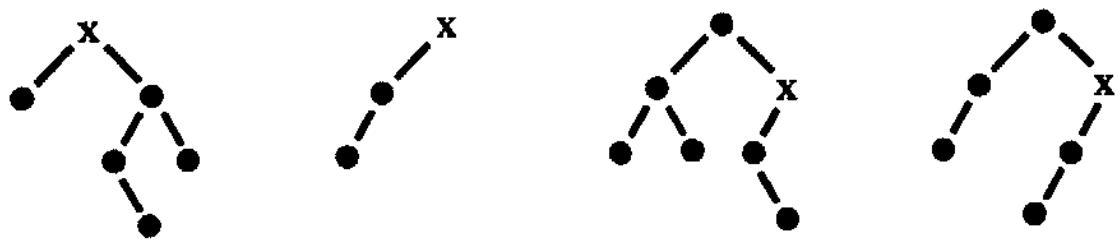


FIGURA 9.9 • Arboles que no son AVL.

Los árboles AVL son una clase importante de estructuras de datos. Las operaciones usadas para construir y mantener árboles AVL se describen en Knuth [1973b], Standish [1980] y en otros autores. Los árboles AVL no son por sí mismos aplicables directamente a la mayoría de los problemas de estructuras de archivos porque, como todos los árboles estrictamente *binarios*, tienen demasiados niveles; son demasiado *profundos*. Sin embargo, en el contexto del análisis general del problema de acceso y mantenimiento de índices que son demasiado grandes para entrar en memoria, los árboles AVL son interesantes porque sugieren la posibilidad de definir procedimientos que mantengan un balance en la altura.

El hecho de que un árbol AVL esté balanceado en altura garantiza que el desempeño de la búsqueda se aproxima al de un árbol *completamente balanceado*. Por ejemplo, la forma completamente balanceada de un árbol construido a partir de las llaves de entrada

B C G E F D A

se ilustra en la figura 9.10, y el árbol AVL resultante de las mismas llaves de entrada, que llegan en la misma secuencia, se ilustra en la figura 9.11.

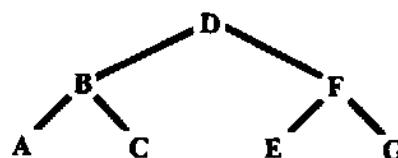


FIGURA 9.10 • Árbol de búsqueda completamente balanceado.

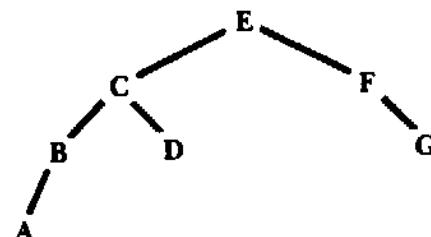


FIGURA 9.11 • Árbol de búsqueda construido empleando procedimientos AVL.

Para un árbol completamente balanceado, el peor caso de la búsqueda para encontrar una llave, considerando N llaves posibles, busca en

$$\log_2(N + 1)$$

niveles del árbol. Para un árbol AVL, el peor caso de la búsqueda podría ser buscar en

$$1.44 \log_2(N + 2)$$

niveles. Así que, para 1 000 000 de llaves, un árbol completamente balanceado requiere desplazamiento en 20 niveles para algunas de las llaves, pero nunca en 21 niveles. Si el árbol es AVL, el número máximo de niveles se incrementa a sólo 28. Este es un resultado interesante, dado que los procedimientos AVL garantizan que una sola reorganización requiere no más de cinco reasignaciones de apuntadores. Estudios empíricos realizados por Vandoren y Gray [1974], entre otros, han demostrado que tales reorganizaciones locales se requieren para aproximadamente una de cada dos inserciones en el árbol y para cada cuarta eliminación. Así, el balance en altura usando los métodos AVL garantiza que se obtendrá una aproximación razonable a un desempeño óptimo del árbol binario, a un costo que es aceptable en la mayoría de las aplicaciones que usan la memoria primaria de acceso aleatorio.

Cuando se usa el almacenamiento secundario, un procedimiento que requiere más de cinco o seis desplazamientos para encontrar una llave es menos que deseable; uno que necesita 20 o 28 es inaceptable. Volviendo a los dos problemas que se identificaron anteriormente en este capítulo:

- La búsqueda binaria requiere demasiados desplazamientos, y
- Mantener un índice en orden es costoso,

se puede observar que los árboles balanceados en altura proporcionan una solución aceptable al segundo problema. Ahora es necesario considerar el primero.

9.5

ARBOLES BINARIOS PAGINADOS

Una vez más se afronta lo que es quizá la característica más crítica de los dispositivos de almacenamiento secundario: desplazarse a una localidad específica toma un tiempo relativamente largo, pero una vez que la cabeza lectora está en posición y lista, leer o escribir un conjunto

de bytes contiguos se efectúa rápidamente. Esta combinación de desplazamiento lento y transferencia rápida lleva naturalmente a la idea de paginación. En un sistema con páginas no se incurre en el costo de un desplazamiento en disco sólo para obtener unos pocos bytes, sino que, una vez que se ha tomado el tiempo para desplazarse a un área del disco, se lee una página completa del archivo. Esta página puede consistir en muchos registros individuales. Si la siguiente porción de información que se requiere del disco está en la página que se leyó, se ha ahorrado el costo de un acceso a disco.

La paginación, entonces, es una solución potencia al problema de la búsqueda. Al dividir un árbol binario en páginas y después almacenar cada página en un bloque de localidad contiguas en disco, debe ser posible reducir el número de desplazamientos asociados con cualquier búsqueda. La figura 9.12 ilustra dicho árbol paginado. En este árbol es posible localizar cualquiera de los 63 nodos del árbol con no más de dos accesos a disco. Nótese que cada página almacena siete nodos y puede ramificarse en ocho páginas nuevas. Si se extiende el árbol a un nivel adicional de paginación se agregarán 64 páginas nuevas; se puede encontrar cualquiera de 511 nodos con sólo tres desplazamientos. Agregar un nivel más de paginación permite encontrar cualquiera de 4095 nodos con sólo cuatro desplazamientos. Una búsqueda binaria de una lista de 4095 ítems puede tomar hasta 12 desplazamientos.

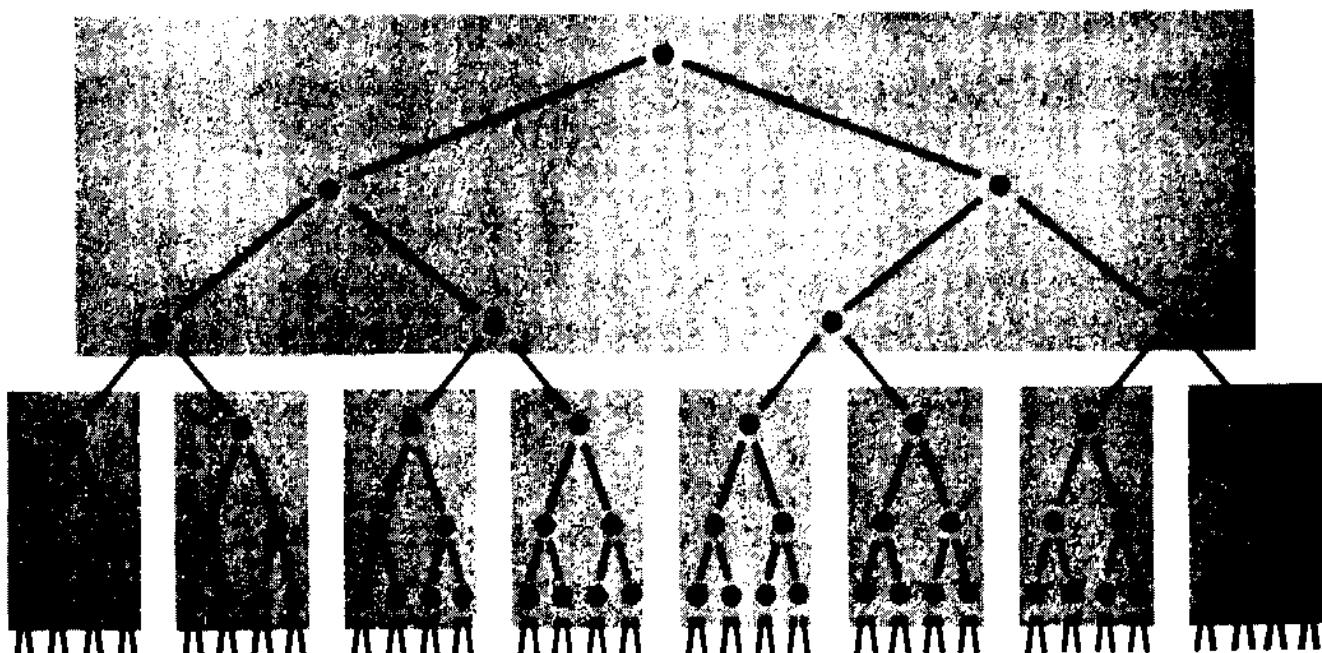


FIGURA 9.12 • Árbol binario paginado.

En concreto, dividir el árbol en páginas permite búsquedas más rápidas en almacenamientos secundario, y proporciona una extracción mucho más rápida que cualquier otra de las formas de acceso por llave que se han considerado hasta este punto. Además, el uso de un tamaño de página de siete en la figura 9.12 lo imponen más las restricciones de la página impresa que algún otro aspecto que tenga que ver con los dispositivos de almacenamiento secundario. Un ejemplo típico de tamaño de página puede ser de 8 kilobytes, capaz de almacenar 511 parejas de campos de llaves y referencias. Con este tamaño de página, y suponiendo que cada página contiene un árbol lleno, completamente balanceado, y que las páginas por sí mismas están organizadas como un árbol lleno completamente balanceado, entonces es posible encontrar cualquiera de 134 217 727 llaves con sólo tres desplazamientos. Esa es la clase de desempeño que se está buscando. Nótese que, mientras que el número de desplazamiento requeridos para el peor caso de la búsqueda en un árbol balanceado completamente lleno es

$$\log_2(N + 1)$$

en donde N es el número de llaves del árbol, el número de desplazamientos requeridos para las versiones *paginadas* de un árbol balanceado completamente lleno es

$$\log_{k+1}(N + 1)$$

donde N es, una vez más, el número de llaves. La nueva variable, k , es el número de llaves almacenadas en una sola página. La segunda fórmula es en realidad una generalización de la primera, puesto que el número de llaves en una página de un árbol puramente binario es 1. Es el efecto logarítmico del tamaño de la página lo que vuelve drástico el impacto de la paginación:

$$\log_2(134\ 217\ 727 + 1) = 27 \text{ desplazamientos}$$

$$\log_{511+1}(134\ 217\ 727 + 1) = 3 \text{ desplazamientos}$$

El uso de páginas grandes no es gratuito. Todo acceso a una página requiere la transmisión de una gran cantidad de datos, la mayoría de los cuales no se usan. Sin embargo, este tiempo de transmisión adicional justifica su costo debido a que ahorra muchos desplazamientos, los cuales consumen más tiempo que las transmisiones adicionales. Un problema mucho más serio, que se examina a continuación, se refiere a la manera de mantener la organización del árbol paginado.

9.6

**EL PROBLEMA CON LA
CONSTRUCCION DESCENDENTE
DE LOS ARBOLES PAGINADOS**

Dividir un árbol en páginas es una estrategia bastante adecuada a las características de los dispositivos de almacenamiento secundario, tales como los discos. El problema, una vez que se decide realizar un árbol paginado, es cómo construirlo. Si se tiene todo el conjunto de llaves a la mano antes de construir el árbol, la solución es relativamente sencilla: puede clasificarse la lista de llaves y construir el árbol a partir de esta lista. Lo más importante es que, si se planea iniciar la construcción a partir de la raíz, se sabe que la llave situada a la mitad de la lista clasificada debe ser la *llave raíz* dentro de la *página raíz* del árbol. Abreviando, se sabe dónde comenzar, y se asegura que este punto inicial divide el conjunto de llaves en forma balanceada.

Por desgracia, el problema es mucho más complejo cuando se reciben llaves en orden aleatorio y se insertan tan pronto como se reciben. Supóngase que debe construirse un árbol en páginas conforme se recibe la siguiente secuencia de llaves de una sola letra:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Se construirá un árbol binario paginado con un máximo de tres llaves por página. Conforme se insertan las llaves, se rotan dentro de una página cuando sea necesario para mantener cada página tan balanceada como sea posible. El árbol resultante se ilustra en la figura 9.13. Evaluado en términos de la profundidad del árbol (medido en páginas), este árbol no está tan mal. (Considérese, por ejemplo, qué sucedería si las llaves llegaran en orden alfabético.)

Aunque este árbol no está demasiado deformé, ilustra claramente los problemas inherentes a la construcción de un árbol binario paginado en forma descendente. Cuando se comienza a partir de la raíz, las llaves iniciales deben, por necesidad, ir en la raíz. En este ejemplo al menos dos de estas llaves, C y D, no son las que se quisiera tener ahí. Son adyacentes en la secuencia y tienden hacia el principio del conjunto total de llaves. En consecuencia, hacen que el árbol salga de balance.

Una vez que las llaves equivocadas se colocan en la raíz del árbol (o en la raíz de cualquier subárbol que esté más abajo en el árbol), ¿qué es lo que se puede hacer? Por desgracia, la respuesta no es fácil. No se puede simplemente rotar páginas completas del árbol en la misma forma en que se rotarían las llaves individuales en un árbol que no esté paginado. Si se rota el árbol de tal forma que la página raíz inicial se

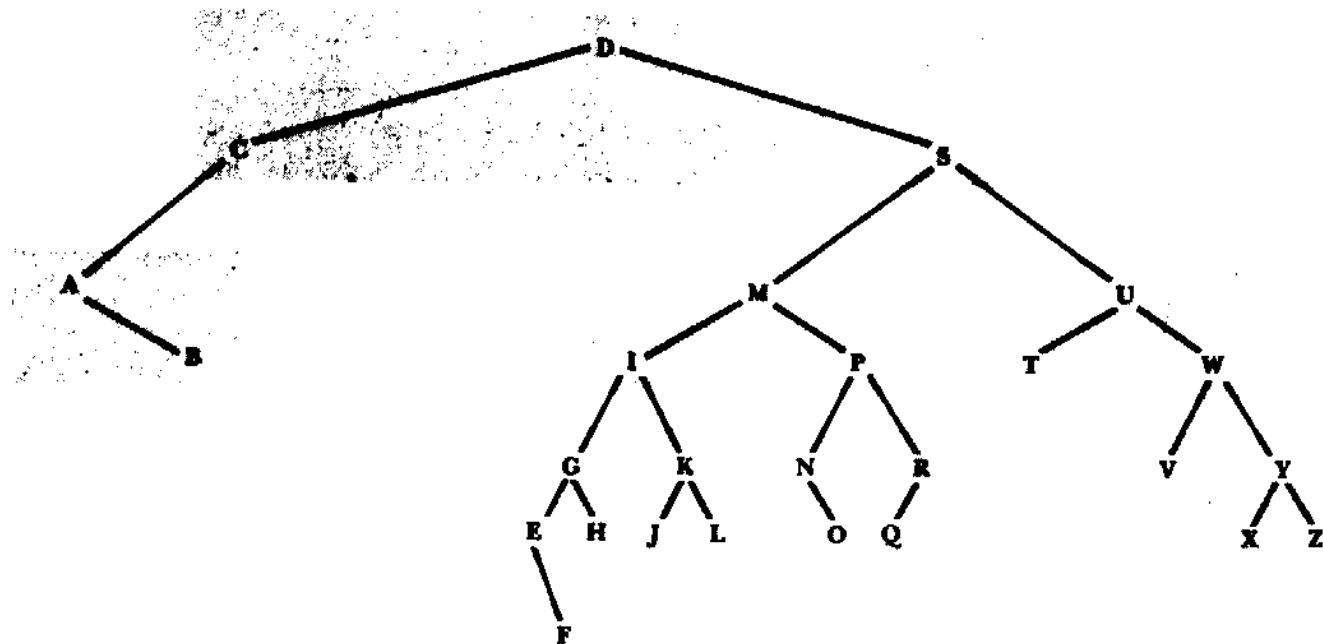


FIGURA 9.13 Árbol paginado construido a partir de llaves que llegan en secuencia aleatoria.

mueva hacia abajo a la izquierda, trasladando las llaves *C* y *D* a una mejor posición, entonces la llave *S* queda fuera de lugar. Por lo tanto, deben dividirse las páginas.

Esto abre todo un mundo de posibilidades... y problemas. Dividir las páginas implica reacomodarlas para crear páginas nuevas que queden balanceadas internamente y además bien acomodadas en relación con las otras. Intente crear un algoritmo de reacomodo de páginas para el sencillo árbol paginado a tres llaves de la figura 9.13. Resultará muy difícil crear un algoritmo que tenga sólo efectos locales si se reacomodan sólo unas cuantas páginas. La tendencia es hacia reacomodos y ajustes que se extienden a lo largo de una gran parte del árbol. La situación se vuelve aún más compleja con mayores tamaños de página.

Así, aunque se ha determinado que la idea de agrupar llaves en páginas es muy buena considerando la reducción de desplazamientos en disco, no se ha encontrado una forma de agrupar las llaves correctas. Todavía quedan al menos dos preguntas sin respuesta:

- ¿Cómo puede asegurarse que las llaves de la página raíz sean buenas llaves separadoras, que dividan en forma más o menos pareja el conjunto de las otras llaves?

- ¿Cómo evitar que se agrupen llaves, como *C, D y S* en el ejemplo, que no deben compartir una página?

Además, existe una tercera pregunta que aún no se había planteado debido a lo pequeño de la página del árbol del ejemplo:

- ¿Cómo puede garantizarse que cada una de las páginas contenga al menos algún número mínimo de llaves? Cuando se trabaja con un tamaño de página más grande, como 8191 llaves por página, se desea evitar situaciones en las que cada una de un gran número de páginas contenga sólo unas cuantas docenas de llaves.

El artículo de 1972 de Bayer y McCreight sobre árboles B proporciona una solución precisamente a estos problemas.

9.7

ARBOLES B: CONSTRUCCION ASCENDENTE

Muchas ideas elegantes y poderosas que se han aplicado en las ciencias de la computación surgieron al cambiar de enfoque para examinar un problema. Los árboles B son un ejemplo de este fenómeno de cambio de punto de vista.

La idea clave requerida para pasar de los tipos de árboles que se han considerado a una nueva solución, los árboles B, es que puede elegirse construir los árboles hacia arriba a partir de la base, en lugar de hacerlo hacia abajo desde la cima. Hasta ahora se ha dado por un hecho la necesidad de iniciar la construcción a partir de la raíz. Luego, al descubrir que se tienen llaves equivocadas en la raíz, se ha intentado encontrar formas para corregir el error con algoritmos de reacomodo. Bayer y McCreight reconocieron que la decisión de trabajar hacia abajo a partir de la raíz era el problema. En lugar de encontrar formas para resolver una mala situación, decidieron evitar del todo la dificultad. Con los árboles B se permite que la raíz *emerja*, en vez de colocarla y después encontrar la manera de cambiarla.

9.8

DIVISION Y PROMOCION

En un árbol B, una página o *nodo* consiste en una secuencia ordenada de llaves y un conjunto de apuntadores. No existe un árbol explícito para un nodo, como sucede con los árboles paginados que se examinaron



FIGURA 9.14• Hoja inicial de un árbol B con tamaño de página siete.

previamente; sólo una lista ordenada de llaves y algunos apuntadores. El número de apuntadores siempre excede en uno al número de llaves. Al número máximo de apuntadores que pueden almacenarse en un nodo se le llama el *orden* del árbol B. Por ejemplo supóngase que se tiene un árbol B de orden ocho. Cada página puede almacenar a lo más siete llaves y ocho apuntadores. La *hoja* inicial del árbol puede tener una estructura similar a la que se ilustra en la figura 9.14, después de la inserción de las letras

B C G E F D A

Los campos marcados con asteriscos (*) son los campos apuntadores. En esta hoja, como en cualquier otro nodo hoja, el valor de todos los apuntadores indica el final de la lista. Por definición, un nodo hoja no tiene hijos en el árbol; en consecuencia, los apuntadores no remiten a otras páginas del árbol. Se supone aquí que los apuntadores en las páginas hoja por lo regular contienen un valor de apuntador inválido, como -1. Nótese que, incidentalmente, esta *hoja* también es la *raíz*.

En una aplicación práctica también existe, por lo regular, alguna otra información almacenada con la llave, como una referencia a un registro que contiene datos asociados con la llave. En consecuencia, los campos apuntadores adicionales en cada página en realidad pueden remitir a algunos registros de datos asociados que están almacenados en otro lugar. Pero, parafraseando a Bayer y McCreight, para los propósitos presentes, "la información asociada no tiene mayor interés".

Construir la primera página es bastante fácil. Cuando se insertan nuevas llaves, se usa sólo un acceso al disco para transferir la página a la memoria y, como se trabaja en memoria, para insertar la llave en su lugar en la página. Puesto que se trabaja en memoria electrónica, la inserción resulta relativamente barata, comparada con el costo de los accesos adicionales al disco.

Pero ¿qué sucede cuando llegan llaves nuevas? Supóngase que se pretende agregar la llave J al árbol B. Al tratar de insertar la J se encuentra que la hoja está llena. Entonces se *divide* en dos hojas, distribuyendo las llaves entre el nodo anterior y el nuevo tan equitativamente como sea posible, como se muestra en la figura 9.15.



FIGURA 9.15 • División de la hoja para acomodar la nueva llave *J*.

Puesto que ahora se tienen dos hojas, es necesario crear un nivel superior en el árbol para que, cuando se esté buscando, se pueda elegir entre las dos hojas. En pocas palabras, es necesario crear una raíz nueva. Para ello, se *promueve* una llave que *separe* las hojas. En este caso, se promueve la *E* de la primera posición en la segunda hoja, como se ilustra en la figura 9.16.

En este ejemplo se describen las operaciones de división y promoción en dos pasos para hacer el procedimiento tan claro como sea posible; en la práctica, la división y la promoción se realizan en una sola operación.

Ahora se estudiará cómo crece un árbol B, con la secuencia de llaves que produce el árbol binario paginado que se ilustra en la figura 9.13. La secuencia es

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Se usa un árbol B de orden cuatro (cuatro campos apuntadores y tres campos llave por página), porque corresponde al tamaño de página binario paginado. Usar ese pequeño tamaño de página tiene como ventaja adicional que las páginas se dividen con mayor frecuencia, proporcionando más ejemplos de división y promoción. Se omite una indicación explícita de los campos apuntadores, para que pueda entrar un árbol más grande en la página.

La figura 9.17 ilustra el crecimiento del árbol hasta el momento en que el nodo raíz está a punto de dividirse. La figura 9.18 muestra el

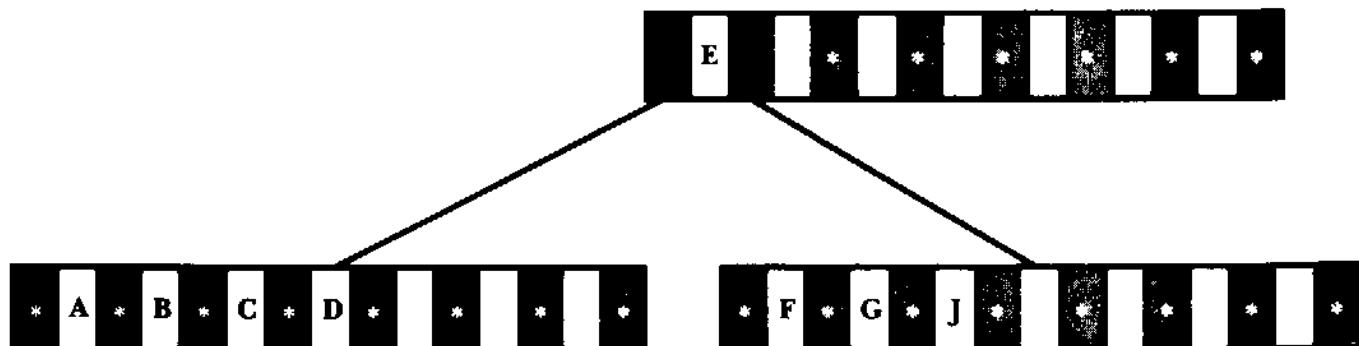
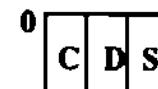
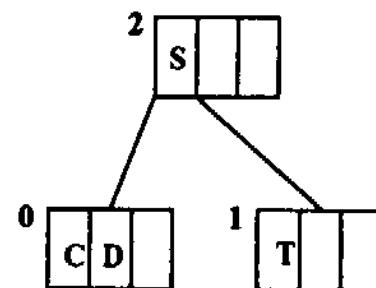


FIGURA 9.16 • Promoción de la llave *E* a un nodo raíz.

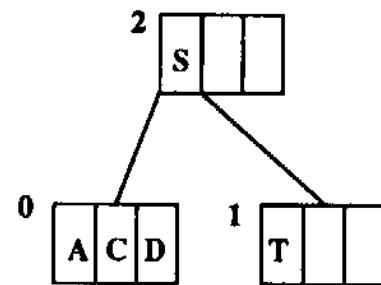
Inserción de C, S y D en la página inicial:



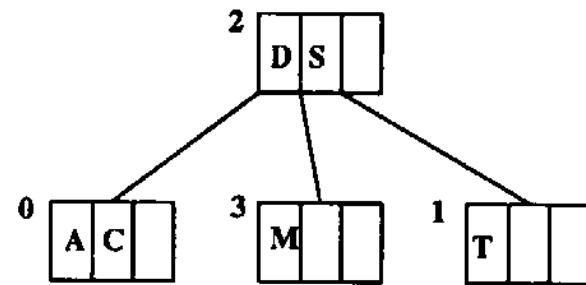
La inserción de T fuerza la división y la promoción de S:



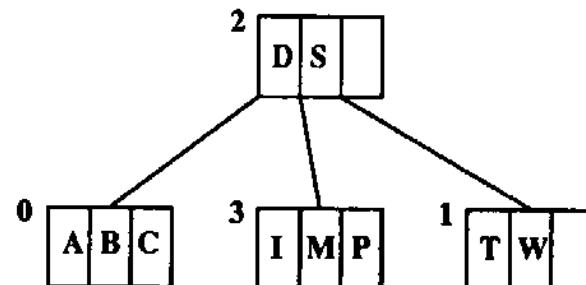
A se agrega sin incidentes:



La inserción de M fuerza otra división y la promoción de D:



P, I, B y W se insertan en las páginas existentes:



La inserción de N provoca otra división, seguida por la promoción de N. G, U y R se agregan a las páginas existentes:

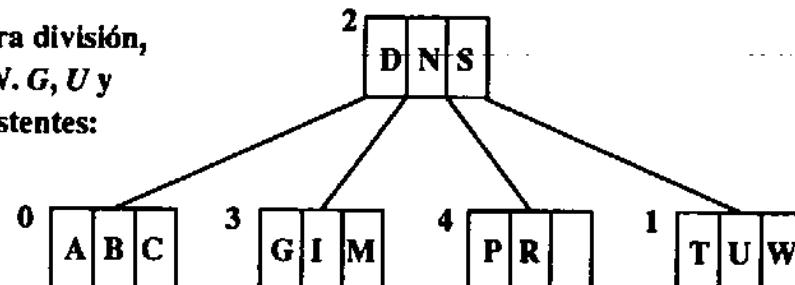
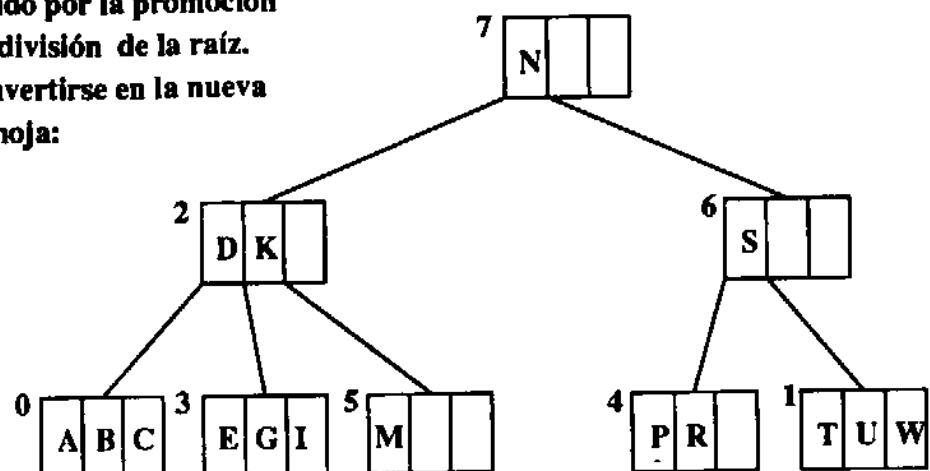
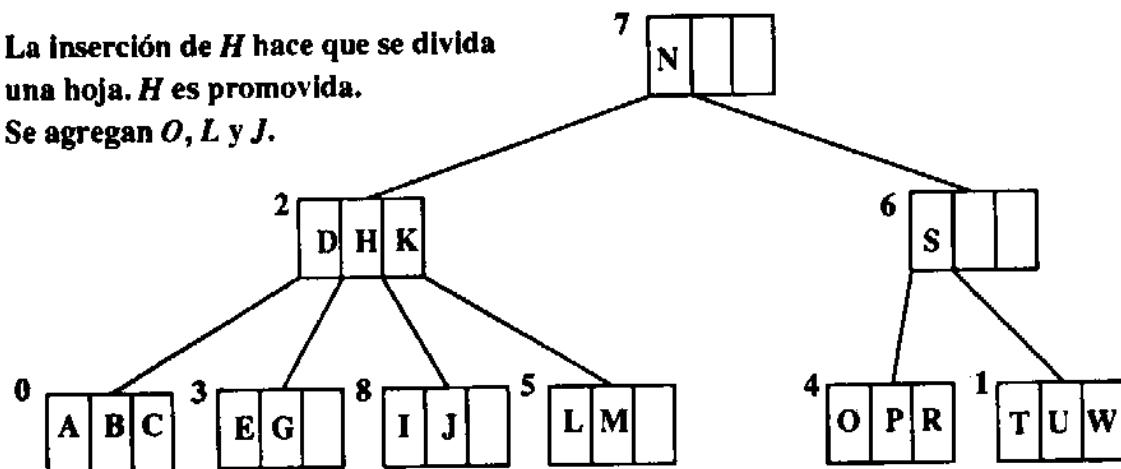


FIGURA 9.17 • Crecimiento de un árbol B, parte I. El árbol crece hasta un punto en el que la división de la raíz es inminente.

La inserción de *K* ocasiona una división en el nivel de hoja, seguido por la promoción de *K*. Esto provoca una división de la raíz. *N* se promueve para convertirse en la nueva raíz. *E* se agrega a una hoja:



La inserción de *H* hace que se divida una hoja. *H* es promovida.
Se agregan *O*, *L* y *J*.



La inserción de *Y* y *Q* fuerza dos divisiones más de hojas de dos promociones. Se agregan las letras restantes.

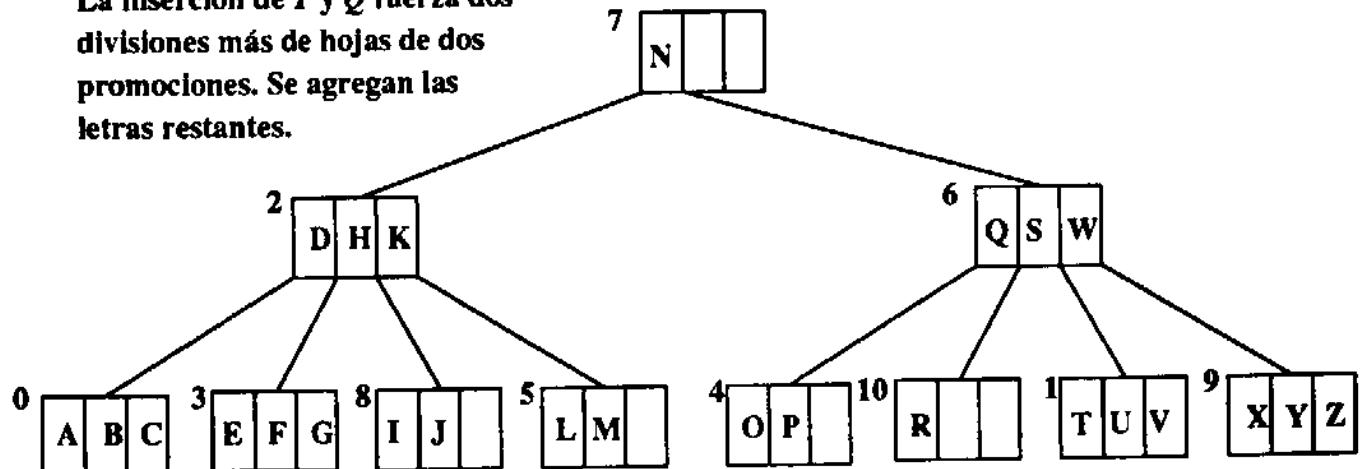


FIGURA 9.18 • Crecimiento de un árbol B parte II. La raíz se divide para agregar un nuevo nivel; se insertan las llaves restantes.

árbol después de la división del nodo raíz. La figura también muestra cómo continúa creciendo el árbol conforme se agregan las llaves restantes de la secuencia. Se numera cada una de las páginas del árbol (esquina superior izquierda de cada nodo), de tal forma que puedan distinguirse las páginas nuevas que se agregaron de las existentes en el árbol.

Nótese que el árbol siempre está perfectamente balanceado con respecto a la altura; la trayectoria que va de la raíz a una hoja es la misma para todas las hojas. También se observa que las llaves que se promueven hacia arriba dentro del árbol son necesariamente el tipo de llaves que se desean en la raíz; llaves que son buenos separadores. Al trabajar por encima del nivel de hoja, dividiendo y promoviendo conforme se llenan las páginas, se evitan los problemas que plagaron los anteriores esfuerzos con árboles binarios paginados.

9.9

ALGORITMOS PARA LA BUSQUEDA E INSERCIÓN EN ARBOLES B

Ahora que se tiene ya una idea de cómo trabajan los árboles B en el papel, se plantean las estructuras y algoritmos necesarios para que trabajen en un computador. La mayor parte del código siguiente es pseudocódigo. Las versiones en C y Pascal de los algoritmos pueden encontrarse al final del capítulo.

ESTRUCTURA DE LA PAGINA. Se comienza con la definición de una forma posible para la página empleada en un árbol B. Como se verá más adelante en este capítulo y en el siguiente, existen muchas formas diferentes de construir las páginas de un árbol B. Se comienza con una sencilla, en la que cada llave es un solo carácter. Si el número máximo de llaves e hijos permitidos por página es MAXLLAVES y MAXHIJOS, respectivamente, entonces las siguientes estructuras expresadas en C y Pascal describen una página llamada PAGINA.

En C:

```
struct PAGINAAB {
    short    CONT_LLAVES;
            /* número de llaves almacenadas en PAGINA */;
    char     LLAVE [MAXLLAVES];
            /* las llaves reales */
    short    HIJO [MAXLLAVES+1];
            /* NRRs de los hijos */
} PAGINA;
```

En Pascal:

```

TYPE
PAGINAAB = RECORD
    CONT_LLAVES : integer ;
    LLAVE       : array [1.. MAXLLAVES] of char;
    HIJO        : array [1.. MAXHIJOS] of integer;
    END;
VAR
PAGINA : PAGINAAB;

```

Tomando en cuenta esta estructura de página, el archivo que contiene el árbol B consiste en un conjunto de registros de longitud fija. Cada registro contiene una página del árbol. Puesto que las llaves del árbol son simples letras, esta estructura usa un arreglo de *caracteres* para almacenar las llaves. De manera típica, el arreglo de llaves es un vector de cadenas en lugar de sólo un vector de caracteres. La variable PAGINA CONT_LLAVES es útil cuando los algoritmos deben deter-

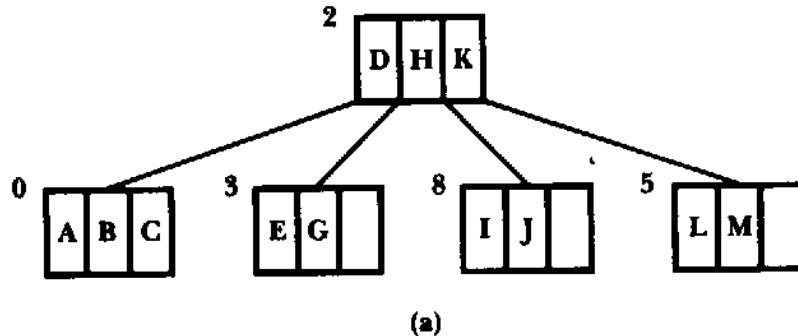
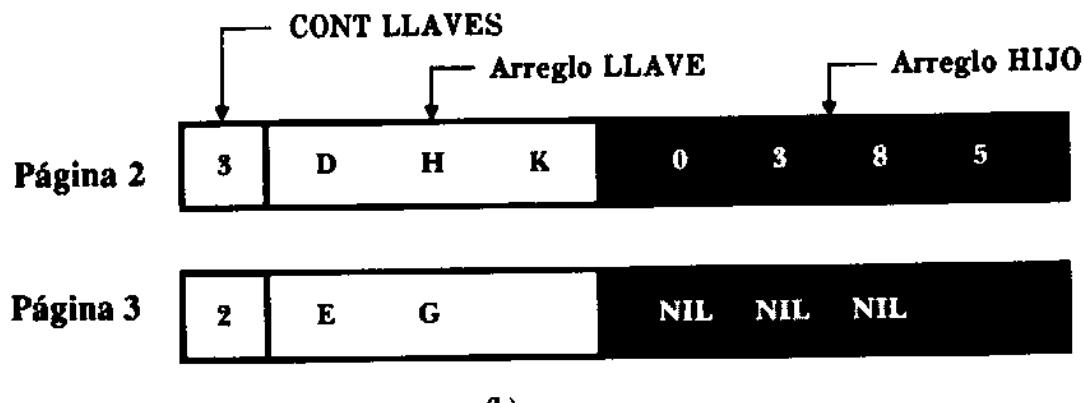
Parte de un árbol B:**Contenido de PAGINA para las páginas 2 y 3:**

FIGURA 9.19 • Árbol B de orden cuatro. (a) Un nodo interno y algunos nodos hoja. (b) Los nodos 2 y 3, como se verían en la estructura PAGINA.

minar si una página está llena o no. El arreglo PAGINA.HIJO[] contiene los NRR de los hijos de PAGINA, si es que los tiene. Cuando no hay descendencia, al elemento correspondiente de PAGINA.HIJO[] se asigna un valor al cual no es posible asignarle dirección, que se llama NULO. La figura 9.19 muestra dos páginas en un árbol B de orden cuatro.

BUSQUEDA. El primero de los algoritmos de árboles B que se examina es un procedimiento de *búsqueda* en el árbol. La búsqueda es un buen punto para comenzar porque, aunque es relativamente simple, ilustra los aspectos característicos de la mayoría de los algoritmos para árboles B:

- Son recursos*, y
- Funcionan en dos etapas, operando en forma alternada sobre páginas enteras y después *dentro* de las páginas.

El procedimiento de búsqueda se llama a sí mismo recursivamente; localiza una página y después busca en ella para encontrar la llave en niveles sucesivamente más bajos del árbol hasta que la encuentra o hasta que no puede descender más, habiendo rebasado el nivel de hoja. La figura 9.20 contiene una descripción del procedimiento de búsqueda en pseudocódigo.

Se va a trabajar a mano a lo largo de la función, buscando la llave *K* en el árbol que ilustra la figura 9.21. Se comienza llamando a la función con el argumento NRR de la raíz (2). Este NRR no es NULO, así que la función lee la raíz en PAGINA y después busca *K* entre los elementos de PAGINA.LLAVE[]. No se encuentra la *K*. Puesto que *K* debe ir entre *D* y *N*, POS identifica la posición 1[†] en la raíz como la posición del apuntador donde la búsqueda debe continuar. *Busca()* se llama a sí misma, esta vez usando el NRR almacenado en PAGINA.HIJO [1]. El valor de este NRR es 3.

En la siguiente llamada, *busca()* lee la página que contiene las llaves *G*, *I* y *M*. Una vez más, la función busca *K* entre las llaves de PAGINA.LLAVE[]. De nuevo, *K* no se encuentra. Esta vez PAGINA.HIJO [2] indica dónde debe continuar la búsqueda. *Busca()* se llama a sí misma de nuevo, esta vez usando el NRR almacenado en PAGINA.HIJO [2].

Puesto que esta llamada es desde un nivel hoja, PAGINA.HIJO [2] es NULO, por lo que la llamada a *busca()* falla inmediatamente. El valor

[†]Se usará indexación con origen cero en estos ejemplos, de tal forma que la llave que está más a la izquierda en una página es PAGINA.LLAVE[0], y el NRR del hijo que está más a la izquierda es PAGINA.HIJO[0].

```

FUNCION: busca (NRR, LLAVE, NRR_ENCONTRADO, POS_ENCONTRADA)

    si NRR == NULO entonces /*Condición de detención de la recursión */
        devuelve NO ENCONTRADA
    otro
        lee la página NRR en PAGINA
        busca LLAVE en PAGINA, haciendo POS igual a la posición donde
            LLAVE esté o debería estar.
        si se encontró la LLAVE entonces
            NRR_ENCONTRADO := NRR /*El NRR actual contiene la llave */
            POS_ENCONTRADA := POS
            devuelve ENCONTRADA
        otro /* Se sigue la referencia HIJO al siguiente nivel inferior */
            devuelve (busca (PAGINA, HIJO[POS], LLAVE, NRR_ENCONTRADO,
                POS_ENCONTRADA))
        fin si
    fin si

fin FUNCION

```

FIGURA 9.20 • La función *busca (NRR, LLAVE, NRR_ENCONTRADO, POS_ENCONTRADA)* busca en forma recursiva a lo largo del árbol para encontrar LLAVE. Cada llamada busca la página referenciada por NRR. Los argumentos NRR_ENCONTRADO y POS_ENCONTRADA identifican la página y posición de la llave, si se encuentra. Si *busca()* encuentra la llave, devuelve ENCONTRADA. Si rebasa el nivel de hoja sin encontrar la llave, devuelve NO ENCONTRADA.

NO ENCONTRADA se devuelve a través de los varios niveles de proposiciones *return()*, hasta que el programa que originalmente llama a *busca()* recibe la información de que la llave no se encontró.

Ahora se *busca()* para buscar *M*, que sí está en el árbol. *Busca()* sigue la misma trayectoria descendente que recorrió para *K*, pero esta vez se encuentra la *M* en la posición 2 de la página 3. Almacena los valores 3 y 2 en NRR_ENCONTRADO, y POS_ENCONTRADA, respectivamente, indicando que *M* está en la posición 2 de la página 3, y devuelve el valor ENCONTRADA.

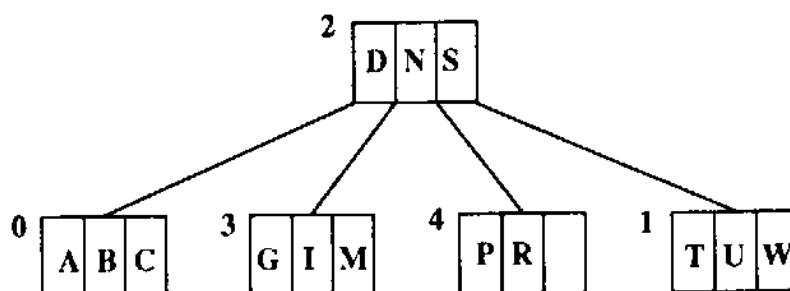


FIGURA 9.21 • Arbol B usado para el ejemplo de búsqueda.

INSERCIÓN, DIVISIÓN, Y PROMOCIÓN. Hay dos observaciones importantes que pueden hacerse acerca del proceso de inserción, división y promoción:

- Comienza con una búsqueda que llega abajo hasta el nivel de hoja, y
- Despues de encontrar el lugar de inserción en el nivel hoja, el trabajo de inserción, división y promoción continúa en forma ascendente desde abajo.

En consecuencia, se puede pensar en el procedimiento recursivo con tres fases:

1. Un paso de búsqueda en páginas que, como en la función *busca()*, ocurre antes de la llamada recursiva;
2. La llamada recursiva misma, que mueve la operación hacia abajo en el árbol conforme busca ya sea la llave o el lugar para insertarla, y
3. La lógica de inserción, división y promoción que se ejecuta después de la llamada recursiva; la acción tiene lugar en la trayectoria de vuelta luego del descenso recursivo.

Se necesita un ejemplo de una inserción para que pueda verse el trabajo del procedimiento de inserción a lo largo de estas fases. Se va a insertar el carácter \$ dentro del árbol que se muestra en la mitad superior de la figura 9.22, que contiene todas las letras del alfabeto. Puesto que la secuencia de caracteres ASCII coloca el carácter \$ antes del carácter A, la inserción se realiza dentro de la página con el NRR 0. Esta página y su padre ya están completas, así que la inserción provoca división y promoción, que dan como resultado el árbol que se muestra en la mitad inferior de la figura 9.22.

Ahora se estudiará cómo la función *inserta()* realiza esta división y promoción. Puesto que la función opera en forma recursiva, es importante comprender cómo se usan en llamadas sucesivas los argumentos de la función. La función *inserta()* que se describe emplea cuatro argumentos:

NRR_ACTUAL	El NRR de la página del árbol B que se usa actualmente. Como la función desciende y asciende recursivamente en el árbol, se usan todos los NRR en la trayectoria de búsqueda e inserción.
LLAVE	La llave que se va a insertar.
LLAVE_PROMO	Argumento que se usa sólo para regresar el valor devuelto. Si de la inserción resultan una división y la promoción de una llave, LLAVE_PROMO contiene la llave promovida en el regreso ascendente del árbol.

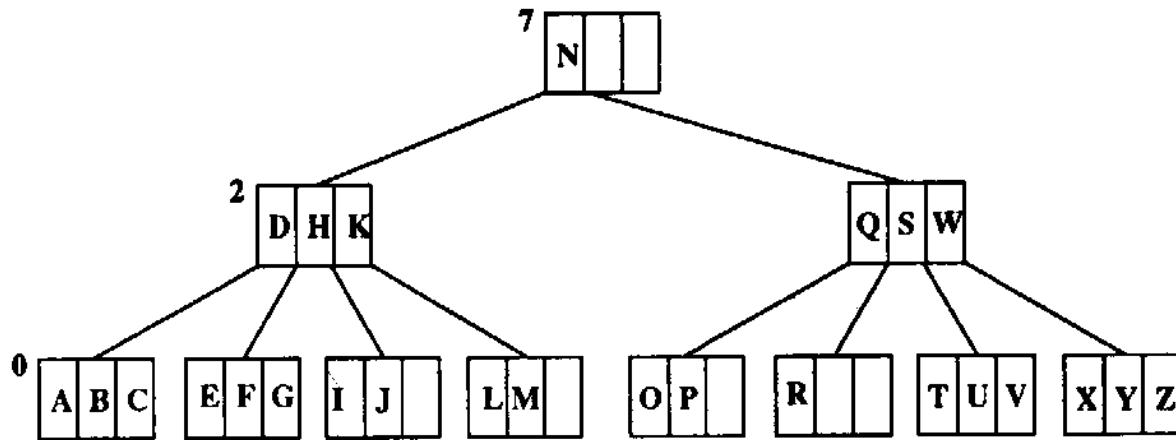
HIJO_D_PROMO

Este es otro valor de argumento devuelto. Si hay una división, los niveles superiores de la secuencia de llamadas no sólo deben insertar la llave promovida, sino también el NRR de la nueva página creada en la división. Cuando LLAVE_PROMO se inserta, HIJO_D_PROMO es el apuntador al hijo derecho que se inserta con ella.

Además de los valores devueltos por medio de los argumentos LLAVE_PROMO e HIJO_D_PROMO, *inserta()* devuelve el valor PROMOCION, si se hace una promoción NO PROMOCION, si se hace una inserción y no se promueve nada, y ERROR si no puede hacerse la inserción.

La figura 9.23 ilustra la forma en que cambian los valores de estos argumentos cuando se llama a la función *inserta()* y ésta se llama a sí misma para efectuar la inserción del carácter \$. La figura señala varios puntos importantes:

Antes de insertar \$:



Después de insertar \$:

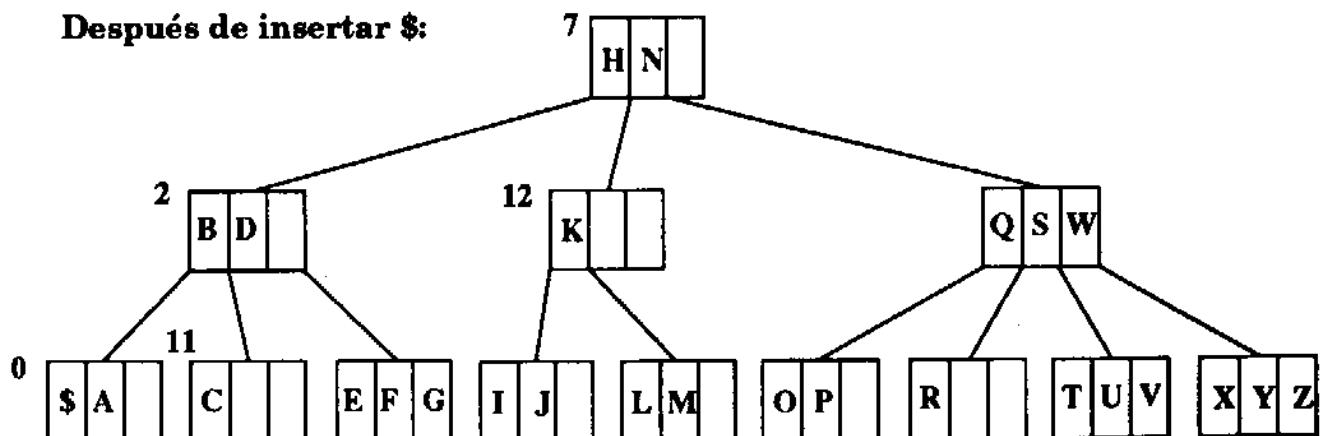


FIGURA 9.22 • Efecto de agregar \$ al árbol construido en la figura 9.18.

- Durante el paso de búsqueda de la inserción, mientras la función se llama a sí misma sólo NRR_ACTUAL cambia, haciendo el descenso en el árbol. Esta trayectoria de búsqueda de llamadas sucesivas incluye todas las páginas del árbol que puedan verse afectadas por la división y promoción en la trayectoria de retorno.
- El paso de búsqueda termina cuando NRR_ACTUAL es NULO; ya no existen niveles adicionales en donde se pueda buscar.
- Conforme regresa cada llamada recursiva, se ejecuta la lógica de inserción y división en ese nivel. Si la función de nivel inferior devuelve el valor PROMOCION, entonces se tiene una llave para insertar en este nivel. En caso contrario, no hay nada que hacer y sólo regresa. Por ejemplo, se puede insertar *H* en el nivel más alto (raíz) del árbol sin división, y por lo tanto se devuelve NO PROMOCION desde este nivel. Esto significa que la LLAVE_PPROMO y el HIJO_D_PPROMO de este nivel no significan nada.

Con esta introducción a la operación de la función *inserta()*, el lector está preparado para examinar un algoritmo para la función mostrada en la figura 9.24. Ya se han descrito los argumentos de *inserta()*. También existen algunas variables locales importantes:

PAGINA	La página que de momento está examinando <i>inserta()</i> .
PAGINA NUEVA	La nueva página que se crea si ocurre una división.
POS	La posición en PAGINA donde ocurre la llave (si está presente) u ocurriría (si se inserta).
NRR_P_A	El número relativo de registro promovido <i>desde abajo y hasta este nivel</i> . Si ocurre una división en el siguiente nivel inferior, NRR_P_A contiene el número relativo de registro de la nueva página que se creó durante la división. NRR_P_A es el hijo derecho que se inserta con la LLAVE_P_A en PAGINA.
LLAVE_P_A	La llave promovida <i>desde abajo y hasta este nivel</i> . Esta llave, junto con NRR_P_A, se inserta en PAGINA.

Cuando se codifica en un lenguaje real, *inserta()* emplea varias funciones de apoyo. La obvia es *divide()*, la cual crea una página nueva, distribuye las llaves entre la página original y la nueva, y determina cuál llave y cuál NRR hay que promover. La figura 9.25 contiene una descripción detallada de un procedimiento simple de *divide()*, el cual también se codifica en C y Pascal al final del capítulo.

Debe ponerse especial atención en la forma en que *divide()* mueve datos. Nótese que sólo se promueve la llave a partir de la página de trabajo: todos los HIJO_D_NRR son transferidos de regreso a PAGINA y PAGINANUEVA. El NRR que se promueve es el NRR de PAGINA-

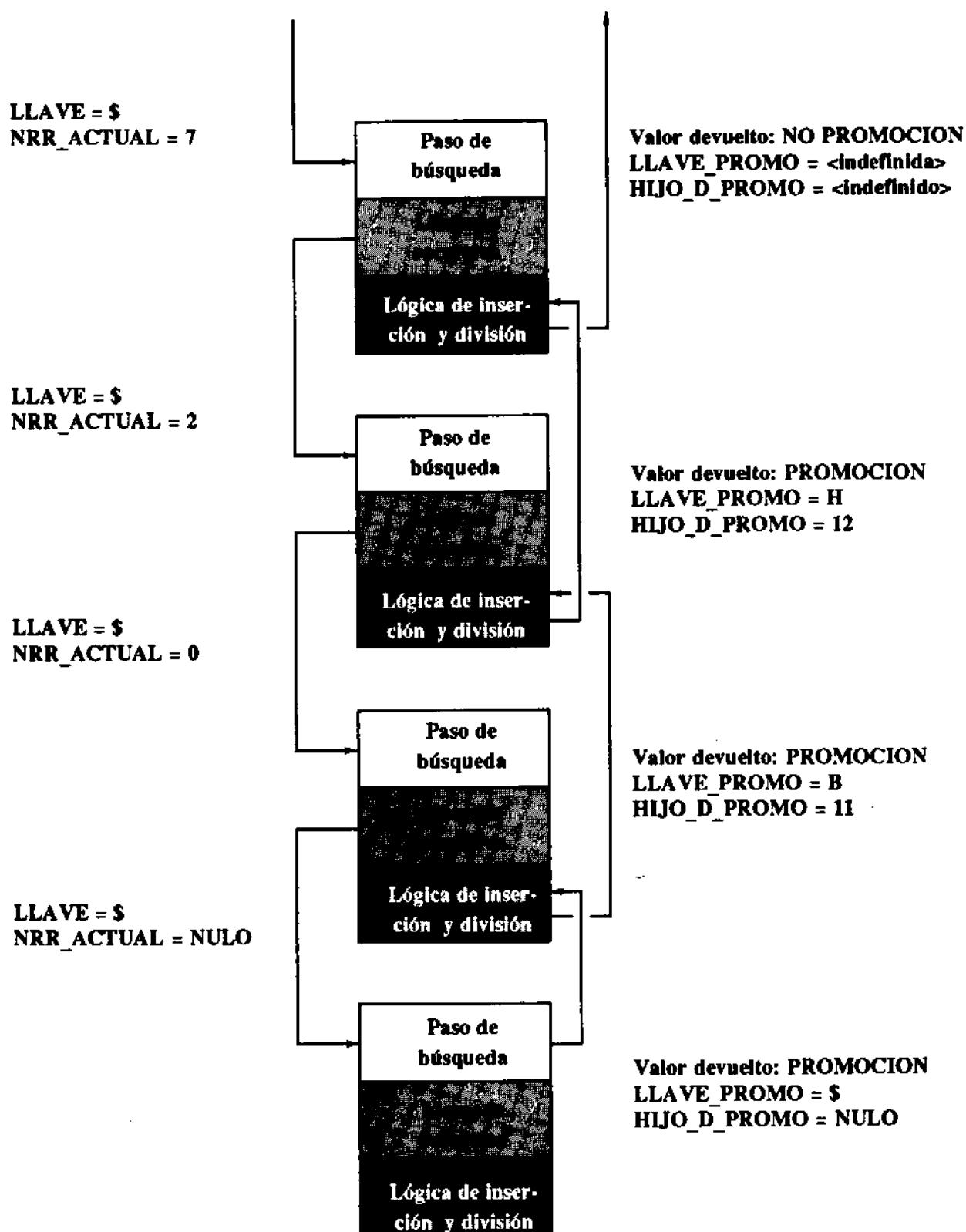


FIGURA 9.23 • Patrón de llamadas recursivas para insertar \$ dentro del árbol B que se ilustra en la figura 9.22.

```

FUNCION: inserta (NRR_ACTUAL, LLAVE, HIJO_D_PROMO, LLAVE_PROMO)

si NRR_ACTUAL = NULO entonces /* Después del fondo del árbol */
    LLAVE_PROMO := LLAVE
    HIJO_D_PROMO := NULO
    devuelve PROMOCION /* promueve la llave original y NULO */
otro
    leer la página de NRR_ACTUAL en PAGINA
    buscar la LLAVE en PAGINA. LLAVE (LLAVE)
    POS: = la posición en donde está debería estar LLAVE.

si se encuentra LLAVE entonces
    emitir mensaje de error indicando llave duplicada
    devuelve ERROR
VALOR_DEVUELTO := inserta (PAGINA.HIJO [POS], LLAVE, NRR_P_A,
                           LLAVE_P_A)
si VALOR_DEVUELTO == NO PROMOCION o ERROR entonces
    devuelve VALOR_DEVUELTO
otro si hay espacio en PAGINA para LLAVE_P_A entonces
    insertar LLAVE_P_A y NRR_P_A (promovida desde abajo, en PAGINA)
    devuelve NO PROMOCION
otro
    divide (LLAVE_P_A, NRR_P_A, PAGINA, LLAVE_PROMO, HIJO_D_PROMO,
            PAGINANUEVA)
    escribe PAGINA al archivo en el NRR_ACTUAL
    escribe PAGINANUEVA al archivo en el nrr HIJO_D_PROMO
    devuelve PROMOCION /* promoción de LLAVE_PROMO e HIJO_D_PROMO */
fin si

fin FUNCION

```

FIGURA 9.24 • La función *inserta (NRR_ACTUAL, LLAVE, HIJO_D_PROMO, LLAVE_PROMO)* inserta una LLAVE en un árbol B. El intento de inserción comienza en la página con el número relativo de registro NRR_ACTUAL. Si esta página no es una página hoja, la función se llama a sí misma en forma recursiva hasta que encuentra LLAVE en una página o alcanza una hoja. Si encuentra LLAVE, emite un mensaje de error y termina, devolviendo ERROR. Si existe espacio para LLAVE dentro de PAGINA, se inserta LLAVE. En caso contrario, PAGINA se divide. Una división asigna el valor de la llave de enmedio a LLAVE_PROMO, y el número relativo de registro de la nueva página creada a HIJO_D_PROMO, de tal modo que la inserción pueda continuar en el retorno recursivo ascendente del árbol. Si ocurre una promoción, *inserta()* lo indica devolviendo PROMOCION. En caso contrario, devuelve NO PROMOCION.

NUEVA, porque PAGINANUEVA es el descendiente derecho de la llave que se promueve. La figura 9.26 ilustra la actividad de las páginas

PROCEDIMIENTO: divide (LLAVE_I, NRR_I, PAGINA, LLAVE_PROMO,
HIJO_D_PROMO, PAGINANUEVA)

copiar todas las llaves y apuntadores de PAGINA en una página de trabajo que puede almacenar una llave adicional y un hijo.

insertar LLAVE_I y NRR_I en sus lugares correctos en la página de trabajo.

asignar e iniciar nueva página en el archivo del árbol B para que contenga PAGINANUEVA.

asignar a LLAVE_PROMO el valor de la llave de en medio, la cual será promovida después de la división.

asignar a HIJO_D_PROMO el NRR de PAGINANUEVA.

copiar las llaves y apuntadores a los hijos que preceden a LLAVE_PROMO, de la página de trabajo a PAGINA.

copiar las llaves y apuntadores a los hijos que siguen a LLAVE_PROMO, de la página de trabajo a PAGINANUEVA.

fin PROCEDIMIENTO

FIGURA 9.25 • *divide(LLAVE_I, NRR_I, PAGINA, LLAVE_PROMO, HIJO_D_PROMO, PAGINANUEVA)*, procedimiento que inserta LLAVE_I y NRR_I, provocando saturación. Crea una nueva página llamada PAGINANUEVA, distribuye las llaves entre la PAGINA original y la PAGINANUEVA, y determina cuál llave y NRR se debe promover. La llave y el NRR promovidos se devuelven por medio de los argumentos LLAVE_PROMO e HIJO_D_PROMO.

de trabajo entre PAGINA Y PAGINANUEVA, la página con que se está trabajando, y los argumentos de la función.

La versión de *divide()* descrita aquí es menos eficiente de lo que se pudiera desear, ya que mueve más datos de los necesarios. En el ejercicio 16 se pide al lector realizar una versión más eficiente de *divide()*.

EL NIVEL SUPERIOR. Se necesita una rutina para unir los procedimientos *inserta()* y *divide()*, y para realizar algunas operaciones que las rutinas de nivel inferior no llevan a cabo. El manejador debe ser capaz de hacer lo siguiente:

- Abrir o crear el archivo del árbol B, e identificar o crear la página raíz.
- Leer las llaves que van a almacenarse en el árbol B, y llamar a *inserta()* para colocarlas.
- Crear un nuevo nodo raíz cuando *inserta()* divide la página que en ese momento es la página raíz.

El contenido de PAGINA se copia a la página de trabajo.

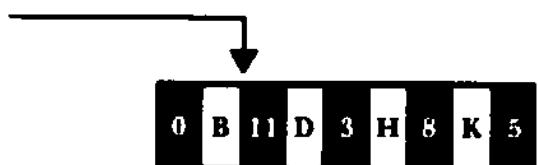
PAGINA



Página de trabajo



LLAVE_I(B) y NRR_I(11) se insertan en la página de trabajo.



El contenido de la página de trabajo se divide entre PAGINA y PAGINANUEVA, excepto por la llave de enmedio (H). H se promueve, junto con el NRR(12) de PAGINANUEVA.

PAGINA



PROMO_RRN

PAGINANUEVA

12

PROMO_KEY



PAGINANUEVA

FIGURA 9.26 • Movimiento de los datos en *divide()*.

La rutina *manejador()* mostrada en la figura 9.27 lleva a cabo estas tareas de alto nivel. Se supone que en NRR de la raíz está almacenado en el mismo archivo del árbol B, si el archivo existe. Si existe, *manejador()* lo abre y toma el NRR de nodo raíz. Si no existe, *manejador()* debe crear el archivo y crear la página raíz original. Como una raíz debe contener por lo menos una llave, esto implica tomar la primera llave que se va a insertar en el árbol y colocarla en la raíz. En seguida, *manejador()* lee las llaves que se van a insertar, una por una, y llama a *inserta()* para insertarlas en el archivo de árbol B. Si *inserta()* divide el nodo raíz, promueve una llave y su hijo derecho en LLAVE_PPROMO e HIJO_D_PROMO, y *manejador()* los usa para crear una raíz nueva.

PROCEDIMIENTO PRINCIPAL: manejador

```

    si el archivo del árbol B existe, entonces
        abrir el archivo del árbol B
    otro
        crear un archivo de árbol B y colocar la primera llave en la raíz
        tomar el NRR de la página raíz del archivo y almacenarla en RAIZ
        tomar una llave y almacenarla en LLAVE
        mientras existan llaves
            si (inserta (RAIZ, LLAVE, HIJO_D_PROMO, LLAVE_PROMO) == PROMOCION)
                entonces
                    crear una nueva página raíz con la llave      := HIJO_D_PROMO
                    hijo izquierdo := RAIZ e hijo derecho := HIJO_D_PROMO
                    asignar#RAIZ el NRR de la nueva página raíz
                    tomar la siguiente llave y almacenarla en LLAVE#fin mientras
                    escribe el NRR almacenado en RAIZ de regreso al archivo del árbol B
                    cerrar el archivo del árbol
    fin PROCEDIMIENTO PRINCIPAL

```

FIGURA 9.27 • Manejador para la construcción de un árbol B.

9.10

NOMENCLATURA DE ARBOLES B

Antes de pasar al análisis del desempeño de los árboles B y a las variaciones sobre sus algoritmos básicos, es necesario formalizar la terminología sobre los árboles B. Las definiciones cuidadosas de términos tales como *orden* y *hoja* permiten establecer en forma precisa las propiedades que deben estar presentes en una estructura de datos para cumplir con los requisitos de un árbol B. Esta definición de las propiedades de los árboles B, a su vez, documenta el análisis de temas como el procedimiento de eliminación de llaves de un árbol B.

Desgraciadamente, en la literatura sobre árboles B no hay uniformidad en cuanto al uso de los términos relacionados con ellos. Por lo tanto, abordar la bibliografía y mantenerse informados de los nuevos desarrollos requiere flexibilidad y algunas bases: el lector necesita estar advertido sobre los diferentes usos de algunos de los términos fundamentales.

Por ejemplo, Bayer y McCreight [1972], Comer [1979] y algunos otros se refieren al *orden* de un árbol B como el *mínimo* número de *llaves* que pueden estar dentro de una página de un árbol. En esta forma, el ejemplo inicial de árbol B (Fig. 9.16), que puede almacenar un *máximo* de siete llaves por página, tiene un *orden* de tres, empleando la terminología de Bayer y McCreight. El problema con esta definición de orden es que se vuelve difícil de aplicar cuando se toman en cuenta páginas que almacenan un número máximo de llaves que es *impar*. Por

ejemplo, considérese la siguiente pregunta: dentro del esquema de Bayer y McCreight, ¿la página de un árbol B de la orden tres está llena cuando contiene seis llaves o cuando contiene siete?

Knuth [1973b] y otros han resuelto la confusión par/impar definiendo el *orden* de un árbol B como el número *máximo* de *descendientes* que puede tener una página. Esta es la definición de *orden* que se usa en este texto. Nótese que esta definición difiere de la de Bayer y McCreight en dos aspectos: hace referencia a un *máximo*, no a un *mínimo*, y cuenta *descendientes* en vez de *llaves*.

El uso de la definición de Knuth debe acoplarse al hecho de que el número de llaves en una página de árbol B siempre es uno menos que el número de descendientes de la página. En consecuencia, un árbol B de orden 8 tiene un máximo de siete llaves por página. En general, para un árbol B de orden m , el número máximo de llaves por página es $m - 1$.

Cuando se divide la página de un árbol B, los descendientes se dividen lo más equitativamente posible entre la página nueva y la vieja. En consecuencia, cada página, excepto la raíz y las hojas, tiene *al menos* $m/2$ descendientes. Expresado en términos de una función de redondeo hacia arriba, puede decirse que el mínimo número de descendientes es $\lceil m/2 \rceil$. Se concluye que el *mínimo* número de *llaves* por página es $\lceil m/2 \rceil - 1$, de tal forma que el árbol B inicial de ejemplo tiene orden ocho, lo que significa que no puede almacenar más de siete llaves por página, y que todas las páginas, excepto la raíz, contiene por lo menos tres llaves.

Otro de los términos que los autores emplean en forma diferente es *hoja*. Bayer y McCreight se refieren al nivel más bajo de llaves dentro de un árbol B como el nivel *hoja*. Esto es consistente con la nomenclatura que se ha usado en este texto. Otros autores, entre ellos Knuth, consideran que las hojas de un árbol B están un nivel *por debajo* del nivel más bajo de llaves. En otras palabras, consideran que las hojas son los registros de datos reales a los cuales puede apuntar el nivel más bajo de llaves del árbol. Aquí no se usará esta definición; en su lugar se considera la idea de la hoja como el nivel más bajo de llaves en el árbol B.

9.11

DEFINICION FORMAL DE LAS PROPIEDADES DE LOS ARBOLES B

Con estas definiciones de orden y hoja puede formularse una definición precisa de las propiedades de un árbol B de orden m .

1. Cada página tiene un máximo de m descendientes.
2. Cada página, excepto la raíz y las hojas, tiene por lo menos $\lceil m/2 \rceil$ descendientes.
3. La raíz tiene por lo menos dos descendientes (a menos que sea una hoja).
4. Todas las hojas aparecen en el mismo nivel.
5. Una página que no es hoja con k descendientes contiene $k - 1$ llaves.
6. Una página hoja contiene por lo menos $\lceil m/2 \rceil - 1$ llaves y no más de $m - 1$ llaves.

9.12

PROFUNDIDAD DE LA BUSQUEDA EN EL PEOR CASO

Es importante comprender en término cuantitativos la relación entre el tamaño de la página de un árbol B, el número de llaves que serán almacenadas en el árbol, y el número de niveles que el árbol puede abarcar. Por ejemplo, si se sabe que necesitan almacenarse 1 000 000 de llaves y que, dada la naturaleza del hardware de almacenamiento disponible y el tamaño de las llaves, es razonable emplear un árbol B de orden 512 (un máximo de 511 llaves por página). Tomando en cuenta estos dos hechos, se necesita poder contestar la pregunta "En el peor caso, ¿cuál será el número máximo de accesos a disco requeridos para localizar una llave en el árbol?" Esto es lo mismo que preguntar cuál será la profundidad del árbol.

En principio, puede contestarse esta pregunta con la observación de que el número de descendientes de cualquier nivel de un árbol B es uno más que el número de llaves contenidas en ese nivel y todos los que lo preceden. La figura 9.28 ilustra esta relación para el árbol que se construyó anteriormente en el capítulo. Este árbol contiene 27 llaves (todas las letras del alfabeto y \$). Si se cuenta el número de descendientes potenciales que se desprenden del nivel hoja, se observará que hay 28.

A continuación es necesario observar que puede usarse la definición formal de las propiedades de los árboles B para calcular el número *mínimo* de descendientes que pueden extenderse desde cualquier nivel de un árbol B de algún orden dado. Esto es importante, porque se está interesado en la profundidad del *peor caso* del árbol. El peor caso ocurre cuando cada página del árbol tiene sólo el número mínimo de descendientes. En tal caso las llaves se distribuyen sobre una *altura máxima* para el árbol y una *amplitud mínima*.

Para un árbol B de orden m , el número mínimo de descendientes a partir de la página raíz es dos, así que el segundo nivel del árbol

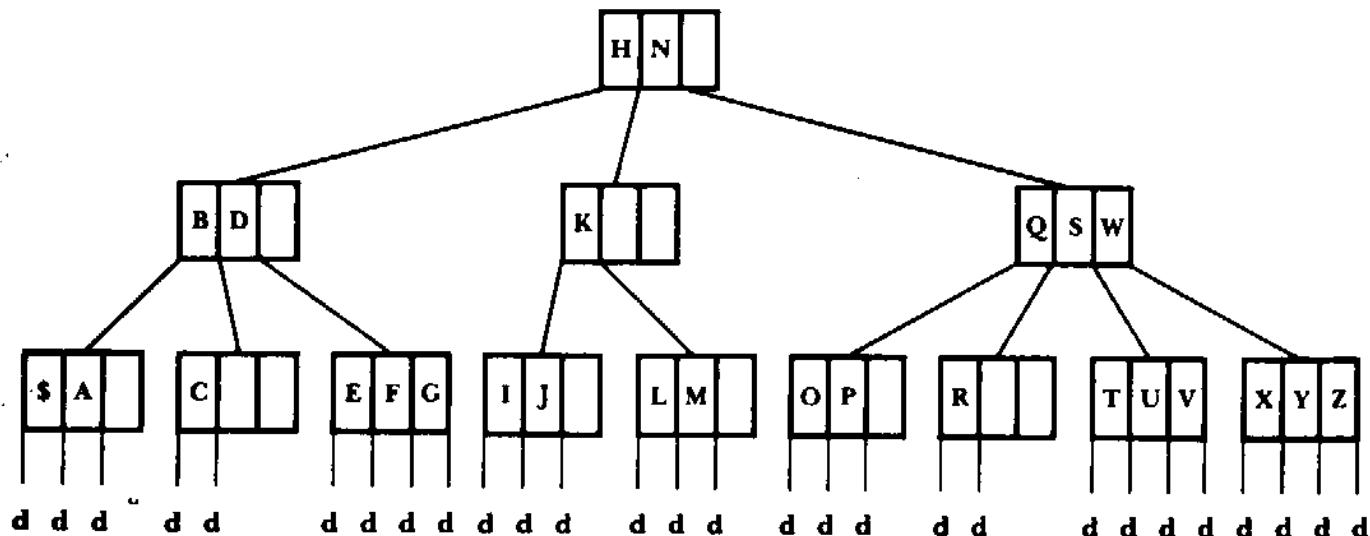


FIGURA 9.28 • Un árbol B con N llaves puede tener $(N+1)$ descendientes a partir del nivel hoja.

contiene sólo dos páginas. Cada una de estas páginas, a su vez, tiene al menos $\lceil m/2 \rceil$ descendientes. Por tanto, el tercer nivel contiene

$$2 \times \lceil m/2 \rceil$$

páginas. Puesto que cada una de estas páginas, una vez más, tiene un mínimo de $\lceil m/2 \rceil$ descendientes, el patrón general de la relación entre la profundidad y el número mínimo de descendientes toma la siguiente forma:

Nivel	Número mínimo de descendientes
1 (raíz)	2
2	$2 \times \lceil m/2 \rceil$
3	$2 \times \lceil m/2 \rceil \times \lceil m/2 \rceil + 2 \times \lceil m/2 \rceil^2$
4.	$2 \times \lceil m/2 \rceil^3$
.	.
.	.
.	.
d	$2 \times \lceil m/2 \rceil^{d-1}$

Así, en general, para cualquier nivel d de un árbol B, el mínimo número de descendientes que se extienden a partir de es nivel es

$$2 \times \lceil m/2 \rceil^{d-1}$$

Se sabe que un árbol con N llaves tiene $N + 1$ descendientes desde su nivel hoja. A la profundidad del árbol en el nivel hoja se le llamará

d. Se puede expresar la relación entre los $N + 1$ descendientes y el número mínimo de descendientes de un árbol de altura d como

$$N + 1 \geq 2 \times \lceil m/2 \rceil^{d-1}$$

puesto que se sabe que el número de descendientes de cualquier árbol no puede ser menor que el número para el peor caso de un árbol de esa profundidad. Despejando d , se llega a la siguiente expresión:

$$d \leq 1 + \log_{\lceil m/2 \rceil} ((N + 1)/2).$$

Esta expresión proporciona un límite superior para la profundidad de un árbol B con N llaves. Se encontrará el límite superior para el árbol hipotético que se describe al inicio de esta sección: un árbol de orden 512 que contenga 1 000 000 de llaves. Al sustituir estos números específicos dentro de la expresión, se encuentra que

$$d \leq 1 + \log_{256} 500000.5,$$

o sea,

$$d \leq 3.37.$$

De esta manera se puede decir que, dadas 1 000 000 de llaves, un árbol B de orden 512 tiene una profundidad de no más de tres niveles.

9.13

ELIMINACION, REDISTRIBUCION Y CONCATENACION

La indización de 1 000 000 de llaves en no más de tres niveles de un árbol es precisamente el tipo de eficiencia que se busca. Como se vio, este desempeño se predice considerando las propiedades de los árboles B descritas anteriormente; en particular, la posibilidad de garantizar que los árboles B sean amplios y bajos en lugar de delgados y profundos está ligada con las reglas que establecen que:

- cada página, excepto la raíz y las hojas, tiene por lo menos $\lceil m/2 \rceil$ descendientes;
- una página que no es hoja con k descendientes contiene $k - 1$ llaves, y
- una página hoja contiene por lo menos $\lceil m/2 \rceil - 1$ y no más de $m - 1$ llaves.

Ya se ha visto que el proceso de la división de páginas garantiza que estas propiedades se mantengan cuando se insertan llaves dentro del árbol. Es necesario desarrollar algún tipo de garantía igualmente confiable para que se mantengan estas propiedades cuando se *eliminan* llaves del árbol.

Trabajar a mano a través de situaciones simples de eliminación ayudará a demostrar que la eliminación de una llave puede ocasionar varias situaciones diferentes. La figura 9.29 ilustra cada una de estas situaciones y las respuestas asociadas con ellas cuando se hacen varias eliminaciones en un árbol B de orden seis.

La situación más sencilla se ilustra en el caso 1. Eliminar la llave *J* no provoca que el contenido de la página 5 caiga por debajo del mínimo número de llaves. En consecuencia, la eliminación no implica más que quitar la llave de la página y reacomodar las llaves *dentro* de la página para ajustar el espacio.

Eliminar la *M* (caso 2) es más complejo. Si se elimina simplemente la *M* de la raíz, resulta muy difícil reorganizar el árbol para mantener su estructura de árbol B. Como este problema puede ocurrir siempre que se elimine una llave de una página que no sea hoja, siempre se eliminarán llaves de páginas hojas. Si se va a eliminar una llave que no está en una hoja, existe un forma sencilla de ponerla en una hoja: se intercambia con su sucesor inmediato, el cual se garantiza que está en una hoja, y después se le elimina inmediatamente de la hoja. En el ejemplo, se puede intercambiar la *M* con la *N* en la página 6, y entonces eliminar la *M* de la página 6. Esta operación simple no pone a la *N* fuera de orden, puesto que todas las llaves del subárbol del cual *N* es parte deben ser más grandes que *N*. (¿Puede el lector percibir por qué esto es así?)

En el caso 3 se elimina *R* de la página 7. Si simplemente se elimina *R* y no se hace nada más, la página en cuestión tiene sólo una llave. El número mínimo de llaves para una página hoja en un árbol de orden seis es

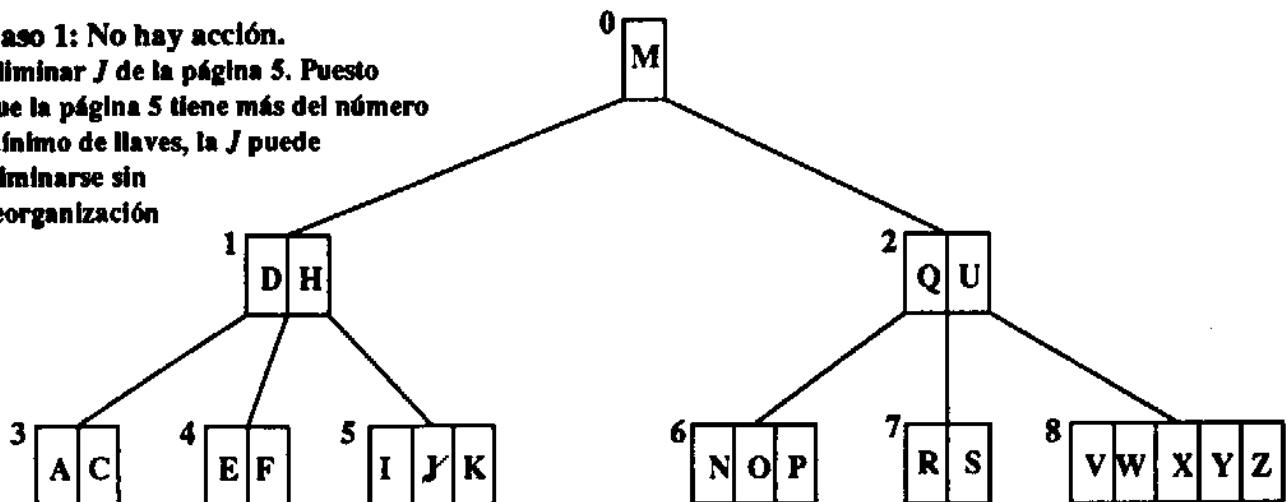
$$\lceil 6/2 \rceil - 1 = 2.$$

Por lo tanto, tiene que tomarse algún tipo de acción para corregir este estado de insuficiencia. Puesto que en la página vecina 8 (llamada *hermana* porque tiene el mismo padre) se excede el número mínimo de llaves, la acción correctiva consiste en redistribuir las llaves entre las páginas. La *redistribución* también debe ocasionar un cambio en la llave que está en la página padre, para que continúe actuando como separador entre las páginas de nivel inferior. En el ejemplo, se trasladan la *U* y la *V* a la página 7, y la *W* a la posición de separador en la página 2.

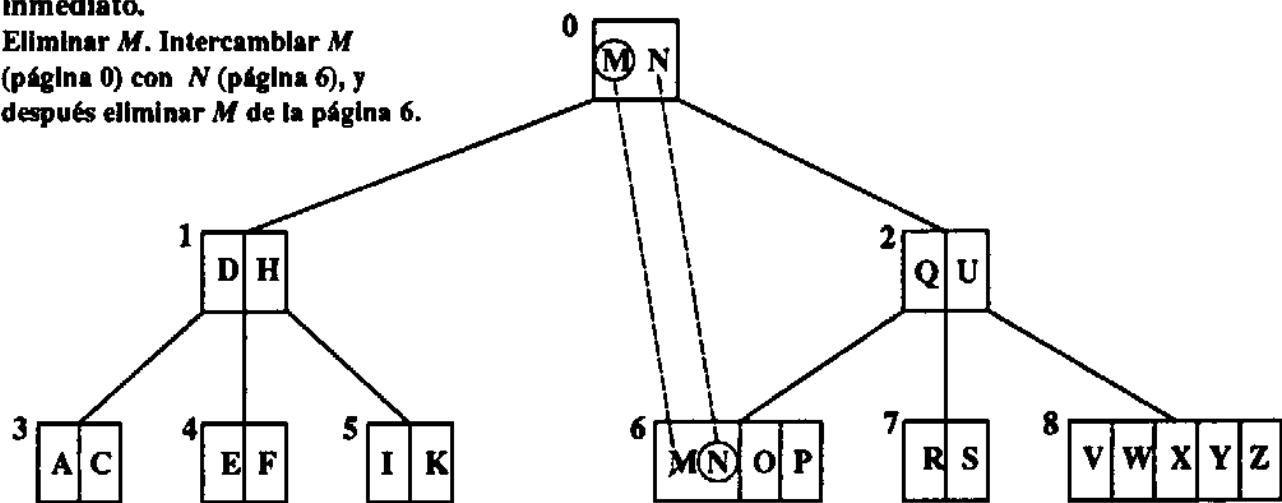
La eliminación de *A* en el caso 4 provoca una situación que no puede resolverse por redistribución. Al atacar la insuficiencia en la página 3

Caso 1: No hay acción.

Eliminar J de la página 5. Puesto que la página 5 tiene más del número mínimo de llaves, la J puede eliminarse sin reorganización

**Caso 2: Intercambio con el sucesor inmediato.**

Eliminar M. Intercambiar M (página 0) con N (página 6), y después eliminar M de la página 6.

**Caso 3: Redistribución.**

Eliminar R. Ocurre insuficiencia. Redistribuir llaves entre las páginas 2, 7 y 8 para restituir el balance entre las hojas.

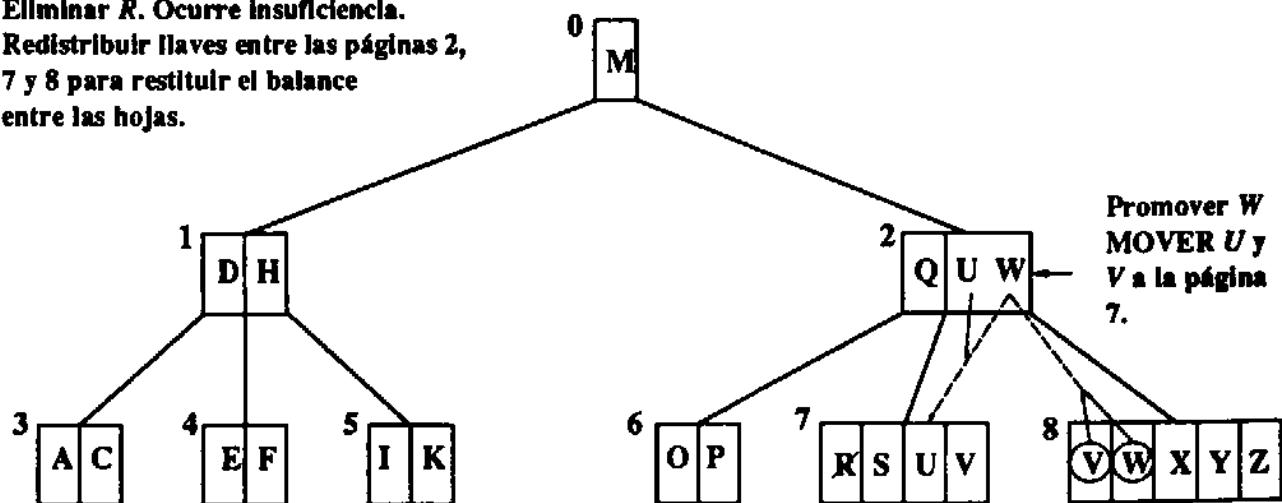
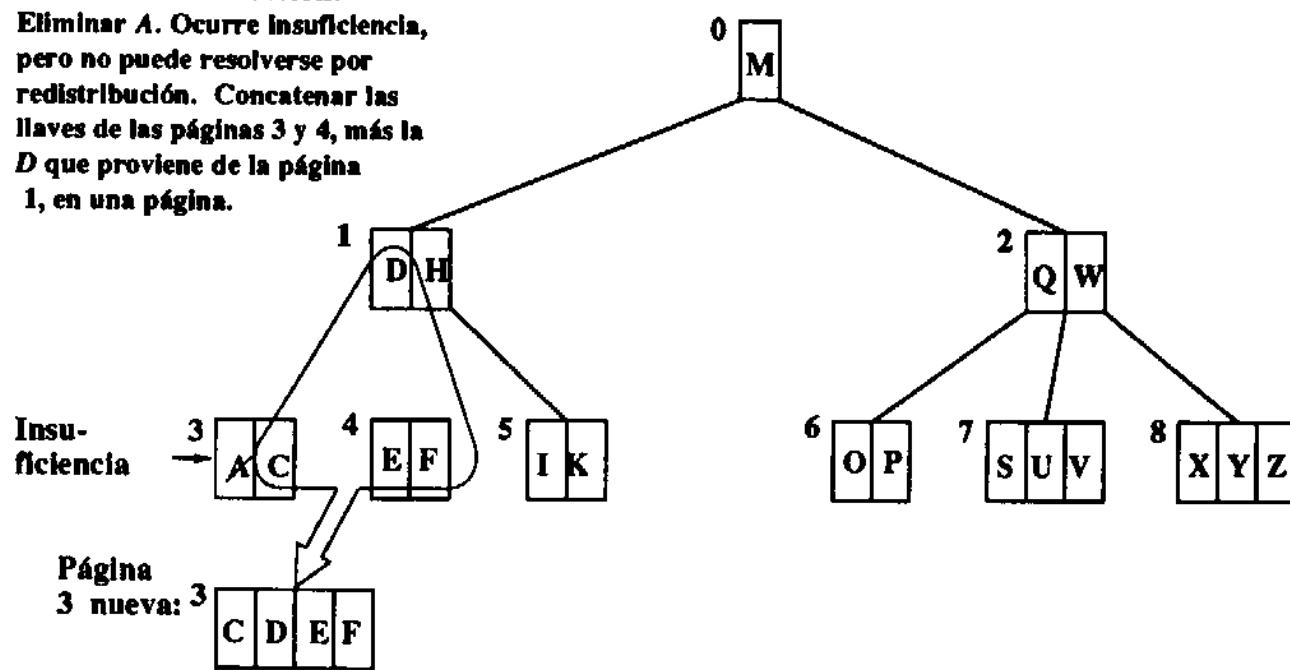
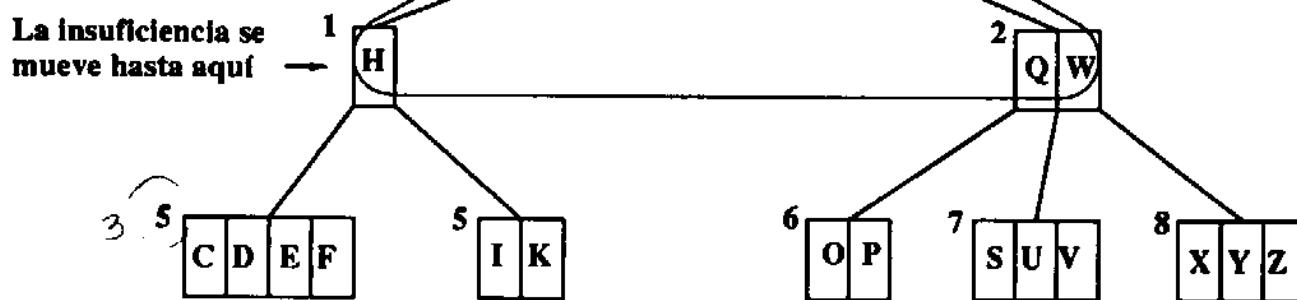
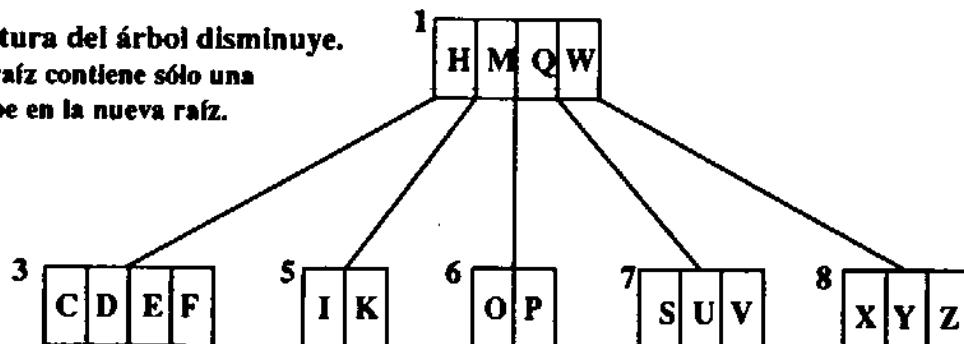


FIGURA 9.29 • Seis situaciones que pueden ocurrir durante las eliminaciones.

Caso 4: Concatenación.

Eliminar A. Ocurre insuficiencia, pero no puede resolverse por redistribución. Concatenar las llaves de las páginas 3 y 4, más la D que proviene de la página 1, en una página.

**Caso 5: La insuficiencia se propaga hacia arriba. Ahora la página 1 tiene insuficiencia. Una vez más, no puede redistribuirse, de modo que se concatena.****Caso 6: La altura del árbol disminuye. Puesto que la raíz contiene sólo una llave, se absorbe en la nueva raíz.****Figura 9.29 (continuación)**

moviendo llaves desde la página 4, sólo se transfiere el estado de insuficiencia. No hay suficientes llaves para repartir en dos páginas. La solución es *concatenar*, combinando las dos páginas y la llave de la página padre, para hacer una sola página completa.

La concatenación es, en esencia, lo inverso de la división. Como ella puede propagarse hacia arriba a través del árbol B. Así como la división promueve una llave, la concatenación debe implicar el descenso de llaves y esto, a su vez, puede causar insuficiencia en la página padre. Esto es lo que sucede en el ejemplo. La concatenación de las páginas 3 y 4 lleva a la llave *D* de la página padre hacia abajo, al nivel de hoja, y se llega al caso 5: la pérdida de la *D* de la página padre, a su vez, crea insuficiencia. De nuevo, la redistribución no resuelve el problema, por lo que debe usarse concatenación.

Nótese que la propagación del estado de insuficiencia no necesariamente implica la propagación de la concatenación. Si la página 2 (*Q* y *W*) hubiese tenido otra llave, entonces podría usarse redistribución y no concatenación para resolver la condición de insuficiencia en el segundo nivel del árbol.

El caso 6 muestra lo que sucede cuando la concatenación llega hasta la raíz. La concatenación de las páginas 1 y 2 absorbe la única llave de la página raíz, disminuyendo en un nivel la altura del árbol.

Los pasos de la eliminación de llaves de un árbol B pueden resumirse como sigue:

1. Si la llave que se va a eliminar no está en una hoja, intercambiarla con su sucesor inmediato, el cual está en una hoja.
2. Eliminar la llave.
3. Si la hoja ahora contiene por lo menos el número mínimo de llaves, no se requiere ninguna acción adicional.
4. Si la hoja ahora contiene una menos de las llaves mínimas, examínese los hermanos izquierdo y derecho.
 - a) Si un hermano tiene más del número mínimo de llaves, hay que redistribuir.
 - b) Si ninguno de los dos hermanos tiene más del mínimo, se concatenan las dos hojas y la llave de en medio del padre a una hoja.
5. Si las hojas se concatenaron, aplicar los pasos 3-6 al *padre*.
6. Si se elimina la última llave de la raíz, entonces la altura del árbol decrece.

9.13.1 REDISTRIBUCION

A diferencia de la concatenación, que es un tipo de división inversa, la redistribución es una idea nueva. El algoritmo de inserción no incluye operaciones análogas a la redistribución.

La redistribución difiere de la división y de la concatenación en que no se propaga. Se garantiza que tiene efectos estrictamente locales. Nótese que el término *hermandad* implica que las páginas tengan la misma página padre. Si hay dos nodos en el nivel hoja lógicamente adyacentes pero que no tiene el mismo padre (por ejemplo, IJK y NOP en el árbol de la parte superior de la Fig. 9.29), no son hermanos. En general, los algoritmos de redistribución se escriben de tal forma que no consideran el movimiento de llaves entre nodos que no son hermanos, aun cuando sean lógicamente adyacentes. ¿Puede visualizar el lector el razonamiento que hay detrás de esta restricción?

Otra diferencia entre la redistribución, por un lado, y la concatenación y la división, por el otro, es que no existe una regla fija o necesaria para reacomodar las llaves. Una sola eliminación en un árbol B adecuadamente formado no puede causar una insuficiencia de más de una llave. Por lo tanto, la redistribución puede restituir las propiedades del árbol B trasladando sólo una llave de un hermano a la página que ha tenido la insuficiencia, aun cuando la distribución de las llaves entre las páginas sea muy dispares. Supóngase, por ejemplo, que se maneja un árbol de orden 101. El número mínimo de llaves que pueden estar en una página es 50, y el máximo es 100. Suponiendo que se tiene una página que contiene el mínimo y un hermano que contiene el máximo, si se elimina una llave de la página que contiene 50 ocurre un estado de insuficiencia. Puede corregirse por medio de la redistribución al mover una llave, 50 llaves, o cualquier número entre 1 y 50. La estrategia usual es dividir las llaves lo más equitativamente posible entre las páginas. En este caso, eso significa mover 25 llaves.

9.14

REDISTRIBUCION DURANTE LA INSERCIÓN: UNA FORMA DE MEJORAR LA UTILIZACION DEL ALMACENAMIENTO

Como se recordará, la inserción en los árboles B no requiere una operación análoga a la redistribución; la división es capaz de encargarse de todas las posibilidades de saturación. Sin embargo, esto no significa que no sea deseable usar la redistribución como opción durante la inserción, en particular porque un conjunto de algoritmos de mantenimiento de árboles B ya debe incluir un procedimiento de redistribución para manejar la eliminación. Considerando que ya hay un procedimiento de redistribución, ¿qué ventajas pueden obtenerse empleándolo en lugar de la división de nodos?

La redistribución durante la inserción es una forma de evitar, o al menos de posponer, la creación de páginas nuevas. En lugar de dividir

una página completa y crear dos medio llenas, la redistribución permite acomodar dentro de otra página algunas de las llaves que provocan saturación. Por lo tanto, la redistribución en lugar de la división debe tender a hacer el árbol B más eficiente en términos de su utilización del espacio.

Es posible cuantificar esta eficiencia de utilización de espacio concibiendo la cantidad de espacio empleado para almacenar información como un porcentaje de la cantidad total de espacio requerido para almacenar el árbol B. Después de que un nodo se divide, cada una de sus dos páginas resultantes está llena casi a la mitad. De esta forma, en el peor caso, la utilización de espacio en un árbol B, usando división en dos, es de alrededor del 50 por ciento. Por supuesto, el grado real de la utilización de espacio es mejor que esta cifra del peor caso. VanDoren [1975] y Yao [1978] han demostrado que, para árboles grandes de orden relativamente alto, la utilización de espacio se aproxima a un promedio teórico de alrededor del 69 por ciento, cuando la inserción se realiza mediante la división en dos.

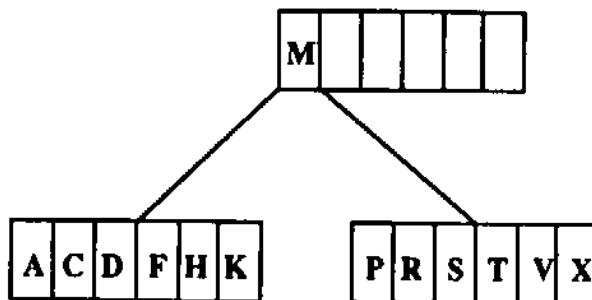
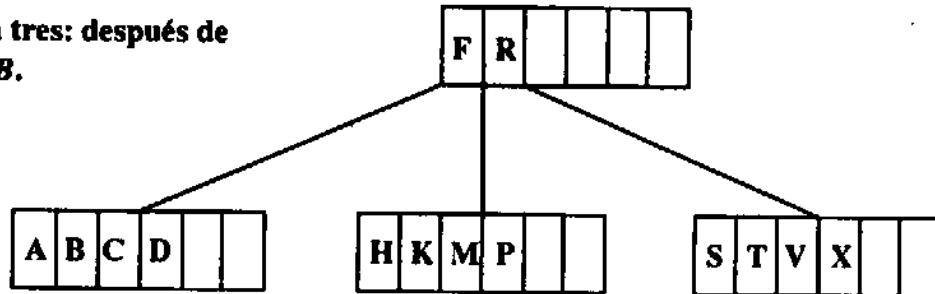
La idea de usar la redistribución como alternativa de la división, cuando sea posible, y de dividir una página sólo cuando ambos hermanos estén completos, se considera en el artículo original de Bayer y McCreight [1972]. El artículo incluye algunos resultados experimentales que demuestran que la división en dos resulta en una utilización de espacio del 67 por ciento para un árbol de orden 121 después de 5 000 inserciones aleatorias. Cuando se repitió el experimento, empleando redistribución cuando era posible, la utilización de espacio se incrementó por arriba del 86 por ciento. Pruebas empíricas subsecuentes, realizadas por Davis [1974] (árbol B de orden 49) y Crotzer [1975] (árbol B de orden 303), también mostraron una utilización de espacio que excedía el 85 por ciento cuando se usaba redistribución. Estos y otros resultados sugieren que cualquier aplicación seria de árboles B, aun para archivos moderadamente grandes, debe realizar los procedimientos de inserción que resuelven la saturación por medio de la redistribución, cuando sea posible.

9.15

ARBOLES B*

En su estudio y ampliación del trabajo sobre árboles B, Knuth [1973] extiende la idea de la redistribución durante la insercción para incluir nuevas reglas para la división. Knuth llama *árbol B** a la variación resultante de la forma fundamental de árbol B.

Considérese un sistema en el que se pospone la división mediante la redistribución, como se propuso en la sección anterior. Si se considera

Árbol original**División de dos a tres: después de insertar la llave B.****FIGURA 9.30 • División de dos a tres.**

cualquier página que no sea la raíz, se sabe que cuando finalmente llegue el tiempo de dividir, la página tendrá al menos un hermano que también estará completo. Esto abre la posibilidad de dividir de dos a tres en lugar de dividir uno a dos. La figura 9.30 ilustra dichas divisiones.

El aspecto más importante de esta división de dos a tres es que produce páginas que se llenan hasta las dos terceras partes, en vez de sólo la mitad. Esto hace posible definir un nuevo tipo de árbol B, llamado árbol B*, que tiene las siguientes propiedades:

1. Cada página tiene un máximo de m descendientes.
2. *Cada página, excepto la raíz y las hojas, tiene al menos $(2m-1)/3$ descendientes.*
3. La raíz tiene al menos dos descendientes (a menos de que sea hoja).
4. Todas las hojas aparecen en el mismo nivel.
5. Una página que no sea hoja con k descendientes contiene $k-1$ llaves.
6. *Una página hoja contiene por lo menos $\lfloor (2m-1)/3 \rfloor$ llaves, y no más de $m-1$.*

Los cambios críticos entre este conjunto de propiedades y el conjunto que se define para un árbol B convencional están en las reglas 2 y 6: un árbol B* tiene páginas que contienen un mínimo de $\lfloor (2m-1)/3 \rfloor$ llaves. Por supuesto, esta nueva propiedad afecta los procedimientos de eliminación y redistribución.

Para realizar los procedimientos de árboles B* también se debe abordar la cuestión de dividir la raíz, la cual, por definición, nunca tiene hermanos. Si no existen hermanos, no es posible la división de dos a tres. Knuth sugiere permitir que la raíz crezca hasta un tamaño mayor que las demás páginas, de tal forma que, cuando se divide, pueda producir dos páginas cada una llena casi a las dos terceras partes. Esta sugerencia tiene la ventaja de asegurar que todas las páginas por debajo del nivel de la raíz se adhieren a las características de los árboles B*. Sin embargo, tiene la desventaja de requerir que los procedimientos sean capaces de manejar una página que sea de mayor tamaño que todas las demás. Otra solución es realizar la división de la raíz como una división convencional de uno a dos. Esta segunda solución evita cualquier lógica especial de manejo de página. Por otro lado, complica la eliminación, la redistribución y otros procedimientos que deben ser sensibles al número mínimo de llaves permitidas en una página. Tales procedimientos tendrían que ser capaces de reconocer que las páginas descendientes de la raíz legalmente pueden estar sólo llenas.

9.16

MANEJO DE PAGINAS EN BUFFERS: ARBOLES B VIRTUALES

Se ha visto que, gracias a algunos refinamientos adicionales, el árbol B puede ser una estructura de almacenamiento muy eficiente y flexible, que mantiene sus propiedades de balanceo luego de repetidas eliminaciones e inserciones y que proporciona acceso a cualquier llave con sólo unos cuantos accesos al disco. Sin embargo, al enfocar sólo los aspectos estructurales, como se ha hecho hasta ahora, se corre el riesgo de omitir otras formas de emplear esta estructura para obtener las mayores ventajas. Por ejemplo, el hecho de que un árbol B tenga una profundidad de tres niveles para nada significa que se necesite hacer tres accesos al disco para extraer de las páginas del nivel hoja. Eso se puede mejorar mucho.

Obtener mejor desempeño de los árboles B implica examinar detalladamente el problema original. Es necesario encontrar una forma de hacer un uso eficiente de los índices que son demasiado grandes para almacenarse *en su totalidad* en memoria RAM. Hasta este punto se ha enfocado el problema con la idea de todo o nada: un índice se ha almacenado completo en la memoria RAM, organizado como lista o árbol binario, o bien se ha guardado todo en el almacenamiento secundario, empleando una estructura de árbol B. Pero al afirmar que no puede almacenarse todo el índice en la memoria RAM no implica que no pueda almacenarse parte de él.

Por ejemplo, supóngase que se tiene un índice que contiene un megabyte de registros y que no es posible usar más de 256K de memoria RAM para almacenar el índice. Considerando un tamaño de página de 4K, que almacene alrededor de 64 llaves por página, el árbol B puede estar contenido en tres niveles. Es posible obtener cualquiera de las llaves en no más de tres accesos a disco. Esto es ciertamente aceptable, pero ¿por qué debe uno contentarse con este tipo de desempeño? ¿Por qué no se intenta encontrar una forma de hacer que el número promedio de accesos al disco por búsqueda descienda hasta uno o menos?

Considerando este problema estrictamente en términos de estructuras de almacenamiento físico, una extracción que utilice en promedio un acceso a disco o menos parece algo imposible. Pero recuérdese que el objetivo fue encontrar una forma de manejar el megabyte de índice con 256K de memoria RAM, no con los 4K necesarios para almacenar una sola página del árbol.

Se sabe que toda búsqueda en el árbol requiere acceder a la página raíz. En vez de acceder a la página raíz una y otra vez al inicio de cada búsqueda, podría transferirse la página raíz a memoria RAM y mantenerla ahí. Esta estrategia incrementa el requerimiento de memoria RAM de 4K a 8K, porque se necesitan 4K para la raíz y 4K para cualquier otra página que se lea, pero esto es aún mucho menor que los 256K que están disponibles. Esta estrategia tan simple reduce la búsqueda en el peor caso a dos accesos al disco, y la búsqueda promedio a menos de dos accesos (las llaves en la raíz no requieren acceso al disco; las llaves en el primer nivel requieren un acceso).

Esta sencilla estrategia de mantener la raíz sugiere un importante enfoque más general: en lugar de almacenar sólo la página raíz en memoria RAM, se crea un *buffer de páginas* para almacenar algunas páginas del árbol B, quizás 5, 10 o más. Conforme se leen páginas del disco en respuesta a las solicitudes del usuario, se va llenando el buffer. Entonces, cuando se solicita una página, se accede a ella desde la memoria RAM, si se puede, evitando así un acceso al disco. Si la página no está en la memoria RAM, entonces se la transfiere al buffer desde el almacenamiento secundario, reemplazando una de las páginas que estaban allí. A un árbol B que usa un buffer de memoria RAM en esta forma se le llama *árbol B virtual*.

9.16.1 REEMPLAZO LRU

Está claro que tal esquema de manejo de buffers funciona sólo cuando es más probable que se solicite una página que está en el buffer que una que no esté allí. Al proceso de acceder al disco para extraer una página que no está en el buffer se le llama *interrupción por página*. Existen dos causas de interrupciones por página.

1. Nunca se ha usado la página.
2. Ya estuvo en el buffer pero ha sido reemplazada con una página nueva.

La primera causa de interrupción por página es inevitable: si aún no se ha leído y usado una página, no hay forma de que pueda estar ya en el buffer. Pero puede intentarse minimizar la segunda causa por medio del manejo de buffers. La decisión crítica aparece cuando necesita transferirse una página nueva a un buffer que ya está completo: ¿cuál página habrá que reemplazar?

Un método común es reemplazar la página cuyo uso es menos reciente; se conoce como reemplazo *LRU* (por sus siglas en inglés: *least recently used*). Nótese que esto no es lo mismo que reemplazar la página que se transfirió al buffer menos recientemente. Puesto que la página raíz siempre se lee primero, el simple reemplazo de la página más vieja resulta ser el reemplazo de la raíz, lo cual es un resultado indeseable. En lugar de esto, el método LRU mantiene información acerca de los pedidos reales de páginas. Como la raíz se solicita en cada búsqueda, casi nunca se selecciona para reemplazo. La página que va a reemplazarse es aquella que lleva más tiempo sin una solicitud de uso.

Algunas investigaciones realizadas por Webster [1980] muestran el efecto que tiene incrementar el número de páginas que pueden almacenarse en el área de buffer con una estrategia de reemplazo LRU. La tabla 9.1 resume una parte pequeña, pero representativa, de los resultados de Webster. Lista el número promedio de accesos al disco por búsqueda para cantidades diferentes de buffers de página. Estos resultados se obtienen empleando una estrategia de reemplazo LRU simple sin tomar en cuenta la altura de la página.

TABLA 9.1

Efecto de usar más buffers con una estrategia de reemplazo LRU simple

Páginas en el buffer	1	5	10	20
Número promedio de acceso por búsqueda	3.00	1.71	1.42	0.97

Número de llaves = 2400

Total de páginas = 140

Altura del árbol = 3 niveles

El estudio de Webster se llevó a cabo empleando árboles B⁺ en vez de árboles B simples. En el siguiente capítulo, donde se examinan con mayor detalles los árboles B⁺, se observará que su naturaleza explica el hecho de que, dado un buffer, la longitud media de búsqueda es 3.00. Con los árboles B⁺, todas las búsquedas deben recorrer todo el camino hasta el nivel de hoja durante cada vez. Sin embargo, no por el hecho de que Webster usará B⁺ sus resultados dejan de tener utilidad para ilustrar el impacto positivo del manejo de páginas de buffers. Mantener menos del 15 por ciento del árbol en memoria RAM (20 páginas del total de 140) reduce el número promedio de accesos por búsqueda a menos de uno. Los resultados son aún más drásticos con un árbol B simple, porque no todas las búsquedas tienen que llegar al nivel de hoja.

Nótese que la decisión de usar reemplazo LRU está basada en la suposición de que es más probable que se necesite una página que se ha usado recientemente que una que nunca se ha usado, o una que usó hace mucho tiempo. Si esta suposición no es válida, entonces no existe razón alguna para retener en forma preferente páginas que se hayan usado recientemente. El término para esta clase de suposición es *localidad temporal*. Se hace la suposición de que existe una especie de *acumulamiento* de uso de ciertas páginas en el tiempo. La naturaleza jerárquica de un árbol B hace que este tipo de suposiciones sea razonable.

Por ejemplo, durante la redistribución, después de una saturación o una insuficiencia, se accede a una página y después se accede a sus hermanas. Como los árboles B son jerárquicos, el acceder a un conjunto de páginas hermanas implica accesos repetidos a la página padre en sucesión rápida. Este es un ejemplo de localidad temporal; es fácil observar cómo está relacionada con la jerarquía del árbol.

9.16.2 REEMPLAZO SEGUN LA ALTURA DE LA PAGINA

Existe otra forma más directa de usar la naturaleza jerárquica para guiar las decisiones sobre el reemplazo de páginas en los buffers. La estrategia simple de mantener la raíz ejemplifica esta alternativa: siempre retener las páginas que ocurren en los niveles más altos del árbol. Dada una cantidad mayor de espacio para buffers, puede ser posible retener no sólo la raíz, sino todas las páginas del segundo nivel de un árbol.

Se analizará esta idea con un ejemplo previo, en el cual se tenía acceso a 256K de memoria RAM y un índice de 1 megabyte. Puesto que el tamaño de página es 4K, podría construirse un área de buffer que almacenara 64 páginas dentro del área de memoria. Supóngase que el

megabyte de índice requiere alrededor de 1.2 megabytes de almacenamiento en disco (utilización de almacenamiento del 83 por ciento). Considerando el tamaño de página de 4K, estos 1.2 megabytes requieren poco más de 300 páginas. Supóngase que el número promedio de descendientes de cada una de las páginas es de alrededor de 30. De lo anterior se deduce que el árbol de tres niveles tiene, por supuesto, una sola página en el nivel raíz, seguida de nueve o diez páginas en el segundo nivel, con todas las páginas restantes en el nivel hoja. Al emplear una estrategia de reemplazo que siempre retiene las páginas de nivel superior, está claro que el buffer de 64 páginas contendrá, a la larga, la raíz y todas las páginas del segundo nivel. Los aproximadamente 50 espacios de buffer restantes se usan para almacenar páginas de nivel hoja. Para decidir cuáles páginas se deben reemplazar puede considerarse una estrategia LRU. Para muchas búsquedas, todas las páginas requeridas ya estarán en el buffer, por lo que la búsqueda no requiere accesos al disco. Es fácil ver, dado un buffer adecuado, cómo se puede reducir el número promedio de accesos al disco por búsqueda a un número menor que uno.

El trabajo de Webster [1980] también explica el efecto de tomar en cuenta la altura de la página, dando preferencia a las páginas que están más arriba en el árbol cuando llega el momento de decidir cuáles mantener en los buffers. Aumentar la estrategia LRU con un factor de peso que tome en cuenta la altura de la página reduce el número promedio de accesos, para un buffer de diez páginas, de 1.42 accesos por búsqueda.

9.16.3 IMPORTANCIA DE LOS ARBOLES B VIRTUALES

Es difícil exagerar la importancia que tiene el incluir un esquema de manejo de buffers en cualquier implantación de una estructura de árboles B. Debido a que la estructura del árbol B es, por sí misma, tan interesante y poderosa, es fácil caer en la trampa de pensar que su organización misma es suficiente para solucionar el problema de acceder a índices grandes que deben mantenerse en almacenamiento secundario. Como se ha señalado, caer en esta trampa es perder de vista el problema original: encontrar una forma de *reducir* la cantidad de memoria requerida para manejar índices grandes. Sin embargo, no fue necesario reducir la cantidad de memoria a la cantidad requerida para una sola página de índice. Por lo regular, es posible encontrar memoria suficiente para almacenar varias páginas. Esto puede incrementar en forma drástica el desempeño del sistema.

9.17

COLOCACION DE LA INFORMACION ASOCIADA CON LA LLAVE

Anteriormente en este capítulo se puso atención en el índice de los árboles B, dejando a un lado cualquier consideración sobre la información real asociada con las llaves. Se parafraseó a Bayer y McCreight y se estableció que "la información asociada no tiene mayor interés".

Pero, por supuesto, en cualquier aplicación real la información asociada es, de hecho, el verdadero objeto de interés. Rara vez se desea indizar llaves sólo para poder encontrar las llaves mismas. Por lo regular es la información asociada con la llave la que en realidad se desea encontrar. Así que, antes de cerrar el análisis de los índices de árboles B, es importante preguntar dónde y cómo almacenar la información indizada por las llaves en el árbol. Fundamentalmente, se tiene dos opciones: se puede

- almacenar la información en el árbol B junto con la llave, o
- colocar la información en un archivo separado; dentro del índice se acopla la llave con un número relativo de registro, o un byte apuntador de dirección, que hace referencia a la localización de la información en ese archivo separado.

Las ventajas que tiene el primer enfoque sobre el segundo es que una vez que se encuentra la llave, no se requiere más accesos a disco. La información está ahí con la llave. Sin embargo, si la cantidad de información asociada con cada llave es relativamente grande, entonces almacenarla con la llave reduce el número de llaves que pueden colocarse en una página del árbol B. A medida que se reduce el número de llaves por página, se reduce el orden del árbol y tiende a ser de más altura, ya que hay menos descendientes de cada página. De esta forma, la ventaja del segundo método es que, suponiendo que la información asociada es de gran longitud en relación con la longitud de la llave, colocar la información asociada en otro lugar permite construir un árbol de orden más alto y, por lo tanto, tal vez de menor altura.

Por ejemplo, supóngase que se necesita indizar 1000 llaves y registros de información asociada, y que la longitud requerida para almacenar una llave y su información asociada es de 128 bytes. Además, supóngase que si se almacena la información asociada en otro lugar, se puede almacenar la llave y un apuntador a la información asociada en sólo 16 bytes. Considerando una página de árbol B que tuviese 512 bytes disponibles para llaves e información asociada, las dos alternativas fundamentales de almacenamiento se traducen en los siguientes órdenes de árboles B:

- Información almacenada con la llave:* cuatro llaves por página en un árbol de orden cinco, y
- Apuntador almacenado con la llave:* 32 llaves por página en un árbol de orden 33.

Usando la fórmula para encontrar la profundidad para el peor caso de los árboles B, desarrollada anteriormente:

$$\begin{aligned} d_{(\text{inf con la llave})} &\leq 1 + \log_3 500.5 = 6.66 \\ d_{(\text{inf en otro lugar})} &\leq 1 + \log_{17} 500.5 = 3.19 \end{aligned}$$

Así, si se almacena la información con las llaves, el árbol tiene una profundidad, para el peor caso, de seis niveles. Si se almacena la información en otro lugar, se termina reduciendo la altura del árbol en el peor caso a tres. Incluso aunque la indirección adicional asociada con el segundo método cuesta un acceso a disco, el segundo método todavía reduce el número total de accesos para encontrar un registro en el peor caso.

En general, entonces, la decisión acerca de dónde conviene almacenar la información debe guiarse por algunos cálculos que comparan las profundidades de los árboles que resultan. El factor crítico que influye en estos cálculos es la relación que guarda la longitud del registro completo con la longitud de sólo una llave y el apuntador. Si pueden ponerse muchas parejas llave/apuntador en el área requerida para una sola pareja completa llave/registro, es recomendable la eliminación de la información asociada del árbol B y su colocación en un archivo separado.

9.18

REGISTROS Y LLAVES DE LONGITUD VARIABLE

En muchas aplicaciones, la información asociada con una llave varía en longitud. Los índices secundarios que hacen referencia a listas invertidas son un excelente ejemplo de ello. Una forma de manejar esta variabilidad es colocando la información asociada en un archivo separado de longitud variable; el árbol B contendrá una referencia a la información de este otro archivo. Otro enfoque es permitir un número variable de llaves y registros en una página de árbol B.

Hasta este punto se ha considerado que los árboles B son de algún orden m . Cada página tiene un número fijo máximo y mínimo de llaves

que puede almacenar legalmente. La idea de un registro de longitud variable y, por lo tanto, de un número variable de llaves por página, es una desviación significativa de la perspectiva que se ha desarrollado hasta aquí. Un árbol B con un número variable de llaves por página no tiene claramente un orden fijo único.

La variabilidad en longitud también puede extenderse a las llaves mismas, así como a registros enteros. Por ejemplo, en un archivo en el que los nombres de las personas son las llaves, puede elegirse usar sólo el espacio requerido por un nombre, en lugar de asignar un campo de tamaño fijo para cada llave. Como se estudió en capítulos anteriores, realizar una estructura con campos de longitud variable permite colocar muchos más nombres en una cantidad de espacio determinada, porque elimina la fragmentación interna. Si se colocan más llaves en una página, entonces se tiene un número mayor de descendientes y, probablemente, un árbol con menos niveles.

Permitir esta variabilidad en longitud significa usar diferentes tipos de estructuras de página. En el próximo capítulo se examinan estructuras de página adecuadas para usarse con llaves de longitud variable, cuando se analizan los árboles B*. También se necesita un criterio diferente para decidir cuándo está llena una página y cuándo está en un estado de insuficiencia. En lugar de usar un número máximo y mínimo de llaves por página, se requiere usar un número máximo y mínimo de bytes.

Una vez que se han planteado los mecanismos fundamentales para el manejo de llaves o de registros de longitud variable, surgen nuevas posibilidades interesantes. Por ejemplo, puede considerarse la idea de polarizar el mecanismo de promoción de llaves de manera que se promuevan hacia arriba las llaves (o parejas llave/registro) de longitud variable más corta, en lugar de las llaves más grandes. La idea es que se quiere tener páginas con el mayor número de descendientes en la parte alta del árbol, y no en el nivel de hoja. Ramificar lo más ampliamente posible y lo más alto posible en el árbol tiende a reducir su altura completa. McCreight [1977] analiza esta idea en el artículo "*Pagination of B* Trees with Variable-Length Records*".

El punto principal que se quiere destacar con estos ejemplos de variaciones sobre la estructura de los árboles B es que este capítulo introduce sólo las formas básicas de esta estructura de archivos tan útil y flexible. Las implantaciones reales de los árboles B no siguen perfectamente la forma de los árboles B que aparece en los libros de texto. En lugar de ello, se usan muchas de las otras técnicas de organización que se estudian en este libro, como las estructuras de registros de longitud variable en combinación con la organización fundamental de los árboles B, para hacer nuevas estructuras de archivos de propósito especial, bien adecuadas a los problemas que se presentan.

RESUMEN

Este capítulo comienza considerando el problema que se dejó sin resolver al final del capítulo 7: los índices simples y lineales funcionan bien si se almacenan en memoria RAM electrónica, pero es caro mantenerlos y buscarlos cuando son tan grandes que deben guardarse en almacenamiento secundario. Lo costoso del uso del almacenamiento secundario es más evidente en dos áreas.

- Clasificación del índice, y
- Búsqueda, porque aun la búsqueda binaria requiere de dos o tres accesos al disco.

Primero se estudia la cuestión de estructurar un índice de tal modo que pueda mantenerse en orden sin tener que clasificar. Para ello se usan estructuras de árboles, y se descubre entonces que es necesario un árbol *balanceado* para asegurarse de que el árbol no se vuelva demasiado profundo luego de repetidas inserciones aleatorias. Se observa que los árboles AVL proporcionan una forma de balancear un árbol binario con sólo una pequeña cantidad de gastos adicional.

En seguida se aborda el problema de reducir el número de accesos a disco requeridos para buscar en un árbol. La solución a este problema implica dividir el árbol en páginas, de tal forma que una parte considerable del árbol pueda extraerse con sólo un acceso al disco. Los índices paginados permiten buscar a través de muchas llaves con sólo unos pocos accesos a disco.

Por desgracia, se encuentra que es difícil combinar la idea de *paginación* de estructuras de árboles con la de *balanceo* de estos árboles por el método AVL. La prueba más obvia de esto está asociada con el problema de seleccionar miembros de la página raíz de un árbol o subárbol cuando el árbol se construye en la forma descendente convencional. Esto prepara la escena para introducir el trabajo de Bayer y McCreight sobre árboles B, el cual resuelve el dilema de la paginación y balanceo iniciado desde el nivel de las hojas y promoviendo llaves hacia arriba conforme el árbol crece.

El análisis sobre árboles B comienza con ejemplos de búsqueda, inserción, división y promoción, para mostrar cómo crecen los árboles B mientras se mantiene el balance en una estructura en páginas. En seguida se formaliza la descripción de los árboles B. Esta definición formal permite desarrollar una fórmula para estimar la profundidad del árbol B en el peor caso. La descripción formal también motiva a desarrollo de procedimientos de eliminación que mantienen propiedades de árbol B cuando se eliminan llaves de un árbol.

Una vez que se comprenden la estructura fundamental y los procedimientos de los árboles B, se comienza a refinar y mejorar estas ideas. El primer conjunto de mejoras implica incrementar la utilización del almacenamiento dentro de los árboles B. Por supuesto, el incremento de la utilización del espacio también puede dar como resultado una disminución de la altura del árbol y, por lo mismo, de mejoras en el desempeño. Se encuentra que algunas veces redistribuir llaves durante la inserción, en vez de dividir páginas, puede mejorar la utilización de espacio en los árboles B, de tal manera que sea de alrededor del 85 por ciento. Si se lleva aún más lejos la búsqueda de incrementos de eficiencia en almacenamiento, se encuentra que puede combinarse la redistribución durante la inserción con un tipo diferente de división, para asegurar que alrededor de dos terceras partes y no sólo la mitad de las páginas estén llenas después de una división. Los árboles que usan esta combinación de redistribución y división de dos a tres se llaman *árboles B**.

A continuación se estudia el tema del manejo de páginas en buffers, creando un *árbol B virtual*. Se hace notar que el uso de memoria no es una elección de todo o nada: cuando los índices son demasiado grandes para caber en la memoria no hay necesidad de acceder a ellos por *entero* desde el almacenamiento secundario. Si se almacenan en memoria RAM páginas que probablemente se reutilicen para ahorrarse el costo de releerlas de nuevo del disco. Se desarrollan dos métodos para determinar las páginas del árbol que se van a reutilizar. Un método usa la altura de la página en el árbol para decidir cuáles conviene guardar. Mantener la raíz tiene la más alta prioridad, los descendientes de la raíz tiene la siguiente prioridad, y así sucesivamente. El segundo método para la selección de páginas por guardar en memoria RAM se basa en lo reciente del uso: siempre se reemplaza la página cuyo uso es menos reciente (LRU por sus siglas en inglés), reteniendo las páginas usadas más recientemente. Se observa que es posible combinar estos métodos, y que hacerlo puede permitir encontrar llaves usando un promedio de menos de un acceso a disco por búsqueda.

Posteriormente se pasa a la pregunta de dónde colocar la información asociada con una llave del índice del árbol B. Almacenarla con la llave es atractivo porque, así, encontrar la llave es lo mismo que encontrar la información; no se requieren accesos adicionales al disco. Sin embargo, si la información asociada ocupa mucho espacio, se puede reducir el orden del árbol, incrementando por consiguiente su altura. En tales casos con frecuencia resulta ventajoso almacenar la información asociada en un archivo separado.

Se cierra el capítulo con un breve vistazo al uso de registros de longitud variable en las páginas de un árbol B, haciendo notar que se pueden obtener ahorros significativos en espacio, con la consecuente

reducción de altura del árbol, a partir de la utilización de registros de longitud variable. La modificación de la definición básica del árbol B del libro de texto para incluir el uso de registros de longitud variable es sólo un ejemplo de las muchas variaciones sobre árboles B que se usan en implantaciones reales.



TERMINOS CLAVE

Arboles AVL. Arbol binario balanceado en altura (BA(1)) en el que las inserciones y eliminaciones pueden efectuarse con un mínimo de accesos a nodos locales. Los árboles AVL son interesantes porque evitan que las ramas sean demasiado largas luego de muchas inserciones aleatorias.

Arboles B*. Arbol B especial en el que cada nodo está lleno por lo menos en dos terceras partes. Los árboles B* por lo general proporcionan mejor utilización del almacenamiento que los árboles B.

Arboles balanceados en altura. Estructura de árbol con una propiedad especial: para cada nodo existe un límite en la diferencia que se permite entre las alturas de cualquiera de los subárboles del nodo. Un árbol BA(k) permite que los subárboles estén k niveles fuera de balance. (Véase *árboles AVL*.)

Arboles B de orden m . Arbol de búsqueda múltiple contesta propiedades:

1. Cada nodo tiene un máximo de m descendientes.
2. Cada nodo, excepto la raíz y las hojas, tienen al menos $\lceil m/2 \rceil$ descendientes.
3. La raíz tiene al menos dos descendientes (a menos que sea hoja).
4. Todas las hojas aparecen en el mismo nivel.
5. Una página que no es hoja con k descendientes tiene $k-1$ llaves.
6. Una página hoja contiene al menos $\lceil m/2 \rceil - 1$ y no más de $m-1$ llaves.

Los árboles B se construyen hacia arriba a partir del nivel hoja, de modo que la creación de las páginas nuevas comienzan en el nivel de las hojas.

El poder de los árboles B reside en que están balanceados (no hay ramas demasiado largas); tiene poca profundidad (requieren pocos desplazamientos); permiten eliminaciones e inserciones aleatorias a un costo relativamente bajo mientras se mantiene el balance, y garantizan, al menos, un 50 por ciento de utilización del almacenamiento.

Arboles B virtuales. Índice de un árbol donde se guardan varias páginas en memoria RAM anticipando la posibilidad de un acceso posterior a una o más de ellas. Pueden aplicarse muchas estrategias diferentes para reemplazar páginas en memoria RAM cuando se usan árboles B virtuales, entre ellas las estrategias del uso menos reciente y de la ponderación de altura.

Concatenación. Cuando un nodo de árbol B tiene insuficiencia (está lleno a menos del 50 por ciento), algunas veces resulta necesario combinarlo con un nodo adyacente, disminuyendo así el número total de nodos del árbol. Puesto que la concatenación implica un cambio en el número de nodos del árbol, sus efectos pueden requerir reorganización en muchos niveles del árbol.

División. Creación de dos nodos a partir de uno, debido a que el nodo original se saturó. La división crea la necesidad de promover una llave a un nodo de nivel superior para tener un índice que separe los dos nodos nuevos.

Hoja de un árbol B. Página en el nivel más bajo en un árbol B. Todas las hojas de un árbol B están en el mismo nivel.

Índice paginado. Índice dividido en bloques, o páginas, cada una de las cuales almacena muchas llaves. El uso de índices paginados permite entre muchas llaves con sólo unos cuantos accesos a disco.

Orden de un árbol B. Número máximo de descendientes que puede tener un nodo en el árbol.

Promoción de una llave. El traslado de una llave de un nodo a otro nivel más alto (creándose el nodo de nivel más alto si es necesario), cuando el nodo original se satura y debe dividirse.

Redistribución. Cuando un nodo del árbol B cae en insuficiencia (está lleno a menos del 50 por ciento), puede ser posible trasladarle llaves desde un nodo adyacente con el mismo padre. Esto ayuda a asegurar que se mantenga la propiedad del 50 por ciento cubierto. Cuando las llaves se redistribuyen, es necesario también alterar el contenido del padre. La redistribución, a diferencia de la concatenación, no implica la creación o eliminación de nodos; sus efectos son locales por completo. A menudo, la redistribución también puede usarse como alternativa a la división.

EJERCICIOS

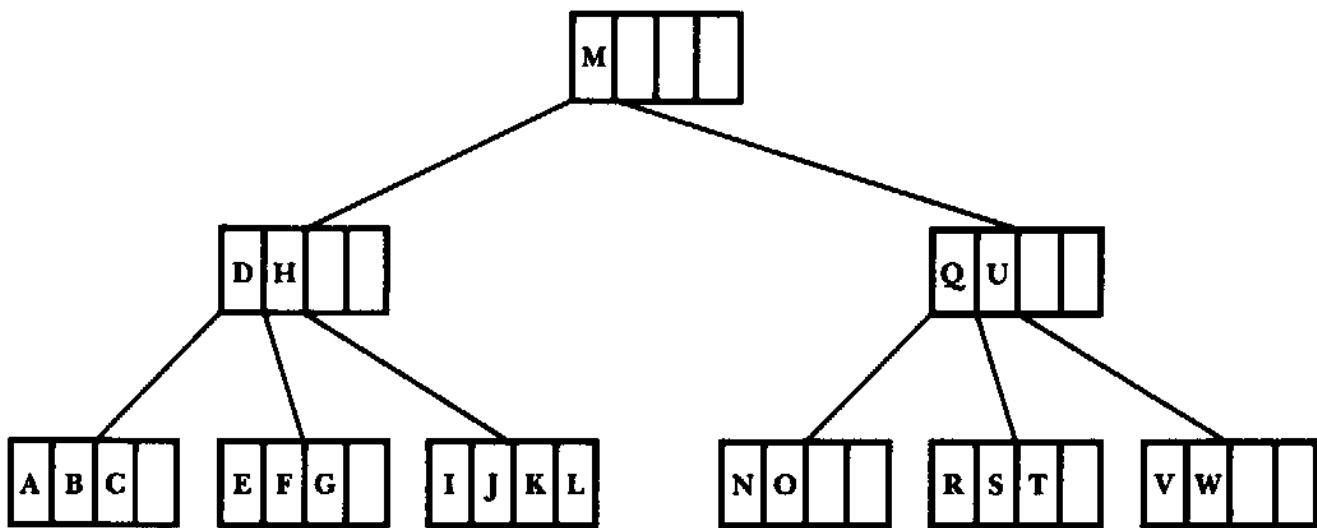
1. Los árboles binarios balanceados pueden ser estructuras de índices efectivas para la indización basada en memoria RAM, pero tienen varios inconvenientes cuando se vuelven tan grandes que parte de ellas o todas, deben guardarse en almacenamiento secundario. Las siguientes preguntas deben ayudar a exponer estos inconvenientes, y de esta forma reforzar la necesidad de una estructura alterna, como los árboles B.

- a) Existen dos grandes problemas con el uso de la búsqueda binaria en un índice simple clasificado en almacenamiento secundario: el número de acceso al disco es mayor que lo deseable, y es considerable el tiempo que toma mantener clasificado el índice. ¿Cuál de esos problemas se resuelve con un árbol binario de búsqueda?
- b) ¿Por qué es importante mantener balanceados los árboles de búsqueda?
- c) ¿En qué forma es un árbol AVL mejor que un árbol binario simple de búsqueda?
- d) Suponga que se tiene con 1 000 000 de llaves almacenado en disco en un árbol binario de búsqueda completamente lleno. Si el árbol no está paginado, ¿cuál es el número máximo de accesos requeridos para encontrar una llave? Si el árbol está paginado en la forma ilustrada en la figura 9.12, pero con cada página es capaz de almacenar 15 llaves y ramificarse a 16 páginas nuevas, ¿cuál es el máximo número de accesos requeridos para encontrar una llave? Si el tamaño de la página se incrementa para almacenar 511 llaves con ramificaciones a 512 nodos, ¿cómo cambia el máximo número de accesos?
- e) considere el problema de balancear el árbol de tres llaves por página de la figura 9.13, reacomodando las páginas. ¿Por qué es difícil crear un algoritmo de balanceo del árbol que tenga sólo efectos locales? Cuando el tamaño de la página se incrementa a un tamaño más adecuado (como 512 llaves), ¿por qué resulta difícil garantizar que cada una de las páginas contenga al menos algún número mínimo de llaves?
- f) Explique la siguiente afirmación: los árboles B se construyen hacia arriba a partir de la base, mientras que los árboles binarios se construyen hacia abajo a partir de la raíz.
- g) Aunque los árboles B por lo general se consideran superiores a los árboles binarios de búsqueda para la búsqueda externa, los árboles binarios son comúnmente usados para búsqueda interna. ¿Por qué?

2. Describa las partes *necesarias* de un nodo hoja de un árbol B. ¿En qué difiere un nodo hoja de un nodo interno?
3. Como los nodos hoja nunca tiene hijos, es posible usar los campos de apuntadores en un nodo hoja para que apunten a registros de datos. Esto podría eliminar la necesidad de campos de apuntadores a los registros de datos en los nodos internos. ¿Por qué? ¿Qué implicaciones tiene esto en términos de utilización del almacenamiento y tiempo de extracción de datos?
4. Muestre los árboles B de orden cuatro que resultan de cargar los siguientes conjuntos de llaves en orden.
- C G J X
 - C G J X N S U O A E B H I
 - C G J X N S U O A E B H I F
 - C G J X N S U O A E B H I F K L Q R T V U W Z
5. La figura 9.23 muestra el patrón de llamadas recursivas necesarias para insertar un \$ en el árbol B de la figura 9.22. Suponga que tras esta inserción, el carácter [se inserta *después* de la Z. (El código ASCII para [es mayor que el código ASCII para la Z.) Dibuje una figura similar a la figura 9.23, que muestre el patrón de llamadas recursivas requerido para efectuar esta inserción.
6. Dado un árbol B de orden 256 que contiene 100 000 llaves,
- ¿cuál es el número máximo de descendientes de un página?
 - ¿cuál es el número mínimo de descendiente de una página (sin contar la raíz y las hojas)?
 - ¿cuál es el número mínimo de descendientes de la raíz?
 - ¿cuál es el número mínimo de descendientes de una hoja?
 - ¿cuántas llaves existen en una página que no es hoja con 200 descendientes?
 - ¿cuál es la máxima profundidad del árbol?
7. Utilizando un método parecido al usado para derivar la fórmula de la profundidad en el peor caso, derive una fórmula para el mejor caso, o profundidad mínima, para un árbol B de orden m con N llaves. ¿Cuál es la profundidad mínima del árbol descrito en la pregunta anterior?
8. Suponga que se tiene un índice en árbol B para un archivo sin clasificar, donde cada llave tiene almacenado el NRR del registro correspondiente. La profundidad del árbol B es d . ¿Cuáles son los números máximos y mínimos de acceso a disco requeridos para
- extraer un registro;
 - agregar un registro;
 - eliminar un registro, y
 - extraer todos los registros del archivo en el orden de clasificación.

Suponga que *no* se usa el manejo de páginas en buffers. En cada caso, indique cómo llegó a su respuesta.

9. Muestre los árboles que resultan después de que se elimina cada una de las llaves *A*, *B*, *Q* y *R* del siguiente árbol B de orden cinco.



10. Suponga que se desea eliminar una llave de un nodo en un árbol B. Se examina el hermano derecho y se encuentra que la redistribución no funciona; sería necesaria la concatenación. Se examina a la izquierda y se ve aquí la redistribución sí es una opción. ¿Se elige concatenar o redistribuir?

11. ¿Cuál es la diferencia entre un árbol B* y un árbol B? ¿Qué mejoras ofrece un árbol B* sobre un árbol B, y qué complicaciones introduce? ¿Cómo se compara la profundidad mínima de un árbol B* de orden *m* con la de un árbol B de orden *m*?

12. ¿Qué es un árbol B virtual? ¿Cómo es posible que el número promedio de accesos por llave sea menor de uno, cuando se extraen llaves de un árbol B virtual de tres niveles? Escriba una descripción en pseudocódigo para un esquema de reemplazo LRU para un buffer de diez páginas usado en la realización de un árbol B virtual.

13. Analice los compromisos entre almacenar la información indizada por las llaves en un árbol B con la llave misma y el almacenar la información en un archivo separado.

14. Se observó que, considerando llaves de longitud variable, es posible optimizar un árbol incluyendo una promoción polarizada hacia llaves más cortas. Con árboles de orden fijo se promueve la llave de enmedio. En un árbol de orden variable y de llaves de longitud variable, ¿cuál es el significado de "la llave de enmedio"? ¿Cuáles son los compromisos asociados con la inclusión de preferencias por llaves más cortas en esta selección de una llave para su promoción? Bosqueje una realización de este proceso de selección y promoción.

EJERCICIOS DE PROGRAMACION

15. Haga los programas al final del capítulo y agregue un procedimiento recursivo que efectúe un recorrido simétrico con paréntesis de un árbol B creado por el programa. Como ejemplo, aquí está el resultado de un recorrido con paréntesis del árbol mostrado en la figura 9.18:

((A,B,C)D(E,F,G)H(I,J)K(L,M))N((O,P)Q(R)S(T,U,V)W(X,Y,Z)))

16. La rutina *división()* en los programas de árboles B no es muy eficiente. Reescribala para hacerla más eficiente.

17. Escriba un programa que busque una llave en un árbol B.

18. Escriba un programa interactivo que permita a un usuario encontrar, insertar y eliminar llaves de un árbol B.

19. Escriba un programa que use llaves que sean cadenas en lugar de caracteres simples.

20. Escriba un programa que construya un índice en árbol B para un archivo de datos en el cual los registros contienen más información que sólo una llave.



LECTURAS ADICIONALES

Los libros de texto actuales sobre archivos y estructuras de datos contiene, sorpresivamente, breves estudios sobre árboles B. Estos estudios, en general, no aportan más información de la que se presenta en este capítulo y el siguiente, por lo que los lectores interesados en adquirir mayor información acerca de árboles B deberán consultar los artículos publicados en revistas durante los pasados 15 años.

El artículo que introdujo los árboles B al mundo es "*Organization and Maintenance of Large Ordered Indexes*", de Bayer y McCreight [1972]. Describe las propiedades teóricas de los árboles B e incluye resultados empíricos que se refieren, entre otras cosas, al efecto de usar redistribución además de la división durante la inserción. Los lectores deben tomar en cuenta que la notación y terminología empleadas en ese artículo difieren de las que usan en este texto en varios aspectos importantes.

En el artículo de presentación de Comer [1979], "*The Ubiquitous B-Tree*", se proporciona un excelente panorama de algunas variaciones importantes sobre la forma básica de los árboles B. El estudio de Knuth [1973b] sobre árboles B, aunque breve, es un recurso importante, en parte debido a que muchas de las formas variantes, tales como los árboles B*, se agruparon por primera vez en su estudio. McCreight [1977] examina en forma específica las operaciones

sobre árboles que usan registros de longitud variable y que, por lo tanto, son de orden variable. Aunque dicho artículo se refiere específicamente a los árboles B*, la consideración de registros de longitud variable puede aplicarse a muchas otras formas de árboles B. En "Time and Space Optimality on B-Trees," Rosenberg y Snyder [1981] analizan los efectos de la asignación de valor inicial de árboles B con el mínimo número de nodos. En "Analysis of Design Alternatives for Virtual Memory Indexes," Murayama y Smith [1977] examinan tres factores que afectan el costo de la extracción de datos: elegir una estrategia de búsqueda, si las páginas del índice están o no estructuradas, y si se comprimen o no las llaves. Zoellick [1986] analiza el uso de estructuras semejantes a los árboles B en discos ópticos.

Puesto que los árboles B en sus diversas formas han llegado a ser una organización estándar de archivos para bases de datos, puede encontrarse bastante material interesante sobre aplicaciones de árboles B en la literatura de bases de datos. Ullman [1982], Held y Stonebraker [1978] y Snyder [1978] analizan el uso genérico de árboles B en sistemas de bases de datos. Ullman [1982] trata el problema del manejo de aplicaciones en las que varios programas tiene acceso en forma concurrente a la misma base de datos, e identifica la literatura que aborda el acceso concurrente a los árboles B.

Los usos de los árboles B para accesos por llaves secundarias se analizan en muchas de las referencias citadas anteriormente. También existe una creciente literatura sobre índices dinámicos multimensionales, entre ella una estructura parecida a un árbol B llamada árbol B *k-d*. Los árboles B *k-d* se describen en trabajos elaborados por Ouskel y Scheuermann [1981] y Robinson [1981]. Otros métodos de indización secundaria incluyen el uso de tries y archivos grid. Los tries se estudian en muchos textos sobre archivos y estructuras de datos, entre ellos Knuth [1973b], Claybrook [1983] y Loomis [1983]. Los archivos grid se analizan a fondo en Nieverglit *et al* [1984].

Un interesante artículo, de los primeros que se publicaron sobre el uso de estructuras dinámicas de árboles para el procesamiento de archivos, es "The Use of Tree Structures for Processing Files", elaborado por Sussenguth [1963]. Wagner [1973] y Keehn y Lacy [1974] examinan las consideraciones de diseño de índices que llevan al desarrollo de VSAM. VSAM emplea una estructura de índices muy parecida a un árbol B pero, según parece, fue desarrollada en forma independiente del trabajo de Bayer y McCreight. Los lectores interesados en aprender más acerca de los árboles AVL encontrarán una análisis bueno y accesible de los algoritmos asociados con estos árboles en Standish [1980]. Knuth [1973b] da un enfoque más riguroso y matemático a las operaciones y propiedades de los árboles AVL.

PROGRAMA EN C PARA INSERTAR LLAVES EN UN ARBOL B

El programa C que se muestra a continuación realiza el programa de inserción descrito en el texto. La única diferencia entre este programa y el del texto es que éste construye un árbol B de orden cinco, mientras que el del texto construye uno de orden cuatro. Los caracteres de entrada se toman de la entrada estándar, indicando con *q* el fin de los datos.

El programa requiere que se usen funciones de varios archivos:

<i>manejador.C</i>	Contiene el programa principal, muy parecido al programa <i>manejador</i> descrito en el texto.
<i>inserta.C</i>	Contiene <i>inserta()</i> , la función recursiva que encuentra el lugar apropiado para una llave, la inserta, y supervisa la división y las promociones.
<i>abes.C</i>	Contiene todas las funciones de apoyo que efectúan directamente la E/S. Los archivos de encabezado <i>arches.h</i> y <i>stdio.h</i> deben estar disponibles para su inclusión en <i>abes.c</i> .
<i>utilab.C</i>	Contiene el resto de las funciones de apoyo, incluyendo la función <i>divide()</i> descrita en el texto.

Todos los programas incluyen el archivo de encabezado llamado *ab.h*

```

/* ab.h
   Archivo de encabezado para programas de árboles B
*/

#define MAXLLAVES 4
#define MINLLAVES MAXLLAVES/2
#define NULO (-1)
#define SINLLAVE    '@'

typedef struct {
    short contllave:           /* número de llaves en la página */
    char llave [MAXLLAVES]:    /* las llaves */
    short hiyo [MAXLLAVES + 1]: /* apunadores a los NNR de los */
} PAGINAAB;                  /* descendientes */

#define TAMPAGINA sizeof (PAGINAAB)

```

===== MANEJADOR.C

```
/* manejador.c...
   Manejador para pruebas de árboles B:

   Abre o crea el archivo del árbol B.
   Obtiene la siguiente llave y llama a inserta para insertar
   la llave en el árbol.
   Si es necesario, crea una nueva raíz.

*/
#include "stdio.h"
#include "ab.h"
main ()
{
    int promovido; /* booleano: indica si hay una promoción */
    short raiz, /* NRR de la raiz */
          nrr_plomo; /* NRR que se promueve */
    char llave_plomo; /* llave que se promueve */
    llave; /* siguiente llave por insertar en el árbol */

    if (abreab()) /* intenta abrir árbol.dat y obtener la raíz*/
        raiz = tomaraiz ();
    else /*si arbolb. dat no existe lo crea */
        raiz = crea-árbol();

    while ((llave = getchar ()) != 'q') {
        promovido = inserta (raiz, llave, &nrr_plomo, &llave_promo);
        if (promovido)
            raiz = crea_raiz (llave_plomo, raiz, nrr_promo);
    }
    cierraab ();
}
```

===== INSERTA.C

```
/* inserta.c...
   Contiene inserta () -- función para insertar una llave dentro del árbol B.;

   Se llama a sí misma hasta alcanzar la base del árbol
   Entonces inserta la llave en el nodo
   Si el nodo está lleno,
       - llama a divide () para dividir el nodo
       - promueve la llave de en medio y el NRR del nodo nuevo

*/
#include "ab.h"

inserta (nrr, llave, hijo_d_plomo, llave_plomo)
short nrr /* NRR de página en donde se va insertar */
         *hijo_d_plomo /* hijo promovido al siguiente nivel */
char llave /* llave por insertar aquí o más abajo */
         * llave_plomo /* llave promovida al siguiente nivel */
```

(continúa)

```

{
    PAGINAAB página,          /* página actual */ 
    pagnueva;                /* página nueva creada si hubo */
                            /* división */ 
    int encontró;            /* valores booleanos */ 
    shor poss, 
    nrr_p_a;                 /* NRR promovido desde abajo */ 
    char llave_p_a;          /* llave promovida desde abajo */ 

    if (nrr == NULO) {        /* se pasó del fondo del Árbol "promoción" */ 
        * llave_plomo = llave; /* de la llave original de modo que */ 
        * hijo_d_plomo = NULO; /* será insertada en el nivel hoja */ 
        return (SI); 
    } 
    leeab(nrr, &página) ; 
    encontró = busca_nodo (llave, &página, &pos) ; 
    if (encontró) { 
        printf ("Error: intento de inserción de llave duplicada: %c \n\007", 
            llave); 
        return (0) ; 
    } 

    promoción = inserta (página.hijo[pos], llave, &nrr_p_a, &llave_p_a); 
    if (!promoción) 
        return (NO); /* no hay promoción */ 
    if (página.contillave < MAXLLAVES) { 
        ins_en_pag (llave_p_a, nrr_p_a, &página);/* listo para insertar */ 
        escribeab(nrr, &página);           /* la llave y el apuntador */ 
                                            /* en esta página */ 
        return (NO) ; /* no hay promoción */ 
    } 
    else { 
        divide(llave_p_a, nrr_p_a, &página, llave_promo, hijo_d_promo, &pagnueva); 
        escribeab(nrr, &página); 
        escribeab(*hijo_d_promo, &pagnueva); 
        return (SI) ; /* hubo promoción */ 
    } 
}
}

```

ABES.C

```

/* abes.c 
   Contiene funciones de árboles B que implican directamente E/S de archivos: 

abreab() -- abre el archivo "arbolb.dat" para almacenar el árbol 
cierraab() -- cierra "arbolb.dat" 
tomaraíz() -- toma el nrr del nodo raíz de los dos primeros bytes de arbold.dat 
colocaraíz() -- coloca el nrr del nodo raíz en los dos primeros bytes de arbolb.dat 
crea-árbol() -- crea "arbolb.dat" y el nodo raíz 
tomapag() -- toma el siguiente bloque disponible de "arbolb.dat" para una página 
nueva

```

```
    leeab() -- lee la página número NRR de "arbolb.dat"
    escribeab() -- escribe la página número NRR en "arbolb"
/*
#include <stdio.h>
#include "ab.h"
#include "arches.h"

int daab; /* descriptor de archivo global para "arbolb.dat" */

abreab()
{
    daab = open ("arbolb.dat", READWRITE);
    return(daab > 0);
}

cierraab()
{
    close(daab);
}

short tomaraiz()
{
    short raiz;
    long lseek();

    lseek(daab, 0L, 0);
    if (read (daab, &raiz, 2) == 0) {
        printf ("Error: No se pudo obtener la raiz. \007\n");
        exit(1);

    }
    return (raiz) ;
}

colocaraiz (raiz)
short raiz;
{
    long lseek();
    lseek(daab, 0L, 0);
    write (daab, &raiz, 2);

}

short crea_árbol()
{
    char llave;

    daab = creat("arbolb.dat", PMODE) ;
    close(daab) ; /* En muchos sistemas se tiene que cerrar y reabrir */
                    /* (continuación)
```

```

abreab() ;           /* para asegurar el acceso de lectura y escritura. */
llave = getchar() ; /* Toma la primera llave */
return (crea_raiz(llave, NULO, NULO));
}

short tomapag()
{
    long lseek(), dir;
    dir = lseek(daab, 0L, 2) -2L;
    return ((short) dir / TAMPAGINA);
}

leeab(nrr, apunt_página)
short nrr;
PAGINaab *apunt_página;
{
    long lseek(), dir;
    dir = (long)nrr * (long)TAMPAGINA + 2L;
    lseek(daab, dir, 0);
    return(read(daab, apunt_página, TAMPAGINA));
}

escribreab (nrr, apunt_página)
short nrr;
PAGINaab *apunt_página;
{
    long lseek(), dir;
    dir = (long) nrr * (long) TAMPAGINA + 2L;
    lseek(daab, dir, 0);
    return(write(daab, apunt_página, TAMPAGINA));
}

```

===== UTILAB.C

```

/* utilab.c
   Contiene funciones de utilleria para programas de árboles B:

  crea_raiz() -- Toma y asigna valores iniciales al nodo raíz e inserta una llave
  iniciapag() -- Coloca el valor SINLLAVE en todos los lugares de las llaves y
                 NULO en los lugares de los hijos.
  busca_nodo() -- Devuelve SI si la llave está en el nodo, en caso contrario, NO.
                  En cualquier caso, coloca la posición correcta de la llave en
                  pos.
  ins_en_pág() -- Inserta la llave y el hijo derecho en la página.
  divide() -- Divide el nodo creando uno nuevo y moviendo la mitad de las llaves
             al nuevo nodo. Promueve la llave y el NRR de enmedio del nodo
             nuevo.
*/
#include "ab.h"

```

```

crea_raíz(llave, izq, der)
char llave;
short izq, der;
{
    PAGINAAB página;
    short nrr;
    nrr = tomapag();
    iniciapag(&página);
    página.llave[0] = llave;
    página.hijo[0] = izq;
    página.hijo[1] = der;
    página.contllave = 1;
    escribeab(nrr, &página);
    colocaraíz(nrr) ;
    return(nrr) ;
}

iniciapag(a_página)
PAGINAAB *a_página; /* a_página: apuntador a una página */
{
    int j;

    for (j = 0; j < MAXLLAVES; j++) {
        a_página-> llave[j] = SINLLAVE;
        a_página-> hijo[j] = NULO;
    }
    a_página -> hijo[MAXLLAVES] = NULO;
}

busca_nodo (llave, a_página, pos)
char llave;
PAGINAAB *a_página;
short *pos; /* posición en donde la llave está o debe insertarse */
{
    int i ;
    for (i = 0; i < a_página-> contllave && llave > a_página-> llave[i]; i++ )
        ;
    *pos = i;
    if( *pos < a_página->contllave && llave == a_página->llave[*pos] )
        return (SI); /* la llave está en la página */
    else
        return (NO); /* la llave no está en la página */
}

ins_en_pág (llave, hijo_d, a_página)
char llave;
short hijo_d;
PAGINAAB *a_página;
{
    int i;
    for (i = a_página -> contllave; llave < a_página->llave[i-1] && i > 0; i--) {
        a_página->llave[i] = a_página->llave[i-1];
        a_página->hijo[i+1] = a_página->hijo[i];
    }
}

```

(continúa)

```

    }
    a_página->contllave++;
    a_página->llave[i] = llave;
    a_página->hijo[i+1] = hijo_d;
}

divide (llave, hijo_d, a_pagant, llave_promo, hijo_d_promo, a_pagnue)
char llave,          /* llave por insertar */
    *llave_promo; /* llave a promover hacia arriba desde aquí */
short hijo_d,        /* NRR del hijo por insertar */

    *hijo_d_promo; /* NRR a promover hacia arriba desde aquí */
PAGINaab *a_pagant, /* apuntadores a las estructuras de las páginas */
    *a_pagnue;      /* nueva y anterior */

{
    int i;
    short mitad;           /* indica dónde debe ocurrir la división */
    char llavesaux[MAXLLAVES+1]; /* almacena temporalmente las llaves, */
                                /* antes de dividir */
    short caraux[MAXLLAVES+2]; /* almacena temporalmente los hijos, */
                                /* antes de dividir */

    for (i=0; i < MAXLLAVES; i++) {           /* mueve llaves e hijos de */
        llavesaux[i] = a_pagant->llave[i]; /* la página anterior a */
        caraux[i] = a_pagant->hijo[i];   /* los arreglos de trabajo */
    }
    llavesaux[i] = a_pagant->hijo[i];
    for (i=MAXLLAVES; llave < llavesaux[i-1] && i > 0; i--)/*{ inserta la */
        llavesaux[i] = llavesaux[i-1];           /* llave nueva */
        caraux[i+1] = caraux[i];
    }
    llavesaux[i] = llave;
    caraux[i+1] = hijo_d;

    *hijo_d_promo = tomapag(); /* crea la página nueva para la */
    iniciapag(a_pagnue);/* división y promueve el NRR de la página nueva */

    for (i = 0; i < MINLLAVES; i++) { /* mueve la primera mitad de las */
        a_pagant->llave[i] = llavesaux[i]; /* llaves e hijos a la página*/
        a_pagant->hijo[i] = caraux[i];   /* anterior y la segunda a la*/
        a_pagnue->llave[i] = llavesaux[i+1+MINLLAVES]; /* página nueva */
        a_pagnue->hijo[i] = caraux[i+1+MINLLAVES]; /* marca la segunda */
        a_pagant->llave[i+MINLLAVES] = SINLLAVE; /* mitad como vacía */
        a_pagant->hijo[i+1+MINLLAVES] = NULO;
    }

    a_pagant->hijo[MINLLAVES] = caraux[MINLLAVES];
    a_pagnue->hijo[MINLLAVES] = caraux[i+1+MINLLAVES];
    a_pagnue->contllave = MAXLLAVES - MINLLAVES;
    a_pagant->contllave = MINLLAVES;

    *llave_promo = llavesaux[MINLLAVES]; /* promueve la llave de enmedio */
}

```

PROGRAMA EN PASCAL PARA INSERTAR LLAVES EN UN ARBOL B

El programa en Pascal que se muestra a continuación realiza el programa de inserción descrito en el texto. La única diferencia entre este programa y el del texto es que éste construye un árbol B de orden cinco, mientras que el del texto construye uno de orden cuatro. Los caracteres de entrada se toman de la entrada estándar, indicando con q el fin de los datos.

El programa principal incluye tres instrucciones no estándar para el compilador:

```
{$B-}
{$I utilab.prc}
{$I Inserta.prc}
```

\$B- indica al compilador de Turbo Pascal que maneje la entrada del teclado como un archivo estándar de Pascal.

La instrucción \$I indica al compilador que incluye los archivos *utilab.prc* e *inserta.prc* en el programa principal. Estos dos archivos contienen funciones que necesita el programa principal. De esta forma el programa del árbol B requiere que se usen funciones de tres archivos:

<i>manejador.pas</i>	Contiene el programa principal, muy parecido al programa manejador descrito en el texto.
<i>inserta.prc</i>	Contiene <i>inserta()</i> , la función recursiva que encuentra el lugar apropiado para una llave, la inserta, y supervisa la división y las promociones.
<i>utilab.prc</i>	Contiene el resto de las funciones de apoyo, incluyendo la función <i>divide()</i> descrita en el texto.

===== MANEJADOR.PAS

```
PROGRAM arbolb (INPUT, OUTPUT);
{
```

Manejador para pruebas de árboles B:
 Abre o crea el archivo del árbol B.
 Obtiene la siguiente llave y llama a inserta

```

        para insertar la llave en el árbol.
        Si es necesario, crea una nueva raíz.
    }

{ $B-}

CONST
  MAXLLAVES = 4;          {máximo número de llaves en la página}
  MAXHIJOS = 5;           {máximo número de hijos en la página}
  MAXLLAVESAUX = 5;       {máximo número de llaves en el espacio auxiliar}
  MAXHIJOSAUX = 6;         {máximo número de hijos en el espacio auxiliar }
  SINLLAVE = '@';         {símbolo que indica que no hay llaves}
  NO = FALSE;
  SI = TRUE;
  NULO = -1;

TYPE
  PAGINAAB = RECORD
    contllave : integer; {número de llaves en la página}
    llave : array [1..MAXLLAVES] of char; {las llaves}
    hijo : array [1..MAXHIJOS] of integer; {apuntadores a los NRR de los
                                             descendientes}
  END ;
VAR
  promovido : boolean ;      {indica si se hay una promoción }
  raíz,                      {NRR de la raíz }
  nrr_promo : integer;        {NRR promovido desde abajo }
  llave_promo                 {llave promovida desde abajo }
  llave: char;                {siguiente llave por insertar en el árbol }
  daab : file of PAGINAAB;   {descriptor de archivo global }
                                {Para "arbolab.dat" }
  MINLLAVES : integer;        {min. número de llaves en una página }
  TAMPAGINA : integer;        {tamaño de una página }
}

{$I utilab.prc}
{$I insert.prc}

BEGIN (principal)
  MINLLAVES := MAXLLAVES DIV 2;
  TAMPAGINA := sizeof(PAGINAAB);

  if abreab then      {intenta abrir arbolab.dat, y tomar la raíz}
    raíz := tomaraíz
  else                  {si no existe arbolab.dat, lo crea}
    raíz := crea_árbol;

  read(llave);
  WHILE (llave < > 'q') DO

    BEGIN
      promovido := inserta(raíz, llave, nrr_promo, llave_promo);
      if promovido then
        raíz := crea_raíz (llave_promo, raíz, nrr_promo);
      read(llave)
    END;

    cierraab

```

===== INSERTA.PRC

```

FUNCTION inserta (nrr: integer; llave: char; VAR hijo_d_promo: integer;
                 VAR llave_promo: char): boolean;

{ Función que inserta una llave en un árbol B:

  Se llama a si misma hasta alcanzar la base del árbol.
  Entonces inserta la llave en el nodo.
  Si el nodo está lleno, llama a divide() para dividir el nodo
  Promueve la llave de en medio y el NRR del nodo nuevo.
}

VAR

  página,           {página actual
  paginanue : PAGINaab; {página nueva creada si hubo división
  encontró,        {indica si la llave ya está en el árbol B
  promoción : boolean ; {indica si la llave se promovió
  pos,             {posición en la que debe estar la llave
  nrr_p_a : integer; {NRR promovido desde abajo
  Llave_p_a : char; {llave promovida desde abajo

BEGIN
  if (nrr = NULO) then      {se pasó del fondo del árbol... "promoción"
    BEGIN                   {de la llave original, de modo que será
      llave_promo := llave; {insertada en el nivel hoja
      hijo_d_promo := NULO;
      inserta := SI
    END
  }
  else
    BEGIN
      leeab(nrr,página);
      encontró := busca_nodo (llave,página,pos);
      if (encontró) then
        BEGIN
          writeln('Error: intento de inserción de una llave duplicada: ',
                  llave);
          inserta := NO
        END
      else      {inserta la llave en el nivel inferior }
        BEGIN
          promoción := inserta(página.hijo[pos],llave,nrr_p_a, llave_p_a) ;
          if (NOT promoción) then
            inserta := NO { no hay promoción }
          else
            BEGIN
              if (página.contllave < MAXLLAVES) then
                BEGIN
                  ins_en_pag(llave_p_a,nrr_p_a,página); {listo para insertar}
                  escribeab(nrr,página); {la llave y el     }
                  inserta := NO           {apuntador en esta   }
                END                     {página no hay     }
              else
                BEGIN
                  inserta := NO           {promoción       }
                END
            END
        END
    END
  }

```

(continúa)

```

        BEGIN
        divide(llave_p_a,nrr_p_a,página,llave_promo,
               hijo_d_promo,paginanue);
        escribeab(nrr,página);
        escribeab(hijo_d_promo,paginanue);
        inserta := SI { hubo promoción }
        END
    END
END;

```

===== UTILAB.PRC

```

FUNCTION abreab : BOOLEAN;
{Función que abre "arbolb.dat" si existe. En caso contrario devuelve falso}

```

```

VAR
    respuesta : char;
BEGIN
    assign(daab, 'arbolb.dat');
    write('¿ya existe arbolb.dat? (responda S o N): ');
    readln(respuesta) ;
    writeln;
    if (respuesta = 'S') or (respuesta = 's') then
        BEGIN
        reset(daab);
        abreab := TRUE
        END
    else
        abreab := FALSE
END;

```

```

PROCEDURE cierraab;
{procedimiento que cierra "arbolb.dat"}

```

```

BEGIN
    Close (daab);
END;

```

```

FUNCTION tomaraiz : integer;
{ Función que obtiene el NRR del nodo raíz del primer registro de arbolb.dat}

```

```

VAR
    raiz: PAGINAAB;
BEGIN
    seek(daab, 0) ;
    if (not EOF) then
        BEGIN
        read(daab,raiz) ;
        tomaraiz := raiz.contllave
        END
END;

```

```

    else
        writeln('Error: no se pudo obtener la raíz.')
END;

FUNCTION tomapag : integer;
{Función que obtiene el siguiente bloque disponible en "arbolb.dat" para una página
nueva}
BEGIN
    tomapag := filesize(daab)
END;

PROCEDURE iniciapag (VAR a_página : PAGINAAB);
{coloca SINLLAVE en todos los lugares de las llaves y NULO en los de los hijos}

VAR
    j : integer;
BEGIN
    for j := 1 to MAXLLAVES DO
        GEBIN
        a_página.llave[j] := SINLLAVE;
        a_página.hijo[j] := NULO;
    END;
    a_página.hijo[MAXLLAVES+1] := NULO
END;

PROCEDURE colocaraíz (raíz: integer);
{Coloca el NRR del nodo raíz en el contador de llave del primer registro de arbolb.dat}
VAR
    nrrraíz : PAGINAAB;
BEGIN
    seek(daab, 0);
    nrrraíz.contllave := raíz;
    iniciapag (nrrraíz);
    write(daab,nrrraíz)
END;

PROCEDURE leeab (nrr : integer ; VAR a_página : PAGINAAB);
{lee la página número NRR de arbolb.dat}

BEGIN
    seek (daab,nrr);
    read(daab,a_página);
END;

PROCEDURE escribeab (nrr : integer; a_página : PAGINAAB);
{escribe la página número NRR en arbolb.dat}
BEGIN
    seek(daab,nrr) ;
    write(daab,a_página);
END;

FUNCTION crea_raíz (llave : char; izquierdo,derecho : integer): integer;
{obtiene y asigna valores iniciales al nodo raíz e inserta una llave}
VAR
    página : PAGINAAB;

```

```

nrr : integer;
BEGIN
  nrr := tomapag;
  iniciapag(página);
  página.llave[1] := llave;
  página.hijo[1] := izquierdo;
  página.hijo[2] := derecho;
  página.contllave := 1;
  escribeab(nrr,página);
  colocaraíz(nrr);
  crea_raiz := nrr
END ;

FUNCTION crea_árbol : integer;
{crea "arbolb.dat" y el nodo raíz}
VAR
  nrrraíz : integer;
BEGIN
  rewrite(daab);
  read (llave);
  Nrrraíz := tomapag;
  colocaraíz (nrrraíz);
  crea_árbol:= crea_raíz(llave, NULO, NULO);
END;

FUNCTION busca_nodo(llave : char ; a_página : PAGINAAB; VAR pos : integer) : boolean;
{devuelve SI si la llave está en el nodo, en caso contrario, NO. En cualquier caso,
coloca la posición correcta de la llave en pos)
VAR
  i : integer;
BEGIN
  i := 1;
  While ( (i <= a_página.contllave) AND (llave > a_página.llave[1])) DO
    i := i + 1;
    pos :=i;
  if (( pos < * a_página.contllave) AND (llave = a_página.llave [pos])) then
    busca_nodo := SI
  else
    busca_nodo := NO
END;

PROCEDURE ins_en_pág (llave : char; hijo_d : integer ; VAR a_página: AGINAAB);
{inserta la llave y el hijo derecho en la página)
VAR
  i : integer;
BEGIN
  i := a_página.contllave + 1;
  while ((llave < a_página.llave[i-1]) AND (i > 1)) DO
    BEGIN
      a_página.llave [i] := a_página.llave[i-1];
      a_página.hijo[i+1] := a_página.hijo[i];
      i := i - 1
    END;

```

```

a_página.contllave := a_página, contllave + 1;
a_página.llave[i] := llave;
a_página.hijo[i+1] := hijo_d
END;

PROCEDURE divide (llave : char; hijo_d : integer; VAR a_pagant : PAGINAAB;
                  VAR llave_promo : char; Var hijo_d_promo : integer;
                  VAR a_pagnue : PAGINAAB);

{divide el nodo creando uno nuevo y moviendo la mitad de las llaves al nodo nuevo.
Promueve la llave y el NRR de enmedio del nodo nuevo. }

VAR
    i : integer;
    llavesaux : array [1..MAXLLAVESAUX] of char; {almacena temporalmente }
                                                {las llaves, antes de   }
    caraux : array [1.. MAXHIJOSAUX] of integer; {dividir almacena tempo- }
                                                {ralmente los hijos, antes  }
                                                {de dividir               }

BEGIN
    for i := 1 TO MAXLLAVES DO {mueve llaves e hijos de la página}
        BEGIN{anterior a los arreglos de trabajo}
            llavesaux[i] := a_pagant.llave[i];
            caraux[i] := a_pagant.hijo[i]
        END;

    caraux[MAXLLAVES+1] := a_pagant.hijo[MAXLLAVES+1];

    i := MAXLLAVES + 1;

    while ((llave < llavesaux[i-1]) AND (i > 1)) DO
        BEGIN
            llavesaux[i] := llavesaux[i-1];{inserta la llave nueva}
            caraux[i+1] := caraux[i];
            i := i - 1;
        END;
    llavesaux[1] := llave;
    caraux[i+1] := hijo_d;
    hijo_d_promo := tomapag; {crea la página nueva para la división y   }
    iniciapag(a_pagnue);     {promueve el NRR de la página nueva   }

    for i := 1 TO MINLLAVES DO {mueve la primera mitad de las llaves   }
        BEGIN
            {e hijos a la página anterior,           }
            a_pagant.llave[i] := llavesaux[i]; {y la segunda a la página nueva   }
            a_pagant.hijo[i] := caraux[i];
            a_pagnue.llave[i] := llavesaux[i+1+MINLLAVES];
            a_pagnue.hijo[i] := caraux[i+1+MINLLAVES];
            a_pagant.llave[i+MINLLAVES] := SINLLAVE; {marca la segunda mitad}
            a_pagant.hijo[i+1+MINLLAVES] := NULO {de la página anterior como vacía}
        END;

    a_pagant.hijo[MINLLAVES+1] := caraux[MINLLAVES+1];
    a_pagnue.hijo[i+1+MINLLAVES] := caraux[i+2+MINLLAVES];
    a_pagnue.contllave := MAXLLAVES - MINLLAVES;
    a_pagant.contllave := MINLLAVES;
    llave_promo := llavesaux[MINLLAVES+1] {promueve la llave de enmedio}
END;

```

10

OBJETIVOS

Presentar los archivos *secuenciales indizados*.

Describir las operaciones sobre un *conjunto de secuencias* de bloques que mantienen registros clasificados por llave.

Mostrar cómo puede construirse un *conjunto de índices* encima del conjunto de secuencias para producir una estructura de archivos secuencial indizada.

Mostrar el uso de un árbol B para mantener el conjunto de índices, y presentar los *árboles B⁺* y los *árboles B⁺ de prefijos simples*.

Ilustrar cómo el conjunto de índices del árbol B puede ser de orden variable en un árbol B⁺ de prefijos simples, y almacenar un número variable de separadores.

Comparar las ventajas y debilidades de los árboles B⁺, los árboles B⁺ de prefijos simples y los árboles B.

LA FAMILIA DE LOS ARBOLES B⁺ Y EL ACCESO A LOS ARCHIVOS SECUENCIALES INDIZADOS

PLAN GENERAL DEL CAPITULO

- 10.1 Acceso secuencial indizado**
- 10.2 Mantenimiento de un conjunto de secuencias**
 - 10.2.1 Uso de bloques**
 - 10.2.2 Elección del tamaño del bloque**
- 10.3 Adición de un índice simple al conjunto de secuencias**
- 10.4 Contenido del índice: separadores en lugar de llaves**
- 10.5 Árboles B* de prefijos simples**
- 10.6 Mantenimiento de árboles B* de prefijos simples**
 - 10.6.1 Cambios localizados en bloques individuales del conjunto de secuencias**
 - 10.6.2 Cambios que involucran varios bloques en el conjunto de secuencias**
- 10.7 Tamaño de bloque del conjunto índice**
- 10.8 Estructura interna de los bloques del conjunto índice: árbol B de orden variable**
- 10.9 Carga de un árbol B* de prefijos simples**
- 10.10 Árboles B***
- 10.11 Los árboles B, B* y B* de prefijos simples en perspectiva**

10.1

ACCESO SECUENCIAL INDIZADO

Las estructuras de archivos secuenciales indizados permiten elegir entre dos formas alternativas de visualizar un archivo.

- Indizado:* el archivo puede verse como un conjunto de registros *indizado por llave*, o
- Secuencial:* se puede acceder secuencialmente el archivo (con registros físicamente contiguos, sin hacer desplazamientos), devolviendo los registros en el orden de la llave.

La idea de tener un solo método organizacional que proporcione ambos puntos de vista es nueva; hasta ahora se había tenido que elegir alguno de ellos. Como un ejemplo algo extremo, aunque ilustrativo, de la divergencia potencial de estas dos opciones, supóngase que se ha desarrollado una estructura de archivo que consiste en un conjunto de

registros de entradas secuenciales, indizados por un árbol B separado. Esta estructura puede dar un excelente acceso *indizado* a cualquier registro individual por llave, aun cuando se agreguen o eliminen registros. Ahora supongamos que también se desea usar este archivo como parte de una intercalación secuencial coordinada. En el procesamiento secuencial coordinado se desea extraer todos los registros en el orden de la llave. Puesto que los registros reales en este archivo están *en secuencia de entrada*, y no físicamente clasificados por llave, la única forma de extraerlos en el orden de la llave es por medio del índice. Para un archivo de N registros, seguir los N apuntadores del índice al conjunto de entradas secuenciales requiere N desplazamientos en principio aleatorios dentro del archivo de registros. Este es un proceso *mucho* menos eficiente que la lectura secuencial de los registros físicamente adyacentes; tanto, que es inaceptable para cualquier situación en la que el procesamiento secuencial coordinado sea frecuente.

Por otro lado, el análisis de la indización mostró que un archivo que consiste en un conjunto de registros clasificados por llave, aunque ideal para el procesamiento secuencial coordinado, es una estructura inaceptable cuando se desea acceder, insertar y eliminar registros por llave en forma aleatoria.

¿Qué sucede si una aplicación incluye tanto el acceso aleatorio interactivo como el procesamiento secuencial coordinado? Existen muchos ejemplos de tales aplicaciones duales. Por ejemplo, los sistemas de registros de estudiantes en universidades requieren acceso por llave a registros individuales, pero también exigen gran cantidad de procesamiento por lotes, como cuando se obtienen listas de calificaciones o cuando se pagan colegiaturas durante las inscripciones. En forma similar, los sistemas de tarjetas de crédito requieren tanto el procesamiento por lotes de captura de cargos como revisiones interactivas de los estados de cuenta. Los métodos de acceso secuencial indizado se desarrollaron en respuesta a este tipo de necesidades.

10.2

MANTENIMIENTO DE UN CONJUNTO DE SECUENCIAS

Por el momento se dejará de lado la parte indizada del acceso secuencial indizado, y se enfocará el problema de mantener un conjunto de registros en orden físico por llave mientras se agregan o eliminan registros. A este conjunto ordenado se le denomina *conjunto de secuencias*. Se supondrá que, cuando se obtenga una forma adecuada de mantener un conjunto de secuencias, también se encontrará una forma de indizarla.

10.2.1 USO DE BLOQUES

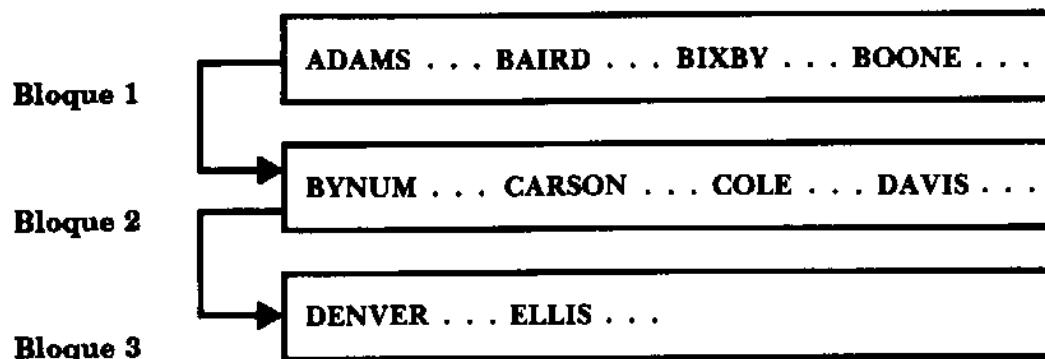
Se puede desechar desde ahora la idea de clasificar y reclasificar todo el conjunto de secuencias conforme se agregan o eliminan registros, pues se sabe que la clasificación de un archivo entero es un proceso costoso. En lugar de ello es necesario encontrar una forma de *localizar los cambios*. Una de las mejores formas de restringir los efectos de una inserción o eliminación sólo a una parte del conjunto de secuencias es usar un recurso que se mencionó por vez primera en los capítulos 3 y 4: se puede agrupar los registros en *bloques*.

Cuando los registros se agrupan en bloques, el bloque se convierte en la unidad básica de entrada y salida: se leen y escriben bloques enteros a la vez. En consecuencia, el tamaño de los buffers que se usan en un programa permite almacenar un bloque entero. Después de leer un bloque, todos los registros del bloque están en memoria RAM, donde se pueden manejar o reacomodar mucho más rápidamente.

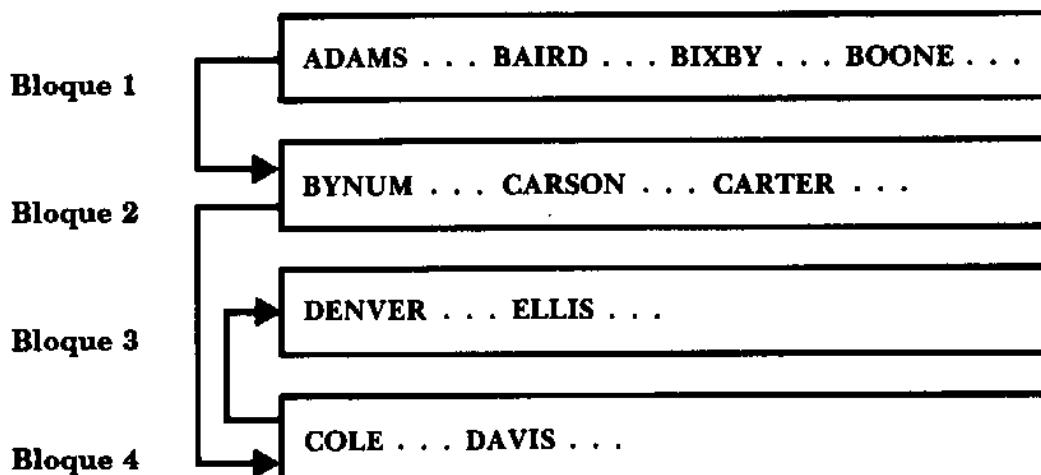
Un ejemplo ayuda a ilustrar cómo el uso de bloques sirve para mantener en orden un conjunto de secuencias. Supóngase que se tienen registros con llaves por apellido y agrupados de tal forma que existen cuatro registros en un bloque. También se incluyen en cada bloque *campos de liga* que apuntan al bloque precedente y al siguiente; estos campos son necesarios porque, como se verá, los bloques consecutivos no son por fuerza adyacentes físicamente.

Como sucede con los árboles B, la inserción de nuevos registros en un bloque puede hacer que el bloque se *sature*. El estado de saturación puede manejarse por medio de un proceso de división de bloques que es análogo, pero no el mismo, que el proceso de división usado en un árbol B. Por ejemplo, la figura 10.1 (a) muestra cómo se ve el conjunto de secuencias en bloques antes de realizar cualquier inserción o eliminación. Sólo se muestran las ligas hacia adelante. En la figura 10.1 (b) se ha insertado un registro nuevo con la llave CARTER. Esta inserción hace que se divida el bloque 2. La segunda mitad de lo que era originalmente el bloque 2 se encuentra en el bloque 4 después de la división. Nótese que este proceso de división de bloques es diferente del de los árboles B. En un árbol B, una división da como resultado la *promoción* de un registro. Aquí las cosas son más simples; sólo se dividen los registros entre dos bloques y se reacomodan las ligas de tal modo que aún sea posible moverse a lo largo del archivo en el orden de las llaves, bloque tras bloque.

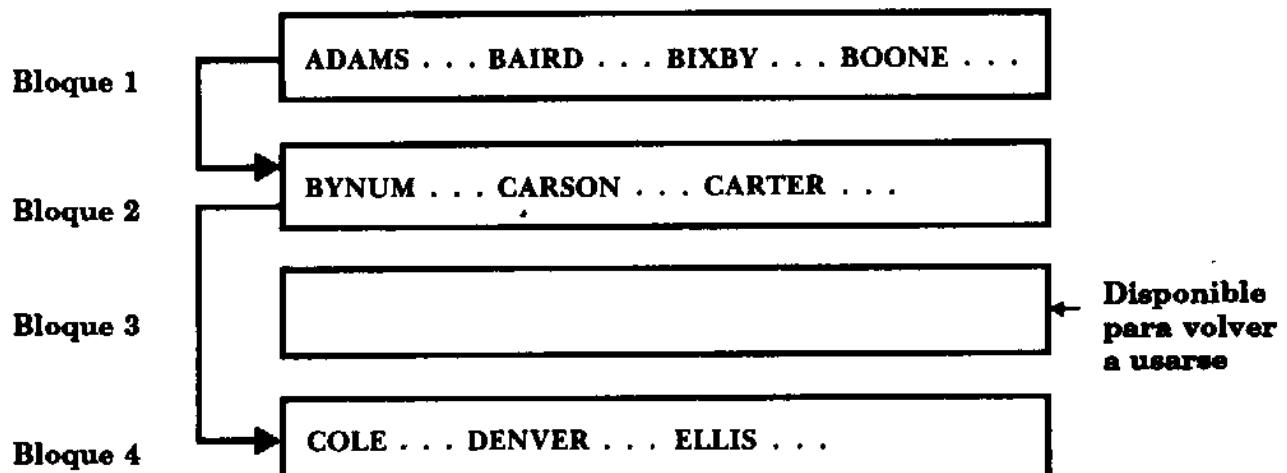
La eliminación de registros puede ocasionar que un bloque esté lleno a menos de la mitad y, por lo tanto, presente *insuficiencia*. Una vez más, este problema y sus soluciones son análogos a los que se encuentran cuando se trabaja con árboles B. La insuficiencia en un árbol B puede conducir a cualquiera de estas dos soluciones.



(a)



(b)



(c)

FIGURA 10.1 • División y concatenación de bloques debidas a inserciones y eliminaciones en el conjunto de secuencias.

(a) Conjunto inicial de secuencias en bloques. (b) Conjunto de secuencias después de la inserción del registro CARTER: el bloque 2 se divide y el contenido se reparte entre los bloques 2 y 4. (c) Conjunto de secuencias después de la eliminación del registro DAVIS: el bloque está lleno a menos de la mitad, así que se concatena con el bloque 3.

- Si un nodo vecino también está medio lleno, se pueden *concatenar* los dos nodos, con lo que se libera uno para que pueda usarse de nuevo.
- Si los nodos adyacentes están llenos hasta más de la mitad, los registros pueden *redistribuirse* entre los nodos para hacer que la distribución sea más equilibrada.

La insuficiencia dentro de un bloque del conjunto de secuencias puede manejarse con los mismos tipos de procesos. Como ocurre con la inserción, el proceso para el conjunto de secuencias es más simple que el proceso para los árboles B porque el conjunto de secuencias *no es un árbol* y, por lo tanto, no existen llaves ni registros en un nodo padre. En la figura 10.1 (c) se muestran los efectos de la eliminación del registro DAVIS. El bloque 4 es insuficiente, por lo que se concatena con su sucesor en la secuencia *lógica*, que es el bloque 3. El proceso de concatenación libera el bloque 3 para su reutilización. No se muestra un ejemplo en el que la insuficiencia conduzca a la redistribución porque es fácil observar cómo funciona este proceso. Los registros simplemente se mueven entre los bloques lógicamente adyacentes.

Tomando en cuenta la separación de registros en bloques, junto con estas operaciones fundamentales de división, concatenación y redistribución de bloques, puede mantenerse un conjunto de secuencias en orden de llaves sin tener que clasificar todo el conjunto de registros. Como siempre, nada es gratis, de modo que evitar la clasificación tiene su costo:

- Una vez que se hacen las inserciones, el archivo ocupa más espacio que un archivo que no esté en bloques de registros clasificados, debido a la fragmentación interna dentro de un bloque. Sin embargo, pueden aplicarse los mismos tipos de estrategias empleadas para incrementar la utilización de espacio en un árbol B (p.ej., usar la redistribución en lugar de la división durante la inserción, la división de dos a tres, etc). Una vez más, la implantación de cualquiera de estas estrategias debe tomar en cuenta el hecho de que el conjunto de secuencias *no es un árbol* y que, por lo tanto, no hay promoción de registros.
- El orden de los registros no es necesariamente secuencial *en forma física* a lo largo del archivo. La máxima extensión garantizada de una secuencia física está contenida en un bloque.

Este último punto conduce al importante aspecto de la elección del tamaño del bloque.

10.2.2 ELECCION DEL TAMAÑO DEL BLOQUE

Al trabajar con el conjunto de secuencias, el bloque es la unidad básica para las operaciones de E/S. Cuando se leen datos del disco, nunca se lee menos de un bloque; cuando se escriben datos, siempre se escribe al menos un bloque. Un bloque también es, como se ha dicho, la máxima extensión *garantizada* de una secuencia física, de ahí que deba pensarse en términos de bloques *grandes*, cada uno con muchos registros. Así que el problema del tamaño del bloque consiste en identificar sus *límites*: ¿Por qué no se hace el bloque tan grande que pueda caber en él el archivo entero?

La respuesta es la misma razón por la que no puede usarse siempre una clasificación en memoria RAM para un archivo: por lo regular no se tiene suficiente memoria disponible. Así pues, la primera consideración con respecto a una frontera superior para el tamaño de bloque es la siguiente.

Consideración 1: El tamaño de bloque debe ser tal que puedan almacenarse varios bloques en memoria RAM a la vez. Por ejemplo, al realizar una división o concatenación es deseable poder almacenar al menos dos bloques en memoria RAM a la vez. Si se hace la división de dos a tres para conservar el espacio en disco, es necesario almacenar al menos tres bloques en memoria RAM a la vez.

Aunque por ahora se dedica la atención a la posibilidad de acceder al conjunto de secuencias *en forma secuencial*, a la larga se querrá considerar el problema de acceder en forma aleatoria a un solo registro del conjunto de secuencias. Se tiene que leer un bloque entero para llegar a cualquier registro dentro de ese bloque. Por lo tanto, puede establecerse una segunda consideración.

Consideración 2: Leer o escribir un bloque no debe tomar demasiado tiempo. Incluso si se tuviera una cantidad ilimitada de memoria RAM, conviene colocar un límite superior al tamaño de bloque de modo que no se termine leyendo el archivo entero para llegar a un solo registro.

Esta segunda consideración es imprecisa: ¿Cuánto tiempo es demasiado? Esta consideración puede refinarse descomponiendo en factores parte del conocimiento que tenemos de las características de desempeño de las unidades de disco:

Consideración 2 (redefinida): El tamaño de bloque debe ser tal que se pueda acceder a uno sin necesidad de pagar el costo de un desplazamiento en el disco dentro de la operación de lectura o escritura del bloque.

Esta no es una limitación *obligatoria*, pero es razonable: nos interesa un bloque porque contiene registros que están físicamente adyacentes, así que no se van a extender los bloques más allá del punto en el que pueda garantizarse dicha adyacencia. ¿Y dónde está ese punto?

Cuando en el capítulo 3 se analizaron los discos con formato por sectores, se introdujo el término *cúmulo*. Un cúmulo es el mínimo número de sectores asignados a la vez. Si un cúmulo consiste en ocho sectores, como en un disco fijo normal en un computador IBM PC/XT, entonces incluso un archivo que contenga sólo un byte usa los ocho sectores (4096 bytes) en el disco. La razón de hacer cúmulos es que se garantiza una cantidad mínima de "secuencialidad" física. Cuando nos movemos de cúmulo en cúmulo en la lectura de un archivo, puede hacerse un desplazamiento en disco, pero dentro de un cúmulo se puede acceder a los datos sin él.

Así, una sugerencia razonable al elegir el tamaño del bloque es hacer cada bloque igual al tamaño de un cúmulo. Con frecuencia el tamaño del cúmulo en un sistema de discos ya ha sido determinado por el administrador del sistema. Pero ¿qué pasa si se configura un sistema de disco para una aplicación en particular y, por lo tanto, se puede elegir un tamaño de cúmulo propio? Entonces será necesario considerar los aspectos relacionados con el tamaño del cúmulo que aparecen en el capítulo 3, junto con las restricciones impuestas por la cantidad disponible de memoria RAM y el número de bloques que se desee almacenar en la memoria a la vez. Como es tan frecuente, la decisión final podría ser un compromiso entre varias consideraciones divergentes.. Lo importante es que el arreglo sea una decisión en verdad documentada, esto es, basada en el conocimiento de cómo trabajan los dispositivos de E/S y las estructuras de archivos, y no una mera suposición.

Si se trabaja con un sistema de disco que no está orientado por sectores, pero que permite elegir el tamaño del bloque para un archivo en particular, un buen punto de partida es imaginar un bloque como una pista entera del disco; o quizás se deseé replantear esto en un nivel inferior, como la mitad de una pista, por ejemplo, dependiendo de las restricciones de memoria, el tamaño del registro y otros factores.

10.3

ADICION DE UN INDICE SIMPLE AL CONJUNTO DE SECUENCIAS

Se ha creado un mecanismo para mantener un conjunto de registros de modo que se pueda acceder a ellos en forma secuencial en el orden de la llave. Esto está basado en la idea de agrupar registros en bloques

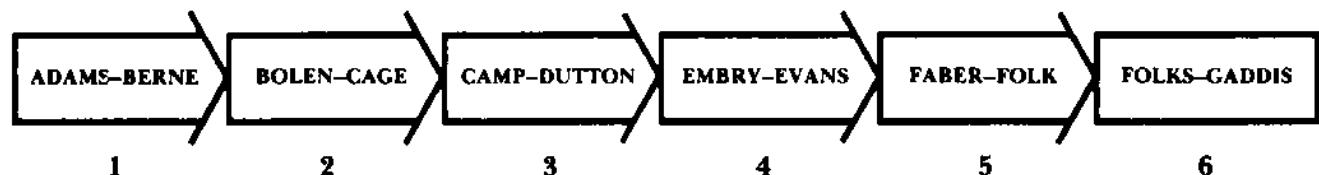


FIGURA 10.2 • Secuencia de bloques que muestra el intervalo de llaves en cada bloque.

y mantener los bloques, conforme se insertan y eliminan registros, por medio de la división, concatenación y redistribución. Ahora se verá si puede encontrarse una forma eficiente de localizar algunos bloques específicos que contengan un registro en particular, conociendo la llave del registro.

Puede considerarse que cada uno de los bloques contiene un *intervalo* de registros, como se ilustra en la figura 10.2. Este es un punto de vista externo respecto a los bloques (en realidad no se ha leído ningún bloque, por lo que no se sabe *exactamente* lo que contienen), pero es lo bastante informativo para ayudar a elegir el bloque que *quizá* tenga el registro que se busca. Puede observarse, por ejemplo, que si se está buscando un registro con la llave BURNS, se querrá extraer e inspeccionar el segundo bloque.

Es fácil comprender cómo podría construirse un índice simple de un solo nivel para estos bloques. Por ejemplo, se puede optar por construir un índice de registros de longitud fija que contenga la llave del último registro en cada bloque, como se muestra en la figura 10.3.

La combinación de este tipo de índices con el conjunto de secuencias de bloques proporciona acceso secuencial indizado completo. Si es necesario extraer un registro específico, se consulta el índice y después se extrae el bloque correcto; si se necesita acceso secuencial, se comienza en el primer bloque y se lee en la lista ligada de bloques hasta que se han leído todos. A este método tan simple se le puede sacar

Llave	Número de bloque
BERNE	1
CAGE	2
DUTTON	3
EVANS	4
FOLK	5
GADDIS	6

FIGURA 10.3 • Índice simple para el conjunto de secuencias ilustrado en la figura 10.2.

mucho provecho siempre y cuando el índice pueda almacenarse en la memoria electrónica RAM. El requisito de que el índice esté almacenado en memoria RAM es importante por dos razones:

- Como éste es un índice simple del tipo que se analiza en el capítulo 7, los registros específicos se encuentran por medio de una búsqueda binaria en el índice. La búsqueda binaria funciona bien si se realiza en memoria RAM, pero, como se vio en el capítulo anterior sobre árboles B, requiere demasiados desplazamientos si el archivo está en un dispositivo de almacenamiento secundario.
- Conforme cambian los bloques en el conjunto de secuencias por medio de la división, concatenación y redistribución, el índice tiene que actualizarse. Actualizar un índice simple de registros de longitud fija de este tipo funciona bien cuando es relativamente pequeño y está contenido en memoria RAM. Sin embargo, si la actualización requiere desplazamientos a registros individuales del índice en el disco, el proceso puede ser muy costoso. Una vez más, éste es un punto que se analizó en forma más completa en capítulos anteriores.

Entonces, ¿qué se hace si el archivo contiene tantos bloques que el índice de bloques no entra en forma concerniente dentro de la memoria RAM? En el capítulo anterior se encontró que la estructura del índice podía dividirse en *páginas*, en forma muy similar a los *bloques* que se analizan aquí, manejando a la vez varias páginas, o bloques, del índice en memoria RAM. De manera más específica, se vio que los árboles B son una estructura de archivo excelente para manejar índices demasiado grandes para caber por completo en la memoria RAM, lo cual sugiere que el índice para el conjunto de secuencias puede organizarse como un árbol B.

El uso de un índice en árbol B para el conjunto de secuencias de bloques es, de hecho, un concepto poderoso. La estructura híbrida resultante se conoce como *árbol B⁺*, lo cual es apropiado porque es un índice en árbol B más un conjunto de secuencias que almacena los registros reales. Antes de poder desarrollar por completo la idea de un árbol B⁺ es preciso pensar más detenidamente en lo que se necesita almacenar en el índice.

10.4

CONTENIDO DEL INDICE: SEPARADORES EN LUGAR DE LLAVES

El propósito del índice que se está construyendo es asistir al usuario cuando busca un registro con una llave específica. El índice debe guiar

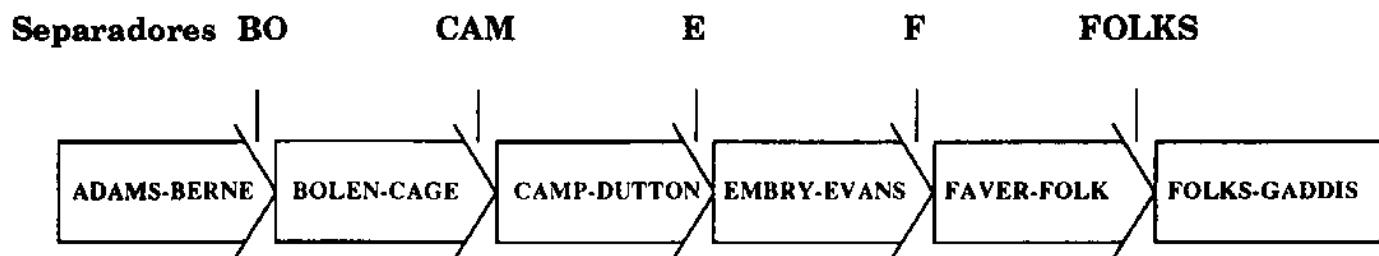


FIGURA 10.4. Separadores entre bloques en el conjunto de secuencias.

hacia el bloque del conjunto de secuencias que contiene el registro, si es que existe en ese conjunto. El índice sirve como una especie de mapa para el conjunto de secuencias. Interesa el contenido del índice sólo en tanto que ayuda obtener el bloque correcto en el conjunto de secuencias; el conjunto índice por sí mismo no contiene respuestas, contiene sólo información acerca de a dónde ir para obtener respuestas.

Con esta consideración sobre el conjunto índice como un mapa puede darse el importante paso de reconocer que *no es necesario tener llaves reales en el conjunto índice*. La necesidad real es de separadores. La figura 10.4 muestra un posible conjunto de separadores para el conjunto de secuencias de la figura 10.2.

Nótese que existen muchos separadores potenciales capaces de distinguir entre dos bloques. Por ejemplo, todas las cadenas mostradas entre los bloques 3 y 4 en la figura 10.5 son capaces de servir como guía en la elección de bloques cuando se busca una llave en particular. Si una comparación de cadenas entre llave y cualquiera de estos separadores muestra que la llave precede al separador, se busca la llave en el bloque 3. Si la llave está después del separador, se busca en el bloque 4.

Si se ha decidido tratar a los separadores como entidades de longitud variable en la estructura del índice (después se hablará acerca de cómo hacer esto), puede ahorrarse espacio colocando el *separador más corto* en la estructura del índice. En consecuencia, se usa *E* como separador para guiar la elección entre los bloques 3 y 4. Nótese que no siempre existe un único separador más corto. Por ejemplo, BK, BN y BO

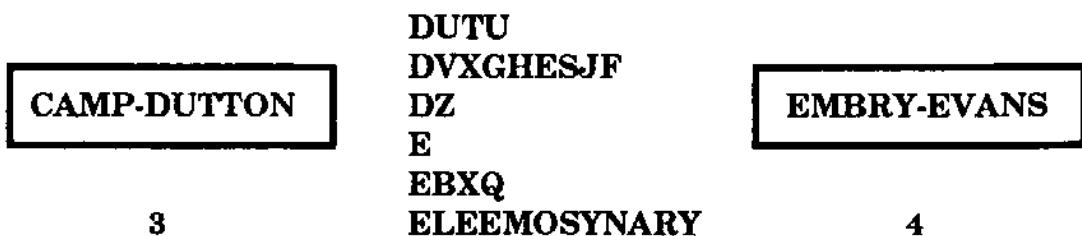


FIGURA 10.5. Una lista de separadores potenciales.

```

/* encuentra_sep(llavel, llave2, sep)...

encuentra la cadena más corta que sirve como separador entre llavel y llave2.
Devuelve este separador por medio de la dirección proporcionada por el parámetro
"sep"

la función supone que llave2 sigue a llavel en la secuencia lexicográfica
*/
encuentra_sep(llavel, llave2, sep)
    char llave[], llave2[], sep[]
{
    while ( (*sep++ = *llave2++) == *llavel++)
        *sep='\0';      /* se asegura de que la cadena separadora termine con nulo */
}

```

FIGURA 10.6• Función en C para encontrar el separador más corto.

son separadores de la misma longitud e igualmente efectivos como separadores entre los bloques 1 y 2 de la figura 10.4. Se eligen BO y todos los demás separadores contenidos en la figura 10.4, usando la lógica incorporada en la función de C que se muestra en la figura 10.6 y en el procedimiento de Pascal listado en la figura 10.7.

```

PROCEDURE encuentra_sep(llavel, llave2 : cadena ; VAR sep : cadena)
{ encuentra la cadena más corta que sirve como separador entre llavel y llave2. Devuelve
este separador por medio de la variable sep. Las cadenas se manejan como arreglos de
caracteres en los que la longitud de la cadena se almacena en la posición 0 del arreglo.
El tipo "cadena" se usa para cadenas.

```

```

La función supone que llave2 sigue a llavel en la secuencia lexicográfica.
Usa dos funciones definidas en el apéndice:
    long_cad (s) - devuelve la longitud de la cadena s.
    min (i, j) - compara i y j y devuelve el valor más pequeño
}
VAR
    i, longmin: integer;
BEGIN
    longmin: = min(longcad(llavel), longcad(llave2));
    i: = 1;
    while (llavel[i] = llave2[i] and (i <= longmin) DO
    BEGIN
        sep[i]: = llave2 [i];
        i: = i +1
    END;
    sep [i]: = llave2 [i];
    sep [0]: = OHR (i) {coloca el indicador de longitud en el arreglo del separador
}
END;

```

FIGURA 10.7• Procedimiento en Pascal para encontrar el separador más corto.

Obsérvese que estas funciones pueden producir un separador igual a la segunda llave. Esta situación se ilustra en la figura 10.4 con el separador entre los bloques 5 y 6, el cual es el mismo que la primera llave contenida en el bloque 6. De esto se sigue que, cuando se usan los separadores como un mapa para el conjunto de secuencias, debe decidirse extraer el bloque que está a la derecha del separador o bien aquel que está a su izquierda, de acuerdo con la siguiente regla:

Relación de la llave de búsqueda y el separador	Decisión
Llave < separador	Ir a la izquierda
Llave== separador	Ir a la derecha
Llave > separador	Ir a la derecha

10.5

ARBOLES B⁺ DE PREFIJOS SIMPLES

La figura 10.8 muestra cómo pueden colocarse los separadores identificados en la figura 10.4 en forma de índice de un árbol B para los bloques del conjunto de secuencias. Al índice en forma de árbol B se le llama *conjunto índice*, y éste y el conjunto de secuencias forman una estructura de archivos llamada *árbol B⁺ de prefijos simples*. El modificador *prefijos simples* indica que el conjunto índice contiene los separadores más cortos, o *prefijos* de las llaves, en lugar de copias de las llaves verdaderas. Los separadores son sencillos porque son simplemente prefijos: en realidad sólo son las letras iniciales de las llaves. Otros métodos más complicados de creación de separadores a partir de prefijos de llaves eliminan caracteres innecesarios del inicio de los separadores, así como del final (Véase Bayer y Unterauer [1977] para un análisis más completo de los árboles B⁺ de prefijos.)†

Nótese que, puesto que el conjunto índice es un árbol B, un nodo que contiene N separadores se ramifica en $N + 1$ hijos. Si se está buscando

† La literatura sobre árboles B⁺ y árboles B⁺ de prefijos simples es notablemente inconsistente en la nomenclatura usada para estas estructuras. A los árboles B⁺ se les llama algunas veces árboles B^{*}; los árboles B⁺ de prefijos simples se llaman algunas veces árboles B de prefijo simples. El importante artículo de Comer en *Computing Surveys* de 1979 ha reducido un poco la confusión proporcionando una nomenclatura estándar y consistente, que se usa aquí.

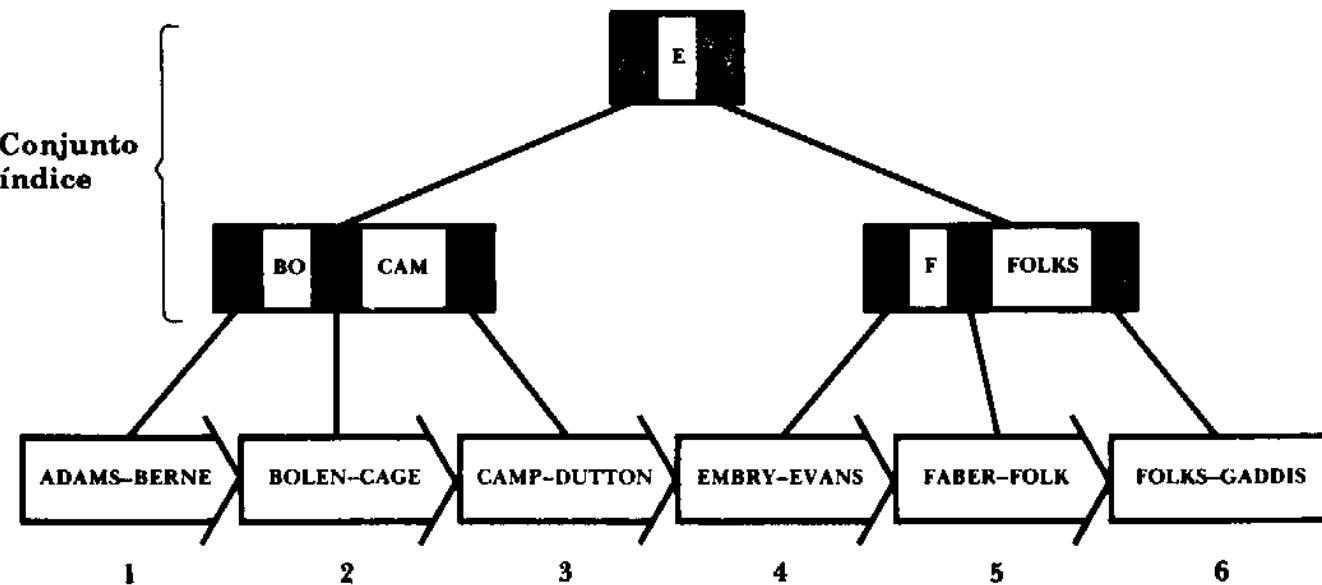


FIGURA 10.8• Un conjunto índice en forma de árbol B para el conjunto de secuencias, que forma un árbol B* de prefijos simples.

el registro con la llave EMBRY, se comienza en la raíz del conjunto índice, comparando EMBRY con el separador E. Puesto que EMBRY viene después de E, se va hacia la derecha y se extrae el nodo que contiene los separadores F y FOLKS. Puesto que EMBRY está aún antes que el primero de estos separadores, se sigue la rama que está a la izquierda del separador F, la cual lleva al bloque 4, el correcto en el conjunto de secuencias.

10.6

MANTENIMIENTO DE ARBOLES B+ DE PREFIJOS SIMPLES

10.6.1 CAMBIOS LOCALIZADOS EN BLOQUES INDIVIDUALES DEL CONJUNTO DE SECUENCIAS

Supongamos que se quiere eliminar los registros de EMBRY y FOLKS, y que esto da como resultado una concatenación o redistribución dentro del conjunto índice. Como no hay concatenación o redistribución, el efecto de estas eliminaciones en el *conjunto de secuencias* está limitado a cambios dentro de los bloques 4 y 6. El registro que antes era el segundo en el bloque 4 (digamos que su llave es ERVIN) es ahora el primer registro. De igual modo, el que era el segundo registro del bloque 6 (se supone que tiene una llave FROST) inicia ahora ese bloque. Estos cambios pueden verse en la figura 10.9.

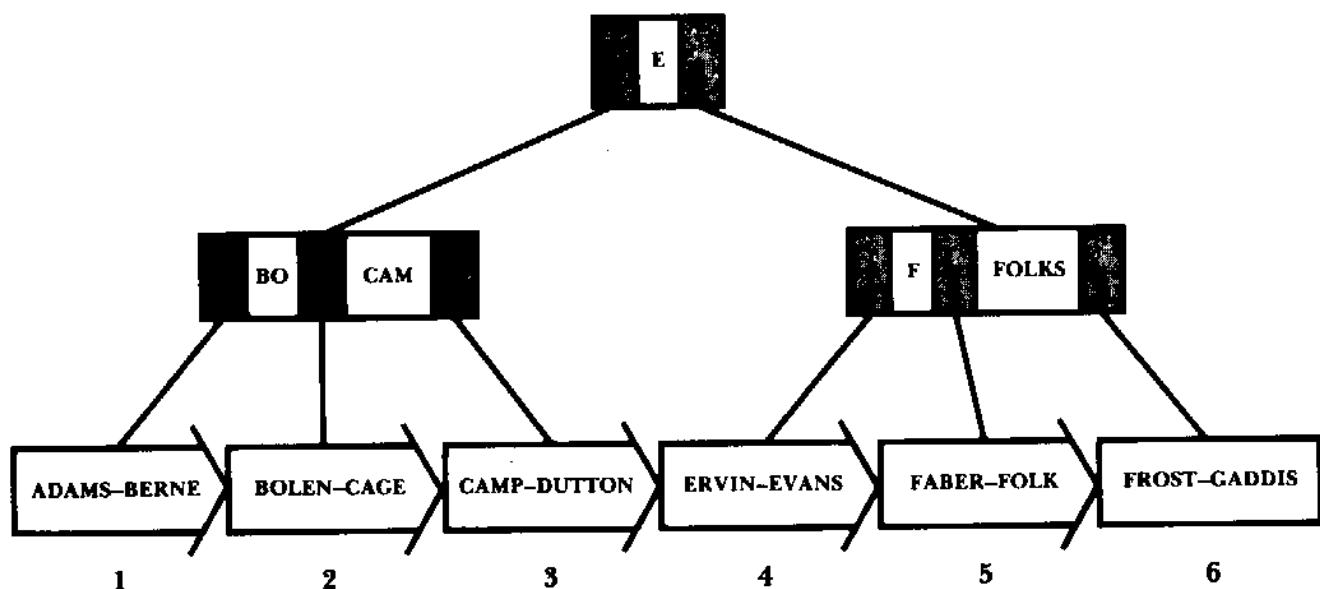


FIGURA 10.9 • La eliminación de los registros EMBRY y FOLKS del conjunto de secuencias no altera el conjunto índice.

La cuestión más interesante es cuál será el efecto, si lo hay, de estas eliminaciones en el *conjunto índice*. La respuesta es que, debido a que el número de bloques del conjunto de secuencias no cambia, y a que ningún registro se mueve entre los bloques, el conjunto índice puede también permanecer sin cambios. Esto es fácil de ver en el caso de la eliminación de EMBRY: E sigue siendo un separador adecuado para los bloques 3 y 4 del conjunto de secuencias, así que no hay razón de cambiarlo en el conjunto índice. El caso de la eliminación de FOLKS es un poco más confuso, porque la cadena FOLKS aparece como llave en el registro eliminado y además como separador dentro del conjunto índice. Para evitar la confusión, recuérdese distinguir claramente entre estos dos usos de la cadena FOLKS: FOLKS puede continuar sirviendo como separador entre los bloques 5 y 6 aunque el registro FOLKS se haya eliminado. (Se podría argumentar que aunque no sea necesario reemplazar el separador FOLKS, debe hacerse de cualquier forma, porque ahora es posible construir un separador más corto. Sin embargo, el costo de hacer tal cambio en el conjunto índice por lo regular es mayor que los beneficios asociados con el ahorro de unos cuantos bytes de espacio.)

El efecto que tiene insertar dentro del conjunto de secuencia registros nuevos que no provocan división del bloque es, en gran medida, el mismo que tienen estas eliminaciones que no dan como resultado la concatenación: El conjunto índice permanece sin cambio. Por ejemplo, supóngase que se inserta un registro para EATON. Siguiendo el camino indicado por los separadores en el conjunto índice,

se encuentra que el registro nuevo se insertará en el bloque 4 del conjunto de secuencias. Por el momento, se supone que hay lugar para el registro en el bloque. El registro nuevo se convierte en el primero del bloque 4, pero no es necesario ningún cambio en el conjunto índice. Esto no es sorprendente, puesto que se decidió insertar el registro en el bloque 4 sobre la base de la información existente en el conjunto índice. De esto se sigue que la información existente en el conjunto índice es suficiente para permitir encontrar el registro nuevamente.

10.6.2 CAMBIOS QUE INVOLUCRAN VARIOS BLOQUES EN EL CONJUNTO DE SECUENCIAS

¿Qué sucede cuando la inserción o eliminación de registros del conjunto de secuencias *hacen* cambiar el número de bloques? Resulta claro que si se tienen más bloques, se necesitan separadores adicionales en el conjunto índice, y si se tienen menos bloques, se necesitan menos separadores. Cambiar el número de separadores ciertamente tiene efecto en el conjunto índice, donde se almacenan los separadores.

Puesto que el conjunto índice para un árbol B⁺ de prefijos simples es en realidad sólo un árbol B normal, los cambios en el conjunto índice se manejan de acuerdo con las reglas familiares para la inserción y eliminación en árboles B.^t

En los siguientes ejemplos se supone que el conjunto índice es un árbol B de orden tres, lo cual significa que el máximo número de separadores que pueden almacenarse en un nodo es dos. Se usa este pequeño tamaño de nodo del conjunto índice para ilustrar la división y concatenación de nodos usando sólo unos cuantos separadores. Como se verá posteriormente, en la implantación real de árboles B⁺ de prefijos simples se colocan muchos más separadores en un nodo del conjunto índice.

Se comienza con una inserción dentro del conjunto de secuencias que se muestra en la figura 10.9. En forma específica, se va a suponer que hay una inserción en el primer bloque, y que esta inserción hace que el bloque se divida. Se trae un bloque nuevo (el bloque 7) para almacenar la segunda mitad de lo que era originalmente en el primer bloque. Este nuevo bloque se liga dentro de la posición correcta en el conjunto de secuencias, siguiendo al bloque 1 y precediendo al bloque 2 (éstos son los números físicos de los bloques). Estos cambios al conjunto de secuencias se ilustran en la figura 10.10.

^tAl estudiar este material, puede ser útil remitirse de nuevo al capítulo 9, donde se analizan las operaciones de los árboles B con mucho mayor detalle.

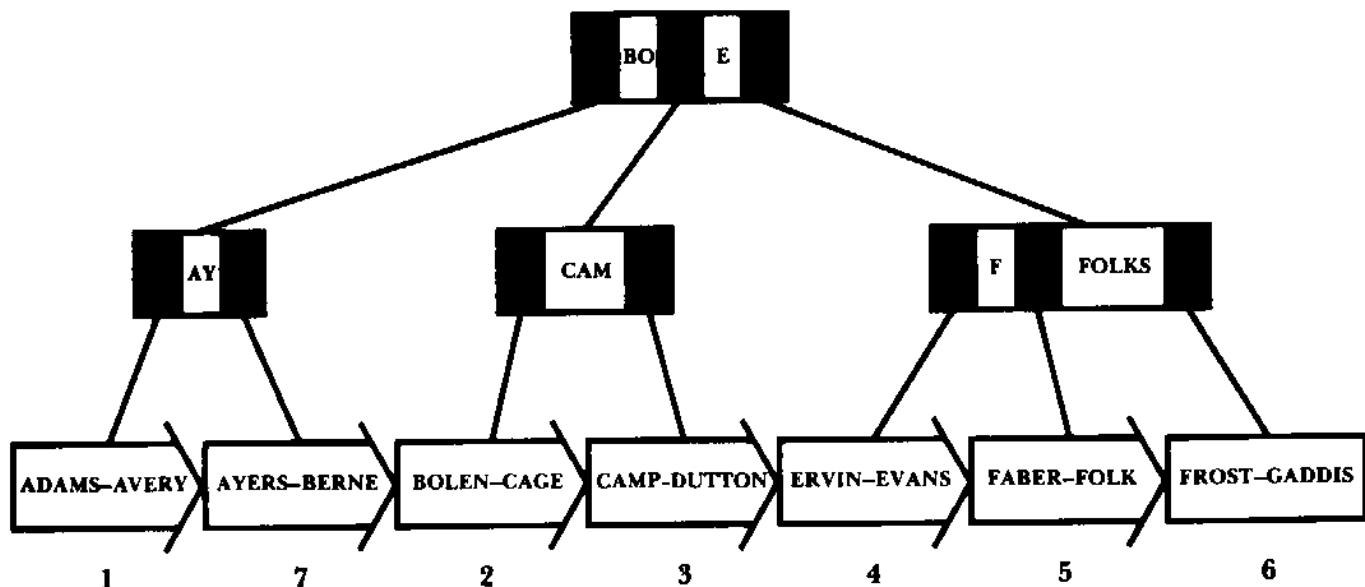


FIGURA 10.10 • Una inserción dentro del bloque 1 provoca una división y la consecuente adición del bloque 7. La adición de un bloque en el conjunto de secuencias requiere un separador nuevo en el conjunto índice. La inserción del separador AY dentro del nodo que contiene BO y CAM provoca una división de nodo en el árbol B que forma el conjunto índice y la consecuente promoción de BO a la raíz.

Nótese que el separador que antes distinguía entre los bloques 1 y 2, la cadena BO, es ahora el separador de los bloques 7 y 2. Se necesita un nuevo separador, con un valor AY, para distinguir entre los bloques 1 y 7. Al colocar este separador dentro del conjunto índice se encuentra que el nodo dentro del cual se desea insertarlo, que contiene BO y CAM, ya está lleno. En consecuencia, la inserción de un nuevo separador provoca una división y promoción, de acuerdo con las reglas usuales para árboles B. El separador promovido, BO, se coloca en la raíz del conjunto índice.

Ahora se supondrá que se elimina un registro del bloque 2 del conjunto de secuencias, que provoca una condición de insuficiencia y la consecuente concatenación de los bloques 2 y 3. Cuando la concatenación se completa, deja de necesitarse el bloque 3 en el conjunto de secuencias, y el separador que distinguía entre los bloques 2 y 3 debe eliminarse del conjunto índice. Eliminar este separador, CAM, provoca insuficiencia en un nodo del conjunto índice, por lo que hay otra concatenación, esta vez en el conjunto índice, que da como resultado la disección del separador BO de la raíz, trayéndolo hacia abajo a un nodo con el separador AY. Al terminar estos cambios, el árbol B⁺ de prefijos simples tiene la estructura que se ilustra en la figura 10.11.

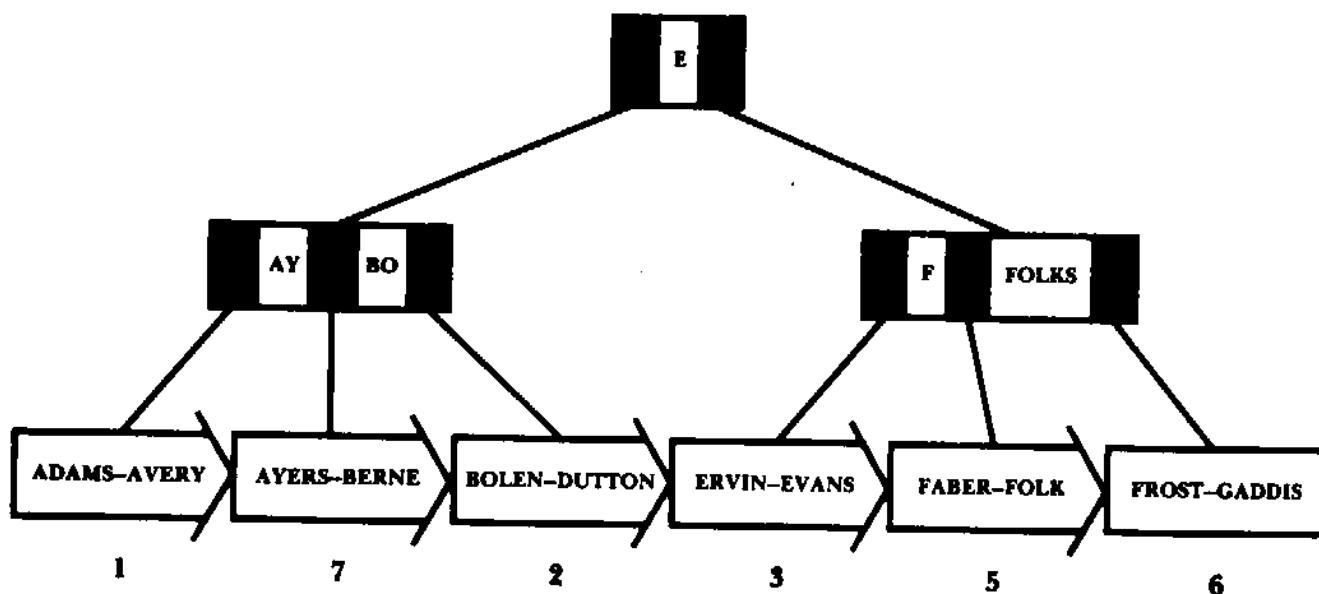


FIGURA 10.11 • Una eliminación en el bloque 2 provoca insuficiencia y la consecuente concatenación de los bloques 2 y 3. Después de la concatenación, el bloque 3 ya no se necesita y puede colocarse en una lista de disponibles, por lo que el separador CAM ya no se necesita tampoco. Eliminar CAM de su nodo en el conjunto índice fuerza una concatenación de nodos del conjunto índice, trayendo a BO hacia abajo desde la raíz.

Aunque una división de bloque en el conjunto de secuencias en estos ejemplos da como resultado una división de nodo en el conjunto índice, y una concatenación en el conjunto de secuencias da como resultado una concatenación en el conjunto índice, no siempre existe esta correspondencia de acciones. Las inserciones y eliminaciones en el conjunto índice se manejan como operaciones estándar de árboles B; el hecho de que ocurra división o inserción sencilla, concatenación o eliminación simple, depende por completo de qué tan lleno esté el nodo del conjunto índice.

Los procedimientos para manejar estos tipos de operaciones se escriben fácilmente si se recuerda que los cambios se realizan *de abajo hacia arriba*. La inserción y eliminación de registros *siempre* tiene lugar en el conjunto de secuencias, ya que es allí donde están los registros. Si la división, concatenación o redistribución son necesarias, la operación se efectúa como si el conjunto de secuencias *no existiera*. Entonces, después de que terminan las operaciones de los registros en el conjunto de secuencias, se hacen los cambios necesarios en el conjunto índice:

- Si los bloques se dividen en el conjunto de secuencias, debe insertarse un nuevo separador dentro del conjunto índice;

- Si se concatenan bloques en el conjunto de secuencias, debe eliminarse un separador del conjunto índice, y
- Si los registros se redistribuyen entre bloques en el conjunto de secuencias, debe cambiarse el valor de un separador en el conjunto índice.

Las operaciones en el conjunto índice se efectúan de acuerdo con las reglas de árboles B. Esto significa que la división y la concatenación de nodos se *propagan* hacia arriba a los niveles más altos del conjunto índice. Esto se observa en los ejemplos cuando el separador BO entra y sale de la raíz. Nótese que las operaciones en el conjunto de secuencias no implican este tipo de programación. Esto se debe a que el conjunto de secuencias es una lista ligada lineal, mientras que el conjunto índice es un árbol. Con facilidad se pierde de vista esta distinción y se considera una inserción o eliminación en términos de una *sola* operación sobre el árbol B⁺ de prefijos simples completo. Así resulta fácil confundirse. Recuérdese que las inserciones y eliminaciones suceden en el *conjunto de secuencias*, porque es allí donde están los registros. Los cambios en el conjunto índice son secundarios; son consecuencia de las operaciones fundamentales sobre el conjunto de secuencias.

10.7

TAMAÑO DE BLOQUE DEL CONJUNTO INDICE

Hasta este punto se han ignorado los importantes aspectos del tamaño y estructura de los nodos del conjunto índice. En los ejemplos se han usado nodos de conjunto índice en extremo pequeños y se les ha tratado como nodos de árbol B de orden fijo, a pesar de que los separadores son de longitud variable. Es necesario desarrollar ideas más realistas y útiles acerca del tamaño y estructura de los nodos del conjunto índice.

El tamaño físico de un nodo para el conjunto índice es, por lo regular, el mismo que el tamaño físico de un bloque en el conjunto de secuencias. Cuando es así, se habla de *bloques* del conjunto índice en lugar de *nodos*, así como se habla de bloques del conjunto de secuencias. Existen varias razones para usar un tamaño común de bloque para los conjuntos índice y de secuencias:

- Por lo regular se elige el tamaño del bloque para conjunto de secuencias porque hay un buen acoplamiento entre este tamaño de bloque, las características del disco y la cantidad de memoria disponible. La elección de un tamaño de bloque de conjunto de índice se rige por la consideración de los mismos factores; por lo

tanto, el mejor tamaño de bloque para el conjunto de secuencias suele ser el mejor para el conjunto índice.

- Un tamaño de bloque común facilita la implantación de un esquema de manejo de buffers para crear un árbol B⁺ de prefijos simples *virtual*, parecido a los árboles B virtuales analizados en el capítulo anterior.
- Los bloques del conjunto índice y los del conjunto de secuencias frecuentemente se incorporan dentro del mismo archivo para evitar desplazamientos en dos archivos separados cuando se tiene acceso a un árbol B⁺ de prefijos simples. El uso de un archivo para ambos tipos de bloques es más sencillo si los tamaños de bloques son iguales.

10.8

ESTRUCTURA INTERNA DE LOS BLOQUES DEL CONJUNTO INDICE: ARBOL B DE ORDEN VARIABLE

Dado un bloque fijo de tamaño grande para el conjunto índice, ¿cómo se almacenan los separadores dentro del bloque? En los ejemplos considerados hasta aquí, la estructura de bloques es de tal forma que sólo puede contener un número fijo de separadores. La razón de usar los separadores *más cortos* es la posibilidad de agruparlos dentro de un nodo. Esto desaparece por completo por si el conjunto índice usa un árbol B de orden fijo en el cual existe un número fijo de separadores por nodo.

Se desea que cada bloque del conjunto índice almacene un número variable de separadores de longitud variable. ¿Cómo debe procederse a la búsqueda por medio de estos separadores? Como los bloques probablemente son grandes, uno sencillo puede almacenar un gran número de separadores. Una vez que se lee un bloque en memoria RAM para su uso, es deseable poder hacer una búsqueda binaria en lugar de secuencial en su lista de separadores. Por ello, es necesario estructurar el bloque de tal forma que pueda manejarse una búsqueda binaria sin importar el hecho de que los separadores sean de longitud variable.

En el capítulo 7, donde se analiza la indización, se observa que el uso de un índice separado puede proporcionar un medio de efectuar búsquedas binarias sobre una lista de entidades de longitud variable. Si el índice mismo consiste en referencias de longitud fija, puede usarse la búsqueda binaria en el índice, extrayendo los registros o campos de longitud variable mediante indirección. Por ejemplo, supóngase que se

va a colocar el siguiente conjunto de separadores dentro de un bloque del índice:

As, Ba, Bro, C, Ch, Cra, Dele, Edi, Err, Fa, Fle.

(Se usan letras minúsculas, en lugar de todas mayúsculas, para que puedan encontrarse los separadores con mayor facilidad cuando se concatenen.) Podrían concatenarse estos separadores y construirse un índice para ellos, como se muestra en la figura 10.12.

Si se usa este bloque del conjunto índice como mapa para ayudar a encontrar el registro en el conjunto de secuencias para "Beck", se efectúa una búsqueda binaria en el índice de los separadores, extrayendo primero el separador de la mitad, "Cra", que comienza en la posición 10. Nótese que puede encontrarse la longitud de este separador examinando la posición inicial del separador que le sigue. La búsqueda binaria indicará, a la larga, que "Beck" cae entre los separadores "Ba" y "Bro". ¿Qué se hace entonces?

El propósito del mapa del conjunto índice es servir como guía hacia abajo, a través de los niveles del árbol B⁺ de prefijos simples, y conducir al bloque del conjunto de secuencias que se desea extraer. En consecuencia, el bloque del conjunto índice necesita alguna forma de almacenar referencias a sus hijos. Se supone que las referencias se hacen en términos de un número relativo de bloque (NRB), el cual es análogo al número relativo de registro excepto en que hace referencia a un bloque de longitud fija y no a un registro. Si hay N separadores dentro de un bloque, el bloque tiene N^* hijos, y por tanto se necesita espacio para almacenar N^* 1 NRB, además de los separadores y el índice a ellos.

Hay muchas formas de combinar la lista de separadores, los índices de los separadores y la lista de NRB dentro de un solo bloque del conjunto índice. Un método posible se ilustra en la figura 10.13. Además del vector de separadores, índice y la lista de números de bloque asociados, esta estructura de bloques incluye:



FIGURA 10.12 • Separadores de longitud variable y el índice correspondiente.

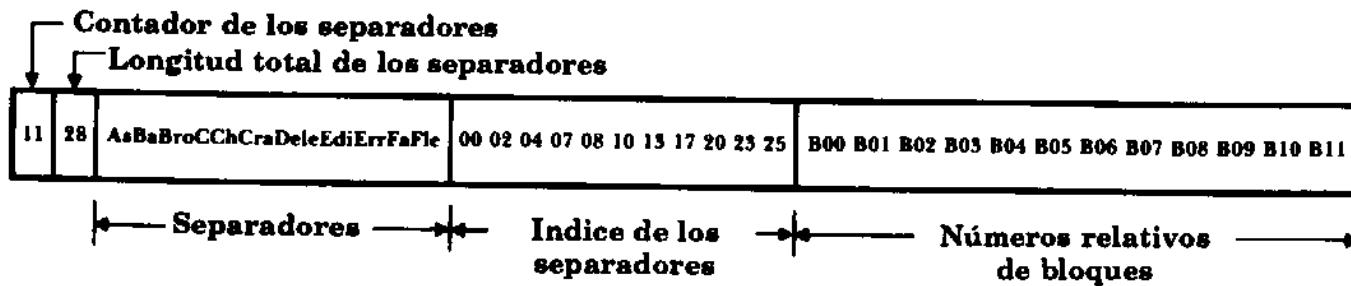


FIGURA 10.13 • Estructura de un bloque del conjunto índice.

- *Contador de separadores.* Se necesita para ayudar a encontrar el elemento del medio en el índice de los separadores, de tal forma que pueda empezarse la búsqueda binaria.
- *Longitud total de los separadores.* La longitud de la lista de separadores concatenados varía de bloque a bloque. Como el índice de los separadores comienza al final de esta lista de longitud variable, se necesita conocer su tamaño para encontrar el inicio del índice.

Supóngase, una vez más, que se busca un registro con la llave "Beck" y que la búsqueda ha conducido al bloque del conjunto índice mostrado en la figura 10.13. La longitud total de los separadores y el contador de separadores permiten encontrar el inicio, el final y, en consecuencia, la parte media del índice de los separadores. Como en el ejemplo anterior, se efectúa una búsqueda binaria de los separadores por medio de este índice, para concluir, finalmente, que la llave "Beck" está entre los separadores "Ba" y "Bro". *Conceptualmente*, la relación que se establece entre las llaves y los NRB es como se ilustra en la figura 10.14. (¿Por qué no es una buena disposición *física*?)

Como se observa en la figura 10.14, descubrir que la llave está entre "Ba" y "Bro" permite saber que el *siguiente* bloque que se necesita extraer tiene el NRB almacenado en la posición BO2 del vector NRB. Este siguiente bloque podría ser otro bloque del conjunto índice, y por

Subíndice de los separadores:

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

B00	Aa	B01	Ba	B02	Bro	B03	C	B04	Ch	B05	Cra	B06	Dele	B07	Edi	B08	Err	B09	Fa	B10	Fle	B11
-----	----	-----	----	-----	-----	-----	---	-----	----	-----	-----	-----	------	-----	-----	-----	-----	-----	----	-----	-----	-----

FIGURA 10.14 • Relación conceptual de los separadores y los números relativos de bloque.

tanto otro bloque del mapa, o podría ser el bloque del conjunto de secuencias que se busca. En cualquier caso, la cantidad y colocación de la información en el bloque del conjunto índice actual es suficiente para que se pueda hacer la búsqueda binaria *dentro* del bloque índice y después proceder al siguiente bloque en el árbol B⁺ de prefijos simples.

Hay muchas formas alternas de colocar los componentes fundamentales de este bloque índice. (Por ejemplo, ¿sería más fácil construir el bloque si el vector de llaves fuese colocado al final del bloque? ¿Cómo podría manejarse el hecho de que el bloque consista en entidades tanto de *caracteres* como de enteros sin ningún punto fijo constante que los dividiera?) Para nuestros propósitos, los detalles específicos de implantación para esta estructura particular de bloques de índices no son tan importantes como su *estructura conceptual*. Este tipo de estructuras de bloque de índice ilustra dos puntos importantes.

El primer punto es que un bloque no es sólo un fragmento arbitrario extraído de un archivo homogéneo; puede ser algo más que un conjunto de registros. Un bloque puede tener una compleja estructura interna propia, que incluya su propio índice interno, un conjunto de registros de longitud variable, conjuntos separados de registro de longitud fija, etc. Esta idea de construir estructuras de datos más complejas dentro de cada bloque se va volviendo atractiva conforme se incrementa el tamaño del bloque. Con bloques muy grandes resulta imperativo que se tenga un forma eficiente de procesar todos los datos dentro de un bloque una vez que se ha transferido a la memoria RAM. Este punto no se aplica sólo a árboles B⁺ de prefijos simples, sino a cualquier estructura de archivos en que se use un tamaño grande de bloque.

El segundo punto se refiere a que un nodo dentro del conjunto índice en forma de árbol B del árbol B⁺ de prefijos simples es de orden variable, puesto que cada bloque del conjunto índice contiene un número variable de separadores. Esta variabilidad tiene implicaciones interesantes:

- El número de separadores en un bloque está directamente limitado por el tamaño del bloque y no por algún *orden* predeterminado (como en un árbol B de *orden M*). El conjunto índice tendrá el orden máximo y, por tanto, la profundidad mínima que son posibles dado el grado de comprensión usado para formar los separadores.
- Como el árbol es de *orden variable*, operaciones como determinar si un bloque está lleno, o medio lleno, ya no son simple cuestión de comparar un contador de separadores con algún máximo o mínimo fijos. Las decisiones acerca de cuándo conviene dividir, concatenar o redistribuir se vuelven más complejas.

Los ejercicios al final del capítulo proporcionan oportunidades para explorar árboles de orden variable en forma más completa.

10.9

CARGA DE UN ARBOL B⁺ DE PREFIJOS SIMPLES

En la descripción anterior del árbol B⁺ de prefijos simples, primero se prestó atención a la construcción de un conjunto de secuencias, y luego se presentó el conjunto índice como algo que se agrega o se construye sobre el conjunto de secuencias. No sólo es posible *concebir* los árboles B⁺ de prefijos simples de esta forma, como un conjunto de secuencias con un índice adicional sino que también se pueden *construir* en esta forma.

Una forma de construir un árbol B⁺ de prefijos simples, por supuesto, es a través de una serie de inserciones sucesivas. Podrían usarse los procedimientos planteados en la sección 10.6, donde se analiza el mantenimiento de árboles B⁺ de prefijos simples, para dividir o redistribuir bloques en el conjunto de secuencias y en el conjunto índice, conforme se agregan bloques al conjunto de secuencias. La dificultad con ese método es que la división y la redistribución son relativamente costosas. Implican buscar hacia abajo, a través del árbol, para cada inserción y después reorganizar el árbol como sea necesario en el camino de vuelta. Estas operaciones son adecuadas para el *mantenimiento* del árbol, pero cuando se carga el árbol no se tiene que enfrentar un *orden aleatorio* de inserción y, por lo tanto, no es necesario recurrir a procedimientos tan poderosos, flexibles y costosos. En vez de eso puede comenzarse por clasificar los registros que se van a cargar, para así garantizar que el siguiente registro que se encuentre sea el siguiente registro que se requiera cargar.

Al trabajar a partir de un archivo clasificado, pueden colocarse los registros en los bloques del conjunto de secuencias, uno por uno, comenzando un bloque nuevo cuando se llena el que se está trabajando. Conforme se hace la transición entre dos bloques del conjunto de secuencias se puede ir determinando el separador más corto para los bloques. Estos separadores pueden agruparse dentro de un bloque del conjunto índice, que se construye y almacena en memoria RAM hasta que se llene.

Para desarrollar un ejemplo de cómo funciona esto, se supondrá que se tienen conjuntos de registros asociados con términos que se están compilando para el índice de un libro. Los registros pueden consistir en una lista de ocurrencias de cada término. En la figura 10.15 se muestran cuatro bloques del conjunto de secuencias que se han escrito en disco y un bloque del conjunto índice que se ha construido en memoria RAM a partir de los separadores más cortos derivados de las llaves del bloque del conjunto de secuencias. Como puede observarse, el siguiente bloque del conjunto se secuencias consiste en un conjunto de términos que van de CATCH a CHECK y, por lo tanto, el siguiente

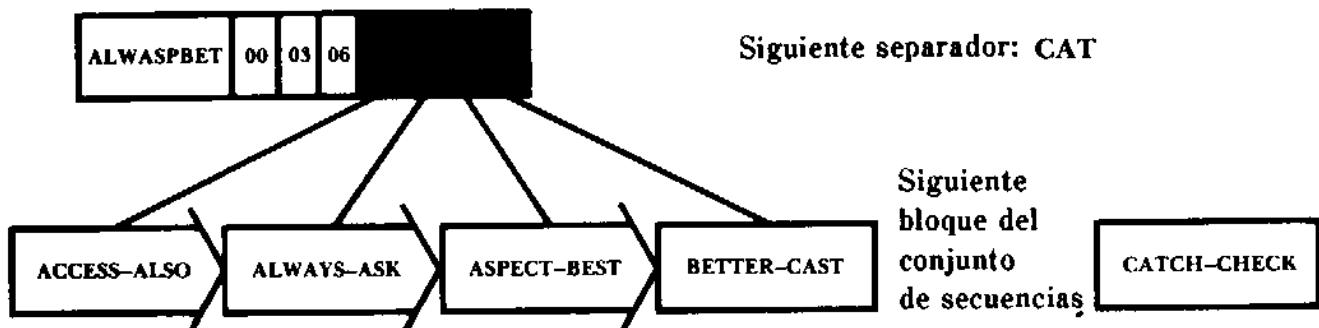


FIGURA 10.15• Formación de los primeros bloques del conjunto índice conforme se carga el conjunto de secuencias.

separador es CAT. Supóngase que el bloque del conjunto índice está completo y se escribe en disco. ¿Qué se hace ahora con el separador CAT?

Está claro que es necesario iniciar un nuevo bloque de índice. Pero no puede colocarse CAT dentro de otro bloque de índice en el mismo nivel que el que contiene los separadores ALW, ASP y BET, puesto que no puede haber dos bloques en el mismo nivel sin que tengan un bloque padre. En lugar de ello se promueve el separador CAT a un bloque de mayor nivel. Sin embargo, el bloque de nivel superior no puede apuntar directamente al conjunto de secuencias: debe apuntar a los bloques de índice del nivel inferior. Esto significa que se estarán construyendo *dos niveles* del conjunto índice en memoria RAM conforme se construye el conjunto de secuencias. La figura 10.16 ilustra este fenómeno de

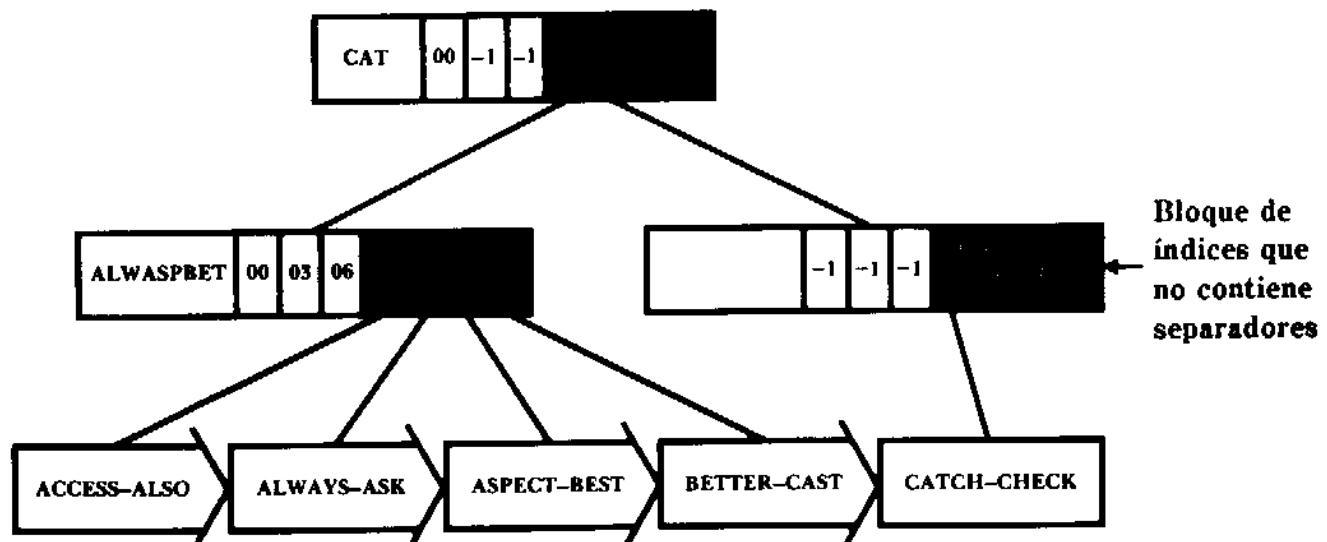


FIGURA 10.16• Construcción simultánea de dos niveles de conjunto índice a la vez que el conjunto de secuencias continúa creciendo.

trabajar en dos niveles: la adición del separador CAT requiere que se inicie un nuevo bloque de índice en el nivel de la raíz, así como un bloque de índice de nivel inferior. (En realidad, se está trabajando en tres niveles a la vez, ya que los bloques del conjunto de secuencias también se construyen en memoria RAM.) La figura 10.17 muestra el índice después de que se agregan aún más bloques al conjunto de secuencias. Como puede observarse, el bloque de índices de nivel más bajo que no contenía separadores cuando se agregó CAT a la raíz se ha llenado

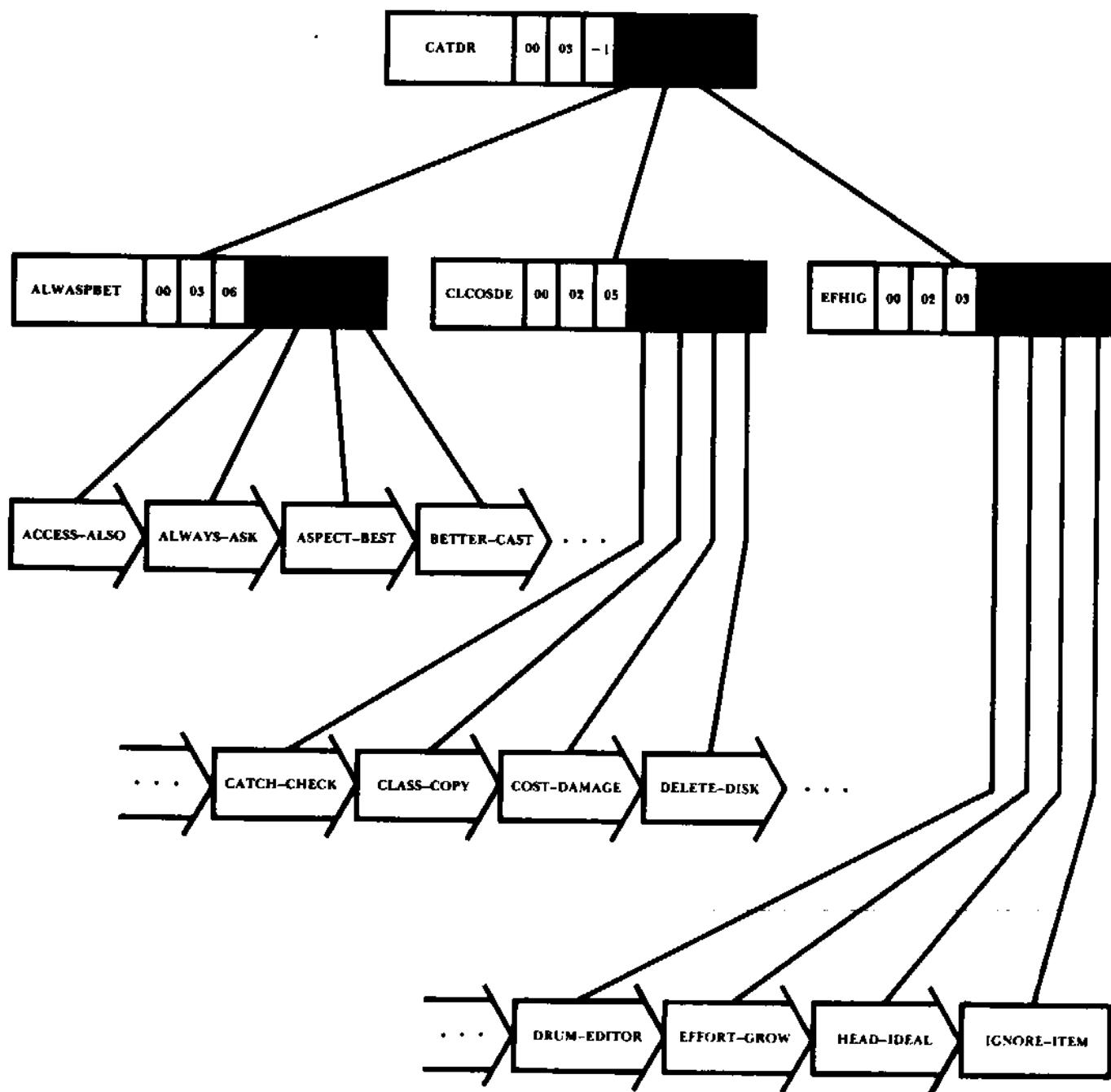


FIGURA 10.17 • Continuación del crecimiento del conjunto índice que se construye a partir del conjunto de secuencias.

ahora. Para confirmar que el árbol funciona, haga una búsqueda para el término CATCH. Después busque los términos CASUAL y CATALOG. ¿Cómo se puede decir que estos términos no están en el conjunto de secuencias?

Es instructivo preguntarse qué sucedería si el último registro fuera CHECK, de tal forma que la construcción de los conjuntos de secuencias y los conjuntos índice terminaran con la configuración mostrada en la figura 10.16. El árbol B⁺ de prefijos simples resultante tendría un nodo de conjunto índice sin separadores. Esta no es una posibilidad aislada, que ocurra sólo una vez. Si se usa este método de carga secuencial para construir el árbol, habrá muchos puntos durante el proceso de carga en los cuales exista un nodo vacío o casi vacío del conjunto índice. Si el conjunto índice crece a más de dos niveles, este problema de nodos vacíos puede ocurrir en niveles aun superiores del árbol creando un severo problema potencial de desequilibrio. Está claro que estas ocurrencias de nodo vacío o casi vacío violan las reglas de árboles B que se aplican al conjunto índice. Sin embargo, cuando el árbol está cargado y tiene un uso regular, el hecho mismo de que un nodo esté violando las condiciones de los árboles B puede usarse para garantizar que el nodo se corregirá por medio de la acción de las operaciones normales de mantenimiento de árboles B. Es fácil escribir los procedimientos de inserción y eliminación para que se efectúe una llamada a un procedimiento de redistribución cuando se encuentre un nodo con insuficiencia.

Casi siempre hay más ventajas al cargar un árbol B⁺ de prefijos simples en esta forma, como una operación secuencial después de una clasificación de registros, que las desventajas asociadas con la posibilidad de crear bloques que contengan muy pocos registros o muy pocos separadores. La principal ventaja es que el proceso de carga es más rápido, ya que:

- La salida puede escribirse en forma secuencial;
- Se realiza sólo un paso sobre los datos, en lugar de los numerosos pasos asociados con las inserciones aleatorias, y
- No es necesario reorganizar ningún bloque conforme se avanza.

Existen dos ventajas más del uso de un proceso de carga separado como el que se ha descrito, y están relacionadas con el desempeño después de cargar el árbol, y no tanto durante la carga:

- La inserción aleatoria produce bloques llenos en promedio de 67 a 80 por ciento. En el capítulo anterior, al analizar los árboles B, se incrementó la utilización del almacenamiento mediante mecanismos tales como el uso de la redistribución durante la

inserción, en lugar del uso de la división de bloques solamente. Pero, no obstante, nunca se tuvo la opción de llenar bloques por completo, de modo que se utilizará el 100 por ciento. El proceso de carga secuencial cambia esto. Si se desea, puede cargarse el árbol de modo que se inicie con el 100 por ciento de utilización. Esta es una opción atractiva si no se pretende agregar muchos registros al árbol. Por otro lado, si se anticipan muchas inserciones, la carga secuencial permite seleccionar cualquier otro grado de utilización que se desee. La carga secuencial proporciona mucho más control sobre la cantidad y colocación del espacio vacío en el árbol recién cargado.

- En el ejemplo de carga que se presenta en la figura 10.16, se escriben los primeros cuatro bloques del conjunto de secuencias y después el bloque del conjunto índice que contiene los separadores para estos bloques del conjunto de secuencias. Si se usa el mismo archivo para los bloques del conjunto se secuencias y los del conjunto índice, se garantiza que un bloque del conjunto índice se inicie en *proximidad física* con los bloques del conjunto de secuencias que son sus descendientes. En otras palabras, el proceso de la carga secuencial crea una medida de *localidad espacial* dentro del archivo. Esto puede minimizar los desplazamientos cuando se realiza una búsqueda hacia abajo en el árbol.

10.10 ARBOLES B⁺

Hasta este punto, los análisis se han centrado en los árboles B⁺ de prefijos simples. Estas estructuras son, en realidad, una variante de un método de organización de archivos conocida simplemente como *árbol B⁺*. La diferencia entre un árbol B⁺ de prefijos simples y un árbol B⁺ sencillo es que esta última estructura no implica el uso de prefijos como separadores, sino que los separadores del conjunto índice son simplemente copias de las llaves reales. El bloque del conjunto índice que se muestra en la figura 10.18 y que ilustra los pasos iniciales de carga para un árbol B⁺ contrasta con el que se ilustra en la figura 10.15, donde se muestra la construcción de un árbol B⁺ de prefijos simples.

Las operaciones efectuadas sobre árboles B⁺ son, en esencia, las mismas que las de los árboles B⁺ de prefijos simples. Tanto los árboles B⁺ como los árboles B⁺ de prefijos simples consisten en un conjunto de registros que se acomodan en el orden de las llaves en un conjunto de secuencias, y se acoplan con un conjunto índice que proporciona acceso rápido al bloque que contiene alguna combinación llave/registro en particular. La única diferencia es que en el árbol B⁺ de prefijos

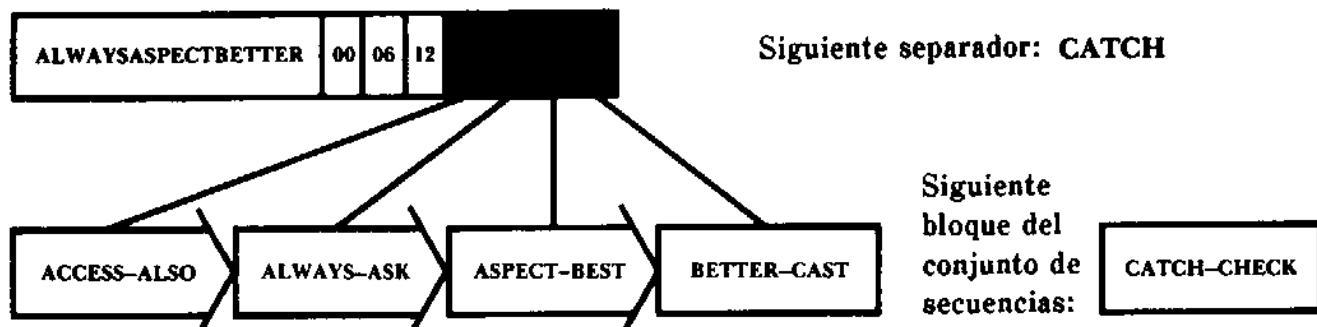


FIGURA 10.18 • Formación del primer bloque del conjunto índice en un árbol B* sin usar los separadores más cortos.

simples se construye un conjunto índice con los separadores más cortos, formados a partir de los prefijos de las llaves.

Unas de las razones que apoyan la decisión de centrarse primero en los árboles B* de prefijos simples, y no en la idea más general de un árbol B*, es que se pretende distinguir entre el papel de los *separadores* del conjunto índice y las *llaves* en el conjunto de secuencias. Es mucho más difícil hacer esta distinción cuando los separadores son copias exactas de las llaves. Comenzar con los árboles B* de prefijos simples tiene la ventaja pedagógica de trabajar con separadores que son claramente distintos de las llaves en el conjunto de secuencias.

Pero otra razón para comenzar con los árboles B* de prefijos simples se relaciona con el hecho de que con mucha frecuencia son una mejor alternativa que los árboles B*. Se quiere un conjunto índice que sea lo menos profundo posible, lo cual implica que se desea colocar tantos separadores dentro del bloque del conjunto índice como se pueda. ¿Por qué usar algo más grande que el prefijo simple en el conjunto índice? En general, la respuesta a esta pregunta es que, de hecho, no se quiere usar algo más grande que un prefijo simple como separador y en consecuencia, los árboles B* de prefijos simples suelen ser una buena solución. Sin embargo, existen al menos dos factores que pueden argüirse en favor del uso de un árbol B* que usa copias completas de llaves como separadores:

- La razón de usar separadores más cortos es empacar más de ellos dentro de un bloque del conjunto índice. Como ya se había dicho, esto implica ineludiblemente el uso de campos de longitud variable dentro de los bloques del conjunto índice. Para algunas aplicaciones, el gasto adicional requerido para mantener y usar esta estructura de longitud variable es mayor que los beneficios de los separadores más cortos. En estos casos se puede elegir

construir un árbol B⁺ directamente, usando como separadores copias de longitud fija de las llaves, a partir de los conjuntos de secuencias.

- Con algunos conjuntos de llaves no se logra mucha compresión cuando se usa el método de prefijos simples para producir separadores. Por ejemplo, supóngase que las llaves consisten en grandes secuencias alfanuméricas consecutivas, como C34C18K756, C34C18K757, C34C18K758, y así sucesivamente. En este caso, para obtener una compresión apreciable es necesario usar técnicas que eliminen la redundancia al inicio de la llave. Bayer y Unterauer [1977] describen tales métodos de compresión. Por desgracia, son más costosos y complicados que la compresión de prefijos simples. Si se determina que la altura del árbol continúa siendo aceptable con el uso de copias completas de las llaves como separadores, puede elegirse la opción de no usar compresión.

10. II

LOS ARBOLES B, B⁺ Y B⁺ DE PREFIJOS SIMPLES EN PERSPECTIVA

En este capítulo y en el anterior se han examinado varias "herramientas" utilizadas en la construcción de estructuras de archivos. Estas herramientas, árboles B, árboles B⁺ y árboles B⁺ de prefijos simples, tienen nombres parecidos y varias características en común. Es necesaria una forma de diferenciar estas herramientas de modo que pueda elegirse en forma confiable la más apropiada para un determinado trabajo de estructuras de archivos.

Sin embargo, antes de plantear este problema de diferenciación, debe señalarse que éstas no son las únicas herramientas de que se dispone. Los árboles B, los árboles B⁺ y sus familias son estructuras de archivos tan poderosas y flexibles que es fácil caer en la trampa de considerarlos como la solución a todos los problemas. Este es un error serio. Las estructuras simples de índices del tipo analizado en el capítulo 7, que se mantienen por completo en memoria RAM, son una solución mucho más sencilla y más limpia cuando son suficientes para el trabajo en cuestión. Como se observó al inicio de este capítulo, los índices simples en memoria RAM no están limitados a situaciones de acceso directo. Ese tipo de índices puede acoplarse con un conjunto de secuencias de bloques para proporcionar un acceso secuencial indizado efectivo. Sólo cuando el índice crece tanto que no puede mantenerse en forma económica en la memoria RAM es que se necesita ir a las estructuras de índices paginadas tales como los árboles B y los árboles B⁺.

En el siguiente capítulo aparece aún otra herramienta, conocida como *dispersión*. Al igual que los índices simples basados en la memoria RAM, la dispersión es una alternativa importante a los árboles B, B⁺, etc. En muchas situaciones, la dispersión puede proporcionar acceso más rápido a un número grande de registros que el que brindaría el uso de algún miembro de la familia de los árboles B.

Es decir, los árboles B, B⁺ y los árboles B⁺⁺ de prefijos simples no son la panacea. Sin embargo, tienen amplia aplicabilidad, en particular en situaciones en que es necesario acceder a un archivo grande tanto en forma secuencial, en orden por llave, como a través de un índice. Estas tres herramientas diferentes comparten las siguientes características:

- Todas son estructuras paginadas de índices, lo que significa que llevan bloques completos de información a la memoria RAM a la vez. Por consiguiente, es posible elegir entre gran cantidad de alternativas (p. ej., las llaves para cientos de miles de registros) con sólo unos cuantos desplazamientos en el almacenamiento en disco. La forma de estos árboles tiende a ser amplia y de poca altura.
- Los tres métodos mantienen árboles de altura. Es decir, los árboles no crecen en forma disparaja, lo que ocasionaría búsquedas potencialmente largas para algunas llaves.
- En todos los casos, los árboles crecen de abajo hacia arriba. El balance se mantiene por medio de división de bloques, concatenación y redistribución.
- Con las tres estructuras es posible obtener mayor eficiencia de almacenamiento usando división de dos a tres y redistribución en lugar de la división de bloques, cuando sea posible. Estas técnicas se describen en el capítulo 9.
- Los tres métodos pueden implantarse como estructuras de árbol virtuales, donde los bloques cuyo uso es más reciente se almacenan en memoria RAM. Las ventajas de los árboles virtuales se describen en el capítulo 9.
- Cualquiera de estos métodos se puede adaptar para usarse con registros de longitud variable, empleando estructuras dentro del bloque similares a las descritas en este capítulo.

Así como existen similitudes, existen algunas diferencias importantes, que se manifiestan al revisar las ventajas y características únicas de cada una de estas tres estructuras de archivos:

Arboles B. Los árboles B contienen información que se agrupa como un conjunto de *parejas*. Un miembro de cada pareja es la *llave* y el otro es la *información asociada*. Estas parejas se distribuyen en todos los nodos del árbol B. En consecuencia, puede

encontrarse la información que se busca en cualquier nivel del árbol B. Esto difiere de los árboles B⁺ y de los árboles B⁺ de prefijos simples, en los cuales se requiere que las búsquedas lleguen hasta abajo, al nivel inferior del conjunto de secuencias del árbol. Puesto que el árbol B mismo contiene las llaves reales y la información asociada, y por tanto no se necesita espacio adicional para almacenar separadores, un árbol B puede ocupar menos espacio que un árbol B⁺.

Con un tamaño de bloque lo suficientemente grande y un programa que trate al árbol como un árbol B virtual, es posible usar un árbol B para el acceso en orden secuencial, así como para el acceso indizado. El acceso secuencial ordenado se obtiene por medio de un recorrido en el orden del árbol. La realización como árbol virtual es necesaria para que este recorrido no implique desplazamientos cuando regrese el siguiente nivel superior del árbol. Este uso de un árbol B para acceso secuencial indizado funciona sólo cuando la información del registro se almacena realmente dentro del árbol B. Si el árbol B tan sólo contiene apuntadores a los registros que están en secuencia de entrada en algún otro archivo, entonces el acceso secuencial indizado no es aplicable debido a los desplazamientos requeridos para extraer la información real del registro.

Los árboles B son más atractivos cuando la llave misma comprende gran parte de cada registro almacenado en el árbol. cuando la llave es sólo una pequeña parte del registro, es posible construir un árbol más ancho y de menor altura usando los métodos de árboles B⁺.

Arboles B⁺. La diferencia primaria entre los árboles B⁺ y los árboles B es que en el árbol B⁺ toda la información de las llaves y los registros está contenida en un conjunto ligado de bloques conocidos como *conjuntos de secuencias*. La información de la llave y el registro no está en la porción del nivel superior en forma de árbol del árbol B⁺. El acceso indizado a este conjunto de secuencias se proporciona por medio de una estructura conceptualmente separada (aunque no por fuerza en forma física) llamada el *conjunto índice*. En el árbol B⁺ el conjunto índice consiste en copias de las llaves que representan los límites entre los bloques del conjunto de secuencias. A estas copias de llaves se les llama *separadores* porque separan un bloque del conjunto de secuencias de su predecesor.

Existen dos ventajas significativas de la estructura de árboles B⁺ sobre la de los árboles B.

- El conjunto de secuencias puede procesarse en una forma verdaderamente lineal y secuencial, proporcionando acceso eficiente a los registros en el orden de la llave, y
- El uso de separadores en lugar de registros completos en el conjunto índice con frecuencia significa que el número de *separadores* que pueden colocarse en un solo bloque del conjunto índice en un árbol B⁺ excede en forma sustancial al número de *registros* que podrían colocarse en un bloque del mismo tamaño en un árbol B. Los separadores (copias de las llaves) sencillamente son más pequeños que las parejas llave/registro almacenadas en un árbol B. Puesto que pueden colocarse más separadores en un bloque de tamaño dado, se deduce que el número de los otros bloques descendientes de ése puede ser mayor. Como consecuencia, el método de árbol B⁺ con frecuencia puede producir un árbol de menor altura que el que daría el método de árbol B.

En la práctica, la última de estas dos ventajas suele ser la más importante. La importancia de la primera ventaja disminuye por el hecho de que muchas veces es posible obtener un desempeño aceptable durante un recorrido ordenado de un árbol B a través del mecanismo de manejo de páginas en buffers de un árbol B virtual.

Arboles B⁺ de prefijos simples. Ya se indicó que la principal ventaja de usar un árbol B⁺ en lugar de un árbol B es que los primeros, algunas veces permiten construir un árbol de menor altura porque pueden obtenerse más ramas en los bloques de nivel superior del árbol. El árbol B⁺ de prefijos simples aprovecha esto haciendo los separadores del conjunto índice *más pequeños* que las llaves del conjunto de secuencias, en lugar de usar sólo copias de estas llaves. Si los separadores son menores, entonces puede haber más en un bloque y obtener así un número aún más alto de ramas. En cierto sentido, el árbol B⁺ de prefijos simples hace que una de las características más fuertes de los árboles B⁺ sea aún mejor.

El precio que debe pagarse para obtener esta comprensión de separadores y el consecuente incremento en el número de ramas es que debe usarse una estructura de conjunto índice que maneje campos de longitud variable. La conveniencia de pagar este precio es una cuestión que debe considerarse caso por caso.

RESUMEN

Este capítulo comenzó con un nuevo problema. En los capítulos anteriores existe acceso indizado o secuencial por el orden de la llave, pero sin encontrar una forma eficiente de proporcionar ambos tipos de acceso. Este capítulo explora soluciones a este problema, basadas en el uso de un conjunto de secuencias en bloques y un conjunto índice asociado.

El conjunto de secuencias almacena todos los registros de datos del archivo ordenados por llave. Como todas las operaciones de inserción o eliminación en el archivo comienzan por modificar el conjunto de secuencias, se inicia el estudio de las estructuras de archivos secuenciales indizados con un método para el manejo de cambios en el conjunto de secuencias. Las herramientas fundamentales usadas para insertar y eliminar registros y mantener el orden dentro del conjunto de secuencias son las que se presentan en el capítulo 9: división de bloques, concatenación de bloques y redistribución de registros entre bloques. La diferencia crítica entre el uso dado a estas herramientas para árboles B y el uso dado aquí es que no existe promoción de registros o de llaves durante la división de bloques en un conjunto de secuencias. Un conjunto de secuencias es sólo una lista ligada de bloques, no un árbol, así que no hay nada que promover. De esta forma, cuando un bloque se divide, todos los registros se dividen entre bloques del mismo nivel; cuando los bloques se concatenan, no hay necesidad de llevar nada hacia abajo desde un nodo padre.

También se analiza en este capítulo la pregunta sobre el tamaño de los bloques del conjunto de secuencias. No se puede dar una respuesta precisa porque las condiciones varían entre aplicaciones y ambientes. En general un bloque debe ser grande, pero no tanto que no puedan almacenarse varios en la memoria RAM o que no pueda leerse uno sin incurrir en el costo de un desplazamiento. En la práctica, con frecuencia los bloques son del tamaño de un cúmulo (en discos cuyo formato está dado por sectores) o del tamaño de una sola pista del disco.

Cuando ya se puede construir y mantener un conjunto de secuencias, se pasa a la cuestión de construir un índice para los bloques de dicho conjunto. Si el índice es lo suficientemente pequeño para caber en memoria RAM, una solución bastante satisfactoria es usar un índice sencillo que contenga, por ejemplo, la llave del último registro en cada bloque del conjunto de secuencias.

Si el índice se vuelve demasiado grande para entrar en la memoria RAM, es recomendable el uso de la misma estrategia que se desarrolló en el capítulo anterior, cuando el uso de un índice simple sobrepasa el

espacio disponible de memoria RAM: se convierte el índice en un árbol B. Esta combinación de un conjunto de secuencias con un conjunto índice en forma de árbol B es el primer encuentro con la estructura conocida como *árbol B⁺*.

Antes de examinar los árboles B⁺ como entidades completas, se analiza con mayor detalle la elaboración del conjunto índice. El conjunto índice no almacena información que se deseara buscar específicamente; más bien se usa sólo como un mapa para guiar las búsquedas en el conjunto de secuencias. El conjunto índice consiste en *separadores* que permiten elegir entre bloques del conjunto de secuencias. Existen muchos posibles separadores para dos bloques cualesquiera del conjunto de secuencias así que puede elegirse el *separador más corto*. El esquema que se usa para encontrar este separador más corto consiste en encontrar el prefijo común de las dos llaves en ambos lados de los límites del bloque en el conjunto de secuencias, y después tomar una letra más en el prefijo común para definir el separador. A un árbol B⁺ con un conjunto índice de separadores formados de esta manera se le llama *árbol B⁺ de prefijos simples*.

Se estudia el mecanismo usado para mantener el conjunto índice conforme se hacen inserciones y eliminaciones en el conjunto de secuencias de un árbol B⁺. La observación principal que se hace acerca de estas operaciones es que la acción primaria se lleva a cabo dentro del *conjunto de secuencias*, ya que es allí donde están los registros. Los cambios en el conjunto índice son secundarios; son un producto de las operaciones fundamentales sobre el conjunto de secuencias. Se agrega un nuevo separador al conjunto índice sólo si se forma un nuevo bloque en el conjunto de secuencias; se elimina un separador del conjunto índice sólo si se elimina un bloque del conjunto de secuencias por medio de la concatenación. La insuficiencia y la saturación en el conjunto índice difieren de las operaciones en el conjunto de secuencias en que el conjunto índice es en potencia una estructura de *varios niveles* y, por tanto, se maneja como un árbol B.

El tamaño de los bloques en el conjunto índice es por lo regular el mismo tamaño escogido para el conjunto de secuencias. Para crear bloques que contengan números variables de separadores de longitud variable y que al mismo tiempo permitan la búsqueda binaria, se desarrolló una estructura interna para el bloque que consiste en campos de encabezado de bloque (para el contador de separadores y la longitud total del separador), los separadores de longitud variable, un índice para ellas y un vector de números relativos de bloques (NRB) para los bloques que son descendientes del bloque del conjunto índice. Esto ilustra un importante principio general acerca de los bloques grandes dentro de estructuras de archivos: son más que una simple parte del conjunto homogéneo de registros; con frecuencia los bloques

tienen una estructura interna compleja aparte de la estructura total del archivo.

En seguida se pasa al problema de cargar un árbol B⁺. Se encuentra que si se comienza con un conjunto de registros clasificados por llave, puede usarse un proceso secuencial de una sola pasada para colocarlos dentro del conjunto de secuencias. Conforme se mueve de bloque a bloque en la construcción del conjunto de secuencias, pueden extraerse los separadores y construirse los bloques del conjunto índice. Comparado con una serie de inserciones sucesivas que se construyen hacia abajo desde el tope del árbol, este proceso de carga secuencial es mucho más eficiente. La carga secuencial también permite elegir el porcentaje de espacio utilizado, teniendo a un objetivo de 100 por ciento.

El capítulo termina con una comparación de los árboles B, B⁺ y B^{*} de prefijos simples. Las principales ventajas que los árboles B⁺ ofrecen sobre los árboles B son:

- Permiten un verdadero acceso secuencial indizado, y
- El conjunto índice contiene sólo separadores, en vez de llaves y registros completos, de modo que con frecuencia es posible crear un árbol B⁺ de menor altura que un árbol B.

Se sugiere que la segunda de estas ventajas suele ser la más importante, ya que tratar un árbol B como un árbol virtual proporciona acceso secuencial indizado aceptable en muchas circunstancias. El árbol B⁺ de prefijos simples toma esta segunda ventaja y la lleva más lejos, comprimiendo los separadores y produciendo en potencia un árbol de menor altura aún. El precio de esta compresión adicional en un árbol B⁺ de prefijos simples es que debe tratarse con campos de longitud variable y un árbol de orden variable.

TERMINOS CLAVE

Acceso secuencial indizado. El acceso secuencial indizado no es en realidad un método único de acceso; más bien es un término usado para describir situaciones en las que un usuario desea tanto acceso secuencial a los registros, clasificados por llave, como acceso indizado a estos mismos registros. Los árboles B⁺ son sólo un método para permitir el acceso secuencial indizado.

Árbol B⁺. Un árbol B⁺ consiste en un *conjunto de secuencias* de registros ordenados por llave en forma secuencial, junto con un *conjunto índice* que proporciona acceso indizado a los registros.

Todos los registros se almacenan en un conjunto de secuencias. Las inserciones y eliminaciones de registros se manejan por medio de división, concatenación y redistribución de bloques en el conjunto de secuencias. El conjunto índice, que se usa sólo como una ayuda para encontrar los bloques en el conjunto de secuencias, se maneja como un árbol B.

Arbol B⁺ de prefijos simples. Árbol B⁺ en el cual el conjunto índice está constituido por los *separadores más cortos* que son prefijos simples, como se describe en la definición del *separador más corto*.

Conjunto de secuencias. El conjunto de secuencias es el nivel básico de una estructura de archivo secuencial indexado, como un árbol B⁺, y contiene todos los registros del archivo. Cuando se lee en el orden lógico, bloque tras bloque, el conjunto de secuencias lista todos los registros por el orden de la llave.

Conjunto índice. El conjunto índice consiste en *separadores* que proporcionan información acerca de los límites entre bloques en el conjunto de secuencias de un árbol B⁺. El conjunto índice puede localizar el bloque del conjunto de secuencias que contiene el registro correspondiente a una cierta llave.

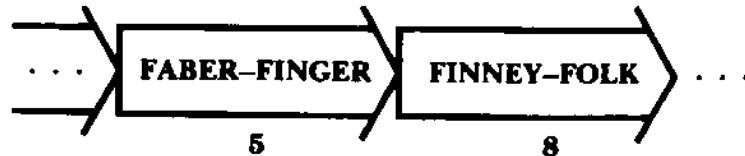
Orden variable. Un árbol B es de orden variable cuando el número de descendientes directos de cualquier nodo del árbol es variable. Esto ocurre cuando los nodos del árbol B contienen un número variable de llaves o de separadores. Esta forma es la que se usa con mayor frecuencia cuando existe variabilidad en las *longitudes* de las llaves o los separadores. Los árboles B⁺ de prefijos simples siempre hacen uso de un árbol B de orden variable como conjunto índice, de modo que es posible aprovechar la compresión de los separadores y colocar más de ellos en un bloque.

Separador. Los separadores se derivan de las llaves de los registros en cualquiera de los lados que limitan un bloque en el conjunto de secuencias. Si una determinada llave está en uno de los dos bloques en algún lado de un separador, el separador dice al usuario en forma confiable cuál de los dos bloques es el que almacena la llave.

Separador más corto. Pueden usarse muchos posibles separadores para distinguir entre dos bloques en el conjunto de secuencias. La clase de los separadores más cortos consiste en aquellos que ocupan el espacio mínimo para una estrategia de compresión en particular. Se examinó cuidadosamente una estrategia de compresión que consiste en eliminar tantas letras como sea posible del final de los separadores, formando el prefijo simple más corto que pueda seguir sirviendo como separador.

EJERCICIOS

1. Describa estructuras de archivo que permitan cada uno de los siguientes tipos de acceso: (a) sólo acceso secuencial; (b) sólo acceso directo; (c) acceso secuencial indizado.
2. La estructura de árbol B⁺ por lo general es mejor que un árbol B para el acceso secuencial indizado. Puesto que los árboles B⁺ incorporan árboles B, ¿por qué no emplear un árbol B⁺ siempre que se necesite una estructura indizada jerárquica?
3. Considere el conjunto de secuencias mostrado en la figura 10.1 (b). Indique cómo quedaría el conjunto de secuencias después de agregar las llaves DOVER y EARNEST, y después muestre cómo quedaría tras eliminar la llave DAVIS. ¿Usó concatenación o redistribución para manejar la insuficiencia?
4. ¿Qué consideraciones intervienen en la elección de un tamaño de bloque para construir un conjunto de secuencias? Si sabe algo acerca de los patrones de acceso esperados (principalmente secuencial, principalmente aleatorio, o bien una combinación equilibrada de ambos), ¿cómo podría esto influir en su elección del tamaño de bloque? En una unidad de disco organizada por sectores, ¿cómo afectarían las dimensiones de sectores y cúmulos su elección del tamaño del bloque?
5. Es posible construir un archivo secuencial indizado sin emplear un índice estructurado en forma de árbol. Podría usarse un índice simple como el que se desarrolló en el capítulo 7. ¿En qué condiciones se puede considerar el uso de dicho índice? ¿En qué condiciones puede ser razonable usar un árbol binario (tal como un árbol AVL) en lugar de un árbol B para el índice?
6. El conjunto índice de un árbol B⁺ es sólo un árbol B, pero, a diferencia de los árboles B analizados en el capítulo 9, los separadores no tienen que ser llaves. ¿Por qué la diferencia?
7. ¿En qué difieren la división de bloques en el conjunto se secuencias de un árbol B⁺ de prefijos simples y la división de bloques en el conjunto índice?
8. Si en el árbol B⁺ de prefijos simples de la figura 10.8 se elimina la llave BOLEN del nodo del conjunto de secuencias ¿qué sucede con el separador BO del nodo padre?
9. Considere el árbol B⁺ de prefijos simples mostrado en la figura 10.8. Suponga que al agregar una llave al bloque 5 resulta una división del bloque 5 y la consecuente adición del bloque 8, de manera que los bloques 5 y 8 aparecen como sigue:



- a) ¿Cómo se ve el árbol después de la inserción?
 - b) Suponga que, tras la inserción, una eliminación provoca insuficiencia y la consecuente concatenación de los bloques 4 y 5. ¿Cómo se ve el árbol después de la eliminación?
 - c) Describa un caso en el que una eliminación ocasiona redistribución, en lugar de concatenación, y muestre el efecto que esto tiene en el árbol.
- 10.** ¿Por qué muchas veces es conveniente usar el mismo tamaño de bloque para el conjunto índice y el conjunto de secuencias en un árbol B⁺ de prefijos simples? ¿Por qué generalmente los nodos del conjunto índice y los del conjunto de secuencias deben mantenerse en el mismo archivo?
- 11.** Muestre una imagen conceptual de un bloque del conjunto índice, parecida a la ilustrada en la figura 10.12, que se carga con los separadores

Ab Arch Astron B Bea

También muestre una imagen más detallada del bloque del índice, como se ilustra en la figura 10.13.

- 12.** Si el conjunto inicial de registros se clasifica por llave, el proceso de carga de un árbol B⁺ puede manejarse por medio de un proceso secuencial de una pasada, en lugar de insertar en forma aleatoria registros nuevos dentro del árbol. ¿Cuáles son las ventajas de este método?
- 13.** Muestre cómo cambia el árbol B⁺ de prefijos simples de la figura 10.17 después de que se añade el nodo

ITEMIZE-JAR

Suponga que el nodo del conjunto índice que contiene los separadores EF, H e IG no tiene espacio para el nuevo separador, pero que hay lugar en la raíz.

- 14.** Use los datos almacenados en el árbol B⁺ de prefijos simples de la figura 10.17 para construir un árbol B⁺. Suponga que el conjunto índice del árbol B⁺ es de orden cuatro. Compare el árbol B⁺ resultante con el árbol B⁺ de prefijos simples.
- 15.** El uso de separadores de longitud variable o de compresión de llaves cambia algunas de las reglas acerca de cómo definir y usar un árbol B y cómo medir su desempeño.

- a) ¿Cómo afecta esto la definición dada en este texto acerca del orden de un árbol B?

- b) Sugiera criterios para decidir cuándo deben efectuarse la división, la concatenación y la redistribución.
- c) ¿Qué dificultades aparecen en la estimación de la altura, el número máximo de accesos y el espacio de un árbol B⁺ de prefijos simples?

16. Elabore una tabla en la que compare los árboles B, B⁺ y B⁺ de prefijos simples en términos de los criterios listados más adelante. Suponga que los nodos del árbol B no contienen registros de datos, sino sólo llaves y los correspondientes NRR de los registros de datos. En algunos casos será posible dar respuestas específicas basadas en la altura de un árbol o su número de llaves. En otros casos las respuestas dependerán de factores desconocidos, como los patrones de acceso o la longitud media de los separadores.

- a) El número de accesos requeridos para extraer un registro de un árbol de altura h (caso medio, mejor caso y peor caso).
- b) El número de accesos requeridos para insertar un registro (mejor y peor casos).
- c) El número de accesos requeridos para eliminar un registro (mejor y peor casos).
- d) El número de accesos requeridos para procesar un archivo de n llaves en forma secuencial, suponiendo que cada nodo puede almacenar un máximo de k llaves y un mínimo de $k/2$ (mejor y peor casos).
- e) El número de accesos requeridos para procesar un archivo de n llaves en forma secuencial suponiendo que hay $h + 1$ buffers disponibles del tamaño de un nodo.

17. Algunas organizaciones de archivos indizados secuenciales disponibles comercialmente están basadas en métodos de división del intervalo del bloque, muy parecidos a los que se usan con los árboles B⁺. VSAM de IBM ofrece al usuario varios modos de acceso, uno de los cuales se llama acceso por *llave en secuencia* y da como resultado un archivo que se organiza en forma similar a un árbol B⁺. Examine una descripción de VSAM y diga cómo se relaciona su organización de llaves en secuencia con un árbol B⁺, y también cómo ofrece al usuario mayores posibilidades de manejo de archivos que una realización directa de un árbol B⁺. (Véase la sección de lecturas adicionales de este capítulo para encontrar artículos y libros sobre VSAM.)

18. Aunque los árboles B⁺ proporcionan las bases para la mayoría de los métodos de acceso secuencial indizado que se usan ahora, esto no siempre fue así. Un método llamado ISAM (véanse las lecturas adicionales de este capítulo) alguna vez fue muy común, en especial en computadores grandes. ISAM usa un índice rígido estructurado en forma de árbol que consiste en un mínimo de dos y uno máximo de tres niveles. Los índices en estos niveles están ajustados a la unidad de disco

específica que se está usando. Los registros de datos están organizados por pistas, de tal forma que al nivel más bajo de un índice ISAM se llama *índice de pistas*. Como el índice de pistas apunta a la pista en la que puede encontrarse el registro de datos, existe un índice de pistas para cada cilindro. Cuando la adición de registros provoca que se sature una pista, ésta no se divide, sino que los registros adicionales se colocan dentro de un área de saturación separada y se encadenan en orden lógico. Por tanto, cada entrada en un índice de pistas puede contener un apuntador al área de saturación, además de su apuntador a la pista correspondiente.

La diferencia esencial entre la organización ISAM y las organizaciones tipo árbol B⁺ radica en la forma en que se manejan los registros saturados. En el caso de ISAM, los registros saturados simplemente se agregan a una cadena de registros saturados: la estructura del índice no se altera. En el caso de los árboles B⁺ no se permiten registros saturados; cuando esto ocurre, el bloque se divide y se altera la estructura del índice para acomodar el bloque de datos adicional.

¿Puede el lector imaginar alguna ventaja de la estructura de índices de ISAM, que es más rígida, con áreas de saturación separadas para manejar los registros saturados? ¿Por qué será que los métodos tipo árbol B⁺ están reemplazando a los que usan cadenas de saturación para almacenar registros saturados? Considere los dos métodos en términos del acceso tanto secuencial como directo, y también de la adición y eliminación de registros.

EJERCICIOS DE PROGRAMACION

[Este capítulo se inició con el análisis de las operaciones sobre un conjunto de secuencias, que no es sino una lista ligada de bloques que contienen registros. Sólo después se agrega el concepto de conjunto índice para proporcionar acceso más rápido a los bloques en el conjunto de secuencias. Los siguientes problemas de programación hacen eco de este enfoque, y requieren que primero se escriba un programa que construya un conjunto de secuencias, después las funciones que mantengan dicho conjunto, y finalmente los programas y funciones para agregar un conjunto índice al conjunto de secuencias, creando un árbol B⁺. Estos programas pueden realizarse en C o en Pascal.]

19. Escriba un programa que lea un archivo de cadenas como entrada. El archivo de entrada debe clasificarse de tal forma que las cadenas estén en orden ascendente. El programa debe usar este archivo de entrada para construir un conjunto de secuencias con las siguientes características:

- Las cadenas se almacenan en registros de 15 bytes;
- El bloque del conjunto de secuencias es de 128 bytes de largo;
- Los bloques del conjunto de secuencias están doblemente ligados;
- El primer bloque del archivo de salida es un bloque de encabezado que contiene, entre otras cosas, una referencia al NRR del primer bloque del conjunto de secuencias;
- Los bloques del conjunto de secuencias se cargan de tal forma que estén tan llenos como sea posible, y
- Los bloques de conjunto de secuencias contienen otros campos (además de los registros reales que contienen las cadenas); según sea necesario.

20. Escriba un programa de actualización que lea cadenas a partir del teclado, junto con una instrucción para buscar, agregar o eliminar la cadena del conjunto de secuencias. El programa debe tener las siguientes características:

- Las cadenas del conjunto de secuencias deben mantenerse ordenadas, por supuesto;
- La respuesta a la instrucción de búsqueda debe ser de encontrado o no encontrado;
- Una cadena no debe agregarse si ya existe en el conjunto de secuencias;
- Los bloques del conjunto de secuencias nunca deben estar llenos a menos de la mitad, y
- Las operaciones de división, redistribución y concatenación deben escribirse como procedimientos separados, de modo que puedan usarse en el subsecuente desarrollo de programas.

21. Escriba un programa que recorra el conjunto de secuencias creado en los ejercicios anteriores y que construya un conjunto índice en forma de árbol B. Puede suponerse que el índice en el árbol B nunca tendrá más de dos niveles de profundidad. El archivo resultante debe tener las siguiente características:

- El conjunto índice y el conjunto de secuencias deben constituir, juntos un árbol B⁺,
- No deben comprimirse las llaves cuando se formen los separadores del conjunto índice;
- Los bloques del conjunto índice, como los del conjunto de secuencias, deben ser de 128 bytes, y
- Los bloques del conjunto índice deben mantenerse en el mismo archivo que los bloques del conjunto de secuencias. El bloque de encabezado debe contener una referencia a la raíz del conjunto

índice, así como a la referencia ya existente al inicio del conjunto de secuencias.

22. Escriba una nueva versión del programa de actualización que actúe sobre el árbol B⁺ completo que se creó en el ejercicio anterior. Debe manejar búsqueda, agregación y eliminación, como el programa anterior de actualización. Las características de árbol B deben mantenerse en el conjunto índice; el conjunto de secuencias, como antes, debe mantenerse de tal forma que los bloques estén siempre al menos llenos a la mitad.

23. Considere la estructura de bloques ilustrada en la figura 10.13, en la cual se usa un índice para los separadores a fin de permitir la búsqueda binaria de una llave en una página del índice. Cada bloque del conjunto índice contiene tres conjuntos de longitud variable: un conjunto de separadores, un índice a los separadores y un conjunto de números de bloque relativo. Desarrolle código en Pascal o en C para almacenar estos datos en un bloque de índice y para buscar un separador en el bloque será necesario contestar preguntas como:

- ¿Dónde deben colocarse los tres conjuntos, uno respecto a otro?
- Dados los tipos de datos permitidos por el lenguaje que se esté usando, ¿cómo puede manejarse el hecho de que el bloque consista tanto en datos de caracteres como en enteros sin ningún punto fijo de división entre ellos?
- Al agregar datos a un bloque, ¿cómo se decide cuándo está demasiado lleno para insertar otro separador?

LECTURAS ADICIONALES

La sugerencia inicial para la estructura de árbol B⁺ aparentemente proviene de Knuth [1973b], aunque él no nombró ni desarrolló el método. Gran parte de la literatura que analiza los árboles B⁺ en detalle (por oposición a la que describe implantaciones específicas tales como VSAM) consiste en artículos, más que en libros de texto. Comer [1979] proporciona lo que quizás constituye el mejor repaso breve sobre árboles B⁺. Bayer y Unterauer [1977] ofrecen un autorizado artículo que describe técnicas para la compresión de separadores. El artículo incluye la consideración de los árboles B⁺ de prefijos simples, así como un método más general llamado *árbol B⁺ de prefijos*. McCreight [1977] describe un algoritmo para aprovechar la variación de las longitudes de los separadores en el conjunto índice de un árbol B⁺. El algoritmo de McCreight intenta

asegurar que se promuevan los separadores cortos, en vez de lo más largos, conforme se dividen los bloques. Con ello se pretende dar al árbol una forma tal que los bloques superiores tengan un número mayor de descendientes inmediatos que, por tanto, se cree un árbol de menor altura.

Rosenberg y Snyder [1981] estudian los efectos de asignar valores iniciales a un árbol B compacto en las inserciones y eliminaciones posteriores. El uso de inserciones y eliminaciones por lote para árboles B, en vez de actualizaciones individuales, se propone y analiza en Lang *et al* [1985]. En Batory [1981] y en *VSAM Planning Guide* de IBM se comparan los árboles B⁺ con organizaciones indizadas secuenciales más rígidas (tales como ISAM).

Existen muchos productos comerciales que usan métodos relacionados con las operaciones de árbol B⁺ que se describen en este capítulo, pero las descripciones detalladas de sus estructuras de archivos son escasas. Una excepción es el método de acceso de almacenamiento virtual (VSAM) de IBM, uno de los productos comerciales más ampliamente usados que proporcionan acceso secuencial indizado. Wagner [1973] y Keehn y Lacy [1974] proporcionan interesantes observaciones sobre las ideas originales que generaron VSAM. También incluyen consideraciones sobre el mantenimiento de llaves, comprensión de llaves, índices secundarios, e índices para conjuntos de datos múltiples. Pueden encontrarse buenas descripciones de VSAM en diversas fuentes, y con distintas perspectivas, en *VSAM Planning Guide* de IBM, Bohl [1981], Comer [1979] (VSAM como ejemplo de un árbol B⁺), Bradley [1981] (hace énfasis en la implantación en un ambiente PL/I), y Loomis [1983] (con ejemplos en COBOL).

Los servicios de manejo de registros de VAX-11 (RMS), subsistema de acceso a archivos y registros de Digital, del sistema operativo VAX/VMS, emplean una estructura tipo árbol B⁺ para manejar el acceso indizado secuencial (Digital, 1979). Pueden encontrarse muchas implantaciones de árboles B⁺ en microcomputadores, entre ellas dBase III y Turbo Toolbox de Borland (Borland, 1984).

OBJETIVOS

Introducir el concepto de *dispersión*.

Examinar el problema de la elección de un buen *algoritmo de dispersión*, presentar uno razonable con detalle y describir algunos otros.

Explorar tres métodos de *reducción de colisiones*: aleatorización de direcciones, uso de memoria adicional y almacenamiento de varios registros por dirección.

Desarrollar y usar herramientas matemáticas para analizar diferencias en el desempeño de diferentes técnicas de dispersión.

Examinar los problemas asociados con el *deterioro del archivo* y analizar algunas soluciones.

Examinar los efectos de los *patrones de acceso a registros* sobre el desempeño.

11

DISPERSION (HASHING)

PLAN GENERAL DEL CAPITULO

11.1 Introducción

11.1.1 ¿Qué es la dispersión?

11.1.2 Colisiones

11.2 Un algoritmo simple de dispersión

11.3 Funciones de dispersión y distribuciones de registros

11.3.1 Distribución de registros entre direcciones

11.3.2 Algunos otros métodos de dispersión

11.3.3 Predicción de la distribución de los registros

11.3.4 Predicción de las colisiones en un archivo lleno

11.4 ¿Cuánta memoria adicional debe usarse?

11.4.1 Densidad de empaquetamiento

11.4.2 Predicción de colisiones para diferentes densidades de empaquetamiento

11.5 Resolución de colisiones mediante saturación progresiva

11.5.1 Funcionamiento de la saturación progresiva

11.5.2 Longitud de búsqueda

11.6 Almacenamiento de más de un registro por dirección: compartimientos

11.6.1 Efectos de los compartimientos en el desempeño

11.6.2 Aspectos de la realización

11.7 La operación de eliminación

11.7.1 Marcas de inutilización para eliminaciones

11.7.2 Implicaciones de las marcas de inutilización para las inserciones

11.7.3 Efectos de las eliminaciones y adiciones en el desempeño

11.8 Otras técnicas de resolución de colisiones

11.8.1 Dispersión doble

11.8.2 Saturación progresiva encadenada

11.8.3 Encadenamiento con un área de saturación separada

11.8.4 Tablas de dispersión: reconsideración de la indización

11.8.5 Dispersión extensible

11.9 Patrones de acceso a los registros

11.1

INTRODUCCION

Ha pasado ya mucho tiempo desde que se consideró la posibilidad de tener realmente acceso directo a los registros. A medida que se ha incrementado la demanda de características y mejoras en el desempeño de las organizaciones de archivos estudiadas en el texto, se han ido encontrando métodos de acceso *indirecto*, en forma de diversos tipos de

índices, que pudieran satisfacer nuestras necesidades. Quizá se pueda afirmar que la indización, en sus diferentes formas, proporciona el conjunto de herramientas más poderoso para organizar archivos. Por supuesto, el uso de índices introduce ciertos costos adicionales: los índices ocupan espacio, toma tiempo mantenerlos organizados y, a menos que sean lo suficientemente pequeños para mantenerse en memoria principal, se requieren al menos dos accesos para encontrar un registro en almacenamiento secundario, y con frecuencia más de dos.

Por lo regular, los costos adicionales asociados con el uso de índices pueden reducirse lo suficiente para que sean justificables, pero hay ocasiones en que las demandas a un sistema de archivos son tan extremas que uno o más de estos costos se vuelven intolerables. Por ejemplo, supóngase que un archivo se usa exclusivamente para acceso aleatorio a registros por la llave principal (incluyendo extracciones, adiciones, eliminaciones y actualizaciones); que estas consultas ocurren a una tasa de 500 por minuto, y que el acceso a disco toma 100 msec en promedio. Un índice de dos niveles, con sólo el nodo raíz en memoria RAM, requiere dos accesos a disco por cada acceso a un registro, más, ocasionalmente, el tiempo adicional para reorganizar el índice. Por lo tanto, el acceso indizado requiere

$$2 \times 100 \text{ msec/acceso} \times 500 \text{ accesos} = 100\,000 \text{ msec} = 100 \text{ segundos}$$

para efectuar los accesos, sin considerar el tiempo necesario para la reorganización. Como no hay 100 segundos en un minuto, esto representa un problema real. A falta de un equipo o un sistema de software más rápido, resulta absolutamente necesaria una forma de acceso que requiera menos de dos accesos a disco en promedio por cada operación sobre un registro.

Es en tales situaciones cuando la necesidad de un acceso rápido a los registros sobrepasa todas las demás consideraciones de diseño. El acceso directo por llave, si puede llevarse a cabo, es el objetivo deseado. En gran medida, el método más efectivo usado para esto es una técnica llamada *dispersión*. En este capítulo se describen estos atributos de la dispersión:

- La dispersión, a diferencia de la indización, no requiere almacenamiento adicional para mantener un índice (aunque normalmente requiere algo de almacenamiento externo adicional);
- La dispersión facilita la inserción y eliminación muy rápidas de los registros, y
- La dispersión permite encontrar un registro con muy pocos accesos al disco en promedio, por lo regular menos de dos.

Por supuesto, estas drásticas mejoras en el desempeño no se obtienen gratis. No pueden usarse registros de longitud variable con la dispersión tan fácilmente como con la indización. Mientras que los archivos indizados pueden considerarse clasificados (aunque indirectamente, por medio del índice), los archivos dispersos, por definición, no están clasificados. La dispersión no permite llaves duplicadas. Finalmente, se verá que los archivos dispersos están organizados de acuerdo con una sola llave, de modo que sólo puede proporcionarse acceso por varias llaves imponiendo estructuras de archivo adicionales (tales como los índices) por encima de la dispersión.

11.1.1 ¿QUE ES LA DISPERSION?

Una función de dispersión, $h(K)$, transforma una llave K en una dirección. La dirección resultante se usa como base para la búsqueda y el almacenamiento de registros.

Una función de dispersión es como una caja negra en la que se deposita una llave y de la cual se extrae una dirección. En la figura 11.1, la función de dispersión transforma la llave LOWELL en la dirección 4. Esto es, $h(\text{LOWELL}) = 4$. Se dice que la dirección 4 es la dirección base de LOWELL.

La dispersión se asemeja a la indización en cuanto a que implica la asociación de una llave con una dirección relativa de registro, pero difiere de ella en dos aspectos importantes:

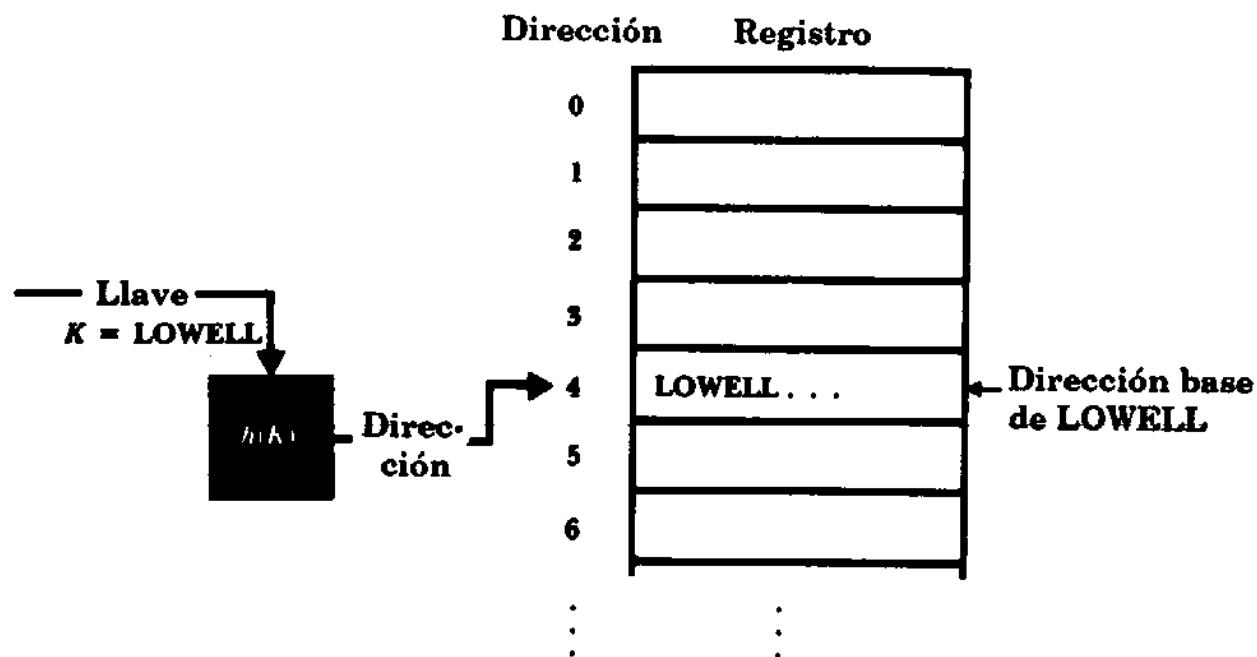


FIGURA 11.1• Dispersion de la llave LOWELL a la dirección 4.

- Con la dispersión, las direcciones generadas parecen ser aleatorias; no hay conexión obvia inmediata entre la llave y la localidad del registro correspondiente, aun cuando la llave se usa para determinar la colocación del registro. Por tal motivo, a la dispersión se le denomina algunas veces *aleatorización*.
- Con la dispersión, dos llaves diferentes pueden transformarse en la misma dirección, de modo que los dos registros pueden enviarse al mismo lugar del archivo. Cuando esto ocurre, se le llama *colisión*, y debe encontrarse un medio de resolverla.

Considérese el siguiente ejemplo sencillo. Supóngase que se quiere almacenar 75 registros en un archivo, donde la llave de cada registro es un nombre de persona. Supóngase también que se deja espacio para 1000 registros. La llave puede dispersarse multiplicando dos números de las representaciones ASCII de los primeros dos caracteres del nombre, y usando después los tres dígitos de la derecha del resultado para asignar la dirección. La tabla 11.1 muestra tres direcciones producidas por tres nombres. Nótese que, aunque los nombres se listan en orden alfabético, no existe un orden evidente en las direcciones. Parecen ser *aleatorias*.

11.1.2 COLISIONES

Ahora supóngase que hay una llave en el archivo del ejemplo con el nombre OLIVIER. Como el nombre OLIVIER comienza con las mismas dos letras que el nombre LOWELL, producirán la misma dirección (004). Hay una *colisión* entre el registro de OLIVIER y el de LOWELL. A las llaves que por dispersión se convierten en la misma dirección se les denomina *sinónimos*.

Las colisiones obviamente ocasionan problemas. No se puede colocar dos registros en el mismo espacio, de modo que es necesario

TABLA 11.1

Un esquema simple de dispersión

Nombre	Código ASCII de las primeras dos letras	Producto	Dirección base
BALL	66 65	$66 \times 65 = 4290$	290
LOWELL	76 79	$76 \times 79 = 6004$	004
TREE	84 82	$84 \times 82 = 6888$	888

enfrentar el problema de resolver las colisiones. Esto se logra en dos formas: eligiendo algoritmos de dispersión basados, en parte, en que produzcan el menor número posible de colisiones y en hacer algunos trucos con la forma en que se almacenan los registros.

Al principio se esperaría encontrar un algoritmo de transformación que evite por completo las colisiones, llamado *algoritmo de dispersión perfecta*. Sin embargo, encontrar un algoritmo de dispersión perfecta resulta mucho más difícil de lo que podría pensarse. Por ejemplo, supóngase que se quiere almacenar 4000 registros entre 5000 direcciones disponibles. Puede demostrarse (Hanson, 1982) que del enorme número de algoritmos de dispersión posibles para ello, sólo uno entre $10^{120,000}$ definitivamente evita todas las colisiones. Por lo tanto, en general no vale la pena intentarlo.†

Lo siguiente que se desea es reducir el número de colisiones a una cantidad aceptable. Por ejemplo, si sólo una de diez búsquedas de registros produce una colisión, entonces el número *promedio* de acceso a disco requerido para extraer un registro permanece bastante bajo. Existen diferentes formas de reducir el número de colisiones, entre ellas las tres siguientes.

- Espaciar los registros.* Las colisiones suceden cuando dos o más registros compiten por la misma dirección. Si se encontrara un algoritmo de dispersión que distribuyera los registros igualmente en forma aleatoria entre las direcciones disponibles, entonces no habría un gran número de registros acumulándose alrededor de ciertas direcciones. Nuestro ejemplo del algoritmo de dispersión, que usa sólo dos letras de la llave, no es bueno en este punto, porque algunas combinaciones de dos letras son bastante comunes al inicio de los nombres, mientras que otras son raras (p. ej., compárese el número de nombres que se inician con "JO" y el número de los que empiezan con "XZ"). Es necesario encontrar un algoritmo de dispersión que distribuya los registros en forma más aleatoria.
- Usar memoria adicional.* Es más fácil encontrar un algoritmo que evite colisiones si se tiene que distribuir sólo pocos registros entre muchas direcciones, que si se tiene casi el mismo número de registros que de direcciones. El algoritmo de dispersión del

† No es irracional intentar generar funciones de dispersión perfecta para conjuntos pequeños de llaves (menos de 500) y estables, tal como el que puede usarse para buscar palabras reservadas en un lenguaje de programación. Pero los archivos por lo general contienen más de unos cuantos cientos de llaves, o bien contienen conjuntos de llaves que cambian con frecuencia, por lo que normalmente no se consideran candidatos para funciones de dispersión perfecta. Para mayor información sobre las funciones de dispersión perfecta, véanse Knuth [1973b], Sager [1985], Chang [1984] y Chichelli [1980].

ejemplo es muy bueno en este punto, ya que existen 1000 direcciones posibles y sólo se generarán 75 (correspondientes a 75 registros). La desventaja obvia de esparcir los registros es que se desperdicia el espacio de almacenamiento. (En el ejemplo se usa el 7.5 por ciento del espacio disponible de registros, y se desperdicia el 92.5 por ciento restante.) No hay una sola respuesta para la pregunta sobre cuánto espacio vacío debe tolerarse a fin de tener el mejor desempeño de la dispersión, pero posteriormente en este capítulo se proporcionan algunas técnicas para medir las ganancias relativas en el desempeño con diferentes cantidades de espacio libre.

- *Colocar más de un registro en una sola dirección.* Hasta ahora se ha supuesto de manera tácita que cada localidad de registro físico en un archivo puede almacenar exactamente un registro, pero en general no hay razón para no poder crear el archivo en forma tal que todas sus direcciones sean lo bastante grandes para almacenar varios registros. Si, por ejemplo, cada registro es de 80 bytes y se crea un archivo con registros físicos de 512 bytes, pueden almacenarse hasta seis registros en cada dirección del archivo. Cada dirección puede tolerar cinco sinónimos. Las direcciones que pueden almacenar varios registros en esta forma se denominan *compartimientos*.

En las siguientes secciones se estudian más a fondo estos métodos de reducción de colisiones, y conforme se analicen se presentarán algunos programas para el manejo de archivos con dispersión.

11.2

UN ALGORITMO SIMPLE DE DISPERSION

Uno de los objetivos en la elección de cualquier algoritmo de dispersión debe ser esparcir los registros tan uniformemente como sea posible en el intervalo de direcciones disponible. El término con el que se designa esta técnica, *dispersión*, sugiere lo que se pretende lograr. El diccionario indica que el verbo *dispersar* significa "separar y diseminar lo que estaba o solía estar reunido."* El algoritmo usando anteriormente separa las dos primeras letras y después usa los códigos ASCII resultantes para producir un número que de nuevo se divide para producir la dirección. No es muy bueno para evitar la acumulación de

* En realidad, el término en inglés es *to hash*, que significa "dividir en partes pequeñas", lo cual, aunque parecido a "dispersar", no es exactamente igual. Se escogió este último término, sin embargo, porque nos parece una adecuada traducción en el entorno computacional. *N. del T.*

sinónimos, porque muchos nombres comienzan con las mismas dos letras. (Un conjunto diferente de llaves puede producir un conjunto satisfactorio de direcciones usando este algoritmo, pero parece ser demasiado simple para producir mucha aleatoriedad.)

El problema con el algoritmo usado antes es que en realidad no hace mucha dispersión. Por ejemplo, usa sólo dos letras de la llave, y no logra mucho con ellas. Ahora se examinará una función de dispersión que logra mucho más aleatoriedad, sobre todo porque usa más de la llave. Es un algoritmo básico razonablemente bueno, y suele dar buenos resultados sin importar qué tipos de llaves se usen. Además, no es demasiado difícil de alterar en caso de que una versión específica de ese algoritmo no funcione bien.

Este algoritmo tiene tres pasos:

1. Representar la llave en forma numérica.
2. Desglosar y sumar.
3. Dividir entre un número primo y usar el residuo como dirección.

PASO 1. REPRESENTAR LA LLAVE EN FORMA NUMERICA. Si la llave ya es un número, entonces este paso ya está hecho. Si es una cadena de caracteres, se usa el código ASCII de cada carácter para formar un número. Por ejemplo,

LOWELL =	76 79 87 69 76 76 32 32 32 32 32 32
	L O W E L L ; ← Espacios → :

En este algoritmo se usa toda la llave, y no sólo las dos primeras letras. Al usar más partes de una llave se incrementa la probabilidad de que las diferencias entre las llaves provoquen diferencias en las direcciones producidas. El tiempo de procesamiento adicional que se requiere por lo regular es insignificante cuando se compara con la mejora potencial en el desempeño.

PASO 2. DESGLOSAR Y SUMAR. *Desglosar y sumar* significa tomar pedazos del número y sumarlos. En este algoritmo se toman pedazos con dos números ASCII cada uno:

76 79 ! 87 69 ! 76 76 ! 32 32 ! 32 32 ! 32 32

Estas parejas de números pueden considerarse como variables enteras individuales (en vez de variables tipo carácter, que es como se comenzó), de tal forma que se puedan hacer operaciones aritméticas. Si se tratan como variables enteras, entonces pueden sumarse. Esto es fácil de lograr en C porque se permiten operaciones aritméticas con caracteres. En Pascal puede usarse la función *ord()* para obtener la posición en

forma de un número entero de un carácter dentro del conjunto de caracteres del computador.

Antes de sumar los números, debe mencionarse el problema de que en la mayoría de los casos los tamaños de los números que pueden sumarse están limitados. Por ejemplo, en algunos microcomputadores, los valores enteros que exceden de 32 767 (16 bits) provocan errores por desbordamiento. Por ejemplo, sumar los cinco primeros números que se mostraron anteriormente da

$$7679 + 8769 + 7676 + 3232 + 3232 = 30\ 588.$$

Sumar el último 3232 haría que, por desgracia, el resultado supere el máximo de 32 767 ($30\ 588 + 3232 = 33\ 820$), y provocara desbordamiento. Puesto que los errores por desbordamiento en general deben evitarse, es necesario asegurarse de que cada suma sucesiva sea menor de 32 767. Esto puede hacerse identificando primero el mayor valor individual que se vaya a agregar en la suma, y asegurándose después de cada paso que el resultado intermedio difiera de 32 767 por esa cantidad.

En este caso se supondrá que las llaves consisten sólo en espacios y caracteres alfabéticos en mayúsculas, de modo que el sumando más grande es 9 090, que corresponde a ZZ. Supóngase que se elige 20 000 como el resultado intermedio más grande que se permite. Esto difiere de 32 767 por mucho más de 9 090, de tal forma que puede confiarse (en este ejemplo) en que ninguna nueva adición provocará desbordamiento. Puede asegurarse en este algoritmo que ningún resultado intermedio exceda de 20 000 usando el operador *mod*, el cual devuelve el residuo cuando un entero se divide entre otro:

$$\begin{aligned} 7679 + 8769 &\rightarrow 16448 \rightarrow 16448 \bmod 20000 \rightarrow 16448 \\ 16448 + 7676 &\rightarrow 24124 \rightarrow 24124 \bmod 20000 \rightarrow 4124 \\ 4124 + 3232 &\rightarrow 7356 \rightarrow 7356 \bmod 20000 \rightarrow 7356 \\ 7356 + 3232 &\rightarrow 10588 \rightarrow 10588 \bmod 20000 \rightarrow 10588 \\ 10588 + 3232 &\rightarrow 13820 \rightarrow 13820 \bmod 20000 \rightarrow 13820 \end{aligned}$$

El número 13 820 es el resultado de la operación de desglosar y sumar.

PASO 3. DIVIDIR ENTRE EL TAMAÑO DEL ESPACIO DE DIRECCIONES. El propósito de este paso es recortar el número producido en el paso 2, para que esté dentro del intervalo de direcciones de registros en el archivo. Se puede hacer esto dividiendo ese número entre el tamaño de las direcciones del archivo; el residuo será la dirección base del registro.

Esta operación puede representarse en forma simbólica como sigue: si *s* representa la suma producida en el paso 2 (13 820 en el

ejemplo), n representa el divisor (el número de direcciones en el archivo), y a representa la dirección que se está intentando producir, se aplica la fórmula

$$a = s \bmod n.$$

El residuo producido por el operador mod será un número entre 0 y $n - 1$.

Por ejemplo, supóngase que se decide usar las 100 direcciones de 0 a 99 en el archivo. En términos de la fórmula anterior.

$$\begin{aligned} a &= 13\ 820 \bmod 100 \\ &= \underline{\underline{20}}. \end{aligned}$$

Como el número de direcciones asignadas para el archivo no tiene que ser de algún tamaño en especial (mientras sea lo suficientemente grande para guardar todos los registros que se almacenarán en el archivo), se tiene un amplio grado de libertad en la elección del divisor n . Esto es bueno, porque la elección de n puede afectar en gran medida la forma en que se esparzan los registros.

Es común usar un número *primo* como divisor porque los números primos tienden a distribuir residuos en forma mucho más uniforme que los que no lo son. Sin embargo, uno que no es primo puede funcionar bien en muchos casos, especialmente si no tiene divisores primos menores de 20 (Hanson, 1982). Dado que el residuo va a ser la dirección de un registro, se elige un número tan próximo como sea posible al tamaño deseado del espacio de direcciones. Este número en realidad determina el tamaño del espacio de direcciones. Para un archivo con 75 registros, una buena elección puede ser 101, porque llenaría un 74.3 por ciento del archivo ($74/101 = 0.743$).

Si 101 es el tamaño del espacio de direcciones, la dirección base del registro del ejemplo se convierte en

$$\begin{aligned} a &= 13\ 820 \bmod 101 \\ &= \underline{\underline{84}}. \end{aligned}$$

Por lo tanto, el registro cuya llave es *LOWELL* se asigna al registro número 84 del archivo.

El procedimiento recién descrito puede llevarse a cabo con una función denominada *dispersión()*, descrita casi en pseudocódigo en la figura 11.2. El procedimiento *dispersión()* toma dos entradas: *LLAVE*, que debe ser un arreglo de códigos ASCII de al menos 12 caracteres, y

FUNCION dispersión (LLAVE, MAXDIR)

```

asigna 0 a SUM
asigna 0 a J

mientras (J < 12 )
    asigna (SUM + 100*LLAVE [J] + llave [J + 1] mod 20000 a SUM;
    incrementa J en 2
fin mientras

regresa (SUM mod MAXDIR)
fin FUNCION

```

FIGURA 11.2 • La función *dispersión (LLAVE, MAXDIR)* usa desglose y división entre un número primo para calcular una dirección de dispersión.

MAXDIR, la cual tiene el tamaño de la dirección. El valor devuelto por *dispersión()* es la dirección.

11.3

FUNCIONES DE DISPERSION Y DISTRIBUCIONES DE REGISTROS

De las dos funciones de dispersión que se han examinado, una esparce muy bien los registros y la otra no lo hace nada bien. En esta sección se analizan formas para describir distribuciones de registros en los archivos. Comprender las distribuciones facilita el análisis de otros métodos de dispersión.

11.3.1 DISTRIBUCION DE REGISTROS ENTRE DIRECCIONES

La figura 11.3 ilustra tres diferentes distribuciones de siete registros entre diez direcciones. Idealmente, una función de dispersión debe distribuir los registros en un archivo de modo que no existan colisiones, como se ilustra en la distribución (a). Tal distribución se denomina *uniforme* porque los registros se espacien en forma uniforme entre las direcciones. Se señaló antes que las distribuciones completamente uniformes son tan difíciles de encontrar que, por lo general, no vale la pena intentar hacerlo.

La distribución (b) ilustra el peor tipo posible de distribución. Todos los registros comparten la misma dirección base, dando como resultado el máximo número de colisiones. Cuanto más se parezcan las distribuciones a ésta, mayor problema serán las colisiones.

La distribución (c) ilustra una distribución en la que los registros están algo esparcidos, con pocas colisiones. Este es el caso más probable si se tiene una función que distribuya llaves *en forma aleatoria*. Si una función de dispersión es aleatoria, entonces, para una determinada llave, cualquier dirección tiene la misma probabilidad que las demás de ser elegida. El hecho de que cierta dirección sea elegida para una llave no disminuye ni incrementa la probabilidad de que vuelva a ser elegida para otra llave.

Debe quedar claro que si se usa una función aleatoria de dispersión para generar un número grande de direcciones a partir de un número grande de llaves, entonces simplemente *por casualidad* algunas direcciones van a ser generadas con más frecuencia que otras. Por ejemplo, si se tiene una función aleatoria de dispersión que genere direcciones entre 0 y 99 y se dan 100 llaves a la función, es de esperarse que algunas de las 100 direcciones sean elegidas más de una vez y algunas no se elijan nunca.

Aunque una distribución aleatoria de registros entre las direcciones disponibles no es lo ideal, es una alternativa aceptable, dado que es prácticamente imposible encontrar una función que logre una distribución uniforme. Las distribuciones uniformes pueden descartarse totalmente, pero a veces pueden encontrarse mejores distribuciones que las aleatorias, en el sentido de que mientras generen un adecuado

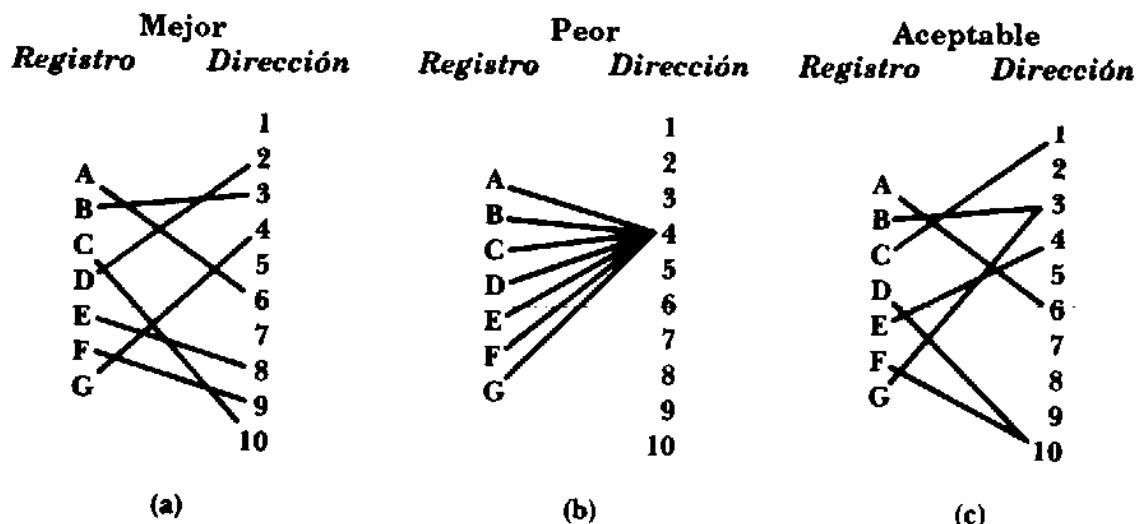


FIGURA 11.3 • Diferentes distribuciones. (a) Sin sinónimos (uniforme). (b) Todos son sinónimos (peor caso). (c) Pocos sinónimos.

número de sinónimos, distribuyen los registros entre las direcciones más uniformemente que una distribución aleatoria.

11.3.2 ALGUNOS OTROS METODOS DE DISPERSION

Sería bueno que hubiese una función de dispersión que garantizara una distribución mejor que la aleatoria en todos los casos, pero no la hay. La distribución generada por una función de dispersión depende del conjunto de llaves que se dispersan; por lo tanto, elegir una función de dispersión apropiada implica considerar con inteligencia las llaves que se van a dispersar, y quizá realizar algunos experimentos. Los métodos para elegir una función de dispersión razonable que se estudian en esta sección son aquellos cuyo buen funcionamiento se ha comprobado bajo circunstancias adecuadas. Pueden encontrarse detalles adicionales sobre éstos y otros métodos en Knuth [1973b], Maurer [1975], Hanson [1982] y Sorenson *et al.* [1978].

He aquí algunos métodos que son potencialmente mejores que el aleatorio:

- Examinar las llaves para buscar un patrón.* Algunas veces las llaves exhiben patrones que se esparcen en forma natural. Esto sucede con mayor probabilidad entre llaves numéricas que entre llaves alfabéticas. Por ejemplo, un conjunto de números de identificación de empleados puede estar ordenado de acuerdo con su fecha de entrada a la organización, lo cual quizás conduzca incluso a *no tener* sinónimos. Si alguna *parte* de la llave muestra un patrón útil, puede usarse una función de dispersión que extraiga esa parte de la llave.
- Desglosar partes de la llave.* Desglosar es una etapa del método recién analizado, e implica extraer dígitos de parte de una llave y sumar las partes extraídas. Este método destruye los patrones originales de las llaves, pero en determinadas circunstancias puede preservar la separación entre ciertos *subconjuntos* de llaves que se esparcen por sí mismas en forma natural.
- Dividir la llave entre un número.* Dividir entre el tamaño de la dirección y usar el residuo suele ocurrir en algún lugar de una función de dispersión, porque el propósito de la función es producir un dirección dentro de cierto intervalo. La división preserva las secuencias de llaves consecutivas para que puedan aprovecharse las secuencias que efectivamente esparcen llaves. Sin embargo, si hay *varias* secuencias de llaves consecutivas, la división entre un número que tenga muchos factores pequeños puede dar como resultado muchas colisiones. Las investigaciones

han demostrado que los números sin divisores menores de 19 por lo general evitan este problema. Es aún más probable que la división entre un *primo* genere resultados diferentes a partir de diferentes secuencias consecutivas que los obtenidos con la división entre un número que no sea primo.

Los métodos anteriores se idearon para sacar partido del orden natural de las llaves. Los dos métodos siguientes deben intentarse cuando, por alguna razón, los métodos mejores que el aleatorio no funcionan. En estos casos, la aleatoriedad es el objetivo.

- Elevar al cuadrado la llave y tomar la mitad.* Este conocido método (también llamado método del *medio cuadrado*) implica tratar la llave como un solo número grande, elevarlo al cuadrado y extraer de la mitad del resultado el número de dígitos que sea necesario. Por ejemplo, supóngase que se desea generar direcciones entre 0 y 99. Si la llave es el número 453, su cuadrado es 205 209 y al extraer los dos dígitos del medio se obtiene un número entre 0 y 99, en este caso, 52.

Mientras las llaves no contengan muchos ceros adelante o atrás, este método por lo regular produce resultados razonablemente aleatorios. Una característica poco atractiva es que con frecuencia requiere aritmética de precisión múltiple.

- Transformación de la base.* Este método implica convertir la llave a alguna otra base numérica que no sea con la que se esté trabajando, y tomar después el resultado del módulo con la máxima dirección como la dirección de dispersión. Por ejemplo, supóngase que quieren generarse direcciones entre 0 y 99. Si la llave es el número decimal 453, su equivalente en base 11 es 382; $382 \bmod 99 = 85$, de modo que 85 es la dirección de dispersión.

La transformación de base por lo general es más confiable que el método del medio cuadrado para aproximarse a lo verdaderamente aleatorio, aunque se ha encontrado que el medio cuadrado da buenos resultados cuando se aplica a algunos conjuntos de llaves.

11.3.3 PREDICCIÓN DE LA DISTRIBUCIÓN DE LOS REGISTROS

Como es prácticamente imposible lograr una distribución uniforme de registros entre las direcciones disponibles en un archivo, es importante poder predecir cómo se distribuirán. Por ejemplo, si se sabe que es probable que un número grande de direcciones tenga muchos más

registros asignados que los que puede guardar, se sabe que ocurrirán muchas colisiones.

Aunque no se dispone de herramientas matemáticas elegantes adecuadas para predecir colisiones entre distribuciones que sean mejores que la aleatoria, sí las hay para comprender este tipo de comportamiento cuando los registros se distribuyen en forma aleatoria. Si se supone una distribución aleatoria (aun sabiendo que sin duda es mejor que lo aleatorio), pueden usarse estas herramientas para obtener estimaciones conservadoras del comportamiento probable de este método de dispersión.

LA DISTRIBUCION DE POISSON†. Se desea predecir el número probable de colisiones en un archivo que puede guardar sólo un registro por dirección. Primero se examina lo que le sucede a una dirección determinada cuando se aplica una función de dispersión a una llave. Cuando todas las llaves de un archivo se dispersan sería deseable poder contestar qué probabilidad hay de que

- Ninguna se disperse a la dirección dada.
- Exactamente una llave se disperse a la dirección.
- Exactamente dos llaves se dispersen a la dirección (dos sinónimos).
- Exactamente tres, cuatro llaves (y así sucesivamente) se dispersen a la dirección.
- Todas las llaves del archivo se dispersen a la misma dirección.

¿Cuál de estos resultados se esperaría como probable, y cuál como improbable? Supóngase que hay N direcciones en un archivo. Cuando se dispersa una sola llave, existen dos posibles resultados con respecto a la dirección dada:

- A — La dirección no se elige, o
- B — La dirección se elige.

¿Cómo se expresan las probabilidades de los dos resultados? Si $p(A)$ y a denotan la probabilidad de que la dirección no sea elegida, y $p(B)$ y b la probabilidad de que la dirección sea elegida, entonces:

$$p(B) = b = 1/N,$$

†Esta sección desarrolla una fórmula para predecir la forma en que los registros serán distribuidos entre direcciones en un archivo si se usa una función aleatoria de dispersión. El análisis supone conocimiento de algunos conceptos elementales de probabilidad y combinatoria. Si el lector desea omitir el desarrollo e ir directo a la fórmula, ésta se presenta en la siguiente sección.

ya que la dirección tiene una oportunidad en N de ser elegida, y

$$p(A) = a = (N - 1)/N = N - 1/N$$

porque la dirección tiene $N - 1$ oportunidad de N de no ser elegida. Si hay diez direcciones ($N = 10$), la probabilidad de que la dirección sea elegida es $b = 1/10 = 0.1$, y la probabilidad de que no sea elegida es $a = 1 - 0.1 = 0.9$.

Ahora supóngase que se dispersan dos llaves. ¿Qué probabilidad hay de que ambas lleguen a la dirección dada? Como las dos aplicaciones de la función de dispersión son independientes entre sí, la probabilidad de que ambas produzcan la dirección dada es un producto:

$$p(BB) = b \times b = 1/N \times 1/N \text{ para } N = 10: b \times b = 0.1 \times 0.1 = 0.01.$$

Por supuesto, hay otros resultados posibles cuando se dispersan dos llaves. Por ejemplo, la segunda llave se podría dispersar a otra dirección que no sea la dirección dada. La probabilidad de que esto ocurra es el producto

$$p(BA) = b \times a = 1/N \times (1 - 1/N) \text{ para } N = 10: b \times a = 0.1 \times 0.9 = 0.09.$$

En general, cuando se quiere conocer la probabilidad de alguna secuencia de resultados, como *BABBA*, puede reemplazarse cada *A* y *B* por *a* y *b*, respectivamente, y calcular el producto indicado:

$$p(BABBA) = b \times a \times b \times b \times a = a^2 b^3 \quad \text{para } N = 10: a^2 b^3 = (0.9)^2(0.1)^3. \quad 2$$

Este ejemplo muestra cómo encontrar la probabilidad de que tres *B* y dos *A* ocurran en el orden mostrado. Se quiere conocer la probabilidad de que exista un cierto número de *B* y *A*, pero sin importar el orden. Por ejemplo, supóngase que se dispersan cuatro llaves y se desea conocer la probabilidad de que dos lleguen exactamente a la dirección dada. Esto puede ocurrir en seis formas, las cuales tienen la misma probabilidad:

Resultado	Probabilidad	Para $N = 10$
<i>BBAA</i>	$bbaa = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
<i>BABA</i>	$baba = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
<i>BAAB</i>	$baab = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
<i>ABBA</i>	$abba = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
<i>ABAB</i>	$abab = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
<i>AABB</i>	$aabb = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$

Puesto que estas seis secuencias son independientes entre sí, la probabilidad de dos B y dos A es la suma de las probabilidades de los resultados individuales:

$$p(BBAA) + p(BABA) + \dots + p(AABB) = 6b^2 a^2 = 6 \times 0.0036 = 0.0216.$$

El 6 en la expresión $6b^2 a^2$ representa el número de formas en que dos B y dos A pueden distribuirse entre cuatro lugares.

En general, el evento “ r pruebas dan como resultado $r - x$ A y x B ” puede suceder en tantas formas como $r - x$ letras A puedan distribuirse entre r lugares. La probabilidad de cada una de tales formas es

$$a^{r-x} b^x$$

y su número está dado por la fórmula

$$C = \frac{r!}{(r-x)! x!}.$$

Esta es la bien conocida fórmula del número de formas de seleccionar x cosas de entre un conjunto de r cosas. De esto se sigue que cuando se dispersan r llaves, la probabilidad de que se elija una dirección x veces y no sea elegida $r - x$ veces puede expresarse como

$$p(x) = Ca^{r-x} b^x.$$

Más aún, si se sabe que existen N direcciones disponibles, se puede ser preciso sobre las probabilidades individuales de A y B , y la fórmula se convierte en

$$p(x) = C(1 - 1/N)^{r-x}(1/N)^x,$$

donde C tiene la definición dada anteriormente.

¿Qué significa esto? Significa que si, por ejemplo, $x = 0$, puede calcularse la probabilidad de que una dirección dada tenga 0 registros asignados por la función de dispersión usando la fórmula

$$p(0) = C(1 - 1/N)^{r-0}(1/N)^0.$$

Si $x = 1$, esta fórmula proporciona la probabilidad de que un registro sea asignado a una dirección dada:

$$p(1) = C(1 - 1/N)^{r-1}(1/N)^1.$$

La desventaja que hay con esta expresión es que resulta problemático calcularla. (Inténtelo para 1000 direcciones y 1000 registros; $N = r = 1000$.) Afortunadamente, para valores grandes de N y r hay una función que es una muy buena aproximación a $p(x)$ y que es mucho más fácil de calcular. Se llama *función de Poisson*.

LA FUNCION DE POISSON APLICADA A LA DISPERSION. La función de Poisson, que también se denota con $p(x)$, está dada por

$$P(x) = \frac{(r/N)^x e^{-(r/N)}}{x!},$$

donde N , r , x , y $p(x)$ tienen exactamente el mismo significado que en la sección anterior. Esto es, si

N = el número de direcciones disponibles;

r = el número de registros que se van a almacenar, y

x = el número de registros asignados a una dirección dada,

entonces $p(x)$ indica la probabilidad de que a una dirección determinada se hayan asignado x registros luego de haber aplicado la función de dispersión a los n registros.

Por ejemplo, supóngase que hay 1000 direcciones ($N = 1000$) y 1000 registros cuyas llaves se dispersarán a las direcciones ($r = 1000$). Como $r/N = 1$, la probabilidad de que una dirección determinada no tenga llaves dispersadas a ella ($x = 0$) es

$$p(0) = \frac{1^0 e^{-1}}{0!} = 0.368.$$

Las probabilidades de que una determinada dirección tenga exactamente una, dos o tres llaves, respectivamente, asignadas son

$$p(1) = \frac{1^1 e^{-1}}{1!} = 0.368.$$

$$p(2) = \frac{1^2 e^{-1}}{2!} = 0.184$$

$$p(3) = \frac{1^3 e^{-1}}{3!} = 0.061.$$

Si puede usarse la función de Poisson para estimar la probabilidad de que una dirección determinada tenga cierto número de registros, también puede usarse para predecir el número de direcciones que tendrán un número dado de registros asignados.

Por ejemplo, supóngase que hay 1000 direcciones ($N = 1000$) y 1000 registros ($r = 1000$). Multiplicando 1000 por la probabilidad de que una

dirección *determinada* tenga x registros asignados a ella se obtiene el número total esperado de direcciones con x registros asignados a ellas. Esto es, $1000 p(x)$ da el número de direcciones que tienen x registros asignados.

En general, si hay N direcciones, entonces el número esperado de direcciones con x registros asignados a ellas es

$$Np(x).$$

Esto sugiere otra forma de considerar $p(x)$. En vez de tomarla como una medida de probabilidad, $p(x)$ *puede considerarse como lo que da la proporción de direcciones que tienen x registros lógicos asignados por la dispersión.*

Ahora que se tiene una herramienta para predecir la proporción esperada de direcciones que tendrán cero, uno, dos, etc., registros asignados por una función aleatoria de dispersión, se puede aplicar para predecir los números de colisiones.

11.3.4 PREDICCION DE LAS COLISIONES EN UN ARCHIVO LLENO

Supóngase que se tiene una función de dispersión que se cree distribuye registros en forma aleatoria, y que se desea almacenar 10 000 registros en 10 000 direcciones. ¿Cuántas direcciones se esperaría tener sin registros asignados?

Puesto que $r = 10\ 000$ y $N = 10\ 000$, $r/N = 1$. Por lo tanto, la proporción de direcciones con 0 registros asignados debe ser

$$p(0) = \frac{1^0 e^{-1}}{1!} = 0.3679.$$

El *número* de direcciones sin registros asignados es

$$10\ 000 \times p(0) = 3679.$$

¿Cuántas direcciones tendrán uno, dos y tres registros, respectivamente?

$$\begin{aligned}10\ 000 \times p(1) &= 0.3679 \times 10\ 000 = 3679 \\10\ 000 \times p(2) &= 0.1839 \times 10\ 000 = 1839 \\10\ 000 \times p(3) &= 0.0613 \times 10\ 000 = 613.\end{aligned}$$

Como las 3679 direcciones que corresponden a $x = 1$ tienen exactamente un registro asignado, sus registros no tienen sinónimos. Sin embargo, las 1839 direcciones con dos registros cada una, representan problemas potenciales. Si cada una de ellas tiene espacio sólo para un registro, y se les asignan dos, hay una colisión. Esto significa que 1839 registros entrarán en las direcciones, pero otros 1839 no. Habrá 1838 registros en saturación.

Cada una de las 613 direcciones con tres registros tiene un problema aún mayor. Si cada dirección tiene espacio para un solo registro, habrá dos registros en saturación por dirección. Correspondiendo a estas direcciones habrá un total de $2 \times 613 = 1226$ registros en saturación. La situación es mala. Se tienen miles de registros que no entran dentro de las direcciones asignadas por la función de dispersión. Es necesario desarrollar un método para manejar estos registros en saturación. Pero primero se va a intentar reducir su número.

11.4

¿CUANTA MEMORIA ADICIONAL DEBE USARSE?

Se ha visto lo importante que es elegir un buen algoritmo de dispersión para reducir las colisiones. Una segunda forma para reducirlas (y, por tanto, para reducir la longitud media de búsqueda) es usar memoria adicional. Las herramientas desarrolladas en la sección anterior pueden ayudar a determinar el efecto del uso de memoria adicional en el desempeño.

11.4.1 DENSIDAD DE EMPAQUETAMIENTO

El término densidad de empaquetamiento se refiere a la proporción entre el número de registros por almacenar (r) y el número de espacios disponibles (N):†

$$\frac{\text{Número de registro}}{\text{Número de espacios}} = \frac{r}{N} = \text{Densidad de empaquetamiento}$$

†Aquí se supone que sólo puede almacenarse un registro en cada dirección. De hecho, esto no es necesariamente cierto como se estudiará más adelante.

Por ejemplo, si hay 75 registros ($n = 75$) y 100 direcciones ($N = 100$), la densidad de empaquetamiento es

$$75/100 = 0.75 = 75\%.$$

La densidad de empaquetamiento es una medida de la cantidad del espacio que se usa en realidad en un archivo, y es el único valor necesario para evaluar el desempeño en un ambiente de dispersión, suponiendo que el método de dispersión usado logra una distribución de registros razonablemente aleatoria. No importan el tamaño real del archivo ni su espacio de direcciones; lo importante son los tamaños relativos de los dos, que están dados por la densidad de empaquetamiento.

Considérese la densidad de empaquetamiento en términos de latas alineadas en una cerca de tres metros de longitud. Si hay diez latas y se tira una piedra, hay cierta probabilidad de pegarle a una lata. Si hay 20 latas en la misma longitud de cerca, ésta tienen una densidad de empaquetamiento mayor, y es más probable que la piedra le pegue a una lata. Así sucede con los registros en un archivo: cuantos más registros estén empaquetados en un determinado espacio del archivo, más probable es que ocurra una colisión al agregar un registro nuevo.

Es necesario decidir cuánto espacio se está dispuesto a desperdiciar para reducir el número de colisiones. La respuesta depende en gran medida de circunstancias particulares. Es deseable tener el menor número posible de colisiones, pero no a expensas, por ejemplo, de que el archivo use dos discos en lugar de uno.

11.4.2 PREDICCION DE COLISIONES PARA DIFERENTES DENSIDADES DE EMPAQUETAMIENTO

Se necesita una descripción cuantitativa de los efectos que tiene el cambio de densidad de empaquetamiento. En particular, es necesario poder predecir el número de colisiones probables para una densidad de empaquetamiento determinada. Por fortuna, la función de Poisson proporciona la herramienta para lograrlo.

El lector puede haber notado ya que la fórmula de la densidad de empaquetamiento (r/N) ocurre dos veces en la fórmula de Poisson

$$p(x) = \frac{(r/N)^x e^{-r/N}}{x!}$$

Ciertamente, los números de registros (r) y direcciones (N) siempre ocurren juntos en forma del cociente r/N . Nunca aparecen en forma

independiente. Una implicación obvia de esto es que la manera en que se distribuyen los registros depende, en parte, de la proporción entre el número de registro y el número de direcciones disponibles, y no de los números absolutos de registros o de direcciones. Se comportan de la misma manera 500 registros distribuidos entre 1000 direcciones que 500 000 registros distribuidos entre 1 000 000 de direcciones.

Supóngase que se asignan 1000 direcciones para guardar 500 registros en un archivo dispersado en forma aleatoria, y que cada dirección puede almacenar un registro. La densidad de empaquetamiento del archivo es

$$r/N = 500/1000 = 0.5.$$

Se responderá a las siguientes preguntas acerca de la distribución de registros entre las direcciones disponibles del archivo:

- ¿Cuántas direcciones no deberán tener registros asignados?
- ¿Cuántas direcciones deben tener exactamente un registro asignado (no hay sinónimos)?
- ¿Cuántas direcciones deben tener un registro más uno o varios sinónimos?
- Suponiendo que sólo puede asignarse un registro a cada dirección base, ¿cuántas registros en saturación pueden esperarse?
- ¿Qué porcentaje de registro deben ser registros en saturación?

1. *¿Cuántas direcciones no deberán tener registros asignados?*

Como $p(0)$ da la proporción de direcciones sin registros asignados, el número de tales direcciones es

$$\begin{aligned} Np(0) &= 1000 \times \frac{(0.5)^0 e^{-0.5}}{0!} \\ &= 1000 \times 0.607 \\ &= 607. \end{aligned}$$

2. *¿Cuántas direcciones deben tener exactamente un registro asignado (no hay sinónimos)?*

$$\begin{aligned} Np(1) &= 1000 \times \frac{(0.5)^1 e^{-0.5}}{1!} \\ &= 100 \times -0.303 \\ &= 303. \end{aligned}$$

3. *¿Cuántas direcciones deben tener un registro más uno o varios sinónimos?*

Los valores de $p(2)$, $p(3)$, $p(4)$, y demás, dan las proporciones de direcciones con uno, dos, tres y más sinónimos asignados a ella. Por tanto, la suma

$$p(2) + p(3) + p(4) + \dots$$

da la proporción de todas la direcciones que tienen un sinónimo al menos. Parecería que se requieren muchos cálculos, pero no es así, porque los valores de $p(x)$ crecen bastante poco para las x mayores que 3. Esto es verosímil: como sólo el 50 por ciento del archivo está cargado, no se esperaría que se dispersen muchas llaves a una dirección cualquiera. Por lo tanto, debería ser muy pequeño el número de direcciones que tuvieran más de unas tres llaves asignadas. Sólo es necesario calcular los resultados hasta $p(5)$ antes de que lleguen a ser insignificantemente pequeños:

$$\begin{aligned} p(2) + p(3) + p(4) + p(5) &= 0.0758 + 0.0126 + 0.0016 + 0.0002 \\ &= 0.0902. \end{aligned}$$

El *número* de direcciones con uno o más sinónimos es sólo el producto de N y resulta en:

$$\begin{aligned} N[p(2) + p(3) + \dots] &= 1000 \times 0.0902 \\ &= 90. \end{aligned}$$

4. *Suponiendo que sólo puede asignarse un registro a cada dirección base, ¿cuántos registros en saturación pueden esperarse?*

Para cada una de las direcciones representadas por $p(2)$ puede almacenarse un registro en la dirección, y el otro debe ser un registro en saturación. Para cada dirección representada por $p(3)$, un registro puede almacenarse en la dirección, y dos son registros en saturación, y así sucesivamente. Por lo tanto, el número esperado de registros en saturación está dado por

$$\begin{aligned} 1 \times N \times p(2) + 2 \times N \times p(3) + 3 \times N \times p(4) + 4 \times N \times p(5) \\ = N \times [1 \times p(2) + 2 \times p(3) + 3 \times p(4) + 4 \times p(5)] \\ = 1000 \times [1 \times 0.0758 + 2 \times 0.0126 + 3 \times 0.0016 + 4 \times 0.0002] \\ = 107. \end{aligned}$$

5. *¿Qué porcentaje de registros deben ser registros en saturación?*
Si hay 107 registros en saturación y 500 registros en total, entonces la proporción de registros en saturación es

$$107/500 = 0.214 = 21.4\%.$$

Conclusión: si la densidad de empaquetamiento es de 50 por ciento y cada dirección pueden almacenar sólo un registro, puede esperarse que alrededor del 21 por ciento de todos los registros sean almacenados en algún lugar que no sea sus direcciones base.

La tabla 11.2 muestra la proporción de registros que no están almacenados en su dirección base para varias densidades de empaquetamiento diferente. La tabla muestra que si la densidad de empaquetamiento es 10 por ciento, entonces alrededor del 5 por ciento de las veces que se intenta acceder a un registro, ya existe otro allí. Si la densidad es 100 por ciento, entonces alrededor del 37 por ciento de todos los registros se colisionan con otros en sus direcciones base. La tasa del 4.8 por ciento que resulta cuando la densidad de empaquetamiento es del 10 por ciento parece satisfactoria hasta que se descubre que para cada registro en el archivo ¡habrá nueve espacios sin usar!

TABLA 11.2

Efecto de la densidad de empaquetamiento en la proporción de registros no almacenados en sus direcciones base

Densidad de empaquetamiento (%)	Sinónimos como % de los registros
10	4.8
20	9.4
30	13.6
40	17.6
50	21.4
60	24.8
70	28.1
80	31.2
90	34.1
100	36.8

El 36.8 por ciento resultante del uso del 100 por ciento parece aceptable cuando se ve en términos de 0 por ciento de espacio sin uso. Por desgracia, el 36.8 por ciento no cuenta la historia completa. Si el 36.8 por ciento de registros no está en sus direcciones base, entonces estará en algún otro lugar, en muchos casos probablemente usando direcciones base que corresponden a otros registros. Cuanto mayor es el número de registros sin lugar propio, mayor es la competencia por el espacio con los otros registros sin lugar. Pronto pueden formarse címu-

los de registros en saturación, que en algunos casos conducen a búsquedas extremadamente lentas para algunos de los registros. Está claro que la reubicación de los registros en colisión es un aspecto importante. Ahora se analizará un método sencillo para lograrlo.

11.5

RESOLUCION DE COLISIONES MEDIANTE SATURACION PROGRESIVA

Aun cuando un algoritmo de dispersión sea muy bueno, es probable que ocurran colisiones. Por lo tanto, cualquier programa de dispersión debe incorporar algún método para tratar con los registros que no pueden entrar en su dirección base. Existen varias técnicas de manejo de registros en saturación, y la búsqueda de mejores técnicas continúa siendo un área de investigación activa. Se examinarán varios métodos, pero se centrará la atención en uno muy sencillo que con frecuencia funciona bien. La técnica tiene varios nombres, entre ellos *saturación progresiva* y *verificación lineal*.

11.5.1 FUNCIONAMIENTO DE LA SATURACION PROGRESIVA

En la figura 11.4 se muestra un ejemplo de una situación en la que ocurre una colisión. Se quiere almacenar en el archivo el registro cuya llave es York. Por desgracia, el nombre York se asigna a la misma dirección que el nombre de Rosen, cuyo registro ya está almacenado ahí. Como York no puede entrar en su dirección base, es un registro en saturación. Si se usa la saturación progresiva, se busca en las siguientes direcciones en secuencia hasta que se encuentra una vacía. La primera dirección libre se convierte en la dirección del registro. En el ejemplo, la dirección 9 es el primer registro que se encuentra vacío, de modo que el registro que pertenece a York se almacena en la dirección 9.

Pronto será necesario encontrar el registro de York en el archivo. Como York produce aún 6, la búsqueda del registro comienza en la dirección 6. Ahí se encuentra el registro de York, de manera que se continua en los registros sucesivos hasta que se tiene la dirección 9, donde se encuentra.

Cuando se busca un espacio libre o un registro al *final* del archivo ocurre un problema interesante, que se ilustra en la figura 11.5, donde se supone que el archivo puede contener 100 registros en las direcciones 0 a 99. Blue se asigna al registro número 99, el cual está ocupado ya por Jello. Puesto que el archivo tiene sólo 100 registros, no es posible usar

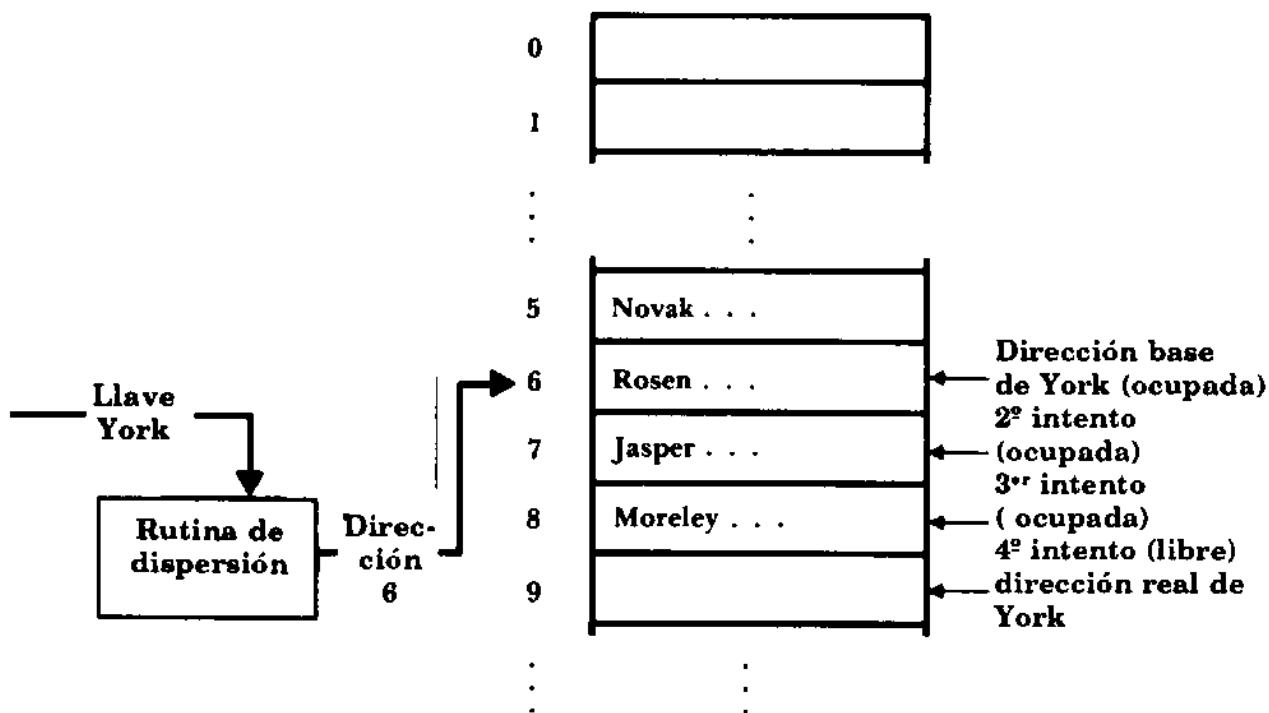


FIGURA 11.4 • Solución de colisiones con saturación progresiva.

100 como la siguiente dirección. La forma de manejar esto con la saturación progresiva es dar la vuelta al espacio de direcciones del archivo y elegir la dirección 0 como siguiente dirección. Como en este caso la dirección 0 no está ocupada, Blue se almacena en la dirección 0.

¿Qué sucede si se busca un registro, pero éste nunca se colocó en el archivo? La búsqueda comienza, como antes, en la dirección base del registro, y después se continúa buscando en localidades sucesivas. Pueden suceder dos cosas:

- Si se encuentra una dirección vacía la rutina de búsqueda puede suponer que esto significa que el registro no está en el archivo, o
- Si el archivo está lleno, la búsqueda vuelve a donde comenzó. Sólo entonces está claro que el registro no está en el archivo. Cuando esto ocurre, o incluso cuando se está llenando el archivo, la búsqueda no puede llegar a ser intolerablemente lenta, esté o no el registro que se busca dentro del archivo.

La gran ventaja de la saturación progresiva es su simplicidad. En muchos casos, es un método perfectamente adecuado. Sin embargo, existen técnicas de manejo de colisiones que operan mejor que la saturación progresiva; algunas de ellas se examinan más adelante en el

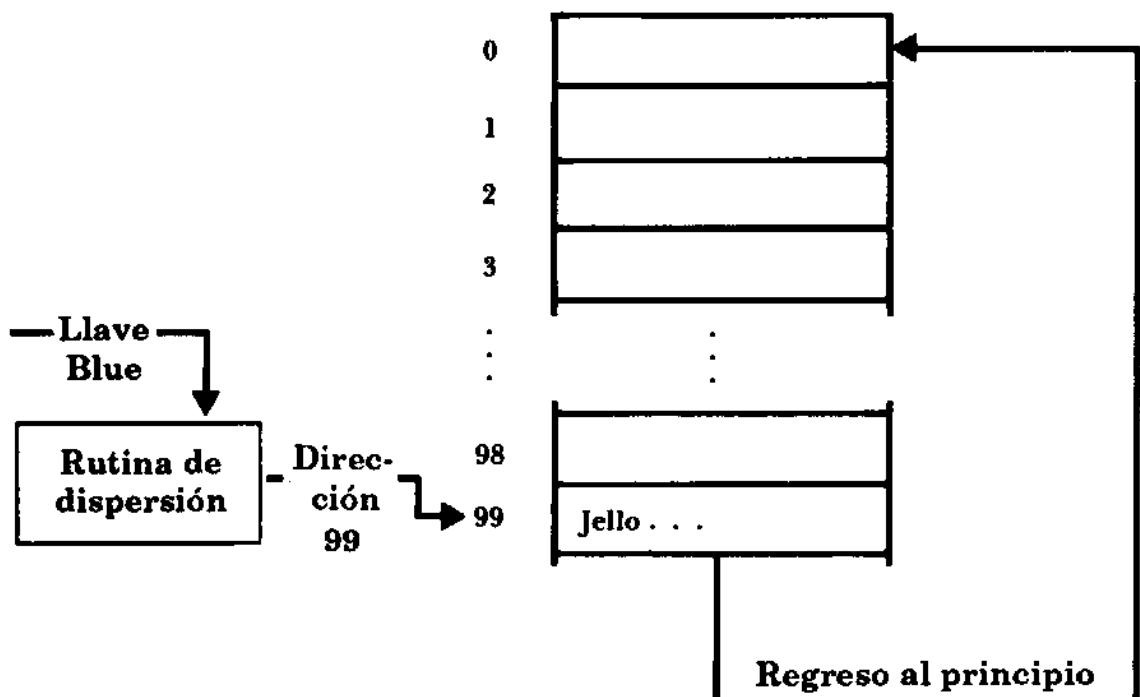


FIGURA 11.5• Búsqueda de una dirección después del final de un archivo.

capítulo. Ahora se examinará el efecto que tienen la saturación progresiva en el desempeño.

1.1.5.2 LONGITUD DE BUSQUEDA

La razón de evitar la saturación es que, por supuesto, ocurren búsquedas adicionales (y, por tanto, accesos adicionales al disco) cuando un registro no se encuentra en su dirección base. Si hay muchas colisiones, habrá bastantes registros en saturación que ocupen espacios que no deben. Pueden formarse cúmulos de registros, lo que da por resultado la colocación de registros a gran distancia de su dirección base, de tal forma que se requieren muchos accesos al disco para extraerlos.

Considérese el siguiente conjunto de llaves y las correspondientes direcciones producidas por alguna función de dispersión.

Llave	Dirección base
Adams	20
Bates	21
Cole	21
Dean	22
Evans	20

Si estos registros se cargan en un archivo vacío y se usa saturación progresiva para resolver las colisiones, sólo dos registros estarán en sus direcciones base, por lo que se requieren accesos adicionales para extraer los demás. La figura 11.6 muestra dónde se almacena cada llave, junto con la información sobre cuántos accesos se requieren para extraerla.

El término *longitud de búsqueda* se refiere al número de accesos requerido para extraer un registro de la memoria secundaria. En el contexto de la dispersión, la longitud de búsqueda se incrementa cada vez que ocurre una colisión. Si un registro está a gran distancia de su dirección base, la longitud de búsqueda puede ser inaceptable. Un buen indicador del problema de la saturación es la *longitud media de búsqueda*, que no es sino el número promedio de veces que se espera acceder al disco para extraer un registro. Una primera estimación de la longitud media de búsqueda puede calcularse mediante la *longitud total de búsqueda* (la suma de las longitudes de búsqueda de los registros individuales) dividida entre el número de registros:

$$\text{Longitud media de búsqueda} = \frac{\text{Longitud total de búsqueda}}{\text{Número total de registros}}$$

Dirección real	Dirección base	Número de accesos necesarios para recuperar el dato
0		
:	:	
20	Adams . . .	20
21	Bates . . .	21
22	Cole . . .	21
23	Dean . . .	22
24	Evans . . .	20
25		5
:	:	

FIGURA 11.6 - Ilustración de los efectos de la acumulación de registros. Al acumularse las llaves, el número de accesos requeridos para acceder a las últimas llaves puede volverse grande.

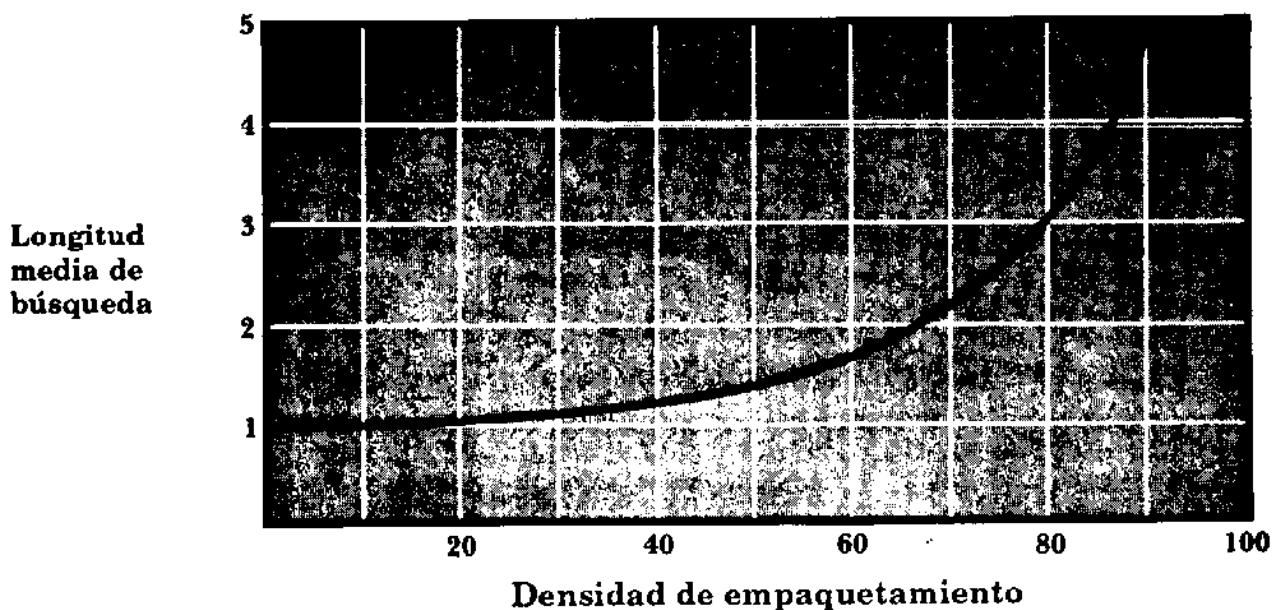


FIGURA 11.7 • Longitud media de búsqueda versus densidad de empaquetamiento en un archivo con dispersión en el cual se puede almacenar un registro por dirección. Se usa la saturación progresiva para resolver las colisiones, y el archivo está recién cargado.

En el ejemplo, la longitud media de búsqueda de los cinco registros es

$$\frac{1 + 1 + 2 + 2 + 5}{5} = 2.2.$$

Sin colisiones, la longitud media de búsqueda es 1, porque sólo es necesario un acceso para extraer cualquier registro. (Se indicó antes que un algoritmo que distribuye los registros en forma tan equitativa que no provoque colisiones se llama con toda propiedad algoritmo de dispersión *perfecta*, y que, desafortunadamente, tal algoritmo es casi imposible de construir.) Por otro lado, si son muchos los registros de un archivo que ocasionan colisiones, la longitud media de búsqueda se vuelve bastante larga. Hay formas de estimar la longitud media de búsqueda esperada para algunas especificaciones de archivos, y se analizan en una sección posterior.

Resulta que, usando la saturación progresiva, la longitud media de búsqueda aumenta rápidamente conforme se incrementa la densidad de empaquetamiento. La curva de la figura 11.7, adaptada de Peterson [1957], ilustra el problema. Si la densidad de empaquetamiento se mantiene baja, al 60 por ciento, el registro medio requiere menos de dos intentos para el acceso, pero para una densidad de empaquetamiento más deseable, de 80 por ciento o más, se incrementa muy rápidamente.

Las longitudes medias de búsqueda mayores de 2.0 por lo general se consideran inaceptables, de modo que parece necesario usar menos

del 40 por ciento del espacio de almacenamiento para tener el desempeño tolerable. Por fortuna, esta situación puede mejorarse en gran medida mediante un pequeño cambio al programa de dispersión. El cambio implica colocar más de un registro en una sola dirección.

11.6

ALMACENAMIENTO DE MAS DE UN REGISTRO POR DIRECCION: COMPARTIMIENTOS

Recuérdese que, cuando un computador recibe información del disco, para el sistema de E/S es igualmente fácil transferir varios registros o transferir uno solo. Recuérdese también que algunas veces puede ser conveniente considerar que los registros se agrupan en *bloques* en lugar de almacenarse individualmente. Entonces, ¿por qué no se amplía la idea de la dirección de un registro en un archivo a una dirección de un *grupo* de registros? Suele usarse la palabra *compartimiento* para describir un bloque de registros que se extraen en un acceso al disco, en especial cuando se consideran compartiendo la misma dirección. En discos que hace referencia por sectores, un *compartimiento* por lo regular consiste en uno o más sectores; en discos que hacen referencia por bloques, un *compartimiento* puede ser un bloque.

Considérese el siguiente conjunto de llaves, que se cargará dentro de un archivo que emplea dispersión.

Llave	Dirección base
Green	30
Hall	30
Jenks	32
King	33
Land	33
Marx	33
Nutt	33

La figura 11.8 ilustra parte de un archivo donde se cargaron los registros con estas llaves. Cada dirección del archivo identifica un *compartimiento* capaz de almacenar los registros que correspondan a tres sinónimos. Sólo el registro que corresponde a Nutt no puede acomodarse en una dirección base.

Dirección de los compartimientos	Contenido de los compartimientos	
:		
30	Green...	Hall...
31		
32	Jenks...	
33	King...	Land... Marks...

(Nutt ... es
un registro
en satu-
ración)

FIGURA 11.8 Ilustración de los compartimientos. Cada uno puede almacenar hasta tres registros. Sólo un sinónimo provoca saturación (Nutt).

Cuando un registro se almacena o se extrae, su *dirección de compartimiento base* se determina por dispersión. El compartimiento completo se carga en memoria primaria. Entonces puede usarse una búsqueda en memoria RAM a través de registros sucesivos en el compartimiento para encontrar el registro deseado. Cuando un compartimiento se llena, persiste el problema de la saturación (como en el caso de Nutt), pero esto ocurre con mucho menor frecuencia cuando se usan compartimientos que cuando cada dirección puede almacenar sólo un registro.

11.6.1 EFECTOS DE LOS COMPARTIMIENTOS EN EL DESEMPEÑO

Cuando se usan compartimientos, la fórmula usada para calcular la densidad de empaquetamiento cambia ligeramente porque cada dirección de compartimiento puede almacenar más de un registro. Para calcular cuánto empaquetamiento hay en un archivo, es necesario considerar tanto el número de direcciones (compartimientos) como el número de registros que pueden colocarse en cada dirección (tamaño del compartimiento). Si N es el número de direcciones y b es el número de registros que caben en un compartimiento, entonces bN es el número de localidades disponibles para registros. Si r es aún el número de registros del archivo, entonces

$$\text{Densidad de empaquetamiento} = r/bN.$$

Supóngase que se tiene un archivo en el que se almacenan 750 registros. Considérense las dos siguientes formas en que puede organizarse.

- Pueden almacenarse 750 registros de datos entre 1000 localidades, donde cada localidad almacena un registro. La densidad de empaquetamiento en este caso es

$$750/1000 = 75\%.$$

- Pueden almacenarse los 750 registros entre 500 localidades, donde cada localidad tiene un compartimiento de tamaño 2. Aún existen 1000 lugares (2×500) para almacenar los 750 registros, de manera que la densidad de empaquetamiento es aún

$$r/bN = 0.75 = 75\%.$$

Puesto que la densidad de empaquetamiento no cambia, podría esperarse que el uso de compartimientos en esta forma no mejore el desempeño, pero de hecho lo mejora considerablemente. La clave de la mejora es que, aunque hay menos direcciones, cada dirección individual tiene más espacio para variación del número de registros que se le asignan.

Calculemos la diferencia de desempeño de estas dos formas de almacenar el mismo número de registros en la misma cantidad de espacio. El punto de partida para los cálculos es la descripción fundamental de cada estructura de archivo.

	Archivo sin compartimientos	Archivo con compartimientos
Números de registros	$r = 750$	$r = 750$
Número de direcciones	$N = 1000$	$N = 500$
Tamaño del compartimiento	$b = 1$	$b = 2$
Densidad de empaquetamiento:	0.75	0.75
Proporción entre registros y direcciones:	$r/N = 0.75$	$r/N = 1.5$

Para determinar el número esperado de registros en saturación para cada archivo, recuérdese que, cuando se usa una función de dispersión aleatoria, la función de Poisson

$$p(x) = \frac{(r/N)^x e^{-r/N}}{x!}$$

da la proporción esperada de direcciones asignadas con x registros. Al evaluar la función para las dos organizaciones distintas de archivos, se

encuentra que los registros se asignaron a las direcciones de acuerdo con la distribución mostrada en la tabla 11.3.

En la tabla se observa que, cuando no se usan compartimientos, el 42.3 por ciento de las direcciones no tiene registros asignados, mientras que cuando se usan compartimientos de dos registros, sólo el 22.3 por ciento de las direcciones carece de registros asignados. Esto debería estar claro: puesto que en el caso de los registros existe tan sólo la mitad de direcciones para elegir, se deduce que hay que elegir una proporción mayor de direcciones para contener al menos un registro.

Nótese que la columna con compartimiento de la tabla 11.3 es más larga que la columna donde no los hay. ¿Significa esto que hay más sinónimos cuando hay compartimientos que cuando no los hay? De hecho así es, pero la mitad de esos sinónimos no provoca registros en saturación porque cada compartimiento puede almacenar dos registros. Se examinará esto más ampliamente mediante el cálculo del número exacto de registros en saturación que pueden ocurrir en los dos casos.

En el caso del archivo con compartimientos de tamaño uno, ninguna dirección a la que se asigne exactamente un registro tiene saturación; cualquier dirección con más de un registro sí la tiene. Recuérdese que el número esperado de registros en saturación está dado por

$$N \times [1 \times p(2) + 2 \times p(3) + 3 \times p(4) + 4 \times p(5) + \dots]$$

que para $r/N = 0.75$ y $N = 1000$, es aproximadamente

TABLA 11.3

Distribuciones de Poisson para las dos organizaciones de archivo diferentes

$p(x)$	Archivo sin compartimientos $(r/N = 0.75)$	Archivo con compartimientos $(r/N = 1.5)$
$p(0)$	0.423	0.223
$p(1)$	0.354	0.335
$p(2)$	0.133	0.251
$p(3)$	0.033	0.126
$p(4)$	0.006	0.047
$p(5)$	0.001	0.014
$p(6)$	—	0.004
$p(7)$	—	0.001

$$1000 \times [1 \times 0.1328 + 2 \times 0.0332 + 3 \times 0.0062 + 4 \times 0.0009 + 5 \times 0.0001] = 222.$$

Los 222 registros en saturación representan saturación de 29.6 por ciento.

Cuando el archivo tiene compartimientos, ninguna dirección a la que se asignan uno o dos registros tiene saturación. El valor de $p(1)$ (con $r/N = 1.5$) da la proporción de direcciones a las que se asigna exactamente un registro, y $p(2)$ (con $r/N = 1.5$) indica la proporción de aquellas a las que se asignan exactamente dos registros. Hasta que se tiene $p(3)$ se encuentran direcciones para las cuales hay registros en saturación. Para cada dirección representada por $p(3)$ pueden almacenarse dos registros en la dirección y uno debe ser un registro en saturación. En forma análoga, para cada dirección representada por $p(4)$ hay dos registros en saturación, y así sucesivamente. Por lo tanto, el número de registros en saturación en el archivo con compartimientos es

$$N \times [1 \times p(3) + 2 \times p(4) + 3 \times p(5) + 4 \times p(6) + \dots],$$

que para $r/N = 1.5$ y $N = 500$, es aproximadamente

$$500 \times [1 \times 0.1255 + 2 \times 0.0471 + 3 \times 0.0141 + 4 \times 0.0035 + 5 \times 0.0008] = 140.$$

Los 140 registros en saturación representan saturación de 18.7 por ciento.

Se ha demostrado que con un registro por dirección y una densidad de empaquetamiento del 75 por ciento, el número esperado de registros en saturación es del 29.6 por ciento. Cuando se usan 500 compartimientos, cada uno capaz de almacenar dos registros, la densidad de empaquetamiento continúa siendo del 75 por ciento, pero el número esperado de registros en saturación disminuye a 18.7 por ciento, esto es, alrededor del 37 por ciento de decremento en el número de veces que el programa tiene que buscar un registro en otro lugar. A medida que el tamaño del compartimiento aumenta, el desempeño mejora.

La tabla 11.4 muestra la proporción de colisiones que ocurren para diferentes densidades de empaquetamiento y para diferentes tamaños de compartimientos. En la tabla se observa, por ejemplo, que si se mantiene la densidad de empaquetamiento en 75 por ciento y se incrementa el tamaño del compartimiento a diez, el acceso a los registros produce saturación sólo el 4 por ciento de las veces.

Debe quedar claro que el uso de compartimientos puede mejorar mucho el desempeño de la dispersión. La pregunta es: "¿qué tan grandes deben ser los compartimientos?"

TABLA 11.4

Sinónimos que provocan colisiones expresados como un porcentaje de registros para diferentes densidades de empaquetamiento y diferentes tamaños de compartimientos

Densidad de empaquetamiento (%)	Tamaño del compartimiento				
	1	2	5	10	100
10	4.8	0.6	0.0	0.0	0.0
20	9.4	2.2	0.1	0.0	0.0
30	13.6	4.5	0.4	0.0	0.0
40	17.6	7.3	1.1	0.1	0.0
50	21.3	10.4	2.5	0.4	0.0
60	24.8	13.7	4.5	1.3	0.0
70	28.1	17.0	7.1	2.9	0.0
75	29.6	18.7	8.6	4.0	0.0
80	31.2	20.4	10.3	5.3	0.1
90	34.1	23.8	13.8	8.6	0.8
100	36.8	27.1	17.6	12.5	4.0

Desafortunadamente no hay una respuesta sencilla a esta pregunta, porque depende en buena parte de varias características del sistema, entre ellas los tamaños de los buffers que el sistema operativo maneje, las capacidades de sectores y pistas de los discos, y los tiempos de acceso del equipo (tiempos de desplazamientos, rotación y transferencia de datos).

Como regla general, probablemente no sea buena idea usar compartimientos mayores de una pista (a menos que los registros sean muy grandes). Sin embargo, aun una pista puede ser algunas veces demasiado grande cuando se considera el tiempo que toma transmitir la completa, comparado con el tiempo que toma transmitir unos cuantos sectores. Como la dispersión casi siempre implica la extracción de un solo registro por búsqueda, cualquier tiempo de transmisión adicional que resulte del uso de compartimientos extragrandes es esencialmente un gasto innútil.

En muchos casos un cúmulo es el mejor tamaño para un compartimiento. Por ejemplo, supóngase que un archivo con registros de 200 bytes se almacena en un sistema de discos que usa cúmulos de 1024 bytes. Se podría considerar cada cúmulo como un compartimiento, almacenar cinco registros por cúmulo, y dejar los 24 bytes restantes sin usar. Como en términos de tiempo de desplazamiento no es más costoso

acceder a un cúmulo de cinco registros que a un solo registro, la única pérdida que ocasiona el uso de compartimientos es el tiempo de transmisión adicional y los 24 bytes que no se usan.

Ahora la pregunta obvia es: "¿cómo afectan las mejoras en el número de colisiones el tiempo medio de búsqueda?" La respuesta depende en gran medida de las características de la unidad de disco en la cual esté cargado el archivo. Si hay numerosas pistas en cada cilindro, habrá un pequeño tiempo de desplazamiento porque es poco probable que los registros en saturación se traslapen de un cilindro a otro. Por otro lado, si hay sólo una pista por cilindro, el tiempo de desplazamiento consumirá una gran cantidad del tiempo de búsqueda.

Una medida menos exacta de la cantidad de tiempo requerido para extraer un registro es la longitud media de búsqueda, que se presentó anteriormente. En el caso de los compartimientos, la longitud media de búsqueda representa el número promedio de compartimientos a los que debe accederse para extraer un registro. La tabla 11.5 muestra las longitudes medias de búsqueda para archivos con diferentes densidades de empaquetamiento y tamaño de compartimientos, suponiendo que se usa la saturación progresiva para manejar las colisiones. Está claro que el uso de compartimientos parece ayudar en gran medida a reducir la longitud media de búsqueda. Cuanto más grande es el compartimiento, menor es la longitud de búsqueda.

11.6.2 ASPECTOS DE LA REALIZACION

En los primeros capítulos del texto se prestó bastante atención a los aspectos relacionados con la producción, uso y mantenimiento de archivos de acceso aleatorio con registros de longitud fija, a los cuales se accede mediante un número relativo de registro (NRR). Como un archivo con dispersión es un archivo de registros de longitud fija cuyo acceso se realiza por medio del NRR, ya debe saberse bastante bien cómo realizar los archivos con dispersión. Sin embargo, este tipo de archivos difiere de los ya analizados en dos aspectos importantes:

1. Puesto que una función de dispersión depende de que haya un número fijo de direcciones disponibles, el tamaño lógico de un archivo con dispersión debe fijarse antes de que pueda cargarse con registros, y debe permanecer fijo mientras se use la misma función de dispersión. (Se usa el término *tamaño lógico* para dejar abierta la posibilidad de que el espacio físico se asigne conforme se necesite.)
2. Como el NRR base de un registro en un archivo con dispersión es único en relación con su llave, cualquier procedimiento que agregue, elimine o cambie un registro debe hacerlo sin romper el vínculo

TABLA 11.5

Número promedio de accesos requeridos en una búsqueda con éxito mediante saturación progresiva

Densidad de empaquetamiento (%)	Tamaño del compartimiento				
	1	2	5	10	50
10	1.06	1.01	1.00	1.00	1.00
30	1.21	1.06	1.00	1.00	1.00
40	1.33	1.10	1.01	1.00	1.00
50	1.50	1.18	1.03	1.00	1.00
60	1.75	1.29	1.07	1.01	1.00
70	2.17	1.49	1.14	1.04	1.00
80	3.00	1.90	1.29	1.11	1.01
90	5.50	3.15	1.78	1.35	1.04
95	10.50	5.6	2.7	1.8	1.1

(Adaptado de Donald Knuth, *The Art of Computer Programming*, © 1973, Addison-Wesley, Reading, Mass., página 536. Reproducido con permiso.)

entre el registro y su dirección base. Si este vínculo se rompe, el registro ya no será accesible por dispersión.

Deben tomarse en cuenta estas necesidades especiales cuando se escriban programas que trabajen con archivos con dispersión.

ESTRUCTURA DEL COMPARTIMIENTO. La única diferencia entre un archivo con compartimientos y uno en el que cada dirección puede almacenar sólo una llave es que en el primero cada dirección tiene suficiente espacio para almacenar más de un registro lógico. Todos los registros que se guardan en el mismo compartimiento comparten la misma dirección. Por ejemplo, supóngase que se quiere almacenar cinco nombres en un compartimiento. He aquí tres de tales compartimientos con diferentes números de registros.

Compartimiento vacío:	0					
-----------------------	---	--	--	--	--	--

Dos entradas:	2	JONES	ARNSWORTH			
---------------	---	-------	-----------	--	--	--

Compartimiento lleno:	5	JONES	ARNSWORTH	STOCKTON	BRICE	THROOP
-----------------------	---	-------	-----------	----------	-------	--------

Cada compartimiento contiene un *contador* que mantiene información sobre cuántos registros se han almacenado. Las colisiones pueden ocurrir sólo cuando la adición de un registro nuevo hace que el contador exceda el número de registros que un compartimiento puede almacenar.

El contador señala cuántos registros están almacenados en un compartimiento, pero no señala cuáles espacios están ocupados y cuáles no. Se necesita una forma de señalar si una entrada está vacía. Una forma sencilla de hacerlo es usar un marcador especial para indicar un registro vacío, tal como se hizo anteriormente con los registros eliminados. En la ilustración anterior se usa el valor de llave //// para marcar los registros vacíos.

INICIO DE UN ARCHIVO PARA DISPERSION. Como el tamaño lógico de un archivo con dispersión debe permanecer fijo, en la mayoría de los casos es conveniente asignarle espacio físico antes de empezar a almacenar registros en él. Por lo general, para esto se crea un archivo con espacios vacíos para todos los registros y después se llenan los espacios con los registros de datos según sea necesario. (No es preciso construir un archivo con registros vacíos antes de colocar los datos allí, pero hacerlo incrementa la probabilidad de que los registros se almacenen uno al lado del otro en el disco, evita el error que ocurre cuando se pretende leer un registro inexistente, y facilita el procesamiento secuencial del archivo, sin tener que tratar con registros vacíos en alguna forma especial.)

CARGA DE UN ARCHIVO CON DISPERSION. Un programa para cargar un archivo con dispersión es similar en muchos aspectos a los programas anteriores que se usan para cargar archivos con registros de longitud fija, excepto en dos aspectos. Primero, el programa usa la función *dispersión()* para producir una dirección base para cada llave. Segundo, el programa busca un espacio libre para el registro comenzando con el compartimiento almacenado en su dirección base, y después, si el compartimiento nativo está lleno, continúa con compartimientos sucesivos hasta que encuentre uno que no esté lleno. El registro nuevo se inserta en este compartimiento, el cual se vuelve a transcribir al archivo en el lugar desde el cual se carga.

Si al buscar un compartimiento vacío un programa de carga rebasa la dirección máxima permitida, debe volver a la dirección de inicio. En la carga de un archivo con dispersión ocurre un problema potencial cuando se han cargado tantos registros en el archivo que ya no hay espacios vacíos. Una ingenua búsqueda de un espacio vacío con facilidad puede dar como resultado un ciclo infinito. Evidentemente, es deseable prevenir que esto ocurra haciendo que el programa se asegure de que cada registro nuevo tiene un lugar seguro en alguna parte del archivo.

Otro problema que aparece con frecuencia cuando se agregan registros a los archivos sucede cuando se intenta cargar un registro que ya está almacenado en el archivo. Si existe peligro de que ocurran llaves duplicadas, y no se permiten llaves duplicadas en el archivo, debe encontrarse algún mecanismo para solucionar este problema.

11.7

LA OPERACION DE ELIMINACION

Eliminar un registro de un archivo con dispersión es más complejo que agregarlo, por dos razones:

- No debe permitirse que el espacio liberado por la eliminación obstaculice las búsquedas posteriores, y
- Debe ser posible reutilizar el espacio liberado para las adiciones posteriores.

Cuando se usa la saturación progresiva, la búsqueda de un registro termina si se encuentra una dirección vacía. Por ello, no es conveniente dejar direcciones vacías que terminen la búsqueda por saturación en forma inapropiada. El siguiente ejemplo ilustra el problema.

Adams, Jones, Morris y Smith se almacenan en un archivo con dispersión en el cual cada dirección puede almacenar un registro. Tanto Adams como Smith se asignan a la dirección 5, y Jones y Morris a la dirección 6. Si se cargan en orden alfabético usando saturación progresiva

Registro	Dirección base	Dirección real	:	:
Adams	5	5	4	
Jones	6	6	5	Adams . . .
Morris	6	7	6	Jones . . .
Smith	5	8	7	Morris . . .
			8	Smith . . .
			:	
			:	

FIGURA 11.9 • Organización del archivo antes de las eliminaciones.

4	
5	Adams . . .
6	Jones . . .
7	
8	Smith . . .
9	
10	

FIGURA 11.10 • Misma organización de la figura 11.9, pero eliminando Morris.

para las colisiones, se almacenan en las localidades mostradas en la figura 11.9.

La búsqueda de Smith comienza en la dirección 5 (su dirección base), busca Smith sucesivamente en las direcciones 6, 7 y 8, y lo encuentra en la 8. Ahora supóngase que Morris se elimina y queda un espacio vacío, como se ilustra en la figura 11.10. La búsqueda de Smith se inicia de nuevo en la dirección 5 y después busca en las direcciones 6 y 7; pero como la dirección 7 ahora está vacía, para el programa es razonable concluir que el registro de Smith no está en el archivo.

11.7.1 MARCAS DE INUTILIZACION PARA ELIMINACIONES

En el capítulo 5 se analizaron técnicas para abordar el problema de la eliminación. Una técnica sencilla usada para identificar registros eliminados reemplaza el registro eliminado (o sólo su llave) con un marcador que indique que alguna vez hubo un registro, pero ya no. Tal marcador se denomina algunas veces marca de *inutilización* (*tombstone*, “lápida”, en inglés) (Wiederhold 1983). El atractivo que tiene el uso de marcas de inutilización es que resuelve los dos problemas descritos previamente:

- El espacio liberado no rompe la secuencia de búsquedas de un registro, y
- El espacio liberado obviamente está disponible y puede ser usado para adiciones posteriores.

5	Adams . . .
6	Jones . . .
7	#####
8	Smith . . .

FIGURA 11.11 • Mismo archivo que el de la figura 11.9, después de la inserción de una marca de inutilización para Morris.

La figura 11.11 ilustra cómo se ve el archivo del ejemplo después de que se inserta la marca de inutilización ##### para el registro eliminado. Ahora la búsqueda de Smith no se detiene en el registro vacío número 7, sino que usa ##### como indicación de que la búsqueda debe continuar.

No es necesario insertar marcas de inutilización cada vez que ocurre una eliminación. Por ejemplo, supóngase que en el ejemplo anterior se elimina el registro de Smith. Como el espacio que sigue del registro Smith está vacío, no se pierde nada con marcar el lugar de Smith como vacío en vez de insertar una marca de inutilización. En verdad, sería inapropiado insertar una marca de inutilización donde no se necesita. (Si después de colocar una marca de inutilización innecesaria en el lugar de Smith se agrega un registro en la dirección 9, ¿cómo afectaría esto a una búsqueda de Smith subsecuente sin éxito?)

11.7.2 IMPLICACIONES DE LAS MARCAS DE INUTILIZACION PARA LAS INSERCIIONES

Con el uso de las marcas de inutilización, la *inserción* de registros se vuelve ligeramente más difícil de lo que implicaban los análisis previos. Puesto que los programas que efectúan el cargado inicial únicamente buscan la primera ocurrencia de un espacio vacío (señalado por la presencia de la llave ////), ahora es permisible insertar un registro en donde ocurran como llave tanto el signo //// como el signo #####.

Esta nueva característica, deseable porque provoca una mayor longitud media de búsqueda, acarrea cierto peligro. Por ejemplo, considérese

el ejemplo anterior, en el que se elimina Morris, dando una organización de archivo como la que se muestra en figura 11.11. Ahora supóngase que se quiere que un programa inserte Smith en el archivo. Si el programa simplemente busca hasta que encuentra un #####, nunca notará que Smith ya está en el archivo. Es casi seguro que no se quiere colocar un segundo registro Smith en el archivo. Puesto que hacerlo significaría que las búsquedas posteriores nunca encontrarían el registro de Smith anterior. Para que esto no ocurra, el programa debe examinar todo el cúmulo de llaves contiguas y marcas de inutilización a fin de asegurarse de que no existen llaves duplicadas, y después regresar e insertar el registro en la primera marca de inutilización disponible, si es que existe alguna.

11.7.3 EFECTOS DE LAS ELIMINACIONES Y ADICIONES EN EL DESEMPEÑO

El uso de marcas de inutilización permite que los algoritmos de búsqueda trabajen y ayuda en la recuperación de almacenamiento, pero aun así se puede esperar cierto deterioro en el desempeño después de varias eliminaciones e inserciones en el archivo.

Por ejemplo, considérese el pequeño archivo de cuatro registros de Adams, Jones, Smith y Morris. Después de eliminar Morris, Smith está un espacio más adelante de lo necesario de su dirección base. Si la marca de inutilización nunca se usa para almacenar otro registro, cada extracción de Smith requiere un acceso más de lo absolutamente necesario. En general, después de un gran número de inserciones y eliminaciones puede esperarse encontrar muchas marcas de inutilización ocupando lugares que podrían ser ocupados por registros cuyas direcciones base los preceden, pero que están almacenados después de ellas. En efecto, cada marca de inutilización representa una oportunidad desaprovechada de reducir en uno el número de localidades que deben examinarse mientras se buscan estos registros.

Algunos estudios experimentales muestran que después de un 50 a un 150 por ciento de movimientos de registros, un archivo con dispersión alcanza un punto de equilibrio, de modo que es igualmente probable que la longitud media de búsqueda mejore o empeore (Bradley, 1982; Peterson, 1957). Sin embargo, para ese entonces el desempeño de la búsqueda se ha deteriorado a tal punto que el registro medio está tres veces más lejano (en términos de accesos) de su dirección base que lo que podría estar después del cargado inicial. Esto significa, por ejemplo, que si después del cargado inicial la longitud media de búsqueda es 1.2, después de alcanzar el punto de equilibrio será de alrededor de 1.6.

Existen tres tipos de soluciones para el problema del deterioro de las longitudes medias de búsqueda. Una implica hacer un poco de

reorganización local cada vez que suceda una eliminación. Por ejemplo, el algoritmo de eliminación puede examinar los registros que siguen a una marca de inutilización para saber si la longitud de búsqueda puede acortarse moviendo el registro hacia atrás, rumbo a su dirección base. Otra solución implica la reorganización completa del archivo después de que la longitud media de búsqueda alcanza un valor inaceptable. Una tercera posibilidad es usar un algoritmo de resolución de colisiones por completo diferente.

11.8

OTRAS TECNICAS DE RESOLUCION DE COLISIONES

A pesar de su simplicidad, la dispersión aleatoria mediante saturación progresiva con tamaños de compartimiento razonables por lo general se desempeña bien. Sin embargo, si no funciona como se requiere, existen algunas variaciones que pueden desempeñarse aún mejor. En esta sección se analizan algunas con las que se puede mejorar el desempeño de la dispersión cuando se usa almacenamiento externo.

11.8.1 DISPERSION DOBLE

Uno de los problemas de la saturación progresiva es que, cuando muchos registros se dispersan a compartimientos en la misma vecindad, pueden formarse cúmulos de registros. Conforme la densidad de empaquetamiento se aproxima a uno, esta acumulación tiende a provocar búsquedas extremadamente largas para algunos registros. Un método de evitar el acumulamiento consiste en almacenar los registros en saturación a una gran distancia de sus direcciones base mediante *dispersión doble*, lo cual significa que, cuando sucede una colisión, se aplica una segunda función de dispersión a la llave para producir un número c que es el primo relativo al número de direcciones.[†] El valor c se agrega a la dirección base para producir la dirección en saturación. Si ya está ocupada, se le agrega c para producir otra dirección en saturación. Este procedimiento continúa hasta que se encuentra una saturación que esté libre.

La dispersión doble tiende a esparcir los registros en el archivo, pero adolece de un problema potencial que se presenta en varios métodos mejorados de saturación: viola la localidad al trasladar deliberadamente los registros en saturación a cierta distancia de sus direcciones base, incrementando así la probabilidad de que el disco necesite tiempo

[†] Si N es el número de direcciones, entonces c y N son primos relativos si no tienen divisores comunes.

adicional para obtener la nueva dirección en saturación. Si el archivo cubre más de un cilindro, esto podría requerir un costoso movimiento adicional de la cabeza. Los programas con dispersión doble pueden resolver este problema si son capaces de generar direcciones en saturación de tal modo que los registros en saturación se mantengan en el mismo cilindro que los registros con direcciones base.

11.8.2 SATURACION PROGRESIVA ENCADENADA

La saturación progresiva encadenada es otra técnica diseñada para evitar los problemas causados por el acumulamiento. Funciona de la misma manera que la saturación progresiva, excepto que los sinónimos se ligan con apuntadores. Esto es, cada dirección base contiene un número que indica el lugar del siguiente registro con la misma dirección base. El siguiente registro contiene, a su vez, un apuntador al siguiente registro con la misma dirección base, y así sucesivamente. El efecto neto de esto es que para cada conjunto de sinónimos hay una lista ligada que conecta sus registros, y es esta lista en la que se busca cuando se requiere un registro.

La ventaja de la saturación progresiva encadenada sobre la sencilla es que en cualquier búsqueda sólo se necesita acceder a los registros con llaves que son sinónimos. Por cada ejemplo, supóngase que el conjunto de llaves que se muestra en la figura 11.12 se carga en el orden expuesto a un archivo de dispersión con tamaño de compartimiento uno, y que se usa la saturación progresiva. La búsqueda de Cole implica un acceso a Adams (un sinónimo) y a Bates (que no es un sinónimo). Flint, el peor caso, requiere seis accesos, sólo dos de los cuales implican sinónimos.

Puesto que Adams, Cole y Flint son sinónimos, un algoritmo de encadenamiento forma una lista ligada que conecta esos tres nombres, con Adams a la cabeza de la lista. Como Bates y Dean también son

Llave	Dirección base	Dirección real	Longitud de búsqueda
Adams	20	20	1
Bates	21	21	1
Cole	20	22	3
Dean	21	23	3
Evans	24	24	1
Flint	20	25	6
Longitud de búsqueda = $(1 + 1 + 3 + 3 + 1 + 6) / 6 = 2.5$			

FIGURA 11.12• Dispersión con saturación progresiva.

sinónimos, forman un segunda lista. Esta disposición se muestra en la figura 11.13. La longitud media de búsqueda decrece de 2.5 a

$$(1 + 1 + 2 + 2 + 1 + 3)/6 = 1.7.$$

El uso de la saturación progresiva encadenada implica atender algunos detalles que no se requieren en la saturación progresiva simple. Primero, debe agregarse un *campo de liga* a cada registro, lo que requiere un poco más de almacenamiento. Segundo, un algoritmo de encadenamiento debe garantizar la posibilidad de llegar a cualquier sinónimo partiendo de su dirección base. Este último requisito no es trivial, como lo muestra el siguiente ejemplo.

Supóngase que en el ejemplo la dirección base de Dean es 22 en lugar de 21. Cuando se carga Dean la dirección 22 ya está ocupada por Cole, de modo que va a parar a la dirección 23. ¿Significa esto que el apuntador de Cole debe apuntar a 23 (la dirección real de Dean) o a 25 (la dirección del sinónimo Flint de Cole)? Si el apuntador es 25, la lista ligada que agrupa a Adams, Cole y Flint permanece intacta, pero se pierde Dean. Si el apuntador es 23, se pierde Flint.

El problema aquí es que una dirección (22) que debe ser ocupada por un registro base (Dean) está ocupada por un registro diferente. Una solución es procurar que toda dirección que se califique como dirección

Dirección base	Dirección real	Datos	Dirección del siguiente sinónimo	Longitud de búsqueda
20	20	Adams . . .	22	1
21	21	Bates . . .	23	1
20	22	Cole . . .	25	2
21	23	Dean . . .	-1	2
24	24	Evans . . .	-1	1
20	25	Flint . . .	-1	3
			:	

FIGURA 11.13 • Dispersión con saturación progresiva encadenada. Adams, Cole y Flint son sinónimos; Bates y Dean son sinónimos.

base para algún registro del archivo realmente almacene un registro base. El problema es fácil de manejar si el archivo se carga desde el principio usando una técnica de carga en dos pasos.

La *carga en dos pasos*, como indica su nombre, implica cargar un archivo de dispersión en dos pasos. En el primer paso sólo se cargan los registros con direcciones base. Los registros que no son base se guardan en un archivo separado. Esto garantiza que ninguna dirección base potencial esté ocupada por registros en saturación. En el segundo paso se carga y almacena cada registro en saturación en una de las direcciones libres, de acuerdo con la técnica de solución de colisiones que se esté usando.

La carga en dos pasos garantiza que toda la dirección base potencial sea en realidad una dirección base, de modo que se resuelve el problema del ejemplo. Sin embargo, no se garantiza que las eliminaciones e inserciones posteriores no vuelvan a crear el mismo problema. Mientras se usa el archivo para almacenar registros en direcciones base y también en saturación, permanecerá el problema de que los registros en saturación desplacen a los nuevos que se asignan a una dirección ocupada por uno en saturación.

Los métodos usados para eliminar estos problemas después de la carga inicial son un tanto complejos y, en un archivo muy volátil, pueden requerir muchos accesos adicionales a disco. (Para más información sobre las técnicas de mantenimiento de apuntadores, véase Knuth [1973b] y Bradley [1982]. Sería bueno poder evitar de algún modo todo este problema de las listas de saturación que se topan unas con otras y esto es lo que hace el siguiente método.)

11.8.3 ENCADENAMIENTO CON UN AREA DE SATURACION SEPARADA

Una forma de evitar que los registros en saturación ocupen las direcciones base en donde no deben estar es moverlos a un área de saturación separada. Muchos esquemas de dispersión son variaciones de este enfoque básico. Al conjunto de direcciones base se le llama *área principal de datos*, y al conjunto de direcciones de saturación se le llama *área de saturación*. La ventaja de este enfoque es que mantiene todas las direcciones base potenciales que no se han usado libres para inserciones posteriores.

En términos del archivo que se examinó en la sección de Cole, Dean y Flint se habrían almacenado en un área de saturación separada y no en direcciones base potenciales para registros que lleguen más tarde (Fig. 11.14). Ahora no ocurre ningún problema cuando se agrega un registro nuevo. Si hay lugar para su dirección base, se almacena ahí. Si no, se mueve al archivo de saturación, donde se agrega a la lista ligada que comienza en la dirección base.

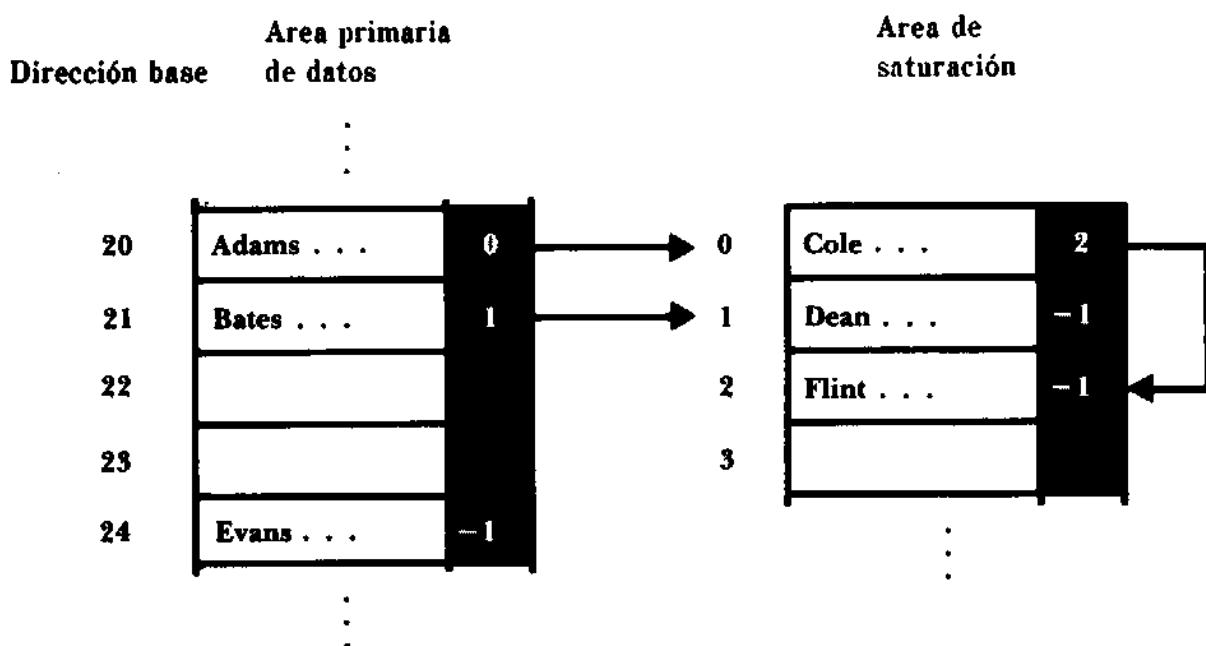


FIGURA 11.14 • Encadenamiento a un área separada de saturación. Adams, Cole y Flint son sinónimos; Bates y Dean son sinónimos.

Si el tamaño de compartimiento para archivo primario es lo suficientemente grande para evitar números excesivos de registros en saturación, el archivo de saturación puede ser un archivo sencillo de entradas secuenciales con tamaño de compartimiento uno. Puede asignarse espacio a los registros en saturación sólo cuando sea necesario.

El uso de un área de saturación separada simplifica un poco el proceso, y parecería que mejora el desempeño, en especial cuando ocurren muchas inserciones y eliminaciones. Sin embargo, no siempre es así. Si el área de saturación separada está en un cilindro diferente del de la dirección base, toda búsqueda de un registro en saturación implicará un movimiento de cabeza muy costoso. Los estudios muestran que por lo general el tiempo de acceso real es peor cuando los registros en saturación están almacenados en un área de saturación separada que cuando se almacenan en el área de saturación principal (Lum, 1971).

Una situación en la que se requiere un área de saturación separada se presenta cuando la densidad de empaquetamiento es mayor de uno: hay más registros que direcciones base. Si, por ejemplo, se anticipa que un archivo rebasará la capacidad del conjunto inicial de direcciones base y que no es razonable volver a dispersar el archivo con un espacio de direcciones más grande, entonces debe usarse un área de saturación separada.

11.8.4 TABLAS DE DISPERSION: RECONSIDERACION DE LA INDIZACION

Supóngase que se tiene un archivo de dispersión que no contiene registros, sólo apunadores a registros. El archivo obviamente sólo es un índice donde se busca por dispersión en lugar de usar otro método. Este método de organización de archivos se conoce como tabla de dispersión (Severance, 1974), y en la figura 11.15 se ilustra la forma de organizar un archivo con este tipo de tablas.

La organización por tablas de dispersión ofrece muchas de las ventajas que por lo regular proporciona la indización simple, con la ventaja adicional de que la búsqueda del índice mismo requiere sólo un acceso. (Por supuesto que ese acceso es uno más que los que requieren otras formas de dispersión, a menos que la tabla de dispersión pueda almacenarse en la memoria principal.) El archivo de datos puede ser de diferentes formas. Por ejemplo, puede consistir en un conjunto de listas ligadas de sinónimos (como se muestra en la figura 11.15), un archivo clasificado, o un archivo de entradas secuenciales. Además las tablas de dispersión manejan convenientemente el uso de registros de longitud variable. Para más información sobre las tablas de dispersión, véanse Severance [1974] y Teorey y Fry [1982].

11.8.5 DISPERSION EXTENSIBLE

Todas las técnicas estudiadas hasta aquí se basan en la idea de que el número original de posibles direcciones de dispersión debe permanecer constante. Si el archivo se llena tanto, o las cadenas de saturación se vuelven tan largas que la respuesta resulta inadmisible, se tiene que incrementar el espacio de direcciones y, usando una nueva función de dispersión, redispersar el archivo completo. Aplicar de nuevo la dispersión puede ser muy costoso, porque implica mover en forma aleatoria cada registro del archivo. Cuando nos encontramos con los altos costos de la reorganización de un archivo aleatorio, en el capítulo 6, se buscó una manera de mover registros sin moverlos en realidad y finalmente se construyó un índice. Podría reorganizarse el índice dejando los registros en su lugar.

¿Puede aplicarse la misma idea a la dispersión? Con las tablas de dispersión ya se tiene el índice, así que lo único que se necesita es una forma eficiente de reorganizar las tablas de dispersión sin mover demasiado los registros de datos. Se han propuesto varios esquemas para organizar y reorganizar las tablas de dispersión. Una técnica relativamente sencilla (Ullman, 1982) implica doblar el tamaño de la

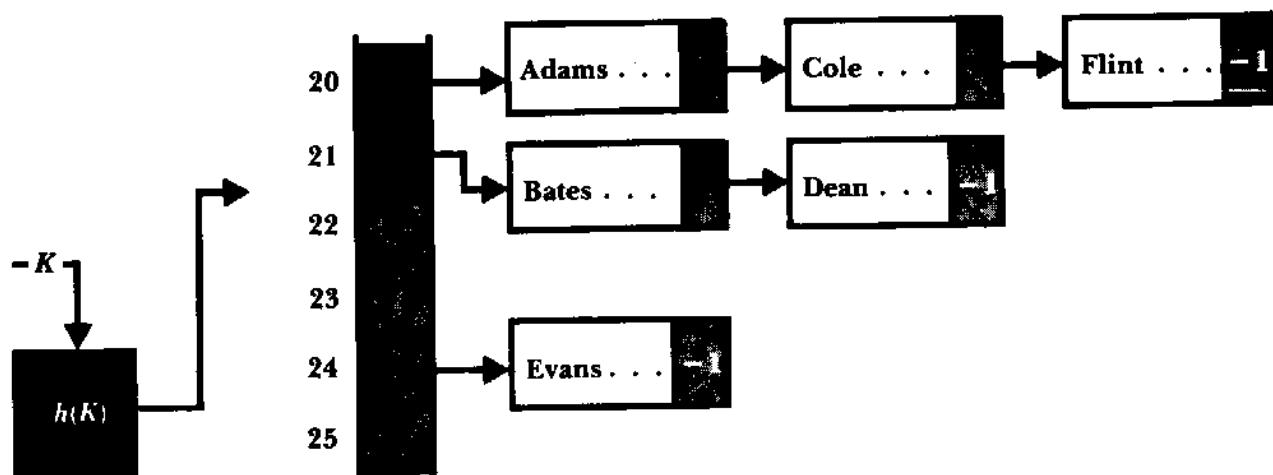


FIGURA 11.15 • Ejemplo de una estructura de tabla de dispersión. Como la parte dispersada en un índice, el archivo de datos puede organizarse de cualquier forma que sea apropiada.

tabla de dispersión cuando sea necesario *redispersar* el archivo. Si se cumplen las siguientes restricciones, la técnica funciona con razonable eficiencia.

- La función de dispersión debe generar un entero mucho mayor que el número máximo de direcciones que se usarán; luego hay que dividir este número entre el tamaño actual de direcciones y tomar el residuo.
- Cuando se reorganiza la tabla, el número de direcciones debe multiplicarse por un entero fijo, comúnmente 2 (la tabla de dispersión duplica su tamaño).

Si la tabla se redispersa doblando su tamaño de n a $2n$, por ejemplo, los registros que originalmente se dispersaban a la dirección i ahora se dispersan a i o a $i + n$, y ningún registro de otros compartimientos se dispersa a i o a $i + n$. Se necesita dividir sólo aquellos compartimientos que contengan registros en saturación (todos los demás pueden encontrarse en su dirección original), por lo que se desperdicia muy poco tiempo en el proceso de reorganización de la tabla original.

Han surgido varias técnicas a partir de la idea de que la tabla de dispersión puede ser un índice con apuntadores a compartimientos que contienen los registros de datos reales. Los términos *dispersión extensible* y *dispersión virtual* se aplican algunas veces a este tipo de técnicas de dispersión. Puede encontrarse información adicional sobre los aspectos de la dispersión extensible en Fagin *et al.* [1979], Faloutsos [1985] y Scholl [1981].

11.9

PATRONES DE ACCESO A LOS REGISTROS

El veinte por ciento de los pescadores atrapan el 80 por ciento del pescado.

El veinte por ciento de los ladrones roban el 80 por ciento del botín.

L.M. Boyd

El uso de diferentes técnicas de solución de colisiones no es la única ni necesariamente la mejor manera de elevar el desempeño en un archivo con dispersión. Si se sabe algo, por ejemplo, acerca de los patrones de acceso a los registros, entonces se hace posible usar técnicas sencillas de saturación progresiva que logran un muy buen desempeño.

Supóngase que se tiene una tienda con 10 000 categorías diferentes de artículos, y que se tiene en el computador un archivo de dispersión de inventarios con un registro por cada uno de los 10 000 artículos que maneja la compañía. Cada vez que se vende un artículo, debe accederse al registro que corresponde a ese artículo. Como el archivo está dispersado, es razonable suponer que los 10 000 registros están distribuidos en forma aleatoria entre las direcciones disponibles que componen el archivo. ¿También es razonable suponer que la distribución de accesos a los registros en el inventario está distribuida aleatoriamente? Tal vez no. Por ejemplo, la leche se solicita con mucha frecuencia; el queso *brie*, rara vez.

Existe un principio usado por los economistas llamado Principio de Pareto, o el "concepto de los pocos importantes y los muchos triviales", lo cual, en términos de archivos, quiere decir que un pequeño porcentaje de los registros de un archivo emplea un porcentaje grande de los accesos. Una versión popular del Principio de Pareto es la regla empírica del 80/20: el 80 por ciento de los accesos se realiza sobre el 20 por ciento de los registros. En el archivo de víveres, la leche estaría entre el 20 por ciento de los artículos de alta actividad y el queso *brie* entre el resto.

Sólo se puede sacar partido del principio 80/20 en una estructura de archivos si se conoce algo acerca de la distribución probable de los accesos a los registros. Una vez que se tiene esta información, se necesita encontrar una forma de colocar los artículos de alta actividad donde puedan encontrarse mediante el menor número posible de accesos. Si se pueden cargar los artículos en el archivo de tal forma que el 20 por ciento (más o menos) de aquellos a los que se accederá con mayor probabilidad se carguen en sus direcciones base o cerca de ellas, entonces la mayoría de las transacciones accederán a registros con longitudes de búsqueda cortas, y la longitud media de búsqueda efectiva será más corta que la nominal que se definió anteriormente.

Por ejemplo, supóngase que el programa que maneja el archivo de la tienda mantiene información del número de veces que se accede a cada artículo durante un periodo de un mes. Puede hacer esto almacenando con cada registro un contador que se inicia en cero y se incrementa cada vez que se accede al artículo. Al final del mes se vierten los registros de todos los artículos del inventario en un archivo que se clasifica en orden descendente de acuerdo con el número de veces que se hayan solicitado. Cuando el archivo clasificado se redispersa y se vuelve a cargar, los primeros registros que se cargarán son los que de acuerdo con la experiencia del mes anterior se solicitarán con mayor probabilidad. Como son los primeros que se cargan, también son los más factibles de ser cargados en sus direcciones base. Si se usan compartimientos de tamaño razonable, no habrá, o serán *muy pocos* los artículos con alta actividad que no estén en sus direcciones base y, por tanto, que no se extraigan en un acceso.

RESUMEN

Existen tres modos principales de acceder a los archivos: en forma secuencial, a través de un índice, y de manera directa. La dispersión representa la principal forma de organización de archivos para permitir el acceso directo.

La dispersión puede proporcionar un acceso más rápido que la mayoría de las otras organizaciones que se han estudiado, y por lo regular con muy poco desperdicio de almacenamiento; además es adaptable a casi todo tipo de llaves primarias. En teoría, la dispersión hace posible encontrar cualquier registro con sólo un acceso al disco, pero este ideal se logra rara vez. La desventaja principal de la dispersión es que los archivos con dispersión no pueden clasificarse por llave.

La dispersión implica la aplicación de una función de dispersión $h(K)$ a una llave de registro K para producir una dirección. La dirección se toma como la dirección base del registro cuya llave es K , y forma la base para la búsqueda del registro. Las direcciones producidas por las funciones de dispersión por lo general parecen ser aleatorias.

Cuando dos o más llaves se dispersan a la misma dirección se les llama *sinónimos*. Si en una dirección no caben todos sus sinónimos sobrevienen las *colisiones*, y esto hace que algunos de los sinónimos no puedan almacenarse en la dirección base y deban estar en algún otro lugar. Como las búsquedas de registros comienzan en sus direcciones base, las búsquedas de registros que no están en sus direcciones base

por lo general implican accesos adicionales al disco. El término *longitud media de búsqueda* describe el número promedio de accesos a disco que se requieren para extraer un registro. La longitud media de búsqueda ideal es 1.

Gran parte del estudio de la dispersión se refiere a las técnicas para disminuir el número y los efectos de las colisiones. En este capítulo se examinan tres métodos generales para reducir el número de colisiones:

- Esparcir los registros;
- Usar memoria adicional, y
- Usar compartimientos.

Esparcir los registros implica elegir una función de dispersión que distribuya los registros en el espacio de direcciones al menos en forma aleatoria. Una *distribución uniforme* esparce los registros en forma equitativa, evitando así la ocurrencia de colisiones. Una distribución *aleatoria* o casi aleatoria es mucho más fácil de lograr, y usualmente se considera aceptable.

En este capítulo se desarrolla un algoritmo sencillo de dispersión para mostrar los tipos de operaciones que se realizan en un algoritmo de dispersión. Los tres pasos del algoritmo son:

1. Representar la llave en forma numérica.
2. Desglosar y sumar.
3. Dividir entre el tamaño del espacio de direcciones, produciendo una dirección válida.

Cuando se examinan varios tipos diferentes de algoritmos de dispersión, se observa que a veces pueden encontrarse algunos que produzcan distribuciones *mejor que aleatorias*. Pero, en su defecto, se sugieren algunos algoritmos que por lo general producen distribuciones que son aproximadamente aleatorias.

La *distribución de Poisson* proporciona una herramienta matemática para examinar en detalle los efectos de una distribución aleatoria. Las funciones de Poisson pueden usarse para predecir los números de direcciones a las que es probable asignar 0, 1, 2 o más registros, considerando el número de registros que van a dispersarse y el número de direcciones disponibles. Esto permite predecir el número de colisiones que pueden ocurrir cuando se disperse el archivo, el número probable de registros en saturación, y algunas veces la longitud media de búsqueda.

Usar memoria adicional es otra forma de evitar colisiones. Cuando se dispersa un número fijo de llaves, la probabilidad de que ocurran sinónimos decrece conforme se incrementa el número de direcciones

posibles. Por tanto, una organización de archivos que asigna muchas más direcciones de las que se usarán tiene menos sinónimos que aquella que asigna pocas direcciones adicionales. El término *densidad de empaquetamiento* describe la proporción del espacio de direcciones disponible que realmente está almacenando registros. La función de Poisson se usa para determinar cómo influyen las diferencias en la densidad de empaquetamiento sobre el porcentaje de registros que probablemente serán sinónimos.

Usar *compartimientos* es el tercer método para evitar colisiones. Las direcciones del archivo pueden almacenar uno o más registros, dependiendo de cómo organizó el archivo el diseñador. El número de registros que pueden almacenarse en una dirección dada, llamado *tamaño del compartimiento* determina el punto en el cual los registros asignados a la dirección provocarán saturación. Puede usarse la función de Poisson para explorar los efectos de las variaciones en los tamaños de los compartimientos y las densidades de empaquetamiento. Los compartimientos grandes, combinados con una densidad de empaquetamiento baja, pueden dar como resultado longitudes medias de búsqueda muy pequeñas.

Aunque el número de colisiones puede reducirse, se necesita algún medio para enfrentarse a ellas cuando ocurran. Se examinó en detalle una técnica sencilla de solución de colisiones: la *saturación progresiva*. Si el intento de almacenar un registro nuevo da como resultado una colisión, la saturación progresiva implica la búsqueda ordenada en las direcciones que siguen de la dirección base del registro, hasta que se encuentre una para almacenar el registro nuevo. Si se busca un registro y no se encuentra en su dirección base, se busca en direcciones sucesivas hasta que se encuentra el registro o bien una dirección vacía.

La saturación progresiva es sencilla y algunas veces funciona muy bien; sin embargo, crea longitudes de búsqueda grandes cuando la densidad de empaquetamiento es alta y el tamaño del compartimiento es reducido. En ocasiones también produce cúmulos de registros, creando longitudes de búsqueda muy grandes para registros nuevos cuyas direcciones base estén en los cúmulos.

Hay tres problemas asociados con la eliminación de registros en los archivos con dispersión, y son:

1. La posibilidad de que los espacios vacíos creados por las eliminaciones impidan búsquedas posteriores de registros en saturación.
2. La necesidad de recuperar el espacio que queda disponible cuando se eliminan registros.
3. El deterioro de las longitudes medias de búsqueda causado por los espacios vacíos, los cuales mantienen a los registros más adelante de lo necesario de su dirección base.

Los primeros dos problemas pueden resolverse usando *marcas de inutilización* a fin de identificar los espacios que están vacíos (y que pueden reutilizarse para registros nuevos), pero no deben detener la búsqueda de un registro. Las soluciones al problema del deterioro incluyen la reorganización local, la reorganización completa del archivo, y la elección de un algoritmo de resolución de colisiones que no provoque deterioro.

Como los registros en saturación tienen un influencia decisiva en el desempeño, se han propuesto muchas técnicas diferentes de manejo de la saturación. Cinco de las técnicas que resultan apropiadas para aplicaciones de archivos se analizan brevemente:

1. *La dispersión doble* reduce el acumulamiento local, pero puede colocar algunos registros en saturación a gran distancia de su dirección nativa, por lo que se necesitan desplazamientos adicionales.
2. *La saturación progresiva encadenada* reduce las longitudes de búsqueda porque exige que sólo se examinen los sinónimos cuando se está buscando un registro. Para que funcione, toda dirección que pueda ser un registro base para algún registro del archivo debe almacenar un registro base. Se analizan los mecanismos que sirven para asegurar que esto ocurra.
3. *El encadenamiento con un área de saturación separada* simplifica considerablemente el encadenamiento, y tiene la ventaja de que el área de saturación puede ser organizada en formas más apropiadas para el manejo de los registros en saturación. Un peligro de este método es que se puede perder localidad.
4. *Las tablas de dispersión* combinan la indización con la dispersión. Este método proporciona mucha más flexibilidad en la organización de archivos de datos. Una desventaja de las tablas de dispersión es que, a menos que el índice pueda almacenarse en memoria RAM, requiere un acceso adicional al disco para cada búsqueda.
5. *La dispersión extensible* también combina la indización con la dispersión, pero permite que el tamaño del espacio de direcciones crezca dinámicamente en tanto crece el archivo y el desempeño comienza a deteriorarse.

Cómo en muchos casos se accede a algunos registros con más frecuencia que a otros (la *regla empírica del 80/20*), a menudo vale la pena tomar en cuenta los patrones de acceso. Si pueden identificarse aquellos registros a los cuales se accede con mayor probabilidad, pueden tomarse precauciones para asegurarse que estén almacenados

más cerca de la dirección base que aquellos a los cuales se accede con menor frecuencia; esto disminuye la longitud media de búsqueda efectiva. Una de tales medidas es cargar los registros a los cuales se accede con mayor frecuencia antes que los otros.

TERMINOS CLAVE

Aleatorización. Producir un número (p. ej., por dispersión) que parezca ser aleatorio.

Colisión. Situación en la que un registro es asignado a una dirección que no tiene suficiente lugar para almacenarlo. Cuando ocurre una colisión deben encontrarse medios para resolverla.

Compartimiento. Área de espacio en el archivo que se trata como un registro físico para propósitos de almacenamiento y extracción, pero que puede almacenar varios registros *lógicos*. Al almacenar y extraer registros lógicos en compartimientos en vez de hacerlo individualmente, los tiempos de acceso pueden mejorar considerablemente en muchos casos.

Densidad de empaquetamiento. Proporción de espacio del archivo asignado que en realidad almacena registros. (Algunas veces se denomina *factor de carga*). Si un archivo está medio lleno, su densidad de empaquetamiento es 50 por ciento. La densidad de empaquetamiento y el tamaño del compartimiento son las dos medidas más importantes en la determinación de la probabilidad de que ocurra una colisión cuando se busca un registro en un archivo.

Desglosar y sumar. Método de dispersión en el que codificaciones de las partes de tamaño fijo de una llave se extraen (p. ej., cada dos bytes) y se suman. La suma resultante puede usarse para producir una dirección.

Dirección base. Dirección generada por una función de dispersión para una llave dada. Si un registro está almacenado en su dirección base, entonces la longitud de búsqueda del registro es uno, porque sólo se requiere un acceso para extraerlo. Para extraer o almacenar un registro que no está en su dirección base se requiere más de un acceso.

Dirección vacía. Véase *Saturación progresiva*.

Dispersión. Técnica para generar un dirección base única para una llave dada. La dispersión se usa cuando se requiere acceso rápido a una llave (o a su registro correspondiente). En este

capítulo las aplicaciones de la dispersión implican el acceso directo a los registros de un archivo, pero la dispersión con frecuencia se usa también para acceder a datos en arreglos en memoria RAM. En la indización, por ejemplo, un índice puede estar organizado para la dispersión más que para la búsqueda binaria cuando se desea que la búsqueda sea extremadamente rápida.

Dispersión doble. Esquema con el cual se resuelven las colisiones aplicando una segunda función de dispersión a la llave para producir un número c , el cual se suma a la dirección original (módulo el número de direcciones) tantas veces como sea necesario hasta que se localice el registro deseado o se encuentre un espacio vacío. La dispersión doble evita parte del acumulamiento que ocurre con la saturación progresiva.

Dispersión extensible. Tipo de dispersión que permite que el tamaño de una tabla de dispersión crezca en forma dinámica sin tener que reorganizar la tabla entera cada vez que crece.

Dispersión indizada. En lugar de usar los resultados de una dispersión para producir la dirección de un registro, la dispersión puede usarse para identificar una localidad en un índice que a su vez apunte a la dirección del registro. A pesar de que en este método cada búsqueda requiere un acceso adicional, permite organizar los registros de datos reales de un modo que facilita otros tipos de procesamiento, como el procesamiento secuencial.

Dispersión mínima. Esquema de dispersión donde el número de direcciones es exactamente igual al número de registros. No se desperdicia espacio de almacenamiento.

Distribución de Poisson. Distribución generada por la función de Poisson, que puede usarse para aproximar la distribución de registros entre direcciones cuando la distribución es aleatoria. Una distribución de Poisson en particular depende de la proporción entre el número de registros y el número de direcciones disponibles. Una versión particular de la función de Poisson, $p(x)$, da la proporción de direcciones que tendrán x llaves asignadas. Véase *Mejor que aleatoria*.

División por número primo. Dividir un número entre un número primo y usar el residuo como dirección. Si se toma el tamaño de direcciones como un número primo p , un número grande puede transformarse en una dirección válida dividiéndolo entre p . En la dispersión, a menudo es preferible la división entre números primos que la división entre no primos, porque los primos tienden a producir residuos más aleatorios.

Función de dispersión perfecta. Función de dispersión que distribuye registros en forma uniforme, minimizando el número

de colisiones. Las funciones de dispersión perfecta son muy deseables, pero extremadamente difíciles de encontrar para conjuntos grandes de llaves.

Longitud media de búsqueda. Se define la longitud media de búsqueda como la *suma del número de accesos requeridos para cada registro en el archivo*, dividida entre *el número de registros del archivo*. Esta definición no toma en cuenta el número de accesos requeridos por búsquedas sin éxito, ni tampoco el hecho de que es más probable que se acceda a algunos registros con más frecuencia que a otros. Véase la *regla empírica de 80/20*.

Marca de inutilización. Marcador especial colocado en el campo llave de un registro para indicar que ya no es válido. El uso de marcas de inutilización resuelve dos problemas asociados con la eliminación de registros: que el espacio liberado no rompa la búsqueda secuencial de un registro, y que el espacio liberado sea fácilmente reconocido como disponible y pueda ser utilizado para adiciones posteriores.

Mejor que aleatoria. Este término se aplica a las distribuciones donde los registros se esparcen más uniformemente que si lo hiciera en forma aleatoria la función de dispersión. Por lo regular, la distribución producida por una función de dispersión es un poco mejor que aleatoria.

Método del medio cuadrado. Método de dispersión donde una representación de la llave se eleva al cuadrado y se usan dígitos de la mitad del resultado para producir la dirección.

Regla empírica del 80/20. La suposición de que un porcentaje grande (p. ej., 80 por ciento) de los accesos se efectúa sobre un pequeño porcentaje (p. ej., 20 por ciento) de los registros de un archivo. Cuando se aplica la regla del 80/20, la longitud media de búsqueda *efectiva* está determinada en gran parte por las longitudes de búsqueda de los registros más activos, de tal forma que los intentos de volver *estas* longitudes de búsqueda más cortas pueden dar como resultado un desempeño considerablemente mejor.

Saturación (desbordamiento). Situación que ocurre cuando un registro no puede almacenarse en su dirección base.

Saturación progresiva. Técnica de manejo de saturación en la que las colisiones se resuelven almacenando un registro en la siguiente dirección disponible después de su dirección base. La saturación progresiva no es la técnica de manejo de saturación más eficiente, pero es una de las más simples y es adecuada para muchas aplicaciones.

Sinónimos. Dos o más llaves diferentes que se asignan a la misma dirección base. Cuando cada dirección del archivo puede

almacenar un solo registro, los sinónimos siempre ocasionan colisiones. Si se usan compartimientos, pueden almacenarse sin colisiones varios registros cuyas llaves sean sinónimos.

Uniforme. Término aplicado a una distribución en la que los registros se esparcen en forma equitativa entre las direcciones. Los algoritmos que producen distribuciones uniformes son mejores que los que manejan la aleatoriedad: los primeros tienden a evitar la ocurrencia de algunas de las colisiones casuales que permitiría un algoritmo con aleatoriedad.

EJERCICIOS

1. Use la función *dispersión* (LLAVE, MAXDIR) descrita en el texto para contestar las siguientes preguntas.

- ¿Cuál es el valor de *dispersión* ("Browns", 10)?
- Encuentre dos palabras diferentes de más de cuatro letras que sean sinónimos.
- En el texto se supone que la función *dispersión()* no necesita generar un entero mayor de 20 000. Esto presentaría un problema si se tuviera un archivo con direcciones mayores de 20 000. Proponga algunas formas de evitar este problema.

2. En el estudio de la dispersión es importante comprender las relaciones entre el tamaño de la memoria disponible, el número de llaves que se van a dispersar, el intervalo de llaves posibles y la naturaleza de las llaves. Se va a dar nombre a estas cantidades, como sigue.

M = número de espacios disponibles de memoria (cada uno capaz de almacenar un registro);

r = número de registros que van a almacenarse en los espacios de memoria;

n = número de direcciones base únicas producidas por la dispersión de las r llaves de los registros, y

K = una llave, que puede ser cualquier combinación de exactamente cinco caracteres en mayúsculas.

Suponga que $h(K)$ es una fusión de dispersión que genera direcciones entre 0 y $M - 1$.

- a) ¿Cuántas llaves únicas son posibles? (Sugerencia: Si K fuera una letra mayúscula, en vez de cinco habría 26 llaves únicas posibles.)
- b) ¿Cómo están relacionadas n y r ?
- c) ¿Cómo están relacionadas r y M ?
- d) Si h fuera una función de dispersión perfecta mínima, ¿cómo estarían relacionadas n , r y M ?

3. La siguiente tabla muestra distribuciones de llaves resultantes de tres diferentes funciones de dispersión en un archivo con 6000 registros y 6000 direcciones.

	Función A	Función B	Función C
$d(0)$	0.71	0.25	0.04
$d(1)$	0.05	0.50	0.36
$d(2)$	0.05	0.25	0.15
$d(3)$	0.05	0.00	0.05
$d(4)$	0.05	0.00	0.02
$d(5)$	0.04	0.00	0.01
$d(6)$	0.05	0.00	0.01
$d(7)$	0.00	0.00	0.00

- a) ¿Cuál de las tres funciones (si existe) genera una distribución de registros aproximadamente aleatoria?
- b) ¿Cuál genera la distribución más cercana a la uniforme?
- c) ¿Cuál (si existe) genera una distribución peor que aleatoria?
- d) ¿Cuál función debe elegirse?

4. Según un resultado matemático sorprendente llamado *la paradoja del cumpleaños*, si hay más de 23 personas en una habitación, entonces existe más del 50% de posibilidades de que dos de ellos tengan la misma fecha de cumpleaños. ¿En qué forma la paradoja del cumpleaños ilustra un importante problema asociado con la dispersión?

5. Suponga que hay 10 000 direcciones para almacenar 8000 registros en un archivo con dispersión aleatoria, y que cada dirección puede almacenar un registro. Calcule los siguientes valores.

- a) La densidad de empaquetamiento del archivo.
- b) El número esperado de direcciones a los que la función de dispersión no asignó registros.
- c) El número esperado de direcciones con un registro asignado (sin sinónimos).
- d) El número esperado de direcciones con un registro más uno o varios sinónimos.

- e) El número esperado de registros que causan saturación.
- f) El porcentaje esperado de registros que causan saturación.

6. Considere el archivo descrito en el ejercicio anterior. Diga cuál es el número esperado de registros que causan saturación si las 10 000 localidades se reorganizan como

- a) 5000 compartimientos de dos registros, y
- b) 1000 compartimientos de diez registros.

7. Haga una tabla que muestre valores de la función de Poisson para $r/N = 0.1, 0.5, 0.8, 1, 2, 5$ y 10. Examine la tabla y analice las características y patrones que proporcionen información de utilidad con respecto a la dispersión.

8. Existe una técnica de manejo de la saturación llamada *saturación progresiva con conteo de llaves* (Bradley, 1982) que funciona con discos que hacen referencia por bloques, como sigue. En lugar de generar un número relativo de registro a partir de una llave, la función de dispersión genera una dirección que consiste en tres valores: un cilindro, una pista y un número de bloque. Los tres números correspondientes constituyen la dirección base de los registros.

Puesto que las unidades de disco organizadas por bloques (véase el Cap. 3) a menudo pueden buscar en una pista para encontrar un registro con una determinada llave, no hay necesidad de cargar un bloque a la memoria para saber si contiene o no un registro en particular. El procesador de E/S puede ordenar a la unidad de disco que busque el registro deseado en una pista. Incluso puede instruir al disco para buscar un espacio vacío si el registro no se encuentra en su dirección base realizando efectivamente una saturación progresiva.

- a) ¿En qué radica la superioridad de esta técnica sobre las de saturación progresiva que pueden aplicarse en unidades de disco organizadas por sectores?
- b) La principal desventaja de esta técnica es que sólo puede usarse con un tamaño de compartimiento de 1. ¿Por qué es así, y por qué se trata de una desventaja?

9. En el análisis de los aspectos de realización, se sugirió iniciar el archivo de datos creando registros reales marcados como vacíos antes de cargar el archivo con datos reales. Existen buenas razones para ello. Sin embargo, podría haber alguna razón para no hacerlo así. Por ejemplo, suponga que se quiere un archivo de dispersión con densidad de empaquetamiento muy baja y no puede permitirse tener asignado el espacio sin uso. Explique cómo puede diseñarse un sistema de manejo

de archivos para trabajar con un archivo lógico muy grande, pero asignando espacio sólo para los bloques que en realidad contengan datos.

10. Este ejercicio (inspirado en un ejemplo de Wiederhold, 1983, p. 136) se relaciona con el problema del deterioro. Se hacen varias inserciones y eliminaciones en un archivo. Se usan marcas de inutilización donde sea necesario para preservar las trayectorias de búsqueda para los registros que causan saturación.

- a) Muestre cómo se ve el archivo después de las siguientes operaciones y calcule la longitud media de búsqueda.

Operación	Dirección nativa
Agregar Alan	0
Agregar Bates	2
Agregar Cole	4
Agregar Dean	0
Agregar Evans	1
Eliminar Bates	
Eliminar Cole	
Agregar Finch	0
Agregar Gates	2
Eliminar Alan	
Agregar Hart	3

¿Cómo es que el uso de las marcas de inutilización ha causado el deterioro del archivo?

¿Qué efecto tendría recargar los registros restantes en el archivo en el orden Dean, Evans, Finch, Gates, Hart?

b) ¿Qué efecto tendría recargar los registros restantes usando la carga en dos pasos?

11. Suponga que se tiene un archivo donde el 20 por ciento de los registros causan el 80 por ciento de los accesos, y que se desea almacenar el archivo con una densidad de empaquetamiento de 0.8 y un tamaño de compartimiento de 5. Cuando se carga el archivo, se carga primero el 20 por ciento activo de los registros. Después de cargar el 20 por ciento activo de los registros y antes de cargar los demás, ¿cuál es la densidad de empaquetamiento del archivo parcialmente lleno? Con esta densidad de empaquetamiento, calcule el porcentaje del 20 por ciento activo que serían registros en saturación. Comente los resultados.

12. En los cálculos de las longitudes medias de búsqueda se consideraron sólo los tiempos de las búsquedas con éxito. Si el archivo de dispersión fuese usado en tal forma que con frecuencia se hicieran búsquedas de datos que no estuvieran en el archivo, sería útil tener estadísticas de la longitud media para una búsqueda sin éxito. Si no se tiene éxito en lugar de un gran porcentaje de búsquedas en un archivo de dispersión, explique cómo se esperaría que esto afecte el desempeño total, si la saturación se maneja mediante

- a) saturación progresiva, o
- b) encadenamiento con un área de saturación separada.

(Véase Knuth, 1973b, págs. 535-539 para estudiar estas diferencias.)

13. Aunque por lo general los archivos de dispersión no están diseñados para manejar el acceso a registros en cualquier orden de clasificación, puede haber ocasiones en que se necesite efectuar grupos de transacciones sobre un archivo de datos con dispersión. Si el archivo de datos está clasificado (en vez de disperso), las transacciones por lo regular serían realizadas con algún tipo de procesamiento secuencial coordinado, lo cual significa que el archivo de transacciones también tendría que clasificarse. Si el archivo de datos está disperso, el archivo de transacciones puede también estar preclasificado, pero sobre la base de las direcciones base de sus registros, y no con algún criterio más "natural".

Suponga que se tiene un archivo a cuyos registros se accede por lo común en forma directa, pero que periódicamente se actualizan a partir de un archivo de transacciones. Enumere los factores que tendrían que considerarse para decidir si conviene usar una organización indizada secuencial o de dispersión. (Véase Hanson, 1982, págs. 280-285, para un estudio de estos aspectos).

14. Se supone a lo largo de este capítulo que un programa para dispersión debe poder indicar correctamente si una llave determinada está localizada en una cierta dirección. Si no fuese así, habría ocasiones en que se supondría que un registro existe sin que sea cierto, resultado aparentemente desastroso. Pero considérese lo que hizo Doug McIlroy en 1978, cuando diseñó un programa de revisión de ortografía. Descubrió que si dejaba que su programa pasara como válida una de cada 4000 palabras mal escritas (y usando unos cuantos trucos más), podría introducir un diccionario ortográfico de 75 000 palabras en 64K de RAM, mejorando así extraordinariamente el desempeño.

McIlroy aceptó tolerar una palabra mal escrita de cada 4000 porque observó que los borradores de documentos rara vez contenían más de 20 errores, de modo que se podría esperar que, por cada 200 ejecuciones, el programa fallara a lo sumo una vez en detectar una palabra mal

escrita. ¿Puede el lector imaginar algunos otros casos en que sea razonable informar que existe una llave cuando no es así?

Jon Bentley [1985] proporciona un excelente informe del programa de McIlroy, con algunos comentarios sobre el proceso de resolución de problemas de esta naturaleza. D. J. Dodd [1982] analiza este método general de dispersión, llamado *dispersión con revisión*. Lea los artículos de Bentley y de Dodd, y presente un informe al respecto. Quizá le sirvan de inspiración para escribir un programa de revisión de ortografía.

EJERCICIOS DE PROGRAMACION

15. Haga y pruebe una versión de la función *dispersión()*.
16. Cree un archivo de dispersión con un registro para cada ciudad de su estado. La llave de cada registro es el nombre de la ciudad correspondiente. (Para los fines de este ejercicio, no hay necesidad de otros campos además de la llave.) Comience creando una lista clasificada de los nombres de todas las ciudades y pueblos del estado. (Si hay poco tiempo o espacio, sólo haga una lista de los nombres que comiencen con la letra 'S'.)
 - a) Examine la lista clasificada. ¿Qué patrones podrían afectar la elección de una función de dispersión?
 - b) Realice la función *dispersión()* de tal forma que pueda alterarse el número de caracteres que se desglosan. Suponga una densidad de empaquetamiento de 1 y disperse el archivo completo varias veces, cada vez desglosando un número diferente de caracteres, y produciendo las siguientes estadísticas para cada ejecución.
 - El número de colisiones, y
 - El número de direcciones a la que se asignaron 0, 1, 2,..., 10, y más de diez registros.
- c) Analice los resultados del experimento en términos de los efectos de desglosar diferentes números de caracteres, y cómo se comparan con los resultados que pueden esperarse de una distribución aleatoria.
17. Use algún conjunto de llaves, como los nombres de los pueblos del estado, para hacer lo siguiente.

- a) Escriba y pruebe un programa para cargar las llaves en tres diferentes archivos de dispersión, usando tamaños de comportamiento de 1, 2 y 5, respectivamente, y una densidad de empaquetamiento de 0.8. Use saturación progresiva para manejar las colisiones.
- b) Haga que su programa lleve estadísticas sobre la longitud media de búsqueda, la longitud máxima de búsqueda, y porcentaje de registros que están en saturación.
- c) Suponiendo una distribución de Poisson, compare los resultados con los valores esperados para la longitud media de búsqueda y el porcentaje de registros que están en saturación.

18. Repita el ejercicio 17, pero use la dispersión doble para manejar la saturación.

19. Repita el ejercicio 17, pero maneje la saturación usando saturación encadenada en un área de saturación separada. Suponga que la densidad de empaquetamiento es la proporción entre el número de llaves y las direcciones *base* disponibles.

20. Escriba un programa que pueda efectuar inserciones y eliminaciones en el archivo creado en el problema anterior, usando un tamaño de compartimiento de 5. Haga que el programa mantenga estadísticas sobre la longitud media de búsqueda. (También puede implantarse un mecanismo que indique el momento en el cual la longitud de búsqueda se ha deteriorado hasta tal punto que el archivo debe reorganizarse.) Analice en detalle los problemas que deben afrontarse para decidir cómo manejar las inserciones y eliminaciones.

LECTURAS ADICIONALES

Existen varios buenos estudios sobre la dispersión y los aspectos relacionados con ella en forma general, entre ellos Knuth [1973b], Severance [1974], Maurer [1975], y Sorenson, Tremblay y Dutscher [1978]. Los libros de texto que tratan el diseño de archivos por lo general contienen abundante material sobre dispersión, y con frecuencia proporcionan amplias referencias para un estudio complementario. Las siguientes pueden ser útiles:

- Hanson [1982] está lleno de resultados analíticos y experimentales que exploran todos los aspectos que se han abordado y muchos otros. También contienen un buen capítulo sobre comparación de diferentes organizaciones de archivos.

- Bradley [1982] aborda la dispersión en forma general, pero también ofrece mucha información sobre programación para archivos de dispersión usando IBM PL/I.
- Loomis [1983], también analiza la dispersión en forma general, pero da especial importancia a la programación para archivos de dispersión en COBOL.
- Teorey y Fry [1982] y Wiederhold [1983] serán textos de utilidad para interesados en analizar las ventajas y desventajas de los métodos de dispersión básicos.

Una de las aplicaciones de la dispersión que recientemente han despertado gran interés es el desarrollo de los *revisores de ortografía*, cuyas características especiales hacen que requieran tipos de dispersión bastante distintos de los que se describen en este texto. Los artículos de Bentley [1985] y Dodds [1982] son una introducción a la literatura sobre este tema. (Véase también el ejercicio 14.)

Los esquemas de *dispersión extensible* han generado a últimas fechas mucho interés respecto a aplicaciones de bases de datos, porque permiten el uso de archivos que pueden crecer o disminuir sin necesidad de una reorganización total. Scholl [1981] proporciona una introducción a la literatura sobre estos métodos.

APENDICES

A

TABLA DE CODIGOS DE CARACTERES ASCII

	Dec.	Oct.	Hex.		Dec.	Oct.	Hex.		Dec.	Oct.	Hex.		Dec.	Oct.	Hex.
nul	0	0	0	sp	32	40	20	@	64	100	40	'	96	140	60
sol	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
stx	2	2	2	"	34	42	22	B	66	102	42	b	98	142	62
etx	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
eot	4	4	4	\$	36	44	24	D	68	104	44	d	100	144	64
enq	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
ack	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
bel	7	7	7	'	39	47	27	G	71	107	47	g	103	147	67
bs	8	10	8	(40	50	28	H	72	110	48	h	104	150	68
ht	9	11	9)	41	51	29	I	73	111	49	i	105	151	69
nl	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
vt	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
np	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
cr	13	15	D	-	45	55	2D	M	77	115	4D	m	109	155	6D
so	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
si	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
dle	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
dcl	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
dc2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
dc3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
dc4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
nak	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
syn	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
etb	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
can	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
em	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
sub	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
esc	27	33	1B	;	59	73	3B	[91	133	5B	{	123	173	7B
fs	28	34	1C	<	60	74	3C	\	92	134	5C	:	124	174	7C
gs	29	35	1D	=	61	75	3D]	93	135	5D	}	125	175	7D
rs	30	36	1E	>	62	76	3E	^	94	136	5E	-	126	176	7E
us	31	37	1F	?	63	77	3F	-	95	137	5F	del	127	177	7F

B**FUNCIONES DE CADENAS EN
PASCAL: *herramientas.prc*****FUNCIONES Y PROCEDIMIENTOS
EMPLEADOS PARA OPERAR SOBRE *cadena***

Las siguientes funciones y procedimientos conforman las herramientas para trabajar con variables que son declaradas como:

TYPE

```
cadena = packed array [0.. LONG_MAX_REG] of char;
```

La longitud de la *cadena* se almacena en el byte cero del arreglo como un carácter representativo de la longitud. Nótese que las funciones de Pascal CHR() y ORD() se usan para convertir enteros en caracteres y viceversa.

Funciones incluidas:

<i>long_cad (cad)</i>	
<i>limpia_cad (cad)</i>	
<i>copia_cad (cad1, cad2)</i>	
<i>conc_cad (cad1, cad2)</i>	
<i>lee_cad (cad)</i>	

Devuelve la longitud de <i>cad</i> .
Limpia <i>cad</i> colocando su longitud en cero.
Copia el contenido de <i>cad2</i> en <i>cad1</i> .
Concatena <i>cad2</i> al final de <i>cad1</i> .
Coloca el resultado en <i>cad1</i> .
Lee <i>cad</i> como entrada del teclado.

<i>escribe_cad (cad)</i>	Transcribe el contenido de <i>cad</i> a la pantalla.
<i>lee_arch_cad_(fd, cad, longitud)</i>	Lee una <i>cad</i> de longitud <i>longitud</i> del archivo <i>fd</i> .
<i>escribe_arch_cad(fd,cad)</i>	Escribe el contenido de <i>cad</i> en el archivo <i>fd</i> .
<i>cad_espac(cad)</i>	Elimina los espacios en blanco del final de <i>cad</i> . Devuelve la longitud de <i>cad</i> .
<i>mayúsculas (cad1, cad2)</i>	Convierte <i>cad1</i> en mayúsculas, y almacena el resultado en <i>cad2</i> .
<i>hazllave (apellido, nombre, llave)</i>	Combina apellido y nombre en una llave en forma canónica, almacena el resultado en <i>llave</i> .
<i>mín (int1, int2)</i>	Devuelve el mínimo de dos enteros.
<i>comp_cap (cad1, cad2)</i>	Compara <i>cad1</i> con <i>cad2</i> : Si <i>cad1 = cad2</i> , <i>comp_cad</i> devuelve 0. Si <i>cad1 < cad2</i> , devuelve un número negativo. Si <i>cad1 > cad2</i> , devuelve un número positivo.

```

FUNCTION long_cad (cad: cadena) : integer;
{ long_cad ( ) devuelve la longitud de cad }
BEGIN
  long_cad: = ORD(cad [0])
END;

PROCEDURE limpia_cad (VAR cad: cadena);
{ Procedimiento que limpia cad colocando su longitud en 0 }
BEGIN
  cad [0] := CHR (0)
END;

PROCEDURE copia_cad (VAR cad1: cadena; cad2: cadena);
{ Procedimiento para copiar cad2 en cad1 }
VAR
  i : integer;
BEGIN
  for i := 1 to long_cad (cad2) DO
    cad1 [i] := cad2 [i];
  cad1 [0] := cad2 [0]
END;

PROCEDURE conc_cad (VAR cad1: cadena; cad2: cadena);
{ conc_cad ( ) concatena cad2 al final de cad1 y almacena el resultado en
  cad1 }
VAR
  i : integer;
BEGIN
  for i := 1 to long_cad (cad2) DO
    cad1 [long_cad (cad1)+i] := cad2 [i];

```

```
cad1[0] := CHR (long_cad (cad1) + long_cad (cad2))
END;

PROCEDURE lee_cad (VAR cad: cadena);
{ Procedimiento que lee cad como entrada desde el teclado }
VAR
    longitud : integer;
BEGIN
    longitud :=0;
    while (not EOLN) and (longitud < = TAM_MAX_REG) DO
    BEGIN
        longitud := longitud +1;
        read (cad [longitud])
    END;
    readln;
    cad [0] := CHR (longitud)
END;

PROCEDURE escribe_cad (VAR cad: cadena);
{ escribe_cad () escribe cad en la pantalla }
VAR
    i : integer;
BEGIN
    for i := 1 to long_cad (cad) DO
        write (cad [i]);
    writeln;
END;

PROCEDURE lee_arch_cad (VAR fd: text; VAR cad: cadena; longitud: integer);
{ lee_arch_cad lee una cad con longitud longitud de fd }
VAR
    i : integer;
BEGIN
    for i := 1 to longitud DO
        read (fd, cad[i]);
    cad [0] := CHR (longitud)
END;

PROCEDURE escribe_arch_cad (VAR fd: text; cad : cadena);
{ escribe_arch_cad () escribe cad al archivo fd }
VAR
    i : integer;
BEGIN
    for i := 1 to long_cad (cad) DO
        write (fd; cad [i]);
END;
```

```

FUNCTION cad_espac (VAR cad: cadena): integer;
{ cad_espac( ) elimina los espacios del final de cad y devuelve su nueva
longitud }
VAR
  longitud : integer;
BEGIN
  longitud := long_cad (cad);
  while cad [longitud] = ' ' DO
    longitud := longitud -1;
  cad [0] := CHR (longitud);
  cad _ espac := longitud
END

```

```

PROCEDURE mayúsculas (cad1: cadena; VAR cad2: cadena);
{ mayúsculas( ) convierte cad1 en letras mayúsculas y almacena la cadena
resultante en cad2 }
VAR
  i : integer;
BEGIN
  for i := 1 to long_cad (cad1) DO
  BEGIN
    if (ORD(cad1 [i]) >= ORD('a')) AND (ORD(cad1 [i]) <= ORD('z')) then
      cad2 (i) := CHR (ORD (cad1 [i])- 32)
    else
      cad2 [i] := cad1 [i];
  END;
  cad2 [0] := cad1[0]
END;

```

```

PROCEDURE hazllave (apellido: cadena; nombre: cadena; VAR llave: cadena);
{ hazllave () elimina los espacios del final de las cadenas de apellido y
nombre, concatena apellido y nombre separándolos con un espacio y convierte
las letras en mayúsculas }
VAR
  longl  : integer;
  longf  : integer;
  cad_espacios: cadena;
BEGIN
  longl := cad_espac (apellido);
  copia_cad (llave, apellido);
  cad_espacios [0] := CHR (1);
  cad_espacios [1] := ' ';
  conc_cad (llave, cad_espacios);
  longf := cad_espac (nombre);

```

```
conc_cad (llave, nombre);
mayúsculas (llave, llave)
END;
```

```
FUNCTION min (int1, int2: integer): integer;
{ min ( ) devuelve el mínimo de dos enteros }
BEGIN
  if int1 <= int2 then
    min := int1
  else
    min := int2
END;
```

```
FUNCTION comp_cad (cad1: cadena; cad2: cadena): integer;
{ Función que compara cad1 con cad2. Si cad1 = cad2, entonces
  comp_cad devuelve 0. Si cad1 < cad2, entonces devuelve un número negativo.
  Si cad1 > cad2, entonces devuelve un número positivo. }

VAR
  i : integer;
  longitud : integer;
BEGIN
  if long_cad (cad1) = long_cad (cad2) then
    BEGIN
      i :=1;
      while cad1 [i] = cad2 [i] DO
        i := i + 1;
      if (i - 1) = long_cad (cad1) then
        comp_cad :=0
      else
        comp_cad := (ORD (cad1 [i])) - (ORD (cad2 [i]))
    END
  else BEGIN
    longitud :=min (long_cad (cad1), long_cad (cad2));
    i := 1;
    while (cad1 [i] = cad2 [i]) and (i (<= longitud) DO
      i := i + 1;
    if i > longitud then
      comp_cad := long_cad (cad1) - long_cad (cad2)
    else
      comp_cad := (ORD (cad1 [i])) - (ORD (cad2 [i]))
  END
END;
```

C

INTRODUCCION A C

INTRODUCCION

C.1

HISTORIA Y CARACTERISTICAS DISTINTIVAS

C es un lenguaje de programación de propósito general, especialmente adecuado para la programación de sistemas. Fue diseñado a principios de la década de los setenta por Dennis Ritchie, en los laboratorios Bell. Su aplicación más famosa es su uso en la escritura del sistema operativo UNIX.

C es un lenguaje de dimensiones relativamente reducidas, por lo que es adecuado para implantarse en computadores pequeños. Hay compiladores de C disponibles para la mayoría de los computadores grandes, así como para muchos de los pequeños, y se presentan con una biblioteca estándar de funciones que le proporcionan las capacidades que tienen otros lenguajes mucho más amplios, manteniendo no obstante fuera del lenguaje mismo las características que son dependientes de la máquina. La mayoría de las implantaciones de C se adhieren estrechamente a la definición del lenguaje que proporciona el texto. *El lenguaje de programación C* de Brian Kernighan y Dennis Ritchie [1978], de manera que el *C de Kernighan y Ritchie* suele ser el estándar. Uno de los efectos del alto grado de estandarización de C es que los programas en C son más portátiles que los programas de la mayoría de los lenguajes con capacidades similares.

Si el lector está familiarizado con un lenguaje de programación moderno, como Pascal, PL/I o Fortran 77, podrá reconocer la mayoría de los tipos de datos, operadores y estructuras de control de C, y será capaz de leer la mayoría de los programas sencillos en C sin gran dificultad. Sin embargo, existen algunas diferencias importantes entre los tipos de datos y operadores de C y los de otros lenguajes. Por ejemplo, C proporciona tipos de datos y operadores muy cercanos a los proporcionados por el hardware de la máquina. Uno de los resultados es que los programas en C pueden hacerse muy eficientes; otro es que en C pueden efectuarse operaciones de bajo nivel (como la manipulación de bits y E/S de archivos) con mucho menos exigencias que en otros lenguajes.

Uno de los resultados potencialmente negativos de las operaciones de bajo nivel de C es que no protege al diseñador de programas como lo hacen otros lenguajes. El usuario debe ser más cuidadoso con lo que está haciendo, porque el compilador y el ambiente en tiempo de ejecución no apoyan al usuario en la misma forma en que otros lenguajes.

Cuando se usa C, cabe esperar que el estilo de programación usado sea diferente del que se emplea en la programación con otros lenguajes. Los programas en C tienden a apoyarse fuertemente en funciones definidas por el usuario y en las de biblioteca. Las funciones en C tienden a ser cortas. El fuerte uso de tipos de datos (como los apuntadores) y operadores (como el de *incremento*) poco comunes da a los programas en C una apariencia considerablemente diferente de la de los programas escritos en otros lenguajes. Los programas deben ser muy legibles, y deben tener además una brevedad que demanda que sean leídos y escritos con cuidado.

C.2

EL AMBIENTE C

El compilador de C es sólo una parte del ambiente de programación C. Muchas de las características que el lector usa cuando programa en C no son, estrictamente hablando, parte de C, sino que son proporcionadas en alguna otra parte del ambiente de programación de C. Por ejemplo, no existe una orden para imprimir en C, como en la mayoría de los lenguajes, pero hay una función de impresión que siempre está disponible en la *biblioteca estándar de E/S de C*.

C.2.1 BIBLIOTECAS

Las *bibliotecas estándar* de C consisten en grupos de *funciones precompiladas* que deben ligarse a todo programa que las llame antes

de que se ejecute. Las funciones de las bibliotecas son estándar en el sentido de que están especificadas como un estándar existente en Kernighan y Ritchie [1978] o en otros autores. (Si se tiene UNIX, la definición de lo que está disponible en la biblioteca estándar aparece en forma precisa en el manual de UNIX. Si se dispone de otro sistema operativo, la biblioteca estándar disponible debe estar especificada en la documentación que acompaña al compilador.)

Si se quiere disponer de funciones precompiladas que no están en las bibliotecas estándar, el lector puede hacerlas por su cuenta. Las funciones en estas bibliotecas también deben ligarse al programa compilado antes de la ejecución. Tomando en cuenta el uso de las bibliotecas, puede considerarse que, antes de que pueda ser ejecutado, el programa pasa por las etapas que se muestran en la figura C.1.

Con probabilidad el lector las reconocerá como las etapas que siguen la mayoría de los programas antes de poder ser ejecutados. Aquí se destacan en forma especial porque el uso de bibliotecas es en extremo importante en C, y porque la comprensión del siguiente tema depende de que se hayan comprendido bien los pasos de compilación, ligado y ejecución.

C.2.2 LOS ARCHIVOS *# include*

Además de las bibliotecas de funciones precompiladas, los programas fuente comúnmente hacen uso de archivos de programa que no han sido compilados. Suponga, por ejemplo, que se tiene un archivo que contiene las definiciones de las constantes y estructuras de datos, y que dichas definiciones son usadas con frecuencia por un gran número de los

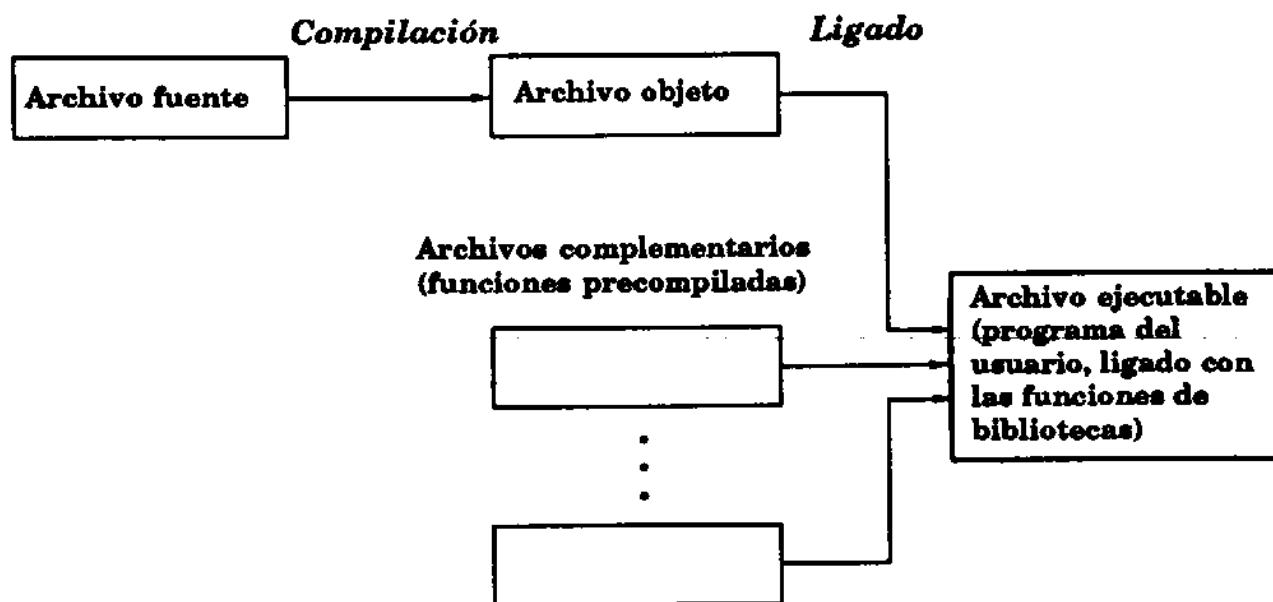


FIGURA C. 1 • Las etapas de preejecución de un programa.

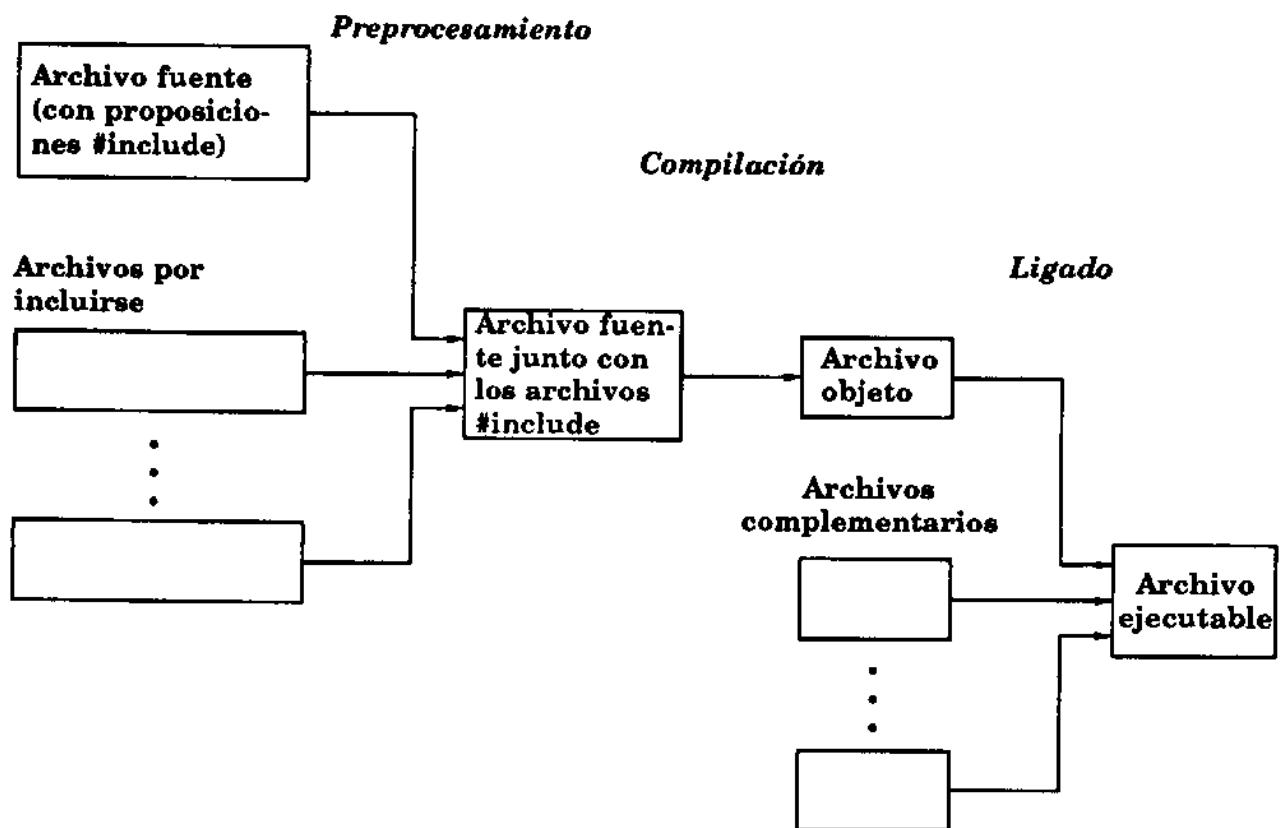


FIGURA C. 2 • Las etapas en el procesamiento de un programa C.

programas que se escriben. Sería agradable si se pudiera incluir convenientemente el archivo de definiciones con cualquier programa fuente que pudiera necesitarlo.

El uso de archivos fuente adicionales de este tipo es muy común en C. Por ejemplo, existen varias estructuras de datos y otros objetos relacionados con E/S que son útiles en virtualmente todo programa fuente. Un conjunto completo de las definiciones adecuadas de esos objetos se guarda en un archivo llamado *stdio.h* [la *h* significa archivo de encabezado (*header*, en inglés), ya que se inserta a la cabeza del archivo fuente.] Antes de compilar el código en el principal archivo fuente de un programa, debe *incluirse* el archivo adicional *stdio.h* en el archivo fuente.

La orden *#include* permite incluir archivos en el archivo fuente principal. Estos archivos se insertan dentro del archivo fuente principal antes de la compilación, de modo que desde el punto de vista del compilador, el contenido de los archivos incluidos parece ser parte del archivo fuente principal. Al programa que inserta los archivos *#include* se le llama *preprocesador* de C. Tomando en cuenta el uso de los archivos *#include*, es necesario alterar ligeramente las etapas relacionadas con el procesamiento de un programa en C, como se muestra en la figura C.2.

C.2.3. CONJUNTOS DE HERRAMIENTAS

Se subrayan las distintas formas en que pueden presentarse los archivos juntos para componer un programa ejecutable, porque es mucho más común hacer este tipo de operaciones en C que en otros lenguajes. Es raro el programa fuente en C que no *incluye* (`#include`) por lo menos uno o dos archivos de encabezado. Asimismo, es raro un programa en C que no haga uso de las funciones de biblioteca. La idea de mantener archivos de propósito especial que contengan definiciones y funciones útiles en diversos contextos es fundamental para la mayor parte de la programación en C. A tales conjuntos de archivos se les denomina *conjuntos de herramientas*, porque proporcionan herramientas que pueden usarse para diferentes trabajos.

C.3

CREACION, COMPILEACION Y EJECUCION DE UN PROGRAMA EN C

He aquí un programa sencillo que demuestra que está en funcionamiento enviando un mensaje a la pantalla.

```
main()
{
    printf ("¡Funciono!\n");
}
```

Si se quiere ejecutar este programa, lo primero que debe hacerse es crear un archivo que lo contenga. Este archivo puede tener cualquier nombre válido, pero algunos sistemas requieren que termine con la extensión `.c`, como en *trabajo.c*.

El siguiente paso es compilar el programa.[†] El formato preciso para llamar al compilador C varía de un compilador a otro, de manera que debe consultarse en la documentación. Puede no ser precisamente como se muestra enseguida, pero sin duda es similar.

```
cc trabajo.c
```

La llamada cc normalmente hace que se efectúen las siguientes tareas.

[†] Si se tiene acceso a un intérprete C, los pasos son un poco diferentes. Consultese el manual del sistema para mayores detalles.

Compilación:

- Efectúa las funciones de procesamiento, tales como la inclusión de archivos `#include`.
- Revisa el programa para buscar errores de sintaxis (llaves no apareadas, palabras mal escritas, y demás) e imprime mensajes de error cuando los encuentra.
- Traduce partes individuales del programa a código ensamblador.
- Traduce el código ensamblador a código objeto: instrucciones de máquina que son equivalentes al código C. Estas instrucciones se colocan en el *archivo de código objeto*.

Ligado:

- Liga* el programa con algunos otros que necesite de una *biblioteca* para efectuar su trabajo. (En el programa del ejemplo, la biblioteca tiene un programa llamado `printf()` que se usa en el programa.)
- Coloca el programa ejecutable resultante en un archivo ejecutable.

En muchos ambientes los dos pasos no son automáticos. Puede ser necesaria una solicitud específica para que el programa se ligue. Revisese la documentación del sistema.

Ahora se está listo para ejecutar el programa. Esto se hace por lo regular digitando el nombre del archivo ejecutable. El nombre asignado al archivo ejecutable varía de sistema a sistema. En los sistemas UNIX puede darse un nombre especial (véase la siguiente lista) o puede digitarse el nombre por omisión

a.out.

después de lo cual (en el programa de ejemplo) aparece

;Funciono!

en la pantalla.

Existen muchas opciones que pueden llamarse con la orden `cc`. Por desgracia varían de un sistema operativo a otro. Dos opciones importantes en UNIX y sistemas parecidos son `-o` y `-c`. He aquí algunos de sus usos.

cc -o trabajo trabajo.c Hace que el código ejecutable se almacene en un archivo llamado *trabajo*, en lugar de *a.out*.

cc -c trabajo.c Indica al compilador que compile, pero no ligue, el código de *trabajo.c*. Al archivo objeto resultante se le nombra automáticamente

trabajo.o, y puede ligarse posteriormente (véase lo siguiente).

cc -o progtot trabajo.c archa.c archb.c

Supone que se tienen tres archivos llamado *trabajo.c archa.c*, y *archb.c* y que deben copiarse y ligarse para producir un programa que funcione. El archivo *trabajo.c* puede contener el programa principal, mientras que los otros dos contienen funciones de apoyo. (Una vez compilados, podrían ser los archivos definidos por el usuario descritos anteriormente.)

Como resultado de la ejecución de la orden se crean tres archivos objeto llamados *trabajo.o*, *archa.o*, y *archb.o*, más un archivo ejecutable (*progtot*) creado al ligar los tres archivos objeto y las bibliotecas estándar necesarias.

cc -o progtot trabajo.c archa.o archb.o

Aprovecha el hecho de que dos de los tres archivos de entrada no necesiten recompilarse. Compila *trabajo.c* (produciendo *trabajo.o*), pero no recompila los otros dos archivos, sino que, partiendo del hecho de que tienen extensión *.o*, supone que son archivos objeto apropiados, y simplemente los liga a *trabajo.o* para producir el archivo ejecutable *progtot*.

Esta opción puede ser muy fácil de manejar cuando es preciso corregir un defecto en un archivo fuente y no es necesario cambiar los otros. Esto ahorra tiempo de compilación, que puede ser considerable cuando los diferentes archivos se vuelven grandes o son numerosos.

Si el lector no ha introducido, compilado y ejecutado el programa *trabajo.c*, debe hacerlo ahora.

C.4

ERRORES

Sin duda el lector está consciente de que el proceso de compilación, ligado y ejecución no siempre marcha sobre ruedas. El programa puede tener errores que hagan que la compilación o la ejecución no procedan. Debe advertirse que es bastante fácil cometer errores en C, en especial en las primeras etapas de aprendizaje del lenguaje. La mejor recomendación al respecto es programar tan cuidadosa y defensivamente como sea posible mientras se especializa en C.

Por ejemplo, supóngase que se introduce un error en el programa *trabajo.c*, al quitar el punto y coma al final de la proposición *printf*. Al intentar compilar el programa erróneo es probable que se reciba un mensaje de error que no proporcione suficiente información, por ejemplo: "Incluso reporta la línea errónea porque no detecta el error sino hasta que llega a la llave cerrada y observa que tiene una proposición que no termina en punto y coma."

A lo largo del texto, se intenta ayudar al lector a enfrentar las frustraciones por los errores, señalando las áreas donde se necesita ser particularmente cuidadoso, y analizando algunas de las situaciones de error más desconcertantes que pueden ocurrir. Sin embargo, debe estar advertido que se espera que un programador de C sea más cuidadoso de lo que pudiera ser con otros lenguajes. C proporciona al usuario la capacidad de hacer cosas que otros lenguajes resguardan. Es un recurso que ofrece grandes ventajas, pero debe manejarse con cuidado.

CARACTERISTICAS COMUNES DE LOS LENGUAJES

La mayoría de los lenguajes algorítmicos tienen los mismos bloques básicos de construcción: una estructura de programa claramente definida, símbolos especiales y palabras reservadas, tipos de datos, operadores, proposiciones para control del flujo de ejecución del programa, subrutinas, y reglas que definen el alcance de las variables y subrutinas. Puesto que se supone que el lector ya conoce un lenguaje de programación, no se desperdiciará tiempo explicando el porqué de la mayoría de estas características o cómo deben usarse. Se prefiere enfocar ante todo la manera en que se manifiestan sintácticamente, y cómo pueden ser diferentes en C.

C.5

ESTRUCTURA Y SIMBOLOS ESPECIALES

En C, todo programa consiste en un conjunto de funciones, una de las cuales tiene que llamarse *main*. Considérese el siguiente programa simple en C

```
main ()  
{
```

```

    printf("O o no O... \n");
}

```

Cuando se ejecuta un programa en C, la función llamada *main()* se ejecuta primero. Por lo tanto, todas las demás funciones que componen el programa deben ser llamadas, directa o indirectamente, por *main()*. En el ejemplo, *main()* llama a la función *printf()*, la cual se encuentra en una biblioteca cuyas funciones están disponibles para todos los programas C.

Después se examinan en detalle otras características de las funciones (p. ej., cómo se pasan los parámetros), pero por ahora el estudio se queda en las funciones *main()*, las cuales tienen la forma general:

```

main()
{
    <proposiciones>
}

```

donde

<i>main()</i>	Declara la función llamada <i>main()</i> . En la definición de una función, el nombre debe ir seguido por los paréntesis. Después se verá que los paréntesis pueden incluir parámetros.
{...}	Son un par de llaves, las cuales deben encerrar el cuerpo de la función. Cuando un compilador de C detecta una llave que cierra, para complementar la llave que abre, supone que la función está totalmente definida. Si nunca encuentra la llave que cierra, indica un error.
< <i>proposiciones</i> >	Son proposiciones de C que componen el cuerpo de la función. Estas proposiciones pueden ser tan simples o complejas como sea necesario. Toda proposición simple de C debe terminar con un punto y coma.

C.6

TIPOS DE DATOS

La mayoría de los lenguajes proporcionan dos formas de referirse a los datos: constantes y variables. Algunos, entre ellos C, permiten usar un tercer tipo de referencia: apuntadores. En secciones posteriores se aborda el uso de los apuntadores. Las constantes y variables se representan de manera muy semejante en C y en la mayoría de los demás lenguajes.

En C, toda variable debe ser declarada para que tenga un cierto tipo de datos. Usar una variable no declarada provoca un error de compilación. Sin embargo, la declaración de una variable en C no tiene las mismas implicaciones que tiene en muchos otros lenguajes, porque C no es un lenguaje con tipos estrictos. Ahora es necesario analizar las implicaciones de esto.

C.6.1 C NO ES UN LENGUAJE CON TIPOS ESTRICHTOS

Algunos lenguajes, como Pascal, tienen *tipos estrictos*. Esto significa que para un tipo de datos determinado sólo pueden efectuarse ciertas operaciones. Por ejemplo, en Pascal no puede tratarse un carácter como entero. Como C no es un lenguaje con tipos estrictos, puede efectuarse casi todo tipo de operaciones que se quiera con cualquier tipo de datos.

Esto tiene implicaciones importantes. Para una gran parte del trabajo que deben hacer los programas, es esencial que puedan tratar ciertas unidades de datos en diferentes formas. Los programas serían más grandes, con menor probabilidad de ser correctos y con menos facilidad de mantenerlos, si el usuario se viera obligado a usar procedimientos complejos para sortear las restricciones de los tipos de datos estrictos.

Por otro lado, existen ciertos peligros inherentes al hecho de no manejar tipos de datos estrictos. Cuando no se tienen, es fácil tratar los datos inadvertidamente en forma equivocada. A medida que se programa en C, debe aprenderse a ser muy cuidadoso al mezclar tipos de datos para evitar un tratamiento de datos innecesariamente intrincado y para documentar el necesariamente intrincado tratamiento de los datos.

C.6.2 ENTEROS Y CARACTERES

En C existen cuatro y medio tipos de enteros: *int*, *short*, *long*, *unsigned* y *char*. La razón de que sean tantos es que se desea impartir flexibilidad en la programación.

int

Corresponde al tipo de datos enteros que se observa en la mayoría de los lenguajes. Se refiere a un número completo con signo (sin partes fraccionarias), almacenado en una palabra del computador. El tamaño de un *int* depende del tamaño de palabra más común usado por un determinado computador. Un *int* puede tener un tamaño de 16 a 36 bits. Una constante *int* se representa con uno o más dígitos, posiblemente precedidos por un signo, como en 1285 y —137.

short y long

Proporcionan diferentes tamaños de enteros, y sus tamaños son menos variables de máquina a máquina. Un *short* es usualmente de dos bytes (16 bits), y un *long* es por lo regular de cuatro bytes (32 bytes). Como sus tamaños varían menos de una máquina a otra, y como con frecuencia es importante ser preciso acerca de la cantidad de almacenamiento que un cierto dato usa, a menudo se usan *short* y *long* en lugar del menos preciso *int*. Una constante *long* se representa en la misma forma que un *int*, excepto que debe ir seguida de una *L* o *l* como *608L* o *608l*. Se refiere a enteros que no pueden ser negativos.

unsigned

Obedecen las leyes de la aritmética 2^n , donde *n* es el número de bits en un *int*. También existen *unsigned short*, *longs* y *chars* en muchas implantaciones de C. Se refiere a enteros de un byte capaces de almacenar un carácter en el conjunto de caracteres de la máquina. Puesto que es un entero, pueden realizarse operaciones aritméticas con un carácter. Como C no tiene tipos de datos estrictos, puede tratarse también como un carácter. Una constante *char* se representa con un solo carácter encerrado por apóstrofos, como '*a*'. El valor entero asociado con una constante *char* es el valor numérico del carácter en el conjunto de caracteres de la máquina. Por ejemplo, el código ASCII de '*a*' (minúscula) es el valor decimal 97.

char

En C, es importante poder representar ciertos códigos de caracteres que no tienen representación gráfica. Por ejemplo, es necesario representar los códigos de *línea nueva* y *tabulador*. Para ello ciertas letras clave se preceden de una diagonal invertida, como en

'\n'	Línea nueva
'\t'	Tabulador (correspondiente a la tecla de tabulación en la máquina de escribir)
'\0'	Nulo (el <i>valor</i> 0, no el carácter 0)
'\b'	Retroceso
'\r'	Retroceso de carro
'\f'	Alimentación de forma (página nueva)
'\\'	Diagonal invertida
'\''	Apóstrofo
'\ddd'	Una constante octal

El siguiente programa ilustra algunos usos de los enteros, así como de otras conocidas características de programación. Se puede aprender bastante acerca de la sintaxis de C estudiando el programa cuidadosamente. Este programa no efectúa ninguna función útil.

```

/* ejemplos.c...
 * Programa que ilustra los usos de enteros en C. */
main()
{
    /* --- Declaraciones (las siguientes siete proposiciones) --- */
    int a;
    int b = 3;           /* asignación durante una declaración. */
    int x;
    int nombre_curioso; /* los nombres de las variables pueden ser
                           largos, pero con frecuencia para los primeros
                           nombres internos sólo cuentan los primeros
                           ocho caracteres. Existen algunas variaciones
                           de máquina a máquina. El único carácter
                           especial permitido es el subrayado "_". */
    short sh, Sh;        /* Las mayúsculas y las minúsculas son
                           diferentes. Se puede declarar más de una
                           variable al mismo tiempo. */
    long l;
    char c, d;          /* c se usa comúnmente para almacenar
                           un char. */

    /* --- Expresiones que usan enteros (siguientes nueve
       proposiciones) --- */
    a = b + 3;           /* Proposición de asignación */
    Nombre_curioso = 'f'; /* La ausencia de tipos estrictos permite
                           esto. Cuando un char se asigna a un int,
                           automáticamente se convierte
                           a int. */
    x = 'A' - 65;        /* Lo mismo para expresiones aritméticas. */
    sh = a;              /* int asignado a short. Esto es correcto si
                           a es lo suficientemente pequeño. En caso
                           contrario, provoca un error molesto (es
                           decir, difícil de detectar). */
    .
    Sh = 40000;          /* !!NO FUNCIONA!! 40 000 es demasiado grande
                           para un short. 32 767 es el short más
                           grande permitido. */
    l = b + 1000000L;    /* Los tipos de datos más cortos se
                           convierten a los más grandes. */
    b =
        25               /* Los espacios, líneas nuevas y
                           comentarios pueden estar en cual-
                          quier lugar en las proposiciones. */
        -2               /* ¿Puede encontrar el punto y coma? */
    ;c = 'r' + 1;         /* Sumar 1 a un carácter da el siguiente
                           carácter. */

```

```

d = '\007';           /* La constante octal 7. Si la terminal
                      tiene campana, con probabilidad d la hará
                      sonar. */
/* ---- Impresión de los resultados (las siguientes nueve líneas) --- */

printf ("a=%d/n", a); /* printf es una función de biblioteca para dar
                      formato a la salida. Esto envía lo siguiente
                      a la terminal:
                      a = 6
                      seguido de una línea nueva. Los detalles
                      vienen a continuación. */

printf ("b = %d\t%o\t%x.\t", b,b,b);
                      /* Imprime el valor de b en decimal, octal
                      y hexadecimal. Nótese el uso de tabulador
                      ('\t'). */

printf ("nombre_curioso= %c o %d.\n", /* Imprime nombre_curioso
nombre_curioso      /* como char y como
nombre_curioso);   /* int decimal. */

printf ("x = %d.\n", x); /* ¿Qué es lo que se esperaría aquí?      */
                      /* Sugerencia: el ASCII de 'A' es 65.      */
printf ("sh = %d.\n", sh);
printf ("Sh = %d.\n", Sh); /* Debería ser 40 000, pero Sh es      */
                      /* demasiado corto para almacenarlo. */

printf ("l = %ld.\n", l); /* Los enteros largos necesitan una l      */
                      /* (ele) en sus especificaciones de      */
                      /* formato. */

printf ("c = %c\n", c);
printf ("%c", d);

```

La salida que produce este programa es como sigue:

```

a=6
b = 23          27      17.
nombre_curioso = f o 102.
x = 0.
sh = 6.
Sh = -25536.          (Nótese el resultado erróneo.)
I = 1000003.
c = s
^G          ("Control-G", que por lo regular provoca un sonido.
El "G" probablemente no aparecerá en la pantalla.)

```

C.6.3 NUMEROS DE PUNTO FLOTANTE

Los programadores de C no usan demasiado los números de punto flotante, pero ocasionalmente resulta conveniente hacerlo. Una constante flotante en C tiene una parte entera, un punto decimal, una parte

fraccionaria, y una *e* o *E* seguida de un exponente entero con signo. Algunos de éstos son opcionales. Ejemplos: 2.718 y 2.3E-6.

Las variables de punto flotante vienen en dos tamaños, *float* y *double*. La representación y precisión verdaderas de las variables y constantes de punto flotante varía de máquina a máquina. Un *float* por lo general ocupa 32 bits, y proporciona alrededor de seis dígitos decimales de precisión. Un *double* por lo general ocupa 64 bits, y proporciona alrededor de 15 dígitos decimales de precisión. Ambos proporcionan exponentes de al menos 10^{-38} .

Las constantes de punto flotante siempre se almacenan en formato *double*. Las conversiones de valores de punto flotante a enteros son dependientes de la máquina. Algunas veces truncan; algunas veces redondean. Las conversiones de entero a punto flotante se comportan mejor. Por supuesto, puede perderse cierta precisión cuando un entero es demasiado grande.

C.6.4 CONSTANTES HEXADECIMALES Y OCTALES

Las representaciones hexadecimales y octales son especialmente útiles cuando quiere tenerse una imagen de cómo se ven en realidad los datos, o cuando no existe otra representación conveniente para un valor que se quiera almacenar en algún lugar. En C, una constante entera octal consiste en el dígito 0 seguido de una secuencia de dígitos octales (0-7). Una constante entera hexadecimales consiste en 0x o 0X seguido de una secuencia de dígitos hexadecimales (0-9, A o a – F o f). Ejemplos:

Octal	Hexadecimal	Decimal
05	0x5	5
037	0x1f	31

Se supone que una constante hexadecimal u octal es del tipo *int*, a menos que sea demasiado grande, o que esté seguida de la letra 'l' o 'L' (que significa *long*.)

C.7

FUNCION *print()* PARA PRESENTACION EN PANTALLA

La función *print()* es una función de C de propósito general para enviar la salida a la pantalla. Puesto que en este texto no es de interés diseñar salidas elegantes en pantalla, no es necesario entrar en detalle con diversos formatos de *print()*. La forma general de una proposición *print()* es

```
print (<formado>, <lista de argumento>);
```

El **<formato>** se encierra entre comillas y contiene los caracteres que se enviarán a la pantalla, más los códigos de los formatos. Los códigos de los formatos indican cómo dar formato a los valores referidos en la **<lista de argumento>**. Se dispone, entre otros, de los siguientes códigos de formato.

%c	Carácter
%d	Entero decimal
%f	Número de punto flotante
%o	Entero octal
%s	Cadena
%x	Entero hexadecimal
%%	El carácter %

Se puede dar formato a los enteros largos mediante la inclusión de "l", como en: **%ld**.

Nótese que el formato usado para imprimir un dato no necesita estar relacionado con el tipo del dato. C permite interpretar el contenido de una localidad de memoria en cualquier forma que se quiera, aun cuando no tenga sentido.

Ejercicio 1. Escriba un programa que haga las siguientes asignaciones a variables tipo **carácter**. **x = 7** (octal) **y = 44** (octal), **a = 3c** (hexadecimal) y **b = 3a** (hexadecimal). Haga que el programa imprima después las representaciones hexadecimales y de **caracteres** de estos valores.

Ejercicio 2. Escriba un programa para asignar los valores 100 y 200 a las variables **x** y **y**, respectivamente; después almacene la suma de **x** y **y** en la variable **sum**, y por último imprima los resultados de tal forma que la salida se vea como lo siguiente.

La suma de 100 y 200 es 300 (decimal), 454 (octal), y 12c (hexadecimal).

^ ^ ^ ^

Los números marcados con el signo (^) deben imprimirse a partir de los valores de variable correspondientes (p.ej., el 100 se imprime con la **x** usando el formato **%d**.) Los signos ^ no deben aparecer en la salida.

Ejercicio 3. Escriba un programa para mostrar los resultados de la impresión de las constantes de carácter especiales '\t', 'b', '\r', '\f', '\\', y '\"' (p. ej., la proposición **print ("\\tx\n\t\\tx")**; muestra cómo trabaja el carácter tabulador.)

C.8

ARREGLOS

Existen diferencias importantes en la forma de manejar arreglos en C y en la mayoría de los demás lenguajes, pero como casi todas ellas estriban en que muchas operaciones de arreglos implican el uso de apuntadores, estas diferencias se abordan en la sección de apuntadores. En esta sección se analizan características de los arreglos con las que el lector probablemente está familiarizado gracias al conocimiento que tenga de algún otro lenguaje. Un arreglo

- Es un agregado de un solo tipo de datos (incluido el de arreglo);
- Puede tener cualquier número de dimensiones;
- Debe declararse antes de usarse;
- Se indica comenzando con 0, y
- Usa índices enteros (constantes o expresiones).

La declaración

```
int a[16];
```

crea un agregado de 16 variables enteras *a[0]*, *a[1]*, ... *a[15]*. Puesto que la indización se inicia en 0, no existe un *a[16]*.

Los arreglos multidimensionales se declaran y se hace referencia a ellos usando paréntesis cuadrados separados para cada dimensión, como en

```
short tabla[20][30];
char nombres[100][15];
```

que asigna 600 espacios de dos bytes para los enteros short, y 1500 bytes para los caracteres. Los elementos del arreglo se almacenan por renglones (el último índice varía más rápido).

En tiempo de ejecución, C no revisa los límites de las referencias a arreglos. Por lo tanto, en C es perfectamente legal escribir

```
j = 20;
a[j] = 7;
```

haciendo referencia al arreglo *a[]* declarado antes. Los resultados de tales asignaciones son difíciles de predecir, pero podrían ser desastrosos.

A menudo se comete el error de olvidar que el número que indica la longitud de un arreglo no puede usarse como subíndice. Por ejemplo, *a[16]* hace referencia a un elemento que sobrepasa los límites de *a[]* según se acaba de describir.

C no tiene operadores especiales para manejar arreglos, tales como asignaciones que transfieran arreglos completos. Si se quieren tales operadores en C, los debe crear uno mismo en forma de funciones en C.

Ejercicio 4. Examine el siguiente programa y determine lo que será impreso cuando se ejecute. Ejecute el programa y observe qué es lo que en realidad se imprime. Explique los resultados. ¿Por qué no es conveniente usar un valor de 10 para *j*?

```
main ()
{
    int a[10], b[10], j = 10;

    a[0] = 33;
    b[0] = 44;
    a[j] = 55;
    b[j] = 66;
    printf(" a[0] = %d\n b[0] = %d\n", a[0], b[0]);
}
```

C.9

CADENAS

C no tiene un tipo de datos declarable llamado *cadena*. En lugar de ello, una cadena es un arreglo de caracteres que, por convención, termina con el carácter nulo '\0'. Por ejemplo, el siguiente código muestra una forma burda de almacenar la cadena "¿Quién?" en el arreglo de caracteres *str[]*.

```
char c[10]; /* Declaración del arreglo de caracteres */
c[0] = '¿'; /* Asignación de una cadena a c[], */
c[1] = 'Q'; /* un carácter a la vez */
c[2] = 'u';
c[3] = 'i';
c[4] = 'e';
c[5] = 'n';
c[6] = '?';
c[7] = '\0'; /* Nótese que el carácter nulo se almacena */
/* en el arreglo. */
```

Puede pensarse que esta cadena se representa internamente como

c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7] . . .

'¿'	'Q'	'u'	'i'	'e'	'n'	'?'	'\0'	(Basura...)
-----	-----	-----	-----	-----	-----	-----	------	-------------

Sería difícil exagerar la importancia del byte nulo al final de cada cadena. Si se le deja fuera, muchas funciones de cadenas no sabrán cuándo han llegado al final de una cadena. O, si no toma en cuenta el byte adicional necesario para el carácter nulo, un programa podría terminar accidentalmente almacenando un byte nulo en una posición posterior al final del arreglo. En ambos casos, es probable que el programa hiciera estragos en los datos que debieron dejarse intactos.

En C existen *constantes de cadena*. Una constante de cadena se representa encerrando la cadena deseada entre comillas, como en "¿Quién?" Cuando existen constantes de cadena en un programa, se les asigna espacio internamente y se almacenan en la misma forma que las cadenas en arreglos de caracteres. Por lo tanto, la cadena "¿Quién?" se representa internamente tal como se encuentra en la anterior ilustración de cadena, excepto que no tiene nombre. Las constantes de cadena se usan con mucha frecuencia, como en

```
printf ("¿Quién?");
```

No existen operaciones predefinidas sobre cadenas. En lugar de ello, existen varias *funciones* disponibles para efectuar operaciones sobre cadenas. Todas estas funciones suponen que se trabaja con un grupo de bytes consecutivos que terminan con '\0'. Por ejemplo, existe una función llamada *strcpy()* para copiar una cadena a un arreglo. Para asignar la cadena "¿Quién?" a *c[]*, se escribe

```
strcpy (c, "¿Quién?");
```

En la sección de operadores se analizan otras funciones útiles de manejo de cadenas.

Ejercicio 5. Escriba un programa para copiar la cadena "¿Quién es primero?" a un arreglo *a*, y después imprimir la cadena a partir del arreglo empleando el formato %s.

C.10

ESTRUCTURAS

En esta sección se examina brevemente lo que es una estructura, y algunas operaciones sencillas sobre estructuras. El tratamiento completo de las estructuras debe diferirse hasta que se estudien los apuntadores, porque son fundamentales para trabajar con ellas.

Una estructura es un agregado, como un arreglo, pero con una diferencia importante: los distintos elementos de una estructura pueden

tener diferentes tipos de datos, lo que no sucede con los arreglos. Las estructuras son útiles en especial en el procesamiento de archivos, ya que muchas veces es necesario agrupar diferentes tipos de datos de un archivo en un lugar y referirse a ellos en forma colectiva, en vez de hacerlo por datos individuales.

Por ejemplo, considérese un registro de un archivo de personal que contenga el nombre de la persona, la edad, el número de empleado y el salario bruto. Las estructuras permiten agregar toda esta información bajo un nombre de modo que, cuando se quiera, pueda hacerse referencia al agregado completo. He aquí una declaración de una estructura en C para el registro de empleados. Se le llama *reg_emp*.

```
struct {
    char nombre[20];
    short edad;
    char número_empleado[8];
    long salario_bruto;
} reg_emp;
```

Existen varias formas alternativas. Por ejemplo, es posible definir un tipo de estructura sin declarar en realidad una estructura correspondiente. Esto se hace colocando después de la palabra reservada *struct* un identificador, o *rótulo de la estructura*. Aquí el rótulo de la estructura es *A*:

```
struct A {
    char nombre[20];
    short edad;
    char número_empleado[8];
    long salario_bruto;
};
```

Más tarde, el bloque que describe las partes de la estructura puede reemplazarse por el rótulo de la estructura, como en

```
struc A reg_emp;
```

A los campos individuales de una estructura se les denomina con el nombre de la estructura, seguido de un punto, seguido del nombre del miembro, como en

```
reg_emp.edad = 37;
printf("Nombre: %s\n", reg_emp.nombre);
if (reg_emp.salario_bruto > 3000)
    impuesto reg_emp.salario_bruto * 0.35;
```

Ejercicio 6. Diseñe una estructura que contenga la siguiente información con respecto a un miembro de una liga de boliche: *nombre del bolichista* (no más de 20 caracteres), *nombre del equipo* (no más de 12 caracteres), *marca más alta* (entero menor de 301), y *marca media* (número de punto flotante).

Sally Spare pertenece al equipo de boliche los Gatos Bolichistas. Tiene como marca más alta 214 y como marca media 183.7. Escriba un programa que asigne las estadísticas de Sally a la estructura, e imprima los elementos de la estructura en un formato adecuado.

C.10.1 UNIONES

Las uniones permiten que diferentes variables comparten el mismo espacio de almacenamiento; con esto se logran ahorros porque el mismo espacio de almacenamiento se usa para diferentes propósitos en distintos momentos. También pueden usarse para crear estructuras de datos que se adapten fácilmente a diversas combinaciones de datos.

Las uniones usan una notación muy parecida a la de las estructuras. Por ejemplo, las siguientes proposiciones definen una unión de cuatro bytes para almacenar ya sea un *long* o un arreglo de caracteres de cuatro bytes,

```
unión cualquiera { /* "cualquiera" es el rótulo
                     de la unión */
    long número;
    char letras[4];
};
```

En el siguiente código, la primera proposición declara una variable de unión mediante el nombre *compartido*. La segunda proposición almacena una cadena en la localidad de memoria asignada a *compartido*, y la tercera almacena ahí un entero largo, eliminando la cadena que se había almacenado.

```
unión cualquiera compartido;
strcpy (compartido.letras, "ave");
compartido.número = 15L;
```

Ejercicio 7. Escriba un programa que declare una unión que contenga un arreglo de 22 caracteres y un arreglo de seis enteros largos. Haga que el programa almacene la cadena *Primera parte* al inicio del arreglo de caracteres y después haga que almacene los

enteros 255 y 4095 en las dos últimas posiciones del arreglo de enteros largos. Imprima la unión como una serie de caracteres hexadecimales.

C.11

FUNCION *scanf()* PARA LECTURA DEL TECLADO

La función de propósito general *scanf()* de C lee del teclado. La forma general de una proposición *scanf()* es

```
scanf (<formato>, <lista de argumentos>);
```

El *<formato>* se encierra en comillas y contiene los códigos de los formatos de la entrada esperada. La *lista de argumentos* contiene las direcciones de las variables a las que les será asignada la entrada. La dirección de una variable o estructura simple se indica colocando un & antes del nombre; la dirección de un arreglo se indica simplemente al dar el nombre del arreglo. (Los usos de las direcciones se analizan con más detalles en secciones posteriores.)

La siguiente llamada a *scanf()* lee caracteres del dispositivo estándar de entrada (por lo regular el teclado), los interpreta como enteros decimales, los traduce a binario y almacena el resultado en la variable *int r*.

```
scanf ("%d", &r);
```

Cuando se van a leer varios valores, deben separarse mediante cualquier número o combinación de espacios en blanco, línea nueva o tabuladores. Por ejemplo, considérese el siguiente código.

```
short a; float b; char título[30];
scanf ("%d %f %s", &a, &b, título);
```

Si la entrada del teclado es

10

654.3E-2

asistente

entonces los valores 10, 6.543 y la cadena *asistente* (terminada en nulo) son asignados a *a*, *b* y *título*, respectivamente.

Al igual que con *printf()*, no se analizan aquí algunas de las características refinadas de *scanf()*, tales como el uso de anchura del campo y justificación derecha e izquierda de la entrada.

Ejercicio 8. Escriba un programa que use `scanf()` para leer tres números en decimal e imprimálos después como valores hexadecimales.

C.12

OPERADORES

Los operadores en C se diferencian de los de muchos lenguajes familiares en dos aspectos: primero, algunos de los más comunes (como los operadores de cadena) están notoriamente ausentes. C soluciona esto mediante una biblioteca de tiempo de ejecución, la cual contiene funciones útiles que proporcionan efectivamente estos operadores y muchos más. Segundo, C clasifica como operadores algunos elementos (como el operador de asignación) que otros lenguajes no llaman operadores.

En esta sección se analiza la mayoría de los operadores de C, pero algunos requieren un tratamiento especial, por lo que se posterga su estudio hasta las últimas secciones. Entre ellos se encuentran *la dirección de (&)*, *la indirección (*)*, *el apuntador a estructura (→)* y el operador condicional (?:).

C.12.1 OPERADORES + - * / % & & : : !> < =, ETC.

Entre los *operadores aritméticos* se encuentran los operadores binarios familiares +, -, *, /, y % (módulo, sólo para enteros). La división (/) entre dos enteros produce un cociente entero truncado.

Entre los *operadores relacionales* se cuentan <(menor que), > (mayor que), <= (menor o igual que), >= (mayor o igual que), == (igual) y != (distinto). El no distinguir == (igual) y = (asignación) es una fuente de errores común entre los programadores principiantes de C. Se analiza la diferencia entre == y = en la sección donde se estudia el flujo de control.

Los operadores relacionales devuelven 1 para *verdadero* y 0 para *falso*. (En general, 0 significa *falso*, y *distinto de cero* significa *verdadero*).

Entre los *operadores lógicos* están && (Y), || (O) y ! (NO). Tienen los significados usuales.

Los *operadores de bits* permiten la manipulación de bits. Los operadores de bits son & (Y), | (O), ^ (O exclusivo) y ~ (NO). No deben confundirse los operadores de bits con los operadores lógicos correspondientes. Mientras que los operadores lógicos tratan a un operando completo como un valor *verdadero* o *falso*, los operadores de bits se aplican en paralelo a las posiciones individuales de los bits es un operando. Los operadores de bits pueden usarse sólo con tipos de datos enteros.

El operador de bits Y (`&`) puede usarse para ocultar bits. Por ejemplo, si `n` es un entero corto.

```
a = n & 0xFFFF0;
```

pone en cero los cuatro bits inferiores de `n`.

Otros dos operadores son `<<` (corrimiento a la izquierda), y `>>` (corrimiento a la derecha).

`x << n` recorre el valor de `x` a la izquierda `n` posiciones de bits, llenando las posiciones vacías con 0.

Por ejemplo,

```
a = x << 2;
```

recorre `x` dos posiciones efectivamente multiplicando `x` por 4.

La operación `x >> n` recorre `x n` posiciones a la derecha. Si `x` no tiene signo, las posiciones vacías se llenan con 0. Si `x` es otro tipo de entero, el valor que llena las posiciones vacías depende de la máquina.

Ejercicio 9. Prediga los resultados de la ejecución del siguiente programa. Teclee el programa y ejecútelo para revisar sus resultados.

```
main ()
{
    printf ("Aritmético: \n\t%d \n\t%d \n\t%d \n\t%d \n",
            17/5, -17/5, 17%5, -17%5);
    printf ("Relacional: \n\t%d \n\t%d \n\t%d \n\t%d \n",
            5 > 3, 5 < 3, 7 == 7, 8 == 7);
    printf ("Lógico: \n\t%d \n\t%d \n\t%d \n",
            (3>2) || (2>3), (3>2) && (2>3), !(3>2) );
    printf ("Bits: \n\t%d \n\t%d \n\t%x \n\t%c \n",
            3 & 2, 3 | 2, ~0xffff8, 'a' & 0137);
    printf ("Corrimiento de bits: \n\t%d \n\t%d \n",
            37 << 2, 37 >> 2);
}
```

C.12.2 ASIGNACION

En muchos lenguajes, la asignación simplemente implica colocar los datos en un lugar identificado por un nombre de variable. Como se ha visto, esto se hace en C usando el operador `=`, como en `a = b + c`; sin embargo, la asignación en C llega más lejos que esta forma simple, en dos aspectos.

Primero, existen diversas formas de asignación que hacen que se efectúen operaciones antes de que se realice la asignación real. Estas asignaciones toman la forma general

<variable> <op> = <expresión>

lo cual significa lo mismo que:

<variable> = **<variable>** **<op>** **<expresión>**

Considérese *<expresión>* como una expresión semejante a $b + c$, que devuelve un valor. El *<op>* puede ser cualquiera de los operadores aritméticos o de bits. Por ejemplo,

$x += y;$ es equivalente a $x = x + y;$
 $a *= b + c;$ es equivalente a $a = a * (b + c);$

La razón de usar estas notaciones abreviadas es que pueden producir código de ensamblador más reducido y rápido, y mejoran la legibilidad.

La segunda diferencia entre las asignación C y las de muchos lenguajes comunes es que en C un operador de asignación (= o $<op>=$) es un operador verdadero, en el sentido de que devuelve un valor. Sobre este valor puede a su vez operarse. Por lo tanto, en C es posible decir

Esta característica puede explotarse para producir código muy compacto. También puede usarse para producir código obscuro y propenso a errores, como el lector probablemente descubrirá.

Ejercicio 10. Prediga los resultados de la ejecución del siguiente programa. Teclee el programa y ejecútelo para revisar sus resultados.

```
main()
{
    int a=2, b=3, c=5 d, e=6;

    a += 17;
    e *= (d = b + c);
    printf (" a = %d\n b = %d\n c = %d\n d = %d\n e = %d\n",a, b, c, d, e);
}
```

C. 12.3 INCREMENTO Y DECREMENTO

Incrementar (agregar 1) y decrementar (restar 1) es tan común en la programación que los diseñadores de C han proporcionado operadores especiales para ello. El operador de incremento es `++`; el operador de decremento es `--`. Por ejemplo,

<code>++a;</code>	es equivalente a	<code>a=a+1;</code>
<code>--a;</code>	es equivalente a	<code>a=a-1;</code>

Al igual que otras formas de asignación, pueden usarse los resultados de una operación de incremento o decremento, de manera que

<code>b = ++a;</code>	es equivalente a	<code>++a;</code>
		<code>b = a;</code>

El `++` y el `--` pueden usarse antes o después de una variable, pero cuando se usan después de ella tienen un significado un poco diferente. Si `a ++` es parte de alguna expresión más larga, `a` se incrementa sólo después de que se usó su valor. Por lo tanto,

<code>b = a ++;</code>	es equivalente a	<code>b = a;</code>
		<code>a ++;</code>

Ejercicio 11. Prediga los resultados de la ejecución del siguiente programa. Teclee el programa y ejecútelo para revisar sus resultados.

```
main() {
    int a, b, c, d, u, v, w, x;
    a = b = c = d = 2;
    u = ++a;
    v = b++;
    w = 3 + --c;
    x = 3 + d--;
    printf (" a = %d\n b = %d\n c = %d\n d= %d\n", a, b, c, d);
    printf (" u = %d\n v = %d\n w = %d\n x = %d\n" u,v,w,x);
```

C.12.4 FUNCIONES DE PROCESAMIENTO DE CADENAS COMO OPERADORES

Recuérdese que una cadena es un arreglo de caracteres con un terminador nulo ('\0'). He aquí una lista de funciones de procesamiento de cadenas que son particularmente importantes cuando se programa en C. Se encuentran en la biblioteca estándar.

<i>strcat (c1, c2)</i>	Agrega <i>c2</i> a <i>c1</i> . Después de que se ejecuta <i>strcat</i> , <i>c1</i> consiste en todos los caracteres de la <i>c1</i> original, excepto el '\0' del final, seguidos de todos los de <i>c2</i> .
<i>strcmp (c1, c2)</i>	Compara <i>c1</i> con <i>c2</i> ; después devuelve un valor <i>int</i> que es menor que 0 si <i>c1</i> es léxicamente menor que <i>c2</i> , igual a 0 si <i>c1</i> es igual a <i>c2</i> , y mayor que 0 si <i>c1</i> es mayor que <i>c2</i> .
<i>strcpy (c1, c2)</i>	Copia el contenido de <i>c2</i> en <i>c1</i> , sobreescribiendo el contenido original de <i>c1</i> .
<i>strlen (c1)</i>	Devuelve un <i>int</i> que es el número de caracteres en <i>c1</i> , sin considerar el terminador nulo.

Ejercicios 12. Prediga los resultados de la ejecución del siguiente programa. Teclee el programa y ejecútelo para revisar sus resultados.

```
main ()
{
    char c1[40] , c2[40];
    strcpy (c1, "Abril es el mes más cruel. ");
    strcpy (c2, "abril ");
    if (strcmp (c1, c2) < 0)
        printf ("c2 es más grande. \n");
    else if (strcmp (c1, c2) ==0)
        printf ("c1 y c2 son iguales .\n ");
    else
        printf ("c1 es más grande. \n");
    strcat (c1, c2);
    printf ("\nc1 y c2 juntas: %s\n", c1);
    printf ("\nLa nueva longitud de c1: %d", strlen (c1));
}
```

C.12.5 EL OPERADOR *sizeof()*

Un operador especial útil en C es *sizeof()*, que durante la compilación calcula tamaño en bytes de un tipo u objeto de datos. Puede usarse *sizeof()* con objetos de datos reales (variables simples, arreglos, estructuras), tipos de datos (*int*, *short*, *float*, y otros) y expresiones.

Por ejemplo:

```
struct vehículo { /* vehículo es el rótulo de la estructura */
    char nombre[10];
    int peso;
    short año;
};
```

```

struct vehículo camión;      /* camión es un vehículo          */
                           /*          */

v = sizeof (struct vehículo); /* Calcula el número total de bytes usados
                           /* de este tipo de estructura.      */

w = sizeof (camión)         /* Calcula el número de bytes usados por
                           /* esta estructura específica.     */

x = sizeof (short);         /* En la mayoría de las máquinas asigna
                           /* 2 a x.                         */

y = sizeof (int);           /* Depende del tamaño de palabra más
                           /* conveniente de la máquina.       */

z = sizeof (x + 3.2);       /* Devuelve el tamaño del resultado que
                           /* sería devuelto si se evaluara x + 3.2.
                           /* En realidad no evalúa x + 3.2. */

```

Ocurren tres aplicaciones particularmente útiles de `sizeof()` cuando

- El tipo u objeto de datos se define lejos de donde se necesita su tamaño, tal como en un archivo `#include`,
- Es difícil para el programador calcular un tamaño real del objeto, como en el caso de una estructura, o
- El tamaño real de un objeto puede cambiar cuando el programa se lleva a un sistema diferente, como en el caso de las variables `int`.

Ejercicio 13. Escriba un programa para determinar los resultados de los operadores `sizeof` mostrados anteriormente.

C.12.6 CONVERSIONES FORZOSAS (*cast*)

Algunas veces es importante convertir un valor de un tipo de datos a otro. Por ejemplo, considérese la siguiente expresión, en la cual *a* y *b* son variables `int`, y *x* es una variable `float`.

```
x = a / b; /* se pierde la parte fraccional del cociente */
```

Supóngase que se quiere retener la parte fraccional del cociente. Una forma de hacerlo es mediante una *conversión forzosa* de *a* y *b* a `float`, y después una división:

```
x = (float) a/b
```

Las conversiones forzadas tienen la forma general

(<tipo>)

En otras palabras, una conversión forzosa es simplemente un tipo de datos encerrados en paréntesis. Cuando precede a una expresión, los resultados de la expresión se convierten al tipo de datos encerrado entre paréntesis antes de ser usados. Como el operador de conversión forzosa tiene precedencia más alta que cualquier operador binario, efectúa la conversión antes de que se efectúe cualquier operación sobre la variable.

C.12.7 RESUMEN, CON PRECEDENCIA Y ASOCIATIVIDAD

En la tabla c.1,[†] los operadores dentro de un grupo tienen la misma precedencia. Los operadores de precedencia más alta están en la parte más alta de la tabla. La asociatividad de izquierda a derecha significa que cuando varios operadores usados de una expresión tienen la misma precedencia, se evalúan en orden de izquierda a derecha. Por ejemplo, en $a * b / c$, el * y / tiene la misma precedencia, por lo que la expresión se interpreta como $(a * b) / c$.

C.13

FLUJO DE CONTROL

Prácticamente todos los lenguajes tienen varias formas de alterar el orden en que se ejecutan las proposiciones. Por lo regular, incluyen ramificación incondicional, ramificación condicional, ciclos y llamadas a procedimientos. Los procedimientos se estudian en la siguiente sección. Las otras formas se abordan aquí.

C.13.1 BLOQUES DE PROPOSICIONES

Un grupo de proposiciones (que incluye declaraciones) puede tratarse como una sola proposición si se incluyen en un *bloque*, o *proposición compuesta*. Cualquier grupo de proposiciones encerradas entre llaves se considera un bloque. Por ejemplo, el cuerpo de un programa principal es un bloque. De ahora en adelante, cuando se indique <*proposición*>, significa que puede usarse una proposición simple o un bloque de proposiciones.

[†] Adaptado de *The C. Programmer's Handbook*, de M.I. Bolski. Prentice-Hall, 1985, pág.18

TABLA C.1

Precedencia y asociatividad en operadores de C

Operador	Función	Orden de evaluación
()	Llamada a función	IZQUIERDA A DERECHA
[]	Elemento de arreglo	
.	Estructura o miembro de unión	
→	Apuntador a estructura o miembro de unión	
!	No lógico	DERECHA A IZQUIERDA
-	Complemento a uno	
-	Menos unario	
++	Incremento	
--	Decremento	
&	Dirección de	
*	Indirección	
(tipo)	Conversión (cast)	
sizeof	Tamaño en bytes	
*	Multiplicación	IZQUIERDA A DERECHA
/	División	
%	Módulo (o residuo)	
+	Suma	IZQUIERDA A DERECHA
-	Resta	
<<	Corrimiento a la izquierda	IZQUIERDA A DERECHA
>>	Corrimiento a la derecha	
<	Menor que	IZQUIERDA A DERECHA
<=	Menor o igual que	
>	Mayor que	
>=	Mayor o igual que	
==	Igual	IZQUIERDA A DERECHA
!=	Distinto	
&	Operación Y con bits	IZQUIERDA A DERECHA
:	Operación O con bits	IZQUIERDA A DERECHA
&&	Y lógico	IZQUIERDA A DERECHA
::	O lógico	IZQUIERDA A DERECHA
?:	Condicional	DERECHA A IZQUIERDA
=	Asignación (también *= /= %= etc.)	DERECHA A IZQUIERDA

La proposición *if* tiene varias formas. La más simple es

```
if (<expresión>) <proposición>
```

como en

```
if (a==7) ++a;
```

y

```
if (b>=3) {
    d++;
    printf ("d se incrementó.\n");
}
```

Cuando se ejecuta un *if*, se evalúa la expresión entre paréntesis. Si su valor no es 0, se considera *verdadero*, y se ejecuta la <proposición>. Si <expresión> es 0 (falso), se ignora la <proposición>. (Esta interpretación para verdadero y falso se explota mucho en la programación en C.) Por lo tanto, en el primer ejemplo, la expresión

```
a == 7
```

devuelve un 0 si *a* es distinto de 7, y no se ejecuta *++a*. Si *a* es igual a 7, devuelve un 1 y se ejecuta *++a*.

En el segundo ejemplo, *if b >= 3*, se ejecutan las dos proposiciones que componen el bloque. En caso contrario se ignora el bloque.

Es importante apuntar que <expresión> puede ser cualquier expresión válida; cualquier cosa que devuelva un valor. Incluso podría ser una proposición de asignación. Considérese el siguiente ejemplo.

```
main ()
{
    int r = 4,
        p = 8,
        a = 0;

    if (r=p-6) /* examine esta línea con cuidado. */
        a++
    printf ("Se incrementa a.\n");
}
printf ("r = %d\na = %d\n", r, a );
}
```

Si el lector no está acostumbrado a C, es casi seguro que interpretará este ejemplo de forma incorrecta. La expresión *r = p - 6* no compara el valor de *r* con *p - 6*, como se hace en la mayoría de los lenguajes.

valor de r con $p = 6$, como se hace en la mayoría de los lenguajes. (Recuérdese, == se usa para comparar; = se para asignar.) En lugar de esto, $r = p - 6$ asigna el valor 2 a r y, como 2 no es 0, el if supone verdadero y ejecuta el siguiente bloque. (¿Qué valor se imprime para r en el segundo printf?)

C.13.3 LA PROPOSICION *if...else*

La proposición *if... else* tiene la forma e interpretación esperadas. Por ejemplo, en

```
if (j > n)
    t = 7 ;
else {
    a++;
    t = 3;
}
```

Si $j > 1$, se ejecuta la proposición $t=7$; en caso contrario, se ejecutan las proposiciones $a++$; y $t=3$;

C.13.4 EXPRESIONES CONDICIONALES

C tiene un operador que puede usar para representar un *if... else...* condicional. El truco para comprender la utilidad de este operador es considerar que un condicional devuelve un valor en vez de simplemente elegir entre una o más proposiciones por ejecutar. Por ejemplo, supóngase que se quiere asignar a a el máximo de los dos valores de x y y . He aquí una forma de hacerlo.

```
if (x > y) a = x; else a = y ;
```

Pero C proporciona una abreviatura conveniente para tal expresión condicional. La proposición

<e1> ? <e2> <e3>

significa

si <e1> es verdadero, entonces devuelve el valor de <e2>
en caso contrario devuelve el valor de <e3>.

Por lo tanto, esto también asigna el máximo de x y y a a

```
a = (x > y) ? x : y ;
```

Esta construcción puede usarse para acortar el código y expresar su propósito más claramente de lo que puede hacerlo el *if...else...* También puede usarse en algunos lugares donde no puede usarse *if...else...*, como en macrodefiniciones.

Ejercicio 14. Si la variable *n* es un entero, ¿qué imprime la siguiente proposición?

```
n = -1;
printf ("%s\n", (n > 0) ? "mayor que cero" :
           "menor que cero");
```

C.13.5 LA PROPOSICION *if...else if...else if... ANIDADA*

Las proposiciones *if* anidadas se usan con frecuencia para seleccionar entre varias opciones. En el siguiente ejemplo, el código de la derecha se usa para determinar un valor de *impuesto* basado en la tabla de la izquierda.

Tabla:

w	Impuesto
w < 1000	50
1000 ≤ w < 3000	300
3000 ≤ w < 5000	650
5000 ≤ w	800

Código:

```
if      (w < 1000) impuesto = 50;
else if (w < 3000) impuesto = 300;
else if (w < 5000) impuesto = 650;
else          impuesto = 800;
```

C.13.6 LA PROPOSICION *switch*

La proposición *switch* prueba si el valor de una <expresión> corresponde a uno de varios valores constantes mutuamente exclusivos, y después actúa en conformidad con ello.

La forma general de *switch* es[†]

```
switch (<expresión>) {
    case <exp-c> : <0 o más proposiciones>
    case <exp-c> : <0 o más proposiciones>
    :
    case <exp-c> : <0 o más proposiciones>
    default:       <0 o más proposiciones>   ← Opcional
```

[†]Considérese <expresión> como cualquier expresión que devuelve un valor *int*. Considérese <exp-c> como una expresión que contiene sólo constantes enteras, constantes de carácter, y/o operaciones *sizeof*.

Los casos se examinan en forma secuencial hasta que se encuentre una <c-exp> que tenga el mismo valor que <expresión>. Despues de ejecutan todas las <proposiciones> subsecuentes hasta que se encuentra ya sea el final del *switch* o la proposición.

```
break;
```

Si ningún caso coincide, se selecciona el caso opcional *default*, y se ejecutan sus proposiciones. Si los casos no coinciden y no hay proposición *default*, el *switch* no ejecuta ninguna proposición.

Se examinará un ejemplo. Supóngase que, en respuesta a un menú, el usuario digita un carácter (*A*, *C*, *D*, o *E*) seleccionando una de cuatro operaciones a efectuarse en un archivo. Supóngase que cada elección implica un o dos llamadas a función, como se muestra en la tabla. (No importa qué hacen las funciones o cómo trabajan.) La variable *c* es una variable *int* que almacena el carácter tecleado por el usuario.

Valor de <i>c</i>	Significado	Función(es) que se van a llamar
<i>A</i>	Agrega un registro	<i>agr_reg()</i>
<i>C</i>	Cambia un registro	<i>busca()</i> , <i>exhibe()</i>
<i>D</i>	Elimina un registro	<i>elimina_reg()</i>
<i>E</i>	Examina un registro	<i>busca()</i> , <i>exhibe()</i>

Nótese que las selecciones '*C*' y '*E*' requieren la misma acción inicial. He aquí una proposición *switch* que lleva a cabo las llamadas.

```
switch (c) {
case 'A' :           /* Si c == 'A' efectúa todas las proposi- */ 
    agr_reg();        /* ciones que siguen hasta que se encuen- */
    break;            /* tran un break */
                    /* */

case 'O' :           /* Como 'O' y 'E' requieren las mismas */
                    /* acciones, se agrupan. Si 'O'.
case 'E' :           /* coincide, se ejecutan todas las */
    busca();          /* proposiciones subsecuentes hasta que
    exhibe();          /* se encuentra el break. */
    break;
                    /* */

case 'D' :
    elimina_reg();
    break;
default:             /* Esto es opcional, pero recomendado */
    printf ("\nOrden desconocida. \n");
    break;
}
```

lenguaje, asegúrese de comprender por qué el *switch* de C es diferente. En particular, se debe estar consciente de la necesidad de *break* cuando no se quiere que la ejecución continúe a los casos subsecuentes.

La proposición *break* puede usarse además en otros lugares que no sean las proposiciones *switch*. Un *break* hace que el control salga del *while*, *do*, *for*, o *switch* en el que se encuentra inmediatamente dentro.

C.13.7 CICLO *while*

El ciclo *while* de C se parece a la mayoría de los ciclos *while* y funciona más o menos igual que ellos.

```
while (<expresión>
      <proposición>
```

Para cada iteración se evalúa primero la <expresión>. Si es distinta de cero (verdadero) se ejecuta <proposición>. En caso contrario se termina el ciclo. Por ejemplo:

```
i = 0;
while (i < 10) {
    printf ("%d\n", i);
    ++i;
}
```

C.13.8 EL CICLO *for*

Lo equivalente en C al ciclo con conteo es más general que la mayoría. En realidad es una forma especial del ciclo *while* con un formato conveniente para asignar un valor inicial, probar y alterar las variables que controlan la ejecución del ciclo. La forma general

```
for (<expr1>; <expr2>; <expr3>
      <proposición>
```

es equivalente a

```
<expr1>
    while (<expr2> {
        <proposición>
        <expr3>;
    }
```

Este es un ciclo *for* equivalente al *while* de la sección anterior.

Este es un ciclo *for* equivalente al *while* de la sección anterior.

```
for (i = 0; i <10; i++)
    printf ("%d\n",i);
```

El ciclo *for* se prefiere al ciclo *while* cuando la asignación de valores iniciales, la prueba y la alteración de la variable de control del ciclo son simples, ya que los presenta juntos al principio del ciclo.

Una diferencia entre el ciclo *for* de C y los ciclos de conteo de algunos otros lenguajes es que la variable de control siempre retiene su valor después de la finalización del ciclo. Además, no hay restricciones para la alteración de cualquier variable de control dentro del cuerpo del ciclo.

<Expr1>, <expr2> y <expre3> pueden consistir en más de una expresión, siempre que las expresiones que contengan estén separadas por comas. Por ejemplo, he aquí un ciclo *for* que cuenta el número de espacios en una cadena *c* cuya longitud no puede exceder de 100:

```
for (cont = 0, j = 0; c[j] != '\0' && j<100; j++)
    if (c[j] == ' ') cont++;
```

C.13.9 EL CICLO *do...while*

El ciclo *do...while* es parecido al ciclo *while*, excepto que efectúa la prueba de terminación en la parte inferior del ciclo en lugar de hacerlo en la parte superior. Por lo tanto, su forma es

```
do
    <proposición>
    while (<expresión>) ;
```

Ejercicio 15. Escriba tres programas para asignar el valor inicial 7 a cada entero en un arreglo de 100 enteros. Use *while*, *for* y *do...while*, respectivamente.

Ejercicio 16. Se supone que el siguiente programa busca la primera aparición de un espacio en blanco en la cadena *cad* y después escribe su posición. Parte del código no aparece. El Lector debe insertar el código faltante. Hágalo una vez empleando un ciclo *for* y otra usando un ciclo *while*. Pruebe su código con la cadena "Contestar el teléfono.", después con la cadena "Nunca." (el espacio sólo al final), luego con "Hola." (espacio al principio), y después con "Hola." (sin espacios).

```

static char *cads [] = {"Contestar el teléfono.", /*Esta útil estructura de */
                      " Nunca. ",           /* datos          */
                      " Hola. ",            /* tendrá más sentido*/
                      " Hola. ",            /* cuando se      */
                      " " } ;              /* aprenda acerca de */
                               /* apuntadores.   */

main()
{
    char cad[20];
    int n, j;

for (j = 0 ; cads[j][0]!='\0' ; j++) {
    strcpy (cad, cads [j]) ;

    /* El código del ciclo for o while que usted escribe va aquí */

    if (cad [n] == ' ')
        printf (" El espacio en blanco está en la posición %d\n" , n);
    else
        printf ("No se encontró el espacio en blanco. \n");

} /* Fin del ciclo más externo */

```

C.14

FUNCIONES

Pocos lenguajes refuerzan tanto el desarrollo de una biblioteca de funciones como lo hace C. La idea de construir un conjunto de funciones-herramienta para usarse en la construcción de programas de alto nivel es básica en la filosofía de C.

C.14.1 LO BASICO

C no hace distinción entre subrutinas y funciones. Todo subprograma de C es una función en el sentido de que devuelva un valor. Esto es consistente con la idea de que toda expresión, incluida la asignación, devuelve un valor.

La forma general de una definición de función es:

```

<nombre> (<lista de argumentos, si existe>)
<declaración de argumentos, si existen>
{
    <declaraciones y proposiciones, si existen>
}

```

La función <nombre> debe ser precedida por un tipo si devuelve un valor cuyo tipo no es *int*.

Hay dos formas de devolver valores de una función: por medio de una lista de argumentos y por medio de una proposición *return*. Posteriormente se analizará el uso de la lista de argumentos para la devolución de valores, puesto que implica el uso de apuntadores. La proposición *return* tiene la forma

```
return (<expresión>) ;
```

y puede aparecer en cualquier lugar dentro del cuerpo de la función, tantas veces como sea necesario. La proposición *return* hace dos cosas. Devuelve el valor de <expresión>, y le devuelve el control a la función que llama.

He aquí una función llamada *max3()*, que devuelve el mayor de tres argumentos *int*,

```
max3 (a1, a2, a3)
int a1,a2,a3
{
    int m;
    m = (a1 > a2)      ? a1 : a2;
    return ((m > a3) ? m ; a3);
}
```

Las funciones se invocan nombrándolas y proporcionando sus argumentos reales. Por ejemplo, *max3()* puede llamarse mediante una proposición como la siguiente:

```
x = max3 (x, y, 100) ;
```

Para imprimir el máximo de *x*, *y* y 100, se puede decir:

```
printf ("%d\n", max3 (x, y, 100)) ;
```

Puesto que algunas funciones no necesitan devolver valores, no es preciso colocar algo después de la palabra *return*. De hecho, la proposición *return* puede omitirse por completo si no se devuelve algún valor. Si las proposiciones del cuerpo de la función no incluyen un *return*, el control se devuelve cuando la ejecución alcanza la llave que cierra la función. He aquí una función que no devuelve ningún valor.

```
saludo (n)
int n;
```

```

{
    if (n == 0) return;
    if (n > 0)
        printf ("Bueno... Hola, ¿qué tal? \n");
    else
        printf ("Oh... eres tú .\n");
}

```

He aquí una definición de una función que llena las primeras 79 columnas de la pantalla con un determinado símbolo, junto con un programa *main()*, que llama a la función.

```

main()
{
    llena_pantalla ('+');
}

llena_pantalla (símbolo)
char símbolo;
{
    int r, o;
    for (r = 0; r < 25; r++) {
        for (c= 0; c<79; c++) printf ("%c", símbolo) ;
        printf ("\n");
    }
}

```

La proposición *llena_pantalla ('+')*; hace que la pantalla se llene con el símbolo '+'. Las funciones *main()* y *llena_pantalla()* pueden escribirse en un solo archivo, que puede entonces compilarse y ejecutarse.

Nótese que *llena_pantalla()* está separado de *main()*. No se incluye dentro del bloque que define a *main()*. Las funciones de C no pueden definirse dentro de otras funciones, ni dentro de *main()*.

Ejercicio 17. Reescriba *llena-pantalla()* para que sólo llene cierto número de columnas a la izquierda de la pantalla con el símbolo; el número de columnas debe darse en la lista de argumentos.

Ejercicio 18. Escriba y pruebe una función que tome un carácter como único argumento y devuelva el entero 0 si el carácter es '0', 1 si el carácter es '1', 2 si es '2', y así sucesivamente. Si el argumento no es un carácter que represente un dígito, la función debe devolver -1.

C.14.2 ARGUMENTOS DE FUNCIONES (PRIMER RECORRIDO) Y OTROS ASPECTOS

Los argumentos de funciones se pasan por valor. Esto significa que cuando se llama a una función, se evalúa cada argumento y se pasa una copia del resultado a la función, que usa esa copia. Los argumentos de la función son locales a la función y no son accesibles fuera de ella. Cuando se devuelve el control no se ha realizado ningún cambio en los argumentos originales.

Esto parecería implicar que no es posible devolver valores de una función por medio de la lista de argumentos. En realidad sí es posible, pero se requiere el uso de apuntadores, los cuales se analizarán en una sección posterior. El uso de variables externas proporciona otra manera (por lo regular menos deseable) de lograr que las funciones modifiquen el valor de las variables. Las variables externas se analizan en la sección sobre accesibilidad de variables.

C no verifica que los tipos de los argumentos formal y real sean los mismos. Es responsabilidad del programador asegurarse de que los tipos declarados de los argumentos de la función sean iguales a los tipos de los argumentos empleados cuando se invoca la función. El no revisar el tipo de los argumentos es una de las principales fuentes de error en C. (Esto es particularmente cierto cuando se transportan programas entre máquinas que usan diferentes tamaños de variables *int*.)

Las funciones en C pueden usarse en forma recursiva. No tiene que hacerse ninguna declaración adicional cuando se escriben funciones recursivas.

Las definiciones de funciones no pueden anidarse. En la práctica esto significa que todas las funciones son externas y que, con una excepción, cualquier función puede tener acceso a cualquier otra. (La excepción ocurre cuando las funciones se declaran *static*, este caso se analiza en la siguiente sección.)

C.15

ACCESIBILIDAD DE LAS VARIABLES

Cuando se escribe un programa grande es útil poder definir el número de veces y los lugares en que las variables serán accesibles. La accesibilidad de las variables en C está determinada en dos formas: la especificación de *clases de almacenamiento* para variables, y las *reglas de alcance*. La clase de almacenamiento de una variable es un atributo especificable que indica cuándo y dónde puede ser usada: su *alcance*. Las reglas de alcance determinan la accesibilidad de una variable, con

base en el lugar donde sea declarada. Es evidente que las clases de almacenamiento y las reglas de alcance están estrechamente relacionadas.

C. 15.1 CLASES DE ALMACENAMIENTO

Ya se había mencionado que los argumentos de las funciones son variables locales. Cualquier variable declarada en el cuerpo de una función es también local. Por definición, las variables locales tienen clase de almacenamiento automática, lo cual significa que son creadas cuando la función se activa, y se eliminan cuando se sale de ella. Puede declararse una variable para que tenga la clase de almacenamiento *estática* () si se quiere que mantenga su valor de una llamada de función a otra.

Las funciones de C pueden tener acceso a variables que no son locales. Las variables que se definen fuera de cualquier función tienen la clase de mantenimiento *externo* (*external*). Normalmente, las variables externas están disponibles para todas las funciones. (Exceptuando las variables externas estáticas, que están disponibles sólo dentro del archivo donde se declaran.)

Lo siguiente es un resumen de las clases de almacenamiento usadas en C.

Automática

Estas variables son locales respecto a un bloque y se desechan al salir de él. A menos de que se especifique otra cosa, las variables declaradas dentro de una función o bloque son automáticas. Esto incluye los argumentos formales de la función.

Estática

Estas variables retienen sus valores a lo largo de la ejecución de un programa. Las variables estáticas deben ser declaradas en forma específica con el atributo *static*.

Las variables estáticas que se declaran dentro de un bloque o función son internas (como las automáticas) en el sentido de que son accesibles sólo dentro del bloque o función donde se declararon. Si se sale del bloque y posteriormente se vuelve a entrar, las variables estáticas pueden reutilizarse sin volver a asignar el valor inicial. Tienen el mismo valor que tenían cuando se usaron por última vez.

Las variables estáticas que se declaran fuera de cualquier función son accesibles a cualquier función dentro del archivo fuente donde se declararon, pero sólo allí. Este tipo de variable estática es útil cuando

Externa

se quiere tener variables que son globales respecto a cierto archivo de funciones, pero inaccesibles a otros. Estas variables son globales. Esto es, retienen sus valores y son accesibles a muchas funciones. Una variable definida fuera de cualquier función es externa. Las variables externas declaradas *static* son accesibles sólo dentro del archivo en el cual se declaran. Otras variables externas son accesibles a funciones en otros archivos sólo si son declaradas *extern* dentro de esos otros archivos. (Esta situación se explica en la sección C.15.2.)

Registro

Estas variables son automáticas, cuyos valores se almacenan en registros rápidos. No tiene mayor interés aquí el uso de las variables de registro.

Typedef

En realidad, esto no es una clase de almacenamiento, pero con frecuencia se incluye en descripciones de clases de almacenamiento. Puede usarse *typedef* para definir identificadores que pueden usarse posteriormente como si fueran palabras clave de tipos de datos. Por ejemplo, una definición compleja de una estructura puede definirse como un tipo, como se muestra en el siguiente ejemplo.

```
typedef struct {
    char llave[30];
    long nrr;
} NODOLLAVE;
```

De aquí en adelante, puede tratarse a *NODOLLAVE* como si fuera un tipo de dato, como en las declaraciones

```
NODOLLAVE nodo; /* nodo simple del tipo NODOLLAVE */
NODOLLAVE arreglo_nodo[100] /* nodos del tipo NODOLLAVE */
```

El uso de *typedef* puede mejorar significativamente la legibilidad del código en ciertas circunstancias, sobre todo, del código que implica usos complejos de apuntadores.

C.15.2 DECLARACIONES DE DEFINICION CONTRA DECLARACIONES DE REFERENCIA A VARIABLES

Hay dos cosas que un compilador C debe hacer antes de que una variable esté disponible para usarse. Primero, debe asociar el nombre de la variable con un bloque de almacenamiento que separa. A esto se llama *definición* de la variable. Despues debe asociar al almacenamiento

con ciertos atributos que gobiernan la forma de tratar los bits. Esto permite al programa hacer *referencias* a la variable en una forma significativa.

El término *declaración* implica que se están realizando los dos fenómenos. Por ejemplo, cuando se declara una variable del tipo *short* dentro de una función, como en

```
short sum;
```

el compilador asigna dos bytes de almacenamiento con el nombre *sum* (la declaración de definición), y asocia al nombre *sum* información acerca de cómo se interpretan los datos almacenados en los dos bytes (la declaración de referencia).

En circunstancias normales, las declaraciones de definición y de referencia ocurren al mismo tiempo que la declaración. Sin embargo, hay una excepción. Cuando una variable se declara en forma externa (fuera de cualquier función) en un cierto archivo se define (se le asigna almacenamiento), pero sólo se puede hacer referencia a ella desde el archivo. No está disponible para referencias desde otro archivo. A fin de que la variable esté disponible para las funciones de otros archivos, debe de declararse dentro de estos archivos. Tales redeclaraciones comienzan con la palabra clave *extern*, como en

```
extern short sum;
```

Esta declaración no define a *sum*. Se supone que *sum* ya ha sido definida. En general, una variable externa sólo puede definirse una vez. Todas las otras declaraciones deben comenzar con la palabra clave *extern*, para asegurarse de que no se están redefiniendo.

C.15.3 REGLAS DE ALCANCE

Las reglas de alcance de C son similares a las de la mayoría de los lenguajes estructurados en bloques:

1. Si una variable se declara fuera de cualquier función, entonces es accesible desde el punto donde fue declarada y en el resto del archivo fuente en el que se declaró.
2. Un parámetro formal es accesible sólo dentro de la función en la cual se declara.
3. Una variable declarada dentro de un bloque es accesible en el bloque en el que se declara y en cualesquiera bloques que estén anidados dentro de él.

4. Si un bloque contiene una declaración de una variable con el mismo nombre que el de una variable declarada en el bloque que lo rodea (incluyendo una declaración externa), la variable declarada en el bloque más externo no es accesible si no hasta que se abandona el bloque más interno.
5. Si una función declara una variable como *extern*, debe existir la declaración externa de *definición* correspondiente en algún otro lugar entre las bibliotecas y archivos que constituyen el programa completo. La declaración de definición no contiene la palabra clave *extern*.

C.16

REGLAS DE ALCANCE PARA FUNCIONES

Puesto que las definiciones de función no pueden anidarse, todas las funciones de C tienen la clase de almacenamiento externo. Esto significa que, en general, toda función es accesible desde cualquier lugar en un programa.

La única excepción son las funciones que son declaradas *estáticas* en su definición. Se puede acceder a las funciones estáticas sólo desde el archivo en el cual fueron definidas. Esta característica de las funciones estáticas hace posible escribir funciones de C que tengan la misma naturaleza privada y local que un procedimiento interno en el lenguaje como PL/I.

Todos los problemas en C deben tener al menos una función *main()*. La ejecución de un programa en C siempre se inicia por la función *main()*. Todas las demás funciones están subordinadas a *main()* y son llamadas directa o indirectamente a partir de allí.

Las otras dos categorías de funciones en C son las funciones estándar de biblioteca y las funciones definidas por el usuario.

C.16.1 BIBLIOTECAS Y OTROS ARCHIVOS QUE CONTIENEN FUNCIONES

Virtualmente toda implantación de C incluye uno o más archivos que contienen la funciones de C más necesarias. Uno de tales archivos contiene la biblioteca estándar de E/S que se describió antes. Una implantación típica de C también tiene bibliotecas con funciones que efectúan otras importantes tareas de programación, como las funciones matemáticas comunes. El lector debe consultar la documentación del sistema que emplea para informarse acerca de las bibliotecas disponibles.

De algunas bibliotecas, como la biblioteca estándar de E/S, se dispone en forma automática. Otras, entre ellas las creadas por los programadores para propósitos especiales, no están automáticamente disponibles. Las bibliotecas deben incluirse con un archivo fuente antes de la compilación, usando la orden *#include* del preprocesador, o bien deben ligarse al archivo objeto después de la compilación. Consultese la documentación del sistema para informarse sobre cómo crear y ligar archivos objeto de biblioteca.

EL PREPROCESADOR

Antes de compilar un archivo fuente C, un preprocesador examina el texto fuente para conocer ciertas órdenes para el *preprocesador*. Las órdenes para el preprocesador brindan medios convenientes para definir constantes y macros, para incluir otros archivos junto con el archivo fuente y para la compilación condicionada. Muchos lenguajes proporcionan posibilidades de preprocesamiento, pero pocos las incorporan en forma tan integral en el proceso de desarrollo de programas como lo hace C. El uso efectivo de los conjuntos de herramientas reside principalmente en el uso del preprocesamiento.

Todas las órdenes para el preprocesador comienzan con el carácter '#'. (La mayoría de los compiladores requieren que el # esté en la columna 1.)

C.17

LA ORDEN # *define*

Una orden de la forma

```
#define <nombre> <cadena-lexicográfica>
```

hace que <cadena-lexicográfica> sustituya a <nombres> cada vez que <nombre> ocurre en el archivo fuente después de la definición. A tal sustitución se le llama *macrosustitución*. En su forma más simple <cadena-lexicográfica> es sólo una constante, pero puede ser cualquier cadena de caracteres que el programador desee que se sustituya en el código. También puede contener un tipo de argumentos. He aquí algunos ejemplos.

```
#define LONG_MAXIMA 350           /* una constante numérica */
#define PETICION     "\nProporcione dos números\n"/* una constante de cadena */
#define IMP_A        printf ("a= %d\n", a)      /* una llamada específica
                                                a función */
#define MAX (a,b)    ((a) > (b) ? (a) : (b))   /* uso de argumentos */
```

Usualmente las proposiciones `#define` preceden a `main()`, pero pueden aparecer en cualquier lugar. He aquí un ejemplo de cómo pueden usarse algunos de los `#defines`.

```
#define PETICION     "\nProporcione dos números\n"
#define MAX (a, b)    ((a) > (b) ? (a) : (b))
main()
{
    int x, y;

    printf (PETICION);
    scanf (" %d %d", &x, &y);
    printf ("El ganador es . . . %s\n", MAX (x,y));
}
```

Los argumentos de la macro (p. ej., *a* y *b* en la definición de *MAX*) no deben confundirse con argumentos de función. Cuando un nombre de variable se usa como argumento en una función, su **valor** se pasa a la función. Cuando un argumento se usa en una sustitución de macro, no interviene ningún valor, sino que los símbolos que se usan en lugar de los argumentos (*x* y *y* en el ejemplo) simplemente reemplazan a los argumentos mismos.

He aquí algunas razones que justifican el uso de `#define`.

- Una constante usada con frecuencia puede definirse para que tenga cierto valor en una aplicación de un programa, y después cambiarse en otra aplicación simplemente cambiando una proposición `#define`.
- Puede ocultarse código complejo empleando expresiones sencillas en lugar del código correspondiente, como en la macro *MAX(a, b)*, en el ejemplo anterior.
- Una macro se ejecuta más rápido que la función correspondiente, porque no implica el gasto generado por llamar y regresar.
- Los argumentos de la macro pueden ser genéricos; puesto que no se declaran, pueden construirse de manera que puedan ser sustituidos por datos de cualquier tipo.

C.18

LA ORDEN #*include*

La orden **#include** facilita llevar a un archivo funciones de herramientas, **#define** y declaraciones de variables globales. En la introducción se analizó el importante papel de **#include**. Una orden de la forma

```
#include "nombre_archivo"
```

hace que el contenido del archivo indicado se procese como si apareciera en el mismo lugar que la orden **#include**.

El preprocesador busca el archivo indicado, primero en el directorio que almacena el programa en C original y luego, si no lo encuentra ahí, en los directorios del sistema estándar. Si se desea que el preprocesador busque sólo en los directorios del sistema estándar, se reemplazan las comillas con < y >, respectivamente, como en

```
#include <nombre_archivo>
```

C.19

COMPILACION CONDICIONAL

Entre las proposiciones del preprocesador hay algunas que permiten que parte de un programa se compile o no, dependiendo de cierta condición. La compilación condicional puede ser útil cuando se depura, o cuando la elección de las proposiciones que se han de compilar depende de la máquina que se esté usando. Aquí no se estudia la compilación condicional.

Ejercicio 19. Para que el lector se asegure de que sabe cómo funciona **#include**, cree un archivo con los **#define** ilustrados en el texto, y un segundo archivo con el programa principal que usa las definiciones. Use una proposición **#include** en el segundo archivo para incluir las definiciones del primero.

APUNTADORES

Todos los tipos de datos que se han analizado tienen significado fuera de sus usos en computación. Por ejemplo, los enteros y los caracteres han existido desde que el hombre comenzó a contar y a escribir. Los apuntadores son distintos; son específicos para la computación.

Los apuntadores proporcionan una forma de referirse a la localidad de la memoria del computador donde está almacenado el patrón de bits que representa algún dato. Los apuntadores son direcciones de memoria. La mayoría de los lenguajes de computación no pide a los programadores el uso de apuntadores, pero C lo exige.

Por lo regular, cuando se trabaja con variables simples no hay necesidad de averiguar la dirección de un valor, y menos aún manipular esa dirección. Sin embargo, a veces es muy conveniente poder manipular las direcciones. Kernighan y Ritchie [1978] explican: "Los apuntadores se usan mucho en C, en parte porque algunas veces son la única forma de expresar un cálculo, y en parte porque por lo regular llevan a un código más compacto y eficiente que el que puede obtenerse en otras formas."

C.20

LO BASICO

Para hablar de los apuntadores es necesario revisar el concepto usual de una variable. Una variable en C tiene asociados los siguientes cuatro objetos:

- Un nombre;
- Un valor en la forma de un patrón de bits que se almacena en algún lugar de la memoria del computador;
- Un tamaño, que indica cuántos bytes de memoria requiere, y
- Un apuntador, que identifica el lugar de la memoria donde está almacenado el valor.

Cuando se usa un nombre de variable sencillo en un programa, el computador debe determinar en qué sitio de la memoria se localiza el valor asociado con el nombre. Por lo tanto, para todo nombre de variable debe existir un apuntador. Por ejemplo, supóngase que se tiene la declaración y asignación

```
short sum;
sum = 463;
```

El valor se *sum* se almacena en algún lugar de la memoria (p. ej., la localidad 1036) y ocupa dos bytes. En forma esquemática:

<i><nombre></i>	<i><apuntador></i>	—————>	<i><valor></i>
sum	localidad 1036	—————>	463

Si se desea trabajar con un apuntador en forma explícita, debe dársele un nombre. Hay dos formas de hacer esto en C. Una es usar el operador `&` para la dirección de, como en

`&sum`

En el ejemplo, `&sum` se refiere a la localidad 1036. Una forma más general de referirse a un apuntador es usar una variable tipo apuntador, la cual puede declararse como sigue.

```
short *asum;           /*apuntador de tipo apuntador a
                           entero */
```

El `*` indica al compilador que a `asum` se le puede asignar la dirección de un `short`. Nótese que un tipo apuntador se describe en términos del tipo de dato al que apunta, de modo que `asum` es un *apuntador a short*. Pronto se verá la importancia que esto tiene. Para asignar a `asum` la dirección de `sum`, escribase

```
asum = &sum; /* asignación de una dirección */
```

Cuando no se usa en una declaración, el operador unario `*` representa el contenido de la localidad a la cual apunta, por lo que en el siguiente código las dos proposiciones de impresión hacen lo mismo.

```
asum = &sum;
printf ("La suma es %d\n", sum);
printf ("La suma es %d\n", *asum);
```

C.21

NOMBRES DE ARREGLOS COMO APUNTADORES

En C, el nombre de un arreglo es un apuntador al primer elemento del arreglo nombrado. Los elementos individuales del arreglo se identifican con el nombre del arreglo seguido de un subíndice. Considérese la declaración

```
long a[100];
long *aa;
```

La primera declaración hace que el computador asigne espacio para almacenar 100 enteros largos (de cuatro bytes cada uno). La variable `a` es un apuntador a `long` para el primer elemento del arreglo. Proporciona

la dirección del primer elemento del arreglo (que es un *long*). El nombre con subíndice *a[0]* proporciona el valor del primer elemento del arreglo.

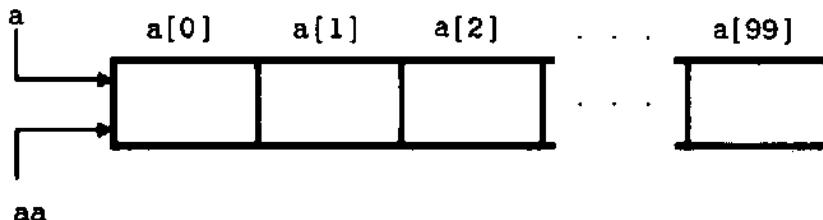
Como *aa* es un *apuntador a long*, tiene el mismo tipo de datos que *a*, y a *aa* se le puede asignar el valor de *a*, como en

```
aa = a;
```

Como *&a[0]* también apunta al primer elemento de *a*, puede hacerse lo mismo con

```
aa = &a[0];
```

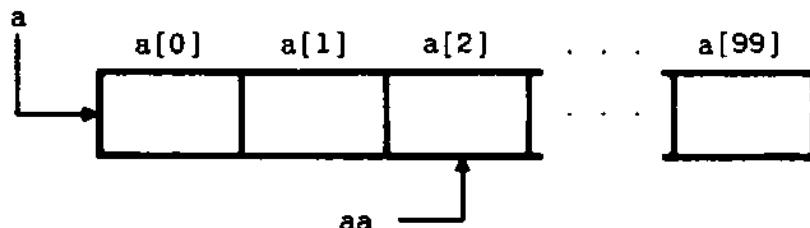
Este diagrama ilustra el resultado:



La proposición

```
aa = &a[2];
```

cambia el dibujo a



C.22

ARITMETICA DE APUNTADORES Y ARREGLOS

Con los apuntadores pueden efectuarse algunos tipos de operaciones aritméticas muy convenientes. Si un entero *i* se suma al apuntador *aa*, el resultado es un apuntador al *i*-ésimo elemento más allá del apuntado por *aa*. Por ejemplo, el siguiente código suma los elementos numerados en forma impar en el arreglo apuntado por *a*.

```

long    a[100], *aa, sum = 0;

for (aa = a; aa < &a[100]; aa += 2)
    sum += *aa;

```

Nótese que en la expresión $aa += 2$, el computador no sólo suma el valor 2 a la dirección aa , sino que debe considerar el tamaño del tipo de dato apuntado por aa . Como aa es un apuntador a un entero long, probablemente sume $8(2 \times 4 = 8)$ para obtener cada nueva dirección.

En general, cuando se suma un entero i a un apuntador, i se multiplica por el tamaño del objeto al cual apunta el apuntador y después se suma al valor del apuntador. El resultado es un apuntador que apunta a un elemento del arreglo desplazado i elementos del elemento original.

Los operadores de incremento y decremento funcionan en forma análoga: $aa++$ suma a aa el tamaño del objeto apuntado por aa (cuatro en el ejemplo). Por lo tanto, si aa apunta al primer elemento del arreglo a , las siguientes expresiones son equivalentes.

$aa + 3$ $a + 3$ $\&a[0] + 3$ $\&a[3]$

La resta de un entero a un apuntador funciona en forma similar. Sin embargo, si un apuntador se resta de otro apuntador, el resultado es un entero que corresponde al número de elementos que hay entre los dos.

Ejercicio 20. Escriba un programa que asigne la cadena

**“El para-sí se anuncia a sí mismo con totalidad
destotalizada.”**

**a un arreglo, y después imprima un carácter de cada tres,
comenzando con el primero (E). Use un apuntador para recorrer
el arreglo. No use subíndices.**

C.23

APUNTADORES Y ARGUMENTOS DE FUNCIONES

En C, los argumentos de funciones se pasan *por valor*. Esto significa que una función no conoce la localidad de una variable cuyo valor recibe por medio de un argumento. Como resultado, las funciones no son capaces de cambiar los argumentos. En otras palabras, una función puede

recibir un valor a través de su lista de argumentos, pero no puede cambiarlo a través de ella. Esa es la mala noticia.

La buena noticia es que, usando apuntadores, es posible cambiar el valor *apuntado* mediante un argumento de función. Cuando un argumento de función es un *apuntador* a la dirección de una variable, la función sabe dónde se almacena el valor de la variable. Entonces puede cambiar el valor de la variable cambiando el valor almacenado en la localidad apuntada.

C.23.1 CAMBIO DE PARAMETROS QUE SON VARIABLES SIMPLES

Se desarrollará una función que determine la posición en una cadena donde ocurra por primera vez un carácter en particular, y también en donde ocurra por última vez. Si el carácter no está presente, se espera que la función devuelva 0.

La función devuelve tres valores separados. A los primeros dos, que son parámetros, se les llama *primero* y *último*. El tercer valor es devuelto por medio de la proposición *return*. He aquí una función que hace el trabajo.

```
encuentra_pu (cad, car, primero, último)
char cad[];
char car;
int *primero, *último;
{
    /* la cadena que se va a recorrer */
    /* el carácter que se va a buscar */
    /* primero y último son direcciones.
       *primero y *último serán valores int
       almacenados en las direcciones primero y
       último */
    lon = strlen (cad);
    for (i = 0; cad[i] != car && i <lon; i++)
        ;
    if (i == lon)           /* si no lo encuentra, regresa */
        return (0);
    else
        *primero = i;        /*en caso contrario, asigna i al lugar */
        /* de memoria apuntado por primero */
    for (i = lon - 1; cad[i] != car; i--)   /*buscar el *último
                                               hacia atrás */
        ;
    *último = i;
    return (1);
}
```

Cuando se llama a *encuentra_pu()*, sus argumentos deben corresponder, por supuesto a los argumentos formales proporcionados en la definición. En particular, *primero* y *último* deben ser apuntadores.

Se sabe que puede hacerse referencia al apuntador de una variable empleando el operador `&`, dirección de, de tal forma que una llamada a `encuentra_pu()` puede ser algo parecido a esto:

```
int x, y; /* los parámetros reales, destinados a primero
y último */  
  
r = encuentra_pu ("uno#dos#tres#cuatro#cinco", '#', &x, &y);  
printf ("primero = %d y último =%d\n", x, y);
```

Ejercicio 21. Escriba una función `clasifica(c, M, m, d, o)` que examine la cadena en el arreglo `c` para determinar cuántas letras mayúsculas, minúsculas, dígitos y otros caracteres existen, y devuelva las cantidades de `M, m, d` y `o`, respectivamente.

C. 23.2 CAMBIO DE PARAMETROS QUE SON ARREGLOS

En C, el *nombre de un arreglo* es un apuntador al primer elemento de un arreglo. Por lo tanto, cuando un arreglo se pasa a una función como parámetro, no hay necesidad de usar el operador `&`. Al declarar un parámetro en una función que corresponde a un arreglo, se tienen dos opciones. Puede declararse como apuntador, como en

char *cad; /* apuntador a un valor de tipo char */
o puede declararse como arreglo lo que hace de su nombre un apuntador, como en

char cad [] ; /* así se hizo en `encuentra_pu()` */

Las dos versiones son equivalentes. La elección de una de ellas dependerá de cuál se juzgue más legible en el contexto de la tarea que se está efectuando.

La siguiente función copia el contenido de un arreglo de tipo `short` a otro.

```
copiashort (a1, a2, longitud) /* Copia todos los elementos de a2 a a1.
                                longitud da el tamaño de a2 */  
  
short      a1[], a2[] ;
int       longitud ;
{
    int i ;
    for (i = 0 ; i <longitud ; i++)
        a1[i] = a2[i] ;
}
```

Tampoco es difícil hacer una llamada de *copiashort*, como en

```
short x[20], y [20], j;

for (j = 0; j <20; j++)           /* Carga y [] para que */
    y [j] = j*j;                 /* exista algo que copiar */

copiashort (x, y, 20);
```

Nótese que no se requiere el operador & con los parámetros *x* y *y*, porque ya son apuntadores. Si el operador & se usara en este caso, el resultado no sería el deseado.

A menudo se explota en C el hecho de que se definan *variables* tipo apuntador a nombres de arreglos. Por ejemplo, considérese la siguiente función, que hace lo mismo que *copiashort()*.

```
nuevo_copiashort (a1, a2, longitud)
short      *a1, *a2; /*lo mismo que short a1[], a2[] */
int       longitud;
{
    .
    int i ;
    for (i = 0; i <longitud; i++)
        *a1++ = *a2++; /* Bastante distinto que a1[i] = a2[i]; */
}
```

La declaración *short*a1, *a2;* en la nueva versión es exactamente la misma que *short a1[], a2[];* en la versión anterior. Pero las proposiciones que transfieren los números de *a2* a *a1* funcionan de manera ligeramente distinta en las dos versiones.

En *nuevo_copiashort()* la proposición **a1++ = *a2++;* hace que el valor apuntado por *a1* sea el mismo que el apuntado por *a2*, y después incrementa los apuntadores *a1* y *a2* de modo que apunten a las siguientes posiciones en sus arreglos correspondientes. El efecto es el mismo que si *a1* y *a2* fueran subindizados (como en la primera versión). La ventaja que tiene hacerlo en la segunda forma es que toma mucho menos tiempo incrementar un apuntador que subindizar un arreglo.

Ejercicio 22. Escriba una función *subcad(c1, c2 , inicio, n)* que copie *n* caracteres de *c1* a *c2*, comenzando con la posición indicada por *inicio*, y coloca un nulo al final de los caracteres que se mueven a *c2* (para volverla una cadena válida). La función *subcad()* debe devolver 0 si la cadena *c1* no es lo suficientemente larga.

Ejercicio 23. Recuérdese que la función *strcpy(c1, c2)* copia el contenido de *c2* a *c1*. Si un programador teclea el siguiente código, ocurre un error.

```
char cad[30];
strcpy (cad, '¡No me muevas!');
```

Es difícil predecir cuál será el mensaje de error que se exhiba. (El sistema puede continuar en forma silenciosa sin exhibir mensaje alguno.) El motivo del error es que la proposición está intentando acceder a una parte del almacenamiento que no está disponible. Explique.

C.24

APUNTADORES Y ESTRUCTURAS

Existen sólo tres operaciones que pueden efectuarse sobre una estructura de C: tomar su dirección, acceder a uno de sus miembros y tomar su tamaño. Una de las implicaciones que tiene esto es que no se puede pasar simplemente una estructura completa a una función, como se haría con una variable simple.[†] Hay dos formas de lograr que una función acceda a una estructura.

- Hacer la estructura *externa* respecto a la función, y
- Pasar la *dirección* de la estructura (lo cual es una variable simple) a la función, de manera que pueda operar sobre la estructura mediante su dirección.

En ambos casos, la función opera sobre la versión original de la estructura, no sobre una copia. En este sentido, las estructuras y arreglos se manejan en forma similar. En la sección C.15 se analiza el uso de variables externas. Aquí se estudian las implicaciones del paso de la dirección de una estructura.

Supóngase que se quiere pasar una estructura completa a una función *imprimevals()*, que imprime los valores de sus miembros. El siguiente código muestra cómo se pasa cierta dirección de estructura a la función. La forma de la estructura se define fuera del procedimiento principal, de modo que se pueden hacer referencias a la definición de manera conveniente donde sea necesario.

[†]Si un miembro de una estructura, como *res.emp.nombre*, es una variable simple, puede pasarse a una función tal como cualquier otra variable simple. El problema que se estudia en esta sección se aplica al paso de estructuras completas.

```

typedef struct {
    char num_id[8];          /* Definición de un           */
    short cantidad;          /* tipo de estructura. Cuando */
    char nombre[20];         /* REG_PARTE se usa después de una */
} REG_PARTE;                /* declaración, será           */
                            /* asignado espacio para una estructura */
                            /* de esta forma en particular. */

main()
{
    (REG_PARTE parte_camión;  /* parte_camión se declara con      */
                            /* la forma definida por           */
                            /* REG_PARTE                      */
    imprimevals (&parte_camión); /* Envía la dirección del registro */
                            /* parte_camión a imprimevals().   */
}

```

Si *imprimevals()* funciona correctamente sobre los miembros de *parte_camión*, debe conocer no sólo la dirección de *parte_camión*, sino también los tamaños y formas de sus miembros. En otras palabras, *imprimevals()* debe tener dentro de su definición una definición de una estructura equivalente a la *parte_camión*. Gracias al *typedef* de *REG_PARTE*, esto se logra fácilmente:

```

imprimevals (reg_ent)

    REG_PARTE *reg_ent;    /* reg_ent es un apuntador local   */
{                           /* a una estructura de la forma   */
                           /* definida por REG_PARTE.        */
    :
}

```

El siguiente problema es imaginarse cómo accederá *imprimevals()* a los miembros individuales de la estructura apuntada por *reg_ent*. Por ejemplo, supóngase que se quiere imprimir el valor del miembro *cantidad* de la estructura. Recuérdese que, mediante *reg_ent*, “lo apuntado por” se denota en C con **reg_ent*. Para denotar “el miembro *cantidad* de lo apuntado por *reg_ent*”, puede escribirse

(*reg_ent).cantidad



Como esto es una notación problemática y poco expresiva de aquello a lo que se hace referencia, los diseñadores de C han proporcionado un

operador que significa lo mismo pero que expresa mejor el significado. El operador `—>`:

`reg_ent —> cantidad` significa lo mismo que `(*reg_ent).cantidad`

Ahora se pondrá una proposición en `imprimevals()` para imprimir el valor del miembro `cantidad` de la estructura.

```
imprimevals(reg_ent)
{
    REG_PARTE *reg_ent;
    .
    printf ("cantidad = %d\n", reg_ent->cantidad);
}
```

`REG_PARTE`, `parte_camión` y `reg_ent` son tres nombres que parecen referirse a lo mismo, pero en realidad no es así. Conviene revisar:

<code>REG_PARTE</code>	El nombre de la <i>definición</i> de un tipo de datos que sucede ser una estructura. No se le asocia almacenamiento. No se refiere a datos reales. Simplemente indica cómo se verá un objeto de tipo <code>REG_PARTE</code> , si alguna vez se declara ese objeto.
<code>parte_camión</code>	El nombre de la estructura real que puede almacenar datos reales, y para la cual se asigna espacio real de memoria. Cuando <code>parte_camión</code> se declara en <code>main()</code> , se declara para que tenga la forma definida por <code>REG_PARTE</code> .
<code>reg_ent</code>	Una variable tipo apuntador que contiene la dirección de <code>parte_camión</code> . Es una variable apuntadora simple, así que puede usarse como un argumento de función. Cuando se hace la llamada de función <code>imprimevals(&parte_camión)</code> , la dirección <code>&parte_camión</code> se asigna a <code>reg_ent</code> .

Ejercicio 24. Complete la función `imprimevals()` para que imprima todas las partes de la estructura.

D

COMPARACION DE UNIDADES DE DISCO

Existen enormes diferencias entre los distintos tipos de unidades de disco en términos de la cantidad de datos que pueden guardar, el tiempo que les toma acceder a los datos, el costo total, el costo por bit y la inteligencia. Más aún, los dispositivos y medios de disco evolucionan tan rápido que las cifras sobre velocidad, capacidad e inteligencia que se aplican en cierta fecha pueden ser anacrónicas un mes después.

Se recordará que el tiempo de acceso está compuesto por el tiempo de desplazamiento, el retardo rotacional y el tiempo de transferencia.

Los *tiempos de desplazamiento* normalmente se describen de dos formas: *tiempo mínimo* y *tiempo medio*. Por lo regular, aunque no siempre, el tiempo mínimo de desplazamiento incluye el tiempo que le toma a la cabeza acelerar desde el reposo, trasladarse a una pista y detenerse. Algunas veces se da el tiempo pista a pista, con un valor independiente para el tiempo de detención de la cabeza. Se debe ser cuidadoso con tales valores, ya que sus significados no siempre se establecen con claridad.

El tiempo medio de desplazamiento es el tiempo medio que toma un desplazamiento cuando hay la misma probabilidad de que el sector deseado esté en un cilindro o en cualquier otro. En un ambiente de acceso completamente aleatorio, puede mostrarse que el número de cilindros que abarca un desplazamiento medio es de cerca de un tercio del número total de cilindros (Pechura y Schoeffler, 1983). Las estimaciones del tiempo medio de desplazamiento suelen basarse en este resultado.

Ciertas unidades de disco, llamadas *unidades de disco de cabeza fija*, no requieren tiempo de desplazamiento. Las unidades de disco de cabeza fija tiene una o más cabezas de lectura/escritura por pista, de tal forma que no es necesario trasladarlas de pista a pista. Las unidades de disco de cabeza fija son muy rápidas, pero también considerablemente más costosas que las de cabezas móviles.

Generalmente no existen diferencias significativas en el *retraso rotatorio* entre unidades de discos similares. La mayoría de las unidades de disco flexible giran a una velocidad de entre 300 y 600 rpm, y la mayoría de las unidades de disco duro giran a 3600 rpm. Por lo regular los discos flexibles no giran continuamente, de modo que el acceso intermitente del disco flexible puede implicar un retraso adicional de un segundo o más debido al arranque. En algunas circunstancias, las estrategias tales como la intercalación de sectores pueden mitigar los efectos del retraso rotatorio.

La razón de transferencia está restringida por dos elementos: velocidad de rotación y densidad de grabación. Puesto que las velocidades de rotación varían muy poco, las principales diferencias entre unidades de disco se deben a diferencias en la densidad de grabación. En años recientes se han logrado enormes mejoras en las densidades de grabación

TABLA D.1

Comparación de unidades de disco

	Disco flexible 5-1/4 pulg (IBM PC/AT)	Winchester 5-1/4 pulg (QUBIE PC20)	Removable pequeño (DEC RMO3)	Fijo, grande y por sectores (DEC RP07)	Fijo, grande y por bloques (IBM 3380)
Tiempos de desplazamiento					
Mínimo (mseg)	*	3	6	5	*
Promedio (mseg)	95	105	30	23	16
Retraso rotatorio (mseg)	100	8.5	8.3	8.3	8.3
Razón de transferencia (KB/seg)	23	484	1 200	2 200	3 000
Capacidades					
Bytes por pista	4 608	8 192	16 384	25 600	47 476
Pistas por cilindro	2	4	5	16	15
Cilindros por unidad	40	611	823	1 260	1 770
Megabytes por unidad	0.36	20	67	516	1 260

* No está disponible.

en discos de todos tipos. Las diferencias en las densidades de grabación se expresan por lo regular en términos del número de *pistas por superficie*, y el número de *bytes por pista*. Si los datos están organizados por sectores en un disco y se transfiere más de un sector al mismo tiempo, la razón de transferencia de datos efectiva depende también del método de intercalación empleado. Por supuesto, el efecto de la intercalación puede ser considerable, ya que con frecuencia los sectores lógicamente adyacentes están muy separados físicamente.

Aunque es muy posible que la mayoría de los valores de la tabla D.1 ya sean obsoletos cuando se publique este texto, sirven para proporcionar una idea básica de la magnitud y gama de características de desempeño de los discos. El hecho de que *estén* cambiando tan rápidamente también debe servir para destacar la importancia que tiene conocer las características de desempeño de la unidad de disco cuando se tiene que elegir entre diferentes.

Por supuesto, además de las diferencias cuantitativas entre las unidades, existen otras diferencias importantes. Por ejemplo, la unidad de disco IBM 3380 tiene muchas características incorporadas, entre ellas unos brazos actuadores separados que permiten efectuar dos accesos en forma simultánea. También tiene buffers locales grandes y una buena inteligencia local, que permite optimizar muchas operaciones que tienen que ser supervisadas por el computador central en unidades de disco menos complejas.

Bohl [1981] y Hanson [1982] proporcionan un estudio exhaustivo de las unidades de disco grandes y complejas. Estas unidades evolucionan en forma mucho más paulatina que las unidades relativamente baratas empleadas con los microcomputadores.

BIBLIOGRAFIA

- Baase, S., *Computer Algorithms: Introduction to Design and Analysis*, Reading, Mass., Addison-Wesley, 1978.
- Batory, D.S., "B⁺ trees and indexed sequential files: A performance comparison", *ACM SIGMOD*, 1981, págs. 30-39.
- Bayer, R. y E. McCreight, "Organization and maintenance of large ordered indexes", *Acta informatica* 1, núm. 3, 1972, págs. 173-189.
- Bayer, R. y K. Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems* 2, núm. 1, marzo, 1977, págs. 11-26.
- Bentley, J., "Programming pearls: A spelling checker", *Communications of the ACM* 28, núm. 5, mayo, 1985, págs. 456-462.
- Bohl, M., *Introduction to IBM Direct Access Storage Devices*. Chicago: Science Research Associates, Inc., 1981.
- Borland, *Turbo Toolbox Reference Manual*, Scott's Valley, Calif., Borland International, Inc., 1984.
- Borne, S.R. *The United System*, Reading, Mass., Addison-Wesley, 1984.
- Bradley, J., *File and Data Base Techniques*, Nueva York, Holt, Rinehart and Winston, 1982.
- Chaney, R. y B. Johnson, "Maximizing hard-disk performance", *Byte* 9, núm. 5, mayo 1984, págs. 307-304.
- Chang, C.C., "The study of and ordered minimal perfect hashing scheme", *Communications of the ACM* 27, núm. 4, abril, 1984, págs. 384-387.

- Chichelli, R.J., "Minimal perfect hash functions made simple", *Communications of the ACM* 23, núm. 1, enero, 1980, págs. 17-19.
- Claybrook, B. G., *File Management Techniques*, Nueva York, John Wiley & Sons, 1983.
- Comer, D., "The ubiquitous B-tree", *ACM Computing Surveys* 11, núm. 2, junio, 1979, págs. 121-137.
- Cooper, D., *Standard Pascal User Reference Manual*, Nueva York, W. W. Norton & Co., 1983.
- Crotzer, A.D., "Efficacy of B-trees in an information storage and retrieval environment", tesis de maestría, inédita, Oklahoma State University, 1975.
- Davis, W.S., "Empirical behavior of B-trees", tesis de maestría, inédita, Oklahoma State University, 1974.
- Deitel, H., *Introducción a los sistemas operativos*, Wilmington, Delaware, Addison-Wesley Iberoamericana, 1987.
- Digital, *Introduction to VAX-11 Record Management Services*, orden núm. AA-DO24A-TE, Digital Equipment Corporations, 1978.
- Digital, *Peripherals Handbook*, Digital Equipment Corporation, 1981.
- Digital, *RMS-11 User's Guide*, Digital Equipment Corporation, 1979.
- Digital, *VAX-11 SORT/MERGE User's Guide*, Digital Equipment Corporation, 1984.
- Digital, *VAX Software Handbook*, Digital Equipment Corporation, 1982.
- Dodds, D.J., "Pracniqe: Reducing dictionary size by using a hashing technique", *Communications of the ACM* 25, núm. 6, junio, 1982, págs. 368-370.
- Dwyer, B., "One more time-how to update a master file", *Communications of the ACM* 24, núm. 1, enero, 1981, págs. 3-8.
- Fagin, R.J., Nievergelt, N. Pippenger y H. R. Strong, "Extendible hashing—a fast access method for dynamic files", *ACM Transactions on Database Systems* 4, núm. 3, septiembre, 1979, págs. 315-344.
- Faloutsos, C., "Access methods for text", *ACM Computing Surveys* 17, núm. 1, marzo, 1985, págs. 49-74.
- Flores, I., *Peripheral Devices*, Englewood Cliffs, N. J., Prentice-Hall, 1973.
- Gonnet, G. H., *Handbook of Algorithms and Data Structures*, Reading, Mass., Addison-Wesley, 1984.
- Hanson, O., *Design of Computer Data Files*, Rockville, Md., Computer Science Press, 1982.
- Held, G. y M. Stonebraker, "B-trees reexamined", *Communications of the ACM* 21, núm. 2, febrero, 1978, págs. 139-143.

- Hoare, C.A.R., "The emperor's old clothes", discurso pronunciado al recibir el premio Turing. *Communications of the ACM* 24, núm. 2, febrero, 1981, págs. 75-83.
- IBM, *DFSORT General Information*, orden de IBM núm. GC33-4033-11.
- IBM, *OS/VS Virtual Storage Access Method, VSAM Planning Guide*, orden de IBM núm. GC26-3799.
- Jensen, K. y N. Wirth, *Pascal User Manual and Report*, 2^a. ed. Springer Verlag, 1974.
- Keehn, D.G. y J.O. Lacy, "VSAM data set design parameters", *IBM Systems Journal* 13, núm. 3, 1974, págs. 186-212.
- Kelley, A., e I. Pohl, *El lenguaje C. Introducción a la programación*, Wilmington, Delaware, Addison-Wesley Iberoamericana, 1987.
- Kernighan, B. y R. Pike, *The UNIX programming Environment*, Englewood Cliffs, N.J., Prentice Hall, 1984.
- Kernighan, B. y D. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J., Prentice-Hall, 1978.
- Knuth, D., *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 2^a. ed., Reading, Mass., Addison-Wesley, 1973a.
- Knuth, D., *The Art of Computer Programming*, vol. 3, *Searching and Sorting*, Reading, Mass., Addison-Wesley, 1973b.
- Lang, S.D., J.R. Driscoll y J.H. Jou, "Batch insertion for tree structured file organizations—improving differential database representation", CS-TR-85, Department of Computer Science, University of Central Florida, Orlando, Flor.
- Levy, M.R., "Modularity and the sequential file opdate problem", *Communications of the ACM* 25, núm. 6, junio, 1982, págs. 362-367.
- Loomis, M., *Data Management and File Processing*, Englewoods Cliffs, N.J., Prentice-Hall, 1983.
- Lorin, H., *Sorting and Sort Systems*. Reading, Mass., Addison-Wesley, 1975.
- Madnick, S.E. y J.J. Donovan, *Operating Systems*, Englewood Cliffs, N.J., Prentice-Hall, 1974.
- McCreight, E., "Paginations of B^{*} trees with variable length records", *Communications of the ACM* 20, núm. 9, septiembre, 1977, págs. 670-674.
- McKusick, M.K., W.M. Joy, S.J. Leffer y R.S. Fabry, "A fast file system for UNIX", *ACM Transactions on Computer Systems* 2, núm. 3, agosto, 1984, págs. 181-197.
- Maurer, W.D. y T.G. Lewis, "Hash table methods", *ACM Computing Surveys* 7, núm. 1, marzo, 1975, págs. 5-19.

- Microsoft, Inc., *Disk Operating System. Version 2.00*, IBM Personal Computer Languaje Series, IBM, 1983.
- Murayama, K. y S.E. Smith, "Analysis of design alternatives for virtual memory indexes", *Communications of the ACM* 20, núm. 4, abril, 1977, págs. 245-254.
- Nievergelt, J., H. Hinterberger y K. Sevcik, "The grid file: an adaptive symmetric, multikey structure", *ACM Transactions on Database Systems* 9, núm. 1, marzo, 1984, págs. 38-71.
- Ouskel, M. y P. Scheuermann, "Multidimensional B-trees: Analysis of dynamic behavior", *BIT* 21, 1981, págs. 401-418.
- Pechura, M.A. y J.D. Schoeffler, "Estimating file access of floppy disk", *Communications of the ACM* 26, núm. 10, octubre, 1983, págs. 754-763.
- Peterson, J.L. y A. Silbershatz, *Operating Systems Concepts*, 2^a ed., Reading, Mass., Addison-Wesley, 1985.
- Peterson, W.W., "Addressing for random access storage", *IBM Journal of Research and Development* 1, núm. 2, 1957, págs. 130-146.
- Pollack, S. y T. Sterling, *A Guide to Structured Programming and PL/I*, 3^a ed., Nueva York, Holt Rinehart and Winston, 1980.
- Ritchie, B. y K. Thompson. "The UNIX time-sharing system", *Communications of the ACM* 17, núm. 7, julio, 1974, págs. 365-375.
- Robinson, J.T., "The K-d B-tree: A search structure for large multidimensional dynamic indexes", *ACM SIGMOD 1981 International Conference on Management of Data*. 29 de abril—1º de mayo, 1981.
- Rosenberg, A.L. y L. Snyder, "Time and space optimality in B-trees", *ACM Transactions on Database Systems* 6, núm. 1, marzo, 1981, págs. 174-183.
- Sager, T.J., "A polynomial time generator for minimal perfect hash functions", *Communications of the ACM* 28, núm. 5, mayo, 1985, págs. 523-532.
- Salton, G. y M. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.
- Scholl, M., "New file organizations based on dynamic hashing", *ACM Transactions on Database Systems* 6, núm. 1, marzo, 1981, págs. 194-211.
- Severance, D.G., "Identifier search mechanisms: A survey and generalized model", *ACM Computing Surveys* 6, núm. 3, septiembre, 1974, págs. 175-194.
- Snyder, L., "On B-trees reexamined", *Communications of the ACM* 21, núm. 7, julio, 1978, pág. 594.
- Spector, A. y D. Gifford, "Case study: The space shuttle primary computer system", *Communications of the ACM* 27, núm. 9, septiembre, 1984, págs. 872-900.

- Standish, T.A. *Data Structure Techniques*, Reading, Mass., Addison-Wesley, 1980.
- Sussenguth, E.H., "The use of tree structures for processing files", *Communications of the ACM* 6, núm. 5, mayo, 1963, págs. 272-279.
- Sweet, F., "Keyfield design", *Datamation*, octubre 1, 1985, págs. 119-120.
- Teory, T.J. y J.P. Fry, *Desing of Database Structures*, Englewood Cliffs, N.J., Prentice-Hall, 1982.
- The Joint ANSI/IEEE Pascal Standards Committee, "Pascal: Forward to the candidate extension library", *SIGPLAN Notices* 19, núm. 7, julio, 1984, págs. 28-44.
- Tremblay, J.P. y P.G. Sorenson, *An Introduction to Data Structures with Applications*, Nueva York, McGraw-Hill, 1984.
- Ullman, J., *Principles of Database Systems*, 2^a ed., Rockville, Md., Computer Science Press, 1980.
- VanDoren, J., "Some empirical results on generalized AVL trees", *Proceedings of the NSF-CBMS Regional Research Conference on Automatic Information Organization and Retrieval*, University of Missouri at Columbia, julio, 1973, págs. 46-62.
- VanDoren, J. y J. Gray., "An algorithm for maintaining dynamic AVL trees", En *Information Systems, COINS IV*, Nueva York, Plenum Press, 1974, págs. 161-180.
- Wagner, R. E., "Indexing design considerations", *IBM Systems Journal* 12, núm. 4, 1973, págs. 351-367.
- Webster, R. E., "B⁺ trees", tesis de maestría, inédita, Oklahoma State University, 1980.
- Wiederhold, G., *Database Design*, 2^a ed., Nueva York, McGraw-Hill, 1983.
- Wirth, N., "An assessment of the programming language Pascal", *IEEE Transactions on Software Engineering SE-1*, núm. 2, junio, 1975).
- Yao, A. Chi-Chih, "On random 2-3 trees", *Acta Informatica* 9, núm. 2, 1978, págs. 159-170.
- Zoellick, B., "CD-ROM software development", *Byte* 11, núm. 5, mayo, 1986, págs. 173-188.

INDICE DE MATERIAS

- acceso a archivos, 119-121
 - definición, 124
- acceso directo, 113-115
 - definición, 123
- acceso directo a memoria (DMA), 39
 - definición de, 79
- acceso secuencial, 123
- acceso secuencial indizado, 398-426
 - definición, 432
 - e ISAM, 436
 - y árboles B de orden variable, 416-415, 427-428
 - y árboles B⁺, 424-429
 - y árboles B⁺ de prefijos simples, 409-415, 420-424, 427, 429
 - y conjuntos de índices, 404-409
 - y conjuntos de secuencias, 399-404
 - y VSAM, 349, 436, 440
- actualización. *Véase* adición de registros; eliminación de registros
- adición de registros
 - de registros de longitud variable, 174-177
 - en árboles B, 337-353, 362-363
 - en árboles B⁺, 410-415
 - en conjuntos de secuencias, 400-402
 - y archivos de índices, 235-236, 240-243
 - y marcas de inutilización, 481-482
 - y tiempo de clasificación, 204-205
- administrador de archivos, 45
- definición, 79
- y transmisión de datos, 67-69
- reglas de alcance, 556
- algoritmo de dispersión perfecta, 446
 - definición, 497
- almacenamiento redundante en línea, 64
- almacenamiento primario, 2
 - Véase también* memoria de acceso aleatorio (RAM)
- almacenamiento secundario
 - acceso al, 3-4
 - caché para disco, 64
 - cinta magnética, 56-62
 - comparación del, 65-66
 - costo del, 2-3
 - discos, 39-56
 - discos en RAM, 64
 - respaldo y almacenamiento de archivos, 63-64
 - tarjetas perforadas, 62
- Véase también* discos; cinta magnética
- árbol AVL, 329-332
 - definición, 375
- árbol B, 322-324, 337-372
 - búsqueda en, 344-345
 - y colocación de información, 370-371
 - comparación del, 424, 429
 - definición, 354-355, 375-376
 - división y promoción en, 337-342, 346-353

- eliminación en, 357-362
- estructura del, 337-342
- inserción en, 346-350, 362
- invención del, 322-324
- nodos y apuntadores en, 337-338, 354
- orden del, 337-338, 354, 375
- de orden variable, 365-369
- profundidad de, 355-357
- redistribución en, 358-361
- y registros de longitud variable, 371-372
- virtual, 365-369
- árbol B de orden variable, 416-419
 - definición, 433
- árbol B virtual, 365-369
 - definición, 376
- árbol B', 424-426
 - comparación del, 424-429
 - definición, 432
 - Véase también árbol B' de prefijos simples
- árbol B' de prefijos simples, 405-416
 - carga del, 420-424
 - comparación del 424-429
 - definición, 433
 - inserción y eliminación en un, 411-415
- árbol B*, 363-365, 409n
 - definición, 375
- árbol de búsqueda, 286-287, 475-476, 482-483
 - Véase también árbol AVL; árbol binario de búsqueda; árbol B'; árbol B"; árbol B
- árbol binario de búsqueda, 287, 325-329
 - construcción descendente del, 335-337
 - y balance, 328-329
 - nodos e hijos en, 325-328
 - paginación, 332-337, 342-344
 - profundidad del, 286-287
- árbol de selección, 286-287
 - definición, 312
 - Véase también árbol binario de búsqueda
- árbol balanceado en altura, 329-331
 - definición, 375
- árbol binario paginado, 332-337
 - estructura del, 342-344
- árbol de torneo, 287
- archivo
 - apertura y creación de un, 13-17
 - caracteres inesperados en, 27-28
 - cierra de, 17
 - control del acceso a, 16
 - definición, 5-6, 9
 - detección del fin del, 22
 - escritura de un, 19-22
 - estructura de un, 6-7, 9
 - físico y lógico, 12-13, 29
 - lectura de un, 18-19
 - modificación de un, 164-180
 - protección, 17
 - inicio de un archivo para dispersión, 478
 - archivo físico, 12-13
 - definición, 29
 - archivo grande
 - indización, 237-239
 - clasificación por intercalación, 287-303
 - archivo lógico, 12-13
 - archivo con entradas secuenciales, 229-237
 - archivo de índices para, 229-235
 - definición, 232, 257
 - y listas ligadas, 252
 - arreglo de llaves, 196-200
 - arreglo de índices, 200
 - caracteres ASCII, 100, 172
 - y el vaciado hexadecimal, 102-106
 - tabla de, 509
 - proposición *asigna()*, 13
- Bayer, R.
 - "Organization and Maintenance of Large Ordered Indexes", 323, 353-354, 362-363
- bit de eliminación, 188
- bloque, 111-13
 - definición, 79, 123
 - en cinta, 58
 - en conjuntos de secuencias, 399-404
 - tamaño de, 403-404, 415-416
 - Véase también compartimientos
 - manejo de registros en bloques. Véase bloque
- bpi (bits por pulgada), 80
- programa *abes.c*, 384-386
- archivo de encabezado *ab.h*, 382
- programa *utilab.c*, 386-388
- programa *utilab.prc*, 392-395
- buffer de E/S, 69-70
- buffer de página, 366
- buffers, manejo de, 74-77, 100
 - definición, 30
 - en un depósito común, 76
 - y árboles B virtuales, 365-369
 - y cuello de botella con un buffer, 74
 - y manejo doble de buffers, 75-76
 - y modo de movimiento y modo de direcciones, 76
 - y porciones de intercalación, 289-290
 - y selección por reemplazo, 299-302
- buscar. Véase búsqueda binaria; búsqueda secuencial
- SEEK()*, 23
 - definición, 30
- función *busca()*, 344
- localización, 22-26
 - en C, 23-25
 - en Pascal, 25-26

- búsqueda binaria, 193-195, 203
 - y archivos de índices, 228-229, 238
 - comparada con la búsqueda secuencial, 195
 - definición, 214
 - limitaciones de la, 203-205,
 - y clasificación en RAM, 196-205
- búsqueda_binaria()*, función, 194
- procedimiento *búsqueda_en_secundario()*, 241
- Búsqueda secuencial, 109-111
 - y manejo de registros en bloques, 111-112
 - comparada con la búsqueda binaria, 195
 - definición, 124
 - evaluación de la, 109-111
- desplazamientos, número de
 - para procesamiento secuencial coordinado, 289-290, 293, 300-301
 - para árboles binarios paginados, 334
- C (lenguaje), 514-569
 - y accesibilidad de variables, 552-553, 555-556
 - y bibliotecas, 515-516, 556-557
 - y apuntadores, 559-569
 - y arreglos, 529-530, 561-563, 565-567
 - y bloques de proposiciones, 541-549
 - y cadenas, 530-531
 - características de, 514-515
 - y clases de almacenamiento, 552-556
 - comparado con Pascal, 101, 114-115
 - y compilación condicional, 559
 - y conjuntos de herramientas, 518
 - y constantes hexadecimales y octales, 527
 - y conversiones forzosas (*casts*), 540-541
 - creación, compilación y ejecución de programas en, 518-520
 - orden # *define*, 557-558
 - y enteros y caracteres, 523-526
 - y estructuras, 531-534, 567-569
 - funciones en, 549-557
 - orden # *include*, 516-517, 559
 - función *main()*, 521-522
 - y números de punto flotante, 526-527
 - operadores en, 535-541
 - función *printf()*, 527-528
 - programas, 383-384, 386-388, 131-145, 218-222
 - y reglas de alcance, 555-557
 - función *scanf()*, 534-535
 - tipos de datos en, 514-515, 522-527
 - y uniones, 523-524
 - almacenamiento caché para disco, 64
 - definición, 79
 - campo
 - definición, 92n, 124
 - delimitado, 93-94
 - de longitud fija, 92-93
 - de longitud variable, 93-94
 - organización del, 116-117
 - campo apuntador, 168
 - campo llave, 257
 - campo de conteo de bytes, 124
 - campo de referencia, 239-242, 253-255
 - definición, 257
 - anulación de la eliminación, 166
 - caracteres de fin de línea, 27
 - carga en dos pasos, 486
 - CLOSE(), 18
 - definición, 30
 - ciclo de sincronización, 266-268
 - definición, 313
 - cilindro, 41
 - definición, 80
 - cinta magnética, 56-62
 - aplicaciones de la, 62
 - clasificación en, 289, 303-309
 - organización de la, 56-58
 - requerimientos de longitud de, 58-60
 - tiempo de transmisión de datos en, 60-61
 - tiempos de arranque y detención, 87
 - llave, 106-109
 - forma canónica de la, 106-107, 197, 231, 241
 - definición, 124
 - duplicada, 240
 - en archivos de índices, 230-239
 - primaria, 124, 230-239
 - secundaria, 239-240
 - llave de búsqueda. Véase llave
 - llave primaria, 124
 - en archivos de índices, 230-239
 - Véase también llave
 - llave secundaria, 239-240
 - Unión de huecos, 180, 189
 - definición, 186
 - colisión, 445-447
 - definición, 495
 - predicción, 459-460, 461-465
 - resolución, 465-470, 483-489
 - Comer, Douglas
 - "The Ubiquitous B-tree", 323-324
 - compactación, 165-166
 - definición, 185
 - compactación del almacenamiento, 165-166
 - compartimiento, 447, 470-479
 - definición, 495
 - y densidad de empaquetamiento, 471
 - estructura del, 477-478
 - tamaño del, 471-475

- concatenación
 en árboles B, 361
 en árboles B⁺, 413-415
 en conjuntos de secuencias, 400-402
 definición, 376
- procedimiento *correspondencia*, 267
 rutina de entrada para el, 268
- conjuntos de índices, 407-410
 definición, 433
Véase también árbol B⁺
- conjunto de secuencias, 399
 adición de un índice simple al, 405-406
 bloques para el, 399-404, 415-416
 definición, 433
Véase también árbol B⁺
- controlador, 80
- controlador de disco, 73-74
- costo de almacenamiento, 2-3
- CREATE(), 17
 definición, 30
- programa *hazllave.c*, 140
- cúmulo, 44-45, 403-404, 475
 definición, 80
 en VAX SAR, 84
 y fragmentación, 47
- delimitador, 94,100
 definición, 124
- retraso por rotación, 54
 definición, 82
- densidad de empaquetamiento, 460-465
 y compartimientos, 471
 definición, 495
 y longitud media de búsqueda, 469
- densidad de grabado nominal, 80
- densidad de grabado efectiva, 60
 definición, 80
- saturación progresiva, 463-470
 definición, 497
 saturación progresiva de conteo de llave, 500
 encadenado, 482-487
 y fin de archivo, 465-466
 y registros faltantes, 466
- distancia en bytes, 114, 174
 definición, 30
- indirección, 196-197
 definición, 214
- direccionamiento abierto. *Véase* saturación progresiva
- disco fijo, 16
 definición, 80
- disco láser, 3
- disco óptico, 64
- disco en RAM, 64
 definición, 80
- discos, 39-56
- 672/765
- capacidad de almacenamiento de los, 42-43
 y compartimientos, 470
- costo del almacenamiento en, 3
- y sobrecarga por datos usados para control, 50-52
- organización de los, 39-42
- organización por bloques, 48-50
 y sectores, 43-48, 403-404
 tiempo de acceso para, 3-4, 52-56
- dispersión, 440-491
 algoritmo para, 447-451
 algoritmo de, perfecta, 446
 características de las, 443-445
 y colisiones, 445-447, 483-489
 y compartimientos, 470-478
 definición, 495-496
 y densidad de empaquetamiento, 460-465
 y saturación progresiva, 465-470
 y diccionarios ortográficos, 502-503, 505
 y distribución de registros, 451-460
 y dispersión con revisión, 503
 y eliminaciones, 479-483
 y tamaño de archivo, 476
 asignación de valores iniciales a un archivo y carga del archivo, 478
- función *dispersión()*, 451
- dispersión extensible, 488-489, 505
 definición, 496
- dispersión indizada, 488
 definición, 496
- dispersión mínima, 496
- dispersión virtual. *Véase* dispersión extensible
- dispositivo de acceso secuencial, 80
- dispositivo de almacenamiento de acceso directo (DAAD), 39
 definición, 81
- distribución de registros. *Véase* redistribución
- distribución uniforme, 451
 definición, 498
- procedimiento *divide()*, 351, 362
- división, 337-342, 350-351
 en árboles B⁺, 413-415
 de bloques, 400-402
 definición, 376
- división por números primos, 450, 454
 definición, 496
- dispersión doble, 483
 definición, 496
- dirección base, 444
 definición, 495
- porción, 288-291
 definición, 314
- eliminación de registros, 165-185
 en árboles B, 357-361

- en árboles B+, 410-415
- en archivos con dispersión, 479-483
- y archivos de índices, 236-237, 241-242
- marcas, 165, 478
- de registros de longitud fija, 171-173, 177-178
- de registros de longitud variable, 173-183
- función *elimina_reg* (RRN), 177
- función *elim_reg* (RRN), 172
- programa *encuentra*, 110
 - encuentra.c*, 138-139
 - encuentra.pas*, 154-156
- función *encuentra_sep()* (C), 408
- procedimiento *encuentra_sep()* (Pascal), 408
- enlace, 239-241, 253-255
 - definición, 257
 - Véase también registros fijos
- entero binario, 100, 101-103, 172
- error por desbordamiento (saturación), 449
 - área separada para, 486-487
 - en conjuntos de secuencias, 400
 - definición, 497
 - y eliminaciones, 479
 - Véase también saturación progresiva
- programa *escribe_sec*, 91
 - escribe_sec.c*, 132-133
 - escribe_sec.pas*, 146-148
- programa, *escribe_reg*, 100
 - escribe_reg.c*, 134-135
 - escribe_reg.pas*, 150-152
- WRITE(), 19-20, 67
 - definición, 31
- estrategia de colocación, 181-183
 - definición, 189
- balance
 - de árboles AVL, 329-331
 - de árboles B, 342
 - de árboles binarios de búsqueda, 328-329
- sobrecarga por datos usados para control, 50-52
- hueco entre bloques, 82
- estructura de datos, 6-7
- archivos multilistas, 262
- extensión, 46
 - definición, 81
- factor de bloque, 49-59
 - definición, 81
- factor de intercalación, 44
 - definición, 81
- archivo de encabezado *arches.h*, 131-132
- fin de archivo
 - caracteres de, 27-28
 - y saturación progresiva, 465-467
 - deteccción, 22
 - en procedimiento de correspondencia, 268
 - en procedimiento de intercalación, 271-272
- secuencia
 - de bytes, 90-92, 125
 - de campos, 95-96
- forma canónica, 108-109, 197
 - en archivos de índices, 231, 242
 - definición, 124
 - Véase también llave
- dar formato, 80
- fragmentación, 46-48
 - definición, 81, 186
 - y registros de longitud fija, 177-178
 - y registros de longitud variable, 178-179
- fragmentación externa, 180
 - definición, 186
 - Véase también fragmentación
- fragmentación interna, 47, 178-180
 - definición, 186
 - Véase también fragmentación
- intercalación, 270-272
 - cálculo de comparaciones en la, 287, 291-293
 - cálculo de desplazamientos en la, 289, 293, 300-301
 - y configuración de disco, 302
 - definición, 313
 - y porciones, 288-291
 - en cinta, 289, 303-309
 - y paquetes de clasificación por intercalación, 309
 - para clasificar archivos grandes, 287-303
 - de varios pasos, 291-294
 - múltiple, 284-287
 - resumen, 302-303
 - y selección por reemplazo, 294-302
- procedimiento *intercala*, 270
 - rutina de entrada para el, 271
- intercalación en cascada, 307
- intercalación balanceada, 304-306
 - definición, 313
- intercalación en varias frases, 306-309
 - definición, 313
- intercalación de varios pasos, 291-294
 - definición, 313-314
- intercalación polifásica, 307
 - definición, 314
- intercalación múltiple, 284-287
- intercalación de K formas, 284-287
 - ciclos para la, 285-286
 - Véase también intercalación
- hermano, 361
- programa *herramientas.prc*, 509-513
- indicador de longitud, 94-95, 100-107
- índice, 99-100, 209-211, 228-255

- adicción al conjuntos de secuencias de un, 404-406
y árboles B *k-d*, 381
y árboles binarios de búsqueda, 325
y archivos grid, 381
de archivos con entradas secuenciales, 229-237, 251
carga del, 234
definición, 237
demasiado grande para almacenarse en memoria, 237-238
y modificación de registros, 235-237, 240-242
paginados, 323-325
reescritura del archivo de, de la memoria, 234-235
secundario, 239-253
simple, 228-255
Véase también árbol binario de búsqueda; árbol B
índice paginado, 253, 261, 323-325
definición, 376
índice secundario, 239-253
campos de referencia para el, 241-243, 253-255
y llaves duplicadas, 241
e índices selectivos, 253
y listas invertidas, 246-253
y modificación de registros, 241-243
y extracción de información mediante combinaciones de llaves, 244-246
índice selectivo, 253
definición, 257
índice simple, 257
Véase también índice
inserción. Véase adición de registros
función *insertar()*, 350
inserta.c, 383-384
inserta.prc, 391-392
intersección. Véase procedimiento *correspondencia*
- Kernighan, Brian
The C Programming Language, 514, 559-560
- Knuth, Donald, 297-299, 303, 354, 363-365
- marca de inutilización, 480-482
definición, 497
- programa *leesec*, 95
leesec.c, 133-134
leesec.pas, 149-150
- READ()*, 19
definición, 30
- programa *leereg*, 107
leereg.c, 136
leereg.pas, 152-153
- programa de libro mayor, 275-283
programa para él, 282
rutinas de entrada para él, 279, 280
programa *mayor*, 282
rutinas de entrada para él, 279, 280
- programa LISTA
en C, 20
en Pascal, 21
- lista de disponibles, 168-171
definición, 186
y eliminación de registros, 171-177
ordenamiento de la, 181
- lista invertida, 246-253
definición, 258
- lista ligada, 249-253
definición, 186
para eliminación de registros de longitud fija, 164-171
para eliminación de registros de longitud variable, 173-175
- localidad, 252
definición, 258
- longitud de búsquedas, 467-470
media, 475-476, 482, 494
- longitud media de búsqueda, 467-470, 475-476
definición, 497
y marcas de inutilización, 482-483
- función *lseek()*, 23-24
- procedimiento *manejador*, 353
manejador.c, 383
manejador.pas, 389-390
- marco, 57
definición, 81
- McCreight, E., 372
“Organization and Maintenance of Large Ordered Indexes”, 323, 253-254, 363
- mejor ajuste, 181
definición, 186
- distribución mejor que la aleatoria, 451-453
definición, 497
- memoria de acceso aleatorio (RAM), 2
definición, 9
clasificación en, 196-206, 287-288, 294-301
y tamaño de bloque, 403-404
y tamaño de índice, 405-406
tiempo de acceso a la, 4, 7
- método del medio cuadrado, 454
definición, 497
- modo de acceso, 16
definición, 30

- nodo
 - en árboles B, 337
 - en árboles binarios de búsqueda, 325-328
 - Véase también* nodo hoja
- nodo hoja
 - de árboles B, 338, 354, 376
 - de árboles binarios de búsqueda, 327
- número relativo de bloque (NRB), 417
- número relativo de registro (NRR), 113
 - en un archivo con dispersión, 476-477
 - y búsqueda binaria, 200, 203-204
 - definición, 124
 - en una lista ligada, 250-251
 - y apilamiento, 169
- función, *toma_disp()*, 179
- programa *toma.prc*, 153-154
- programa *tomarc.c*, 137-138
- OPEN(), 14
 - definición, 26
- operación *correspondencia*, 246, 265-269
 - definición, 313
- operaciones booleanas, 245-246
- operación secuencial coordinada, 264-309
 - aplicación de libro mayor, 275-283
 - y cinta, 290, 303-308
 - y correspondencia, 265-269
 - definición, 314
 - e intercalación, 270-272, 284-294
 - resumen, 272-274
 - y selección por reemplazo, 294-297
- orden del árbol B, 337-338, 353
 - definición, 375-376
- programa *clasifllave*, 206
- clasificación en memoria RAM, 196-206, 287, 294-301
 - algoritmo para, 201
 - Véase también* búsqueda binaria; intercalación; programa *clasifram*
- clasificación por llave, (*clasifllave*) 205-211
 - definición, 214
- clasificación por marca. *Véase* clasificación por llave (*clasifllave*)
- programa *clasifram*, 202
 - clasifram.c*, 218-222
 - clasifram.pas*, 222-226
- Quicksort, 6, 300
- función *clasif_shell()* 202
- organización de archivos, 119-121
 - definición, 124
- organización por bloques,
 - definición, 81
 - en unidad de disco IBM 3350, 84
- organización por sectores, 43-48
- y cúmulos, 44-46, 404, 475
- disposición física, 43
- definición, 81
- y extensiones, 46-47
- y fragmentación, 46-47
- en unidad de disco digital RM05, 86
- Pareto, principio de, 490-491
- paridad, 57-58
 - definición, 82
- desglosar y sumar, 448-449, 453
 - definición, 495
- Pascal
 - comparado con C, 101, 114-115
 - programas en, 146-162, 222-226, 389-395, 509, 523
- paquete de discos, 41, 63
 - definición, 82
- peor ajuste, 182-183
 - definición, 186
- pila, 168-171
 - definición, 186
- pista, 404, 436
 - definición, 41, 82
- plato, 41
 - definición, 82
- Poisson, distribución de, 455-460
 - definición, 496
- prefijo, 409-410, 424-426
 - Véase también* árbol B⁺ de prefijos simples
- primer ajuste, 181
 - definición, 186
- procesador de E/S, 71-73
 - definición, 82
- procesamiento en saltos secuenciales, 128
- programa *actualiza.c*, 141-145
- programa *actualiza.pas*, 156-162
- programa AGREGA
 - en C, 25
 - en Pascal, 34-35
- promoción, 339-342, 346-353
 - en árboles B⁺, 413-415
 - definición 376
- modo de protección, 16-17
- procedimiento *recupera_registro()*, 233
- recursión, 344
- redispersión, 488-501
- redistribución
 - en árboles B, 361-363
 - en árboles B⁺, 414-415
 - en archivos con dispersión, 451-460
 - en conjuntos de secuencias, 400-402
 - definición, 376
- procedimiento *reescribe_índice()*, 234

- registro, 96-100
 - definición, 92n, 96, 124
 - delimitado, 95, 100
 - diseño del, 115-118
 - lectura del, 106
 - extracción de, por llave, 106-109
 - Véase también registro de longitud fija;
registro de longitud variable
- registro de encabezado, 119, 171
 - definición, 125
- registro fijo, 211
 - definición, 215
 - Véase también enlace
- registro de longitud fija, 97, 115
 - definición, 125
 - eliminación de un, 167-173, 177-179
 - Véase también registro
- registro de longitud variable, 93-94, 98-102
 - y árboles B, 371-372
 - y archivos de índices, 229-239
 - definición, 125
 - eliminación de un, 173-177
 - Véase también registro
- registro vacío. Véase eliminación de registros
- regla empírica del 80/20, 490-491
 - definición, 497
- proposición *reset()*, 16
- revisión de secuencia, 272-273
 - definición, 314
- proposición *rewrite()*, 16
- Ritchie, Dennis
 - The C Programming Language*, 514, 559-560
- función *toma_disp()*, 176-177
- sector, 40-42
 - definición, 82
- selección por reemplazo, 294-302
 - costo de la, 299-302
 - definición, 314
 - longitud de las porciones para, 297-299
- separador, 406-409
 - definición, 433
- separador más corto, 408, 416
 - definición, 433
- Shell, clasificación de, 201-203
 - definición, 214
- sinónimo, 445-446
 - definición, 497-498
- sistema de almacenamiento masivo, 63
 - definición, 82
- verificación lineal. Véase saturación progresiva
- programa *caddat.prc*, 162
- programa *funscads.c* 140-141
- cadena, 509-513
- sub-bloque, 50
 - definición, 82
- sub-bloque de llave, 50
 - definición, 82
- sub-bloque de conteo, 50
 - definición, 82
- reemplazo LRU, 261, 366-368
- reemplazo
 - según la altura de la página, 368-369
 - con la estrategia LRU, 368-369
- tabla de asignación de archivos (FAT), 45
 - definición, 82
- tablas de dispersión, 488
- tiempo de acceso, 3-5, 52-56
 - cálculo del, 54-55
- tiempo de desplazamiento, 52-53
 - cálculo del, 53
 - definición, 83
- tiempo de transferencia, 54
 - definición, 83
- transformación de la base, 454
- transmisión de datos, 65-74
 - y administrador de archivos, 67-69
 - y buffer de E/S, 69-70
 - y controlador de disco, 73-74
 - y procesador de E/S, 71-72
- unidad de cinta de grabado continuo (*streamers*), 63
 - definición, 83
- unidades de disco, 39, 570-572
- programas de servicio de clasificación por intercalación, 307-309
- vaciado hexadecimal, 102-106, 118
- valor de fin de lista, 167
- procedimiento, *inicializar()*, 269
- VALOR_ALTO, 272
 - definición, 315
- VALOR_BAJO, 270
 - definición, 315
- VanDoren, James, 318
- tasa de transmisión nominal, 83
- tasa de transmisión efectiva, 83
- volatilidad, 164
- VSAM, 436
- Webster, Robert, 367, 368

VOCABULARIO TECNICO BILINGÜE*

accessibility of variables
accesibilidad de variables

access mode
modo de acceso

access time
tiempo de acceso

APPEND program
programa AGREGA

arrays
arreglos

ASCII characters
caracteres ASCII

assign() statement
proposición *asigna()*

avail list
lista de disponibles

average search length
longitud media de búsqueda

AVL tree
árbol AVL

backup and archival
respaldo y almacenamiento

balance
balance

balanced merge
intercalación balanceada

best fit
mejor ajuste

better than random distribution
distribución mejor que aleatoria

binary integer
entero binario

* Ante la falta de un lenguaje estandarizado en castellano para las ciencias de la computación, se ha elaborado el presente vocabulario con la traducción que hemos dado en este libro a los principales términos de la versión original en inglés. Esta labor se verá compensada por el servicio que pueda prestar al lector. (*Nota del Editor.*)

binary search	búsqueda binaria	btutil.prc program	programa <i>utilab.prc</i>
binary search tree	árbol de búsqueda binaria	bucket	compartimiento
binding	enlace	buffer bottleneck	cuello de botella en un buffer
bin_search() function	función <i>busca_bin()</i>	buffering	manejo de buffers
block	bloque	buffer pooling	depósito común de buffers
blocking factor	factor de bloque	byte count field	campo de conteo de bytes
block organization	organización por bloques	byte offset	distancia en bytes
boolean operations	operaciones booleanas	canonical form	forma canónica
bpi	bpi (bits por pulgada)	cascade merging	intercalación en cascada
btio.c program	programa <i>abes.c</i>	cast	conversión forzosa de tipo
B-tree	árbol B	CLOSE()	CLOSE()
B* tree	árbol B*	cluster	cúmulo
B + tree	árbol B ⁺	coalescence	fusión
btree.h header file	archivo de encabezado <i>ab.h</i>	collision	colisión
btutil.c program	programa <i>utilab.c</i>		

combined key retrieval	data-types
extracción de información mediante combinaciones de llaves	tipos de datos
compaction	#define command
compactación	orden #define
concatenation	delete bit
concatenación	bit de eliminación
conditional compilation	delete_record (RRN) function
compilación condicional	función <i>elimina_registro</i> (NRR)
controller	deletion
controlador	eliminación
CREATE()	delimiter
CREATE()	delimitador
cosequential operation	del_rec (RRN) function
operación secuencial coordinada	función <i>elim_reg</i> (NRR)
cosequential processing	digital RM05 disk drive
procesamiento secuencial coordinado	unidad de disco digital RM05
count key	direct access
llave de conteo	acceso directo
count subblock	direct access storage device (DASD)
sub-bloque de conteo	dispositivo de almacenamiento de acceso directo (DAAD)
cylinder	direct memory access (DMA)
cilindro	acceso directo a memoria (ADM)
check-hashing	disks
dispersión con revisión	discos
data structure	disk cache
estructura de datos	almacenamiento caché para disco
data transmission	disk configuration
transmisión de datos	configuración de disco

disk controller	entry sequenced file
controlador de disco	archivo con entradas secuenciales
disk drives	extendible hashing
unidades de disco	dispersión extensible
disk pack	extent
paquete de discos	extensión
double buffering	external fragmentation
manejo doble de buffers	fragmentación externa
double hashing	field
dispersión doble	campo
<i>driver.c</i>	delimited
<i>manejador.c</i>	delimitado
<i>driver.pas</i>	fixed length
<i>manejador pas</i>	de longitud fija
driver procedure	variable length
procedimiento <i>manejador</i>	de longitud variable
duplicate key	file
llave duplicada	archivo
effective recording density	closing
densidad de grabado efectiva	cerrar un,
effective transmission rate	controlling access to
tasa de transmisión efectiva	control del acceso a un,
empty record	detecting end of
registro vacío	detección del fin de un,
end-of-file characters	modifying
caracteres de fin de archivo	modificación de un,
end-of-line characters	opening and creating
caracteres de fin de línea	apertura y creación de un,
end-of-list value	physical and logical
valor de fin de lista	físico y lógico
	protecting
	protección de un,
	reading
	lectura de un,
	structure of
	estructura de un,
	unexpected characters in
	caracteres inesperados
	en un,

writing	fixed length record
escritura de un,	registro de longitud fija
file access	floating point numbers
acceso a archivos	números de punto flotante
file allocation table (FAT)	folding and adding
tabla de asignación de archivos (FAT)	desglosar y sumar
fileio.h header file	formatting
archivo de encabezado <i>arches.h</i>	dar formato
file manager	fragmentation
administrador de archivos	fragmentación
file organization	frame
organización de archivos	marco
file size	general ledger program
tamaño de archivo	programa de libro mayor
find.c	get_avail() function
<i>encuentra.c</i>	función <i>toma_disp()</i>
find.pas	get.prc program
<i>encuentra.pas</i>	programa <i>toma.prc</i>
find program	getrf.c program
programa <i>encuentra</i>	programa <i>tomarc.c</i>
find_sep() function (C)	grid files
función <i>encuentra_sep()</i> (en C)	archivos grid
find_sep() procedure (Pascal)	hash() function
procedimiento <i>encuentra_sep()</i> (en Pascal)	función <i>dispersión()</i>
first fit	hashing
primer ajuste	dispersión
fixed disk	header record
disco fijo	registro de encabezado
	height-balanced tree
	árbol balanceado en altura

hex dump	input routines
vaciado hexadecimal	rutinas de entrada
HIGH_VALUE	insert.c
VALOR_ALTO	<i>inserta.c</i>
home address	insert() function
dirección base	función <i>inserta()</i>
#include command	insertion
orden #include	inserción
index	insert.prc
índice	<i>inserta.prc</i>
index array	intersection
arreglo de índices	intersección
indexed sequential access	insertion
acceso secuencial indizado	inserción
index file	inverted list
archivo índice	lista invertida
indexed hash	interblock gap
dispersión indizada	hueco entre bloques
indexing	interleaving factor
indización	factor de intercalación
index set	internal fragmentation
conjuntos de índices	fragmentación interna
indirection	I/O processor
indirección	procesador de E/S
initialize() procedure	I/O buffer
procedimiento <i>initializar()</i>	buffer de E/S
initializing and loading file	ISAM
asignación de valores inciales y carga de un archivo	ISAM
initializing hashed file	key
inicio de un archivo para disper- sión	llave

key array	LIST program
arreglo de llaves	programa LISTA
key field	loading
campo llave	carga
keysort	locality
clasificación por llaves	localidad
keysort program	locate mode
programa <i>clasifllave</i>	modo de direcciones
key subblock	logical file
sub-bloque de llave	archivo lógico
K-way merge	loops
intercalación de K formas	ciclos
large file	LOW_VALUE
archivo grande	VALOR_BAJO
laser disc	LRU replacement
disco láser	reemplazo LRU
leaf node	<i>lseek()</i> function
nodo hoja	función <i>lseek()</i>
ledger program	magnetic tape
programa <i>mayor</i>	cinta magnética
length indicator	<i>main()</i> function
indicador de longitud	función <i>main()</i>
libraries	<i>makekey.c</i> program
acervos	programa <i>hazllave.c</i>
linear probing	mass storage system
verificación lineal	sistema de almacenamiento masivo
linked list	matching
lista ligada	correspondencia
linked lists	match operation
listas ligadas	operación de correspondencia

match procedure	procedimiento <i>correspondencia</i>	nominal recording density	densidad de grabación nominal
match procedures	procedimientos de correspondencia	nominal transmission rate	tasa de transmisión nominal
merge procedure	procedimiento <i>intercala</i>	nondata overhead	sobrecarga por datos usados para control
merge procedures	procedimientos de intercalación	OPEN()	OPEN()
merge runs	ejecuciones de intercalación	open addressing	direcciónamiento abierto
merging	intercalación	open() function	función <i>open()</i>
mid-square method	método del medio cuadrado	optical disc	disco óptico
minimum hashing	dispersión mínima	order, of B-tree	orden del árbol B
missing records	registros faltantes	overflow error	error por desbordamiento
move mode	modo de movimiento	overflow record	registro en saturación
multilist structures	estructuras multilistas	packing density	densidad de empaquetamiento
multiphase merge	intercalación en varias fases	page buffer	buffer de página
multistep merge	intercalación de varios pasos	paged	paginado
multiway merging	intercalación múltiple	paged binary tree	árbol binario paginado
node	nodo	paged binary trees	árboles binarios paginados

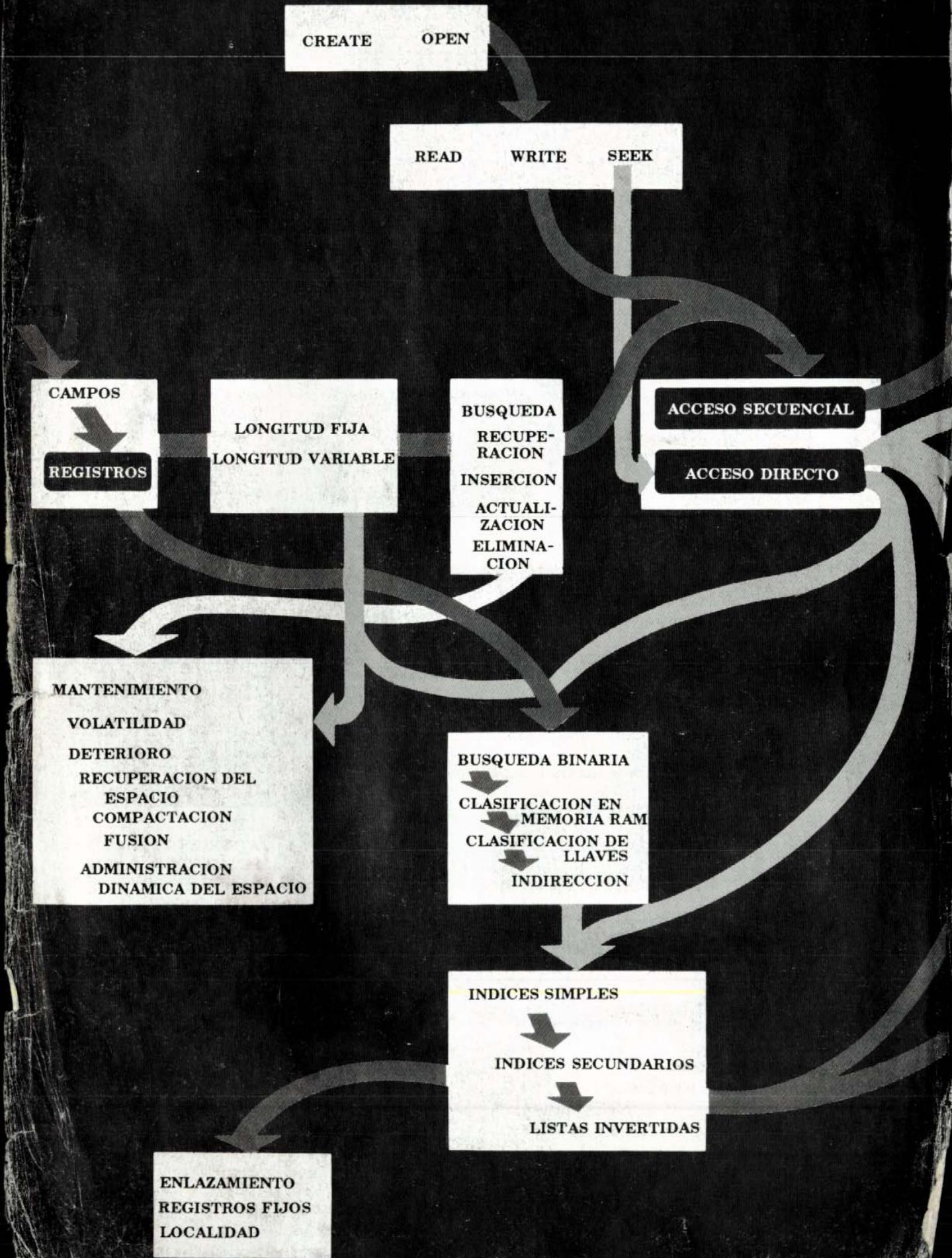
paged index	prefix
índice paginado	prefijo
page height	primary key
altura de página	llave primaria
paging	primary storage
paginación	almacenamiento primario
Pareto principle	prime division
principio de Pareto	división por número
parity	printf() function
paridad	función <i>printf()</i>
perfect hashing algorithm	progressive overflow
algoritmo de dispersión perfecta	saturación progresiva
physical file	chained
archivo físico	encadenada
physical placement	count-key
colocación física	con conteo de llaves
pinned record	promotion
registro fijo	promoción
placement strategy	protection mode
estrategia de colocación	modo de protección
platter	punched cards
plato	tarjetas perforadas
pointer field	Quicksort
campo apuntador	Quicksort
Poisson distribution	radix transformation
distribución de Poisson	transformación de la base
polyphase merge	RAM disk
intercalación polifásica	disco RAM
pop_avail() function	ramsort.c
función <i>saca_disp()</i>	<i>clasifram.c</i>
	ramsort.pas
	<i>clasifram.pas</i>

ramsort program	redistribution
programa <i>clasifram</i>	redistribución
random access memory (RAM)	redundant online storage
memoria de acceso aleatorio	almacenamiento redundante en
(RAM)	línea
READ()	reference field
READ()	campo de referencia
<i>readrec.c</i>	rehashing
<i>leereg.c</i>	redispersión
<i>readrec.pas</i>	relative block number (RBN)
<i>leereg.pas</i>	número relativo de bloque (NRB)
readrec program	relative record number (RRN)
programa <i>leereg</i>	número relativo de registro (NRR)
<i>readstrm.c</i>	replacement
<i>leesec.c</i>	reemplazo
<i>readstrm.pas</i>	replacement selection
<i>leesec.pas</i>	selección por reemplazo
readstrm program	reset statement
programa <i>leesec</i>	proposición <i>reset</i>
record	retrieve_record() procedure
registro	procedimiento <i>recupera_registro()</i>
record blocking	retrieving by key
manejo de registros en bloques	extracción por llave
record deletion	rewrite_index() procedure
eliminación de registros	procedimiento <i>reescribe_índice()</i>
record distribution	rewrite statement
distribución de registros	proposición <i>rewrite</i>
recursion	rotational delay
recursión	retraso por rotación

80/20 rule of thumb	secondary storage
regla empírica del 80/20	almacenamiento secundario
run	sector
porción	sector
run length	sector organization
longitud de las porciones	organización por sectores
scanf() function	SEEK()
función <i>scanf()</i>	SEEK()
scatter table	seeking
tabla de dispersión	localización
scope rules	seeks
reglas de alcance	desplazamiento
search	seek time
buscar	tiempo de desplazamiento
search() function	selection tree
función <i>busca()</i>	árbol de selección
search length	selective index
longitud de búsqueda	índice selectivo
search key	separator
llave de búsqueda	separador
search_on_secondary() procedure	sequence checking
procedimiento	revisión de secuencia
<i>búsqueda_en_secundario()</i>	
search tree	sequence set
árbol de búsqueda	conjunto de secuencias
secondary index	sequential access
índice secundario	acceso secuencial
secondary key	sequential access device
llave secundaria	dispositivo de acceso secuencial

sequential search	búsqueda secuencial	stack	pila
shortest separator	separador más corto	stacking	apilamiento
shell_sort() function	función <i>clasif_shell()</i>	starting and stopping times	tiempos de arranque y detención
Shell's sort	clasificación de Shell	statement blocks	bloques de proposiciones
sibling	hermano	std.prc program	programa <i>caddat.prc</i>
simple index	índice simple	storage capacity	capacidad de almacenamiento
simple prefix B⁺ tree	árbol B ⁺ de prefijos simples	storage classes	clases de almacenamiento
skip sequential processing	procesamiento secuencial con saltos	storage compaction	compactación del almacenamiento
sorting	clasificación	storage cost	costo del almacenamiento
sorting by merging	clasificación por intercalación	storage fragmentation	fragmentación de almacenamiento
sorting in RAM	clasificación en RAM	stream	secuencia
sort/merge utility programs	programas de servicio de clasificación por intercalación	streaming tape drive	unidad de cinta de grabado continuo (<i>streamers</i>)
split() procedure	procedimiento <i>divide()</i>	strfuncs.c program	programa <i>funscads.c</i>
splitting	división	strng	<i>cadena</i>
strings	cadenas	update.pas program	programa <i>actualiza.pas</i>
subblock	sub-bloque	updating	actualización

synchronization loop	undeleting
ciclo de sincronización	anulación de eliminación
synonym	uniform distribution
sinónimo	distribución uniforme
tag sorting	unions
clasificación por marca	uniones
tape	variable length record
cinta	registro de longitud variable
tombstone	variable order B-tree
marca de inutilización	árbol B de orden variable
toolkit	virtual B-tree
conjunto de herramientas	árbol B virtual
tools.prc program	virtual hashing
programa <i>herramientas.prc</i>	dispersión virtual
top-down construction	volatility
construcción descendente	volatilidad
tournament tree	VSAM
árbol de torneo	VSAM
track	worst fit
pista	peor ajuste
transfer time	WRITE()
tiempo de transferencia	WRITE()
two-pass loading	writrec.c
carga en dos pasos	<i>escribereg.c</i>
update.c program	writrec.pas
programa <i>actualiza.c</i>	<i>escribereg.pas</i>
writrec program	writstrm.pas
programa <i>escribereg</i>	<i>escribesec.pas</i>
writstrm.c	writstrm program
<i>escribesec.c</i>	programa <i>escribesec</i>



PROCESAMIENTO SECUENCIAL
COORDINADO
PAREAMIENTO/INTERCALACION
SELECCION POR REEMPLAZO
CLASIFICACION POR
INTERCALACION

ACCESO DIRECTO

CORRESPONDENCIA DE LLAVE A
DIRECCION

DISPERSION
BUSQUEDA
INSERCIÓN
...

COLISIONES
CUMULOS
LONGITUD DE BUSQUEDA
DENSIDAD DE EMPA-
QUETAMIENTO
LOCALIDAD

DISTRIBUCIONES
SATURACION
COMPARTIMENTOS

ACCESO INDIZADO SIMPLE

ARBOL BINARIO SIMPLE

ARBOL AVL

ARBOL BINARIO PAGINADO

ARBOLES N-ARIOS PAGINADOS

ARBOL B
BUSQUEDA
INSERCIÓN

ALTURA
PATRONES DE ACCESO
MANEJO DE BUFFERS
ENLACE
LOCALIDAD

CONJUNTO DE
SECUENCIAS
BUSQUEDA
INSERCIÓN
...

ACCESO SECUENCIAL INDIZADO

CONJUNTO INDICE DE ARBOL B

ARBOL B+

SEPARADORES MAS CORTOS

ARBOL B+ DE PREFIJOS SIMPLE