

Por último, gracias a nuestras esposas, Miriam Tenenbaum, Vivienne Langsam y Gail Augenstein por sus consejos y su impulso durante la larga y ardua tarea de producir este libro.

AARON TENENBAUM

YEDIDYAH LANGSAM

MOSHE AUGENSTEIN

1

Introducción a la estructura de datos

Una computadora es una máquina que maneja información. El estudio de la ciencia de la computación incluye saber la forma en que se organiza la información en una computadora, cómo puede manejarse y la manera en que puede utilizarse esa información. Por tanto, es de suma importancia para un estudiante de la ciencia de la computación, entender los conceptos de organización y manejo de información para continuar el estudio de la disciplina.

1.1 INFORMACION Y SIGNIFICADO

Si la ciencia de la computación es fundamentalmente el estudio de la información, la primera pregunta que surge es: ¿qué es la información? Por desgracia, aunque ese concepto es la piedra angular de la disciplina, la interrogante anterior no puede contestarse con precisión. En este sentido, el concepto de información en esta disciplina es similar a los conceptos de punto, recta y plano para la geometría: son un grupo de términos indefinidos con los que se pueden hacer proposiciones, pero no pueden definirse con conceptos más elementales.

En la geometría es posible hablar de la longitud de una recta, a pesar de que el concepto de recta es en sí mismo indefinido. La longitud de una recta es una medida cuantitativa. De manera similar, en la ciencia de la computación podemos hablar de cantidades de información. La unidad básica de información es el *bit*, cuyo valor confirma una de dos posibilidades excluyentes. Por ejemplo, un interruptor puede estar en una de dos posiciones pero no en ambas al mismo tiempo, pero el hecho es que el estar en la posición de “encendido” o en la de “apagado” representa un bit.

de información. Si un dispositivo puede estar en más de dos estados, al tomar un estado particular se tiene más de un bit de información. Por ejemplo si un disco selector tiene ocho posibles posiciones, al estar en la posición 4 descarta las otras siete posibilidades; de la misma forma, cuando un conmutador está en la posición de “encendido”, descarta la única posibilidad restante.

Otra manera de pensar en este fenómeno es la siguiente: supongamos que sólo tenemos conmutadores de dos posiciones, pero podemos utilizar tantos como necesitamos. ¿Cuántos de dichos dispositivos serán necesarios para representar un disco selector con ocho posiciones? Claramente, un conmutador nada más puede representar dos posiciones (Fig. 1.1.a). Dos conmutadores bastarían para cuatro posiciones diferentes (Fig. 1.1.b), y se requerirían tres de éstos para ocho posiciones diferentes (Fig. 1.1.c). En general, n apagadores pueden representar 2^n posiciones diferentes.

Los dígitos binarios 0 y 1 se usan para representar dos estados posibles de un bit particular (en realidad la palabra bit es una contracción de las palabras “binary digit”). Dado n bits, se utiliza una cadena formada con n dígitos 1 y 0 para representar una configuración. Por ejemplo, la cadena 101011 representa 6 conmutadores: el primero está “encendido” (1), el segundo “apagado” (0), el tercero encendido, el cuarto apagado, y el quinto y el sexto encendidos.

Hemos visto que tres bits son suficientes para representar ocho posibilidades. Las ocho configuraciones posibles de esos tres bits (000, 001, 010, 011, 100, 101, 110 y 111) pueden utilizarse para representar los enteros del 0 al 7. Sin embargo, no hay nada intrínseco en esas configuraciones que impliquen que cada una de ellas represente un entero en particular. Cualquier asignación que se haga de valores enteros para ciertas combinaciones de bits es válida siempre y cuando dos valores enteros no se asigne a una misma configuración. Una vez que se ha determinado tal asignación, cualquier combinación particular de bits puede interpretarse sin ambigüedad como un entero específico. Examinemos algunos métodos muy usados para la interpretación de combinaciones de bits como enteros.

Enteros binarios y decimales

El método utilizado con mayor frecuencia para interpretar combinaciones de bits como enteros no negativos es el *sistema de números binario*. En ese sistema, cada posición de bit representa una potencia de 2. La posición de la extrema derecha representa 2^0 (que es igual a 1), la siguiente posición a la izquierda es 2^1 (igual a 2), la siguiente significa 2^2 (que es cuatro), y así sucesivamente. Un entero se representa como suma de potencias de 2. Una cadena compuesta únicamente de ceros representa al número 0. Si aparece el dígito 1 en una posición de un bit particular, la potencia de 2 que representa dicha posición se incluye en la suma; pero si hay un 0, no se toma en cuenta. Por ejemplo, el grupo de bits 00100110 tiene un 1 en las posiciones 1, 2 y 5 (si se cuenta de derecha a izquierda y si se considera la posición de la extrema derecha como la posición 0). Así, 00100110 representa al entero $2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$. Según esta interpretación, cualquier cadena de bits cuya longitud sea n representa un número entero único no negativo entre 0 y $2^n - 1$; asimismo, cualquier entero no negativo entre 0 y $2^n - 1$ puede representarse por una sola cadena de bits de longitud n .

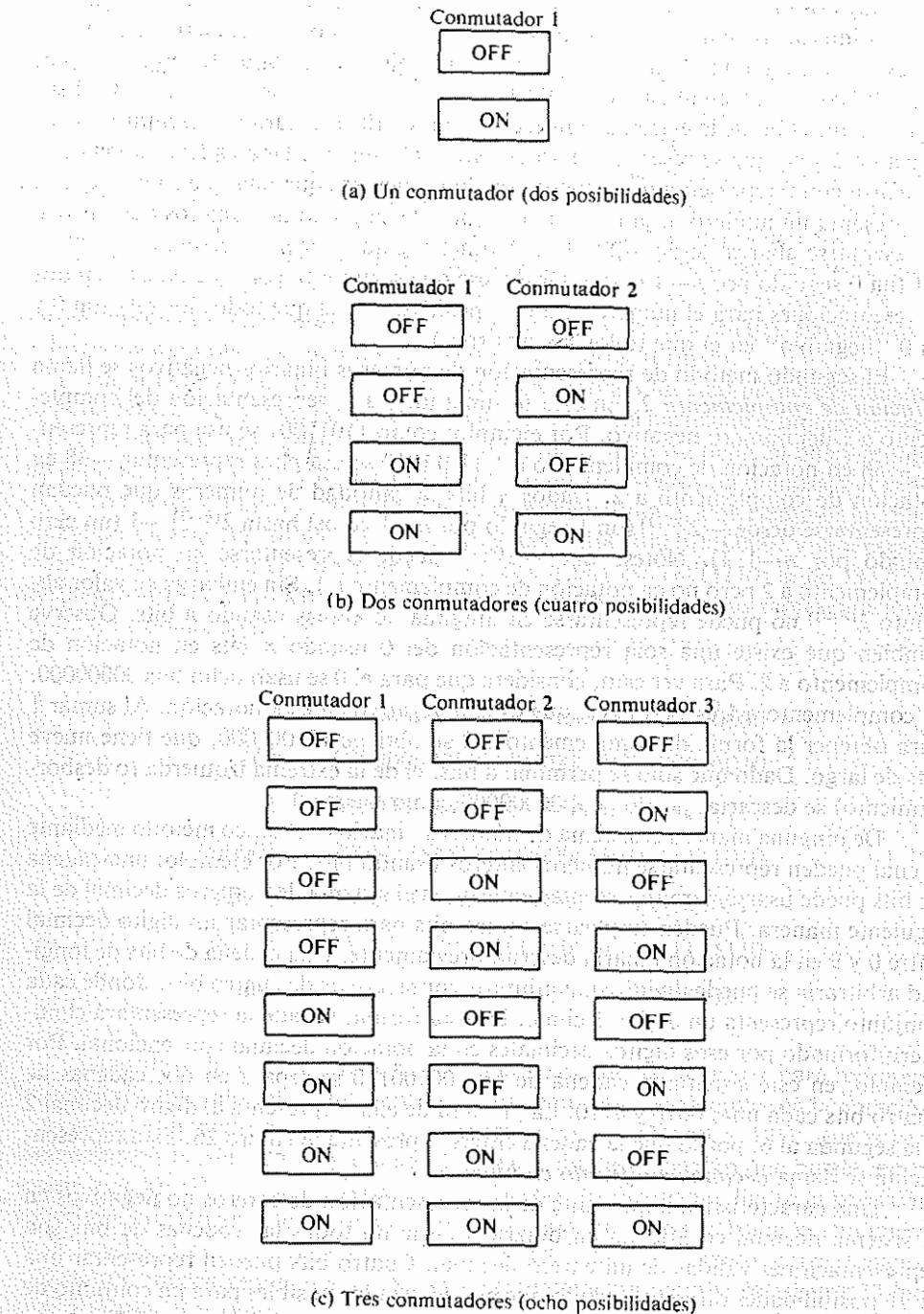


Figura 1.1.1

Hay dos métodos muy utilizados para representar números binarios negativos. En el primero, llamado notación de complemento a uno, un número negativo se representa cambiando el valor absoluto de cada bit al valor opuesto. Por ejemplo, como 00100110 representa al 38, 11011001 se emplea para representar al -38. Esto significa que el bit de la extrema izquierda ya no se utilizará para representar una potencia de 2 sino que se reservará para el signo del número. Una cadena de bits que comienza con 0 representa un número positivo, mientras que una que comienza con 1 representa un número negativo. Dados n bits, la cantidad de números que pueden representarse abarca desde $-2^{(n-1)} + 1$ (un 1 seguido por $n-1$ ceros) hasta $2^{(n-1)} - 1$ (un 0 seguido por $n-1$ unos). Obsérvese que según esta convención existen dos representaciones para el número 0: un 0 "positivo" en el que todos los bits son 0 y un 0 "negativo" en el que todos los bits son 1.

El segundo método de representación de números binarios negativos se llama **notación de complemento 2**. En ésta se suma un 1 a la representación del complemento a 1 del número negativo. Por ejemplo: como 11011001 se usa para representar -38 en notación de complemento a 1 11011010 se usa para representar -38 en notación de complemento a 2. Dados n bits la cantidad de números que puedan representarse desde $-2^{(n-1)}$ (un 1 seguido por $n-1$ ceros) hasta $2^{(n-1)} - 1$ (un cero seguido por $n-1$ 1). Nótese que $-2^{(n-1)}$ puede representarse en notación de complemento a 2 pero no en notación de complemento a 1. Sin embargo su valor absoluto $2^{(n-1)}$ no puede representarse en ninguna de ambas usando n bits. Observe también que existe una sola representación del 0 usando n bits en notación de complemento a 2. Para ver esto, considere que para el 0 se usan ocho bits 00000000. El complemento a 1 es 11111111, que es el 0 negativo en esta notación. Al sumar 1 para obtener la forma de complemento a 2 se obtiene 10000000, que tiene nueve bits de largo. Dado que sólo se permiten 8 bits, el de la extrema izquierda (o desbordamiento) se descarta, por lo cual 00000000 como menos 0.

De ninguna manera el sistema de números binario es el único método mediante el cual pueden representarse números enteros usando bits. Por ejemplo: una cadena de bits puede usarse para representar enteros en el sistema de números decimal de la siguiente manera. Pueden emplearse cuatro bits para representar un dígito decimal entre 0 y 9 en la notación binaria descrita previamente. Una cadena de bits de longitud arbitraria se puede dividir en conjuntos consecutivos de cuatro bits, donde cada conjunto representa un dígito decimal. De esta forma, la cadena representará el número formado por esos dígitos decimales en la notación decimal convencional. Por ejemplo, en este sistema la cadena de bits 00100110 se separa en dos cadenas de cuatro bits cada una: 0010 y 0110. La primera de ellas representa al dígito decimal 2 y la segunda al 6, por lo que la cadena entera representa al entero 26. Esta representación se llama **decimal codificado en binario**.

Una característica importante de la representación de enteros no negativos en el sistema decimal codificado en binario es que no todas las cadenas de bits son representaciones válidas de un entero decimal. Cuatro bits pueden representar una de 16 posibilidades diferentes, dado que hay 16 estados posibles para un conjunto de cuatro bits. Sin embargo, en la representación de enteros en decimal codificado en binario se utilizan solamente 10 de estas posibilidades. Esto es, códigos como 1010 y 1100, cuyos valores binarios son mayores o iguales que 10, no son válidos en un número decimal codificado en binario.

Números reales

El método usual que emplean las computadoras para representar números reales es la **notación de punto flotante**. Hay muchas variaciones de esta notación y cada una de ellas tiene características individuales. El concepto clave es que un número real se representa por un número, llamado **mantisa**, multiplicado por una **base** elevada a una potencia entera, llamada **exponente**. Por lo común, para representar números reales diferentes la base se fija y la mantisa y el exponente varían. Por ejemplo: si se fija la base como 10, el número 387.53 podría representarse como 38753×10^{-2} (recuerde que 10^{-2} es .01). La mantisa es 38753 y el exponente -2. Otras representaciones posibles son 38753×10^3 y 387.53×10^0 . Escogemos la representación donde la mantisa es un entero sin ceros al final.

En la notación de punto flotante que describimos (la cual no necesariamente está implantada tal y como está descrita en una máquina particular), un número real se representa por una cadena de 32 bits en la que la mantisa tiene 24 bits y le sigue un exponente con 8 bits. La base está fijada como 10. Ambos, la mantisa y el exponente son enteros binarios en notación de complemento a 2. Por ejemplo, la representación binaria con 24 bits de 38753 es 000000001001011101100001 y la representación binaria de -2 en complemento a 2 con 8 bits es: 11111110; la de 387.53 es 0000000010010111011000011111110. Otros números reales y sus representaciones de punto flotante son:

0	00000000000000000000000000000000
100	0000000000000000000000000000100000010
.5	00000000000000000000000000001011111111
.000005	00000000000000000000000000001011111010
12000	0000000000000000000000000000110000000011
-387.53	111111101101000100111111111110
-12000	1111111111111111111010000000011

La ventaja de la notación de punto flotante es que puede usarse para representar números con valor absoluto muy grande o pequeño. Por ejemplo, en la notación vista antes, el número mayor que puede representarse es $(2^{23}-1) \times 10^{127}$, que es verdaderamente un número muy grande. El menor número positivo que puede representarse es 10^{-128} , el cual es ciertamente pequeño. El factor que limita la precisión con la que pueden representarse números en una máquina en particular es el número de dígitos binarios significativos en la mantisa. No todos los números entre el mayor y el menor pueden representarse. Nuestra representación permite solamente 23 bits significativos. Por tanto, un número como 10 millones 1 —que requiere 24 dígitos binarios significativos en la mantisa— debería aproximarse por 10 millones, (1×10^7) que sólo requiere un dígito significativo.

Cadenas de caracteres

Como todos sabemos, la información no se interpreta siempre numéricamente. Los nombres, títulos de trabajo y direcciones también tienen que representarse de alguna forma en una computadora. Para permitir la representación de tales objetos no numéricos, es necesario otro método adicional de interpretación de cadenas de bits. Tal información se representa por lo general en forma de cadenas de caracteres. Por ejemplo, en algunas computadoras los ocho bits 00100110 representan el carácter '&'. Una pauta diferente de ocho bits representa el carácter 'A', se usa otro para 'B', otro para 'C' e incluso otros más para cada carácter que tenga una representación en una máquina en particular. Una máquina rusa usa pautas de bits para representar caracteres rusos, mientras que una israelí lo hace para representar caracteres hebreos. (En realidad los caracteres empleados resultan transparentes para la máquina; el conjunto de caracteres puede cambiarse usando una cadena de impresión diferente en la impresora.)

Si 8 bits representan un carácter, entonces se pueden representar hasta 256 caracteres diferentes, dado que hay 256 patrones de ocho bits diferentes. Si la cadena 11000000 se utiliza para simbolizar el carácter 'A' y 11000001 para el 'B', la cadena de bits 1100000011000001 podría representar la cadena de caracteres 'AB'. En general, una cadena de caracteres (STR) se representa mediante la concatenación de las cadenas de bits que representan los caracteres individuales.

Como en el caso de los enteros, no hay nada en una cadena de bits particular que la haga intrínsecamente apropiada para representar un carácter específico. La asignación de cadenas de bits a caracteres es por completo arbitraria, pero debe asignarse de manera consistente. Es conveniente apegarse a una regla para la asignación de cadenas de bits a caracteres. Por ejemplo, dos cadenas de bits pueden asignarse a dos letras, de manera que aquella con menor valor binario sea asignada a la letra que primero aparece en el alfabeto. Sin embargo, tal regla no será más que una conveniencia; no está regida por una relación intrínseca entre cadenas de bits y caracteres. En realidad las computadoras también difieren en cuanto al número de bits empleados para representar caracteres. Algunas computadoras usan 7 bits (y por consiguiente permiten sólo hasta 128 caracteres), otras usan 8 (hasta 256 caracteres) y algunas más usan 10 (hasta 1024 caracteres posibles). El número de bits necesario para representar un carácter en una computadora se llama *tamaño de byte* y un grupo con ese número de bits se llama *byte*.

Note que cuando se usa ocho bits para representar un carácter significa que pueden representarse 256 caracteres posibles. No es muy frecuente encontrar una computadora que emplea tantos caracteres distintos (aunque es concebible que una computadora que emplee tantos caracteres distintos (aunque es concebible que una vas, negritas y otros caracteres) por lo que muchas de las combinaciones de ocho bits no se usan para representar caracteres.

Así, vemos que la información por sí misma no tiene significado. Puede asignarse cualquier significado a una pauta particular de bits, siempre y cuando esto se haga de manera consistente. Es la interpretación de una pauta de bits la que le confiere significado. Por ejemplo, la cadena de bits 00100110 puede interpretarse como el número 38 (binario), como el número 26 (decimal codificado binario), o como el carácter '&'. Un método de interpretación de una pauta de bits con frecuencia se

conoce como el *tipo de datos*. Hemos presentado varios tipos de datos: enteros binarios, enteros decimales no negativos codificados en binario, números reales y cadenas de caracteres. Las preguntas clave son cómo determinar los tipos de datos que se pueden aprovechar para interpretar patrones de bits y los tipos de datos que habrán de utilizarse para la interpretación de un patrón de bits particular.

Hardware y software

La *memoria* (también llamada *almacenamiento* o *memoria principal*) de una computadora es simplemente un grupo de bits (comutadores). En todo momento de la operación de una computadora cualquier bit particular en la memoria es 0 o 1 (encendido o apagado). El 1 o 0 que contiene un bit se llama *valor* o *contenido*.

Los bits están agrupados en unidades más grandes (como bytes) en la memoria de la computadora. En algunas computadoras se agrupan varios bytes en unidades llamadas *palabras*. A cada una de estas unidades (byte o palabra, según sea la máquina) se le asigna una *dirección*; es decir, un nombre que identifica en la memoria a una unidad particular del resto. Esta dirección suele ser numérica, de tal manera que podemos hablar del byte 746 o de la palabra 937. Es frecuente llamar *localidad* a una dirección; además, los contenidos de una localidad son los valores de los bits que constituyen una unidad en esa localidad.

Toda computadora tiene un conjunto de tipos de datos "nativos", lo cual significa que se construyó con un mecanismo para manejar pautas de bits, compatible con los objetos que éstos representan. Por ejemplo, suponga que una computadora contiene una instrucción para sumar dos enteros binarios y poner su suma en una localidad de memoria para su uso posterior. Entonces tiene que existir un mecanismo dentro de la computadora para

1. Extraer de dos localidades dadas pautas de bits de los operandos.
2. Producir una tercera pauta de bits que represente al entero binario, el cual es la suma de los dos enteros binarios representados por los dos operandos.
3. Almacenar la pauta de bits resultante en una localidad dada.

La computadora "sabe" interpretar la pauta de bits de las localidades de memoria dadas como enteros binarios porque el hardware que ejecuta esa instrucción particular está diseñado para hacerlo. Esto es similar a una luz que "sabe" estar encendida cuando el conmutador está en una posición particular.

Si la misma máquina tiene también una instrucción para sumar dos números reales, debe existir un mecanismo interconstruido para interpretar los operandos como números reales. Se requieren dos instrucciones distintas para las dos operaciones, y cada instrucción porta consigo una identificación implícita de los tipos de sus operandos, así como sus localidades explícitas. Es, por consiguiente, responsabilidad del programador conocer los tipos de datos que están contenidos en cada localidad que se utiliza. También incumbe al programador elegir el uso de adición de reales o de enteros para obtener la suma de dos números.

Un lenguaje de programación de alto nivel ayuda mucho en esta tarea. Por ejemplo, si un programador en C declara

```
int x, y;  
float a, b;
```

se reserva espacio en cuatro localidades para cuatro números diferentes. Los *identificadores* *x*, *y*, *a*, y *b*, pueden hacer referencia a esas 4 localidades. Se usa un identificador en lugar de una dirección numérica para referirse a una localidad particular de memoria porque así conviene al programador. Los contenidos de las localidades reservadas para *x* y *y*, serán interpretados como enteros mientras que los de *a* y *b* serán interpretados como números de punto flotante. El compilador, que es responsable de traducir los programas en C al lenguaje de la máquina traducirá el “+” en la instrucción

x = *x* + *y*;

como adición de enteros, y el “+” en la instrucción

a = *a* + *b*;

como suma de puntos flotantes. Un operador como “+” en realidad es un operador *genérico* pues tiene varios significados diferentes dependiendo del contexto. El compilador libera al programador de la especificación del tipo de suma que debe ejecutarse al examinar el contexto y usar la versión adecuada.

Es importante reconocer el papel clave que desempeñan las declaraciones en un lenguaje de alto nivel, pues por medio de declaraciones el programador especifica cómo el programa debe interpretar los contenidos de la memoria. Al hacerlo, una declaración especifica cuánta memoria se necesita para una entidad particular, cómo deben interpretarse los contenidos de dicha memoria y otros detalles vitales. Las declaraciones especifican también con exactitud al compilador el significado de los símbolos de operación que se usan subsecuentemente.

El concepto de implantación

Hasta ahora hemos visto los tipos de datos como un método para interpretar los contenidos de la memoria de una computadora. El conjunto de tipo de datos nativos que puede tener una computadora está determinado por las funciones que han sido alambradas dentro de su hardware. Sin embargo, podemos ver el concepto de tipo de datos desde una perspectiva muy diferente; es decir, no verlo en términos de lo que puede hacer una computadora sino en términos de lo que quiere el usuario hacer. Por ejemplo, si deseamos obtener la suma de dos enteros, no necesitamos preocuparnos por el mecanismo detallado mediante el cual se obtendrá la suma. Nos interesa la manipulación del concepto matemático de suma y no la manipulación de los bits de hardware. Puede usarse el hardware de la computadora para representar un entero y es útil sólo en cuanto sea exitosa la representación.

Cuando sepáramos el concepto de “tipo de datos” de las capacidades del hardware de una computadora, podemos considerar un sinnúmero de tipos de datos. Un tipo de datos es un concepto abstracto definido por un conjunto de propiedades

lógicas. Una vez que se define el tipo de datos abstracto y se especifican las operaciones permitidas que involucran a este tipo (o uno muy parecido), puede *implantarse*. Una *implantación* puede ser una implantación de hardware en la cual se diseña y construye como parte de una computadora el circuito necesario para ejecutar la operación requerida, o una *implantación de software*, en la cual se escribe un programa que consta de instrucciones existentes en el hardware para interpretar en la forma deseada las cadenas de caracteres y ejecutar las operaciones requeridas. Así que una implantación de software incluye una especificación de cómo se representa un objeto del tipo de datos nuevo por medio de objetos de tipos de datos ya existentes, de acuerdo también a una especificación de cómo se maneja un objeto.

En lo que resta del libro, usaremos el término “implantación” en el sentido de implantación de software.

Un ejemplo

Ilustremos los conceptos con un ejemplo. Supongamos que el hardware de una computadora contiene una instrucción

MOVE (source, dest, length)

que copia una cadena de caracteres de una longitud de bytes de una dirección especificada por *origen* a una dirección especificada por *destino*. (Escribimos las instrucciones de hardware con mayúsculas.) La longitud debe especificarse mediante un entero, por lo cual la escribimos con minúsculas. *Orden* y *destino* pueden especificarse por medio de identificadores que representen localidades de memoria.) Un ejemplo de esta instrucción es MOVE (*a*, *b*, 3), que copia los tres bytes que inician en la localidad *a*, en los primeros tres bytes que inician en la localidad *b*.

Observe los diferentes papeles que desempeñan los identificadores *a* y *b* en esta operación. El primer operando de la instrucción MOVE consiste en los contenidos de la localidad especificada por el identificador *a*. El segundo operando, sin embargo, no consta de los contenidos de la localidad *b* ya que son irrelevantes para la ejecución de la instrucción. Más bien, la localidad misma es el operando pues especifica el destino de la cadena de caracteres. Aunque un identificador se usa siempre para denotar una localidad, es común que se use para referir los contenidos de esa localidad. Siempre es evidente por el contexto si se usa un identificador para referirse a una localidad o a sus contenidos. El identificador que aparece como el primer operando de la instrucción MOVE se refiere a los contenidos de memoria, mientras que el segundo se refiere a la localidad.

Suponemos también que el hardware de la computadora contiene las instrucciones aritméticas y de saltos normales que indicamos al usar una notación similar a la de C. Por ejemplo, la instrucción

z = *x* + *y*;

interpreta los contenidos de los bytes en las localidades *x* y *y* como enteros binarios, los suma e inserta la representación binaria de su suma en el byte de la localidad *z*.

(Nosotros no operamos con enteros de longitud mayor que un byte e ignoramos la posibilidad de desbordamiento.) De nuevo, aquí x y y se usan para referirse a los contenidos de memoria, mientras que z se refiere a la localidad de memoria, pero la interpretación adecuada es clara por el contexto.

En ocasiones es deseable sumar una cantidad a una dirección para obtener otra. Por ejemplo, si a es una localidad en la memoria, podríamos querer referirnos a la localidad cuatro bytes más allá de a . No podemos hacerlo por medio de $a + 4$, pues esta notación está reservada para los contenidos enteros de la localidad $a + 4$. En consecuencia introducimos la notación $a[4]$ para referirnos a dicha localidad; también introducimos la notación $[x]$ para referirnos a la dirección obtenida al sumar a la dirección a , los contenidos binarios enteros del byte en x .

La instrucción MOVE exige especificar al programador la longitud de la cadena por copiar. Así, su operando es una cadena de caracteres de longitud fija (esto es, debe conocerse la longitud de la cadena). Una cadena de caracteres de longitud fija y un entero binario del tamaño de un byte pueden considerarse tipos de datos nativos de esta máquina en particular.

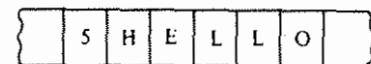
Suponga que deseamos implantar en esta máquina cadenas de caracteres de longitud variable. Es decir, queremos que los programadores puedan usar la instrucción

MOVEVAR (source, dest) para mover una cadena de caracteres de longitud variable de la localización *origen* a la localización *destino* sin estar obligados a especificar ninguna longitud.

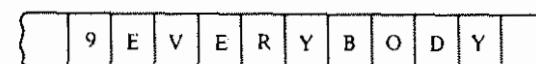
Para implantar este nuevo tipo de datos, tenemos que decidir primero cómo debe representarse en la memoria de la máquina e indicar después cómo debe manejarse esta representación. Claramente, es necesario conocer cuántos bytes deben moverse para ejecutar esta instrucción. Como la operación MOVEVAR no especifica el número, debe incluirse en la representación misma de la cadena de caracteres. Una cadena de caracteres de longitud variable de longitud l , puede representarse por medio de un conjunto contiguo de $l + 1$ bytes ($l < 256$). El primer byte contiene la representación binaria de la longitud l y los bytes restantes contienen la representación de los caracteres en la cadena. Las representaciones de estas tres cadenas se muestran en la figura 1.1.2. [Observe que los dígitos 5 y 9 en esas figuras no están colocados para representar las pautas de bits de los caracteres '5' y '9' sino las pautas 00000101 y 00001001 (si suponemos 8 bits en un byte), que representan a los enteros 5 y 9. De manera similar, en la figura 1.1.2c el 14 está representado por la pauta de bits 00001110. Observe también que esta representación es muy diferente de la manera en que son implantadas realmente las cadenas de caracteres en el lenguaje C.]

El programa para implantar la operación MOVEVAR puede escribirse como sigue (i es una localidad de memoria auxiliar):

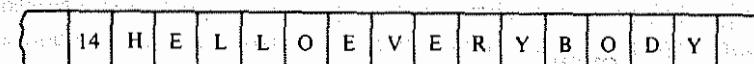
```
MOVE(source, dest, 1);
for (i=1; i < dest; i++)
    MOVE(source[i], dest[i], 1);
```



(a)



(b)



(c)

Figura 1.1.2 Cadena de caracteres de longitud variable.

De igual manera podemos implantar una operación CONCATVAR ($c1, c2, c3$) para concatenar dos cadenas de caracteres de longitud variable $c1$ y $c2$, y colocar el resultado en $c3$. La figura 1.1.2c muestra la concatenación de las dos cadenas de caracteres de las figuras 1.1.2a y 1.1.2b:

```
/* se mueve la longitud */
z = c1 + c2;
MOVE(z, c3, 1);
/* se mueve la primera cadena */
for (i = 1; i <= c1; MOVE(c1[i], c3[i], 1));
/* se mueve la segunda cadena */
for (i = 1; i <= c2) {
    x = c1 + i;
    MOVE(c2[i], c3[x], 1);
} /* fin de for */
```

Sin embargo una vez que la operación MOVEVAR ha sido definida, CONCATVAR puede implantarse al usar MOVEVAR como sigue:

```
MOVEVAR(c2, c3[c1]); /* se mueve la segunda cadena */
MOVEVAR(c1, c3); /* se mueve la primera cadena */
z = c1 + c2; /* se calcula la longitud de la cadena resultante */
MOVE(z, c3, 1);
```

La figura 1.1.3 muestra las fases de esta operación con las cadenas de la figura 1.1.2. Aunque la última versión es más corta, no es necesariamente más eficiente, pues se

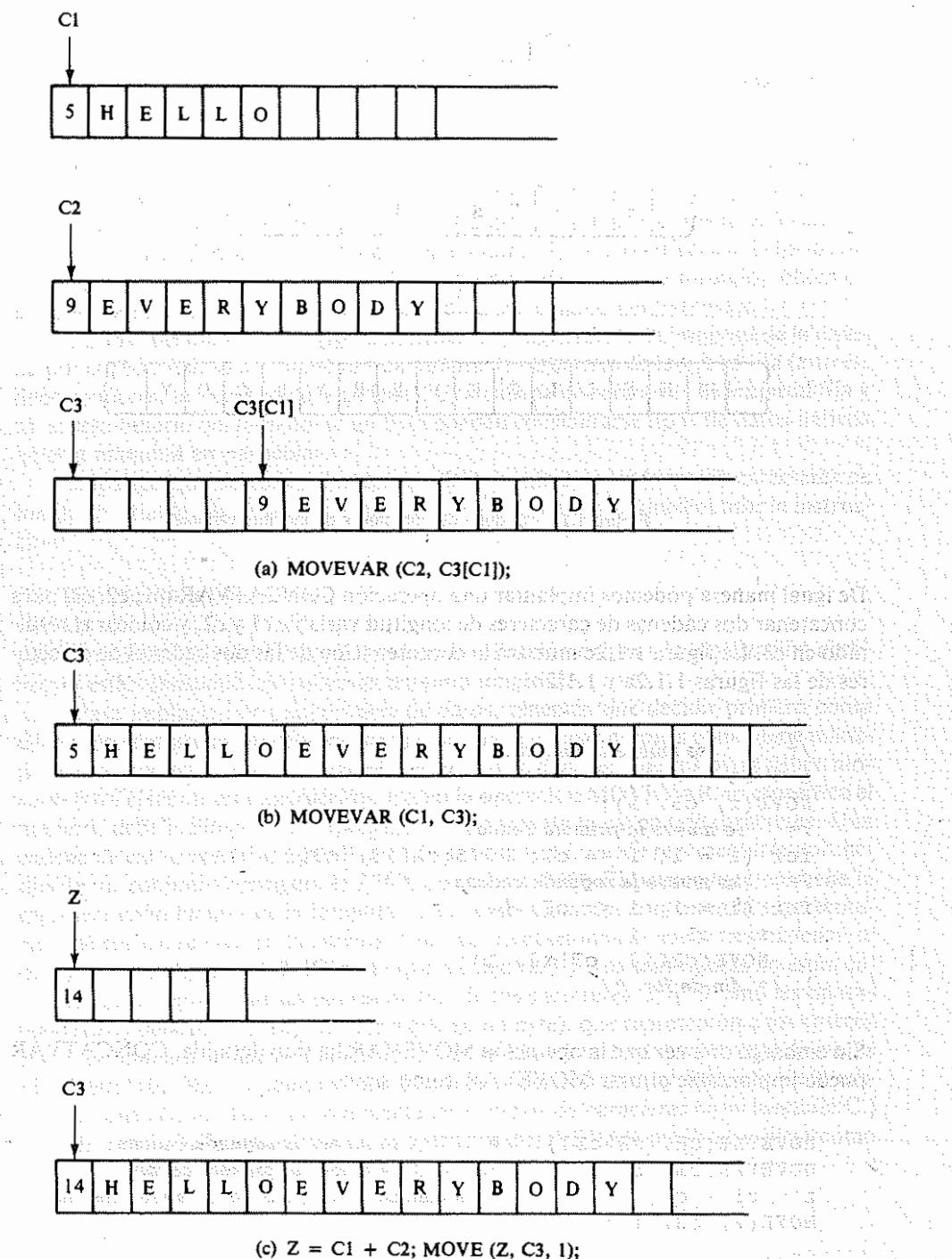


Figura 1.1.3 Las operaciones CONCATVAR.

ejecutan todas las instrucciones usadas al implantar MOVEVAR cada vez que se usa esta operación.

La instrucción $z = c_1 + c_2$ en los dos algoritmos anteriores tiene particular interés. La instrucción de suma opera en forma independiente del uso que tengan sus operandos (en este caso, partes de cadenas de longitud variable). La instrucción está diseñada para tratar a sus operandos como enteros de un solo byte, sin tomar en cuenta ningún uso que pueda darles el programador. De manera similar, al $c_3[c_1]$ refiere a la localidad cuya dirección está dada al sumar los contenidos del byte en la localidad c_1 en la dirección c_3 . Así, se considera que el byte en c_1 contiene un entero binario, aunque también es el principio de una cadena de caracteres de longitud variable, lo cual ilustra el hecho de que los tipos de datos son un método para tratar los contenidos de la memoria, los cuales no tienen un significado propio.

Observe que esta representación de cadenas de caracteres de longitud variable solamente permite cadenas cuya longitud es menor o igual al mayor entero binario que corresponde a un byte. Si un byte tiene ocho bits, esto significa que la cadena más grande posible es de 255 (2^{8-1}) caracteres. Para permitir cadenas más grandes debemos escoger otra representación y escribir otro conjunto de programas. Si usamos esta representación de cadenas de caracteres de longitud variable, la operación de concatenación quedará invalidada en caso de que la cadena resultante sea de una longitud mayor a los 255 caracteres. Como el resultado de esta operación no está definido, pueden implantarse diversas acciones si intentamos realizar la operación. Una posibilidad es usar solamente los primeros 255 caracteres del resultado. Otra, es ignorar por completo la operación y no mover nada al campo de resultado. También existe la posibilidad de imprimir un mensaje de advertencia o de suponer que el usuario quiere lograr cualquier resultado que decida.

En realidad, el lenguaje C utiliza una implantación de cadenas de caracteres por completo diferente que evita esta limitación en la longitud de las cadenas. En el lenguaje C, todas las cadenas terminan con el carácter especial '\0' el compilador lo coloca de manera automática al final de cada cadena, y dicho carácter nunca aparece cuando se escribe la cadena. Como no sabemos con antelación la longitud, todas las operaciones con cadenas deben realizarse en un carácter a la vez hasta encontrar '\0'.

El programa para implantar la operación MOVEVAR bajo esta implantación, puede escribirse como sigue:

```
i = 0;
while (source[i] != '\0') {
    MOVE(source[i], dest[i], 1);
    i++;
}
dest[i] = '\0';
/* se indica terminación de cadena de destino con '\0' */
```

Para implantar la operación de concatenación CONCATVAR (c_1, c_2, c_3) podemos escribir:

```

i = 0;
/* se mueve la primera cadena */
while (c1[i] != '\0') {
    MOVE(c1[i], c3[i], 1);
    i++;
}
/* se mueve la segunda cadena */
j = 0;
while (c2[j] != '\0') {
    MOVE(c2[j++], c3[i++], 1);
}
/* se indica terminación de cadena de destino con un '\0' */
c3[i] = '\0';

```

Una desventaja de la implantación en el lenguaje C de cadenas de caracteres es que sólo se sabrá la longitud de la cadena, si se cuenta carácter por carácter y se encuentra el símbolo '\0'. Esto está más que compensado por la ventaja de no tener un límite arbitrario sobre la longitud de la cadena.

Una vez que se ha elegido una representación para objetos de un tipo de datos determinado y que se han escrito las rutinas para operar con esas representaciones, el programador tiene la posibilidad de usar dichos tipos de datos para resolver problemas. El hardware original de la máquina más los programas para implantar tipos de datos más complejos que los que proporciona el hardware, pueden pensarse como una máquina "mejor" que la integrada solamente por el hardware. El programador de la máquina origina no tiene que preocuparse por la manera en que está diseñada la computadora ni por cuáles circuitos se usan para ejecutar cada instrucción. El programador sólo necesita saber cuáles instrucciones están disponibles y cómo pueden usarse. De manera similar, al programador que utiliza la máquina "extendida" (ésta consiste en hardware y software), o "computadora virtual", como suele conocerse, no necesita ocuparse de los detalles de cómo fueron implantados varios tipos de datos. Todo lo que tiene que conocer es cómo puede manipularse el programador.

Siendo así la ventaja de utilizar una estructura de datos abstracta es que el programador no tiene que ocuparse de los detalles de cómo se implementa el tipo de datos.

Tipos de datos abstractos

Una herramienta útil para especificar las propiedades lógicas de los tipos de datos es el tipo de datos *abstracto* o *ADT* (por sus siglas en inglés), el cual es, fundamentalmente, una colección de valores y un conjunto de operaciones con esos valores. La colección y las operaciones forman una construcción matemática que puede implantarse utilizando una estructura de datos particular, ya sea de hardware o de software. El término "tipo de datos abstracto" se refiere al concepto matemático básico que define el tipo de datos.

Al definir como concepto matemático un tipo de datos abstracto, no nos importa la eficiencia de tiempo o espacio. Estos son aspectos de la implantación. En realidad, la definición de un ADT no tiene nada que ver con las características de su implantación. Ni siquiera puede ser posible implantar un ADT en específico al usar una pieza de hardware o un sistema de software particular. Por ejemplo, ya vimos que el ADT *integer* no es universalmente implantable. Sin embargo, al especificar las

propiedades lógicas y matemáticas de un tipo de datos o estructura, el ADT es un principio muy útil para quienes realizan las implantaciones y una herramienta útil para los programadores que desean usar correctamente los tipos de datos.

Hay muchos métodos para especificar un ADT. El método que usamos es semi-formal y se parece mucho a la notación del lenguaje C aunque la amplía cuando es necesario. Para ilustrar tanto el concepto de ADT como nuestro método de especificación, considérese el ADT *RATIONAL*, que corresponde al concepto matemático de número racional. Un número racional es un número que puede expresarse como el cociente de dos enteros. Las operaciones que definimos con números racionales son la creación de un número racional a partir de dos enteros, la suma, la multiplicación y la prueba de igualdad. La especificación inicial de este ADT es la siguiente:

```

/* definición de valores */
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] <> 0;

/* definición de operadores */
abstract RATIONAL makerational(a,b)
int a,b;
precondition b <> 0;
postcondition makerational[0] == a;
makerational[1] == b;

abstract RATIONAL add(a,b) /* escrito como a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b) /* escrito como a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
mult[1] == a[1] * b[1];

abstract equal(a,b) /* escrito como a == b */
RATIONAL a,b;
postcondition equal == (a[0]*b[1] == b[0]*a[1]);

```

Un ADT consiste en dos partes: una definición de valor y una de operador. La de valor define la colección de valores para el ADT y consta de dos partes: una cláusula de definición y una de condición. Por ejemplo, la definición de valor para el ADT *RATIONAL* postula que un valor *RATIONAL* consiste en dos enteros, el segundo de los cuales es distinto de 0. Por supuesto, los dos enteros que forman un número racional son el numerador y el denominador. Usamos notación de arreglos (corchetes) para indicar las partes de un tipo abstracto.

Las palabras reservadas *abstract* *typedef* introducen una definición de valor y la palabra reservada *condition* se usa para especificar cualquier condición sobre el tipo definido. En esta definición, la condición especifica que el denominador no puede

ser 0. Se requiere la cláusula de definición aunque no es necesaria la de condición para todo ADT.

Después de la definición de valor sigue en forma inmediata la de operador. Cada operador se define como una función abstracta de tres partes: un encabezado, las precondiciones opcionales y las postcondiciones. Por ejemplo, la definición de operador del ADT *RATIONAL* incluye las operaciones de creación (*makerational*), suma (*add*), y multiplicación (*mult*) así como una prueba de igualdad (*equal*). Consideremos primero la especificación para la multiplicación pues esta es la más simple; contiene un encabezado y postcondiciones pero no precondiciones:

```
abstract RATIONAL mult(a,b) /* escrito como a*b */
RATIONAL a,b;
postcondition mult[0] == a[0]*b[0];
    mult[1] == a[1]*b[1];
```

El encabezado de esta definición consta de las primeras dos líneas, que son iguales al encabezado de una función en lenguaje C. La palabra reservada *abstract* indica que no se trata de una función en lenguaje C sino de la definición de operador de un ADT. El comentario que inicia con la palabra reservada nueva *written* indica otra forma de escribir la función.

La postcondición especifica qué hace la operación. En una postcondición, se usa el nombre de la función (en este caso *mult*) para denotar el resultado de la operación. Así, *mult[0]* representa el numerador y *mult[1]* el denominador del resultado, respectivamente. Esto es, especifica qué condiciones son verdaderas después de realizada la operación. En este ejemplo, la postcondición especifica que el numerador del resultado de una multiplicación racional es igual al producto entero de los numeradores de las dos entradas y el denominador es igual al producto entero de los dos denominadores.

La especificación para la suma (*add*) es sencilla y tan sólo afirma que:

$$\frac{a_0}{a_1} + \frac{b_0}{b_1} = \frac{a_0 * b_1 + a_1 * b_0}{a_1 * b_1}$$

La operación de creación (*makerational*) crea un número racional a partir de dos enteros y contiene el primer ejemplo de una precondición. En general las precondiciones especifican cualquier restricción que deba satisfacerse antes de aplicar la operación. En este ejemplo, la precondición plantea que *makerational* no puede aplicarse si el denominador es cero.

La especificación para igualdad (*equal*) es más significativa y compleja en concepto. En general, dos valores cualesquiera en un ADT son “iguales” si y sólo si lo son los valores de sus componentes. Ciertamente, por lo común suponemos que existe una operación de igualdad (y desigualdad) definida del modo anterior, por lo que no necesitamos definir en forma explícita un operador de igualdad. La operación de asignación (asignar el valor de un objeto al de otro) es otro ejemplo de una operación asumida que se da por supuesta la mayoría de las veces para un ADT y no se especifica de manera explícita.

Sin embargo, para algunos tipos de datos, podemos considerar iguales a dos valores con componentes desiguales. En efecto, esto sucede con los números racionales; por ejemplo, los números racionales $\frac{1}{2}$, $\frac{2}{4}$, $\frac{3}{6}$ y $\frac{18}{36}$ son iguales sin importar la desigualdad de sus componentes. Se consideran iguales dos números racionales cuando lo son sus componentes en la forma reducida. (Esto es, cuando el numerador y el denominador han sido ambos divididos por su máximo común divisor). Una manera de verificar la igualdad entre racionales consiste en expresarlos como fracciones reducidas y luego verificar la igualdad entre denominadores y numeradores. Otra manera es la de comprobar si los productos cruzados (es decir, la multiplicación del denominador de uno por el numerador del otro) son iguales. Este es el método que hemos usado para especificar la operación abstracta *equal*.

La especificación abstracta ilustra el papel de un ADT en cuanto definición puramente lógica de un nuevo tipo de datos. Dos pares ordenados son desiguales si lo son sus componentes considerados como colección de dos enteros, aunque pueden ser iguales si se consideran como números racionales. Es improbable que alguna implantación de números racionales pruebe la igualdad al formar, realmente, los productos cruzados, pues pueden ser demasiado grandes para representarlos como enteros en la máquina. Es más probable que una implantación reduzca primero las entradas a fracciones reducidas para verificar luego la igualdad de los componentes.

En efecto una implantación razonable insistiría en que *makerational*, *suma* y *mult* sólo produce números racionales en términos menores. Sin embargo, las definiciones matemáticas tales como la de tipos de datos abstractos no están relacionadas con los detalles de la implantación.

En realidad, el hecho de que puedan ser iguales dos racionales aun cuando sus componentes no lo son nos obliga a escribir de nuevo las postcondiciones para *makerational*, *add* y *mult*. Es decir, si

no es necesario que m_0 sea igual a $a_0 * b_0$ y que m_1 sea igual a $a_1 * b_1$, sólo que $m_0 * a_1 * b_1$ sea igual a $m_1 * a_0 * b_0$. Una especificación de un ADT más precisa para un *RATIONAL* es la siguiente:

```
/* definición de valores */
abstract typedef<int> int RATIONAL;
condition RATIONAL[1] <> 0;
/* definición de operadores */
abstract equal(a,b) /* escrito como a==b */
RATIONAL a,b;
postcondition equal == (a[0]*b[1] == b[0]*a[1]);

abstract RATIONAL makerational(a,b) /* escrito como [a,b] */
int a,b;
precondition b <> 0;
postcondition makerational[0]*b == a*makerational[1]
```

```

abstract RATIONAL add(a,b) /* escrito como a + b */;
RATIONAL a,b;
postcondition add == [a[0] * b[1] + b[0] * a[1], a[1]*b[1]];

abstract RATIONAL mult(a,b) /* escrito como a*b */;
RATIONAL a,b;
postcondition mult == [a[0] * b[0], a[1] * b[1]];

```

Aquí se define primero el operador *equal*, y se extiende el operador *==* para la igualdad de racionales usando la cláusula *escribe*. Este operador se usa después para especificar los resultados de las operaciones racionales subsecuentes (*add* y *mult*).

El resultado de la operación *makerrational* con los enteros *a* y *b* produce un racional que es igual a *a/b*, pero la definición no especifica los valores reales del numerador y denominador resultantes. La especificación para *makerrational* introduce también la notación *[a, b]* para el racional formado por *a* y *b*, la cual se usa después para la definición de *add* y *mult*.

Las definiciones de *add* y *mult* especifican que sus resultados son iguales a los resultados no reducidos de la operación correspondiente, pero los componentes individuales no son necesariamente iguales.

Nótese otra vez que al definir esos operadores no especificamos cómo deben calcularse, sino sólo cuáles deben ser sus resultados. Cómo se calculan 1, está determinado por su implantación y no por su especificación.

Secuencias como definiciones de valor

Al desarrollar las especificaciones para varios tipos de datos, usamos a menudo notación teórica de conjuntos para especificar los valores de un ADT. En particular, es de gran ayuda usar la notación de secuencias matemáticas que introducimos ahora.

Una *secuencia* sólo es un conjunto ordenado de elementos. Una secuencia *S* se escribe, a veces, numerando sus elementos, por ejemplo

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

Si *S* contiene *n* elementos, se dice que *S* tiene longitud *n*. Suponemos que existe una función de longitud tal que es la longitud de la secuencia *S*. Suponemos también que existe una función *first* (*S*), que regresa el valor del primer elemento de *S* (*s₀* en el ejemplo anterior) y una función *last* (*S*) que regresa el valor del último elemento de *S* (*s_{n-1}* en el ejemplo anterior). Hay una secuencia especial de longitud 0, llamada *nilseq*, que no contiene elementos. *last* (*nilseq*) y *first* (*nilseq*) quedan indefinidas.

Deseamos definir un ADT *stp1* cuyos valores sean secuencias de elementos. Si las secuencias pueden ser de longitud arbitraria y constan de elementos que son todos del mismo tipo *tp*, entonces *stp1* puede definirse como sigue:

```
abstract typedef <<tp>> stp1;
```

De otra manera, podríamos definir un ADT *stp2*, cuyos valores fueran secuencias de longitud fija de elementos de tipos específicos. En tal caso, la especificaríamos.

```
abstract typedef <tp0, tp1, tp2, ..., tpn> stp2;
```

Claro que podríamos querer especificar una secuencia de longitud fija, con todos sus elementos de un mismo tipo. En ese caso escribiríamos:

```
abstract typedef <<tp,n>> stp3;
```

En este caso *stp3* representa una secuencia de longitud *n*, donde todos sus elementos son del mismo tipo, *tp*.

Por ejemplo, usando la notación anterior podemos definir los siguientes tipos:

```

abstract typedef <<int>> intseq; /* secuencia de enteros de
                                         cualquier longitud */
abstract typedef <integer, char, float> seq3; /* secuencia de longitud 3
                                         consistente de un entero
                                         un carácter y un punto flotante */

```

```
abstract typedef <<int,10>> intseq; /* secuencia de 10 enteros */

```

```
abstract typedef <<,2>> pair; /* secuencia arbitraria de
                                         longitud 2 */

```

Dos secuencias son *iguales* si cada elemento de la primera es igual al elemento correspondiente de la segunda. Una *subsecuencia* es una porción contigua de una secuencia. Si *S* es una secuencia, la función *sub* (*S*, *i*, *j*) se refiere a la subsecuencia de *S* que comienza en la posición *i* en *S* y consiste en *j* elementos consecutivos. Así, si *T* es igual a *sub* (*S*, *i*, *k*) y *T* es la secuencia $\langle t_0, t_1, \dots, t_{k-1} \rangle$, tenemos que $t_0 = s_i$, $t_1 = s_{i+1}, \dots, t_{k-1} = s_i + k - 1$. Si *i* no está entre 0 y *len* (*S*) — *k*, entonces *sub* (*S*, *i*, *k*) se define como *nilseq*.

La concatenación de dos secuencias, escrita como *S* + *T*, es la secuencia que consiste en todos los elementos de *S* seguidos de todos los de *T*. A veces, es deseable especificar la inserción de un elemento en medio de una secuencia. *place* (*S*, *i*, *x*) se define como la secuencia *S* con el elemento *x* insertado inmediatamente después de la posición *i* (o en el primer elemento de la secuencia si *i* es —1). Todos los elementos subsecuentes se recorren una posición. Es decir, *place* (*S*, *i*, *x*) es igual a *sub* (*S*, 0, *i*) + $\langle x \rangle$ + *sub* (*S*, *i* + 1, *len* (*S*) — *i*).

La supresión de un elemento de una secuencia puede especificarse de dos maneras. Si *x* es un elemento de la secuencia *S*, *S* — $\langle x \rangle$ representa la secuencia *S* sin todas las ocurrencias del elemento *x*. La secuencia *delete* (*S*, *i*) es igual a *S* con el ele-

mento de la posición i borrado. $\text{delete}(S, i)$ puede también escribirse en términos de otras operaciones como $\text{sub}(S, 0, i) + \text{sub}(S, i + 1, \text{len}(S) - i)$.

Un ADT para cadenas de caracteres de longitud variable

Como ejemplo del uso de la notación de secuencias en la definición de un ADT, desarrollamos la especificación de un ADT para cadenas de caracteres de longitud variable. Existen cuatro operaciones básicas (sin tomar en cuenta la asignación y la igualdad) que se incluyen con frecuencia en los sistemas que mantienen tales cadenas:

- length* función que regresa la longitud actual de la cadena
- concat* función que regresa la concatenación de dos cadenas de entrada
- substr* función que regresa una subcadena de una cadena dada
- pos* función que regresa la primera posición de una cadena como subcadena de otra

```
abstract typedef <<char>> STRING;  
  
abstract length(s)  
STRING s;  
postcondition length == len(s);  
  
abstract STRING concat(s1,s2)  
STRING s1,s2;  
postcondition concat == s1 + s2;  
  
abstract STRING substr(s1,i,j)  
STRING s1;  
int i,j;  
precondition 0 <= i < len(s1);  
0 <= j <= len(s1) - i;  
postcondition substr == sub(s1,i,j);  
  
abstract pos(s1,s2)  
STRING s1,s2;  
postcondition /*lastpos = len(s1) - len(s2) */  
((pos == -1) && (for(i = 0;  
i <= lastpos; i++)  
(s2 <> sub(s1,i,len(s2)))))  
||  
((pos >= 0) && (pos <= lastpos)  
&& (s2 == sub(s1,pos,len(s2)))  
&& (for(i = 1; i < pos; i++)  
(s2 <> sub(s1,i,len(s2)))));
```

La postcondición para *pos* es compleja e introduce novedades en la notación, por lo que la reconsideraremos aquí. Primero, nótese que el contenido del comenta-

rio inicial tiene la forma de una instrucción de asignación en lenguaje C. Esto indica tan sólo que queremos definir el símbolo *lastpos* como representación del valor de $\text{len}(s1) - \text{len}(s2)$ para usarlo dentro de la postcondición y simplificar la apariencia de la condición. Aquí, *lastpos* representa el máximo valor posible del resultado (esto es, la última posición de *s1* en donde puede comenzar una subcadena de longitud igual a la de *s2*). *lastpos* se usa dos veces dentro de la postcondición misma. Podríamos haber escogido en ambos casos el uso de la expresión $\text{len}(s1) - \text{len}(s2)$, que es más grande, pero elegimos un símbolo más pequeño (*lastpos*) para tener más claridad.

La postcondición misma afirma que debe cumplirse una de dos condiciones. Las dos condiciones, separadas por el operador $\|$, son las siguientes:

1. El valor de la función (*pos*) es -1 y *s2* no aparece como subcadena de *s1*.
2. El valor de la función se encuentra entre 0 y *lastpos*, *s2* aparece como subcadena de *s1* y comienza en la posición del valor de la función; *s2* no aparece como subcadena de *s1* en ninguna posición anterior.

Observe el uso de un pseudolazo *for* en una condición. La condición

```
for (i = x; i <= y; i++)  
(condition(i))
```

es verdadera si *condition(i)* es verdadera para todo *i* desde *x* hasta *y*, inclusive. También es verdadera si *x > y*. En otro caso, la condición *for* completa es falsa.

Tipos de datos en lenguaje C

El lenguaje C contiene cuatro tipos de datos básicos: *int*, *float*, *char* y *double*. En la mayoría de las computadoras estos cuatro tipos son nativos del hardware de la máquina. Ya vimos cómo pueden implantarse en el hardware enteros, valores de punto flotante y caracteres. Una variable *double* es un número de punto flotante de doble precisión. Existen tres calificadores que pueden aplicarse a enteros: *short* (corto), *long* (largo) y *unsigned* (sin signo). Una variable de entero *short* o *long* se refiere al tamaño máximo del valor de la variable. El tamaño máximo real que implican *short int*, *long int*, o *int* varía de acuerdo con la máquina. Un entero sin signo es un entero que siempre es positivo y se rige por las leyes del módulo 2^n , donde *n* es el número de bits en un entero.

Una declaración de variable en el lenguaje C hace dos especificaciones. Primera: especifica cuanta memoria debe apartarse para objetos declarados con ese tipo. Por ejemplo, una variable de tipo *int* debe tener suficiente espacio para el mayor valor posible de un entero. Segunda: especifica cómo han de interpretarse los datos representados por cadenas de bits. Los mismos bits en una localidad de memoria específica pueden interpretarse como entero o como número de punto flotante, lo cual da lugar a dos valores numéricos por completo diferentes.

Una declaración de variable especifica qué debe reservarse memoria para el objeto del tipo especificado y que podemos referirnos a dicho objeto por medio del identificador de variable especificado.

Apunadores en lenguaje C

En realidad, el lenguaje C permite al programador referirse tanto a las localidades como a los objetos (esto es, a los contenidos de localidades) mismos. Por ejemplo, si se declara a *x* como entero, *&x* se refiere a la localidad que se ha reservado para *x*. A *&x* se le conoce como *apuntador*.

Es posible declarar una variable cuyo tipo de datos sean apuntador y cuyos valores posibles sean localidades de memoria. Por ejemplo, las declaraciones

```
int *pi;
float *pf;
char *pc;
```

declaran tres variables de tipo apuntador: *pi* apunta a un entero, *pf* apunta a un número de punto flotante y *pc* apunta a un carácter. El asterisco indica que los valores de las variables declaradas son apuntadores de los tipos especificados en la declaración más que objetos de ese tipo.

En muchos aspectos, un apuntador es similar a cualquier otro tipo de dato en lenguaje C. El valor de un apuntador es una localidad de memoria de la misma manera que el valor de un entero es un número. Los valores del apuntador pueden asignarse como cualesquiera otros. Por ejemplo, la instrucción *pi = &x* asigna el apuntador del entero *x* a la variable apuntador *pi*.

La notación **pi* en lenguaje C alude al entero en la localidad referida por el apuntador *pi*. La instrucción *x = *pi* asigna el valor de este entero a la variable de entero *x*.

Note que el lenguaje C recalca que una declaración de un apuntador especifica el tipo de datos al cual apunta. En las declaraciones anteriores, cada una de las variables *pi*, *pf* y *pc* son apuntadores a un tipo de datos específico: *int*, *float* y *char*, respectivamente. El tipo de *pi* no es, simplemente, un “apuntador” sino un “apuntador a un entero”. De hecho, los tipos de *pi* y *pf* son diferentes: *pi* es un apuntador a un entero y *pf* es un apuntador a un número de punto flotante. Cada tipo *dt* de datos en el lenguaje C genera otro tipo de datos, *pdt*, llamado “apuntador a *dt*”. Llamamos a *dt* el *tipo base* de *pdt*.

La conversión de *pf* del tipo “apuntador a un número de punto flotante” al tipo “apuntador a un número entero” puede hacerse al escribir

```
pi = (int *)pf; /* convertir pf, que es apuntador a float, en apuntador a int */
```

donde el prefijo (*int **) convierte el valor de *pf* al tipo “apuntador a un *int*” o “*int ***”.

La importancia de cada apuntador asociado con un tipo de datos particular se aclara cuando revisamos las herramientas aritméticas que brinda el lenguaje C para los apuntadores. Si *pi* es un apuntador a un entero, entonces *pi + 1* es el apuntador al entero que sigue inmediatamente al entero **pi* en la memoria, *pi - 1* es el apunta-

dor al entero que precede en forma inmediata a **pi*, *pi + 2* es el apuntador al segundo entero que sigue a **pi* y así sucesivamente. Por ejemplo, supóngase que una máquina particular utiliza direccionamiento de byte, un entero requiere cuatro bytes y el valor de *pi* es 100 (es decir, si apunta al entero **pi* en la localidad 100). Entonces el valor de *pi - 1* es 96, el de *pi + 1* es 104 y el de *pi + 2*, 108. El valor de *(pi - 1)* es el de los contenidos de los cuatro bytes 96, 97, 98 y 99 interpretados como enteros; el de *(pi + 1)* es el contenido de los bytes 104, 105, 106 y 107 interpretados como enteros; y el valor de *(pi + 2)* es el entero que está en los bytes 108, 109, 110 y 111.

De manera parecida, si el valor de la variable *pc* es 100 (recuérdese que *pc* es un apuntador a *char*) y un carácter tiene un byte de longitud, *pc - 1* refiere a la localidad 99, *pc + 1* a la 101, y *pc + 2* a la 102. Así, el resultado de la aritmética de apuntadores en el lenguaje C depende del tipo base del apuntador.

Observe también la diferencia entre **pi + 1*, que refiere al 1 sumado al entero **pi*, y *(*pi + 1)* que refiere al entero que sigue al de la localidad *pi*.

Un área en la cual desempeñan un papel prominente los apuntadores en el lenguaje C es en la transmisión de parámetros a funciones. Por lo común, los parámetros se transmiten *por valor* a una función en el lenguaje C, es decir, se copian los valores transmitidos en los parámetros de la función llamada en el momento en que se invoca. Si cambia el valor de un parámetro dentro de la función, no cambia su valor en el programa que la llama. Por ejemplo, considérese el siguiente fragmento y función de programa (el número de línea solamente es una guía):

```
1. x = 5;
2. printf("%d\n", x);
3. funct(x);
4. printf("%d\n", x);
5. funct(y);
6. int y;
7. {
8.     y += x;
9.     printf("%d\n", y);
10. } /* fin de funct */
```

La línea 2 imprime 5 y después 3 llama a *funct*. El valor de *x*, que es 5, se copia en *y* y comienza la ejecución de *funct*. Después, la línea nueve imprime 6 y regresa *funct*. Sin embargo, cuando la línea 8 incrementa el valor de *y*, el valor de *x* permanece invariable. Así, la línea 4 imprime 5; *x* y *y* refieren a dos variables diferentes que suceden de que tienen el mismo valor al principio de *funct*. *y* puede cambiar independientemente de *x*.

Si deseamos usar *funct* para modificar el valor de *x*, debemos trasmitir la dirección de *x* de la siguiente manera:

```
1. x = 5;
2. printf("%d\n", x);
3. funct(&x);
4. printf("%d\n", x);
...
```

```

5   funct(py)
6   int *py;
7   {
8     ++(*py);
9     printf("%d\n", *py);
10 } /* fin de funct */

```

La línea 2 imprime nuevamente 5 y la línea 3 llama a *funct*. Ahora, sin embargo, el valor transferido no es el valor entero de *x* sino el valor apuntador *&x*. Este es la dirección de *x*. El parámetro de *funct* no es más y de tipo *int*, sino *py* de tipo *int**. (Es conveniente nombrar a las variables apuntadores comenzando con la letra *p* para recordar tanto al programador como al lector del programa que se trata de un apuntador. Sin embargo, no es un requisito del lenguaje C, así que hubiéramos podido llamar *y* al parámetro apuntador.) La línea 8 ahora aumenta al entero en la localidad *py*; *py*, sin embargo, no cambia y conserva su valor inicial *&x*. Así, *py* apunta al entero *x* de tal manera que cuando aumenta **py*, aumenta *x*. La línea 9 imprime 6 y cuando regresa *funct*, la línea 4 imprime también 6. Los apuntadores son el mecanismo usado en el lenguaje C para permitir a una función llamada modificar las variables de la función que llama.

Estructuras de datos en el lenguaje C

Un programador en lenguaje C puede pensar que este lenguaje define una máquina nueva con sus propias capacidades, tipos de datos y operaciones. El usuario puede plantear una solución a un problema en términos de las más útiles construcciones en el lenguaje C en vez de hacerlo en términos de las construcciones de lenguajes de máquina de bajo nivel. Por lo tanto, pueden resolverse los problemas con mayor facilidad debido a que hay un amplio conjunto de herramientas disponibles.

El estudio de las estructuras de datos implica, en consecuencia, dos metas complementarias. La primera es la de identificar y desarrollar las entidades y operaciones matemáticas más útiles y determinar qué clase de problemas pueden resolverse mediante ellas. La segunda es la de determinar las representaciones para esas entidades abstractas y para implantar las operaciones con esas representaciones concretas. La primera, concibe a los tipos de datos de alto nivel como una herramienta que puede usarse para resolver otros problemas; la segunda concibe la implantación de un tipo de datos como un problema a resolver por medio de los tipos de datos ya existentes. Al determinar las representaciones para las entidades abstractas, debemos ser cuidadosos para especificar cuáles herramientas tenemos para la construcción. Por ejemplo, debe establecerse si disponemos del lenguaje C en su totalidad o si estamos restringidos a las herramientas del hardware de una máquina en particular.

En las próximas dos secciones de este capítulo estudiamos varias estructuras de datos que ya existen en el lenguaje C: el arreglo y la estructura. Describimos las herramientas disponibles en el lenguaje C para utilizar esas estructuras. También nos enfocamos a la definición abstracta de esas estructuras de datos y cómo pueden ser útiles para la solución de problemas. Finalmente, examinamos cómo pueden implantarse si no estuvieran disponibles en el lenguaje C. (Aunque un programador

en el lenguaje C puede usar simplemente las estructuras de datos tal y como están definidas en el lenguaje sin importarle la mayoría de esos detalles de implantación).

En lo que resta del libro, desarrollamos estructuras de datos más complejas y mostramos su utilidad en la solución de problemas. Mostramos también cómo implantar dichas estructuras usando las que ya estén disponibles en el lenguaje C. Como los problemas que aparecen mientras intentamos implantar estructuras de datos de alto nivel son bastante complejos, esto nos permitirá también investigar con más profundidad el lenguaje C y obtener gran experiencia en su uso.

A menudo, no existe una implantación de software ni de hardware, que pueda modelar un concepto matemático por completo. Por ejemplo, es imposible representar números enteros arbitrariamente grandes en una computadora, debido a que el tamaño de la memoria es finito. En consecuencia, no es el tipo de datos "entero" el que representa el hardware sino el tipo "entero entre *x* y *y*", donde *x* y *y* son los enteros más pequeño y más grande, respectivamente, representables por la máquina.

Es importante reconocer las limitaciones de una implantación particular. A menudo, será posible representar varias implantaciones de un mismo tipo de datos, cada una con su fuerza y debilidad. Una implantación específica puede ser mejor que otra para una aplicación particular y el programador tiene que estar enterado de las posibles ventajas y desventajas que conlleva.

Una consideración importante en cualquier implantación es su eficiencia. En realidad, la razón por la cual las estructuras de datos de alto nivel que analizamos aquí no están construidas en el lenguaje C se debe a la sobrecarga significativa que acarrearían. Hay lenguajes de bastante más alto nivel que el lenguaje C que tienen incluidos estos tipos de datos, pero muchos son ineficientes, por lo que no se ha difundido su empleo.

La eficiencia se mide por lo común mediante dos factores: espacio y tiempo. Si una aplicación particular depende demasiado de la manipulación de estructuras de datos de alto nivel, la velocidad a la cual pueden ejecutarse, será el factor determinante en la velocidad de toda la aplicación. De manera parecida, si un programa utiliza un gran número de estas estructuras, una implantación que use una cantidad excesiva de espacio para representarlas será impracticable. Por desgracia, hay por lo general un compromiso entre las dos eficiencias, de manera que una implantación que es rápida usa más memoria que una lenta. La elección de la implantación, en tal caso, implica una cuidadosa evaluación de las ventajas y desventajas entre las diversas posibilidades.

EJERCICIOS

- 1.1.1. En el libro se hace una analogía entre la longitud de una recta y el número de bits de información en una cadena de bits. ¿En qué sentido es inadecuada dicha analogía?
- 1.1.2. Determine qué tipos de datos del hardware están disponibles en la computadora de su instalación particular y qué operaciones pueden ejecutarse con ellos.
- 1.1.3. Pruebe que hay 2^n maneras diferentes de configurar *n* conmutadores (que tienen sólo dos posiciones posibles). Suponga que queremos tener *m* configuraciones. ¿Cuántos conmutadores serían necesarios?

1.1.4. Interprete las siguientes combinaciones de bits como enteros binarios positivos, como enteros binarios en notación de complementos a 2 y como enteros decimales codificados en binario. Si una combinación no puede interpretarse como un entero decimal codificado en binario, explique la razón.

- (a) 10011001
- (b) 1001
- (c) 000100010001
- (d) 01110111
- (e) 01010101
- (f) 100000010101

1.1.5. Escriba las funciones en lenguaje C *add*, *subtract*, y *multiply*, que lean dos cadenas de 0 y 1 que representan enteros binarios no negativos e imprima la cadena representando su suma, su diferencia y su producto, respectivamente.

1.1.6. Suponga que en una computadora ternaria la unidad básica de memoria es un trit (dígito ternario) en lugar de un bit. Un trit puede tener tres posibles posiciones (0, 1 y 2) en lugar de dos (0 y 1). Muestre cómo pueden representarse enteros negativos en notación ternaria usando tales trits por un método análogo al de la notación binaria usando bits. ¿Hay algún entero no negativo que pueda representarse usando notación ternaria y trits pero que no se pueda usando notación binaria y bits? ¿Hay alguno con el que ocurra lo contrario? ¿Por qué son más comunes las computadoras binarias que las ternarias?

1.1.7. Escriba un programa en lenguaje C para leer una cadena de 0 y 1 que representen un entero positivo en binario e imprima una cadena de 0, 1 y 2 que represente el mismo número en notación ternaria (véase el ejercicio anterior). Escriba otro programa en lenguaje C para leer un número ternario y escribir su equivalente en binario.

1.1.8. Escriba una especificación de ADT para números complejos $a + bi$, donde $\text{abs}(a + bi)$ es $\sqrt{a^2 + b^2}$, $(a + bi) + (c + di)$ es $(a + c) + (b + d)i$, $(a + bi) * (c + di)$ es $(a * c - b * d) + (a * d + b * c)i$, y $-(a + bi)$ es $(-a) + (-b)i$.

1.2. ARREGLOS EN LENGUAJE C

En ésta y la siguiente sección examinamos varias estructuras de datos que son parte valiosa del lenguaje C. Veremos cómo usar estas estructuras y cómo implantarlas. Esas estructuras son tipos de datos *compuestos* o *estructurados*; es decir, están conformadas por estructuras de datos más simples que existen en el lenguaje. El estudio de ese tipo de estructuras de datos trae consigo un análisis de cómo combinar estructuras simples para formar complejas y cómo extraer un componente específico de las estructuras compuestas. Suponemos que el lector haya visto ya esas estructuras de datos en un curso introductorio de programación en lenguaje C y que sepa cómo se definen y usan en el lenguaje C. Por consiguiente, no haremos hincapié, en estas secciones, en los diversos detalles asociados con esas estructuras, sino que destacaremos aquellas características interesantes desde el punto de vista de la estructura de datos.

El primero de estos tipos de datos es el *arreglo*. La forma más simple de arreglo es un *arreglo unidimensional* que puede definirse de manera abstracta como un conjunto finito ordenado de elementos homogéneos. Por “finito” entendemos que hay un número específico de elementos en el arreglo; número que puede ser grande o pequeño, pero debe existir. Por “ordenado” entendemos que los elementos están

dispuestos de tal manera que hay un elemento cero, un elemento primero, uno segundo, un tercero y así sucesivamente. Por “homogéneo”, entendemos que todos los elementos del arreglo son del mismo tipo. Por ejemplo, un arreglo puede contener elementos enteros o caracteres, pero no ambos.

Sin embargo, la especificación de la forma de una estructura de datos no permite describir por completo la estructura. Tenemos que explicar también cómo se tiene acceso a la misma. Por ejemplo, en lenguaje C, la declaración

```
int a[100];
```

especifica un arreglo de 100 enteros. Las dos operaciones básicas que admiten un arreglo son la *extracción* y *almacenamiento*. La extracción es una función que acepta un arreglo *a* y un índice *i* y regresa un elemento del arreglo. En lenguaje C, el resultado de esta operación se denota por *a[i]*. El almacenamiento acepta un arreglo *a*, un índice *i* y un elemento *x*. En lenguaje C esta operación se denota por medio de la instrucción de asignación *a[i] = x*. Las operaciones se definen por la regla según la cual después de ejecutada la instrucción de asignación anterior, el valor de *a[i]* es *x*. Antes de que se haya asignado un valor a un elemento del arreglo, su valor está indefinido y no está permitido referirnos a él en una expresión.

El índice más pequeño de un arreglo se le conoce como su *límite inferior*, el cual en el lenguaje C siempre es 0, y el más grande se conoce como *límite superior*. Si *límite 1* es el límite inferior de un arreglo y *límite 2* el superior, el número de elementos del arreglo, llamado su *rango*, está dado por *límite 2 — límite 1 + 1*. Por ejemplo, en el arreglo *a*, declarado antes, el límite inferior es 0, el superior 99 y el rango es 100.

Una característica importante de los arreglos en lenguaje C es que ni el límite superior ni el inferior (y, por tanto, tampoco el rango) pueden cambiarse durante la ejecución de un programa. El límite superior siempre se fija en 0 y el superior se fija en el momento en que se escribe el programa.

Una técnica muy útil es la de declarar un límite como identificador constante, de tal manera que se minimiza el trabajo requerido para modificar el tamaño de un arreglo. Por ejemplo, considérese el siguiente fragmento de programa para declarar e inicializar un arreglo:

```
int a[100];
for(i = 0; i < 100; a[i++] = 0);
```

Para cambiar el tamaño del arreglo por uno menor (o mayor), debe cambiarse la constante 100 en dos lugares; una en las declaraciones y otra en la instrucción *for*. Considérese la siguiente opción equivalente:

```
#define NUMELTS 100
int a[NUMELTS];
for(i = 0; i < NUMELTS; a[i++] = 0);
```

Ahora, sólo precisamos un cambio único en la definición de la constante para cambiar el límite superior.

El arreglo como ADT

Podemos representar un arreglo como un tipo de datos abstracto al ampliar un poco las convenciones y la notación discutidas antes. Suponemos que existe la función *type(arg)*, que regresa el tipo de su argumento *arg*. Claro que tal función no puede existir en el lenguaje C, puesto que éste no puede determinar de manera dinámica el tipo de una variable. Sin embargo, como aquí no nos importa la implantación sino la especificación, podemos usar tal función.

Digamos que *ARRTYPE(ub, eltype)* denota el ADT correspondiente al arreglo del lenguaje del tipo *eltype array[ub]*. Este es nuestro primer ejemplo de un ADT con parámetros, en el cual el ADT preciso está determinado por los valores de uno o más parámetros. En este caso, *ub* y *eltype* son los parámetros; nótese que *eltype* es un indicador de tipo, no un valor. Podemos considerar ahora cualquier arreglo unidimensional como una entidad de tipo *ARRTYPE*. Por ejemplo, *ARRTYPE(10, int)* representaría el tipo del arreglo *x* en la declaración *int x[10]*. La especificación es la siguiente:

```
abstract typedef <<eltype, ub>> ARRTYPE(ub, eltype);
condition type(ub) == int;

abstract eltype extract(a,i) /* escrito como a[i] */
ARRTYPE(ub, eltype) a;
int i;
precondition 0 <= i < ub;
postcondition extract == a[i];

abstract store(a, i, elt) /* escrito como a[i] = elt */
ARRTYPE(ub, eltype) a;
int i;
eltype elt;
precondition 0 <= i < ub;
postcondition a[i] == elt;
```

La operación *store* es nuestro primer ejemplo de una operación que modifica uno de sus parámetros; en este caso, el arreglo *a*. En la postcondición se indica al especificar el valor del elemento del arreglo al cual se ha asignado *elt*. Suponemos que todos los parámetros conservan el mismo valor después de aplicada la operación en una postcondición, como antes, a menos que se especifique un valor modificado en la postcondición. No es necesario especificar que no cambia ninguno de los valores de este tipo. Así, en este ejemplo no cambia el valor de ningún elemento del arreglo, excepto el elemento al que se le asignó *elt*.

Nótese que una vez definida la operación *extract*, junto con su notación de corchetes *a[i]*, puede usarse esta notación en la postcondición para especificar después la operación *store*. Sin embargo, dentro de la postcondición de *extract*, tiene que usarse la notación de secuencias subindexadas, pues estamos definiendo la notación de corchetes para arreglos.

Uso de arreglos unidimensionales

Cuando es necesario guardar un gran número de objetos en la memoria y referirnos a todos de la misma manera usamos un arreglo unidimensional. Veamos cómo se aplican estos dos requerimientos a situaciones prácticas.

Supóngase que queremos leer 100 enteros, encontrar su promedio y determinar cuánto se desvía cada entero de ese promedio. El siguiente programa lo hace:

```
#define NUMELTS 100
aver()
{
    int num[NUMELTS]; /* arreglo de números */
    int i;
    int total; /* suma de los números */
    float avg; /* promedio de los números */
    float diff; /* diferencia entre cada
    /* número y el promedio */
    total = 0;
    for (i = 0; i < NUMELTS; i++) {
        /* leer el número en el arreglo y sumarlo */
        scanf("%d", &num[i]);
        total += num[i];
    } /* fin de for */
    avg = total / NUMELTS; /* cálculo del promedio */
    printf("\nindiferencia del número"); /* imprime encabezado */
    /* imprime cada número y su diferencia */
    for (i = 0; i < NUMELTS; i++) {
        diff = num[i] - avg;
        printf("\n %d %d", num[i], diff);
    } /* fin de for */
    printf("\n el promedio es: %d", avg);
} /* fin de aver */
```

Este programa utiliza dos grupos de 100 números. El primero es el conjunto de los enteros de entrada y está representado por el arreglo *num*, el segundo es el conjunto de las diferencias, resultado de la asignación sucesiva de valores a la variable *diff* en el segundo ciclo. Surge la pregunta: ¿por qué se usa un arreglo para guardar todos los valores del primer grupo simultáneamente, mientras que sólo se usa una única variable por vez para guardar un valor del segundo grupo?

La respuesta es sencilla. Cada diferencia se computa e imprime sin que se necesite otra vez. De modo que la variable *diff* puede volverse a usar para la diferencia del entero siguiente y su promedio. Sin embargo, todos los enteros originales, que son los valores del arreglo *num*, tienen que guardarse en memoria. Aunque cada uno puede sumarse a *total* como entrada, se tiene que retener hasta después de que se ha calculado el promedio para que lo use el programa que calcula la diferencia entre el valor original y el promedio. Por esto se usa un arreglo.

Claro, podrían haber usado 100 variables por separado para retener los enteros. La ventaja de un arreglo, sin embargo, es que permite al programador declarar

solamente un identificador y obtener así mucho espacio. Además, junto con el ciclo FOR, permite al programador también referirse a cada elemento del grupo de una manera uniforme en lugar de obligarlo a codificar la instrucción de la siguiente manera:

```
scanf("%d%d%d...%d", &num0, &num1, &num2, ..., &num99);
```

Un elemento particular del arreglo puede recuperarse directamente por medio de su índice. Por ejemplo, supóngase que una compañía usa un programa en el cual se declara un arreglo por medio de:

```
int sales[10];
```

El arreglo contendrá los precios de venta por un periodo de diez años. Supóngase que cada línea de entrada del programa contiene un entero entre 0 y 9, el cual representa tanto el año como el precio de venta para ese año y que se desea leer el precio de venta en el elemento apropiado del arreglo. Podemos hacerlo al ejecutar la instrucción

```
scanf("%d%d", &yr, &sales[yr]);
```

dentro de un ciclo. En esta instrucción, un elemento particular del arreglo se admite directamente mediante su índice. Considérese la situación cuando se han declarado diez variables s_0, s_1, \dots, s_9 . Así después de ejecutar `scanf("%d", &yr)`, la cantidad de venta no puede leerse en la variable apropiada ni para establecer `yr` como el entero que representa el año, sin codificar algo parecido a lo siguiente:

```
switch(yr) {  
    case 0: scanf("%d", &s0); break;  
    case 1: scanf("%d", &s1); break;  
    case 2: scanf("%d", &s2); break;  
    case 3: scanf("%d", &s3); break;  
    case 4: scanf("%d", &s4); break;  
    case 5: scanf("%d", &s5); break;  
    case 6: scanf("%d", &s6); break;  
    case 7: scanf("%d", &s7); break;  
    case 8: scanf("%d", &s8); break;  
    case 9: scanf("%d", &s9); break;  
}
```

Lo cual es desastroso con 10 elementos: imagínese la inconveniencia si hubiera cientos o miles de elementos.

Implantación de arreglos unidimensionales

Un arreglo unidimensional puede implantarse fácilmente. La declaración

```
int b[100];
```

en lenguaje C, reserva 100 localidades sucesivas de memoria, cada una lo bastante grande para contener un solo entero. La dirección de la primera de esas localidades se conoce como *dirección base* del arreglo *b* y se denota por *base(b)*. Supóngase que el tamaño de cada elemento individual del arreglo es *esize*. Entonces, al referirnos al

elemento *b[0]* nos referimos a la localidad *base(b)*; cuando lo hacemos a *b[1]* lo hacemos al elemento en *base(b) + esize*; cuando es a *b[2]* lo hacemos al elemento en *base b + esize * esize * 2*. En general, referirse a *b[i]* remite al elemento en la localidad *base(b) + i * esize*. Así que es posible referirse a cualquier elemento en el arreglo, dado su índice.

En realidad, en el lenguaje C, un arreglo variable se implanta como un apuntador variable. El tipo de la variable *b* en la declaración anterior es “apuntador a un entero” o *int **. No aparece asterisco en la declaración pues los corchetes señalan de modo inmediato que la variable es un apuntador. La diferencia entre las declaraciones *int *b* e *int b[100]*, es que la última reserva también 100 localidades enteras comenzando desde la localidad *b*. En el lenguaje C, el valor de la variable *b* es *base(b)*, y el valor de la variable *b[i]*, donde *i* es un entero, es **(b + i)*. Recuérdese, de la sección 1, que como *b* es un apuntador a un entero, **(b + i)* es el valor del *i*-ésimo entero siguiente al de la localidad *b*. *b[i]*, el elemento de la localidad *base(b) + i * esize*, es equivalente al elemento apuntado por *b + i*, que es **(b + i)*.

En el lenguaje C todos los elementos de un arreglo tienen el mismo tamaño fijo y predeterminado. Algunos lenguajes de programación, sin embargo, permiten arreglos de objetos de tamaños diferentes. Por ejemplo, un lenguaje puede permitir arreglos de cadenas de caracteres de longitud variable. En tales casos, no puede usarse el método anterior para implantar el arreglo, lo cual se debe a que el método de cálculo de la dirección de un elemento específico del arreglo depende del conocimiento del tamaño fijo de cada elemento precedente (*esize*). Si no todos los elementos tienen el mismo tamaño, debe usarse una implantación diferente.

Un método para implantar un arreglo de elementos de tamaño variable es reservar un conjunto contiguo de localidades de memoria, cada una de las cuales conserva una dirección. El contenido de cada una de esas localidades es la dirección del elemento del arreglo de longitud variable en alguna otra porción de la memoria. Por ejemplo, la figura 1.2.1a muestra un arreglo de cinco cadenas de caracteres de longitud variable implantado según las dos maneras presentadas en la sección 1.1, para enteros de longitud variable. Las flechas indican las direcciones de otras porciones de memoria. El carácter ‘b’ indica un espacio en blanco. (Sin embargo, en el lenguaje C, las propias cadenas se implantan como arreglos, de tal manera que un arreglo de cadenas en realidad es un arreglo de arreglos: un arreglo bidimensional, no unidimensional.)

Como la longitud de cada dirección es fija, la localidad de la dirección de un elemento particular puede calcularse de la misma manera que un elemento de longitud fija, como en los ejemplos anteriores. Una vez que se conoce dicha localidad, su contenido puede usarse para determinar la localidad del elemento real del arreglo. Esto añade, desde luego, un nivel extra de direccionamiento indirecto al referirnos a un elemento del arreglo al incluir una referencia extra de memoria, lo que disminuye la eficiencia. Sin embargo, es el pequeño precio que debemos pagar por la conveniencia de poder mantener tal arreglo.

Un método similar para implantar un arreglo de elementos de tamaño variable es conservar todas las porciones de longitud fija de los elementos en un área contigua del arreglo, además de conservar las direcciones de la porción de longitud variable en el área contigua. Por ejemplo, en la implantación de cadenas de caracteres de longitud variable presentada en la sección previa, cada cadena contiene una por-

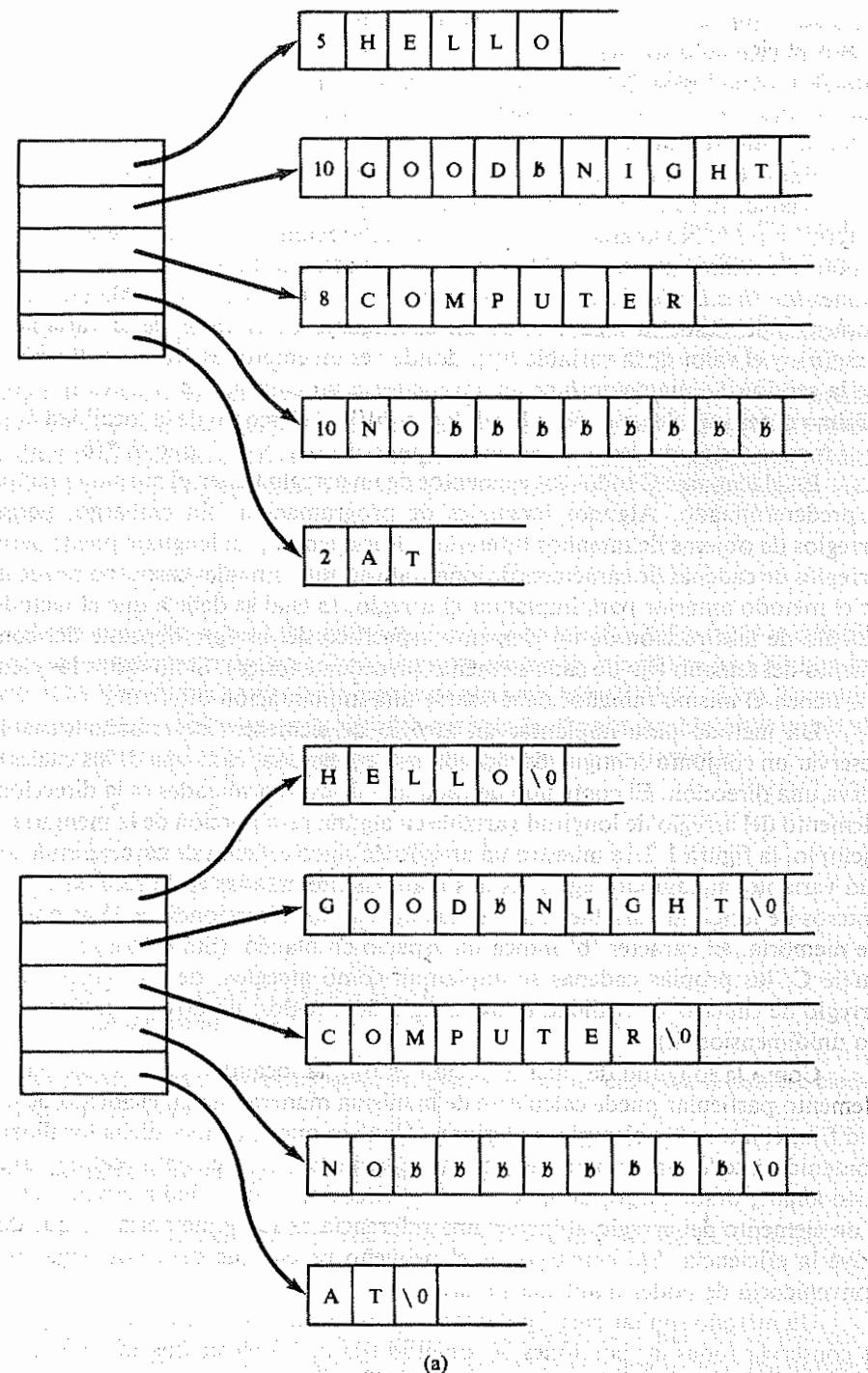


Figura 1.2.1. Implantaciones de un arreglo de cadenas de longitud variable.

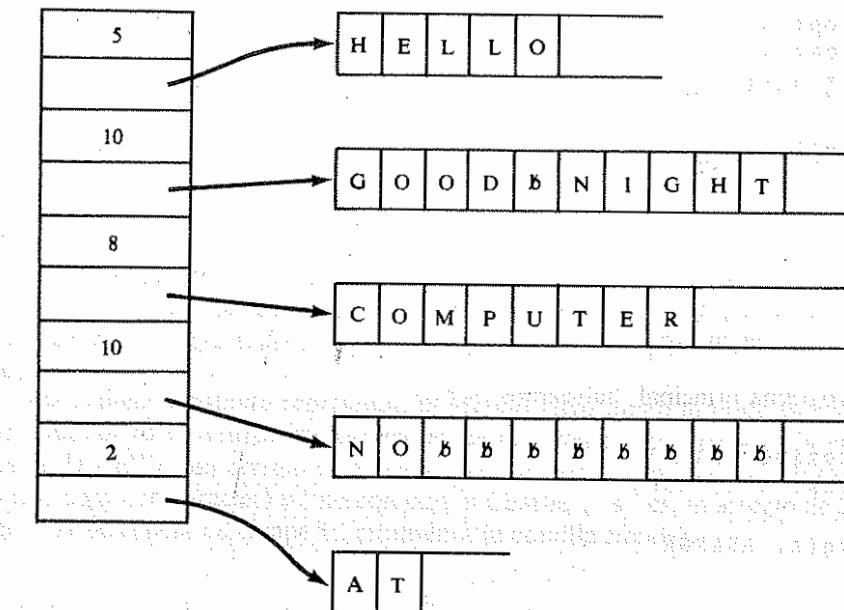


Figura 1.2.1. (cont.) Implantaciones de un arreglo de cadenas de longitud variable.

ción de longitud fija (un campo de un byte de largo) y una porción de longitud variable (la propia cadena de caracteres). Una implantación de un arreglo de cadenas de caracteres de longitud variable conserva el largo de la cadena junto con la dirección tal y como se muestra en la figura 1.2.1b. La ventaja de este método es que pueden examinarse las partes de un elemento que son de longitud fija sin hacer referencia adicional a la memoria. Por ejemplo, una función para determinar la longitud actual de una cadena de caracteres de longitud variable puede implantarse con una sola referencia a la memoria. A la información de longitud fija para un elemento del arreglo de longitud variable que está guardada en el área de memoria contigua al arreglo se le conoce, con frecuencia, como *encabezado*.

Arreglos como parámetros

Todos los parámetros de una función en el lenguaje C tienen que declararse dentro de la función. Sin embargo, el rango de un arreglo unidimensional como parámetro se especifica sólo en el programa principal, pues en el lenguaje C no se asigna memoria nueva a un arreglo como parámetro. Más bien los parámetros hacen referencia al arreglo original que fue asignado en el programa que llama la función. Por ejemplo, considérese la siguiente función para calcular el promedio de los elementos de un arreglo:

```

float avg(a, size)
float a[];           /*      no se indica rango      */
int size;
{
    int i;
    float sum;

    sum = 0;
    for (i=0; i < size; i++)
        sum += a[i];
    return(sum / size);
} /* fin de avg */

```

En el programa principal, habríamos escrito

```

#define ARANGE 100
float a[ARANGE];
...
avg(a, ARANGE);

```

Nótese que si se necesita en la función el rango del arreglo, tiene que transferirse por separado.

Como una variable arreglo en el lenguaje C es un apuntador, los parámetros del arreglo se transfieren *por referencia* y no por valor. Esto es, a la inversa que con las variables únicas transferidas por valor, los contenidos de un arreglo no se copian cuando se transfiere como parámetro en el lenguaje C.

En su lugar, se transfiere la dirección base del arreglo. Si una función que llama el programa contiene la llamada *funct(a)*, donde *a* es un arreglo y la función *funct* tiene el encabezado

```

funct(b)
int b[];
...
b[i] = x;

```

dentro de *funct* modifica el valor de *a[i]* en el interior de la función que llama. Dentro de *funct*, *b* se refiere al mismo arreglo de localidades que *a* en la función que llama.

Transferir un arreglo por referencia y no por valor es más eficiente en cuanto a espacio y a tiempo. Se elimina el tiempo requerido para copiar un arreglo entero cuando se llama una función. Se reduce también el espacio necesario para una segunda copia del arreglo en la función llamada, siendo sólo necesario contar con el espacio para una variable apuntador.

Cadenas de caracteres en el lenguaje C

Una *cadena* se define, en el lenguaje C, como un arreglo de caracteres. Cada cadena termina con el carácter *NULL* (nulo), que indica su final. Se denota una cadena constante mediante cualquier conjunto de caracteres entre dobles comillas. El carácter *NULL* se incluye en forma automática al final de una cadena de caracteres constante cuando se almacena. Dentro de un programa, el carácter *NULL* se representa por la secuencia de escape \ 0. Otras secuencias de escape que pueden usarse son \ n, para el carácter de línea nueva, \ t para el carácter tab, \ b para el espacio de retroceso, \ " para las comillas dobles, \ \ para la antidiagonal \ , \ ' para las comillas simples, \ r para el carácter de retorno de carro y \ f para el carácter avance de hoja.

Una cadena constante representa un arreglo cuyo límite inferior es 0 y cuyo límite superior es el número de caracteres de la cadena. Por ejemplo, la cadena "HOLA TERE" es un arreglo de 10 caracteres (el espacio en blanco y \ 0 cuentan cada uno como un carácter) y "no conozco el Charlie \ ' s" es un arreglo de 24 caracteres (la secuencia de escape \ ' representa la comilla simple).

Operaciones con cadenas de caracteres

Presentemos funciones en lenguaje C para implantar algunas operaciones primitivas con cadenas de caracteres. Suponemos las siguientes declaraciones para todas esas funciones

```

#define STRSIZE 80
char string[STRSIZE];

```

La primera función encuentra la longitud actual de una cadena.

```

strlen(string)
char string[];
{
    int i;

    for (i=0; string[i] != '\0'; i++)
        ;
    return(i);
} /* fin de strlen */

```

La segunda función acepta como parámetros dos cadenas. La función regresa un entero que indica la posición inicial de la primera ocurrencia del segundo parámetro de la cadena parámetro dentro del primero de la primer cadena. Si la segunda no existe dentro de la primera, el resultado es -1.

```

strpos(s1, s2)
char s1[], s2[];
{
    int len1, len2;
    int i, j1, j2;

    len1 = strlen(s1);
    len2 = strlen(s2);
    for (i=0; i+len2 <= len1; i++)
        for (j1=i, j2=0; j2 <= len2 && s1[j1] == s2[j2]; j1++, j2++)
            if (j2 == len2)
                return(i);
    return(-1);
} /* fin de strpos */

```

Otra operación común con cadenas es la concatenación. El resultado de concatenar dos cadenas consiste en los caracteres de la primera seguidos de los de la segunda. La siguiente función asigna a s1 el resultado de la concatenación de s1 y s2.

```

strcat(s1, s2)
char s1[], s2[];
{
    int i, j;

    for (i=0; s1[i] != '\0'; i++)
        ;
    for (j=0; s2[j] != '\0'; s1[i+j] = s2[j++])
        ;
} /* fin de strcat */

```

La última operación con cadenas que presentamos es la operación subcadena. substr(s1, i, j, s2) asigna a s2 j caracteres comenzando en s1[i].

```

substr(s1, i, j, s2)
char s1[], s2[];
int i, j;
{
    int k, m;

    for (k = i, m = 0; m < j; s2[m++] = s1[k++])
        ;
    s2[m] = '\0';
} /* fin de substr */

```

Arreglos bidimensionales

El tipo de componente de un arreglo puede ser otro arreglo. Por ejemplo, podemos definir:

- int a[3][5];

Lo anterior define un nuevo arreglo que contiene tres elementos, cada uno de los cuales es en sí mismo un arreglo que contiene cinco enteros. La fig. 1.2.2 ilustra dicho arreglo. Un elemento de ese arreglo se obtiene especificando dos índices: un número de renglón y un número de columna. Por ejemplo, el elemento sombreado en la fig. 1.2.2 está en el renglón 1 y en la columna 3 y podemos referirnos a él como *a[1][3]*. Un arreglo de este tipo se llama arreglo *bidimensional*. Al número de renglones o de columnas se conoce como el *rango* de la dimensión. En el arreglo *a*, el rango de la primera dimensión es 3 y el de la segunda es 5. Por lo que el arreglo *a* tiene 3 renglones y 5 columnas.

Un arreglo bidimensional ilustra con claridad las diferencias entre el punto de vista *lógico* y *físico* de los datos. Un arreglo bidimensional es una estructura de datos lógica que es muy útil en la programación y en la resolución de problemas. Por ejemplo, tal arreglo es útil para describir un objeto físicamente bidimensional, como un mapa o un tablero. También es útil para organizar un conjunto de valores que depende de dos entradas. Por ejemplo, un programa para una tienda de departamentos que tiene 20 sucursales, cada una de las cuales vende 30 artículos, podría incluir un arreglo bidimensional declarado por

```
int sales[20][30];
```

Cada elemento *sales[i][j]* representa la cantidad de artículos *j* vendidos en la sucursal *i*.

Sin embargo, aunque para el programador es conveniente concebir los elementos de un arreglo organizados en una tabla bidimensional (los lenguajes de programación incluyen en verdad herramientas para tratarlas como un arreglo bidimensional), el hardware de muchas computadoras no tiene tales herramientas. Un arreglo debe almacenarse en la memoria de una computadora, la cual, por lo regular, es lineal; esto es, la memoria de una computadora es, en esencia, un arreglo unidimensional. Una sola dirección (que puede concebirse como el subíndice de un arreglo unidimensional) se usa para localizar una localidad particular de la memoria. Para implantar un arreglo bidimensional, es necesario desarrollar un método para ordenar estos elementos en forma lineal y para transformar una referencia bidimensional en representación lineal.

Un método para representar en la memoria un arreglo bidimensional es la representación mediante *renglón-mayor*. Con esta representación, el primer renglón del arreglo ocupa el primer conjunto de localidades reservado para el arreglo, el segundo ocupa el siguiente conjunto y así sucesivamente. Puede haber también algunas localidades al principio del arreglo físico como encabezado, las cuales contienen los límites superior e inferior de las dos dimensiones. (Este encabezado no debe confundirse con los discutidos antes. Este es del arreglo completo mientras que los mencionados antes eran de los elementos individuales del arreglo). La figura 1.2.3 muestra la representación mediante renglón-mayor del arreglo bidimensional *a* declarado arriba y mostrado en la figura 1.2.2. Además, el encabezado no debe ser contiguo a los elementos del arreglo, aunque podría en su lugar contener la dirección del primer elemento del mismo. También, si los elementos del arreglo bidimensional son objetos de longitud variable, los propios elementos del área contigua pueden

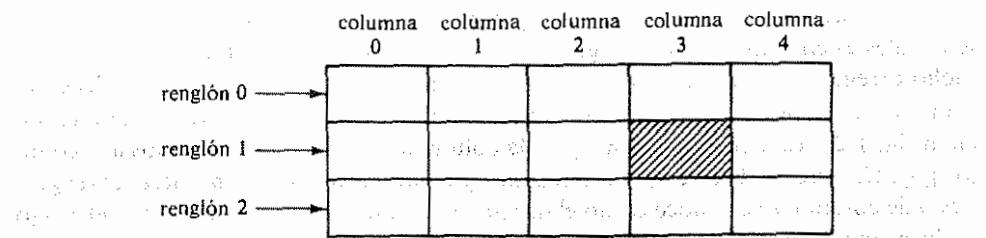


Figura 1.2.2 Arreglos bidimensionales.

Algunos lenguajes de programación permiten declarar un arreglo bidimensional que contiene referencias a otros objetos. Por ejemplo, se puede declarar un arreglo que contiene direcciones de memoria para que el programa pueda contener las direcciones de esos objetos en una forma similar a la que muestra la figura 1.2.1 para arreglos lineales.

Supongamos que un arreglo bidimensional de enteros está conservado mediante renglón —mayor, como en la figura 1.2.3, y que, para un arreglo *ar*, *base(ar)* es la dirección del primer elemento del arreglo. Es decir, si *ar* se declara por:

```
int ar[r1][r2];
```

donde *r1* y *r2* son los rangos de la primera y segunda dimensión, respectivamente, y *base(ar)* es la dirección de *ar[0][0]*. Supongamos también que *esize* es el tamaño de cada elemento del arreglo. Calculemos la dirección de un elemento arbitrario, *ar[i1][i2]*. Como el elemento está en el renglón *i1*, su dirección puede obtenerse al calcular la dirección del primer elemento del renglón *i1* y sumar la cantidad *i2* esize* (esta cantidad representa cuán lejos está el elemento de la columna *i2* dentro del renglón *i1*). Pero para alcanzar el primer elemento del renglón *i1* (es decir, el elemento *ar[i1][0]*), deben recorrerse *i1* renglones completos, cada uno de los cuales contiene *r2* elementos (dado que hay un elemento de cada columna en cada renglón),

0	2
0	4
<i>a[0][0]</i>	
<i>a[0][1]</i>	
<i>a[0][2]</i>	
<i>a[0][3]</i>	
<i>a[0][4]</i>	
<i>a[1][0]</i>	
<i>a[1][1]</i>	
<i>a[1][2]</i>	
<i>a[1][3]</i>	
<i>a[1][4]</i>	
<i>a[2][0]</i>	
<i>a[2][1]</i>	
<i>a[2][2]</i>	
<i>a[2][3]</i>	
<i>a[2][4]</i>	

Figura 1.2.3. Representación de un arreglo bidimensional.

de tal manera que la dirección del primer elemento del renglón *i1* está en *base(ar) + i1 * r2 * esize*. Por consiguiente, la dirección de *ar[i1][i2]* está en

$$\text{base(ar)} + (i1 * r2 + i2) * \text{esize}$$

Como ejemplo, considérese el arreglo *a* de la figura 1.2.2, cuya representación se muestra en la figura 1.2.3. En este arreglo, *r1* = 3, *r2* = 5 y *base(a)* es la dirección de *a[0][0]*. Supongamos también que cada elemento del arreglo requiere una sola unidad de memoria, de tal manera que *esize* es igual a 1. (Esto no es necesariamente cierto, dado que declaramos *a* como arreglo de enteros y un entero puede necesitar más de una unidad de memoria en una máquina particular. Sin embargo, aceptamos la suposición por simplicidad). Entonces la localidad de *a[2][4]* puede calcularse mediante

$$\text{base(a)} + (2 * 5 + 4) * 1 = \text{base(a)} + 14$$

base(a) + 14

Puede confirmarse que *a[2][4]* está 14 unidades después de *base(a)* en la figura 1.2.3.

Otra implantación posible de un arreglo bidimensional es la siguiente: un arreglo *ar*, declarado con límites superiores *u1* y *u2*, consiste en *u1 + 1* arreglos unidimensionales. El primero es un arreglo *ap* de apuntadores *u1*. El *i*-ésimo elemento de *ap* es un apuntador a un arreglo unidimensional *ar[i]*. Por ejemplo, la figura 1.2.4 muestra tal implantación para el arreglo *a* de la figura 1.2.2, donde *u1* es 3 y *u2* es 5.

Para referirse a *ar[i][j]*, se accesa primero *ar* para obtener el apuntador *ar[i]*. El arreglo en esa localidad del apuntador se accesa después para obtener *a[i][j]*.

De hecho esta segunda implantación es la más simple y directa de las dos. Sin embargo, los arreglos *u1* desde *ar[0]* hasta *ar[u1 - 1]* se ubicarían, por lo regular,

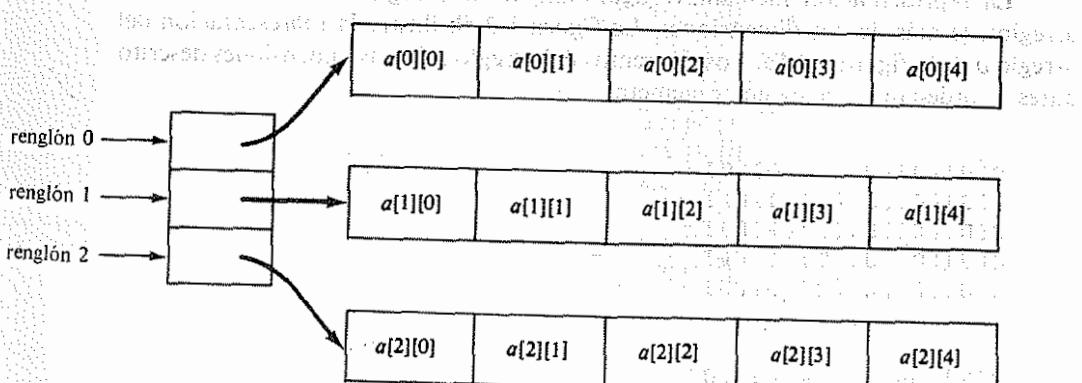


Figura 1.2.4. Implementación alternativa de un arreglo bidimensional.

en forma contigua, con $ar[0]$ seguido de modo inmediato por $ar[1]$, y así sucesivamente. La primera implantación evita tener que guardar el apuntador del arreglo extra ap y calcular el valor de un apuntador explícito al renglón deseado del arreglo. Así es más eficiente en cuanto a tiempo y espacio.

Arreglos multidimensionales

El lenguaje C también permite arreglos de más de dos dimensiones. Por ejemplo, puede declararse un arreglo tridimensional por medio de:

```
int b[3][2][4];
```

el cual se muestra en la figura 1.2.5a. Un elemento de este arreglo se especifica con tres subíndices, por ejemplo $b[2][0][3]$. El primero indica el número de plano, el segundo el de renglón y el tercero el de columna. Tal arreglo es útil cuando un valor está determinado por tres entradas. Por ejemplo, un arreglo de temperaturas puede indexarse por latitud, longitud y altitud.

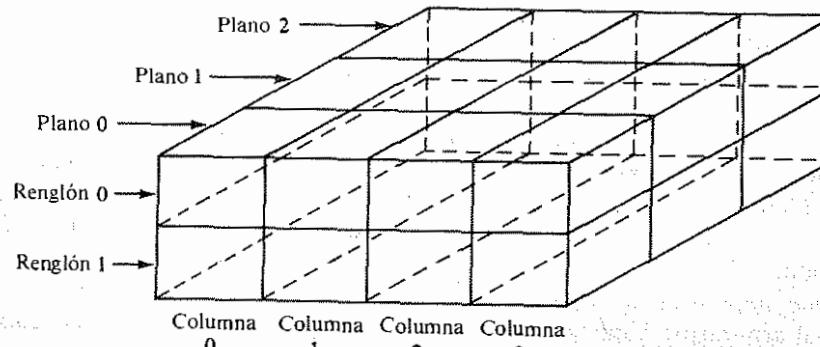
Por razones obvias, la analogía geométrica se rompe cuando vamos más allá de las tres dimensiones. Sin embargo, el lenguaje C permite un número arbitrario de dimensiones. Por ejemplo, puede declararse un arreglo de seis dimensiones por medio de:

```
int c [7][15][3][5][8][2];
```

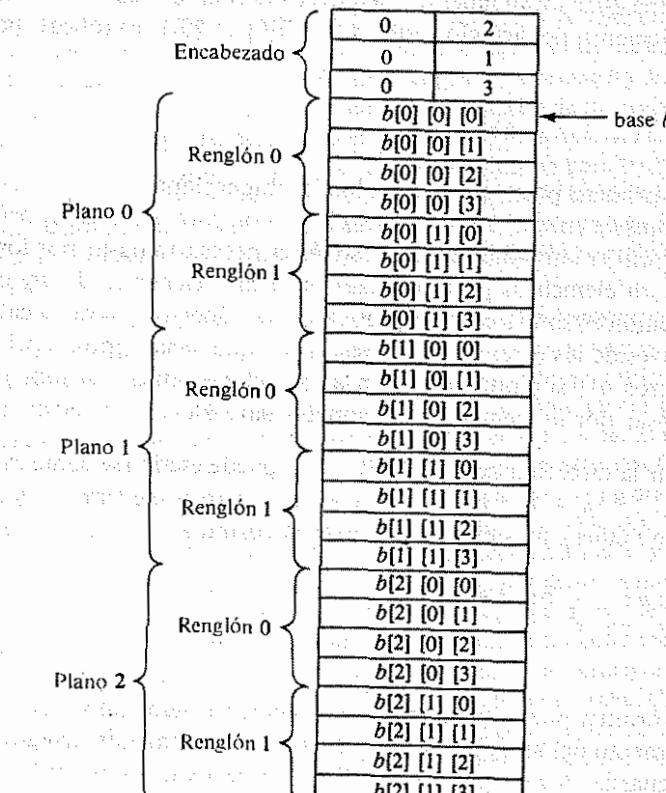
Referirnos a un elemento de este arreglo requeriría de seis subíndices, como $c[2][3][0][1][6][1]$. El número permisible de diferentes subíndices en una posición particular (el rango de una dimensión particular) es igual al límite superior de dicha dimensión. El número de elementos en un arreglo es igual al producto de los rangos de todas sus dimensiones. Por ejemplo, el arreglo b contiene los elementos $3 * 2 * 4 = 24$ y el $c 7 * 15 * 3 * 5 * 8 * 2 = 25200$ elementos.

La representación mediante renglón-mayor de arreglos puede extenderse a arreglos de más de dos dimensiones. La figura 1.2.5b ilustra la representación del arreglo b de la figura 1.2.5a. Los elementos del arreglo c de seis dimensiones descrito antes se ordenan de la siguiente manera:

```
c[0][0][0][0][0][0]
c[0][0][0][0][0][1]
c[0][0][0][0][1][0]
c[0][0][0][0][1][1]
c[0][0][0][0][2][0]
.
.
.
c[6][14][2][4][5][0]
c[6][14][2][4][5][1]
c[6][14][2][4][6][0]
```



(a)



(b)

Figura 1.2.5 Arreglo tridimensional

```
C[6][14][2][4][6][1]
C[6][14][2][4][7][0]
C[6][14][2][4][7][1]
```

Es decir, el último subíndice varía con más rapidez; los subíndices no se incrementan hasta que no se hayan agotado todas las posibles combinaciones de subíndices a su derecha. Esto se parece a un odómetro (el marcador de kilómetros recorridos es similar al cuentamillas de un coche) donde el dígito de la extrema derecha cambia mucho más rápido.

¿Qué mecanismo se necesita para accesar un elemento de un arreglo multidimensional arbitrario? Supóngase que *ar* es un arreglo de *n*-dimensiones declarado por

```
int ar[r1][r2]...[rn];
```

el cual se almacena en orden mediante renglón-mayor. Suponemos que cada elemento de *AR* ocupa *esize* localidades de memoria y definimos *base(ar)* como la dirección del primer elemento del arreglo (esto es, *ar[0][0]....[0]*). Entonces, para accesar el elemento

```
ar[i1][i2]...[in];
```

es necesario primero pasar a través de los *i1* "hiperplanos" completos, cada uno de los cuales consiste en *r2 * r3 * ... * rn* elementos para alcanzar el primer elemento de *ar*, cuyo primer subíndice es *i1*. Despues es necesario pasar por los *i2* grupos de *r3 * r4 * ... rn* elementos para alcanzar el primer elemento de *ar*, cuyo primer y segundo subíndices son *i1* e *i2*, respectivamente. Debemos llevar a cabo un proceso similar a través de las otras dimensiones, hasta que alcancemos el primer elemento, cuyos primeros *n-1* subíndices igualen a los del elemento deseado. Por último, es necesario pasar por *in* elementos adicionales para alcanzar el elemento deseado.

Así que la dirección de *ar[i1][i2]...[in]* puede escribirse como *base(ar) + esize * [i1 * r2 * ... * rn + i2 * r3 * ... * rn + ... + (i(n - 1) * rn + in)]*, lo que puede evaluarse de manera más eficiente al usar la fórmula:

```
base(ar) + esize *
[in + rn * (i(n - 1) + r(n - 1) * (... + r3 * (i2 +
r2 * i1) ...))]
```

Esta fórmula puede evaluarse con el siguiente algoritmo, que calcula la dirección del elemento del arreglo y la coloca en *addr* (suponiendo arreglos *r* e *i* de tamaño *n* para guardar los rangos y los índices, respectivamente):

```
offset = 0;
for (j = 0; j < n; j++)
    offset = r[j] * offset + i[j];
addr = base(ar) + esize * offset;
```

EJERCICIOS

- 1.2.1.a. La *mediana* de un arreglo de números es el elemento *m* para el cual la mitad del resto de los elementos del arreglo es mayor o igual que *m* y la otra mitad es menor o igual que *m*, si el número de elementos es impar. Si es par, la mediana es el promedio del par de elementos *m1* y *m2* para el cual una mitad de los elementos restantes del arreglo es mayor o igual que *m1* y *m2* y la otra mitad es menor o igual que *m1* y *m2*. Escriba una función en lenguaje C que acepte un arreglo de números y regrese su mediana.
- 1.2.1.b. La *MODA* de un arreglo de números es el número *m* del arreglo que se repite con mayor frecuencia. Si hay más de un número que se repite con igual frecuencia máxima, no existe la moda. Escriba una función en lenguaje C que acepte un arreglo de números y regrese su moda o una indicación de que no existe.
- 1.2.2. Escriba un programa en lenguaje C que haga lo siguiente: lea un grupo de registros de temperatura. Un registro consta de dos números; un entero entre -90 y 90 que representa la latitud en la que se tomó el registro y la temperatura observada en esa latitud. Imprima una tabla que consista de cada latitud y de la temperatura promedio en esa latitud. Si no existen registros para una latitud particular, imprimir "NO HAY DATOS" en lugar del promedio. Imprima luego las temperaturas promedio en los hemisferios norte y sur (el norte consiste en las latitudes de la 1 a la 90 y el sur en las latitudes de la -1 a la -90). (Esta temperatura promedio debiera computarse como promedio de promedios y no como promedio de los registros originales). Determine también cuál hemisferio es más cálido. Para hacerlo, tome el promedio de temperatura en todas las latitudes de cada hemisferio para las cuales hay datos, tanto en esa latitud como en la correspondiente en el otro hemisferio. (Por ejemplo, si hay datos para la latitud 57 pero no para la -57, entonces debe ignorarse la temperatura promedio para la latitud 57 en la determinación de cuál hemisferio es más cálido).
- 1.2.3. Escriba un programa para una cadena de 20 tiendas de departamento, cada una de las cuales vende 10 artículos diferentes. Todos los meses, cada supervisor manda una ficha de datos para cada artículo, la cual comprende el número de tienda (de 1 a 20), el número de artículo (de 1 a 10) y una cifra de venta (menor de \$100,000) que representa el monto de las ventas de ese artículo en esa tienda. Sin embargo, algunos supervisores pueden no mandar ficha para algunos artículos (por ejemplo, no todos los artículos son vendidos en todas las tiendas). Se tendrá que escribir un programa para leer esas fichas de datos e imprimir una tabla con 12 columnas. La primera columna debe contener los números de tienda del 1 al 20 y la palabra "TOTAL" en la última línea. Las siguientes 10 columnas deben contener las cifras de venta de cada uno de los diez artículos para cada una de las tiendas con el total de ventas de cada artículo en la última línea. La última columna debe contener el total de ventas de cada una de las 20 tiendas para todos los artículos, con la cifra del total de ventas global de la cadena en la esquina inferior derecha. Cada columna deberá llevar un encabezado apropiado. Si no se registran ventas de un artículo y tienda particular, se supondrán cero ventas. No suponga que su entrada está en algún orden particular.
- 1.2.4. Muestre cómo puede representarse en lenguaje C un tablero por medio de un arreglo. Muestre cómo representar el estado de un juego de damas en un instante particular. Escriba una función en lenguaje C que sea una entrada a un arreglo que represente tal tablero y que imprima todos los posibles movimientos de las negras que puedan hacerse a partir de esa posición.

1.2.5. Escriba una función *printar()* que acepte un arreglo *a* de *m* por *n* enteros e imprima los valores del arreglo en varias páginas de la siguiente manera: cada página debe contener 50 renglones y 20 columnas del arreglo. A lo largo del tope de cada página deben escribirse encabezados "COL 0", "COL 1", y así sucesivamente; así como A"RENGO, RENG 1", etc. a lo largo del margen izquierdo. El arreglo debe imprimirse por subarreglos. Por ejemplo, si *a* fuera un arreglo de 100 por 100, la primera página contendría del *a*[0][0] al *a*[49][19], la segunda del *a*[0][20] al *a*[49][39], la tercera del *a*[0][40] al *a*[49][59], y así hasta la quinta página que contendría del *a*[0][80] al *a*[49][99], la sexta contendría del *a*[50][0] al *a*[99][19] y así sucesivamente. La impresión completa ocuparía 10 páginas. Si el número de renglones no es un múltiplo de 50 o el número de columnas no lo es de 20 las últimas páginas de la impresión deberán contener menos de 100 números.

1.2.6. Suponga que cada elemento de un arreglo *a* que está almacenado en orden mediante renglón mayor ocupa cuatro unidades de memoria. Si se declara *a* de cada una de las siguientes maneras y la dirección del primer elemento de *a* es 100, encuentre la dirección del elemento del arreglo que se indica:

- | | |
|-------------------|-----------------------|
| a. int a[100]; | dirección de a[10] |
| b. int a[200]; | dirección de a[100] |
| c. int a[10][20]; | dirección de a[0][0] |
| d. int a[10][20]; | dirección de a[2][1] |
| e. int a[10][20]; | dirección de a[5][1] |
| f. int a[10][20]; | dirección de a[1][10] |
| g. int a[10][20]; | dirección de a[2][10] |
| h. int a[10][20]; | dirección de a[5][3] |
| i. int a[10][20]; | dirección de a[9][19] |

1.2.7. Escriba una función *listoff* en lenguaje C que acepte como parámetros a dos arreglos unidimensionales del mismo tamaño: *range* y *sub*. *range* representan el rango de un arreglo de enteros. Por ejemplo, si los elementos de *range* son

3 5 10 6 3

range representa un arreglo *a* declarado por

```
int a[3][5][10][6][3];
```

Los elementos de *sub* representan subíndices del arreglo inmediato anterior. Si *sub*[*i*] no se encuentra entre 0 y *range*[*i*] - 1, faltan todos los subíndices del *i*-ésimo en adelante. En el ejemplo anterior, si los elementos de *sub* son

1 3 1 2 3

sub representa el arreglo unidimensional *a*[1][3][1][2]. La función *listoff* debe imprimir los desplazamientos desde la base del arreglo *a* representado por *range* de todos los elementos de *a* que están incluidos en el arreglo (o el desplazamiento del único elemento si todos los subíndices están dentro de los límites) representado por *sub*. Suponga que el tamaño *esize* de cada elemento de *a* es 1. En el ejemplo anterior, *listoff* deberá imprimir los valores 4, 5 y 6.

1.2.8.a. Un arreglo *triangular inferior* *a* es un arreglo de *n* por *n* en el cual *a*[*i*][*j*] == 0, si *i* < *j*. ¿Cuál es el número máximo de elementos distintos de cero en tal arreglo?

¿Cómo pueden almacenarse en forma secuencial estos elementos en la memoria? Desarrolle un algoritmo para accesar *a*[*i*][*j*], donde *i* >= *j*. Defina un arreglo *triangular superior* de manera análoga y haga lo mismo que para el triangular inferior.

- b. Un arreglo *estrictamente triangular inferior* *a* es un arreglo de *n* por *n* en el cual *a*[*i*][*j*] == 0 si *i* <= *j*. Responda las preguntas de la parte a para un arreglo de este tipo.
- c. Sean *a* y *b* dos arreglos triangular inferior de *n* por *n*. Muestre cómo puede usarse un arreglo *c* de *n*-por-(*n* + 1) para contener todos los elementos distintos de cero de los dos arreglos. ¿Cuáles elementos de *c* representan los elementos *a*[*i*][*j*] y *b*[*i*][*j*], respectivamente?
- d. Un arreglo *tridiagonal* *a* es un arreglo de *n* por *n* en el cual *a*[*i*][*j*] == 0, si el valor absoluto de *i* - *j* es mayor que 1. ¿Cuál es el número máximo de elementos distintos de cero en tal arreglo? ¿Cómo pueden almacenarse estos elementos en forma secuencial en la memoria? Desarrolle un algoritmo para accesar *a*[*i*][*j*] si el valor absoluto de *i* - *j* es 1 o menor. Hágase lo mismo para un arreglo *a* en el cual *a*[*i*][*j*] == 0, si el valor absoluto de *i* - *j* es mayor que *k*.

1.3. ESTRUCTURAS EN LENGUAJE C

En esta sección examinamos la estructura de datos del lenguaje C llamada *estructura*. Suponemos que el lector está familiarizado con la estructura por medio de un curso introductorio. En esta sección repasamos sus elementos sobresalientes y puntualizamos algunas características importantes necesarias para un estudio más general.

Una estructura es un grupo de objetos en el que se identifica a cada uno mediante su propio identificador, cada uno de los cuales se conoce como *miembro* de la estructura. (En muchos otros lenguajes de programación se llama "registro" (record) a una estructura y "campo" (field) a un miembro. Se puede usar de vez en vez estos términos en lugar de "estructura" y "miembro", aunque ambos términos tienen un significado diferente en el lenguaje C. Por ejemplo, considérese la siguiente declaración:

```
struct {
    char first[10];
    char midinit;
    char last[20];
} sname, ename;
```

Esta declaración crea dos estructuras variables, *sname* y *ename*, cada una de las cuales contiene tres miembros: *first*, *midinit* y *last*. Dos de esos miembros son cadenas de caracteres y el tercero es un carácter simple. También se puede asignar una etiqueta a la estructura y declarar luego las variables por medio de ella. Por ejemplo, considérese la siguiente declaración que hace lo mismo que la anterior:

```
struct nametype {
    char first[10];
```

```

    char midinit;
    char last[20];
};

struct nametype sname, ename;

```

Esta definición crea una etiqueta de estructura *nametype* que contiene tres miembros, *first*, *midinit* y *last*. Una vez definida la etiqueta de la estructura, pueden declararse las variables *sname* y *ename*. Para mayor claridad del programa, es recomendable declarar una etiqueta para cada estructura y luego declarar variables mediante dicha etiqueta.

Otra alternativa para la estructura de etiqueta es usar la definición *typedef* en C. Por ejemplo:

```

typedef struct {
    char first[10];
    char midinit;
    char last[20];
} NAMETYPE;

```

dice que el identificador *NAMETYPE* es sinónimo de la estructura precedente dondequiera que ocurra *NAMETYPE*. Podemos entonces declarar:

NAMETYPE *sname, ename;*

para acceder a los miembros de la estructura. Declarando la variable *sname* con el tipo *NAMETYPE* se obtiene acceso directo a los miembros de la estructura. Una vez declarada la variable *sname*, se puede acceder a sus miembros directamente. Esto facilita la escritura del código y reduce el riesgo de errores.

Una vez declarada una variable como estructura, cada uno de sus miembros puede accesarse al especificar el nombre de la variable y el identificador del miembro del objeto separados por un punto. Así que la instrucción

```
printf("%s", sname.first);
```

puede usarse para imprimir el primer nombre en la estructura *sname* y la instrucción

```
ename.midinit = 'm'
```

puede usarse para poner la inicial intermedia de la estructura *ename* a la letra *m*. Si un miembro de una estructura es un arreglo, tiene que usarse un subíndice para accesar un elemento particular del arreglo, como en:

```
for (i=0; i < 20; i++)
    sname.last[i] = ename.last[i];
```

Puede declararse un miembro de una estructura como otra estructura. Por ejemplo, dada la definición precedente de *nametype* y la siguiente definición de *addrtype*

```

struct addrtype {
    char straddr[40];
    char city[10];
    char state[2];
    char zip[5];
};

```

puede declararse una nueva etiqueta de estructura *nmadtype* con:

```

struct nmadtype {
    struct nametype name;
    struct addrtype address;
};

```

Si se declaran dos variables

```
struct nmadtype nmad1, nmad2;
```

las siguientes son instrucciones válidas:

```

nmad1.name.midinit = nmad2.name.midinit;
nmad2.address.city[4] = nmad1.name.first[1];
for (i=1; i < 10; i++)
    nmad1.name.first[i] = nmad2.name.first[i];

```

Algunos de los compiladores más nuevos de C, así como el estándar ANSI desarrollado, permiten asignaciones de estructuras del mismo tipo. Por ejemplo, la instrucción *nmad1 = nmad2*, podría ser válida y equivalente a

```

nmad1.name = nmad2.name;
nmad2.address = nmad2.address;
```

Lo cual, a su vez, equivale a

```

for (i=0; i < 10; i++)
    nmad1.name.first[i] = nmad2.name.first[i];
nmad1.name.midinit = nmad2.name.midinit;
for (i=0; i < 20; i++)
    nmad1.name.last[i] = nmad2.name.last[i];
for (i=0; i < 40; i++)
    nmad1.address.straddr[i] = nmad2.address.straddr[i];
for (i=0; i < 10; i++)
    nmad1.address.city[i] = nmad2.address.city[i];

```

```

for (i=0; i < 2; i++)
    nmad1.address.state[i] = nmad2.address.state[i];
for (i=0; i < 5; i++)
    nmad1.address.zip[i] = nmad2.address.zip[i];

```

Sin embargo, como el lenguaje C original, tal y como fue definido por Kernighan y Ritchie y otras versiones implantadas de C no permiten la asignación de estructuras, no se utilizará esta característica en el resto del libro.

Considérese otro ejemplo del uso de estructuras, en el que definimos estructuras que describen un empleado y un estudiante, respectivamente:

```

struct date {
    int month;
    int day;
    int year;
};

struct position {
    char deptno[2];
    char jobtitle[20];
};

struct employee {
    struct nmaddr type nameaddr;
    struct position job;
    float salary;
    int numdep;
    short int hplan;
    struct date datehired;
};

struct student {
    struct nmaddr type nmad;
    float gpindx;
    int credits;
    struct date dateadm;
};

```

Al suponer las declaraciones

```

struct employee e;
struct student s;

```

la instrucción para aumentar 10% a un empleado con un índice de calificaciones como estudiante superior a 3.0 es la siguiente:

```

if ((e.nameaddr.name.first == s.nmad.name.first) &&
    (e.nameaddr.name.midinit == s.nmad.name.midinit) &&
    (e.nameaddr.name.last == s.nmad.name.last)) {
    if (s.gpindx > 3.0)
        e.salary *= 1.10;
}

```

Esta instrucción asegura, primero, que el registro del estudiante y del empleado se refieran a la misma persona al comparar sus nombres. Nótese que no puede simplemente escribirse

```

if (e.nmad.name == s.nmad.name)
    ...

```

dado que no pueden compararse dos estructuras con una sola operación en C.

Puede haberse notado que se usan dos identificadores diferentes, *nameaddr* y *nmad* para los miembros nombre/dirección en los registros del estudiante y el empleado, respectivamente. No es necesario hacerlo, puede usarse el mismo identificador para nombrar los miembros de tipos de estructura diferentes. Esto no causa ninguna ambigüedad ya que el nombre de un miembro debe siempre ir precedido por una expresión que identifica la estructura de un tipo específico.

Implantación de estructuras

Pasemos ahora de la aplicación de estructuras a su implantación. Cualquier tipo de datos en C puede concebirse como un patrón o plantilla. Con esto se quiere decir que un tipo es un método de interpretar una porción de la memoria. Cuando se declara de un cierto tipo a una variable, se quiere decir que el identificador se refiere a una cierta porción de memoria y que los contenidos de esa memoria deben interpretarse de acuerdo al patrón que define el tipo, el cual especifica ambas cosas: la cantidad de memoria apartada para la variable y el método mediante el cual se interpreta dicha memoria.

Por ejemplo, supóngase que bajo cierta implantación de C se representa un entero con cuatro bytes, un número de punto flotante con ocho y un arreglo de 10 caracteres con diez. Entonces las declaraciones:

```

int x;
float y;
char z[10];

```

especifican que deben apartarse cuatro bytes de memoria para *x*, ocho para *y* y diez para *z*. Una vez que se apartaron esos bytes para esas variables, los nombres *x*, *y* y *z* siempre se referirán a las localidades correspondientes. Cuando se haga referencia a *x* se interpretarán sus cuatro bytes como un entero, cuando se haga a *y*, se interpretarán sus ocho bytes como un número real y cuando se haga a *z*, se interpretarán sus diez bytes como una colección de diez caracteres. La cantidad de memoria apartada para cada tipo así como el método según el cual deben interpretarse los contenidos de dicha memoria, varía de una máquina e implantación en C a otra. Pero dada una implantación en C, cualquier tipo indica siempre una cantidad de memoria y un método específico para interpretarla.

Supóngase ahora que se define una estructura por

```

struct structtype {
    int field1;
}

```

```

    float field2;
    char field3[10];
} ;

```

y se declara una variable

```
struct structtype r;
```

Entonces, la cantidad de memoria especificada por la estructura es la suma de las cantidades de memoria especificadas por cada uno de los tipos de sus miembros. De tal manera que el espacio requerido para la variable *r* es la suma del espacio requerido para un entero (4 bytes), un número de punto flotante (8 bytes) y un arreglo de 10 caracteres (10 bytes). En consecuencia, deben apartarse 22 bytes para *r*. Los 4 primeros se interpretan como un entero, los 8 siguientes como un número de punto flotante y los diez últimos como un arreglo de caracteres. (Esto no siempre es cierto. En algunas computadoras, los objetos de cierto tipo no pueden comenzar en cualquier parte de la memoria, sino que deben comenzar a partir de ciertas "fronteras". Por ejemplo, un entero de cuatro bytes podría tener que comenzar a partir de una dirección divisible por cuatro y un número real de longitud 8 bytes a partir de una divisible por 8. De tal manera que en nuestro ejemplo, si la dirección de inicio de *r* fuese 200, el entero ocuparía los bytes del 200 al 203, pero el número real no podría comenzar en el byte 204 dado que esa localidad no es divisible por ocho. Así que el número real tendría que comenzar en la localidad 208 y el registro completo requeriría de 26 bytes en lugar de 22. Los bytes desde el 204 hasta el 207 son espacio desperdiciado).

Para cualquier referencia a un miembro de una estructura debe calcularse una dirección. Con cada identificador de los miembros de una estructura se asocia un *desplazamiento* que especifica qué tan lejos del inicio de la estructura está la localidad de dicho miembro. En el ejemplo precedente, el desplazamiento de *field1* es 0, el de *field2* es 4 (suponiendo que no hay restricciones de frontera) y el de *field3* es 12. Asociada a cada variable de estructura hay una dirección base, que es la localidad del principio de la memoria asignada a dicha variable. Estas asociaciones las establece el compilador y no deben importar al usuario. Para calcularse la localidad de un miembro de una estructura, se suma al desplazamiento del identificador del miembro la dirección base de la variable de estructura.

Por ejemplo, supóngase que la dirección base de *r* es 200. Entonces lo que en realidad ocurre al ejecutar la instrucción *r.field2 = r.field1 + 3.7;* es lo siguiente: primero, se determina la localidad de *r.field1* como la dirección base de *r* (200) más el desplazamiento de campo de *field1* (0), lo que es igual a 200. Los cuatro bytes en las localidades 200 a 203 se interpretan como un entero. Despues se convierte este entero a número de punto flotante y luego se suma al 3.7. El resultado es un número de punto flotante que abarca 8 bytes. Despues se calcula la localidad de *r.field2* como la dirección base de *r* (200) más el desplazamiento del campo *field2*

(4), o 204. Los contenidos de los 8 bytes 204 a 211 se establecen para el número de punto flotante calculado al evaluar la expresión.

Nótese que el proceso para calcular la dirección del componente de una estructura es muy similar al que se usa para calcular la dirección del componente de un arreglo. En ambos casos, se suma un desplazamiento que depende del seleccionador del componente (el identificador del miembro o el valor del subíndice) a la dirección base de la estructura compuesta (la estructura o el arreglo). En el caso de una estructura, se asocia el desplazamiento con el identificador del campo mediante la definición de tipo, mientras que en el caso del arreglo, se calcula el desplazamiento de acuerdo con el valor del subíndice.

Pueden combinarse estos dos tipos de direccionamiento (estructura y arreglo). Por ejemplo, para calcular la dirección de *r.field3[4]*, primero se emplea el direccionamiento de la estructura para determinar la dirección base del arreglo *r.field3* y luego se usa el direccionamiento del arreglo para calcular la localización del quinto elemento de dicho arreglo. La dirección base de *r.field3* está dada por la dirección base de *r* (200) más el desplazamiento de *field3* (12), que es 212. La dirección de *r.field3[4]* se determina después como la dirección base de *r.field3* (212) más 4 (el subíndice 4 menos el límite inferior del arreglo, 0) multiplicado por el tamaño de cada elemento del arreglo (1), lo que es igual a 212 + 4 * 1, o 216.

Como ejemplo adicional considérese otra variable, *rr*, declarada por

```
struct structtype rr[20];
```

rr es un ejemplo de arreglo de estructuras. Si la dirección base de *rr* es 400, entonces la dirección de *rr[14].field3[6]* puede calcularse como sigue. El tamaño de cada componente de *rr* es 22, así que la localidad de *rr[14]* es 400 + 14 * 22 o 708. La dirección base de *rr[14].field3*, será entonces 708 + 12 o 720. Por consiguiente, la dirección de *rr[14].field3[6]* es 720 + 6 * 1 o 726. (Esto ignora, de nuevo, la posibilidad de restricciones de frontera. Por ejemplo, aunque el tipo *rectype* puede requerir sólo 22 bytes, cada *rectype* tal vez tenga que empezar en una dirección divisible por cuatro, de tal manera que se desperdicien dos bytes entre cada elemento de *rr* y su vecino. Si esto sucede, entonces el tamaño real de cada elemento es 24, por lo que la dirección de *rr[14].field3[6]* en realidad es 754 en lugar de 726).

Uniones

Todas las estructuras vistas hasta ahora tenían miembros fijos y formato único. El lenguaje C también permite otro tipo de estructura, la *unión*, mediante la cual puede interpretarse una variable de distintas maneras.

Por ejemplo, considérese una compañía de seguros que ofrece tres tipos de póliza: de vida, de automóvil y de vivienda. Un número identifica cada póliza de seguro, de cualquier tipo que sea. Para los tres tipos es necesario tener nombre, dirección, cantidad asegurada y prima mensual del asegurado. Para las pólizas de automóvil y vivienda es necesaria la cantidad deducible. Para el seguro de vida se necesita la fecha de nacimiento del asegurado y del beneficiario. Para la de automóvil, se requiere el número de licencia, el estado, el modelo y el año. Para la de vivien-

da, se necesita la constancia de antigüedad y el tipo de dispositivos de seguridad. El tipo de estructura de póliza para una compañía como la anterior puede definirse como una unión. Se definen primero dos estructuras auxiliares.

```
#define LIFE 1
#define AUTO 2
#define HOME 3

struct addr {
    char street[50];
    char city[10];
    char state[2];
    char zip[5];
};

struct date {
    int month;
    int day;
    int year;
};

struct policy {
    int polnumber;
    char name[30];
    struct addr address;
    int amount;
    float premium;
    int kind; /* LIFE, AUTO o HOME */
    union {
        struct {
            char beneficiary[30];
            struct date birthday;
        } life;
        struct {
            int autodeduct;
            char license[10];
            char state[2];
            char model[15];
            int year;
        } auto;
        struct {
            int homededuct;
            int yearbuilt;
        } home;
    };
};

policyinfo { ... }
```

Examinemos la unión con más detalle. La definición consiste en dos partes: una fija y una variable. La parte fija consta de todas las declaraciones de los miembros hasta la palabra reservada *unión*, mientras que la parte variable consiste en el resto de la definición.

Ahora que se ha examinado la sintaxis de la definición de unión, examinemos su semántica. Una variable declarada como de tipo unión T (por ejemplo, *struct policy p*) contiene siempre todos los miembros fijos de T. Así, siempre es válido referirse a *p.name*, *p.premium* o *p.kind*. Sin embargo los miembros de unión incluidos en el valor de esa variable depende de lo que haya almacenado el programador.

Es responsabilidad del programador asegurar que es compatible el uso de un miembro con lo que se ha puesto en esa localidad. Una buena idea es tener un miembro fijo separado en una estructura que contenga una unión cuyo valor indique cuál es la alternativa que se usa en ese momento. En el ejemplo precedente, se usa el miembro *kind* con ese propósito. Si su valor es LIFE (1), entonces la estructura contiene una póliza de seguro de vida, si es AUTO (2) una de automóvil y si es HOME (3) una de vivienda. Así, el programador debe ejecutar un código similar al que sigue para referirse a la unión:

```
if (p.kind == LIFE)
    printf("/n%s %d//%d/%d",
        p.policyinfo.life.beneficiary,
        p.policyinfo.life.birthday.month,
        p.policyinfo.life.birthday.day,
        p.policyinfo.life.birthday.year);
else if (p.kind == AUTO)
    printf("/n%d %s %s %d",
        p.policyinfo.auto.autodeduct,
        p.policyinfo.auto.license,
        p.policyinfo.auto.state,
        p.policyinfo.auto.model,
        p.policyinfo.auto.year);
else if (p.kind == HOME)
    printf("/n%d %d",
        p.policyinfo.home.homededuct,
        p.policyinfo.home.yearbuilt);
else
    printf("tipo incorrecto %d en kind", p.kind);
```

En el ejemplo anterior, si el valor de *p.kind* es LIFE, *p* contiene los miembros *beneficiary* y *birthday*. No es válido hacer referencia a *model* o *yearbuilt* cuando el valor de *kind* sea LIFE. De manera similar, si el valor de *kind* es AUTO, puede referirse a *autodeduct*, *license*, *state*, *model*, y *year* pero a ningún otro miembro. Sin embargo, el lenguaje C no requiere un miembro fijo para indicar la alternativa actual de una unión ni nos impone el uso de una alternativa particular dependiendo del valor fijo de un miembro.

Una unión permite que una variable tome varios “tipos” diferentes en puntos distintos de una ejecución. También permite que un arreglo contenga objetos de tipos diferentes. Por ejemplo, el arreglo *a*, declarado por

```
struct policy a[100];
```

puede contener pólizas de automóvil, de vivienda y de vida. Supóngase que se declara un arreglo *a* y que se desea aumentar en 5% las primas de todas las pólizas de

seguro de vida y vivienda contratadas antes de 1950. Puede hacerse de la siguiente manera:

```
for (i=0; i<100; i++)  
    if (a[i].kind == LIFE)  
        a[i].premium = 1.05 * a[i].premium;  
    else if (a[i].kind == HOME &  
            a[i].policyinfo.yearbuilt < 1950)  
        a[i].premium = 1.05 * a[i].premium;
```

Implantación de uniones

Para entender por completo el concepto de unión es necesario examinar su implantación. Una estructura puede considerarse como el mapa de un camino hacia un área de la memoria: define cómo debe interpretarse la memoria. Una unión provee varios mapas de camino diferentes para la misma área de la memoria y es responsabilidad del programador determinar cuál de ellos se usa en cada momento. En la práctica, el compilador asigna suficiente memoria para contener el miembro más grande de la unión. Es el mapa, sin embargo, el que determina cómo debe interpretarse la memoria. Por ejemplo, considérese la unión simple y las estructuras

```
#define INTEGER 1  
#define REAL 2  
  
struct stint {  
    int f3, f4;  
};  
  
struct stfloat {  
    float f5, f6;  
};  
  
struct sample {  
    int f1;  
    float f2;  
    int utype;  
    union {  
        struct stint x;  
        struct stfloat y;  
    } funion;  
};
```

Supóngase de nuevo una implantación en la cual un entero requiere 4 bytes y un número de punto flotante 8. Entonces, los tres miembros fijos f1, f2 y utype ocupan 16 bytes. El primer miembro de la unión, x, requiere 8 bytes, mientras que el segundo, y, requiere 16. La memoria que en realidad se asigna a la parte de la unión de la variable, es el espacio máximo necesario para cualquier miembro. En este caso, en consecuencia, son asignados 16 bytes para la parte de la unión de sample. Sumados a los 16 bytes necesarios para la parte fija de la unión, a sample se le asignan 32.

Los diferentes miembros de la unión se superponen unos a otros. En el ejemplo anterior, si se le asigna espacio a sample desde la localidad 100, de tal manera que sample ocupe los bytes del 100 al 131, los miembros fijos sample.f1, sample.f2 y sample.utype ocupan los bytes del 100 al 103, del 104 al 111 y del 112 al 115, respectivamente. Si el valor del miembro utype es INTEGER (es decir 1), sample.funion.x.f3 y sample.funion.y.f4, ocupan los bytes del 116 al 119 y del 120 al 123, respectivamente y no se usan los bytes del 124 al 131. Si el valor de sample.utype es REAL (es decir 2), sample.funion.y.f5 ocupan los bytes del 116 al 123 y sample.funion.y.f6 del 124 al 131. Esto sucede porque sólo puede existir un miembro único de la unión en un instante. Todos los miembros de la unión usan el mismo espacio y este espacio puede utilizarlo a la vez uno y sólo uno de ellos. El programador determina qué miembro es el apropiado.

Parámetros de una estructura

En el lenguaje C tradicional, una estructura no puede pasarse a una función mediante una llamada por valor. Para pasar una estructura a una función, tenemos que pasar su dirección a la función y referirnos a la estructura por medio de un apuntador (es decir, se llama por referencia). La notación *p — x* en C, es equivalente a la notación *(*p).x* y se usa con frecuencia para referirse a un miembro de un parámetro de estructura. Por ejemplo, la siguiente función imprime un nombre en un formato tabular y devuelve el número de caracteres impresos:

```
writename (name)  
{  
    struct nametype *name;  
    int count, i;  
    printf("\n");  
    count = 0;  
    for (i=0; (i < 10) && (name->first[i] != '\0'); i++) {  
        printf("%c", name->first[i]);  
        count++;  
    } /* fin de for */  
    printf("%c", ' ');  
    count++;  
    if (name->midinit != ' ') {  
        printf("%c%s", name->midinit, ".");  
        count += 3;  
    } /* fin de if */  
    for (i=0; (i < 20) && (name->last[i] != '\0'); i++) {  
        printf("%c", name->last[i]);  
        count++;  
    } /* fin de for */  
    return(count);  
} /* fin de writename */
```

La siguiente tabla ilustra los efectos de la instrucción `x = writename (&sname)` sobre dos valores diferentes de `sname`:

valor de <code>sname.first:</code>	"Sara"	"Irene"
valor de <code>sname.midinit:</code>	'M'	"
valor de <code>sname.last:</code>	"Binder"	"LaClaustra"
salida impresa:	Sara M. Binder	Irene LaClaustra
valor de <code>x:</code>	14	16

De manera similar, la instrucción `x = writename (&ename)` imprime los valores de los campos de `ename` y asigna el número de caracteres impresos a `x`.

El nuevo estándar propuesto para C y algunos compiladores C que permiten la asignación de estructuras permiten también pasárselas por valor, sin aplicar el operador `&`. Esto implica copiar los valores de la estructura completa cuando se llama la función. Sin embargo, en este libro no suponemos esta capacidad y pasamos todas las estructuras por referencia.

Ya se ha visto que el miembro de una estructura puede ser un arreglo u otra estructura. De manera similar puede declararse un arreglo de estructuras. Por ejemplo, si los tipos `employee` y `student` se declaran como anteriormente, pueden declararse dos arreglos de estructura `employee` y `student` como:

```
struct employee e[100];
struct student s[100];
```

El salario del empleado 14º es referido por `e[13].salary` y el apellido por `e[13].nameaddr.name.last`. De manera similar, el año de admisión del primer estudiante es `s[0].dateadm.year`.

Como ejemplo adicional se presenta una función que da al principio de un nuevo año para dar 10% de aumento a todos los trabajadores con más de 10 años de antigüedad y 5% al resto. Primero, debe definirse un nuevo arreglo de estructuras.

```
struct employee empset[100];
```

Y a continuación:

```
#define THISYEAR ...
raise (e)
{
    struct employee e[1];
    int i;
    for (i=0; i < 100; i++)
        if (e[i].datehired.year < THISYEAR - 10)
            e[i].salary *= 1.10;
        else
            e[i].salary *= 1.05;
} /* fin de raise */
```

Como otro ejemplo, supóngase que se añade otro miembro, `sindex`, a la definición de la estructura `employee`. Este miembro contiene un entero e indica el índice académico del empleado en el arreglo `s` particular. Se declara `sindex` (dentro del registro) como sigue:

```
struct employee {
    struct nametype nameaddr;
    ...
    struct datehired ...;
    int sindex;
};
```

Puede referirse al número de créditos cursados por el empleado `i` cuando era estudiante mediante `s[e[i].sindex].credits`.

La siguiente función puede usarse para dar 10% de aumento a todos los empleados cuyo índice académico sea superior a 3.0 y regresar el número de tales empleados. Nótese que ya no se tiene que comparar el nombre de un empleado con el de un estudiante para asegurarse de que sus registros representan a la misma persona (aunque esos nombres deben ser iguales si los comparamos). Más bien, puede usarse en forma directa el campo `sindex` para accesar el registro del estudiante apropiado para un empleado. Supóngase que el programa principal contiene la declaración

```
struct employee e[100];
struct student s[100];

raise2 (e, s)
{
    struct employee e[1];
    struct student s[];
    int i, j, count;
    count = 0;
    for (i=0; i < 100; i++) {
        j = e[i].sindex;
        if (s[j].gpindx > 3.0) {
            count++;
            e[i].salary *= 1.10;
        } /* end if */
    } /* fin de for */
    return(count);
} /* fin de raise 2 */
```

Con mucha frecuencia, se utiliza un arreglo de estructuras grande para contener una tabla importante de datos, para una aplicación determinada. En general sólo hay una tabla para cada arreglo de estructuras de este tipo. La tabla de estudiantes `s` y la de empleados `e`, del ejemplo previo, son buenos ejemplos de este tipo de tablas de datos. En tales casos, se usan con frecuencia las tablas únicas como variables

estático/externas, con una gran número de funciones que las accesan, en lugar de como parámetros, lo cual aumenta la eficiencia mediante la eliminación de la sobre-carga que conlleva la transferencia de parámetros. Puede volver a escribirse fácilmente la función *raise2* anterior, para accesar *s* y *e* como variables estático/externas en lugar de como parámetros, al cambiar simplemente el encabezado de la función a:

```
raise2()
```

El cuerpo de la función no necesita cambiarse toda vez, que las tablas *s* y *e* se declaran en el programa externo.

Representación de otras estructuras de datos

En lo que sigue del libro, se usan las estructuras para representar estructuras de datos más complejas. Agregar datos dentro de una estructura es útil porque permite agrupar objetos dentro de una entidad simple así como nombrar cada uno de esos objetos en forma apropiada, de acuerdo con su función.

Considérense los problemas de la representación de números racionales, como ejemplo de cómo pueden usarse las estructuras de esta manera.

Números racionales

En la sección previa se presentó un ADT para números racionales. Recuérdese que un *número racional* es cualquier número que puede expresarse como el cociente de dos enteros. Así $1/2$, $3/4$, $2/3$ y 2 (o sea $2/1$) son números racionales mientras que $\text{sqrt}(2)$ y π no lo son. Una computadora por lo general representa los números racionales mediante su aproximación decimal. Si se dan instrucciones a la computadora para imprimir $1/3$, ésta responderá con $.333333$. Aunque es una buena aproximación (la diferencia entre $1/3$ y $.333333$ sólo es una trimillónésima), no es exacta. Si se pregunta por el valor de $1/3 + 1/3$, el resultado será $.666666$ (que es igual a $.333333 + .333333$), mientras que el resultado de imprimir $2/3$ podría ser $.666667$. Esto podría significar que el resultado de la pregunta $1/3 + 1/3 = 2/3$ ¡podría ser falso! En muchos casos, la aproximación decimal es muy buena, pero a veces no lo es. Por esto, es deseable implantar una representación de los números racionales para que pueda ejecutarse la aritmética exacta.

¿Cómo puede representarse en forma exacta un número racional? Puesto que un número racional consiste en un numerador y un denominador, puede representarse un número racional (*rational*) mediante estructuras, como sigue:

```
struct rational {
    int numerator;
    int denominator;
```

Otra manera de declarar este nuevo tipo es la siguiente:

```
typedef struct { ... } rational;
```

```
int numerator;
int denominator;
} RATIONAL;
```

Con la primera técnica, un racional *r* se declara como

```
struct rational r;
```

y con la segunda por

```
RATIONAL r;
```

Podría pensarse que se está listo ahora para definir la aritmética de números racionales, pero hay un problema importante. Supóngase que se definen dos números racionales *r1* y *r2* y que se les da un valor. ¿Cómo puede comprobarse si dos números son iguales? Quizá podría querer programarse

```
if (r1.numerator == r2.numerator && r1.denominator == r2.denominator)
    ...
}
```

Es decir, si tanto los numeradores como los denominadores son iguales, los dos números racionales son iguales. Sin embargo es posible que ambos numeradores y denominadores sean diferentes y, no obstante, ser iguales los números racionales. Por ejemplo, los números $1/2$ y $2/4$ son iguales, aunque tanto sus numeradores (1 y 2) como sus denominadores (2 y 4) son distintos. Por consiguiente, se necesita una nueva forma de probar la igualdad para nuestra representación.

Bien, ¿por qué son iguales $1/2$ y $2/4$? La respuesta es que ambos representan el mismo racional. Uno entre dos y dos entre cuatro son, ambos, un medio. Para probar igualdad en números racionales debe transformárseles a fracciones reducidas. Una vez que se ha hecho, puede probarse la igualdad por medio de la simple comparación de numeradores y denominadores.

Se define una *fracción reducida* como un número racional para el que no existe entero que divida exactamente tanto al denominador como al numerador. Así, $1/2$, $2/3$ y $10/1$, son fracciones reducidas, mientras que $4/8$, $12/18$ y $15/6$ no lo son. En nuestro ejemplo, $2/4$ se reduce a $1/2$ de manera que los dos números racionales son iguales.

Puede usarse un procedimiento conocido como el algoritmo de Euclides para reducir un número racional a una fracción reducida cuando el número está en la forma *numerador/denominador*. Este procedimiento puede esbozarse como sigue:

1. Sea *a* el mayor y *b* el menor entre el *numerador* y el *denominador*.
2. Divídase *b* entre *a*, para encontrar un cociente y un resto, *q* y *r* respectivamente (es decir, $a = q * b + r$).
3. Hágase $a = b$ y $b = r$.
4. Repítase los pasos 2 y 3 hasta que *b* sea igual a 0.
5. Divida el *numerador* y el *denominador* por el valor de *a*.

Como ejemplo, represéntese 1032/1976 como fracción reducida:

paso 0	numerador = 1032	denominador = 1976
paso 1	a = 1976 b = 1032	
paso 2	a = 1976 b = 1032	q = 1 r = 944
paso 3	a = 1032 b = 944	
pasos 4 y 2	a = 1032 b = 944	q = 1 r = 88
paso 3	a = 944 b = 88	
pasos 4 y 2	a = 944 b = 88	q = 10 r = 64
paso 3	a = 88 b = 64	
pasos 4 y 2	a = 88 b = 64	q = 1 r = 24
paso 3	a = 64 b = 24	
pasos 4 y 2	a = 64 b = 24	q = 2 r = 16
paso 3	a = 24 b = 16	
pasos 4 y 2	a = 24 b = 16	q = 1 r = 8
paso 3	a = 16 b = 8	
pasos 4 y 2	a = 16 b = 8	q = 2 r = 0
paso 3	a = 8 b = 0	
paso 5	1032/8 = 129	1976/8 = 247

Así, 1032/1976, como fracción reducida es 129/247.

Escribamos una función para reducir un número racional (usamos el método de etiqueta para declarar racionales).

```
reduce (inrat, outrat)
struct rational *inrat, *outrat;
{
    int a, b, rem;
    if (inrat->numerator > inrat->denominator) {
        a = inrat->numerator;
        b = inrat->denominator;
    } /* fin de if */
    else {
        a = inrat->denominator;
        b = inrat->numerator;
    } /* fin de else */
    while (b != 0) {
        rem = a % b;
        a = b;
        b = rem;
    } /* fin de while */
    outrat->numerator /= a;
    outrat->denominator /= a;
} /* fin de reduce */
```

Al usar la función *reduce*, puede escribirse otra función *equal* que determine cuándo son iguales dos números racionales *r1* y *r2*. Si lo son, la función regresa TRUE, en otro caso, regresa FALSE.

```
#define TRUE 1
#define FALSE 0

equal (rat1, rat2)
struct rational *rat1, *rat2;
{
    struct rational r1, r2;
    reduce(rat1, &r1);
    reduce(rat2, &r2);
    if (r1.numerator == r2.numerator &&
        r1.denominator == r2.denominator)
        return(TRUE);
    return(FALSE);
} /* fin de equal */
```

Ahora pueden escribirse funciones para ejecutar aritmética sobre números racionales. Se presenta una función para multiplicar dos números racionales y se deja como ejercicio el problema de escribir funciones similares para la suma, la resta y la división de dichos números.

```
multiply (r1, r2, r3) /* r3 apunta al resultado */
struct rational *r1, *r2, *r3; /* multiplicando
                                r1 y r2 */
struct rational rat3;

rat3.numerator = r1->numerator * r2->numerator;
rat3.denominator = r1->denominator * r2->denominator;
reduce(&rat3, r3);
} /* fin de multiply */
```

Asignación de memoria y alcance de variables

Hasta ahora se ha visto la declaración de variables. Es decir, la descripción de un tipo de variable o atributo. Sin embargo restan dos problemas importantes que deben discutirse: ¿hasta qué punto una variable está asociada con un almacenamiento real (esto es, *asignación de memoria*)?, y ¿hasta qué punto puede referirse a una variable particular un programa (esto es, cuál es el *alcance* de las variables)?

En C, se conocen a las variables y parámetros que se declaran dentro de una función como variables *automáticas*. A éstas se les asigna memoria cuando se llama la función. Cuando termina la ejecución de la función, se pierde la asignación. Por ello las variables automáticas existen sólo mientras está activa la función. Además,

se dice que las variables automáticas son *locales* a la función. Es decir, las variables automáticas se conocen sólo dentro de la función donde están declaradas y otra función no puede referirse a ellas.

Las variables automáticas (es decir, parámetros en el encabezado de una función o variables locales que siguen de inmediato a cualquier llave de apertura) pueden declararse dentro de cualquier bloque y existen hasta que éste se acaba. Puede referirse a la variable a través de todo el bloque a menos que el identificador de la misma se declare de nuevo en un bloque interno. Dentro del bloque interno, una referencia al identificador es una referencia a la declaración más interna y no puede hacerse referencia a las variables de afuera.

La segunda clase de variables en C es la de las variables *externas*. A las variables que se declaran fuera de cualquier función se les asigna almacenamiento en el primer punto donde se les encuentra y existen para el resto de la ejecución del programa. El alcance de una variable externa va desde el punto en que se declara hasta el final del archivo fuente que la contiene. Todas las funciones del archivo fuente pueden referirse a este tipo de variables que estén más allá de su declaración y, en consecuencia, se dice que son *globales* para dichas funciones.

Un caso especial es cuando el programador desea definir una variable global en un archivo fuente y referirse en otro a la variable. Tal variable debe declararse en forma explícita como externa. Por ejemplo, supóngase que se declara un arreglo de enteros que representa una lista de grupos en el archivo fuente 1 y se desea referirse a él a lo largo del archivo fuente 2. Entonces, podrían necesitarse las siguientes declaraciones:

```
archivo 1      #define MAXSTUDENTS ...
                int grades[MAXSTUDENTS];
                ...
fin de archivo 1
```

```
archivo 2      extern int grades[];
                float average();
                {
                ...
                } /* fin de average */
                float mode()
                {
                ...
                } /* fin de mode */
fin de archivo 2
```

Cuando se combinan los archivos 1 y 2 dentro de un programa, se asigna memoria para el arreglo *grades* en el archivo 1 y la asignación permanece hasta el final del archivo 2. Dado que *grades* es una variable externa, es global desde el momento en

que se define dentro del archivo 1 hasta el final de archivo 1 y desde el momento en que se declara en el archivo 2 hasta el final del archivo 2. Por consiguiente, ambas funciones, *average* y *mode*, pueden referirse a *grades*.

Nótese que el tamaño del arreglo se especifica una sola vez, en el momento en el cual se define originalmente la variable. Esto ocurre porque una variable que se declara explícitamente como externa no puede volver a definirse, ni puede asignársele memoria adicional. Una declaración de tipo *external* (externa) sólo sirve para declarar en el resto del archivo fuente que existe tal variable y que ya se había creado.

En ocasiones es deseable definir una variable dentro de una función para la cual la asignación se mantenga de memoria a lo largo de la ejecución del programa. Por ejemplo, podría ser útil mantener un contador local en una función que indicara el número de veces que es llamada una función. Esto puede hacerse si se incluye la palabra *static* dentro de la declaración variable. Una variable interna estática (*static*) es local a esa función pero sigue existiendo a lo largo de la ejecución del programa en vez de que se le asigne memoria y después se le despeje cada vez que es llamada la función. Una variable estática retiene su valor cuando se termina de ejecutar la función y se le vuelve a activar. De manera similar, se le asigna memoria a una variable externa *static* sólo una vez, pero puede referirse a ella cualquier función que siga en el archivo fuente.

Para fines de optimización, podría ser útil dar instrucciones al compilador para mantener el almacenamiento de una variable particular en un registro de alta velocidad en lugar de la memoria ordinaria. A tal variable se le conoce como *variable registro* (*register*) y se define al incluir la palabra *register* en la declaración de una variable automática o en los parámetros formales de una función. Hay muchas restricciones sobre estas variables, que varían con cada máquina. El lector deberá de consultar los manuales apropiados para los detalles respecto a esas restricciones.

Las variables pueden inicializarse de forma explícita como parte de una declaración. A tales variables se les da conceptualmente sus valores iniciales antes de la ejecución. Todas las variables externas y estáticas no inicializadas comienzan con 0, mientras que las variables automáticas y de registro no inicializadas tienen valores indefinidos.

Para ilustrar esas reglas considérese el siguiente programa (los números a la izquierda de cada línea tienen sólo propósitos de referencia).

contenido de file1.c.

```
1  int x, y, z;
2  func1()
3  {
4      int a, b;
5      x = 1;
6      y = 2;
7      z = 3;
8      a = 1;
9      b = 2;
```

```

10     printf("%d %d %d %d\n", x, y, z, a, b);
11 } /* fin de func1 */
12
13 func2()
14 {
15     int a;
16     a = 5;
17     printf("%d %d %d %d\n", x, y, z, a);
18 } /* fin de func2 */
19
20 end of source file1.c
21
22 contenido de file2.c
23
24 #include <stdio.h>
25 #include <file1.c>
26
27 extern int x, y, z;
28
29 main()
30 {
31     func1();
32     printf("%d %d %d\n", x, y, z);
33     func2();
34     func3();
35     func4();
36     printf("%d %d %d\n", x, y, z);
37 } /* fin de main */
38
39 func3()
40 {
41     static int b; /* b se inicializa con 0 */
42     int x, y, z;
43     x = 10;
44     y = 20;
45     z = 30;
46     printf("%d %d %d\n", x, y, z);
47 } /* fin de func3 */
48
49 fin de archivo file2.c

```

La ejecución del programa nos conduce al siguiente resultado:

	a	b	c	d	e	f	g
1	1	2	3	1	2	10	20
2	1	2	3			20	30
3	1	2	3	5			
4	1	3	3	1			
5	1	4	3	2			
6	1	4	3				

Vayamos parte a parte del programa. La ejecución comienza en la línea 1, en la que se definen las variables enteras externas *x*, *y* y *z*. Al ser definidas como externas, se conocerán de modo global en lo que resta de *file 1.c* (líneas 1 y 7). La ejecución procede luego hasta la línea 20, que declara por medio de la palabra *extern* que las variables enteras externas *x*, *y* y *z* deben asociarse con las variables del mismo nombre en la línea 1. En este punto no se asigna ningún nuevo almacenamiento, dado que la memoria sólo se asigna cuando esas variables se definen desde el inicio (línea 1). Al ser externas, *x*, *y* y *z* se conocerán en lo que resta de *file 2.c*, con la excepción de *func4* (líneas 38 a 45), donde la declaración de variables locales automáticas *x*, *y* y *z* (línea 40) remplaza a la definición original.

La ejecución comienza con *main()*, en la línea 21, lo cual llama de inmediato a *func1*.*func1* (líneas 2 a 4), define las variables automáticas locales *a* y *b* (línea 4) y asigna valores a las variables globales (líneas 5 a 7) y a sus variables locales (líneas 8 a 9). La línea 10, en consecuencia, produce la primera línea de salida (línea a). Cuando termina de ejecutarse *func1* (línea 11) se elimina la asignación de memoria para las variables *a* y *b*. Así que ninguna otra función podrá referirse a dichas variables.

El control regresa entonces a la función principal (línea 24). La salida se da en la línea b. Luego se llama a *func2*.*func2* (líneas 12 a 17), define una variable automática local, *a*, a la cual se le asigna memoria (línea 14) y un valor asignado (línea 15). La línea 16 se refiere a las variables (globales) externas *x*, *y* y *z* antes definidas en la línea 1 y con valores asignados en las líneas de la 5 a la 7. La salida se da en la línea b. Nótese que sería ilegal para *func2* intentar imprimir un valor para *b*, dado que esta variable dejó de existir, al ser asignada sólo dentro de *func1*.

Después, el programa principal llama a *func3* dos veces (líneas 26 a 27). *func3* (líneas 31 a 37), le asigna memoria a la variable local estática *b* y la inicializa con 0 (línea 33), cuando se llama por primera vez. *b* será conocida sólo para *func3*, sin embargo existirá todo el resto de la ejecución del programa. La línea 34 incrementa la variable global y la línea 35 hace lo mismo con la variable local *b*. La línea d de la salida se imprime después. La segunda vez, el programa principal, llama a *func3*, sin asignar memoria a *b*; así que, cuando se incrementa *b* en la línea 35 se usa su valor antiguo (de la llamada previa a *func3*). El valor final de *b* reflejará, así, el número de veces que se ha llamado *func3*.

La ejecución continúa después en la función principal *main* que invoca a *func4* (línea 28). Como se había mencionado antes, la definición de las variables enteras automáticas e internas *x*, *y* y *z* en la línea 40 remplaza su definición en las líneas 1 y 20, y permanece vigente sólo durante la validez de *func4* (líneas 38 a 45). Así que la asignación de valores en las líneas 41 a 43 y la salida (línea f) resultante de la línea 44

se refiere sólo a esas variables locales. Tan pronto como termina *func4* (línea 45) se eliminan estas variables. Las referencias subsecuentes a *x*, *y* y *z* (línea 29) se entienden como referencias a las variables globales *x*, *y* y *z* (línea 1 y 20), que producen la salida de la línea *g*.

EJERCICIOS

1.3.1. Implantar números complejos, tal y como se especificó en el ejercicio 1.1.8, por medio de estructuras con partes real y compleja. Escriba programas para sumar, multiplicar y negar dichos números.

1.3.2. Supóngase que un número real está representado por una estructura en C como:

```
struct realtype {
    int left;
    int right;
};
```

donde *left* y *right* representan los dígitos de la derecha y la izquierda del punto decimal, respectivamente. Si *left* es un entero negativo, el número real representado es un número negativo.

- Escriba una rutina que acepte un número real y cree una estructura que represente este número.
- Escriba una función que acepte tal estructura y regrese el número real representado por ésta.
- Escriba rutinas *add*, *subtract* y *multiply* que acepten dos estructuras de ese tipo y den valor a una tercera estructura que represente el número suma, diferencia y producto, respectivamente, de los dos registros de entrada.

1.3.3. Supóngase que un entero necesita cuatro bytes, un número real ocho y un carácter 1 byte. Supóngase las siguientes declaraciones y definiciones:

```
struct nametype {
    char first[10];
    char midinit;
    char last[20];
};

struct person {
    struct nametype name;
    int birthday[2];
    struct nametype parents[2];
    int income;
    int numchildren;
    char address[20];
    char city[10];
    char state[2];
};

struct person p[100];
```

Si la dirección de comienzo de *p* es 100, ¿cuáles son las direcciones de comienzo (en bytes) de:

- p[10]*
- p[20].name.midinit*
- p[20].income*
- p[20].address[5]*
- p[5].parents[1].last[10]*

1.3.4. Supóngase dos arreglos, uno de registros de estudiantes y el otro de empleados. Cada registro de estudiante contiene miembros para el nombre, el apellido y un índice de calificación. Cada registro de empleado contiene miembros para el nombre, el apellido y el salario. Ambos arreglos se ordenan en orden alfabetico por apellido y nombre. Dos registros con el mismo nombre y apellido no pueden aparecer en el mismo arreglo. Escriba una función en C para dar un 10% de aumento a todos los empleados que tengan un registro de estudiante y cuyo índice de calificaciones sea mayor que 3.0.

1.3.5. Escriba una función como en el ejercicio anterior, pero suponga que los registros del empleado y el estudiante están guardados en dos archivos externos ordenados en lugar de en dos arreglos ordenados.

1.3.6. Por medio de la representación de números racionales dada en el texto, escriba rutinas para sumar, restar y dividir dichos números.

1.3.7. El texto presenta una función *equal* para determinar cuándo son iguales dos números racionales *r1* y *r2* por medio de probar igualdad entre las fracciones reducidas de ambos números. Un método alternativo puede ser multiplicar el numerador de cada uno por el denominador del otro para ver después si los productos así obtenidos son iguales. Escriba una función en C *equal2* para implantar este algoritmo. ¿Cuál de los dos métodos es preferible?

Pilas

Uno de los conceptos más útiles dentro de la ciencia de la computación es el de pila. En este capítulo examinaremos dicha estructura de datos, en apariencia sencilla, y veremos por qué tiene un papel tan prominente en las áreas de programación y lenguajes de programación. Definiremos el concepto abstracto de pila y mostraremos cómo puede convertirse en una herramienta concreta y valiosa para la resolución de problemas.

2.1. DEFINICION Y EJEMPLOS

Una *pila* es una colección ordenada de elementos en la que pueden insertarse y suprimirse por un extremo, llamado *tope*, nuevos elementos. Se puede representar una pila como en la figura 2.1.1.

A diferencia del arreglo, la definición de pila incorpora la inserción y supresión de elementos, de tal manera que ésta es un objeto dinámico constantemente variable. En consecuencia, surge la pregunta: ¿cómo cambia una pila? La definición especifica que un extremo de la pila se designa como el tope de la misma. Pueden agregarse nuevos elementos en el tope de la pila (en cuyo caso éste se mueve hacia arriba para corresponder al nuevo elemento que ocupa la cima), o pueden quitarse los elementos que están en el tope (y entonces el tope se mueve hacia abajo para corresponder al nuevo elemento que se encuentra hasta arriba). Para responder la pregunta: ¿cuál camino es hacia arriba? Tenemos que definir a uno de los extremos de la pila como su tope; es decir, en qué extremo pueden colocarse o suprimirse elementos. En la figura 2.1.1 se observa que el hecho que la *F* esté físicamente más arriba que el resto de los elementos de la pila, implica que es el elemento tope actual. Si

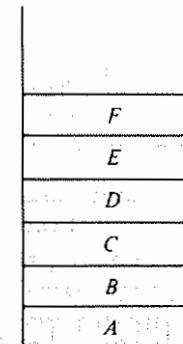


Figura 2.1.1 Una pila y sus elementos.

se agregan nuevos elementos a la pila, éstos deben colocarse encima de *F*, y si se suprime alguno, *F* será el primero en ser borrado. Esto se indica también por las líneas verticales, las cuales se extienden más allá de los elementos de la pila en dirección del tope de la misma.

En la figura 2.1.2a la pila, se encuentra en el mismo estado que el presentado en la figura 2.1.1. En la figura 2.1.2b se añade el elemento *G* a la pila. De acuerdo con la definición, hay sólo un lugar donde éste puede colocarse: en el tope. El elemento tope de la pila ahora es *G*. En las figuras c, d y e se agregan los elementos *H*, *I* y *J* sucesivamente. Obsérvese que el último elemento insertado (en este caso *J*) está en el tope de la pila. Sin embargo, a partir de *f*, la pila comienza a disminuir cuando al suprimirse *J*, y después *I*, *H*, *G*, y *F* en forma sucesiva. En cada punto el elemento tope de la pila varía, dado que cada supresión puede hacerse sólo en el tope. El elemento *G* no pudo quitarse de la pila antes que *J*, *I* y *H* se eliminaran. Esto último ilustra el atributo más importante de una pila: el último elemento insertado en una pila es el primero en eliminarse. Así, *J* se elimina antes que *I* porque aquella se añadió primero. Por esta razón, en algunas ocasiones a las pilas se les conoce como listas LIFO —last-in, first-out— (último en entrar, primero en salir).

Entre j y k, la pila dejó de disminuir y comenzó a las figuras crecer nuevamente cuando se añadió el elemento *K*. Sin embargo, este crecimiento es momentáneo, ya que la pila disminuye a sólo tres elementos en la figura n.

Nótese que no se puede distinguir la figura a y a la figura i si sólo se observa el estado de la pila en las dos situaciones. En ambos casos ésta contiene los mismos elementos en el mismo orden y con el mismo tope. No se mantiene un registro en la pila del hecho de que se añadieron y quitaron cuatro elementos en ese tiempo. Asimismo, no hay forma de distinguir entre las figuras d y f, o entre j y l. Si se necesita un control de los elementos intermedios que han estado en la pila, éste debe conservarse en alguna parte; pues no se registra en la pila misma.

De hecho hemos tenido un amplio panorama de lo que en realidad se observa en una pila. La verdadera imagen de una pila está dada por una vista desde el tope hacia abajo y no de una ojeada incidental. Así, en la figura 2.1.2, no existen diferencias perceptibles entre las figuras h y o. En cada caso, el elemento tope es *G*. Aunque la pila de h no es igual que la de o, la única manera de determinar esto es eliminar todos los elementos en ambas pilas y compararlos de forma individual. Aunque hemos estado viendo una sección transversal de las pilas para tener una mayor compren-

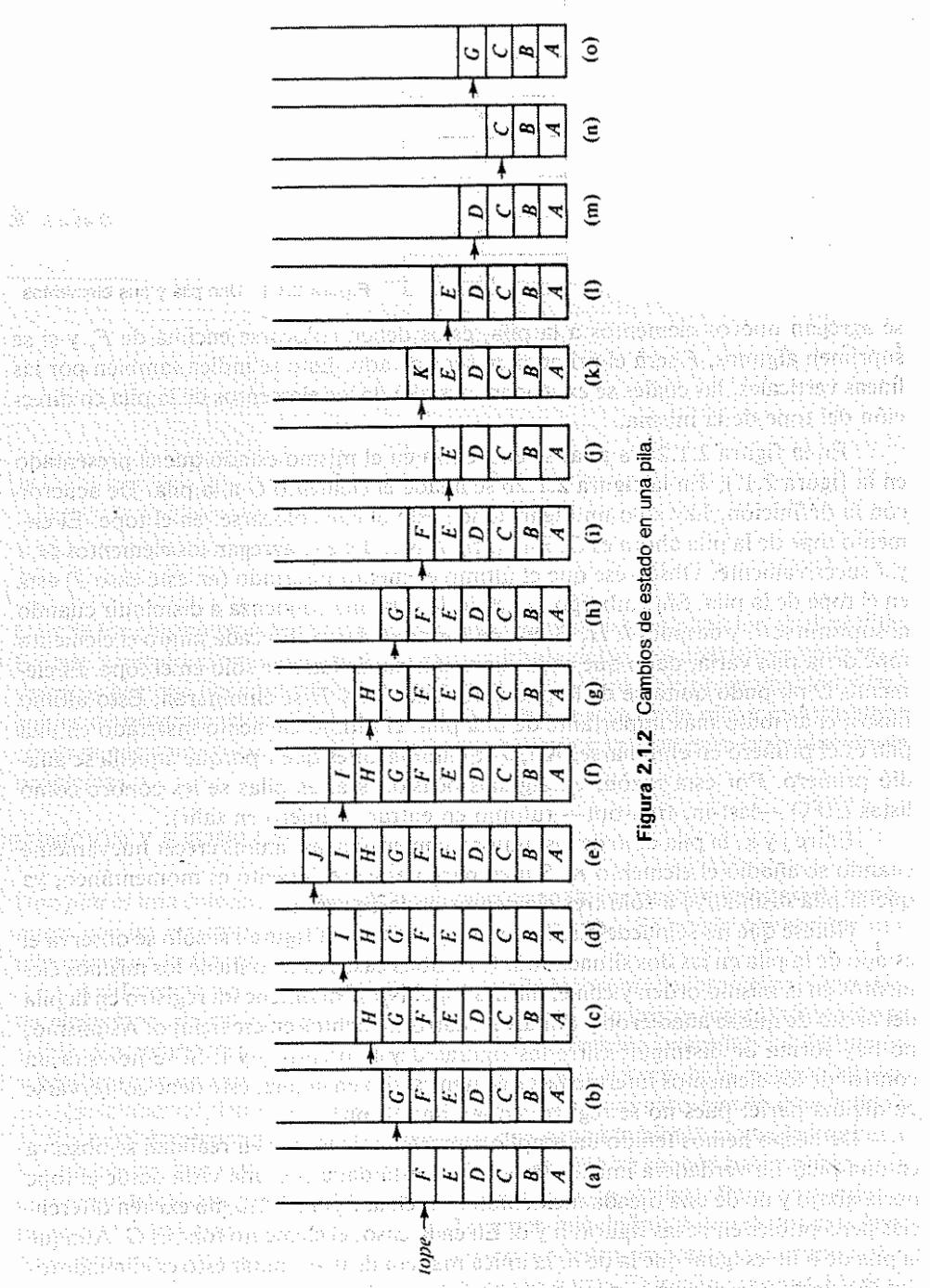


Figura 2.1.2 Cambios de estado en una pila.

sión, hay que señalar que esto es una libertad adicional y no existe ninguna cláusula que obligue a hacerlo así.

Operaciones primitivas

Los dos cambios que se pueden hacer en una pila reciben nombres especiales. Cuando se añade a una pila un elemento, decimos que se agregó (push) en la pila y cuando se suprime, decimos que se quitó (pop) de la pila. Dada una pila S y un elemento i , ejecutar la operación $\text{push}(s, i)$ añade el elemento i al tope de la pila s . De manera parecida, la operación $\text{pop}(s)$ elimina el elemento tope y lo regresa como el valor de la función. Así, la operación de asignación

$i = \text{pop}(s);$

Por ejemplo, si s es la pila de la figura 2.1.2, ejecutamos la operación $\text{push}(s, G)$ para pasar de la figura a a la b. Luego realizamos en forma ordenada las siguientes operaciones:

```

push (s,H);    (figura (c))
push (s,I);    (figura (d))
push (s,J);    (figura (e))
pop (s);        (figura (f))
pop (s);        (figura (g))
pop (s);        (figura (h))
pop (s);        (figura (i))
pop (s);        (figura (j))
push (s,K);    (figura (k))
pop (s);        (figura (l))
pop (s);        (figura (m))
pop (s);        (figura (n))
push (s,G);    (figura (o)).
```

A una pila se le conoce con frecuencia como *lista de apilamiento* debido a la operación push, que añade elementos a la pila.

No existe límite superior sobre el número de elementos que pueden almacenarse en una pila, dado que la definición no especifica cuántos elementos puede permitir la colección. Poner un nuevo elemento en la pila tan sólo ocasiona una mayor colección. Sin embargo, si una pila contiene un solo elemento y se elimina, la pila resultante no contiene elementos y se llama *pila vacía*. Aunque la operación push es aplicable a cualquier pila, no ocurre lo mismo con la operación pop que no puede aplicarse a la pila vacía, que no tiene ningún elemento que eliminar. En consecuencia, antes de aplicar a una pila la operación pop debe asegurarse de que no esté vacía. Si lo está $\text{empty}(s)$ determina si una pila s está o no vacía. Si lo está $\text{empty}(s)$ regresa el valor TRUE, de lo contrario regresa el valor FALSE.

Otra operación que puede ejecutarse en una pila es la de determinar el valor del tope de una pila sin eliminarlo. Esta operación se escribe como $stacktop(s)$ y regresa como resultado el elemento tope s de la pila. La operación $stacktop(s)$ en realidad no es una operación nueva dado que puede descomponerse en las operaciones pop y $push$.

$i = stacktop(s);$

es equivalente a

$i = pop(s);$
 $push(s, i);$

Así como la operación pop , $stacktop$ no está definida para la pila vacía. El resultado de un intento no permitido de accesar o eliminar un elemento de una pila vacía se conoce como *subdesborde*. Lo cual puede evitarse al asegurarse de que $empty(s)$ es falsa antes de intentar la operación $pop(s)$ o $stacktop(s)$.

Un ejemplo

Ahora que ya se ha definido una pila y se han especificado las operaciones que pueden realizarse en ella, se verá cómo puede usarse una pila en la resolución de problemas. Considérese una operación matemática que contenga varios conjuntos de paréntesis anidados, por ejemplo,

$$? - ((X * ((X + Y) / (J - 3))) + Y) / (4 - 2.5)$$

Quiere asegurarse de que los paréntesis estén anidados en forma correcta; es decir, quiere comprobarse:

1. Que hay el mismo número de paréntesis izquierdos y derechos.
2. Que todo paréntesis derecho esté precedido por su pareja izquierdo.

Expresiones del tipo

$$((A + B)) \quad o \quad A + B($$

violan la condición 1 y las del tipo

$$A + B(-C) \quad o \quad (A + B)) - (C + D)$$

violan la condición 2.

Para resolver este problema, piénsese que cada paréntesis izquierdo abre un ámbito y cada paréntesis derecho lo cierra. La *profundidad de anidamiento* en un lugar particular de una expresión es el número de ámbitos que se han abierto pero que aún no se cierra. Es lo mismo que el número de paréntesis izquierdos encontrados

sin que se haya encontrado su pareja derecha. Se define el *conteo de paréntesis* en un lugar determinado de una expresión, como el número de paréntesis izquierdos menos el número de paréntesis derechos que se han encontrado al recorrer la expresión desde su extremo izquierdo hasta el lugar determinado antes. Si el conteo de paréntesis no es negativo, coincide con la profundidad de anidamiento. Las dos condiciones que deben cumplirse si los paréntesis forman una pauta admisible en una expresión son las siguientes:

1. El conteo de paréntesis es 0 al final de la expresión. Esto implica que no se ha dejado abierto ningún ámbito o que se encontraron tantos paréntesis izquierdos como derechos.
2. El conteo de paréntesis no es negativo en ningún lugar de la expresión. Esto implica que no se encontró ningún paréntesis derecho para el cual no se hubiera encontrado con anterioridad su pareja izquierda.

En la figura 2.1.3 se da el conteo de paréntesis de forma directa abajo de cada lugar en cada una de las cinco cadenas. Como sólo la primera cumple las condiciones mencionadas antes, es la única que tiene una pauta correcta de paréntesis.

Cambiemos ahora un poco el problema y supóngase que existen tres tipos diferentes de delimitadores de ámbito, los cuales se indican mediante paréntesis ((y)), corchetes ([y]) y llaves, {{y}}. El símbolo que cierra el ámbito debe ser del mismo tipo que el símbolo que lo abre. Así, cadenas como las siguientes no están permitidas.

$$(A + B], [(A + B)], \{A - (B)\}$$

$$7 - ((X * ((X + Y) / (J - 3))) + Y) / (4 - 2.5))$$

0 0 1 2 2 2 3 4 4 4 4 3 3 4 4 4 3 2 2 2 1 1 2 2 2 2 1 0

$$((A + B)$$

1 2 2 2 2 1

$$A + B ($$

0 0 0 1

$$) A + B (- C$$

-1 -1 -1 -1 0 0 0

$$(A + B)) - (C + D$$

1 1 1 1 0 -1 -1 0 0 0

Figura 2.1.3 Conteo de paréntesis en varios puntos de la cadena.

Es necesario tener en cuenta no sólo cuántos ámbitos se abren sino también de qué tipo son. Se necesita esta información porque cuando se encuentra un símbolo que cierra un ámbito, debe conocerse el símbolo con el que fue abierto para asegurarse de que se ha cerrado en forma correcta.

Una pila puede usarse para rastrear la cantidad de símbolos de un tipo dado que se han encontrado. Cuando se encuentra un símbolo que abre un ámbito, dondequiera que ocurra, se coloca en la pila. Dondequiera que se encuentre un símbolo de cierre, se examina la pila. Si la pila está vacía, el símbolo que cierra no tiene su pareja correspondiente y la cadena en cuestión no es válida. Sin embargo, si la pila no está vacía, extraemos elementos de la misma por medio de *pop* y vemos si alguno corresponde al símbolo de cierre encontrado. Si hay correspondencia continuamos; si no la hay, la cadena no es válida. Cuando se alcanza el final de la cadena, la pila tiene que estar vacía, pues de otro modo se abrieron paréntesis que luego no se cerraron y la cadena no es válida. El algoritmo para este procedimiento se da a continuación. La figura 2.1.4 muestra el estado de la pila después de leer en partes la cadena.

```

    /* Este programa determina si una cadena dada es válida o no. Una cadena es válida si cada carácter que abre un ámbito se cierra con el mismo carácter que lo abrió. Se supone que la cadena es válida.
    * s = la pila vacía
    while (no hemos leído toda la cadena) {
        lee el siguiente símbolo (symb) de la cadena:
        if (symb == '(' || symb == '[' || symb == '{')
            push(s, symb);

        if (symb == ')' || symb == ']' || symb == '}')
            if (empty(s))
                valid = false;
            else {
                i = pop(s);
                if (i no es el símbolo de inicio de symb)
                    valid = false;
            } /* end else */
        } /* end while */
        if (!empty(s))
            valid = false;

        if (valid)
            printf("%s", "la cadena es válida");
        else
            printf("%s", "la cadena no es válida");
    }

```

Veamos por qué la solución de este problema precisa el uso de una pila. El último ámbito abierto debe ser el primero en cerrarse, lo cual se simula mediante una pila en la que el último elemento que llega es el primero en salir. Cada elemento de la pila representa la apertura de uno de estos ámbitos (llaves, paréntesis o corchetes) que no se han cerrado todavía. Poner un elemento dentro de la pila corresponde a abrir un ámbito y sacar un elemento de la pila corresponde a cerrar un ámbito, y se deja un ámbito menos abierto.

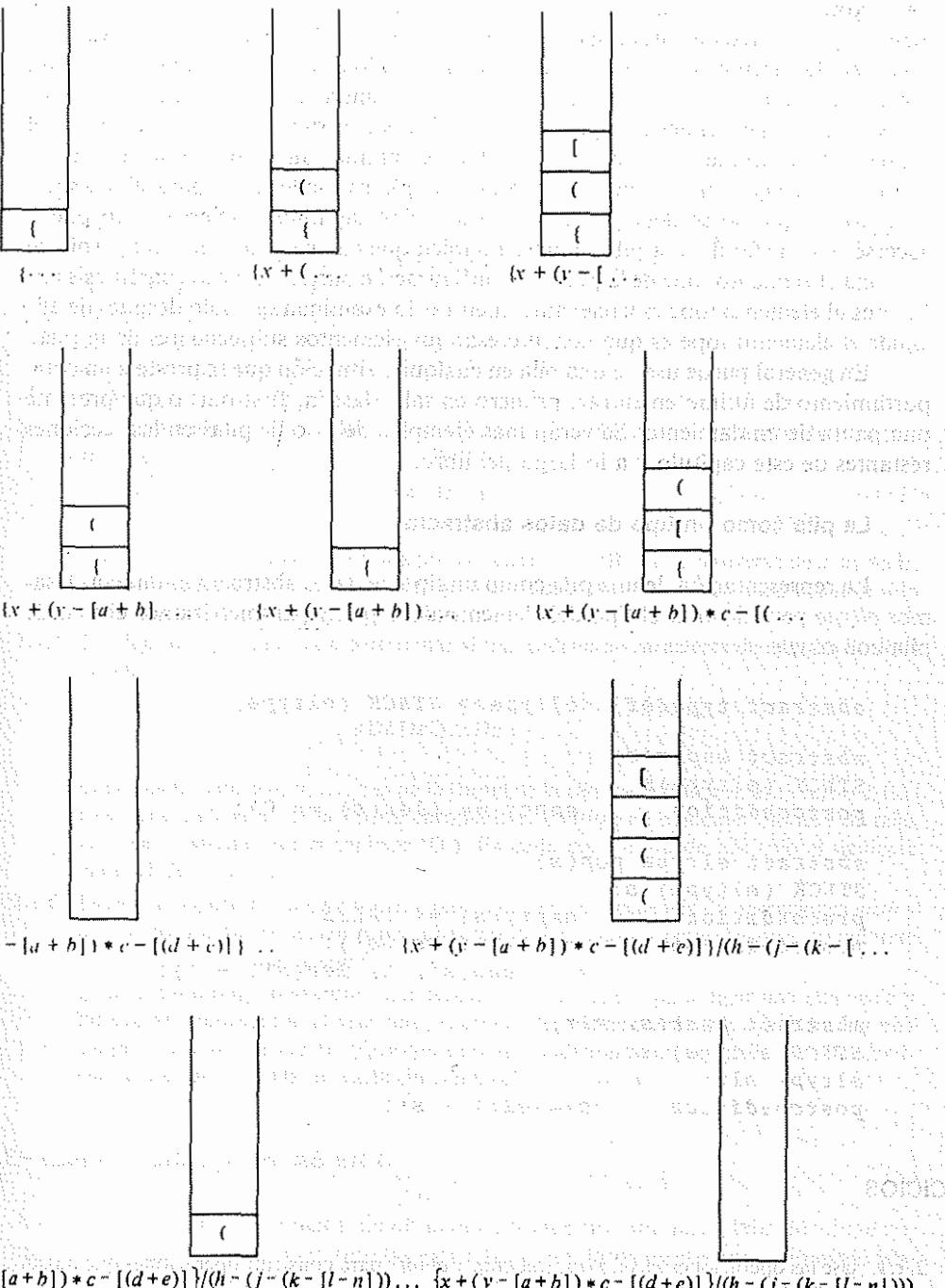


Figura 2.1.4 La pila de paréntesis en varios estados del procesamiento.

Nótese la correspondencia entre el número de elementos en la pila en este ejemplo y el conteo de paréntesis en el ejemplo anterior. Cuando la pila está vacía (y el conteo de paréntesis es 0) y se encuentra un símbolo de cierre de ámbito, se intenta cerrar un ámbito que nunca se había abierto, de manera que la pauta de paréntesis no es válida. En el primer ejemplo, ésto se indica por medio de un valor negativo del conteo y en el segundo por la imposibilidad de eliminar un elemento de la pila. La razón por la cual un conteo de paréntesis simple es inadecuado para el segundo ejemplo, es que no se debe perder de vista el tipo de ámbito abierto. Esto puede hacerse por medio de una pila. Nótese también que en cualquier momento, sólo se examina el elemento tope de la pila. La configuración particular de los paréntesis anteriores al elemento tope es irrelevante mientras lo examinamos. Sólo después de eliminar el elemento tope es que nos interesan los elementos subsecuentes de la pila.

En general puede usarse una pila en cualquier situación que se preste a un comportamiento de último en entrar, primero en salir (last-in, first-out) o que presente una pauta de anidamiento. Se verán más ejemplos del uso de pilas en las secciones restantes de este capítulo y a lo largo del libro.

La pila como un tipo de datos abstracto

La representación de una pila como un tipo de datos abstracto es directa. Usamos *eltype* para denotar el tipo del elemento de la pila y parametrizamos el tipo de pila con *eltype*.

```
abstract typedef <<eltype>> STACK (eltype);

abstract empty(s)
STACK (eltype)s;
postcondition empty == (len(s) == 0);

abstract eltype pop(s)
STACK (eltype) s;
precondition empty(s) == FALSE;
postcondition pop == first(s');
s == sub(s', 1, len(s') - 1);

abstract push(s, elt)
STACK(eltype) s;
eltype elt;
postcondition s==<elt> + s';
```

EJERCICIOS

2.1.1. Use las operaciones *push*, *pop*, *stacktop* y *empty* para construir operaciones que hagan lo siguiente:

- Poner a *i* como el segundo elemento a partir del tope, dejando la pila sin los dos elementos tope.
- Poner a *i* como el segundo elemento a partir del tope, dejando la pila intacta.

c. Dado un entero *n*, poner a *i* como el *n*-ésimo elemento a partir del tope de la pila, dejando la pila sin los *n* elementos tope.

d. Dado un entero *n*, poner a *i* como el *n*-ésimo elemento a partir del tope, dejando la pila intacta.

e. Poner a *i* como el elemento del fondo de la pila, dejando la pila vacía.

f. Poner a *i* como el elemento del fondo de la pila dejando la pila intacta. (Sugerencia: usar una pila auxiliar extra.)

g. Poner a *i* como el tercer elemento a partir del fondo de la pila.

2.1.2. Simule la acción del algoritmo presentado en esta sección para cada una de las siguientes cadenas, mostrando los contenidos de la pila en cada paso.

a. $(A + B)$

b. $\{[A + B] - [(C - D)]$

c. $(A + B) - \{C + D\} - [F + G]$

d. $((H) * \{([J + K])\})$

e. $((((A))))$

2.1.3. Escriba un algoritmo para determinar si una cadena de caracteres de entrada es la forma

C

donde *x* es una cadena constituida por las letras 'A' y 'B' y *y* es el reverso de *x* (es decir, si *x* = "ABABBA", y tiene que ser igual a "ABBABA"). En cada paso se puede sólo leer el siguiente carácter de la cadena.

2.1.4. Escriba un algoritmo para determinar si una cadena de caracteres de entrada es de la forma

$aDbDcD...Dz$

donde cada cadena *a*, *b*, ..., *z* es de la forma de la cadena definida en el ejercicio 2.1.3. (Así, una cadena es de la forma apropiada si consiste en cualquier número de tales cadenas separadas por el carácter 'D'). En cada paso se puede leer sólo el siguiente carácter de la cadena.

2.1.5. Diseñe un algoritmo que no use una pila para leer una secuencia de operaciones *push* y *pop* y determinar si ocurre o no subdesborde en alguna operación *pop*. Implántese dicho algoritmo en lenguaje C.

2.1.6. ¿Qué conjunto de condiciones son necesarias y suficientes para dejar una pila vacía y no causar subdesborde al usar una secuencia de operaciones *push* y *pop* en una pila simple (initialmente vacía)? ¿Qué conjunto de condiciones son necesarias en tal secuencia para dejar una pila no vacía sin cambios?

2.2. REPRESENTACION DE PILAS EN C

Antes de programar la solución de un problema que use una pila, debe decidirse cómo representar una pila mediante las estructuras de datos existentes en nuestro lenguaje de programación. Como se verá, hay varias maneras de representar una pila en lenguaje C. Considérese ahora la más simple. A lo largo del libro, se presentarán otras representaciones posibles. Cada una, sin embargo, es tan sólo una implantación del concepto introducido en la sección 2.1 que tiene ventajas y desventajas en

cuanto a la manera exacta en que refleja el concepto abstracto de pila y el esfuerzo hecho por el programador y la computadora para usarla.

Una pila es una colección ordenada de elementos; el lenguaje C ya incluye un tipo de datos que es una colección ordenada de elementos: el arreglo. Por ello, es tentador comenzar un programa con la declaración de una variable *pila* como arreglo. Siempre que la solución de un problema requiera el uso de una pila. Sin embargo, una pila y un arreglo son dos cosas por completo diferentes. El número de elementos de un arreglo es fijo y se asigna mediante su declaración. En general, el usuario no puede cambiar dicho número. Por otra parte, una pila es en lo fundamental un objeto dinámico cuyo tamaño cambia constantemente en tanto se le agregan o quitan elementos.

Sin embargo aunque un arreglo no puede ser una pila, puede usarse como tal. Es decir, puede declararse un arreglo lo bastante grande para admitir el tamaño máximo de la pila. Durante la ejecución del programa, la pila puede crecer y contraerse dentro de su espacio reservado. El fondo fijo de la pila es un extremo del arreglo, mientras que su tope cambia en forma constante cuando se agregan o quitan elementos. Así, es necesario otro campo para que en cada paso de la ejecución del programa, registre la posición actual del elemento tope de la pila.

En consecuencia, puede declararse una pila en C, como una estructura que contiene dos objetos: un arreglo para guardar los elementos de la pila y un entero para indicar la posición del elemento tope actual dentro del arreglo; lo cual puede hacerse para una pila de enteros por medio de las declaraciones:

```
#define STACKSIZE 100
struct stack {
    int top;
    int items[STACKSIZE];
};
```

Una vez hecho, puede declararse una pila *s*, por medio de:

```
struct stack s;
```

Aquí, se supone que son enteros los elementos de la pila *s* contenidos en el arreglo *s.items* y que la pila nunca contiene más de *STACKSIZE* enteros. En este ejemplo, *STACKSIZE* se fija en 100 para indicar que la pila puede contener 100 elementos (desde *items[0]* hasta *items[99]*).

Por supuesto, no hay razón para restringir una pila que sólo contenga enteros; los elementos *items* podrían declararse fácilmente como *float items[STACKSIZE]* o *char items[STACKSIZE]* o cualquier otro tipo que se quiera dar a los elementos de la pila. De hecho, en caso de surgir la necesidad, una pila puede contener objetos de diferentes tipos al usar uniones en C. Así

```
#define STACKSIZE 100
#define INTGR 1
#define FLT 2
```

```
#define STRING 3
struct stackelement {
    int etype /* etype es igual a INTGR, FLT y STRING */
    /* dependiendo del tipo de */
    /* elemento correspondiente */
    union {
        int ival;
        float fval;
        char *pval; /* apuntador de la cadena */
    } element;
};

struct stack {
    int top;
    struct stackelement items[STACKTOP];
};

define una pila cuyos elementos pueden ser enteros, números de punto-flotante o cadenas, dependiendo del valor correspondiente de etype. Dada una pila s, declarada por
```

```
struct stack s;
```

se puede imprimir su elemento tope como sigue:

```
struct stackelement se;
se = s.items[s.top];
switch (se.etype) {
    case INTGR : printf("% d\n", se.ival);
    case FLT : printf("% f\n", se.fval);
    case STRING : printf("% s\n", se.pval);
} /* fin de switch */
```

Para simplificar, supondremos en el resto de esta sección que una pila sólo contiene elementos homogéneos (de manera que las uniones no son necesarias). El identificador *top* debe declararse siempre con entero, dado que su valor representa la posición del elemento tope de la pila dentro de los elementos del arreglo *items*. Por ello, si el valor de *s.top* es 4, hay 5 elementos en la pila: *s.items[0]*, *s.items[1]*, *s.items[2]*, *s.items[3]* y *s.items[4]*. Cuando se elimina un elemento de la pila cambia el valor de *s.top* a 3 para indicar que ahora sólo hay 4 elementos en ella y que *s.items[3]* es el elemento tope. Por otra parte, si se añade un nuevo objeto a la pila, el valor de *s.top* debe incrementarse en 1 para convertirse en 5 y el nuevo objeto, tiene que insertarse dentro de *s.items[5]*.

La pila vacía no contiene elementos y puede, en consecuencia, indicarse mediante *top* igual a *-1*. Para inicializar una pila *s* en el estado vacío, puede ejecutarse desde el inicio *s.top = -1*.

Para determinar durante la ejecución, si una pila está o no vacía, se verifica la condición *s.top == -1* en una instrucción *if*, tal y como sigue:

```

if (s.top == -1)
    /* la pila está vacía */
else
    /* la pila no está vacía */

```

Esta verificación corresponde a la operación *empty(s)* que se introdujo en la sección 2.1. De otro modo, puede escribirse una función que dé como resultado *TRUE* (verdadero) si la pila está vacía y *FALSE* (falso) en caso contrario, como sigue:

```

empty(ps)
struct stack *ps;
{
    if (ps->top == -1)
        return(TRUE);
    else
        return(FALSE);
} /* fin de empty */

```

Una vez que existe la función, se implanta la verificación para la vacuidad de la pila mediante la siguiente instrucción:

```

if (empty (&s))
    /* la pila está vacía */
else
    /* la pila no está vacía */

```

Nótese que la diferencia entre la sintaxis de la llamada de *empty* en el algoritmo de la sección previa y en el segmento de programa aquí presentado. En el algoritmo, *s* representaba una pila y la llamada a *empty* fue expresada como

```
empty(s)
```

En esta sección nos interesa la implantación real de una pila y sus operaciones. Como en el lenguaje C, los parámetros se transfieren por valor, la única manera de modificar el argumento transferido a una función, es pasar la dirección del argumento en lugar del argumento. Además, la definición original de C (por Kernighan y Ritchie) y muchos compiladores C más antiguos, no permiten pasar una estructura como argumento aun si su valor permanece inalterado. Así, en funciones como *pop* y *push* (que modifican sus argumentos estructurales) así como *empty* (que no lo hace), se adopta la convención de transferir la dirección de la estructura pila, en lugar de la pila misma. (Esta restricción se ha omitido en muchos compiladores nuevos.)

Puede resultar asombroso que los autores del presente libro se ocupen de definir la función *empty* cuando pudiera tan sólo escribirse *if s.top == -1*, cada vez que se quisiera verificar la condición de vacuidad. La razón es que se desea hacer más comprensibles los programas y hacer independiente de su implantación el uso de una pila. Una vez entendido el concepto de pila, la frase “*empty(&s)*” es más signifi-

cativa que la frase “*s.top == -1*”. Si se introdujera después una implantación mejor de pila, de tal manera que “*s.top == -1*” careciera de significado, tendrían que cambiarse todas las referencias al identificador de campo *s.top* en todo el programa. Por otra parte, la frase “*empty(&s)*” tendrá aún su significado, dado que es un atributo propio del concepto de pila y no de su implantación. Todo lo que se precisaría para revisar un programa con el objetivo de acomodar una nueva implantación de pila sería una posible revisión de la declaración de la estructura *stack* en el programa principal y la reformulación de *empty*. (También es posible que la forma de llamar a *empty* tenga que modificarse de manera que no use ninguna dirección.)

Un método importante para hacer un programa más claro y modifiable es el de reunir todos los sitios dependientes de la implantación en unidades pequeñas y fácilmente identificables. Este concepto se conoce como *modularización*, por el cual se aislan funciones individuales en *módulos* de bajo nivel, cuyas propiedades son fáciles de verificar. Esos módulos de bajo nivel pueden usarlos después rutinas más complejas, que no deben tratar los detalles de los módulos de bajo nivel sino sus funciones. A las mismas rutinas complejas pueden tratarlas después como módulos rutinas de nivel mayor, que las usan de modo independiente de sus detalles internos.

Un programador debe interesarse siempre por la claridad de lo que escribe. Un mínimo de atención a la claridad ahorrará mucho tiempo en la búsqueda y depuración de errores. Los programas medianos y grandes casi nunca serán del todo correctos cuando se corren por primera vez. Si se toman precauciones desde el momento en que se escribe, para asegurar que sea fácil de modificar y comprender, el tiempo total necesario para lograr que corra en forma correcta, se reducirá considerablemente. Por ejemplo, la instrucción *if* en la función *empty* podría remplazarse por la más corta y eficiente:

```
return (ps->top == -1);
```

Esta instrucción es la misma que la instrucción más grande:

```

if (s.top == -1)
    return (TRUE);
else return (FALSE);

```

Esto ocurre porque el valor de la expresión *s.top == -1* es *TRUE* si, y sólo si la condición *s.top == -1* es *TRUE*. Sin embargo, quien lea el programa lo hará probablemente en forma más sencilla al leer la instrucción *if*. Con frecuencia se descubrirá que al usar “trucos” de lenguaje al escribir programas, es imposible descifrarlos después de haberlos abandonado por uno o dos días.

Aunque es cierto que un programador de lenguaje C tiene, con frecuencia que ocuparse de la economía del programa, también es importante considerar el tiempo que se empleará sin duda en la búsqueda y depuración de errores. El profesional maduro (sea en C o en cualquier otro lenguaje) se interesa siempre por el balance apropiado entre la economía y la claridad de un programa.

Implantación de la operación *pop*

En la implantación de la operación de *pop*, tiene que considerarse la posibilidad de subdesborde, dado que el usuario puede intentar, sin saberlo, sacar un elemento de una pila vacía. Por supuesto que tal intento no está permitido y debe evitarse. Sin embargo, si se hiciera se informaría al usuario de la condición de desborde. En consecuencia se presenta una función *pop* que ejecuta las tres acciones siguientes:

1. Si la pila está vacía, imprime un mensaje de advertencia y detiene la ejecución.
2. Suprime el elemento tope de la pila.
3. Regresa el elemento al programa de llamada.

Se supondrá que la pila consiste en enteros, de tal manera que la operación *pop* puede implantarse como función, lo cual también sucedería si la pila consiste en algún otro tipo de variable simple. Sin embargo, si la pila está compuesta por estructuras más complejas (por ejemplo, una estructura o una unión), la operación *pop* se implantaría al regresar como resultado o un apuntador al elemento del tipo adecuado (en lugar del elemento mismo) o el valor eliminado de la pila como parámetro (en cuyo caso la dirección del parámetro transferiría en lugar del parámetro, de tal manera que la función *pop* modificalo el argumento real).

```
pop(ps)
{
    struct stack *ps;
    if (empty(ps)) {
        printf("%s", "subdesborde");
        exit(1);
    } /* fin de if */
    return(ps->items[ps->top--]);
} /*fin de pop*/
```

Nótese que *ps* ya es un apuntador a una estructura de tipo pila; de ahí que el operador “*&*” no se usa para llamar a *empty*. En todas las aplicaciones en C, tiene que distinguirse siempre entre apuntadores y objetos de datos reales.

Veamos con mayor detalle la función *pop*. Si la pila no está vacía, se retiene el elemento tope de la misma como valor resultante. Después este elemento se saca de la pila mediante la expresión *ps — > top — —*. Supóngase que cuando se llama a *pop*, *ps — > top* es igual a 87; es decir, hay 88 elementos en la pila. El valor de *ps — > items[87]* se regresa y el valor de *ps — > top* se cambia a 86. Obsérvese que *ps — > items[87]* aún retiene su antiguo valor; el arreglo *ps — > items* permanece sin cambio por la llamada a *pop*. Sin embargo, se modifica la pila, ya que ahora sólo contiene 87 elementos en lugar de 88. Recuérdese que una pila y un arreglo son dos objetos diferentes. El arreglo sólo proporciona un lugar para la pila. En sí la pila contiene sólo aquellos elementos entre el 0 y el *top*-ésimo del arreglo. Así, la reducción en 1 del valor de *ps — > top*, elimina de manera efectiva un elemento de la pila.

Esto es cierto a pesar de que *ps — > items[87]* conserva su antiguo valor.

Para usar la función *quitar*, el programador puede declarar *int x* y escribir

```
x = pop (&s);
```

entonces *x* contiene el valor eliminado de la pila. Si el propósito de la operación *pop* no es recuperar el elemento del tope de la pila sino eliminarlo, el valor de *x* no se volverá a usar en el programa.

Por supuesto, el programador debe asegurarse de que la pila no está vacía cuando llama la operación *pop*. Si no está seguro del estado de la pila, puede determinarse al programar

```
if (!empty(&s))
    x = pop (&s); /* si el resultado es NULL, el programa se detiene */
else
    /* efectúe una acción de corrección */
```

Si el programador llama sin percatarse a *pop* para una pila vacía, la función imprime el mensaje de error subdesborde y se detiene la ejecución. Aunque es algo desafortunado, es mejor de lo que ocurriría si se hubiese omitido por completo la instrucción *if* en la rutina *pop*. En ese caso, el valor de *s.top* sería —1 y se intentaría dar acceso al elemento no existente *s.items[—1]*.

El programador deberá estar prevenido ante la posibilidad casi segura de error; lo cual puede hacerse al incluir diagnósticos significativos en el contexto del problema. Al hacerlo, el programador podrá señalar con precisión el origen de los errores y emprender acciones correctivas de manera inmediata, si ocurren.

Verificación de condiciones excepcionales

En el contexto de un cierto problema, puede no ser necesario detener la ejecución de manera inmediata al detectar subdesborde. En su lugar, podría ser más conveniente para la rutina *pop* señalar al programa de llamada que esto ha ocurrido. La rutina de llamada al detectar dicha señal puede tomar la acción correctiva. Se llamará al procedimiento que elimina un elemento de la pila y regresa una indicación si ha ocurrido subdesborde: *popandtest*:

```
popandtest(ps,px, pund)
{
    struct stack *ps;
    int *pund, *px;
    if (empty(ps)) {
        *pund = TRUE;
        return;
    } /* fin de if */
    *pund = FALSE;
    *px = ps->items[ps->top--];
    return;
} /* fin de popandtest */
```

En el programa de llamada el programador escribiría

```
popandtest(&s, &x, &und);
if (und)
    /* efectúa una acción correctiva */
else
    /* usa el valor de x */
```

Implantación de la operación push

Examinemos ahora la operación *push*. Tal parece que esta operación puede ser muy fácil de implantar al usar la representación por arreglo de una pila. Un primer intento para el procedimiento *push* podría ser el siguiente:

```
push(ps, x)
struct stack *ps;
int x;
{
    ps->items[++(ps->top)] = x;
    return;
} /* fin de push */
```

Esta rutina hace lugar al elemento *x* que se agregará a la pila al incrementar *s.top* en 1, y después inserta *x* en el arreglo *s.items*.

La rutina implanta de manera directa la operación *push* presentada en la última sección. Sin embargo, tal y como se declaró es del todo incorrecta. Permite que se presente un sutil error causado por el empleo de la representación por arreglo de la pila. Recuérdese que una pila es una estructura dinámica que puede en forma constante crecer o contraerse y cambiar así su tamaño. Por otro lado, un arreglo es un objeto fijo de tamaño predeterminado, así que es muy concebible que una pila crezca más que el arreglo reservado para almacenarla, lo cual ocurre cuando el arreglo está lleno, es decir, cuando la pila contiene tantos elementos como el arreglo y se intenta añadirle uno más. El resultado de dicho intento se llama *desborde*.

Supóngase que el arreglo *s.items* está lleno y se llama la rutina de C *push*. Recuérdese que la primera posición del arreglo es 0 y el tamaño arbitrario elegido (*STACKSIZE*) para el arreglo *s.items* es 100. La condición *s.top == 99* indica entonces que el arreglo está lleno, de tal manera que la posición 99 (el 100 elemento del arreglo) es el elemento tope en ese momento. Cuando se llama a *push*, *s.top* se aumenta a 100 y se intenta insertar *x* en *s.items[100]*. Por supuesto, el límite superior de *s.items* es 99 y el intento de inserción conduce a un error impredecible, dependiendo de las localidades de memoria que siguen a la última posición del arreglo. Puede producirse un mensaje de error probablemente no relacionado con su causa.

En consecuencia, el procedimiento *push* debe revisarse de manera que se lea como sigue:

```
push(ps, x)
struct stack *ps;
int x;
{
    if (ps->top == STACKSIZE-1) {
        printf("%s", "desborde de la pila");
        exit(1);
    }
    else
        ps->items[++(ps->top)] = x;
    return;
} /* fin de push */
```

Aquí, se comprueba si el arreglo está lleno antes de intentar añadir otro elemento a la pila. El arreglo está lleno si *ps->top == stacksize - 1*.

Debe observarse de nuevo que siempre y cuando se detecte desborde en *push*, se detiene de manera inmediata la ejecución después de imprimir un mensaje que indica el error. Esta acción, como en el caso de *pop*, puede no ser la más deseable. En algunos casos podría tener más sentido para la rutina de llamada invocar la operación *push* con las siguientes instrucciones

```
pushandtest(&s, x, &overflow);
if (overflow)
    /* se ha detectado desborde, no se colocó
     * x en la pila. Ejecute acción correctiva */
else
    /* se colocó exitosamente a x en la pila
     * continúe el procesamiento */
    /* ... */
```

Esto permite al programa de llamada proceder después de la llamada a *pushandtest* tanto si se detectó o no desborde. La subrutina *pushandtest* se deja como ejercicio al lector.

Aunque las condiciones de desborde y subdesborde se han tratado de manera similar en *push* y *pop*, existe una diferencia fundamental entre ellas. El subdesborde indica que la operación *pop* no puede ejecutarse en la pila, al señalar un posible error en los datos o en el algoritmo. Ninguna otra implantación o representación de la pila resolverá la condición de subdesborde. En su lugar, debe reconsiderarse el problema por completo. (Por supuesto, podría ocurrir subdesborde como una señal de culminación de un proceso e inicio de otro. Pero en tal caso, debe usarse *popandtest* en lugar de *pop*).

El desborde, sin embargo, no es una condición aplicable a una pila como estructura de datos abstracta. De manera abstracta, es posible siempre poner un elemento en la pila. Una pila sólo es un conjunto ordenado y no hay límite para el número de elementos que dicho conjunto puede contener. La posibilidad de desborde se presenta cuando se implanta la pila por medio de un arreglo que contiene un número finito de elementos, con lo cual se prohíbe el crecimiento de la pila por encima de dicho número. Puede suceder que el algoritmo que usa el programador sea

correcto, sólo que su implantación no anticipa que la pila puede llegar a ser tan grande. Así que, en algunos casos la condición de desborde se puede corregir cambiando el valor de la constante *STACKSIZE*, de tal manera que el arreglo *items* contenga más elementos. No es necesario cambiar las rutinas *pop* o *push*, dado que se refieren a cualquier estructura de datos que se haya declarado para el tipo *stack* en las declaraciones del programa, *push* también se refiere a la constante *STACKSIZE* y no al valor 100.

Sin embargo, es más frecuente que un desborde indique un error en el programa, no atribuible a una simple falta de espacio. El programa puede caer en un ciclo infinito en el cual con frecuencia se añaden elementos a la pila sin que se elimine nada. Así, la pila crecerá más que el límite del arreglo sin importar qué tan alto se haya fijado. El programador siempre debe verificar que no suceda lo anterior antes de aumentar en forma indiscriminada el límite del arreglo. Con frecuencia el tamaño máximo de la pila puede determinarse con facilidad a partir del programa y sus entradas, de tal manera que si ocurre desborde, quizás hay algo equivocado con el algoritmo que representa el programa.

Veamos ahora nuestra última operación con pilas, *stacktop(s)* que regresa sin eliminarlo el elemento tope de una pila. Como se señaló en la última sección, *stacktop* en realidad no es una operación primitiva porque puede descomponerse en las dos operaciones:

```
x = pop(s);
push (s, x);
```

Sin embargo, es una manera más bien torpe de recuperar el elemento tope de la pila. ¿Por qué no ignorar la descomposición anterior y recuperar en forma directa el valor apropiado? Por supuesto, tendría entonces que plantearse de manera explícita una verificación de vacuidad y subdesborde, dado que la prueba no está más dirigida por una llamada a *pop*.

Presentamos una función C *stacktop* para una pila de enteros de la siguiente manera:

```
stacktop(ps)
struct stack *ps;
{
    if (empty(ps)) {
        printf("%s", "subdesborde de la pila");
        exit(1);
    }
    else
        return(ps->items[ps->top]);
} /* fin de stacktop */
```

Puede preguntarse por qué se molesta en escribir una rutina separada para *stacktop*, cuando una referencia a *s.items[s.top]* serviría para lo mismo. Hay varias razones para hacerlo. Primera, la rutina *stacktop* incorpora una verificación de subdesborde, de tal manera que no ocurran errores misteriosos si la pila está vacía.

Segunda, permite al programador usar la pila sin preocuparse por su modelo interno. Tercera, si se presenta una nueva implantación de la pila, el programador no necesita recorrer todo el programa para buscar en donde aparece *s.items[s.top]* cambiándolas para hacerlas compatibles con la nueva implantación. Sólo sería necesario cambiar la rutina *stacktop*.

EJERCICIOS

- 2.2.1. Escriba funciones en C que usen las rutinas presentadas en este capítulo para implantar las operaciones del ejercicio 2.1.1.
- 2.2.2. Dada una secuencia de operaciones *push* y *pop* y un entero que representa el tamaño de un arreglo en el cual se implantará una pila, diseñe un algoritmo para determinar si ocurre o no desborde. El algoritmo no debe usar la pila. Implemente el algoritmo como un programa en C.
- 2.2.3. Implemente los algoritmos de los ejercicios 2.1.3 y 2.1.4 en programas C.
- 2.2.4. Muestre cómo implantar una pila de enteros en C por medio del arreglo *int s[STACKSIZE]*, donde *s[0]* se usa para contener el índice del elemento tope, y *s[1]* hasta *s[STACKSIZE - 1]* para contener los elementos en la pila. Escriba una declaración y rutinas *pop*, *push*, *empty*, *popandtest*, *stacktop* y *pushandtest* para dicha implantación.
- 2.2.5. Implemente una pila en C en la cual cada elemento es un número variable de enteros. Seleccione una estructura de datos de C para la pila y diseñe rutinas *push* y *pop* para ella.
- 2.2.6. Considere un lenguaje que no tenga arreglos pero sí pilas como tipo de datos. Es decir, se puede declarar

stack s;

y están definidas las operaciones *push*, *pop*, *popandtest* y *stacktop*. Muestre cómo puede implantarse un arreglo unidimensional mediante dichas operaciones en dos pilas.

- 2.2.7. Diseñe un método para guardar dos pilas dentro de un arreglo lineal simple *\$[spacesize]* de tal manera que no ocurre ningún desborde mientras no se haya usado toda la memoria y una pila entera nunca se desplaza a una localidad de memoria diferente dentro del arreglo. Escriba rutinas en C: *push1*, *push2*, *pop1* y *pop2* para manipular las dos pilas. (Sugerencia: las dos pilas crecen en dirección a la otra.)

- 2.2.8. El estacionamiento Bashemini tiene un solo carril para estacionar hasta 10 coches. Hay una sola entrada/salida en un extremo del carril. Si llega un cliente a recoger su coche y éste no es el más próximo a la salida, se sacan todos los coches que bloquean su paso, se saca el coche del cliente en cuestión y el resto se vuelve a estacionar en el orden original.

Escriba un programa que procese un grupo de líneas de entrada. Cada línea de entrada contiene una 'E' para la entrada, una 'S' para la salida y un número de placas. Supóngase que los coches llegan y salen en el orden especificado por la entrada. El programa debe imprimir un mensaje siempre que un coche entre o salga. Cuando llega un coche, el mensaje debe especificar si hay lugar en el estacionamiento. Si no lo hay, el coche no entra. Cuando un coche se va, el mensaje debe incluir el número de veces que fue sacado del estacionamiento para permitir la salida de otros.

2.3. UN EJEMPLO: NOTACION INFIXA, PREFIJA Y POSTFIJA

Definiciones básicas y ejemplos

Esta sección examina una aplicación fundamental que ilustra los diferentes tipos de pila y las diversas operaciones y funciones definidas sobre ellas. El ejemplo es también por derecho propio un tema importante en la ciencia de la computación.

Considérese la suma de A y B . Se piensa en la aplicación del *operador* “+” a los *operando*s A y B y en escribir la suma como $A + B$. Esta representación en particular se llama *infija*. Hay otras dos notaciones para expresar la suma de A y B mediante los símbolos A , B y $+$, las cuales son:

$+ AB$ prefija

$AB +$ postfija

Los prefijos ‘pre-’, ‘post-’ e ‘in-’ se refieren a la posición relativa del operador respecto a los dos operandos. En la notación prefija el operador precede a los dos operando, en la postfija los sucede y en la infija está entre ambos. Las notaciones prefija y postfija no son en realidad tan difíciles de usar como podría parecer en principio. Por ejemplo, una función en C que regresa la suma de los dos argumentos A y B es llamada por *add(A, B)*. El operador *add* precede a los operando A y B .

Consideremos otros ejemplos. La evaluación de la expresión $A + B * C$, tal y como se escribe en notación infija estandar, requiere que se conozca cuál de las dos operaciones, $+$ o $*$, debe ejecutarse primero. En el caso de $+$ y $*$ “sabemos” que la multiplicación es prioritaria (en ausencia de paréntesis que indiquen lo contrario). Así, $A + B * C$ se interpreta como $A + (B * C)$ a menos que se especifique otra cosa. Decimos que la multiplicación tiene *precedencia* sobre la suma. Supóngase que se desea volver a escribir $A + B * C$ en postfija. Al aplicar las reglas de precedencia, se convierte primero la porción de la expresión evaluada en primer lugar, es decir, la multiplicación. Al hacer esta conversión en etapas

$A + (B * C)$ paréntesis para hacer hincapié

$A + (BC *)$ conversión de la multiplicación

$A(BC *) +$ conversión de la suma

$ABC * +$ forma postfija

Las únicas reglas que hay que recordar durante la conversión son que primero se convierten las operaciones de mayor prioridad y luego de que se ha convertido una porción de la expresión a postfija debe tratarse como un operando simple. Considérese el mismo ejemplo invirtiendo la precedencia de los operadores por medio de la inserción deliberada de paréntesis.

$(A + B) * C$ forma infija

$(AB +) * C$ conversión de la suma

$(AB +) C *$ conversión de la multiplicación

$AB + C *$ forma postfija

En este ejemplo la suma se convierte antes que la multiplicación dada la presencia de paréntesis. Para pasar de $(A + B) * C$ a $(AB +) * C$, los operando son A y B y $+$ es el operador. Para hacerlo de $(AB +) * C$ a $(AB +)C *$, $(AB +)$ y C son los operando y $*$ el operador. Las reglas para pasar de notación infija la postfija son sencillas, a condición de que se conozca el orden de precedencia.

Considérese cinco operaciones binarias: suma, resta, multiplicación, división y exponentiación. Las cuatro primeras están disponibles en C y se denotan por los operadores comunes $+$, $-$, $*$, y $/$. La quinta, la exponentiación, se representa mediante el operador $\$$. El valor de la expresión $A \$ B$ es A elevada a la potencia B , así que $3 \$ 2$ es 9. Para estos operadores binarios el orden de precedencia es el siguiente (de arriba hacia abajo):

exponentiación

multiplicación/división

suma/resta

Cuando se analizan operadores de la misma precedencia sin paréntesis, se supone que el orden es de izquierda a derecha excepto en el caso de la exponentiación, donde se supone de derecha a izquierda. Así, $A + B + C$ significa $(A + B) + C$ mientras que $A \$ B \$ C$ significa $A \$ (B \$ C)$. Al usar paréntesis se puede obviar la precedencia.

Se dan los siguientes ejemplos de conversión infija a postfija. Asegúrese de entender cada uno de ellos (y de que puede realizarlos) antes de proceder con el resto de la sección.

$(A + B) * (C - D)$

$A \$ B * C - D + E / F / (G + H)$

$((A + B) * C - (D - E)) \$ (F + G)$

$A -- B / (C * D \$ E)$

$AB + CD - EF / GH + / +$

$AB + C * DE -- FG + \$$

$ABCDE \$ / -$

Las reglas de precedencia para convertir una expresión de infija a prefija son idénticas. El único cambio de la conversión a postfija es que el operador se coloca antes que los operando y no después. Se presentan las formas prefijas de las expresiones anteriores. Una vez más, el lector debe intentar hacerlo.

Infija	Prefija
$A + B$	$+ AB$
$A + B - C$	$- + ABC$
$(A + B) * (C - D)$	$* + AB - CD$
$A \$ B * C - D + E / F / (G + H)$	$+ - * \$ABCD / / EF + GH$
$((A + B) * C - (D - E)) \$ (F + G)$	$\$- + ABC - DE + FG$
$A - B / (C * D \$ E)$	$- A / B * C \$ DE$

Obsérvese que la forma prefija de una expresión compleja no es la imagen especular de la forma postfija, como puede verse a partir del segundo de los ejemplos anteriores, $A + B - C$. De aquí en adelante sólo se considerarán las transformaciones a postfija y dejaremos como ejercicio al lector la mayor parte del trabajo que involucra la prefija.

Un detalle obvio de modo inmediato acerca de la notación postfija es que no requiere paréntesis. Considérense las dos expresiones $A + (B * C)$ y $(A + B) * C$. Aunque los paréntesis en una de las dos son superfluos [(por convención $A + B * C = A + (B * C)$)], los de la segunda expresión son necesarios para evitar confusión con la primera. Las formas postfijas de estas expresiones son

Infija	Postfija
$A + (B * C)$	$ABC * +$
$(A + B) * C$	$AB + C *$

No hay paréntesis en ninguna de las dos expresiones transformadas. El orden de los operadores en las expresiones postfijas determina el orden real de las operaciones en la evaluación de la expresión, haciendo innecesario el uso de paréntesis.

Para pasar de la forma infija a la postfija se sacrifica la capacidad de notar de un vistazo los operandos relacionados con un operador en particular. Sin embargo, se gana una forma no ambigua de la expresión original al eliminar el uso de paréntesis incómodos. En realidad, la forma postfija de la expresión original se vería más sencilla si no fuera por el hecho de que parece difícil de evaluar. Por ejemplo, ¿cómo sabemos que si $A = 3$, $B = 4$ y $C = 5$ en los ejemplos anteriores, entonces $2 \ 4 \ 5 \ * \ es \ igual \ a \ 23 \ y \ 3 \ 4 \ + \ 5 \ * \ a \ 25?$

Evaluación de una expresión postfija

La respuesta a la pregunta anterior se encuentra en el desarrollo de un algoritmo para evaluar expresiones dadas en notación postfija. Cada operador en una cadena postfija se refiere a los dos operandos previos en la cadena. (Por supuesto, uno de ellos puede ser el resultado de la aplicación previa de un operador.) Supóngase que cada vez que se lee un operando se pone dentro de una pila. Cuando se alcance un operador, sus operandos serán los dos elementos tope de la pila. Entonces pode-

mos extraer dichos elementos, ejecutar la operación indicada y poner el resultado en la pila de tal manera que esté disponible para usarse como operando del próximo operador. El siguiente algoritmo evalúa una expresión en notación posfija mediante este método:

```

opndstk = la pila vacía ; /* se crea una pila vacía */

/* recorre la cadena de entrada leyendo un elemento a la vez y colocándolo en symb */
/* while (no se detecte el final de la entrada) */

symb = siguiente caracter en la entrada ;
if (symb es un operando)
    push(opndstk, symb);
else {
    /* symb es un operador */ 
    opnd2 = pop(opndstk);
    opnd1 = pop(opndstk);
    = value resultado de aplicar symb a opnd1 y opnd2;
    push(opndstk, value);
}
/* fin de else */
} /* fin de while */
return(pop(opndstk));

```

Consideremos un ejemplo. Supóngase que se pide evaluar la siguiente expresión postfixia:

Mostramos los contenidos de la pila *opndstk* y las variables *symb*, *opnd1*, *opnd2* y *valúe* después de cada iteración sucesiva del ciclo. El tope de *opndstk* está a la derecha.

<i>symb</i>	<i>opnd1</i>	<i>opnd2</i>	<i>value</i>	<i>opndstk</i>
6			6	6
2			6,2	6,2
3			6,2,3	6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
*	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
\$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Cada operando se inserta en la pila cuando se encuentra. En consecuencia, el tamaño máximo de la pila es el número de operandos que aparecen en la expresión de entrada. Sin embargo, al tratar con la mayor parte de las expresiones postfijas, el tamaño real necesario para la pila es menor que este máximo teórico, dado que un operador saca elementos de la misma. En el ejemplo previo la pila nunca contiene más de cuatro elementos, a pesar de que aparecen ocho operandos en la expresión postfija.

Programa para evaluar una expresión postfija

Existen varias preguntas que deben considerarse antes de poder escribir en realidad un programa para evaluar una expresión en notación postfija. Una consideración primaria, como en todos los programas, es definir con precisión la forma y las restricciones, si las hay, en la entrada. Con frecuencia al programador se le presenta la forma de la entrada y se le pide diseñar un programa para acomodar los datos proporcionados. Por otra parte, estamos en la afortunada situación de poder escoger la forma de nuestra entrada. Esto permite construir un programa que no esté sobrecargado con problemas de transformación que opaquen el objeto real de la rutina. Al habernos enfrentado con datos en una forma, con la cual es difícil y engorroso trabajar, se podría haber relegado las transformaciones a varias funciones y usar las salidas de esas funciones como entradas de la rutina primaria. En el "mundo real", reconocer y transformar las entradas es un problema fundamental.

Supóngase en este caso que cada línea de entrada está en forma de cadena de dígitos y símbolos de operadores; que los operandos son dígitos simples no negativos, por ejemplo, 0, 1, 2, ..., 8, 9. Así, una línea de entrada podría contener 3 4 5 * + en las primeras cinco columnas seguidas de un carácter de fin-de-línea ('\n'). Se quiere escribir un programa que lea líneas de entrada en este formato, hasta que no quede ninguna, e imprima para cada línea, la cadena de entrada original y el resultado de la expresión evaluada.

Ya que los símbolos se leen como caracteres, debe encontrarse un método para convertir los caracteres operandos a números y los caracteres operadores a operaciones. Por ejemplo, debe tenerse un método para convertir el carácter '5' a número 5 y el carácter '+' a la operación de suma.

La conversión de un carácter a un entero puede manejarse fácilmente en C, si *int x* es un carácter que representa un dígito en C, la expresión *x* - '0' produce el valor numérico de ese dígito. Para implantar la operación correspondiente a un símbolo de operador, se usa una función *oper* que acepta la representación con caracteres y dos operandos como parámetros de entrada y da como resultado el valor de la expresión obtenida mediante la aplicación del operador a los dos operandos. El cuerpo de la función será presentado en breve.

El cuerpo del programa principal sería el siguiente. La constante *MAXCOLS* es el número máximo de columnas en una línea de entrada.

```
#define MAXCOLS 80
main()
{
    char expr[MAXCOLS];
    int position = 0;
    float eval();

    while ((expr[position++] = getchar ()) != '\n');
    expr[--position] = '\0';
    printf("%s%es", "la expresión postfija original es ", expr);
    printf("%f\n", eval(expr));
} /* fin de main */
```

Por supuesto, la parte principal del programa es la función *eval*, que se presenta a continuación. Esta función no es más que la implantación en C del algoritmo de evaluación, tomando en cuenta el formato y el medio específico de los datos de entrada y de las salidas calculadas. *eval* llama a una función *isdigit* que determina si su argumento es o no un operando. La declaración para una pila, que aparece abajo la usa la función *eval* que la sigue, así como las rutinas *pop* y *push* que son llamadas por *eval*.

```
struct stack {
    int top;
    float items[MAXCOLS];
};

float eval(expr)
char expr[];
{
    int c, position;
    float opnd1, opnd2, value;
    struct stack opndstk;
    opndstk.top = -1;
    for (position = 0; (c = expr[position]) != '\0';
        position++)
    if (isdigit(c))
        /* operando-- convertir el carácter que
           representa al dígito en float y
           se coloca en la pila */
        push(&opndstk, (float) (c - '0'));
    else {
        /*
           operador
           opnd2 = pop (&opndstk);
           opnd1 = pop (&opndstk);
           value = oper(c, opnd1, opnd2);
           push (&opndstk, value);
        */ /* fin de else */
        return (pop(&opndstk));
    } /* fin de eval */
```

Para completar se presentan las funciones *isdigit* y *oper*. La función *isdigit* verifica si su argumento es un dígito:

```
isDigit(symb)
char symb;
{
    return(symb >= '0' && symb <= '9');
}
```

Esta función está disponible como marco predefinido en la mayoría de los sistemas C.

La función *oper* verifica si su primer argumento es un operador válido, en cuyo caso, determina el resultado de su operación en los dos argumentos siguientes. Para la exponentiación, suponemos la existencia de una función *expon(op1, op2)*.

```
float oper(symb, op1, op2)
int symb;
float op1, op2;
float expon();
{
    switch (symb) {
        case '+': return (op1 + op2);
        case '-': return (op1 - op2);
        case '*': return (op1 * op2);
        case '/': return (op1 / op2);
        case '$': return (expon(op1, op2));
        default: printf("%s", "operación ilegal");
                  exit(1);
    } /* fin de switch */
} /* fin de oper */
```

Limitaciones del programa

Antes de dejar el programa, deben señalarse algunas de sus deficiencias. Entender aquello que no puede hacer un programa es tan importante como saber lo que puede hacer. Debiera ser obvio que intentar resolver un problema por medio de un programa produzca resultados incorrectos sin la más mínima huella de un mensaje tentar resolver un problema con un programa incorrecto, con lo cual se logra que el programa produzca resultados incorrectos sin la más mínima huella de un mensaje que indique el error. En estos casos, el programador no tiene ninguna indicación de que los resultados son erróneos y puede, por consiguiente, hacer juicios equivocados basados en dichos resultados. Por este motivo, es importante que el programador entienda las limitaciones del programa.

Una crítica fundamental a este programa es que no hace nada en términos de detectar y actuar de acuerdo con los errores. Si los datos en cada línea de entrada representan una expresión postfija válida, el programa funciona. Supóngase, sin embargo, que una línea de entrada tiene demasiados operadores u operandos o que éstos no se encuentran en la secuencia debida. Estos problemas podrían aparecer

como resultado de alguien que en forma inocente usa el programa para una expresión postfija que contiene números de dos dígitos, lo que conduce a un número excesivo de operandos. O quizás, el usuario del programa tiene la impresión de tratar con números negativos y estos tienen que introducirse con el signo menos, que es el mismo que se usa para denotar la operación de resta. Estos signos menos se tratan como operadores de resta, lo cual da como resultado un número excesivo de operadores. Dependiendo del tipo específico de error, la computadora puede tomar una de varias acciones (por ejemplo, detener la ejecución o imprimir resultados equivocados).

Supóngase que en la instrucción final del programa, la pila *opndstk* no está vacía. No recibimos mensajes de error (porque no especificamos ninguno) y *eval* regresa un valor numérico para una expresión que quizás estuvo planteada de manera incorrecta en principio. Supóngase que una de las llamadas a la rutina *pop* suscita la condición de subdesborde. Como no se usó la rutina *popandtest* para eliminar elementos de la pila, el programa se detiene. Esto parece poco razonable, dado que los datos defectuosos en una línea no deberían impedir el procesamiento de los datos de otras. De ninguna manera son éstos los únicos problemas que podrían surgir. Como ejercicio, pueden escribirse programas que acomoden entradas menos restrictivas y otros que detecten algunos errores de los ya mencionados antes.

Conversión de una expresión infija a postfija

Hasta aquí se han presentado rutinas para evaluar una expresión postfija. Aunque se ha discutido un método para transformar la notación infija a postfija, no se ha presentado hasta ahora un algoritmo para hacerlo. A ello se dirige ahora nuestra atención. Una vez que se ha construido dicho algoritmo, se tendrá la capacidad de leer una expresión infija y evaluarla, primero convirtiéndola a postfija y luego evaluándola como tal.

En la discusión previa se mencionó que las expresiones dentro de los paréntesis que son interiores deben convertirse primero a notación postfija de tal manera que puedan tratarse como operandos simples. De esta manera los paréntesis pueden eliminarse en forma sucesiva hasta que se convierta la expresión completa. El último par de paréntesis por abrir dentro de un grupo encierra la expresión que debe transformarse primero dentro de dicho grupo. Este comportamiento de último en entrar, primero en salir debiera sugerir de manera inmediata el uso de una pila.

Considérese las dos expresiones infijas $A + B * C$ y $(A + B) * C$ y sus versiones postfijas respectivas $ABC*+$ y $AB+C*$. En cada caso, el orden de los operandos es el mismo que en las expresiones infijas originales. Al examinar la primera expresión, $A + B * C$, el primer operando, A puede insertarse de manera inmediata dentro de la expresión postfija. De modo claro el símbolo $+$ no puede insertarse hasta después del segundo operando, que aún no se ha examinado. En consecuencia, tiene que guardarse aparte para recuperarse e insertarse en su posición adecuada. Cuando se examina el operando B , se inserta de manera inmediata después de A . Ahora, sin embargo, se han examinado dos operandos. ¿Qué impide recuperar e insertar $(+)$? La respuesta es, por supuesto, el operador $*$ que sigue y tiene precedencia sobre $+$. En el caso de la segunda expresión, el paréntesis que cierra indica que la operación $+$ debe realizarse primero. Recuérdese que en notación postfija, a

diferencia de la infija, el operador que aparece primero en la cadena es el que se aplica primero.

Dado que la prioridad desempeña un papel tan importante en la transformación de infija a postfija, supóngase la existencia de una función $prcd(op1, op2)$, donde $op1$ y $op2$ son caracteres que representan operadores. Esta función regresa *VERDADERO* si $op1$ tiene precedencia sobre $op2$ cuando $op1$ aparece a la izquierda de $op2$ en una expresión infija sin paréntesis; $prcd(op1, op2)$ regresa *FALSO* en otro caso. Por ejemplo, $prcd('+', '+')$ son *VERDADERO*, mientras que $prcd('+', '**')$ es *FALSO*.

Se presenta ahora un esbozo de un algoritmo para convertir una cadena infija sin paréntesis a una cadena postfija. Como se supone que no habrá paréntesis en la cadena de entrada, lo único que rige el orden en el cual aparecen los operadores en la cadena posfija es la precedencia. (Los números de línea que aparecen en el algoritmo se usarán para futuras referencias.)

```

1   opstk = la pila vacía;
2   while( no es fin de entrada) {
3       symb = siguiente carácter en la entrada;
4       if (symb es un operando)
            agrega symb a la cadena postfija
5       else {
            while(!empty(opstk) && prcd(stacktop(opstk),
                symb)) {
6               topsymb = pop(opstk);
                agrega topsymb a la cadena postfija;
8           } /* fin de while */
9           push(opstk, symb);
        } /* fin de else */
    } /* fin de while */
    /* extracción de los operadores restantes */
10  while (!empty(opstk)) {
11      topsymb = pop(opstk);
12      agregar topsymb a la cadena postfija
    } /* fin de while */
}

```

Simúlese el algoritmo con cadenas infijas del tipo “ $A * B + C * D$ ” y “ $A + B * C \$ D \$ E$ ” [donde ‘\$’ representa la exponentiación y $prcd('$', '$')$ es *FALSO*] para convencerse de su corrección. Nótese que en cada punto de la simulación, un operador en la pila tiene menos precedencia que todos los operadores que están por encima de él. Esto se debe a que la pila vacía inicial satisface de manera trivial esta condición y un operador se inserta dentro de la pila (línea 9) sólo si el operador que está en el tope en ese momento tiene precedencia menor que el que va a entrar.

¿Qué modificación deberá realizarse a este algoritmo para utilizar paréntesis? La respuesta es sorprendentemente corta. Cuando se lee un paréntesis tiene que insertarse dentro de la pila. Esto puede hacerse al establecer la convención de que $prcd(op, '(')$ es *FALSO* para cualquier operador op con excepción del paréntesis derecho. Además, se define $prcd('(', op)$ como *FALSO* para cualquier operador op .

[El caso de $op == '='$ se discutirá en breve.] Esto asegura que un operador que aparece después de un paréntesis izquierdo se inserte en la pila.

Cuando se lee un paréntesis de cierre, todos los operadores, hasta el primer paréntesis de apertura deberán extraerse de la pila y colocados en la cadena postfija. Esto puede hacerse al definir $prcd(op, ')')$ como *VERDADERO* para todos los operadores op distintos del paréntesis izquierdo. Cuando estos operadores se han eliminado de la pila y aparece el paréntesis de apertura, deberá realizarse una acción especial. El paréntesis que abre se deberá extraer de la pila y en compañía del paréntesis que cierra descartarán en vez de colocárseles en la cadena postfija o en la pila. Establezcamos $prcd('(', ')')$ como *FALSO*; lo cual asegura que al alcanzarse un paréntesis que abre, se salta el ciclo que inicia en la línea 6, con lo que el paréntesis de apertura no se coloca en la cadena postfija. Por lo tanto la ejecución continúa en la línea 9. Sin embargo, puesto que el paréntesis que cierra no se inserta en la pila, la línea 9 se remplaza por la instrucción

```

9     if (empty(opstk) || symb != ')')
        push(opstk, symb);
    else /* extraer el paréntesis que abre y descartarlo */
        topsymb = pop(opstk);

```

Con las convenciones anteriores para la función $prcd$ y la revisión de la línea 9, el algoritmo puede usarse para convertir cualquier cadena infija a postfija. Se resumen a continuación las reglas de precedencia para paréntesis:

$prcd(' ', op)$ = <i>FALSO</i>	para cualquier operador op
$prcd(op, '(') = FALSO$	para cualquier operador op que no sea ‘(’
$prcd(op, ')' = TRUE$	para cualquier operador op que no sea ‘)’
$prcd(')', op) = undefined$	para cualquier operador op (un intento de comparar estos dos operadores indicará un error)

Se ilustra este algoritmo con algunos ejemplos:

Ejemplo 1:

$A + B * C$

Los contenidos de $symb$, la cadena postfija y $opstk$ se muestran después de examinar cada símbolo. La pila $opstk$ se muestra junto con el valor del tope a la derecha.

	<i>symb</i>	<i>cadena postfija</i>	<i>opstk</i>
1	A	A	
2	+	A	+
3	B	AB	+
4	*	AB	* +
5	C	ABC	* + C
6		ABC *	* +
7		ABC * +	

Las líneas 1, 3 y 5 corresponden al examen de un operando; de ahí que el símbolo (*symb*) se coloca de manera inmediata en la cadena postfija. En la línea 2 se examina un operador y se encuentra que la pila está vacía, en consecuencia se inserta el operador en la pila. En la línea 4 la precedencia del nuevo símbolo (*) es mayor que la del símbolo en el tope de la pila (+); por lo que el símbolo nuevo se coloca en la pila. En los pasos 6 y 7 la cadena de entrada está vacía, por lo tanto se eliminan los elementos de la pila para colocarse en la cadena postfija.

Ejemplo 2:

$$(A + B) * C$$

<i>symb</i>	<i>cadena postfija</i>	<i>opstk</i>
(
A	A	(
+	A	(+
B	AB	(+
)	AB +	(+
*	AB + C	* +
C	AB + C	* +
	AB + C *	

En este ejemplo, cuando se encuentra el paréntesis derecho se extraen los elementos de la pila hasta que se encuentra un paréntesis izquierdo, en cuyo caso se descartan ambos. Al usar paréntesis para forzar un orden de precedencia diferente al que se supone en su ausencia, el orden de aparición de los operadores en la cadena postfija es diferente al del ejemplo 1.

Ejemplo 3:

$$(A - (B + C)) * D \$ E + F)$$

<i>symb</i>	<i>cadena postfija</i>	<i>opstk</i>
(
(((
A	A	((
-	A	((-
(A	((- (
B	AB	((- (
+	AB	((- (+
C	ABC	((- (+ BC
)	ABC +	((- (+ BC +
)	ABC + -	((- (+ BC + -
*	ABC + -	((- (+ BC + - *
D	ABC + - D	((- (+ BC + - * D
)	ABC + - D *	((- (+ BC + - * D
\$	ABC + - D *	((- (+ BC + - * D \$
(ABC + - D *	((- (+ BC + - * D \$ (
E	ABC + - D * E	((- (+ BC + - * D \$ (E
+	ABC + - D * E	((- (+ BC + - * D \$ (+
F	ABC + - D * E F	((- (+ BC + - * D \$ (+ F
)	ABC + - D * E F +	((- (+ BC + - * D \$ (+ F +
	ABC + - D * E F + \$	

¿Por qué el algoritmo de conversión parece tan complicado, mientras que el de evaluación parece tan sencillo? La respuesta es que el primero convierte al orden natural (es decir, primero aparece la operación que debe ejecutarse primero) a partir de un orden de precedencia (regido por la función *prcd* y la presencia de paréntesis). Como son muchas las combinaciones de elementos en el tope de la pila (si no está vacía) y los elementos que van a insertarse en la misma, se vuelve necesario un gran número de instrucciones con el fin de asegurar que se han cubierto todas las posibilidades. Por otra parte, en el otro algoritmo los operadores aparecen precisamente en el orden en que deben ejecutarse. Por este motivo los operandos pueden apilarse hasta que se encuentre un operador, momento en el cual la operación se ejecuta de manera inmediata.

La motivación que hay detrás del algoritmo de conversión es el deseo de sacar los operadores en el orden en el cual deben ejecutarse. En la resolución de este problema a mano, podrían seguirse instrucciones vagas que exigirían convertir de adentro hacia afuera. Esto funciona muy bien en el caso de seres humanos que resuelven un problema con papel y lápiz (si no se confunden o cometen un error). Sin embargo un programa o algoritmo debe ser más preciso en sus instrucciones. No se puede estar seguro de haber alcanzado el paréntesis más interno o el operador con mayor precedencia hasta que se hayan explorado símbolos adicionales. En ese momento debe regresarse a algún punto previo.

En lugar de explorar hacia atrás de manera continua se hace uso de la pila para “recordar” los elementos encontrados con anterioridad. Si se encuentra un operador de precedencia mayor al que se encuentra en el tope de la pila, se le inserta el nuevo operador, lo cual significa que cuando todos los elementos finalmente se extraen de la pila este nuevo operador precederá al que estaba en el tope dentro de la cadena postfija (lo cual es correcto dado que tiene mayor precedencia). Si, por otra parte, la precedencia del nuevo operador es menor, el operador del tope de la pila debe ejecutarse primero. En consecuencia, el tope de la pila se saca y el símbolo que va a entrar se compara con el nuevo tope, y así sucesivamente. Los paréntesis en la cadena de entrada modifican el orden de las operaciones. Así, cuando se encuentra un paréntesis izquierdo, se inserta en la pila. Cuando se encuentra el paréntesis derecho asociado a él, todos los operadores entre ambos se colocan en la cadena de salida, puesto que deberán ejecutarse antes de que los que aparecen después del paréntesis.

Programa para convertir una expresión de infija a postfija

Hay dos cosas que deben hacerse antes de que en realidad se inicie a escribir un programa. La primera es definir con precisión el formato de la entrada y de la salida. La segunda, construir, o al menos definir aquellas rutinas de las que depende el programa principal. Suponemos que la entrada consiste en cadenas de caracteres, una cadena por cada línea de entrada. El final de cada cadena se señala mediante un carácter de fin-de-línea ('/n'). Para simplificar suponemos que todos los operandos son letras o dígitos de un solo carácter. Todos los operadores y paréntesis se representan por ellos mismos y '\$' representa la exponentiación. La salida es una cadena de caracteres. Estas convenciones hacen adecuada la salida del proceso de conver-

sión para la evaluación, con tal de que sean dígitos todos los operandos de un solo carácter en la cadena inicial infija.

Para transformar el algoritmo de conversión en un programa se usan varias rutinas. Entre ellas se encuentran *empty*, *pop*, *push* y *popandtest*, todas modificadas de manera conveniente para que los elementos de pila sean caracteres. También se usa una función *isoperand* que regresa *VERDADERO* si su argumento es un operando y *FALSO* en caso contrario. Esta sencilla función la dejamos como ejercicio al lector.

De manera similar, la función *prcd* se deja como ejercicio al lector. Esta acepta dos símbolos de operadores de un solo carácter como argumentos y regresa *VERDADERO* si el primero tiene precedencia sobre el segundo cuando aparece a su izquierda en una cadena infija y *FALSO* en caso contrario. Por supuesto, la función incorpora las convenciones para los paréntesis presentadas con anterioridad.

Una vez escritas estas funciones auxiliares, puede escribirse la función de conversión *postfix* y un programa que la llame. El programa lee una línea que contiene una expresión en notación infija, llama a la rutina *postfix* e imprime la cadena postfixa. El cuerpo de la rutina principal es la siguiente:

```
#define MAXCOLS 80
main()
{
    char infix[MAXCOLS];
    char postr[MAXCOLS];
    int pos = 0;
    while ((infix[pos++] = getchar()) != '\n');
    infix[--pos] = '\0';
    printf("%s\n", "la expresión infija original es ", infix);
    postfix(infix, postr);
    printf("%s\n", postr);
} /* fin main */
```

La declaración para la pila de operadores y la rutina *postfix* se presentan a continuación:

```
struct stack {
    int top;
    char items[MAXCOLS];
};

postfix(infix, postr)
char infix [];
char postr [];
{
    int position, und;
    int outpos = 0;
    char topsymb = '+';
    char symb;
```

```
struct stack opstk;
opstk.top = -1; /* la pila vacía */

for (position=0; (symb = infix[position]) != '\0'; position++)
{
    if (isoperand(symb))
        postr[outpos++] = symb;
    else {
        popandtest (&opstk, &topsym, &und);
        while (!und && prcd(topsym, symb))
            postr[outpos++] = topsym;
        popandtest (&opstk, &topsym, &und);
    } /* fin de while */
    if (!und)
        push (&opstk, topsym);
    if (und || (symb != ')'))
        push(&opstk, symb);
    else
        topsym = pop (&opstk);
} /* fin de else */
while (!empty(&opstk))
    postr[outpos++] = pop(&opstk);
postr[outpos] = '\0';
return;
} /* fin de postfix */
```

El programa tiene un defecto fundamental que consiste en no verificar si la cadena de entrada es una expresión infija válida. En realidad, sería instructivo para el lector examinar la operación de este programa cuando se presenta con una cadena postfixa válida como entrada. Como ejercicio, proponemos escribir un programa que verifique si la cadena de entrada es una expresión infija válida.

Ahora puede escribirse un programa para leer una cadena infija y computar su valor numérico. Si la cadena original consiste en operandos de un solo dígito sin que sean letras, el siguiente programa lee la cadena original e imprime su valor.

```
#define MAXCOLS 80
main()
{
    char instring[MAXCOLS], postring[MAXCOLS];
    int position = 0;
    float eval ();
    while ((instring[position++] = getchar()) != '\n');
    instring[--position] = '\0';
    printf("%s\n", "la expresión infija es ", instring);
    postfix(instring, postring);
    print("%f\n", "el valor es", eval(postring));
} /* fin de main */
```

Se requieren dos versiones diferentes de rutinas para manipular la pila (*pop*, *push*, etc.) porque *postfix* usa una pila de operadores caracteres (es decir, *opstk*), mientras que *eval* usa una de operandos punto flotantes (es decir, *opndstk*). Por supuesto, es posible usar una sola pila que pueda contener tanto reales como caracteres al definir una unión tal y como se describió en la sección 1.3.

Nuestra mayor atención en esta parte se dirigió a las transformaciones que involucran expresiones postfijas. Un algoritmo para convertir una expresión infija en una postfija explora los caracteres de izquierda a derecha, apilándolos y desapilándolos cuando es necesario. Si se requiriera convertir de infija a prefija, la cadena infija debería explorarse de derecha a izquierda e introducirse los símbolos apropiados a la cadena prefija de derecha a izquierda. Dado que la mayoría de las expresiones algebraicas se leen de izquierda a derecha, la notación postfija es una elección más natural.

Los programas anteriores nada más indican el tipo de rutinas que podrían escribirse para manipular y evaluar expresiones postfijas. De ninguna manera son completos o únicos. Muchas de sus variantes también son aceptables. Algunos de los primeros compiladores de lenguaje de alto nivel en realidad usaban rutinas como *eval* y *postfix* para manejar expresiones algebraicas. Desde entonces se han desarrollado técnicas más sofisticadas para tratar dichos problemas.

EJERCICIOS

2.3.1. Transforme cada una de las siguientes expresiones a prefija y postfija.

- $A + B - C$
- $(A + B) * (C - D) \$ E * F$
- $(A + B) * (C \$ (D - E) + F) - G$
- $A + ((B - C) * (D - E) + F) / G \$ (H - J)$

2.3.2. Transforme cada una de las siguientes expresiones prefijas a infijas.

- $+ - ABC$
- $+ A - BC$
- $+ + A - * \$ BCD / + EF * GHI$
- $+ - \$ ABC * D ** EFG$

2.3.3. Transforme cada una de las siguientes expresiones postfijas a infija.

- $AB + C -$
- $ABC + -$
- $AB - C + DEF - + \$$
- $ABCDE - + \$ * EF * -$

2.3.4. Aplique el algoritmo de evaluación presentado en el libro para evaluar las siguientes expresiones postfijas. Supóngase $A = 1$, $B = 2$ y $C = 3$.

- $AB + C - BA + C \$ -$
- $ABC + * CBA - + *$

2.3.5. Modifique la rutina *eval* para que acepte como entrada una cadena de caracteres de operadores y operandos que representen una expresión postfija y cree la forma infija con paréntesis completos de la expresión postfija original. Por ejemplo, $AB +$ se transformará en $(A + B)$ y $AB + C -$ en $((A + B) - C)$.

2.3.6. Escriba un solo programa combinando las características de *eval* y *postfix* para evaluar una cadena infija. Use dos pilas, una para los operandos y otra para los operadores, no debe convertirse primero la cadena infija a postfija para luego evaluarse como postfija, sino hacerse sobre la marcha.

2.3.7. Escriba una rutina *prefix* para aceptar una cadena infija y crear la forma prefija de la misma, supóngase que la cadena se lee de derecha a izquierda, de la misma forma en que la cadena prefija se crea.

- 2.3.8. Escriba un programa en lenguaje C para convertir
- una cadena prefija a postfija
 - una cadena postfija a prefija
 - una cadena prefija a infija
 - una cadena postfija a infija

2.3.9. Escriba una rutina en C *reduce* que acepte una cadena infija y forme una cadena infija equivalente de donde se hayan eliminado todos los paréntesis innecesarios. ¿Puede hacerse sin usar una pila?

2.3.10. Supóngase una máquina que tiene un solo registro y seis instrucciones.

LD	A	poner el operando A dentro del registro
ST	A	poner los contenidos del registro en la variable A
AD	A	agrega los contenidos de la variable A al registro
SB	A	subtrae los contenidos de la variable A del registro
ML	A	multiplica los contenidos del registro por la variable A
DV	A	divide los contenidos del registro por la variable A

Escriba un programa que acepte una expresión postfija compuesta de operandos de una sola letra y de los operadores $+$, $-$, $*$ y $/$ e imprima una secuencia de instrucciones para evaluar la expresión y dejar el resultado en el registro. Use variables de la forma *TEMPn* como variables temporales. Por ejemplo, al usar la expresión postfija $ABC * + DE / -$ deberá imprimir lo siguiente:

```
LD    B
ML    C
ST    TEMP1
LD    A
AD    TEMP1
LD    B
ML    C
ST    TEMP2
LD    A
AD    TEMP2
LD    B
ML    C
ST    TEMP3
LD    A
AD    TEMP3
LD    B
ML    C
ST    TEMP4
```

Recursión

En este capítulo se presenta la recursión, como una de las herramientas de programación más poderosas y menos entendidas por los estudiantes que se inicián en la disciplina. Asimismo, se define la recursión, se presenta su uso en C y se dan algunos ejemplos. También se examina una implantación de recursión mediante pilas. Por último, se discuten las ventajas y desventajas del uso de la recursión en la solución de problemas.

3.1. DEFINICIÓN Y PROCESOS RECURSIVOS

En matemáticas, muchos objetos se definen al presentar un proceso que los produce. Por ejemplo, π se define como el cociente entre el perímetro de una circunferencia y su diámetro. Esto equivale al siguiente conjunto de instrucciones: obtener el perímetro de una circunferencia y su diámetro, dividir el primero entre el segundo y llamar al resultado π . Es claro que el proceso especificado debe terminar con un resultado determinado.

La función factorial

Otro ejemplo de una definición especificada mediante un proceso es la función factorial que desempeña un papel importante en matemáticas y estadística. Dado un entero positivo n , se define el factorial de n como el producto de todos los enteros entre n y 1. Por ejemplo, el factorial de 5 es igual a $5 * 4 * 3 * 2 * 1 = 120$ y el factorial de 3 es igual a $3 * 2 * 1 = 6$. El factorial de 0 se define como 1. En matemáticas, con fre-

cuencia se usa el signo de admiración (!) para la función factorial. En consecuencia, puede escribirse su definición como sigue:

$$\begin{aligned} n! &= 1 \text{ if } n == 0 \\ n! &= n * (n - 1) * (n - 2) * \dots * 1 \text{ if } n > 0 \end{aligned}$$

Los tres puntos en realidad son una abreviatura que indica la multiplicación de todos los números entre $n - 3$ y 2. Para eliminar la abreviatura en la definición de $n!$ se tendría que elaborar una lista de una fórmula para cada $n!$ por separado de la siguiente manera:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 * 1 \\ 3! &= 3 * 2 * 1 \\ 4! &= 4 * 3 * 2 * 1 \\ \dots &\dots \end{aligned}$$

Por supuesto, no puede esperarse la lista de una fórmula para el factorial de cada entero. Para eliminar todas las abreviaturas y evitar un conjunto infinito de definiciones, sin dejar de definir la función de manera precisa, puede presentarse un algoritmo que acepte un entero n y regrese el valor de $n!$

```
prod = 1;
for (x = n; x > 0; x--)
    prod *= x;
return(prod);
```

Tal algoritmo se llama *iterativo* porque requiere la repetición explícita de un proceso hasta satisfacer cierta condición. Este algoritmo puede transformarse en forma sencilla en una función de C que regresa $n!$, cuando n es el parámetro de entrada. Un algoritmo puede concebirse como un programa para una máquina “ideal” que no tenga ninguna de las limitaciones prácticas de una computadora real y pueda, por lo tanto, usarse para definir una función matemática. Sin embargo, una función en C no puede servir para la definición matemática de la función factorial debido a limitaciones como la precisión y el tamaño finito de los datos en una máquina real.

Examinemos con más detenimiento la definición de $n!$ que da por separado una fórmula para cada valor de n . Puede observarse, por ejemplo, que $4!$ es igual a $4 * 3 * 2 * 1$, que es igual a $4 * 3!$. En realidad, es obvio que, para cualquier $n > 0$, $n!$ es igual a $n * (n - 1)!$. Al multiplicar n por el producto de todos los enteros entre $n - 1$ y 1, se obtiene el producto de todos los enteros desde n hasta 1. Por lo tanto, puede definirse:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 * 0! \\ 2! &= 2 * 1! \end{aligned}$$

$$\begin{aligned}3! &= 3 * 2! \\4! &= 4 * 3! \\&\dots\end{aligned}$$

o, si se usa la notación matemática empleada antes;

$$\begin{aligned}n! &= 1 \text{ if } n == 0 \\&= n * (n - 1)! \text{ if } n > 0.\end{aligned}$$

Esta definición puede parecer muy extraña pues define la función factorial en sus propios términos, lo cual da la impresión de ser circular e inaceptable hasta darse cuenta de que la notación matemática sólo es una manera concisa de escribir el número infinito de ecuaciones necesarias para definir $n!$ para cada n . $0!$ se define en forma directa como 1. Una vez definido $0!$, la definición de $1!$ como $1 * 0!$ no es de ninguna manera circular. Lo mismo ocurre cuando se define $2!$ como $2 * 1!$ una vez definido $1!$. Se puede argumentar que la última definición es más precisa que la definición de $n!$ como $n * (n - 1) * \dots * 1$, para $n > 0$, porque no requiere de los tres puntos suspensivos para sustituirse según (es lo que se espera) la intuición lógica del lector. A tal definición, que define un objeto en términos de un caso más simple de sí mismo, se le llama *definición recursiva*.

Veamos ahora cómo puede usarse la definición recursiva de la función factorial para evaluar $5!$. La definición establece que $5!$ es igual a $5 * 4!$. Entonces, antes de evaluar $5!$ debe evaluarse $4!$. Al usar una vez más la definición, se ve que $4! = 4 * 3!$. Por lo tanto, debe evaluarse $3!$. Al repetir el proceso, se tiene que

$$\begin{aligned}1 \cdot 5! &= 5 * 4! \\2 \cdot 4! &= 4 * 3! \\3 \cdot 3! &= 3 * 2! \\4 \cdot 2! &= 2 * 1! \\5 \cdot 1! &= 1 * 0!\end{aligned}$$

Cada caso se reduce a uno más simple hasta que se llega a $0!$, el cual se define de manera directa como 1. En la línea 6 se tiene un valor definido en forma directa no como el factorial de otro número. Por lo tanto, puede regresarse de la línea 6 a la línea 1, llevando el valor calculado en una línea para evaluar el resultado de la línea anterior. Esto produce:

$$\begin{aligned}6 \cdot 0! &= 1 \\5 \cdot 1! &= 1 * 0! = 1 * 1 = 1 \\4 \cdot 2! &= 2 * 1! = 2 * 1 = 2 \\3 \cdot 3! &= 3 * 2! = 3 * 2 = 6 \\2 \cdot 4! &= 4 * 3! = 4 * 6 = 24 \\1 \cdot 5! &= 5 * 4! = 5 * 24 = 120\end{aligned}$$

Intentemos incorporar este proceso a un algoritmo. Una vez más se quiere que el algoritmo tenga como entrada un número entero n no negativo y calcule en una variable $fact$ el entero no negativo del factorial de n .

```
1 fact = 1
2 if (n == 0)
3   fact = 1;
4 else {
5   x = n - 1;
6   fact = encontrar el valor de x!. Llámelo y;
7   fact = n * y;
8 } /* fin de else */
```

Este algoritmo muestra el proceso usado para calcular $n!$ mediante la definición recursiva. La llave para el algoritmo es, por supuesto, la línea 5, donde se pide “encontrar el valor de $x!$ ”, que requiere volver a ejecutar el algoritmo con entrada x , pues el método para calcular la función factorial es el propio algoritmo. Para ver que el algoritmo por fin se detiene, nótese que al principio de la línea 5, x es igual a $n - 1$. Cada vez que se ejecuta el algoritmo, la entrada disminuye en 1 con respecto a la anterior, de tal manera que (como la entrada original n era un entero no negativo) 0 será, a la larga, la entrada. En este punto, el algoritmo regresa simplemente 1. Este valor es devuelto a la línea 5 que pide la evaluación de $0!$. La multiplicación de y (que es igual a 1) por n (también igual a 1) se ejecuta después y se regresa el resultado. Esta secuencia de multiplicaciones y regresos continúa hasta que se evalúa el $n!$ original. En la siguiente sección se verá cómo convertir este algoritmo en un programa en lenguaje C.

Por supuesto, es más simple y directo usar el método iterativo para evaluar la función factorial. Se expuso como ejemplo simple para presentar la recursión, no como el método más efectivo para resolver este problema en particular. De hecho, todos los problemas en esta sección pueden resolverse de forma más eficiente mediante la iteración. Sin embargo, más adelante en éste y otros capítulos, se encontrarán ejemplos que se resuelven con mayor facilidad mediante métodos recursivos.

Multiplicación de números naturales

Otro ejemplo de definición recursiva es la multiplicación de números naturales. El producto $a * b$, donde a y b son enteros positivos, puede definirse como a sumada así misma b veces. Esta es una definición iterativa. La definición recursiva equivalente es la siguiente:

$$\begin{aligned}a * b &= a \text{ if } b == 1 \\a * b &= a * (b - 1) + a \text{ if } b > 1\end{aligned}$$

Para evaluar $6 * 3$ mediante esta definición, primero debe evaluarse $6 * 2$ y sumarle después 6. Para evaluar $6 * 2$, se evalúa primero $6 * 1$ y luego se le suma 6. Pero $6 * 1$ es igual a 6 según la primera parte de la definición. Entonces

$$6 * 3 = 6 * 2 + 6 = 6 * 1 + 6 + 6 = 6 + 6 + 6 = 18$$

Se pide al lector como un ejercicio sencillo, convertir la definición anterior en un algoritmo recursivo.

Nótese la pauta que existe en una definición recursiva. Se presenta un caso simple del término por definir, de manera explícita (en el caso del factorial, $0!$ se definió como 1; en el de la multiplicación $a * 1$ como a). Los otros casos se definen al aplicar alguna operación al resultado de la evaluación de un caso más simple. Así, $n!$ se define en términos de $(n - 1)!$ y $a * b$ en términos de $a * (b - 1)$. Las simplificaciones sucesivas de algún caso en particular tienen que conducir a la larga al caso trivial definido de manera explícita. En el caso de la función factorial la sustracción sucesiva de 1 al valor n conduce, finalmente, a 0. En el caso de la multiplicación, la sustracción sucesiva de 1 al valor b conduce, finalmente, a 1. Si no sucediera así, la definición no sería válida. Por ejemplo, si definimos:

$$n! = (n + 1)! / (n + 1)$$

o

$$a * b = a * (b + 1) - a$$

sería imposible determinar el valor de $5!$ o $6 * 3$. (Se invita al lector que intente determinar esos valores mediante las definiciones anteriores.) Esto sucede a pesar de que las dos definiciones son válidas. Sumar de manera continua 1 a n o b no producirá al final un caso definido de manera explícita. Aún si $100!$ se definió de forma explícita, ¿cómo podría determinarse el valor de $101!$?

La secuencia de Fibonacci

Examinemos un ejemplo menos familiar. La **secuencia de Fibonacci**, que es la secuencia de enteros:

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

Cada elemento en esta secuencia es la suma de los dos precedentes (por ejemplo $0 + 1 = 1$; $1 + 1 = 2$; $1 + 2 = 3$; $2 + 3 = 5$, ...). Sean $fib(0) = 0$, $fib(1) = 1$ y así sucesivamente, entonces puede definirse la secuencia de Fibonacci mediante la definición recursiva:

$$\begin{aligned} fib(n) &= n \text{ if } n == 0 \text{ or } n == 1 \\ fib(n) &= fib(n - 2) + fib(n - 1) \text{ if } n >= 2 \end{aligned}$$

Por ejemplo, para calcular $fib(6)$, puede aplicarse la definición de manera recursiva para obtener:

$$\begin{aligned} fib(6) &= fib(4) + fib(5) = fib(2) + fib(3) + fib(5) = \\ fib(0) + fib(1) + fib(3) + fib(5) &= 0 + 1 + fib(3) + fib(5) = \\ 1 + fib(1) + fib(2) + fib(5) &= \\ 1 + 1 + fib(0) + fib(1) + fib(5) &= \\ 2 + 0 + 1 + fib(5) &= 3 + fib(3) + fib(4) = \\ 3 + fib(1) + fib(2) + fib(4) &= \end{aligned}$$

$$\begin{aligned} 3 + 1 + fib(0) + fib(1) + fib(4) &= \\ 4 + 0 + 1 + fib(2) + fib(3) &= 5 + fib(0) + fib(1) + fib(3) = \\ 5 + 0 + 1 + fib(1) + fib(2) &= 6 + 1 + fib(0) + fib(1) = \\ 7 + 0 + 1 &= 8 \end{aligned}$$

Obsérvese que la definición recursiva de los números de Fibonacci difiere de las definiciones recursivas de la función factorial y de la multiplicación. La definición recursiva de fib se refiere dos veces a sí misma. Por ejemplo, $fib(6) = fib(4) + fib(5)$, de tal manera que al calcular $fib(6)$, fib tiene que aplicarse de manera recursiva dos veces. Sin embargo, calcular $fib(5)$ también implica calcular $fib(4)$, así que al aplicar la definición hay mucha redundancia de cálculo. En el ejemplo anterior, $fib(3)$ se calcula tres veces por separado. Sería mucho más eficiente "recordar" el valor de $fib(3)$ la primera vez que se calcula y volver a usarlo cada vez que se necesita. Es mucho más eficiente un método iterativo como el que sigue para calcular $fib(n)$:

```
if (n <= 1)
    return(n);
lofib = 0;
hifib = 1;
for (i = 2; i <= n; i++) {
    x = lofib;
    lofib = hifib;
    hifib = x + lofib;
} /* fin de for */
return(hifib);
```

Compárese el número de adiciones (sin incluir los incrementos de la variable índice, i) que se ejecutan para calcular $fib(6)$ mediante este algoritmo al usar la definición recursiva. En el caso de la función factorial, tienen que ejecutarse el mismo número de multiplicaciones para calcular $n!$ mediante ambos métodos: recursivo e iterativo. Lo mismo ocurre con el número de sumas en los dos métodos al calcular la multiplicación. Sin embargo, en el caso de los números de Fibonacci, el método recursivo es mucho más costoso que el iterativo. En una sección posterior se hablará más acerca de los méritos relativos de ambos métodos.

La búsqueda binaria

Pudo haberse obtenido la falsa impresión de que la recursión es una herramienta que está muy a la mano para definir funciones matemáticas pero que no tiene ninguna influencia en actividades de cálculo más prácticas. El siguiente ejemplo ilustra una aplicación de la recursión en una de las actividades más comunes en computación: la búsqueda.

Considérese un arreglo de elementos en el cual se han colocado los objetos en algún orden. Por ejemplo, puede concebirse un diccionario o una libreta de teléfonos como un arreglo cuyas entradas están en orden alfabético. El archivo con la lista de pago puede estar ordenado de acuerdo con los números del seguro social de sus

empleados. Supóngase que tal arreglo existe y que se desea encontrar uno de sus elementos en particular. Por ejemplo, se quiere buscar un nombre en una libreta de teléfonos, una palabra en un diccionario o un empleado en un archivo de personal. El proceso usado para encontrar una entrada se llama **búsqueda**.

Como la búsqueda es una actividad tan común en computación, es conveniente encontrar un método eficiente para ejecutarla. Quizás el método de búsqueda menos refinado es el de búsqueda **lineal** o **secuencial**, en el cual se examina cada uno de los elementos del arreglo y se le compara, cada vez, con el que se busca hasta que ocurre una coincidencia. Si la lista está desordenada y construida al azar, la búsqueda lineal puede ser la única vía de encontrar algo en ella (a no ser, por supuesto, que se ordene primero). Sin embargo, no debería usarse éste para buscar un nombre en un directorio telefónico. En su lugar, se abre el libro en una página cualquiera y se examinan los nombres que aparecen en ella. Como están ordenados en forma alfabética, dicho examen determinará si debe continuarse la búsqueda en la primera o segunda mitad del libro.

Apliquemos esta idea a la búsqueda en un arreglo. Si el arreglo sólo contiene un elemento, el problema es trivial. En otro caso, compárese el elemento que se busca con el elemento central del arreglo. Si son iguales, se ha concluido con éxito la búsqueda. Si el elemento central es mayor, se repite el proceso de búsqueda en la primera mitad del arreglo (dado que si el elemento buscado aparece en algún lugar debe hacerlo en la primera mitad); en caso contrario, se repite el proceso con la segunda mitad. Obsérvese que cada vez que se realiza una comparación, se divide en dos el número de elementos que aún deben buscarse. Para arreglos grandes este método es superior al de búsqueda secuencial en el cual cada comparación reduce sólo en 1 el número de elementos que aún deben examinarse. Este método se llama **búsqueda binaria**, debido a que el arreglo donde se realiza la búsqueda se divide en dos partes iguales.

Obsérvese que se definió la búsqueda binaria, de manera bastante natural, en forma recursiva. Si el elemento que se busca no es igual al central del arreglo, las instrucciones indican buscar en un subarreglo mediante el mismo método. Así, el método de búsqueda se define en términos de sí mismo con un arreglo más pequeño como entrada. Se puede estar seguro de que terminará el proceso porque los arreglos de entrada se hacen cada vez más pequeños y se define de manera no recursiva la búsqueda en un arreglo de un elemento, dado que el elemento intermedio de tal arreglo es su único elemento.

Presentamos ahora un algoritmo recursivo para buscar un elemento x entre $a[low]$ y $a[high]$ en un arreglo ordenado a . El algoritmo regresa un *índice* de a , tal que $a[index] = x$ si existe $index$ entre low y $high$. Si no se encuentra x en esa parte del arreglo $bisrch$ regresa -1 (en C no puede existir un elemento $a[-1]$).

```

1 if (low > high)
2   return(-1);
3 mid = (low + high) / 2;
4 if (x == a[mid])
5   return(mid);
6 if (x < a[mid])

```

```

7   buscar x entre a[low] y a[mid] -1
8 else
9   buscar x entre a[mid] + 1 y a[high]

```

Como se incluye la posibilidad de una búsqueda inútil (es decir, puede no existir el elemento en el arreglo), se ha cambiado un poco el caso trivial. La búsqueda de un elemento en un arreglo no se define en forma directa como el índice apropiado. Más bien se compara el elemento con el que se está buscando. Si no son iguales, la búsqueda continúa en la "primera" o "segunda" mitad (ninguna de las cuales contiene elementos). Este caso se indica por la condición $low > high$ y su resultado se define de manera directa como -1 .

Aplicaremos el algoritmo a un ejemplo. Supóngase que el arreglo a contiene los elementos: 1, 3, 4, 5, 17, 18, 31, 33, en ese orden y se quiere buscar 17 (es decir, x es igual a 17) entre el elemento 0 y el 7 (es decir, low es 0 y $high$ es 7). Al aplicar el algoritmo se obtiene:

línea 1: ¿Es $low > high$? No lo es; entonces ejecútense la línea 3.
línea 3: $mid = (0 + 7)/2 = 3$.
línea 4: ¿Es $x == a[3]$? 17 no es igual a 5, entonces ejecútense la línea 6.
línea 6: ¿Es $x < a[3]$? 17 no es menor que 5, entonces ejecútense la cláusula *else* de la línea 8.
línea 9: Repítase el algoritmo con $low = mid + 1 = 4$ y $high = high = 7$; i.e. búsquese en la mitad superior del arreglo.

línea 1: ¿Es $4 > 7$? No, entonces ejecútense la línea 3.
línea 3: $mid = (4 + 7)/2 = 5$.
línea 4: ¿Es $x == a[5]$? 17 no es igual a 18, entonces ejecútense la línea 6.
línea 6: ¿Es $x > a[5]$? Sí, puesto que $17 < 18$, entonces búsquese x entre $a[low]$ y $a[mid-1]$.

línea 7: Repítase el algoritmo con $low = low = 4$ y $high = mid - 1 = 4$. Se aísle x entre el elemento cuarto y elemento cuarto de a .
línea 1: ¿Es $4 > 4$? No, entonces ejecútense la línea 3.
línea 3: $mid = (4 + 4)/2 = 4$.
línea 4: Como $a[4] == 17$, regresa $mid = 4$ como respuesta. 17 es en realidad el cuarto elemento del arreglo.

Obsérvese la pauta de llamadas y regresos del algoritmo. Un diagrama que representa dicha pauta aparece en la figura 3.1.1. Las flechas continuas indican el flujo de control a lo largo del algoritmo y las llamadas recursivas. Los regresos se indican con línea punteada. Como no hay pasos por ejecutarse en el algoritmo después de la línea 7 u 8, el resultado se regresa intacto a la ejecución previa. Por último, cuando el control regresa a la ejecución original, la respuesta regresa al punto de llamada.

Examinemos cómo busca el algoritmo un elemento que no está en el arreglo. Supóngase que el arreglo a es el mismo que en el ejemplo previo donde se busca x igual a 2.

línea : ¿Es $low > high$? O no es mayor que 7, entonces ejecútese la línea 3.
 línea 3: $mid = (0 + 7)/2 = 3$.
 línea 4: ¿Es $x == a[3]$? 2 no es igual a 5, entonces ejecútese la línea 6.
 línea 6: ¿Es $x < a[3]$? Si, $2 < 5$, entonces búsquese x entre $a[low]$ y $a[mid - 1]$.
 línea 7: Repítase el algoritmo con $low = low = 0$ y $high = mid - 1 = 2$. Si aparece 2 en el arreglo tiene que estar entre $a[0]$ y $a[2]$, inclusive.
 línea 1: ¿Es $0 > 2$? No, ejecútese la línea 3.
 línea 3: $mid = (0 + 2)/2 = 1$.
 línea 4: ¿Es $2 == a[1]$? No, ejecútese la línea 6.
 línea 6: ¿Es $2 < a[1]$? Si, dado que $2 < 3$. Búsquese x entre $a[low]$ y $a[mid - 1]$.
 línea 7: Repítase el algoritmo con $low = low = 0$ y $high = mid - 1 = 0$. Si x existe en a tiene que ser el primer elemento.
 línea 1: ¿Es $0 > 0$? No, ejecútese la línea 3.
 línea 3: $mid = (0 + 0)/2 = 0$.
 línea 4: ¿Es $2 == a[0]$? No, ejecútese la línea 6.
 línea 6: ¿Es $2 < a[0]$? 2 no es menor que 1, entonces ejecútese la cláusula *else* en la línea 8.
 línea 9: Repítase el algoritmo con $low = mid + 1 = 1$ y $high = high = 0$.
 línea 1: ¿Es $low > high$? 2 es mayor que 1, entonces el resultado es -1 . El elemento 2 no existe en el arreglo.

Propiedades de las definiciones o algoritmos recursivos

Resumamos lo que implica una definición o algoritmo recursivo. Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una se-

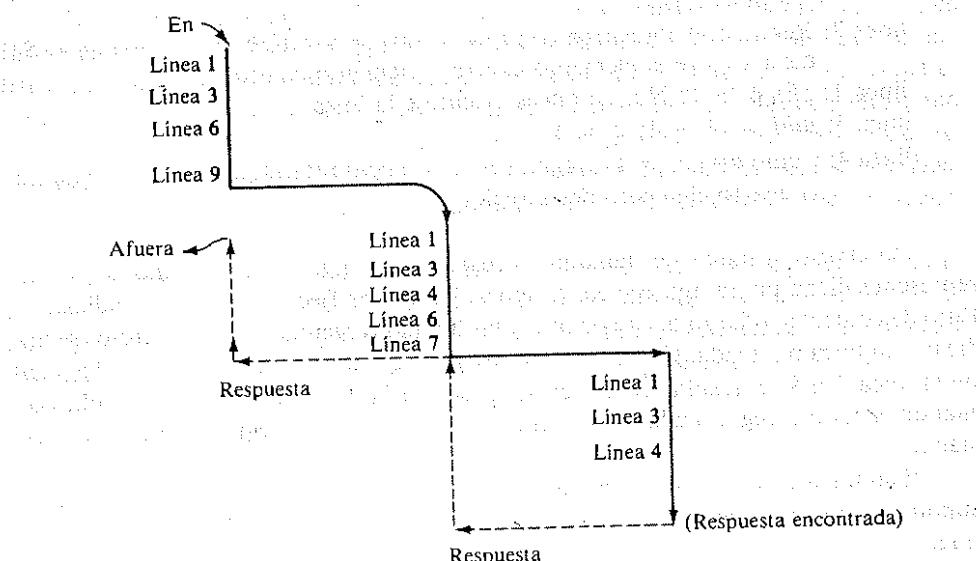


Figura 3.1.1 Representación en diagrama del algoritmo de búsqueda binaria.

cuencia infinita de llamadas a sí mismo. Claro que cualquier algoritmo que genere tal secuencia no termina nunca. Una función recursiva f debe definirse en términos que no impliquen a f al menos en un argumento o grupo de argumentos. Debe existir una "salida" de la secuencia de llamadas recursivas. En los ejemplos de esta sección, los segmentos no recursivos de las definiciones fueron:

```

factorial :      0! = 1
multiplicación : a * 1 = a
secuencia de Fibonacci: fib(0) = 0; fib(1) = 1
búsqueda binaria: if (low > high)
                      return(-1);
                  if (x == a[mid])
                      return(mid);
  
```

Sin esta salida no recursiva, no puede calcularse ninguna función recursiva. Cualquier caso de definición recursiva o invocación de un algoritmo recursivo tiene que reducirse a la larga a alguna manipulación de uno o dos casos más simples no recursivos.

EJERCICIOS

- 3.1.1. Escriba un algoritmo iterativo para evaluar $a * b$ mediante la suma, donde a y b son enteros no negativos.
 - 3.1.2. Escriba una definición recursiva de $a + b$, donde a y b son enteros no negativos, en términos de la función sucesor, *succ*, definida como:
- ```

succ(x)
int x;
{
 return(x++);
} /* fin de succ */

```
- 3.1.3. Sea  $a$  un arreglo de enteros. Presente algoritmos recursivos para calcular
    - a. el elemento máximo del arreglo
    - b. el elemento mínimo del arreglo
    - c. la suma de los elementos del arreglo
    - d. el producto de los elementos del arreglo
    - e. el promedio de los elementos del arreglo
  - 3.1.4. Evalúe mediante ambas definiciones, recursiva e iterativa, cada una de las siguientes expresiones.
    - a.  $6!$
    - b.  $9!$
    - c.  $100 * 3$
    - d.  $6 * 4$
    - e.  $fib(10)$
    - f.  $fib(11)$

- 3.1.5. Suponga que un arreglo de diez enteros contiene los elementos  
 $1, 3, 7, 15, 21, 22, 36, 78, 95, 106$ . Use la búsqueda recursiva binaria para encontrar cada uno de los siguientes elementos en el arreglo

- a. 1
- b. 20
- c. 36

- 3.1.6. Escriba una versión iterativa del algoritmo de búsqueda binaria. (Sugerencia: modifique los valores de *low* y *high* en forma directa.)

- 3.1.7. La función de Ackerman se define de manera recursiva para enteros no negativos, como sigue:

$$\begin{aligned} a(m, n) &= n + 1 && \text{si } m == 0 \\ a(m, n) &= a(m - 1, 1) && \text{si } m != 0, n == 0 \\ a(m, n) &= a(m - 1, a(m, n - 1)) && \text{si } m != 0, n != 0 \end{aligned}$$

- a. Mediante la definición anterior, muestre que  $a(2, 2)$  es igual a 7.
- b. Pruebe que  $a(n, n)$  está definida para todos los enteros no negativos  $m$  y  $n$ .
- c. ¿Se puede encontrar un método iterativo para calcular  $a(m, n)$ ?

- 3.1.8. Cuente el número de adiciones necesarias para calcular  $fib(n)$  para  $0 \leq n \leq 10$  mediante ambos métodos: iterativo y recursivo. ¿Aparece alguna pauta?

- 3.1.9. Si un arreglo contiene  $n$  elementos, ¿cuál es el número máximo de llamadas recursivas hechas por el algoritmo de búsqueda binaria?

## 3.2. RECURSION EN C

### Factorial en C

El lenguaje C permite al programador definir subrutinas y funciones que se llaman a sí mismas, a las cuales se les llama *recursivas*.

El algoritmo recursivo para calcular  $n!$  puede transformarse en forma directa en una función C de la siguiente manera:

```
fact(n)
int n;
{
 int x, y;
 if (n == 0)
 return(1);
 x = n-1;
 y = fact(x);
 return(n * y);
} /* fin de fact */
```

En la instrucción  $y = fact(x)$ ; la función *fact* se llama a sí misma. Este es el ingrediente esencial de una rutina recursiva. El programador supone que la función que se calcula ya fue definida y la usa dentro de su propia definición. Sin embargo, tiene que asegurarse de que no conduzca a una serie interminable de llamadas.

Examinemos la ejecución de esta función cuando la llama otro programa. Por ejemplo, supóngase que el programa de llamada contiene la siguiente instrucción

```
printf("%d", fact(4));
```

Cuando la rutina de llamada invoca a *fact*, el parámetro  $n$  se iguala a 4. Como  $n$  no es 0,  $x$  se iguala a 3. En este punto, se llama a *fact* por segunda vez con un argumento de 3. Por lo tanto la función *fact* se actúa de nuevo y se vuelven a asignar las variables locales ( $x$  y  $y$ ) y el parámetro del bloque ( $n$ ). Como la ejecución todavía no deja la primera llamada a *fact*, se conserva la primera asignación de esas variables. Así, existen dos generaciones simultáneas de esas variables. En cualquier punto de la segunda ejecución de *fact*, sólo se puede hacer referencia a las copias más recientes de estas variables.

En general, cada vez que la función *fact* se ingresa de manera recursiva, se asigna un nuevo conjunto de variables y parámetros y sólo se puede hacer referencia a este conjunto dentro de dicha llamada a *fact*. Cuando se regresa de *fact* a un punto en una llamada previa, se libera la asignación más reciente de dichas variables y se reactiva la copia previa, la cual es aquella asignada durante la entrada original a la llamada previa y es propia de esa llamada.

Esta descripción sugiere el uso de una pila para almacenar las generaciones sucesivas de parámetros y variables locales, la cual conserva el sistema C de manera invisible para el usuario. Cada vez que se ingresa una función recursiva, se inserta en el tope de la pila una nueva asignación de sus variables. Cualquier referencia a una variable local o parámetro se da a través del tope actual de la pila. Cuando se regresa la función, se libera la localidad del tope y la localidad previa se convierte en el tope actual que debe usarse para hacer referencia a las variables locales. Este mecanismo se examina con más detalle en la sección 4, pero por ahora, veamos su aplicación en el cálculo de la función factorial.

La figura 3.2.1 contiene una serie de imágenes de la pila para las variables  $n$ ,  $x$  y  $y$  durante la ejecución de la función *fact*. Al inicio, las pilas están vacías, como se ilustra en la figura 3.2.1a. Después de la primera llamada a *fact* por el procedimiento de llamada, la situación es la que se muestra en la figura 3.2.1b, con  $n$  igual a 4. Las variables  $x$  y  $y$  están asignadas pero no inicializadas. Como  $n$  no es igual a 0,  $x$  se iguala a 3 y se llama a *fact(3)* (figura 3.2.1c). El nuevo valor de  $n$  no es 0, por lo tanto se iguala a 2 y se llama a *fact(2)* (figura 3.2.1d).

Esto continúa hasta que  $n$  es igual a 0 (figura 3.2.1f). En ese momento, se regresa el valor 1 de la llamada a *fact(0)*. La ejecución se reanuda a partir del punto en el cual fue llamada *fact(0)*, que es la asignación del valor que se regresó a la copia de  $y$  declarada en *fact(1)*. Esto se ilustra mediante el estado de la pila que se muestra en la figura 3.2.1g donde se ha liberado las variables asignadas para *fact(0)* y se iguala  $y$  a 1.

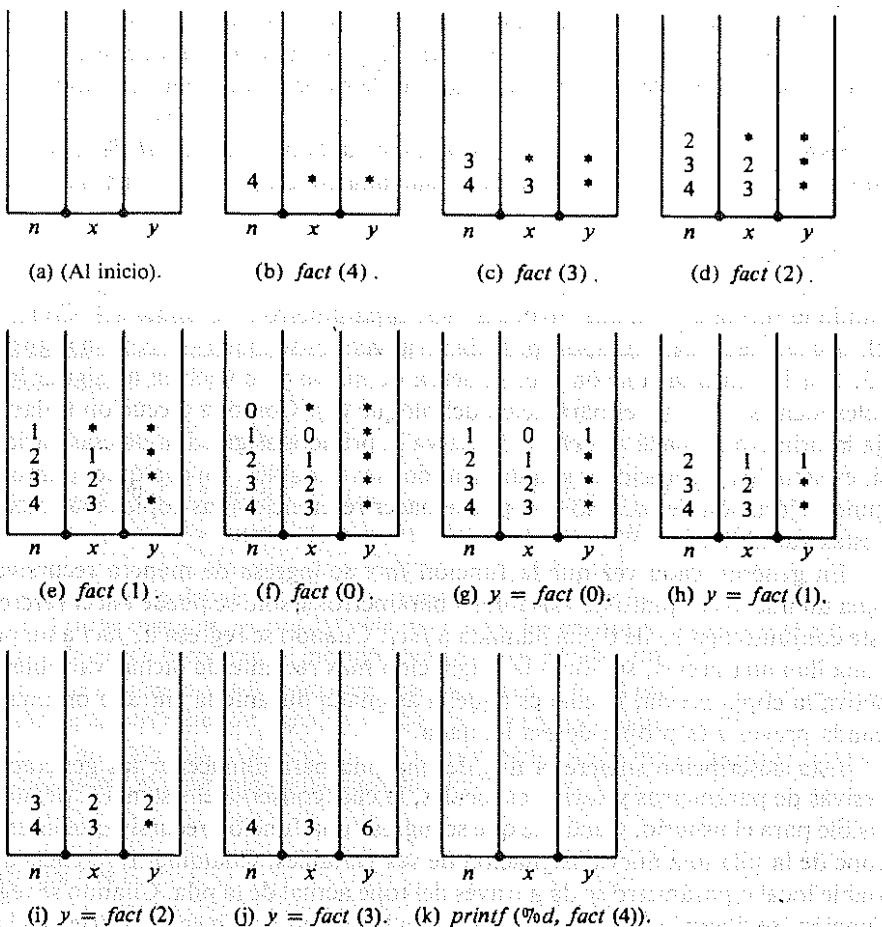


Figura 3.2.1 La pila en diferentes momentos de la ejecución. (El asterisco indica un valor no inicializado.)

Después se ejecuta la instrucción `return(n * y)`, al multiplicar los valores del topo de `n` y `y` para obtener 1 y regresar este valor a `fact(2)` (figura 3.2.1h). Este proceso se repite dos veces más, hasta que, por último, el valor de `y` en `fact(4)` es igual a 6 (figura 3.2.1j). La instrucción `return(n * y)` se ejecuta una vez más. El producto 24 se regresa al procedimiento de llamada donde se imprime mediante la instrucción:

```
printf("%d", fact(4));
```

Obsérvese que cada vez que se regresa una rutina recursiva, lo hace al punto que sigue de manera inmediata a aquél desde el cual se le llamó. Así, la llamada recursiva a `fact(3)` regresa a la asignación del resultado a `y` dentro de `fact(4)`, pero la llamada recursiva a `fact(4)` regresa a la instrucción `printf` en el programa de llamada.

Transformemos algunos de los otros procesos y definiciones recursivos de la sección anterior en programas recursivos en C. Es difícil concebir que un programador de C escriba un programa para calcular el producto de dos enteros positivos en términos de la adición, dado que un asterisco lo hace de manera directa. No obstante, una función que lo haga puede servir como ejemplo para ilustrar la recursión en C. Al seguir la definición de multiplicación de la sección previa, puede escribirse:

```
mult(a, b)
int a, b;
{
 if (b == 1)
 return(a);
 else
 return(b * mult(a, b-1));
} /* fin de mult */
```

Obsérvese cuán similares son este programa y la definición recursiva de la última sección. Dejamos como ejercicio, al lector continuar con la ejecución de esta función cuando se llama con dos enteros positivos; el uso de pilas es de gran ayuda en este proceso de rastreo.

Este ejemplo ilustra cómo puede llamarse a sí misma una función recursiva, aun dentro de una instrucción donde se le asignen valores. De manera similar, podría haberse escrito la función recursiva `fact` en forma más breve como sigue:

```
fact(n)
int n;
{
 return(n == 0 ? 1 : n * fact(n-1));
} /* fin de fact */
```

Esta versión reducida evita el uso explícito de variables locales `x` (para guardar el valor de `n - 1`) y `y` (para guardar el valor de `fact(x)`). Sin embargo, de todos modos se apartan localidades temporales para esos dos valores en cada llamada a la función. Esas localidades se tratan de manera similar a como se trata cualquier variable local explícita. Así, al trazar la acción de una rutina recursiva, puede ser de utilidad declarar todas las variables temporales en forma explícita. Véase si es de alguna manera más fácil seguir la acción de la siguiente versión más explícita de `mult`:

```
mult(a, b)
int a, b;
{
 int c, d, sum;
 if (b == 1)
 return(a);
 c = b-1;
 d = mult(a, c);
 sum = d+a;
 return(sum);
} /* fin de mult */
```

Otro punto que debe analizarse es que tiene particular importancia verificar la validez de los parámetros de entrada en una rutina recursiva. Por ejemplo, examinemos la ejecución de la función *fact* cuando se llama por medio de una instrucción como la siguiente:

```
printf("\n%d", fact(-1));
```

Por supuesto, no se diseñó la función *fact* para producir un resultado significativo para una entrada negativa. Sin embargo, una de los asuntos más importantes para un programador es el de aprender que una función se encontrará invariablemente con una entrada nula y el error resultante puede ser muy difícil de encontrar a menos que se hayan tomado en cuenta las posibles entradas erróneas.

Por ejemplo, cuando se transfiere  $-1$  como parámetro a *fact*, de tal manera que  $n = -1$ ,  $x$  se iguala a  $-2$ , y el cual se transfiere para una llamada recursiva a *fact*. Se asigna otro grupo de valores a  $n$ ,  $x$  y  $y$ ,  $n$  se hace igual a  $-2$  y  $x$  se convierte en  $-3$ . Este proceso continúa hasta que al programa se queda sin tiempo o espacio, o bien el valor de  $x$  se hace muy pequeño. No se envía ningún mensaje indicando la verdadera causa del error.

Si se llamó a *fact* al principio con una expresión complicada como argumento y la expresión se evaluó por error como número negativo, un programador podría pasar horas buscando la causa del error. El problema se puede remediar revisando la función *fact* para verificar sus entradas de manera explícita, como sigue:

```
fact(n)
int n;
{
 int x, y;
 if (n < 0) {
 printf("%s", "parámetro negativo en la función factorial");
 exit(1);
 }
 if (n == 0)
 return(1);
 x = n-1;
 y = fact(x);
 return(n * y);
}
```

De manera similar, debe cuidar que un valor no positivo se encuentre el segundo parámetro de la función *mult*.

### Los números de Fibonacci en C

Prestemos de nuevo atención a la secuencia de Fibonacci. Un programa en C para calcular el  $n$ -ésimo número de Fibonacci puede extraerse casi sin cambio de la definición recursiva:

```
fib(n)
int n;
{
 int x, y;
 if (n <= 1)
 return(n);
 x = fib(n-1);
 y = fib(n-2);
 return(x + y);
} /* fin de fib */
```

Sigamos la acción de esta función cuando se calcula el sexto número de Fibonacci. Puede compararse dicha acción con el cálculo manual que se ejecutó en la última sección para calcular *fib(6)*. El proceso de apilamiento se ilustra en la figura 3.2.2. Cuando se llama el programa la primera vez, se asigna memoria para las variables  $n$ ,  $x$  y  $y$ , y  $n$  se iguala en forma recursiva a 6 (figura 3.2.2a). Como  $n > 1$ , se evalúan  $n - 1$  y se llama en forma recursiva a *fib*. Se asigna un nuevo conjunto a  $n$ ,  $x$  y  $y$ , y se iguala  $n$  a 5 (figura 3.2.2b). El proceso continúa (figuras 3.2.2c-f) con cada valor sucesivo de  $n$  disminuido en uno respecto a su antecesor, hasta que se llama a *fib* con  $n$  igual a 1. La sexta llamada a *fib* regresa 1 a quien la llamó, de tal manera que en la quinta asignación de memoria,  $x$  toma el valor de 1 (figura 3.2.2g).

La siguiente instrucción secuencial,  $y = fib(n - 2)$  se ejecuta después. El valor de  $n$  que se usa es el que se asignó más recientemente, 2. Así, llamamos de nuevo a *fib* con argumento 0 (figura 3.2.2h). De inmediato se regresa el valor de 0, de tal manera que  $y$  se iguala a 0 en *fib(2)* (figura 3.3.2i). Obsérvese que cada llamada recursiva tiene como resultado un regreso al punto de llamada de manera que la llamada a *fib(1)* regresa la asignación  $x$  y la llamada a *fib(0)* a la asignación  $y$ . La siguiente instrucción que debe ejecutarse en *fib(2)* es la que regresa  $x + y = 1 + 0 = 1$  a la instrucción que llama a *fib(2)* en la generación de la función que calcula *fib(3)*. Esta instrucción es la de asignación de  $x$ , por lo que se le da a  $x$  en *fib(3)* el valor *fib(2) = 1* (figura 3.2.2j). El proceso de llamada, inserción, regreso y eliminación continúa hasta que la rutina regresa por última vez al programa principal con el valor 8. La figura 3.2.2 muestra la pila hasta el momento en que *fib(5)* llama a *fib(3)*, de tal manera que su valor pueda asignarse a  $y$ . Se pide al lector completar la ilustración con los estados de la pila para el resto de la ejecución del programa.

Este programa ilustra cómo puede llamarse a sí misma varias veces con argumentos diferentes una rutina recursiva. De hecho, siempre y cuando sólo use variables locales una rutina recursiva, el programador puede usarla tal y como usa cualquier otra y suponer que ejecuta su función y produce el valor deseado. El programador no necesita preocuparse por el mecanismo subyacente de la pila.

### La búsqueda binaria en C

Presentemos ahora un programa en C para realizar la búsqueda binaria. La función para hacerlo acepta como entradas un arreglo  $a$  y un elemento  $x$  y regresa el

| <i>n</i> | <i>x</i> | <i>y</i> | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6        | *        | *        | 5        | *        | *        | 4        | *        | *        | 3        | *        | *        |
| 6        | *        | *        | 6        | *        | *        | 5        | *        | *        | 4        | *        | *        |
|          |          |          | 6        | *        | *        | 5        | *        | *        | 5        | *        | *        |
|          |          |          | 6        | *        | *        | 6        | *        | *        | 6        | *        | *        |
| (a)      | (b)      | (c)      | (d)      | (e)      |          |          |          |          |          |          |          |
| 1        | *        | *        | 2        | 1        | *        | 2        | 1        | *        | 3        | 1        | *        |
| 2        | *        | *        | 3        | *        | *        | 3        | *        | *        | 4        | *        | *        |
| 3        | *        | *        | 4        | *        | *        | 4        | *        | *        | 5        | *        | *        |
| 4        | *        | *        | 5        | *        | *        | 5        | *        | *        | 6        | *        | *        |
| 5        | *        | *        | 6        | *        | *        | 6        | *        | *        |          |          |          |
| (f)      | (g)      | (h)      | (i)      | (j)      |          |          |          |          |          |          |          |
| 1        | *        | *        | 3        | 1        | 1        | 4        | 2        | *        | 1        | *        | *        |
| 3        | 1        | *        | 4        | *        | *        | 2        | *        | *        | 2        | *        | *        |
| 4        | *        | *        | 5        | *        | *        | 4        | 2        | *        | 4        | 2        | *        |
| 5        | *        | *        | 6        | *        | *        | 5        | *        | *        | 5        | *        | *        |
| 6        | *        | *        |          |          |          | 6        | *        | *        | 6        | *        | *        |
| (k)      | (l)      | (m)      | (n)      | (o)      |          |          |          |          |          |          |          |
| 0        | *        | *        | 2        | 1        | 0        | 4        | 2        | 1        | 3        | *        | *        |
| 2        | 1        | *        | 4        | 2        | *        | 5        | 3        | *        | 5        | 3        | *        |
| 4        | 2        | *        | 5        | *        | *        | 6        | *        | *        | 6        | *        | *        |
| 5        | *        | *        | 6        | *        | *        |          |          |          |          |          |          |
| (p)      | (q)      | (r)      | (s)      | (t)      |          |          |          |          |          |          |          |

Figura 3.2.2 La pila de recursión de la función de Fibonacci. Se observa que cada llamada a la función *binsrch* se hace en un subárea del mismo arreglo *a*, con límites *low* y *high* tales que *low <= i <= high*. La figura muestra la ejecución de la función *binsrch* para el valor *x = 17*. La función *binsrch* se llama por primera vez con *low = 0* y *high = n - 1*. La función *binsrch* se llama por primera vez con *low = 0* y *high = n - 1*.

*i = binsrch(a, x, 0, n-1);*

Sin embargo, al fijarse en el algoritmo de búsqueda binaria de la sección 3.1, tomándolo como modelo para una rutina recursiva en C y se observa que se transfieren dos parámetros más en las llamadas recursivas. Las líneas 7 y 9 de dicho algoritmo sólo

llaman a la búsqueda binaria en una parte del arreglo. Así, para que la función sea recursiva, los límites en los cuales debe realizarse la búsqueda también deben especificarse. La rutina se escribe de la siguiente manera:

```
binsrch(a, x, low, high)
int a[];
int x;
int high, low;
{
 int mid;
 if (low > high)
 return(-1);
 mid = (low + high) / 2;
 return(x == a[mid] ? mid : x < a[mid] ?
 binsrch(a, x, low, mid-1) :
 binsrch(a, x, mid+1, high));
} /* fin de binsrch */
```

Cuando se llama a *binsrch* por primera vez desde otra rutina para buscar *x* en un arreglo declarado por

*int a[ARRAYSIZE];*

en el cual los primeros *n* elementos están ocupados, se llama mediante la siguiente instrucción

*i = binsrch(a, x, 0, n-1);*

Se invita al lector a seguir tanto la ejecución de esta rutina como el proceso de la pila por medio del ejemplo de la sección anterior, donde *a* es un arreglo de 8 elementos (*n* = 8) que contiene a 1, 3, 4, 5, 17, 18, 31, 33 en ese orden. El valor buscado es 17 (*x* igual a 17). Nótese que el arreglo *a* se pone en la pila para cada llamada recursiva. Los valores de *low* y *high* son los de los límites inferior y superior del arreglo *a*, respectivamente.

En el curso del recorrido de la rutina *binsrch*, puede haberse notado que los valores de dos parámetros *a* y *x* no cambian durante la ejecución. Cada vez que se llama a *binsrch* se busca el mismo elemento en el mismo arreglo; sólo cambian los límites superior e inferior de la búsqueda. En consecuencia, parece un derroche, el poner y eliminar esos dos parámetros de la pila cada vez que se llama a la rutina recursivamente.

Una solución es permitir que *a* y *x* sean variables globales, declaradas antes del programa por:

```
int a[ARRAYSIZE];
int x;
```

Se llama a la rutina por una instrucción como la siguiente:

```
i = binsrch(0, n-1)
```

En este caso, todas las referencias a *a* y *x* son a sus asignaciones globales de *a* y *x*, declaradas al principio del archivo fuente. Esto permite a *binsrch* dar acceso a *a* y *x* sin asignarles espacio adicional. Se eliminan todas las asignaciones múltiples y liberaciones de espacio para esos parámetros.

Puede volver a escribirse la función *binsrch* como sigue:

```
binsrch(low, high)
int high, low;
{
 int mid;
 if (low > high)
 return(-1);
 mid = (low + high) / 2;
 return (x == a[mid] ? mid :
 x < a[mid] ? binsrch(low, mid-1) : binsrch(mid+1, high));
} /* fin de binsrch */
```

Mediante este esquema, se hace referencia a las variables *a* y *x* por medio del atributo *extern* y no se transfieren con cada llamada recursiva a *binsrch*. *a* y *x* no cambian sus valores y no se ponen en la pila. El programador que desee hacer uso de *binsrch* en un programa sólo necesita transferir los parámetros *low* y *high*. La rutina podría llamarse mediante una instrucción como la siguiente:

```
i = binsrch(low, high);
```

#### Cadenas recursivas

Una función recursiva no necesita llamarse a sí misma de manera directa. En su lugar, puede hacerlo de manera indirecta como en el siguiente ejemplo:

```
a(formal parameters), b(formal parameters)
{
 a(arguments);
 b(arguments);
} /* fin de a */ /* fin de b */ /* fin de a */
```

En este ejemplo la función *a* llama a *b*, la cual puede a su vez llamar a *a*, que puede llamar de nuevo a *b*. Así, ambas funciones, *a* y *b*, son recursivas, dado que se llaman a sí mismas de manera indirecta. Sin embargo, el que lo sean no es obvio a

partir del examen del cuerpo de una de las rutinas en forma individual. La rutina *a*, parece llamar a otra rutina *b* y es imposible determinar que se puede llamar a sí misma de manera indirecta al examinar sólo a *a*.

Pueden incluirse más de dos rutinas en una *cadena recursiva*. Así, una rutina *a* puede llamar a *b*, que llama a *c*, ..., que llama a *z*, que llama a *a*. Cada rutina de la cadena puede potencialmente llamarse a sí misma y, por lo tanto, es recursiva. Por supuesto, el programador debe asegurarse de que un programa de ese tipo no genere una secuencia infinita de llamadas recursivas.

#### Definición recursiva de expresiones algebraicas

Como ejemplo de cadena recursiva considérese el siguiente grupo recursivo de definiciones:

1. Una *expresión* es un *término* seguido por un *signo* más seguido por un *término*, o un *término* solo.
2. Un *término* es un *factor* seguido por un *asterisco* seguido por un *factor*, o un *factor* solo.
3. Un *factor* es una *letra* o una *expresión* encerrada entre *paréntesis*.

Antes de ver algunos ejemplos, obsérvese que ninguno de los tres elementos anteriores está definido en forma directa en sus propios términos. Sin embargo, cada uno de ellos se define de manera indirecta. Una expresión se define por medio de un término, un término por medio de un factor y un factor por medio de una expresión. De manera similar, se define un factor por medio de una expresión, que se define por medio de un término que a su vez se define por medio de un factor. Así, el conjunto completo de definiciones forma una cadena recursiva.

Demos ahora algunos ejemplos. La forma más simple de un factor es una letra. Así *A*, *B*, *C*, *Q*, *Z* y *M* son factores. También son términos, dado que un término puede ser un factor solo. También son expresiones, dado que una expresión puede ser un término solo. Como *A* es una expresión, (*A*) es un factor y, por lo tanto, un término y una expresión. *A* + *B* es un ejemplo de una expresión que no es ni un término ni un factor, sin embargo (*A* + *B*) es las tres cosas. *A* \* *B* es un término y, en consecuencia, una expresión, pero no es un factor. *A* \* *B* + *C* es una expresión que no es ni un término ni un factor. *A* \* (*B* + *C*) es un término y una expresión, pero no es un factor.

Cada uno de los ejemplos anteriores es una expresión válida. Esto puede mostrarse al aplicar la definición de una expresión a cada uno. Considerese, sin embargo, la cadena *A* + \* *B*. No es ni una expresión, ni un término, ni un factor. Sería instructivo para el lector intentar aplicar la definición de expresión, término y factor para ver que ninguna de ellas describe a la cadena *A* + \* *B*. De manera similar, (*A* + *B*) \* *C* y *A* + *B* + *C* son expresiones nulas de acuerdo con las definiciones precedentes.

Escribamos un programa que lea e imprima una cadena de caracteres y luego imprima "válida" si la expresión lo es y "no válida" de no serlo. Se usan tres funciones para reconocer expresiones, términos y factores, respectivamente. Primero,

sin embargo, presentamos una función auxiliar *getsymb* que opera con tres parámetros: *str*, *length* y *ppos*. *str* contiene la entrada de cadena de caracteres; *length* representa el número de caracteres en *str*. *ppos* apunta a un entero *pos* cuyo valor es la posición en la *str* de la que obtuvimos un carácter la última vez. Si *pos < length*, *getsymb* regresa el carácter *cadena str[pos]* e incrementa *pos* en 1. Si *pos >= length*, *getsymb* regresa un espacio en blanco.

```
getsymb(str, length, ppos)
char str[]; int length, *ppos;
{
 /* La condición pos < length se cumple al principio de la ejecución de la función */
 char c;

 if (*ppos < length)
 c = str[*ppos];
 else
 c = ' ';
 (*ppos)++;
 return(c);
} /* fin de getsymb */
```

La función que reconoce una expresión se llama *expr*. Regresa *TRUE* (o 1) (*VERDADERO*) si una expresión válida comienza en la posición *pos* de *str* y *FALSE* (o 0) (*FALSO*) en caso contrario. También vuelve a colocar *pos* en la posición que sigue a la expresión de mayor longitud que puede encontrar. Suponemos también una función *readstr* que lee una cadena de caracteres, poniendo la cadena en *str* y su largo en *length*.

Una vez descritas las funciones *expr* y *readstr*, puede escribirse la rutina principal como sigue. La biblioteca estándar *ctype.h* incluye una función *isalpha* que es llamada por una de las funciones siguientes:

```
#include <stdio.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
#define MAXSTRINGSIZE 100
main()
{
 /* La condición pos < length se cumple al principio de la ejecución de la función */
 char str[MAXSTRINGSIZE];
 int length, pos;
 readstr(str, &length);
 pos = 0;
 if (expr(str, length, &pos) == TRUE && pos >= length)
 printf("%s", "válida");
 else
 printf("%s", "no válida");
} /* fin de main */
/* La condición puede fallar por una de dos razones */
```

```
/*
(o ambas). Si expr(str, length, &pos) == FALSE
entonces no hay una expresión válida al inicio de
pos. Si pos < length puede que se encuentre una
expresión válida, comenzando en pos,
pero no ocupa la cadena completa.
*/
} /* fin de main */
```

Las funciones *factor* y *term* se parecen mucho a *expr* excepto en que son responsables del reconocimiento de factores y términos, respectivamente. También reinicializan *pos* en la posición que sigue al factor o término de mayor longitud que se encuentra en la cadena *str*.

Los códigos para estas rutinas se apegan bastante a las definiciones dadas antes. Cada una intenta satisfacer uno de los criterios para la entidad que se reconoce. Si se satisface uno de esos criterios el resultado es *TRUE (VERDADERO)*. Si no se satisface ninguno, el resultado es *FALSE (FALSO)*.

```
expr(str, length, ppos)
char str[]; int length, *ppos;
{
 /* La condición pos < length se cumple al principio de la ejecución de la función */
 if (term(str, length, ppos) == FALSE)
 return(FALSE);
 /* Se ha encontrado un término; revisar el
 * siguiente símbolo.
 */
 if (get symb(str, length, ppos) != '+') {
 /* Se encontró la mayor expresión
 * (un solo término). Reposicionar pos para que
 * señale la última posición de
 * la expresión.
 */
 (*ppos)--;
 return(TRUE);
 } /* fin de if */
 /* En este punto, hemos encontrado un término
 * y un signo más. Se deberá buscar otro término.
 */
 return(term(str, length, ppos));
} /* fin de expr */
```

La rutina *term*, que reconoce términos, es muy similar, y la presentamos sin comentarios.

```
term(str, length, ppos)
char str[]; int length, *ppos;
{
 if (factor(str, length, ppos) == FALSE)
 return(FALSE);
 if (get symb(str, length, ppos) != '*') {
 (*ppos)--;
 return(TRUE);
 } /* fin de if */
 /* Se ha encontrado un término
 * y un signo multiplicación.
 */
 return(term(str, length, ppos));
} /* fin de term */
```

```

 return(TRUE);
} /* fin de if */
return(factor(str, length, ppos));
} /* fin de term */

```

La función *factor* reconoce factores y debería ser ahora bastante sencilla. Usa el programa común de biblioteca *isalpha* (esta función se encuentra en la biblioteca *ctype.h*), que regresa algo distinto de cero si su carácter de parámetro es una letra y cero (o *FALSO*) en caso contrario.

```

factor(str, length, ppos)
char str[];
int length, *ppos;
{
 int c;
 if ((c = getsymb(str, length, ppos)) != '(')
 return(isalpha(c));
 return(expr(str, length, ppos) &&
 getsymb(str, length, ppos) == ')');
} /* fin de factor */

```

Las tres rutinas son recursivas, dado que cada una se puede llamar a sí misma de manera indirecta. Por ejemplo, si se sigue la acción del programa para la cadena de entrada “ $(a * b + c * d) + (e * (f) + g)$ ” se encontrará que cada una de las tres rutinas *expr*, *term* y *factor* se llama a sí misma.

## EJERCICIOS

- 3.2.1. Determine qué calcula la siguiente función recursiva de C. Escriba una función iterativa que cumpla el mismo propósito.

```

func(n)
int n;
{
 if (n == 0)
 return(0);
 return(n + func(n-1));
} /* fin de func */

```

- 3.2.2. La expresión *min* en C indica el residuo de la división de *m* entre *n*. Defina el *máximo común divisor (MCD)* de dos enteros *x* y *y* mediante

```

gcd(x,y) = y si (y <= x && x % y == 0)
gcd(x,y) = gcd(y,x) si (x < y)
gcd(x,y) = gcd(y, x % y) de lo contrario

```

Escriba una función recursiva en C para calcular *gcd(x, y)*. Encuentre un método iterativo para calcular dicha función.

- 3.2.3. Sea *comm(n, k)* que representa el número de comités diferentes de *k* personas que pueden ser formados, dadas *n* personas de entre las cuales se puede elegir. Por ejemplo, *comm(4, 3) = 4*, dado que dadas 4 personas A, B, C y D hay cuatro posibles comités de tres personas: ABC, ABD, ACD y BCD. Pruebe la identidad:

$$\text{comm}(n, k) = \text{comm}(n - 1, k) + \text{comm}(n - 1, k - 1)$$

Escriba y pruebe un programa recursivo en C para calcular *comm(n, k)* para *n, k >= 1*.

- 3.2.4. Defina una *secuencia generalizada de Fibonacci* de *f0* y *f1* como la secuencia *gfib(f0, f1, 0), gfib(f0, f1, 1), gfib(f0, f1, 2), ...*, donde

$$\begin{aligned} \text{gfib}(f0, f1, 0) &= (f0, f1, 0) \\ \text{gfib}(f0, f1, 1) &= f1 \\ \text{gfib}(f0, f1, n) &= \text{gfib}(f0, f1, n-1) \\ &\quad + \text{gfib}(f0, f1, n-2) \text{ if } n > 1 \end{aligned}$$

Escriba una función en C recursiva para calcular *gfib(f0, f1, n)*. Encuentre un método iterativo para calcular dicha función.

- 3.2.5. Escriba una función recursiva para calcular el número de secuencias de *n* dígitos binarios que no contenga dos unos seguidos. (Sugerencia: calcule cuántas secuencias de este tipo existen que empiezan con 0 y cuántas que empiecen con 1.)

- 3.2.6. Una *matriz de orden n* es un arreglo  $n \times n$  de números. Por ejemplo,

(3)

es una matriz de  $1 \times 1$

(3)

es una matriz de  $2 \times 2$

(3)

- es una matriz de  $4 \times 4$ . Defina el *menor* de un elemento *x* en una matriz como la submatriz formada al borrar el renglón y la columna que contienen una *x*. En el ejemplo precedente de una matriz de  $4 \times 4$ , el menor del elemento 7 es la matriz de  $3 \times 3$

(3)

(3)

(3)

(3)

Se observa con claridad que el orden de un menor de cualquier elemento es 1 menos que el orden de la matriz original. Denote el menor de un elemento  $a[i, j]$  por  $\text{minor}(a[i, j])$ .

Defina el **determinante** de una matriz  $a$  (escrita  $\det(a)$ ) recursivamente como sigue:

- Si  $a$  es una matriz de  $1 \times 1$  ( $x$ ),  $\det(a) = x$ .
- Si  $a$  es de orden mayor que 1, calcule el determinante de  $a$  de la siguiente manera:

- Elija cualquier renglón o columna. Para cada elemento  $a[i, j]$  en ese renglón o columna forme el producto

$$\text{power}(-1, i + j) * a[i, j] * \det(\text{minor}(a[i, j]))$$

donde  $i$  y  $j$  son las posiciones de renglón y columna de los elementos elegidos,  $a[i, j]$  es el elemento que se escogió,  $\det(\text{minor}(a[i, j]))$  es el determinante del menor de  $a[i, j]$  y  $\text{power}(m, n)$  es el valor de  $m$  elevado a la potencia  $n$ -ésima.

- $\det(a) = \text{suma de todos esos productos}$ .  
(De manera más concisa, si  $n$  es el orden de  $a$ ,

$$\det(a) = \sum_i \text{power}(-1, i + j) * a[i, j] * \det(\text{minor}(a[i, j])), \text{ for any } j$$

$$\det(a) = \sum_j \text{power}(-1, i + j) * a[i, j] * \det(\text{minor}(a[i, j])), \text{ for any } i$$

Escriba un programa en C que lea  $a$ , la imprima en forma de matriz, e imprima el valor de  $\det(a)$ , donde  $\det$  es una función que calcula el determinante de una matriz.

### 3.2.7. Escriba un programa recursivo en C para ordenar un arreglo $a$ como sigue:

- Sea  $k$  el índice del elemento medio del arreglo.
- Ordene los elementos hasta  $a[k]$  inclusive.
- Ordene los elementos después de  $a[k]$ .
- Mezcle los dos subarreglos en uno solo ordenado.

Este método se llama ordenamiento por intercalación (*merge sort*).

### 3.2.8. Muestre cómo transformar el siguiente procedimiento iterativo en uno recursivo. $f(i)$ es una función que regresa un valor lógico basado en el valor de $i$ , y $g(i)$ es una función que regresa un valor con el mismo atributo que $i$ .

```
iter(n)
int n;
{
 int i;
 i = n;
 while(f(i) == TRUE) {
 /* ... */
 }
}
```

```
/* cualquier grupo de instrucciones de C */
/* no cambie el valor de i */
i = g(i);
} /* fin de while */
} /* fin de iter */
```

## 3.3. COMO CODIFICAR PROGRAMAS RECURSIVOS

En la sección anterior se vio cómo transformar una definición o algoritmo recursivo en un programa en C. Es mucho más difícil desarrollar una solución recursiva en C para resolver un problema específico cuando no se tiene algoritmo. No es sólo el programa sino las definiciones originales y los algoritmos los que deben desarrollarse. En general, cuando encaramos la tarea de escribir un programa para resolver un problema no hay razón para buscar una solución recursiva. La mayoría de los problemas pueden resolverse de una manera directa usando métodos no recursivos. Sin embargo, otros pueden resolverse de una manera más lógica y elegante mediante la recursión. En esta sección se tratará de identificar los problemas que pueden resolverse de manera recursiva, se desarrollará una técnica para encontrar soluciones recursivas y se darán algunos ejemplos.

Volvamos a examinar la función factorial. El factorial es, probablemente, un ejemplo fundamental de un problema que no debe resolverse de manera recursiva, dado que su solución iterativa es directa y simple. Sin embargo, examinemos los elementos que permiten dar una solución recursiva. Antes que nada, puede reconocerse un gran número de casos distintos que se deben resolver. Es decir, quiere escribirse un programa para calcular  $0!$ ,  $1!$ ,  $2!$  y así sucesivamente. Puede identificarse un caso “trivial” para el cual la solución no recursiva puede obtenerse en forma directa. Es el caso de  $0!$ , que se define como 1. El siguiente paso es encontrar un método para resolver un caso “complejo” en términos de uno más “simple”, lo cual permite la reducción de un problema complejo a uno más simple. La transformación del caso complejo al simple resultaría al final en el caso trivial. Esto significaría que el caso complejo se define, en lo fundamental, en términos del más simple.

Examinemos qué significa lo anterior cuando se aplica a la función factorial.  $4!$  es un caso más complejo que  $3!$ . La transformación que se aplica al número  $4$  para obtener  $3$  sencillamente es restar 1. Si restamos 1 de 4 de manera sucesiva llegamos a 0, que es el caso trivial. Así, si se puede definir  $4!$  en términos de  $3!$  y, en general,  $n!$  en términos de  $(n - 1)!$ , se podrá calcular  $4!$  mediante la definición de  $n!$  en términos de  $(n - 1)!$  al trabajar, primero hasta llegar a  $0!$  y luego al regresar a  $4!$ . En el caso de la función factorial se tiene una definición de ese tipo, dado que:

$$n! = n * ((n - 1)!)$$

Así,  $4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 = 24$ .

Estos son los ingredientes esenciales de una rutina recursiva: poder definir un caso “complejo” en términos de uno “más simple” y tener un caso “trivial” (no recursivo) que pueda resolverse de manera directa. Al hacerlo, puede desarrollarse una

solución si se supone que se ha resuelto el caso más simple. La versión en C de la función factorial supone que está definido ( $n - 1$ )! y usa esa cantidad al calcular  $n$ !.

Veamos cómo se aplican esas ideas a otros ejemplos de las secciones previas. En la definición de  $a * b$ , es trivial el caso de  $b = 1$ , pues  $a * b$  es igual a  $a$ . En general,  $a * b$  puede definirse en términos de  $a * (b - 1)$  mediante la definición  $a * b = a * (b - 1) + a$ . De nuevo, el caso complejo se transforma en un caso más simple al restar 1, lo que lleva, al final, al caso trivial de  $b = 1$ . Aquí la recursión se basa únicamente en el segundo parámetro,  $b$ .

En el caso de la función de Fibonacci, se definieron dos casos triviales:  $fib(0) = 0$  y  $fib(1) = 1$ . Un caso complejo  $fib(n)$  se reduce entonces a dos más simples:  $fib(n - 1)$  y  $fib(n - 2)$ . Esto se debe a la definición de  $fib(n)$  como  $fib(n - 1) + fib(n - 2)$ , donde se requiere de dos casos triviales definidos de manera directa.  $fib(1)$  no puede definirse como  $fib(0) + fib(-1)$  porque la función de Fibonacci no está definida para números negativos.

La búsqueda binaria es un caso interesante de recursión. La recursión se basa en el número de elementos del arreglo que debe buscarse. Cada vez que se llama a la rutina de manera recursiva, el número de elementos que se busca se divide en dos partes (lo más parecidas en tamaño). El caso trivial es aquel en el cual no hay ningún elemento con el cual comparar el que se busca o éste se encuentra en medio del arreglo. Si  $low > high$ , se cumple la primera de las dos condiciones —1 es regresado. Si  $x == a[mid]$ , se cumple la segunda y se regresa a  $mid$ . En el caso más complejo de  $high - low + 1$  elementos para buscarse, la búsqueda se reduce a tomar lugar en una de dos regiones divididas, 1. la mitad inferior del arreglo, de  $low$  a  $mid - 1$ ; 2. la mitad superior del arreglo, de  $mid + 1$  a  $high$ . Así, un caso complejo (una gran área donde debe buscarse el elemento) se reduce a uno más simple (un área donde debe buscarse el elemento cuyo tamaño, es aproximadamente, de la mitad del área original). Esto se reduce, por último, a la comparación con un solo elemento ( $a[mid]$ ) o a buscar en un arreglo que no contenga ninguno.

### El problema de las Torres de Hanoi

Hasta aquí se han visto definiciones de recursión y se ha examinado cómo se ajustan a la pauta establecida. Se verá ahora cómo pueden usar técnicas recursivas para lograr una solución lógica y elegante de un problema que no se especifica en términos recursivos. El problema es el de “las Torres de Hanoi”, cuyo planteamiento inicial se muestra en la figura 3.3.1. Hay tres postes: A, B y C. En el poste A se ponen cinco discos de diámetro diferente de tal manera que un disco de diámetro mayor siempre queda debajo de uno de diámetro menor. El objetivo es mover los cinco discos al poste C usando el B como auxiliar. Sólo puede moverse el disco superior de cualquier poste a otro poste, y un disco mayor jamás puede quedar sobre uno menor. Considérese la posibilidad de encontrar una solución. En efecto, ni siquiera es claro que exista alguna.

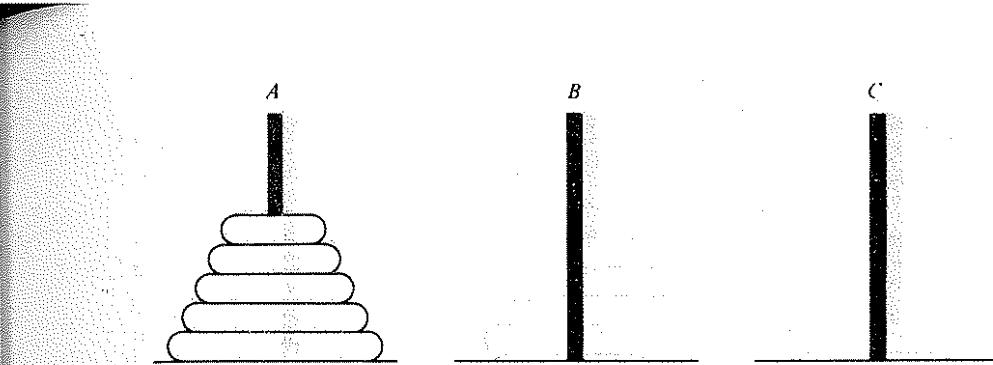


Figura 3.3.1 Planteamiento inicial de las Torres de Hanoi.

Ahora se verá si se puede desarrollar una solución. En lugar de concentrar la atención en una solución para cinco discos, considérese el caso general de  $n$  discos. Supóngase que se tiene una solución para  $n - 1$  discos y que, en términos de ésta, se pueda plantear la solución para  $n - 1$  discos. El problema se resolvería entonces. Esto sucede porque en el caso trivial de un disco (al restar 1 de  $n$  de manera sucesiva se producirá, al final, 1) la solución es simple: sólo hay que mover el único disco del poste A a C. Así se habrá desarrollado una solución recursiva si se plantea una solución para  $n$  discos en términos de  $n - 1$ . Considérese la posibilidad de encontrar tal relación. Para el caso de cinco discos en particular, supóngase que se conoce la forma de mover cuatro de ellos del poste A al otro, de acuerdo con las reglas. ¿Cómo puede completarse entonces el trabajo de mover el quinto disco? Cabe recordar que hay 3 postes disponibles.

Supóngase que se supo cómo mover cuatro discos del poste A al C. Entonces, se podrá mover éstos exactamente igual hacia el B usando el C como auxiliar. Esto da como resultado la situación representada en la figura 3.3.2a. Entonces podrá moverse el disco mayor de A a C (figura 3.3.2b) y por último aplicarse de nuevo la solución para cuatro discos para moverlos de B a C, usando el poste A, ahora vacío, de forma auxiliar (figura 3.3.2c). Por lo tanto, se puede establecer una solución recursiva al problema de las Torres de Hanoi como sigue:

Para mover  $n$  discos de A a C usando B como auxiliar:

1. Si  $n == 1$ , mover el disco único de A a C y parar.
2. Mover el disco superior de A a B  $n - 1$  veces, usando C como auxiliar.
3. Mover el disco restante de A a C.
4. Mover los discos  $n - 1$  de B a C usando A como auxiliar.

Con toda seguridad este algoritmo producirá una solución correcta para cualquier valor de  $n$ . Si  $n == 1$ , el paso 1 será la solución correcta. Si  $n == 2$ , se sabe entonces que hay una solución para  $n - 1 == 1$ , de manera tal que los pasos 2 y 4 se ejecutarán en forma correcta. De manera análoga, cuando  $n == 3$ , ya se habrá producido una solución para  $n - 1 == 2$ , por lo que los pasos 2 y 4 pueden ser ejecutados. De esta forma se puede mostrar que la solución funciona para  $n == 1, 2,$

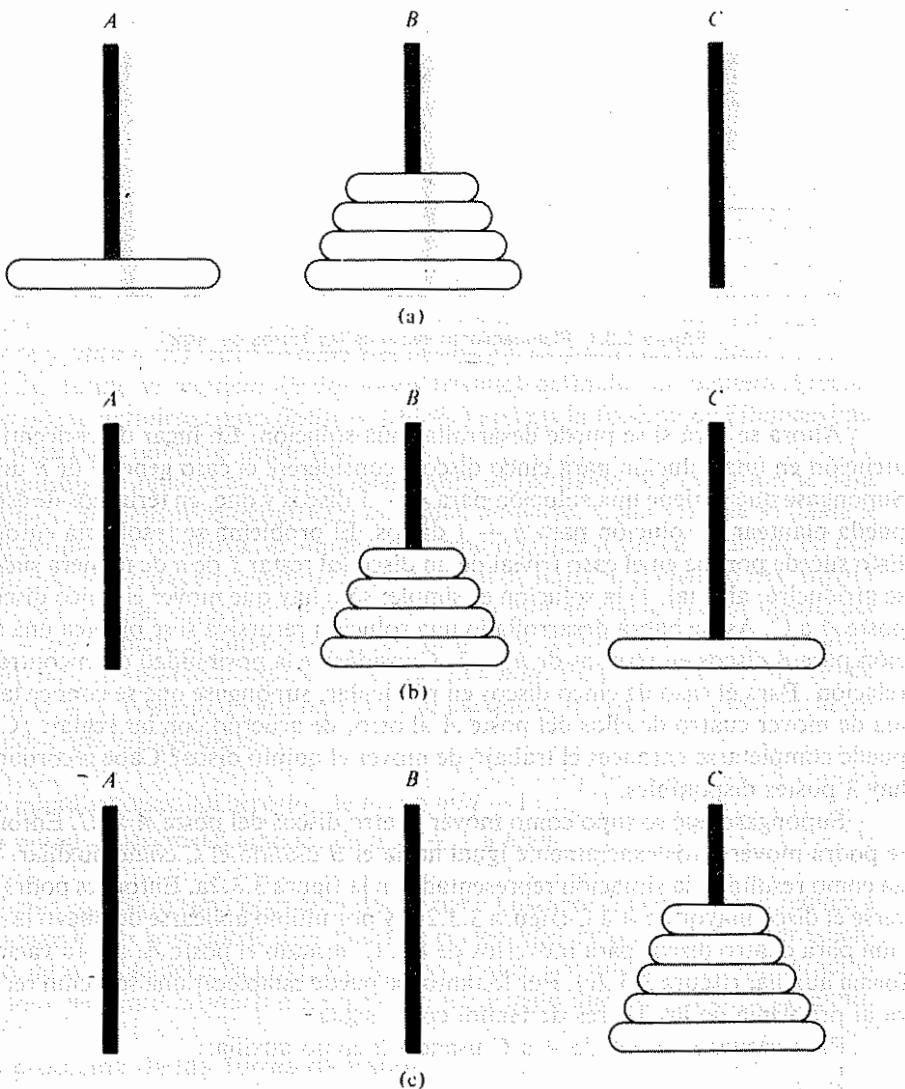


Figura 3.3.2 Solución recursiva al problema de las Torres de Hanoi.

3, 4, 5, ... hasta el valor para el que se desee encontrar una solución. Adviértase que la solución se desarrolló mediante la identificación de un caso trivial ( $n = 1$ ) y una solución para el caso general y complejo ( $n$ ) en términos de un caso más simple ( $n - 1$ ).

¿Cómo se puede convertir esta solución en un programa en lenguaje C? Aquí ya no se trata con una función matemática como factorial, sino con acciones concretas como "mover un disco". ¿Cómo deben representarse dichas acciones en la computadora? El problema no se especifica por completo. ¿Cuáles son las entradas del programa? ¿Cuáles deben ser las salidas? Siempre que se le pida escribir un programa, se deben recibir instrucciones específicas sobre lo que se espera que haga

el programa con exactitud. Es insuficiente, un planteamiento del problema como: "resolver el problema de las Torres de Hanoi". La especificación de tal problema significa por lo general que, además del programa, se deben diseñar las entradas y salidas, de manera que correspondan a la descripción del problema.

El diseño de entradas y salidas es una fase importante de la solución y éste debe recibir tanta atención como el resto del programa. Existen dos razones para ello. La primera es que el usuario (que es en última instancia quien evalúa y emite juicios acerca del trabajo) no verá el refinado método que se incorporó al programa, si no que se esforzará mucho en descifrar la salida o en adaptar los datos de entrada a los usos particulares de entrada establecidos. El descuido de no convenir a tiempo los detalles de entrada y salida ha sido causa de múltiples aflicciones tanto para los programadores como para los usuarios. La segunda razón es que un leve cambio en el formato de entrada o salida puede facilitar en gran medida el diseño del programa. Así, el programador puede hacer el trabajo mucho más fácilmente si tiene la capacidad de diseñar un formato de entrada y salida compatible con el algoritmo. Claro que estas dos consideraciones —la de la conveniencia para el usuario y la del programador— entran en conflicto con frecuencia, por lo que es necesario encontrar un justo medio. Sin embargo, tanto el usuario como el programador deben participar de manera activa en las decisiones acerca de los formatos de entrada y salida.

Ahora puede procederse al diseño de entradas y salidas de este programa. La única entrada necesaria es el valor de  $n$ , el número de discos. Por lo menos, éste puede ser el punto de vista del programador. Es probable que el usuario desee los nombres de los discos (como "rojo", "verde", "azul", etc.) y los nombres de los postes (como "izquierdo", "derecho" y "central"). El programador quizás pueda convencer al usuario de que nombrar los discos como 1, 2, 3, ...,  $n$  y postes como  $A$ ,  $B$ ,  $C$  es igual de conveniente. Si el usuario es obstinado, el programador puede escribir una pequeña función para convertir los nombres del usuario en los suyos y viceversa.

Una forma razonable para la salida podría ser una lista de instrucciones como:

`mover disco nnn del poste yyy al poste zzz`

donde  $nnn$  es el número de disco que se debe mover, y  $yyy$  y  $zzz$  son los nombres de los postes implicados en dicho movimiento. La acción que deberá emprenderse para obtener una solución consiste en ejecutar cada una de las instrucciones de salida en el orden en que aparecen en la misma.

El programador decide entonces escribir la subrutina `towers` (en la que a propósito es ambiguo acerca de los parámetros, en este punto) para imprimir la salida antes mencionada. El programa principal sería

```
main()
{
 int n;
 scanf("%d", &n);
 towers(parameters);
} /* fin de main */
```

Bajo la suposición de que el usuario está satisfecho con denominar los discos 1, 2, 3, ...,  $n$  y los postes  $A$ ,  $B$ ,  $C$ , ¿cuáles serán los parámetros de *towers*? Es evidente que debe incluir a  $n$ , el número de discos que debe moverse. Esto no sólo incluye información acerca de cuántos discos hay, sino también de cuáles son sus nombres. El programador advierte entonces que en el algoritmo recursivo se deben mover  $n - 1$  discos mediante una llamada recursiva a *towers*. Así, en la llamada recursiva, el primer parámetro de *towers* será  $n - 1$ . Pero esto implica que los  $n - 1$  discos superiores tienen la numeración 1, 2, 3, ...,  $n - 1$  y que el menor de todos tiene el número 1. Esto es un buen ejemplo de cómo programar la conveniencia luego de determinar la representación del problema. A priori, no hay razones para etiquetar el disco más pequeño con el número 1; de manera lógica, podría haberse etiquetado el disco más grande con el número 1 y el más pequeño con  $n$ . Sin embargo, como esto conduce a un programa más simple y directo, se optó por etiquetar los discos de tal manera que al disco menor le corresponda el número menor.

¿Cuáles son los otros parámetros de *towers*? A primera vista, parecería que no se necesitan parámetros adicionales, debido a que los postes tienen denominación de  $A$ ,  $B$  y  $C$  por omisión. Sin embargo, una observación más detallada de la solución recursiva da pauta para comprender que en las llamadas recursivas no se trasladan los discos de  $A$  a  $C$  usando  $B$  como auxiliar, sino de  $A$  a  $B$  usando  $C$  (paso 2) o de  $B$  a  $C$  usando  $A$  (paso 4). Por lo tanto, se incluyen tres parámetros más en *towers*. El primero, *frompeg*, representa el poste del que se retiran los discos; el segundo, *topeg*, el poste en que se colocarán los discos; y el tercero, *auxpeg*, el poste auxiliar. Esta situación es típica de las rutinas recursivas; se requieren parámetros adicionales para tratar la situación de llamada recursiva. Un ejemplo de esto puede observarse en el programa de búsqueda binaria, en el que se requirieron los parámetros *low* y *high*.

El programa completo para resolver el problema de las Torres de Hanoi, siguiendo muy de cerca la solución recursiva, se puede escribir como sigue:

```
#include <stdio.h>
main()
{
 int n;
 scanf("%d", &n);
 towers(n, 'A', 'C', 'B');
} /* fin de main */

towers(n, frompeg, topeg, auxpeg)
int n;
char auxpeg, frompeg, topeg;
{
 /* If Si es sólo un disco, efectuar movimiento y regresar */
 if (n == 1) {
 printf("/n%d%d%s%c%s%c%", "mover disco 1 del poste",
 frompeg, "al poste", topeg);
 return;
 } /* fin de if */
 /* Mover los n-1 discos de arriba de A a B, usando */
}
```

```
/* C como auxiliar */
towers(n-1, frompeg, auxpeg, topeg);
/* Mover el disco restante de A a C */
printf("/n%d%d%s%c%s%c%", "mover disco", n, "del poste",
 frompeg, "al poste", topeg);
/* Mover n-1 discos de B hacia C empleando */
/* A como auxiliar */
towers(n-1, auxpeg, topeg, frompeg);
} /* fin de towers */
```

Trazar la acción del programa anterior cuando éste lea el valor 4 para  $n$ . Se debe tener cuidado de tomar en cuenta los valores cambiantes de los parámetros *frompeg*, *auxpeg* y *topeg* y verificar que produzca la siguiente salida:

|               |           |     |          |     |
|---------------|-----------|-----|----------|-----|
| Mover disco 1 | del poste | $A$ | al poste | $B$ |
| Mover disco 2 | del poste | $A$ | al poste | $C$ |
| Mover disco 1 | del poste | $B$ | al poste | $C$ |
| Mover disco 3 | del poste | $A$ | al poste | $B$ |
| Mover disco 1 | del poste | $C$ | al poste | $A$ |
| Mover disco 2 | del poste | $C$ | al poste | $B$ |
| Mover disco 1 | del poste | $A$ | al poste | $B$ |
| Mover disco 4 | del poste | $A$ | al poste | $C$ |
| Mover disco 1 | del poste | $B$ | al poste | $C$ |
| Mover disco 2 | del poste | $B$ | al poste | $A$ |
| Mover disco 1 | del poste | $C$ | al poste | $A$ |
| Mover disco 3 | del poste | $B$ | al poste | $C$ |
| Mover disco 1 | del poste | $A$ | al poste | $B$ |
| Mover disco 2 | del poste | $A$ | al poste | $C$ |
| Mover disco 1 | del poste | $B$ | al poste | $C$ |
| Mover disco 2 | del poste | $C$ | al poste | $A$ |
| Mover disco 1 | del poste | $B$ | al poste | $C$ |
| Mover disco 3 | del poste | $A$ | al poste | $B$ |
| Mover disco 1 | del poste | $C$ | al poste | $A$ |
| Mover disco 2 | del poste | $C$ | al poste | $B$ |
| Mover disco 1 | del poste | $A$ | al poste | $B$ |
| Mover disco 2 | del poste | $A$ | al poste | $C$ |
| Mover disco 1 | del poste | $B$ | al poste | $C$ |

Verificar que la solución anterior funciona en efecto y que no viole las reglas.

#### Conversión de prefija a postfixa por medio de la recursión

A continuación se examina otro problema para el cual la solución recursiva es la más directa y refinada. En este problema se trata de convertir una expresión prefija a postfixa. Las notaciones prefija y postfixa se analizaron en el capítulo anterior. En resumen, las notaciones prefija y postfixa son métodos para escribir una expresión matemática sin usar paréntesis. En la notación prefija cada operador precede de inmediato a sus operandos; en la postfixa, les sigue. A manera de recordatorio,enseguida se presentan unas cuantas expresiones matemáticas (infijas) con sus equivalentes prefijas y postfixas:

| infija                      | prefija            | postfixa    |
|-----------------------------|--------------------|-------------|
| $A + B$                     | +AB                | AB+         |
| $A + B * C$                 | +A * BC            | ABC*+       |
| $A * (B + C)$               | * A + BC           | ABC+*       |
| $A * B + C$                 | + * ABC            | AB*C+       |
| $A + B * C + D - E * F$     | - + + A * BCD * EF | ABC*+D+EF*- |
| $(A + B) * (C + D - E) * F$ | * * + AB - + CDEF  | AB+CD+E-*F* |

La mejor forma de definir postfija y prefija es por medio de la recursión. Si se asume que no hay constantes y que sólo las letras individuales son variables, una expresión prefija es una letra individual, o un operador seguido de dos expresiones prefijas. Es posible definir de manera similar una expresión postfija como una letra individual, o un operador precedido por dos expresiones postfijas. Las definiciones anteriores suponen que todas las operaciones son binarias y esto es, que cada una requiere de dos operandos. La suma, la resta, multiplicación, división y la exponentiación son ejemplos de operaciones binarias. Resulta fácil prolongar las definiciones de prefija y postfija anteriores a las operaciones unarias, como el factorial o la negación, pero por cuestión de simplicidad no se hará aquí. Es necesario verificar la validez de cada una de las expresiones postfijas y prefijas anteriores mostrando que se apegan a las definiciones y asegurándose de que es posible identificar los dos operandos de cada operador:

Más adelante se usarán estas definiciones recursivas, pero antes hay que regresar al problema inicial. Dada una expresión prefija, ¿cómo puede transformarse ésta a postfija? Es posible identificar de inmediato un caso trivial: cuando una expresión prefija consiste de una sola variable, ésta equivale a su propia expresión postfija. Es decir, una expresión como *A* es válida tanto como prefija como postfija.

Considérese ahora una cadena prefija más larga. Si se supo cómo convertir una cadena prefija corta a postfija, ¿es posible convertir esta cadena más larga a postfija? La respuesta es sí, con una excepción. Cualquier cadena prefija que conste de más de una sola variable contiene un operador y un primer y un segundo operandos (cabe recordar que se asumieron únicamente operadores binarios). Si se asume la posibilidad de identificar el primero y el segundo operandos, que son necesariamente más cortos que la cadena original, se puede transformar entonces la cadena prefija larga a postfija convirtiendo en primer lugar el primer operando a postfija; luego el segundo, que se coloca detrás del primero, y por último, se coloca el operador inicial detrás de éstos en la cadena resultante. De este modo se desarrolla un algoritmo recursivo para convertir una cadena prefija a postfija, con la salvedad de que en una expresión prefija se debe especificar un método para identificar los operandos. El algoritmo puede resumirse como sigue:

1. Si la cadena prefija es una sola variable, ésta es su propia equivalencia postfija.
2. *op* será el primer operador de la cadena prefija.
3. Encontrar el primer operando, *opnd1*, de la cadena. Convertirlo a postfija y denominarlo *post1*.
4. Encontrar el segundo operando, *opnd2*, de la cadena. Convertirlo a postfija y denominarlo *post2*.
5. Concatenar *post1*, *post2* y *op*.

Una operación necesaria en este programa es la de la concatenación. Por ejemplo, si dos cadenas representadas por *a* y *b* representan las cadenas "abcde" y "xyz" respectivamente, la llamada a la función

```
strcat(a, b)
```

coloca en *a* la cadena "abcdxyz" (esto es, la cadena constituida por todos los elementos de *a* seguidos por todos los de *b*). También son necesarias las funciones *strlen* y *substr*. La función *strlen* (*str*) da como resultado el largo de la cadena *str*. La función *substr*(*s1*, *i*, *j*, *s2*) forma la cadena *s2* a partir de la subcadena de *s1* comenzando en la posición *i* que contiene *j* caracteres. Por ejemplo, después de ejecutar *substr*("abcd", 1, 3, *s*), *s* es igual a "bc". Las funciones *strcat*, *strlen* y *substr*, son por lo general funciones estándar de biblioteca para cadenas en C.

Antes de transformar el algoritmo de conversión a un programa en C, es necesario examinar sus entradas y salidas. Se desea escribir un procedimiento *convert* que acepte una cadena de caracteres, la que representa una expresión prefija en la que todas las variables son letras individuales y los operadores permisibles son '+', '-', '\*' y '/'. El procedimiento genera una cadena postfija equivalente a la cadena parámetro prefija.

Supóngase la existencia de una función *find* que acepta una cadena y da como resultado un entero que es del mismo tamaño que la expresión previa más larga contenida en la cadena de entrada desde el principio de la misma. Por ejemplo, *find*("A + CD") da como resultado 1, ya que "A" es la cadena prefija más larga encontrada al principio de "A + CD". *find*("+ \* ABCD + GH") da como resultado 5, ya que "+ \* ABC" es la cadena prefija más larga que comienza donde lo hace la cadena dada. Si no existe una cadena prefija que comience donde inicia la cadena de entrada *find* regresa 0. (Por ejemplo, *find*("\* + AB") es igual a 0.) Esta función se usa para identificar el primer y el segundo operandos de un operador prefijo. *convert* también llama a la función de la biblioteca *isalpha*, la que determina si su parámetro es una letra. Si se asume la existencia de la función *find*, una rutina de conversión puede escribirse como sigue:

```
convert (prefix, postfix)
 char prefix[], postfix[];
{
 /* ... */
 char opnd1[MAXLENGTH], opnd2[MAXLENGTH];
 char post1[MAXLENGTH], post2[MAXLENGTH];
 char temp[MAXLENGTH];
 char op[2];
 int length;
 int i, j, m, n;

 if ((length = strlen(prefix)) == 1) {
 if (isalpha(prefix[0])) {
 /* ... La cadena prefija es una sola letra ... */
 postfix[0] = prefix[0];
 postfix[1] = '\0';
 return;
 } /* fin de if */
 printf("\ncadena prefija ilegal ");
 exit(1);
 } /* fin de if */
 /* ... La cadena prefija contiene más de un carácter. Extraer las longitudes de los ... */
 /* ... */
}
```

```

/*
 dos operandos y del operador.
*/
op[0] = prefix[0];
op[1] = '\0';
substr(prefix, 1, length-1, temp);
m = find(temp);
substr(prefix, m+1, length-m-1, temp);
n = find(temp);
if ((op[0] != '+' && op[0] != '-' && op[0] != '*' &&
 op[0] != '/') || (m == 0) || (n == 0)
 || (m+n+1 != length)) {
 printf("/ncadena prefija ilegal");
 exit(1);
} /* fin de if */
substr(prefix, 1, m, opnd1);
substr(prefix, m+1, n, opnd2);
convert(opnd1, post1);
convert(opnd2, post2);
strcat(post1, post2);
strcat(post1, op);
substr(post1, 0, length, postfix);
} /* fin de convert */

```

Hay que advertir que se han incorporado varias verificaciones dentro de *convert* para asegurar que el parámetro sea una cadena prefija válida. Uno de los tipos de errores más difíciles de detectar es aquel en que los errores son resultado de una entrada no válida y del descuido del programador en verificar su validez.

Ahora prestemos atención a la función *find*, la que acepta una cadena de caracteres y una posición inicial y da como resultado la longitud de la cadena prefija mayor contenida en la cadena de entrada dada, comenzando en dicha posición. La palabra "mayor" en esta definición es superflua, ya que hay a lo más una subcadena que comienza en una posición determinada de una cadena que es una expresión prefija válida.

Primero se muestra que hay a lo sumo una expresión prefija válida que comienza al principio de una cadena. Para comprobar esto, debe advertirse que se cumple de manera trivial en una cadena 1 de largo, y supóngase que es verdadero para una cadena corta. Entonces una cadena larga que contenga una expresión prefija como subcadena inicial debe comenzar con una variable, en cuyo caso la variable es la subcadena deseada, o con un operador. En caso de borrar el operador inicial, el resto de la cadena será más corta que la cadena original y, en consecuencia, tener a lo sumo una expresión inicial prefija. Esta expresión es el primer operando del operador inicial. De manera análoga, la subcadena restante (luego de borrar el primer operador) sólo puede tener una subcadena inicial simple, que es una expresión prefija. Esta expresión debe ser el segundo operando. Así se identifica de manera única el operador y los operandos de la expresión prefija comenzando en el primer carácter de una cadena arbitraria, en caso de que exista tal expresión. Ya que existe a lo más una cadena prefija válida que comienza al principio de cualquier cadena, hay a lo sumo una cadena que comienza en cualquier posición de una cadena arbitraria. Esto es

obvio si se considera que la subcadena de la cadena dada comienza en una posición determinada.

Adviértase que esta prueba proporciona un método recursivo para encontrar una expresión prefija en una cadena. Ahora incorpórese este método a la función *find*:

```

find(str)
char str[]; /* Cadena de caracteres que contiene la cadena prefija */
{
 /* Si la cadena es vacía, no es una expresión prefija */
 if (length = strlen(str)) == 0)
 return (0);
 if (isalpha(str[0]) != 0)
 /* El primer carácter es una letra. */
 /* Esa letra es la subcadena inicial */
 return (1);
 /* De otra manera, encontrar el primer operando */
 if (strlen(str) < 2)
 return (0);
 substr(str, 1, length-1, temp);
 m = find(temp);
 if (m == 0 || strlen(str) == m)
 /* no es un operando prefijo válido */
 /* o no hay segundo operando */
 return (0);
 substr(str, m+1, length-m-1, temp);
 n = find(temp);
 if (n == 0)
 return (0);
 return (m+n+1);
} /* fin de find */

```

Para verificar la comprensión de cómo funcionan estas rutinas, hay que trazar sus acciones sobre expresiones válidas y no válidas. Más importante aún es comprobar si se entendió cómo fueron desarrolladas y cómo el análisis lógico condujo a una solución recursiva natural que se pudo traducir en forma directa a un programa en C.

## EJERCICIOS

- 3.3.1. Suponga que se añadió otra excepción al problema de las Torres de Hanoi: que un disco no pueda descansar sobre otro que lo rebasa en más de una talla (por ejemplo, el disco 1 sólo podrá descansar sobre el 2 o en la base; el dos sobre el tres o en la base, y así sucesivamente). ¿Por qué no funciona la solución del texto? ¿Qué es lo que no funciona en la lógica, que conduce a ello de acuerdo con las nuevas reglas?

- 3.3.2. Pruebe que el número de movimientos que ejecuta *towers* para mover  $n$  discos es igual a  $2^n - 1$ . ¿Es posible encontrar un método con menos movimientos para resolver el problema de las Torres de Hanoi? Encontrar dicho método para algunas  $n$  o probar que no existe ninguno.
- 3.3.3. Defina una expresión postfija y prefija que incluya la posibilidad de operadores unarios. Escribir un programa para convertir una expresión prefija que contenga el operador de negación unaria (representada por el símbolo '@') a postfija.
- 3.3.4. Reescriba la función *find* del texto de manera que no sea recursiva y que calcula la longitud de una cadena prefija contando el número de operadores y los operandos de una sola letra.
- 3.3.5. Escriba una función recursiva que acepte una expresión prefija que consista en operadores binarios y operandos enteros de un solo dígito y que dé como resultado el valor de la expresión.
- 3.3.6. Considere el siguiente procedimiento para convertir una expresión prefija a postfija. *conv(prefix, postfix)* se usaría para llamar la rutina.

```
conv(prefix, postfix)
char prefix[], postfix[];
{
 char first[2];
 char t1[MAXLENGTH], t2[MAXLENGTH];

 first[0] = prefix[0];
 first[1] = '\0';
 substr(prefix, 1, strlen(prefix) - 1, prefix);
 if (first[0] == '+' || first[0] == '*' ||
 || first[0] == '-' || first[0] == '/'). {
 conv(prefix, t1);
 conv(prefix, t2);
 strcat(t1, t2);
 strcat(t1, first);
 substr(t1, 0, strlen(t1), postfix);
 return;
 } /* fin de if */
 postfix[0] = first[0];
 postfix[1] = '\0';
} /* fin de conv */
```

Explique cómo funciona el procedimiento. ¿Es mejor o peor que el método del texto? ¿Qué ocurre cuando se llama la rutina mediante una cadena prefija no válida como entrada? ¿Se puede incorporar una verificación para cadenas no válidas en *convert*? ¿Es posible diseñar semejante verificación para el programa de llamada después de que *convert* regresa? ¿Cuál es el valor de  $n$  tras el regreso de *convert*?

- 3.3.7. Desarrolle un método recursivo (y pruébelo) para calcular el número de maneras diferentes en que un número entero  $k$  pueda escribirse como una suma, en la que cada uno de los operandos sea menor que  $n$ .
- 3.3.8. Considere un arreglo  $a$  que contenga enteros positivos y negativos. Defina *contigsum(i, j)* como la suma de los elementos contiguos desde  $a[i]$  hasta  $a[j]$  para todos los índices del arreglo con  $i \leq j$ . Desarrolle un procedimiento recursivo que determine  $i$  y  $j$  de modo que *contigsum(i, j)* sea aumentada al máximo. La recursión deberá considerar las dos mitades del arreglo  $a$ .

- 3.3.9. Escriba un programa recursivo en C para encontrar el  $k$ -ésimo elemento más pequeño de un arreglo  $a$  de números, eligiendo cualquier elemento  $a[i]$  de  $a$  y seccionando  $a$  en aquellos elementos más pequeños, iguales y mayores que  $a[i]$ .
- 3.3.10. El problema de las ocho reinas es colocar ocho reinas en un tablero de ajedrez de manera que ninguna ataque a las demás. El siguiente es un programa recursivo para resolver dicho problema. *board* es un arreglo de  $8 \times 8$  que representa el tablero.  $board[i][j] == \text{TRUE}$  (es verdadero) si hay una reina en la posición  $[i][j]$ , y  $\text{FALSE}$  (falso) en caso contrario. *good()* es una función que da como resultado *TRUE* (verdadero) cuando dos reinas del tablero no se atacan entre sí, y *FALSE* en caso contrario. Al final del programa, la rutina *drawboard()* muestra una solución del problema.

```
static short int board [8][8];
#define TRUE 1
#define FALSE 0

main()
{
 int i, j;

 for(i=0; i<8; i++)
 for(j=0; j<8; j++)
 board[i][j] = FALSE;
 if (try(0) == TRUE)
 drawboard();
 } /* fin de main */

try(n)
int n;
{
 int i;
 for(i=0; i<8; i++) {
 board[n][i] = TRUE;
 if (n == 7 && good() == TRUE)
 return(TRUE);
 if (n < 7 && good() == TRUE && try(n+1) == TRUE)
 return(TRUE);
 board[n][i] = FALSE;
 } /* fin de for */
 return (FALSE);
} /* fin de try */
```

Determinado el tablero (*board*) al momento de llamarla, la función recursiva *try* da como resultado *TRUE*, siempre que es posible, a fin de agregar reinas en los renglones del  $n$  al 7 para alcanzar la solución. *try* da como resultado *FALSE* si no hay una solución que tenga reinas en las posiciones de *board* que ya contienen *TRUE*. Si el re-

sultado es *TRUE*, la función también agrégaa reinas en los renglones del *n* al 7 para generar una solución.

Escriba las funciones *good* y *drawboard* anteriores, y verifique que el programa genere una solución.

(La idea que hay detrás de la solución es la siguiente: *board* representa la situación global durante la tentativa de encontrar una solución. El siguiente paso en la búsqueda de la solución se escoge arbitrariamente (se coloca una reina en la siguiente posición no probada del renglón *n*). Luego se comprueba de manera recursiva si es posible generar una solución que incluya ese paso. Si lo es, se regresa. Si no, se explora a la inversa a partir del siguiente paso intentado —*board[n][i] = FALSE*— y se intenta otra posibilidad. Este método se llama *backtracking* o *búsqueda con retroceso*.)

- 3.3.11. Un arreglo *maze* de  $10 \times 10$ , de ceros y unos, representa un laberinto en el que un viajero debe encontrar un camino de *maze[0][0]* a *maze[9][9]*. El viajero puede moverse de un cuadrado a otro adyacente en la misma columna o renglón, pero no puede saltar ningún cuadrado ni moverse en diagonal. Además, no puede moverse a ningún cuadrado que contenga un 1. *maze[0][0]* y *maze[9][9]* contienen ceros. Escriba una rutina que acepte tal laberinto (*maze*) e imprima un mensaje de que no existe camino a través del laberinto o una lista de posiciones representando un camino de *[0][0]* a *[9][9]*.

## 3.4. SIMULACION DE RECURSION

En esta sección se examinan con más detalle algunos de los mecanismos usados para crear la recursión a fin de poder simular estos mecanismos mediante técnicas no recursivas. Esta actividad es importante por varias razones. En primer lugar, muchos lenguajes de programación de uso común (como **FORTRAN**, **COBOL** y muchos lenguajes de máquina) no permiten programas recursivos. Problemas como el de las Torres de Hanoi y la conversión de prefija a postfija, cuyas soluciones se pueden derivar y establecer de manera muy simple mediante técnicas no recursivas, se pueden programar en esos lenguajes simulando la solución recursiva por medio de operaciones más elementales. Cuando se sabe que una solución recursiva es correcta (y por lo general es muy fácil comprobar que lo es) y se establecen técnicas para convertir una solución recursiva en una que no lo es, es posible crear una solución correcta en un lenguaje no recursivo. No es raro que un programador sea capaz de plantear una solución recursiva a un problema. La habilidad para generar una solución no recursiva de un algoritmo recursivo es indispensable cuando se usa un compilador que no acepta la recursión.

Otro motivo para examinar la creación de recursión es que permite entender sus implicaciones y algunas de sus trampas ocultas. Aunque esas trampas no existen en las definiciones matemáticas que emplean la recursión, parecen ser un acompañante inevitable de la creación en un lenguaje y en una máquina reales.

Por último, incluso en un lenguaje como C que admite la recursión, una solución recursiva es con frecuencia más costosa que una que no lo es, en términos tanto de tiempo y como de espacio. Dicho costo es, por lo general, un pequeño precio que hay que pagar por la simplicidad lógica y la auto-documentación de la solución recursiva. Sin embargo, en un programa de producción (como un compilador, por

ejemplo) que puede correr miles de veces, el costo de recurrencia es un carga pesada para los recursos limitados del sistema.

Por lo tanto, se puede diseñar un programa para incorporar una solución recursiva que reduzca el costo de diseño y certificación, y luego convertir con cuidado dicho programa en una versión no recursiva que se pueda usar diario. Como se verá, en la ejecución de tal conversión muchas veces es posible identificar partes de la creación recursiva que son superfluas en una aplicación particular y, en consecuencia, reducir la cantidad de trabajo que debe desempeñar el programa.

Antes de examinar las acciones de una rutina recursiva, habrá que regresar atrás y examinar la acción de una rutina no recursiva. Más adelante se podrá ver qué mecanismo debe agregarse para sustentar la recursión. Antes de proceder, se debe adoptar la siguiente convención: Suponga que se tiene la instrucción

```
rout(x);
```

donde *rout* queda definida como una función por medio del encabezamiento:

```
rout(a)
```

*x* es asignada como un *argumento* (de la función de llamada) y *a* como un *parámetro* (de la función llamada).

¿Qué pasa cuando se llama una función? La acción de llamar una función se puede dividir en tres partes:

1. Transferencia de argumentos .
2. Ubicación e inicialización de variables locales
3. Transferencia de control a la función

A continuación se examinará cada uno de los pasos.

1. Transferencia de argumentos: Para un parámetro en C, se hace localmente una copia del argumento dentro de la función y todos los cambios al parámetro se transfieren a esa copia local. El efecto de este esquema es que el argumento original de entrada no debe alterarse. En este método, se asigna memoria para el argumento dentro del área de datos de la función.

2. Ubicación e inicialización de variables locales: Luego de pasar los argumentos, se ubican las variables locales. Estas incluyen todas aquellas que son declaradas de manera directa en la función, así como las provisionales que deben crearse durante el curso de la ejecución. Por ejemplo, al evaluar la expresión

$$x + y + z$$

debe reservarse una localidad de memoria para guardar el valor de *x* + *y* de manera que *z* pueda agregarse a éste. Se debe reservar otra localidad de memoria para guardar el valor de toda la expresión una vez que ha sido evaluada. Tales localidades se

conocen como *temporales*, debido a que se necesitan sólo de manera temporal durante el curso de la ejecución. De manera análoga, en una instrucción como

```
x = fact(n)
```

debe reservarse una localidad temporal para guardar el valor de *fact(n)* antes de que éste sea asignado a *x*.

3. Transferencia de control a la función. En este punto, no puede transferirse aún el control a la función, puesto que no se han tomado las provisiones para salvar la *dirección de regreso*. Cuando se le da el control a una función, ésta tiene que regresar finalmente el control a la rutina de llamada por medio de “un salto”. Sin embargo, no puede ejecutar dicho salto a menos que conozca la localidad a la que debe regresar. Como esta localidad está dentro de la rutina de llamada y no en la propia función, la única manera en que la función puede conocer esa dirección es cuando la ha pasado como argumento. Esto es exactamente lo que ocurre. Además de los argumentos especificados por el programador, también hay un conjunto de argumentos implícitos que contienen la información necesaria para ejecutar la función y regresárla de manera correcta. De esos argumentos implícitos, el más importante es la dirección de regreso. La función almacena esta dirección en su propia área de datos. Cuando está lista para restablecer el control al programa de llamada, la función recupera la dirección de regreso y “salta” hacia la localización indicada.

Una vez que se han pasado los argumentos y la dirección de regreso, es posible transferir el control a la función, debido a que ya se realizó todo lo requerido para asegurar que la función opere sobre los datos apropiados y luego regresar a salvo a la rutina de llamada.

### Regreso desde una función

Cuando una función regresa el control, se ejecutan tres acciones. Primero, se recupera la dirección de retorno y se almacena en una localidad segura. Luego se libera el área de datos de la función. Esta área de datos contiene todas las variables locales (incluyendo las copias locales de argumentos), las temporales y la dirección de regreso. Por último, efectúa un “salto” hacia la dirección de regreso salvada con anterioridad. Esto restablece el control a la rutina de llamada en el punto siguiente a la instrucción que inició la llamada. Además, si la función dio como resultado un valor, éste se coloca en una localidad segura, desde la que puede recuperarlo el programa de llamada. Por lo general esta localidad es un registro de hardware reservado para este propósito.

Supóngase que el procedimiento principal llamó a una función *b*, la que llamó a *c*, la que llamó a *d*. Esto se ilustra en la figura 3.4.1a, donde se indica que el control reside en ese momento en alguna parte de *d*. Dentro de cada función, se separa una localidad para la dirección de regreso. Así, el área de dirección de regreso de *d* contiene la dirección de la instrucción en *c* que sigue de inmediato a la llamada a *d*. La figura 3.4.1b muestra la situación que sigue inmediatamente al regreso de *d* a *c*. Se recupera la dirección de regreso dentro de *d* y se transfiere el control a dicha dirección.

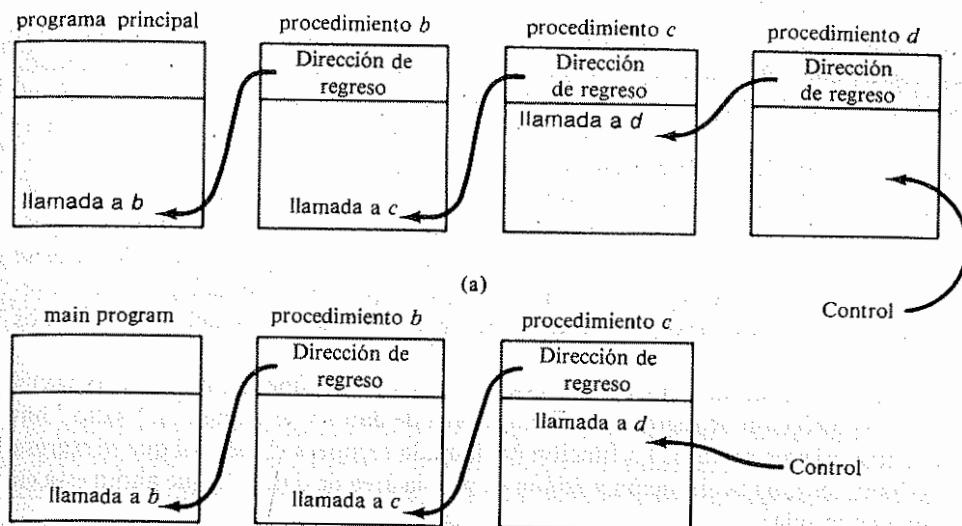


Figura 3.4.1 Serie de procedimientos que se llaman entre sí.

Como pudo observarse, la cadena de direcciones de regreso forma una pila; es decir, la dirección de regreso más reciente que se agregará a la cadena será la primera en ser eliminada. Desde cualquier punto, sólo se puede accesar la dirección de regreso desde el interior de la función que se está ejecutando en ese momento, la cual representa el tope de la pila. Cuando se saca un elemento de la pila (esto es, cuando la función regresa), aparece un nuevo tope en la rutina de llamada. La llamada a una función produce el efecto de poner un elemento dentro de la pila, y el regreso, el de quitar un elemento.

### Ejecución de funciones recursivas

¿Qué debe agregarse a esta descripción en el caso de una función recursiva? La respuesta es sorprendentemente escueta. Cada vez que una función recursiva se llama a sí misma, se asigna toda una nueva área de datos para la llamada particular. Al igual que antes, esta área de datos contiene todos los parámetros, las variables locales y las temporales, así como la dirección de regreso. Hay que recordar que en la recursión un área de datos no está relacionada con una sola función sino con una llamada particular a dicha función. Cada llamada ocasiona la reservación de una nueva área de datos, y cada referencia a un elemento del área de datos de la función se efectúa sobre el área de datos de la llamada más reciente. De manera análoga, cada regreso ocasiona la liberación del área de datos vigente, y el área de datos asignada que precede de inmediato a la vigente se hace vigente. Por supuesto que este comportamiento sugiere el uso de una pila.

En la sección 3.1.2, donde se describió la acción de la función factorial recursiva, se usó un conjunto de pilas para representar las asignaciones sucesivas para cada

una de las variables y parámetros locales. Se puede pensar en esas pilas como pilas separadas, una para cada variable local. Otra posibilidad, más cercana a la realidad, es pensar en todas ellas como una pila simple y enorme. Cada elemento de esta enorme pila es toda una área de datos que contiene partes secundarias para representar las variables locales independientes o parámetros.

Cada vez que se llama a la rutina recursiva, se asigna una nueva área de datos. Los parámetros dentro de dicha área se inicializan para asignar los valores de sus argumentos correspondientes. La dirección de regreso dentro de esta área de datos se inicializa como la dirección de la instrucción que sigue a la de llamada. Cualquier referencia a variables locales o parámetros se efectúa a través del área de datos vigente.

Cuando la rutina recursiva regresa, se almacena el valor resultante (si lo hay) y se salva la dirección de regreso, se libera el área de datos y se ejecuta un "salto" hacia la dirección de regreso. La función de llamada recupera el valor al que se regresa (si lo hay), termina la ejecución y asigna su propia área de datos, la que ahora está en el tope de la pila.

Examíñese ahora la forma en que pueden simularse las acciones de una función recursiva. Para ello, es necesario que una pila para las áreas de datos esté definida por

```
#define MAXSTACK 50;
struct stack {
 int top;
 struct dataarea item[MAXSTACK];
};
```

Por sí misma, *dataarea* es una estructura que contiene los diversos elementos que existen en un área de datos y se la debe definir para que contenga los campos requeridos de la función particular que se está simulando.

### Simulación de factorial

Veamos un ejemplo específico: la función factorial. A continuación se presenta el programa para esta función, el que incluye de manera explícita las variables temporales y omite la prueba para las entradas negativas:

```
fact(n)
int n;
{
 int x, y;
 if (n == 0)
 return(1);
 x = n-1;
 y = fact(x);
 return(n * y);
} /* fin de fact */
```

¿Cómo puede definirse el área de datos de esta función? Esta debe incluir el parámetro *n* y las variables locales *x* y *y*. Como podrá verse, no se necesitan temporales. El área de datos también debe incluir una dirección de regreso. En este caso, hay dos puntos posibles a los que se podría regresar: a la asignación de *fact(x)* a *y* y al programa principal que llama a *fact*. Supóngase que se tienen dos etiquetas y que la etiqueta *label2* es para la parte del código,

```
label2: y = result;
```

```
label1: return(result);
```

No es más que una convención el hecho de que la variable *result* contenga el valor que debe regresar con una llamada a la función *fact*. La dirección de regreso se almacenará como un entero *i* (que puede ser 1 o 2). Para efectuar un regreso desde una llamada recursiva se ejecuta la instrucción:

```
switch(i) {
 case 1: goto label1;
 case 2: goto label2;
} /* fin de case */
```

Así, si *i* == 1, se ejecuta un regreso al programa principal que llamó a *fact*, si *i* == 2, se simula un retorno a la asignación del valor resultante a la variable *y* en la ejecución previa de *fact*.

La pila del área de datos para este ejemplo se puede definir como sigue:

```
#define MAXSTACK 50
struct dataarea {
 int param; /* Parámetro simulado */
 int x; /* Variable local */
 long int y; /* Resultado simulado */
 short int retaddr; /* Dirección de regreso */
};
struct stack {
 int top;
 struct dataarea item[MAXSTACK];
};
```

El campo del área de datos que contiene el parámetro simulado se llama *param*, y no *n*, para evitar confusiones con el parámetro *n* transferido a la función de simulación. También se declara un área de datos vigente para guardar los valores de las variables en la llamada "vigente" simulada a la función recursiva. La declaración es:

```
struct dataarea currarea;
```

Además, se declara una variable simple *result* mediante:

```
long int result;
```

Esta variable se usa para comunicar al usuario el valor resultante de *fact* de una llamada recursiva de *fact* y de *fact* a la función de llamada externa. Como los elementos de la pila de las áreas de datos son estructuras y como en muchas versiones de C una función no puede dar como resultado una estructura, no se usa la función *pop* para eliminar un área de datos de la *pila*. En lugar de ello, se escribe una función *popsub* definida por:

```
popsub(ps, parea)
 struct stack *ps;
 struct dataarea *parea;
```

La llamada *popsub(&s, &area)* elimina elementos de la pila y asigna área al elemento eliminado. Los detalles se quedan a manera de ejercicio.

Un regreso de *fact* se simula mediante el código

```
result = value to be returned;
i = currarea.retaddr;
popsub(&s, &currarea);
switch(i) {
 case 1: goto label1;
 case 2: goto label2;
} /* fin de switch */
```

Es posible simular una llamada recursiva a *fact* poniendo el área de datos vigente en la pila, reinicializando las variables *currarea.param* y *currarea.retaddr* como el parámetro y la dirección de regreso de la llamada en cuestión respectivamente, y transfiriendo después el control al principio de la rutina simulada. Hay que recordar que *currarea.x* guarda el valor de *n* — 1, el que será el nuevo parámetro. También debe recordarse que en una llamada recursiva se desea finalmente regresar a label 2. El código para hacerlo sería:

```
push(&s, &currarea);
currarea.param = currarea.x;
currarea.retaddr = 2;
goto start; /* start es la etiqueta de inicio de */
 /* la rutina simulada . */

*/
```

Por supuesto, las rutinas *popsub* y *push* deben escribirse de manera que permitan eliminar y poner estructuras enteras de tipo *dataarea* en lugar de variables simples. Otra imposición de la implantación de pilas por medio de arreglos es que la variable *currarea.y* debe inicializarse con algún valor, pues de lo contrario resultará un error en la rutina *push* al asignar *currarea.y* al campo correspondiente en el área superior de datos cuando comienza el programa.

Al iniciar la simulación, se debe inicializar el área vigente de manera que *currarea.param* sea igual a *n* y *currarea.retaddr* a 1 (para indicar un regreso a la rutina de llamada). Asimismo, se debe insertar un área de datos de relleno en la pila para que no ocurra un subdesborde al ejecutar *popsub* en el retorno a la rutina principal. Esta área de datos ficticia debe inicializarse también para que no provoque un error en la rutina *push* (ver la última oración del párrafo anterior). Así, la versión simulada de la rutina recursiva *fact* es la siguiente:

```
struct dataarea {
 int param;
 int x;
 long int y;
 short int retaddr;
};

struct stack {
 int top;
 struct dataarea item[MAXSTACK];
};

simfact(n)
int n;
{
 struct dataarea currarea;
 struct stack s;
 short i;
 long int result;
 s.top = -1;
 /* inicialización de un área de datos simulada */
 currarea.param = 0;
 currarea.x = 0;
 currarea.y = 0;
 currarea.retaddr = 0;
 /* colocar el área de datos simulada en el área */
 push (&s, &currarea);
 /* Asignar al parámetro y a la dirección de retorno */
 /* del área de datos actual sus valores apropiados */
 currarea.param = n;
 currarea.retaddr = 1;
 start: /* Este es el punto de inicio de la rutina */
 /* factorial simulada. */
 if (currarea.param == 0) { /* simulación de retorno */
 result = 1; /* el resultado es 1 en el caso de la base */
 i = currarea.retaddr; /* se regresa a la dirección de donde */
 popsub(&s, &currarea);
 switch(i) {
 case 1: goto label1;
 case 2: goto label2;
 } /* fin de switch */
 }
}
```

```

} /* fin de if */
currarea.x = currarea.param - 1;
/* ... simulación de la llamada recursiva a fact */
push(&s, &currarea);
currarea.param = currarea.x;
currarea.retaddr = 2;
goto start;

label2: /* Este es el punto al cual se regresa
 de la llamada recursiva. Asignar a
 currarea.y el valor regresado.
currarea.y = result;
/* simulación de return (n * y)
result = currarea.param * currarea.y;
i = currarea.retaddr;
popsub(&s, &currarea);
switch(i) {
 case 1: goto label1;
 case 2: goto label2;
} /* fin de switch */
label1: /* En este punto se regresa a la rutina main.
return(result);
} /* fin de simfact */

```

Es necesario seguir la ejecución del programa anterior para  $n = 5$  y tener la seguridad de que se ha comprendido qué hace el programa y cómo lo hace.

Adviértase que no se reservó espacio en el área de datos para valores temporales, pues no es necesario almacenarlos para uso posterior. La localidad temporal que guarda el valor de  $n * y$  en la rutina recursiva original se simuló mediante el valor temporal de `currarea.param * currarea.y` en la rutina de simulación. En general, no es éste el caso. Por ejemplo, si una función recursiva *funct* contiene una instrucción

$$x = a * \text{funct}(b) + c * \text{funct}(d);$$

para el valor temporal de  $a * \text{funct}(b)$  debe salvarse durante la llamada recursiva a *funct(d)*. Sin embargo, en el ejemplo de la función factorial no se requiere apilar los valores temporales.

### Perfeccionamiento de la rutina simulada

El análisis anterior conduce de manera natural a preguntarse si en verdad es necesario que todas las variables locales estén en la pila. Se debe salvar una variable en la pila sólo si su valor en el punto de inicio de una llamada recursiva debe usarse otra vez después del regreso de esta llamada. A continuación se examina si las variables  $n$ ,  $x$  y  $y$  cumplen este requisito. Es evidente que  $n$  no debe apilarse. En la instrucción

$$y = n * \text{fact}(x);$$

el antiguo valor de  $n$  debe usarse en la multiplicación después del regreso de la llamada recursiva a *fact*. Sin embargo, no ocurre lo mismo con  $x$  y  $y$ . En realidad, el valor de  $y$  no está definido aún en el momento de la llamada recursiva, por lo que es evidente que no es necesario colócarlo en la pila. De manera análoga, aunque  $x$  si se define al hacer la llamada, no se vuelve a usar de nuevo después de la misma: así que, ¿por qué razón hay que salvarlo?

Este punto puede ilustrarse de manera más precisa por medio del siguiente razonamiento. Si  $x$  y  $y$  no se declaran dentro de la función recursiva *fact*, sino como variables globales, la rutina funcionaría bien. Así, la acción automática de apilamiento y desapilamiento ejecutada por la recursión para las variables locales  $x$  y  $y$  resulta innecesaria.

Otra pregunta interesante sería saber si se necesita realmente la dirección de regreso en la pila. Como sólo hay una llamada recursiva textual a *fact*, existe sólo una dirección de regreso dentro de *fact*. La otra dirección de regreso es hacia la rutina principal que llamó inicialmente a *fact*. Pero, si se supone que al inicio de la simulación no se guardó un área de datos de relleno en la pila, entonces deberá colocarse en la pila un área de datos sólo cuando se simula una llamada recursiva. Si se eliminan elementos de la pila al regresar de una llamada recursiva, esta área debe eliminarse. Sin embargo, si se intenta sacar elementos de la pila durante la simulación de un regreso al procedimiento principal, ocurrirá subdesborde. Mediante el uso de *popandtest*, en lugar de *popsub*, puede verificarse si ocurrirá subdesborde; en caso positivo, hay que regresar en forma directa a la rutina exterior en lugar de hacerlo a través de una etiqueta local, lo que significa que se puede eliminar una de las direcciones de regreso. Como esto deja una sola dirección de regreso posible, es innecesario que ésta esté en la pila.

De este modo, el área de datos ha sido reducida para que contenga el parámetro único y la pila puede declararse como

```

#define MAXSTACK 50
struct stack {
 int top;
 int param[MAXSTACK];
};

```

El área de datos vigente se reduce a una variable simple declarada mediante:

```

int currparam;
Ahora el programa es compacto y comprensible.

```

```

simfact(n)
int n;
{
 struct stack s;
 short int und;
 long int result, y;
 int currparam, x;
}

```

```

 s.top = -1;
 currparam = n;
start: /* Este es el punto de inicio de la rutina factorial simulada. */
if (currparam == 0) {
 /* simulación de return (1) */
 result = 1;
 popandtest(&s, &currparam, &und);
 switch(und) {
 case FALSE: goto label2;
 case TRUE: goto label1;
 } /* fin de switch */
} /* fin de if */
/* currparam != 0 */
x = currparam - 1;
/* simulación de la llamada recursiva a fact */
push(&s, currparam); /* se hace en una sola operación de push */
currparam = x; /* se olviven otras posibilidades que no sirven */
goto start;
label2: /* Este es el punto al que se regresa de la llamada recursiva. Asignar a y el valor regresado */
y = result;
/* simulación de return (n * y); */
result = currparam * y;
popandtest(&s, &currparam, &und);
switch(und) {
 case TRUE: goto label1;
 case FALSE: goto label2;
} /* fin de switch */
label1: /* En este punto se regresa a la rutina principal. */
return(result);
} /* fin de simfact */

```

### Eliminación de gotos

Aunque este último programa es más simple que el anterior, aún está lejos de ser el ideal. Si se viera el programa sin observar su derivación, sería muy difícil identificar que su objetivo es calcular la función factorial. Las instrucciones

`goto start;`

`y`

`goto label2;`

son particularmente molestas porque interrumpen el flujo de pensamiento en el momento en que podría llegarse a entender qué está pasando. Ahora se verá si es posible transformar el programa en una versión más legible.

Varias transformaciones se manifiestan de inmediato. En primer lugar, las instrucciones

```

popandtest(&s, &currparam, &und);
switch(und) {
 case FALSE: goto label2;
 case TRUE: goto label1;
} /* fin de switch */

```

se repiten dos veces para los dos casos `currparam == 0` y `currparam != 0`. Es fácil hacer una combinación con ambas secciones.

Una observación adicional es que a las dos variables `x` y `currparam` se les asignan valores mutuos y que nunca se usan de manera simultánea; de allí que se las pueda combinar y referir como una sola variable `x`. Lo mismo es válido para las variables `result` y `y`, las que se pueden combinar y aludir como una sola variable `y`.

Luego de realizar esas transformaciones, se llega a la siguiente versión de `simfact`:

```

struct stack {
 int top;
 int param[MAXSTACK];
};

simfact(n)
int n;
{
 struct stack s;
 short int und;
 int x;
 long int y;

 s.top = -1;
 x = n;
start: /* Este es el inicio de la rutina factorial asociada */
if (x == 0)
 y = 1;
else {
 push(&s, x--);
 goto start;
} /* fin de else */
label1: popandtest(&s, &x, &und);
if (und == TRUE)
 return(y);
label2: y *= x;
 goto label1;
} /* fin de simfact */

```

Ahora se comienza a llegar a un programa legible. Adviértase que el programa consiste en dos ciclos:

1. El ciclo de toda la instrucción *if*, etiquetado con *start*. Este ciclo termina cuando *x* es igual a 0, en cuyo momento *y* se transforma en 1 y la ejecución continúa en la etiqueta *label1*.
2. El ciclo que comienza en *label1* y termina con la instrucción *goto label1*. Este ciclo termina una vez que se vacía la pila y ocurre el subdesborde, momento en el que se ejecuta un regreso.

Estos ciclos pueden sin dificultad ser transformados en ciclos *while* explícitos como sigue:

```
/* Iteración de resta */
start: while (x != 0) {
 push(&s, x--); /* mover y al finalizar el bucle se apila en s y se resta uno */
 popandtest(&s, &x, &und); /* Aplicar y multiplicar se apila en s y se resta uno */
 label1: while (und == FALSE) {
 y *= x; /* multiplicar y por x */
 popandtest (&s, &x, &und);
 } /* fin de while */
 return(y);
}
```

Examíñese con más detenimiento ambos ciclos. *x* se inicia con el valor del parámetro de entrada *n* y se reduce en 1 cada vez que se répite el ciclo de sustracción. Cada vez que se le asigna un nuevo valor a *x*, se salva el valor antiguo de *x* en la pila. Esto prosigue hasta que *x* es igual a 0. Así, después de ejecutar el primer ciclo, la pila contiene los enteros del 1 al *n*, de la parte superior a la inferior.

Lo único que hace el ciclo de multiplicación es extraer de la pila cada uno de estos valores y poner en *y* el resultado de la multiplicación de dichos valores con el valor antiguo de *y*. Si ya se sabe lo que contiene la pila al principio del ciclo de multiplicación ¿qué caso tiene sacar sus elementos? Dichos valores pueden usarse en forma directa. Se puede eliminar la pila y todo el primer ciclo y remplazar el ciclo de multiplicación por uno que multiplique cada uno de los enteros del 1 al *n* por *y*, paso a paso. El programa resultante es

```
simfact(n)
int n;
{
 int x;
 long int y;

 for (y=x=1; x <= n; x++)
 y *= x;
 return(y);
} /* fin de simfact */
```

Pero este programa es la implantación directa en C de la versión iterativa de la función factorial tal como se presentó en la sección 3.1. El único cambio es que *x* varía de 1 a *n* en lugar de hacerlo de *n* a 1.

### Simulación de las Torres de Hanoi

Ya se demostró que las transformaciones sucesivas de una simulación no recursiva de una rutina recursiva pueden conducir a un programa más simple para resolver un problema. Ahora se verá un ejemplo más complejo de recursión, el problema de las Torres de Hanoi, presentado en la sección 3.3. Se simulará la recursión del problema y se intentará simplificar la simulación para producir una solución no recursiva. En seguida se presenta de nuevo el programa recursivo de la sección 3.3.

```
towers(n, frompeg, topeg, auxpeg)
int n;
char auxpeg, frompeg, topeg;
{
 /* Si es sólo un disco, mover y regresar */
 if (n == 1) {
 printf("\n%s%c%s%c%", "mover disco 1 del poste", frompeg,
 "al poste", topeg);
 return;
 } /* fin de if */
 /* Mover los n-1 discos de arriba de A a B, usando
 como auxiliar */
 towers(n-1, frompeg, auxpeg, topeg);
 /* Move remaining disk from A to C. */
 printf("\n%s%d%s%c%s%c%", "mover disco", n, "del poste",
 frompeg, "al poste", topeg);
 /* Mover n-1 discos de B hacia C empleando a
 A como auxiliar */
 towers(n-1, auxpeg, topeg, frompeg);
} /* fin de towers */
```

Antes de proseguir, hay que asegurarse de qué se ha entendido el problema y la solución recursiva. En caso contrario, vuélvase a leer la sección 3.3.

En esta función, hay cuatro parámetros, cada uno de los cuales está sujeto a cambios en cada llamada recursiva. En consecuencia, el área de datos debe contener elementos que representen a los cuatro. No hay variables locales. Hay un solo valor temporal que se necesita para guardar el valor de *n* — 1, pero ésta se puede representar por un valor temporal similar en el programa de simulación y no tiene que estar apilada. Hay tres puntos posibles a los que regresa la función en varias llamadas: el programa de llamada y los dos puntos que siguen a las llamadas recursivas. Por lo tanto, se necesitan cuatro etiquetas:

```
start:
label1:
label2:
label3:
```

La dirección de regreso se codifica como un entero (1, 2 o 3) dentro de cada área de datos.

Considérese la siguiente simulación no recursiva de *towers*:

```

struct dataarea {
 int nparam;
 char fromparam;
 char topparam;
 char auxparam;
 short int retaddr;
};

struct stack {
 int top;
 struct dataarea item[MAXSTACK];
};

simtowers(n, frompeg, topeg, auxpeg)
int n;
char auxpeg, frompeg, topeg;
{
 struct stack s;
 struct dataarea currarea;
 char temp;
 short int i;

 s.top = -1;
 currarea.nparam = 0;
 currarea.fromparam = ' ';
 currarea.topparam = ' ';
 currarea.auxparam = ' ';
 currarea.retaddr = 0;
 /* Colocar en la pila un área de datos simulados. */
 push(&s, &currarea);
 /* Asignar parámetros y direcciones de regreso
 del área de datos actual a sus valores apropiados. */
 currarea.nparam = n;
 currarea.fromparam = frompeg;
 currarea.topparam = topeg;
 currarea.auxparam = auxpeg;
 currarea.retaddr = 1;
 start: /* Este es el inicio de la rutina simulada */
 if (currarea.nparam == 1) {
 printf("/n%ss%c%s%c", "mover disco 1 del poste",
 currarea.fromparam, "al poste", currarea.topparam);
 i = currarea.retaddr;
 pop(&s, &currarea);
 switch(i) {
 case 1: goto label1;
 case 2: goto label2;
 }
 }
}

```

```

 case 3: goto label3;
} /* fin de switch */
} /* fin de if */
/* Esta es la primera llamada recursiva . */
push(&s, &currarea);
--currarea.nparam;
temp = currarea.auxparam;
currarea.auxparam = currarea.topparam;
currarea.topparam = temp;
currarea.retaddr = 2;
goto start;
label2: /* Se regresa a este punto desde la primera
 llamada recursiva . */
printf("/n%ss%d%s%c%s%c", "mover disco",
 currarea.nparam, "del poste",
 currarea.fromparam, "al poste", currarea.topparam);
/* Caso 2 Esta es la segunda llamada recursiva. */
push(&s, &currarea);
--currarea.nparam;
temp = currarea.fromparam;
currarea.fromparam = currarea.auxparam;
currarea.auxparam = temp;
currarea.retaddr = 3;
goto start;
label3: /* Se regresa a este punto desde la segunda
 llamada recursiva. */
i = currarea.retaddr;
pop(&s, &currarea);
switch(i) {
 case 1: goto label1;
 case 2: goto label2;
 case 3: goto label3;
} /* fin de switch */
label1: return;
} /* fin de simtowers */

```

Ahora se simplificará el programa. En primer lugar, debe observarse que se usan tres etiquetas para indicar direcciones de regreso: una para cada una de las dos llamadas recursivas y otra para el regreso al programa principal. Sin embargo, el regreso al programa principal puede señalarse por un subdesborde en la pila, de la misma forma que en la segunda versión de *simfact*. Esto deja dos etiquetas de regreso. Si pudiera eliminarse otra más, sería innecesario guardar en la pila la dirección de regreso, ya que sólo restaría un punto al que se podría transferir el control si se eliminan los elementos de la pila con éxito. Ahora dirigamos nuestra atención a la segunda llamada recursiva y a la instrucción:

*towers(n-1, auxpeg, topeg, frompeg);*

Las acciones que ocurren en la simulación de esta llamada son las siguientes:

1. Se coloca el área de datos vigente  $a1$  dentro de la pila.
2. En la nueva área de datos vigente  $a2$ , se asignan los valores respectivos  $n - 1$ ,  $auxpeg$ ,  $topeg$  y  $frompeg$  a los parámetros.
3. En el área de datos vigente  $a2$ , se fija la etiqueta de retorno a la dirección de la instrucción que sigue de inmediato a la llamada.
4. Se salta hacia el principio de la rutina simulada.

Después de completar la rutina simulada, ésta queda lista para regresar. Las siguientes acciones se llevan a efecto:

5. Se salva la etiqueta de regreso,  $l$ , del área de datos vigente  $a2$ .
6. Se eliminan de la pila y se fija el área de datos vigente como el área de datos eliminada de la pila,  $a1$ .
7. Se transfiere el control a  $l$ .

Sin embargo,  $l$  es la etiqueta del final del bloque del programa ya que la segunda llamada a *towers* aparece en la última instrucción de la función. Por lo tanto, el siguiente paso es volver a eliminar elementos de la pila y regresar. No se volverá a hacer uso de la información del área de datos vigente  $a1$ , ya que ésta es destruida en la eliminación de los elementos en la pila tan pronto como se vuelve a almacenar. Puesto que no hay razón para volver a usar esta área de datos, tampoco hay razón para salvarla en la pila durante la simulación de la llamada. Los datos se deben salvar en la pila sólo si se van a usar otra vez. En consecuencia, la segunda llamada a *towers* puede simularse en forma simple mediante:

1. El cambio de los parámetros en el área de datos vigente a sus valores respectivos.
2. El "salto" al principio de la rutina simulada.

Cuando la rutina simulada regresa puede hacerlo en forma directa a la rutina que llamó a la versión vigente. No hay razón para ejecutar un regreso a la versión vigente, sólo para regresar de inmediato a la versión previa. Por lo tanto, se elimina la necesidad de guardar en la pila la dirección de regreso al simular la llamada externa (ya que se puede señalar mediante subdesborde) y simular la segunda llamada recursiva (ya que no hay necesidad de salvar y volver a almacenar el área de datos de la rutina de llamada en este momento). La única dirección de regreso que resta es la que sigue a la primera llamada recursiva.

Ya que sólo queda una dirección de regreso posible, no tiene caso guardarla en la pila para que se vuelva a insertar y eliminar con el resto de los datos. Siempre que se eliminan elementos de la pila con éxito, hay una sola dirección hacia la que se puede ejecutar un "salto": la instrucción que sigue a la primera llamada. Si se detecta subdesborde, la rutina regresa a la función de llamada. Como los nuevos valores de las variables del área de datos vigente se obtendrán a partir de los datos antiguos

del área de datos vigente, es necesario declarar una variable adicional, *temp*, de manera que los valores sean intercambiables.

A continuación se presenta una revisión de la no recursiva de *towers*:

```
struct dataarea {
 int nparam;
 char fromparam;
 char toparam;
 char auxparam;
};

struct stack {
 int top;
 struct dataarea item[MAXSTACK];
};

simtowers(n, frompeg, topeg, auxpeg)
int n;
char frompeg, topeg, auxpeg;
{
 struct stack s;
 struct dataarea currarea;
 short int und;
 char temp;

 s.top = -1;
 currarea.nparam = n;
 currarea.fromparam = frompeg;
 currarea.toparam = topeg;
 currarea.auxparam = auxpeg;
 start: /* Este es el inicio de la rutina simulada. */
 if (currarea.nparam == 1) {
 printf("\n%s%c%s%c%", "mover disco 1 del poste",
 currarea.frompeg, "al poste", currarea.toparam);
 /* simular el regreso */
 popandtest(&s, &currarea, &und);
 if (und == TRUE)
 return;
 goto retaddr;
 } /* fin de if */
 /* simular la primera llamada recursiva */
 push(&s, &currarea);
 --currarea.nparam;
 temp = currarea.toparam;
 currarea.toparam = currarea.auxparam;
 currarea.auxparam = temp;
 goto start;
 retaddr: /* Punto de regreso desde la primera
 llamada recursiva */
 printf("\n%s%d%s%c%s%c", "mover disco",
 currarea.nparam, "del poste", currarea.fromparam,
 currarea.nparam, "al poste", currarea.toparam);
 /* simulación de la segunda llamada recursiva */
}
```

```

--currarea.nparam;
temp = currarea.fromparam;
currarea.fromparam = currarea.auxparam;
currarea.auxparam = temp;
goto start;
} /* fin de simtowers */

```

Al examinar la estructura del programa, se observa que éste puede reorganizarse con facilidad en un formato más simple. Para ello, se comienza desde la etiqueta *start*.

```

while (TRUE) {
 while (currarea.nparam != 1) { /* JUEGO */
 push(&s, &currarea);
 --currarea.nparam;
 temp = currarea.toparam;
 currarea.toparam = currarea.auxparam;
 currarea.auxparam = temp;
 } /* fin de while */
 printf("\n%s%c%s%c", mover 1 del poste",
 currarea.fromparam, ' ', "al poste", currarea.toparam);
 popandtest(&s, &currarea, &und);
 if (und == TRUE)
 return;
 printf("\n%s%d%s%c%s%c", "mover disco", currarea.nparam,
 "del poste", currarea.fromparam, "al poste",
 currarea.toparam);
 --currarea.nparam;
 temp = currarea.fromparam;
 currarea.fromparam = currarea.auxparam;
 currarea.auxparam = temp;
} /* fin de while */

```

Hay que seguir la acción de esta problema y ver como refleja la acción de la versión recursiva original.

## EJERCICIOS

- 3.4.1. Escriba una simulación no recursiva de las funciones *convert* y *find* presentadas en la sección 3.3.
- 3.4.2. Escriba una simulación no recursiva del procedimiento recursivo para la búsqueda binaria y transfórmela en un procedimiento iterativo.
- 3.4.3. Escriba una simulación no recursiva de *fib*. ¿Es posible transformar ésta en un método iterativo?
- 3.4.4. Escriba simulaciones no recursivas de las rutinas recursivas de las secciones 3.2 y 3.3 y de los ejercicios de dichas secciones.

- 3.4.5. Muestre que cualquier solución del problema de las Torres de Hanoi que use un número mínimo de movimientos debe satisfacer las condiciones que se enumeran en seguida. Use estos hechos para desarrollar un algoritmo iterativo directo para las Torres de Hanoi. Implemente el algoritmo como un programa en C.
- a. El primer movimiento implica el movimiento del disco más pequeño.
  - b. Una solución que use los movimientos mínimos consiste en mover de manera alterna el disco más pequeño y uno mayor que éste.
  - c. En cualquier punto, sólo hay un posible movimiento que implique un disco que no es el más pequeño.
  - d. Defina la dirección cíclica de *frompeg* a *topeg* a *auxpeg* a *frompeg* en el sentido de las manecillas del reloj y en la dirección opuesta (de *frompeg* a *auxpeg* a *topeg* a *frompeg*). Supóngase que una solución con movimientos mínimos mueve siempre el disco más pequeño en una dirección a fin de mover una torre de  $k$ -discos de *frompeg* a *topeg*. Demuestre que una solución con movimientos mínimos para mover una torre de  $(k+1)$ -discos de *frompeg* a *topeg* movería siempre el disco más pequeño en la dirección contraria. Como la solución para un disco mueve el disco más pequeño en el sentido de las manecillas del reloj (un movimiento simple de *frompeg* a *topeg*), esto significa que el disco más pequeño se mueve siempre en el sentido de las manecillas de reloj cuando el número de discos es impar, y en la dirección contraria cuando el número es par.
  - e. La solución se completa una vez que todos los discos quedan en un solo poste.

- 3.4.6. Convierta el siguiente programa con esquema recursivo en una versión iterativa que no use pila.  $f(n)$  es una función que regresa *TRUE* o *FALSE* de acuerdo con el valor de  $n$ , y  $g(n)$  regresa un valor del mismo tipo que  $n$  (sin modificar  $n$ ).

```

rec(n)
int n;
{
 if (f(n) == FALSE) {
 /* Cualquier grupo de instrucciones en C */
 /* que no modifiquen el valor de n */
 rec(g(n));
 } /* fin de if */
} /* fin de rec */

```

Generalizar el resultado al caso en que *rec* regresa un valor.

- 3.4.7. Sea  $f(n)$  una función, y sean  $g(n)$  y  $h(n)$  funciones que regresan un valor del mismo tipo que  $n$  sin modificarlo. Represente con (*stmts*) cualquier grupo de instrucciones en C que no modifique el valor de  $n$ . Demuestre que el esquema de programa recursivo *rec* es equivalente al esquema iterativo *iter*:

```

rec(n)
int n;
{
 if (f(n) == FALSE) {
 (stmts);
 rec(g(n));
 rec(h(n));
 } /* fin de if */
}

```

```

 } /* fin de rec */

 struct stack {
 int top;
 int nvalues[MAXSTACK];
 };

 iter(n)
 int n;
 {
 struct stack s;
 if (s.top == -1) push(&s, n);
 while (empty(&s) == FALSE) {
 n = pop(&s);
 if (f(n) == FALSE) {
 (stmts);
 push(&s, h(n));
 push(&s, g(n));
 }
 }
 }

```

Demuestre que las instrucciones *if* en *iter* pueden ser remplazadas por el siguiente ciclo:

```

while (f(n) == FALSE) {
 (stmts);
 push(&s, h(n));
 n = g(n);
}

```

### 3.5. EFICIENCIA DE LA RECURSION

En general, una versión no recursiva de un programa se ejecutará con mayor eficacia en cuanto a tiempo y espacio que una recursiva. Esto ocurre porque en la versión no recursiva se evita la sobrecarga debida a la entrada y salida de un bloque. Como ya se observó, a menudo es posible identificar un buen número de variables locales y temporales que no se deben salvar ni volver a almacenar por medio de una pila. Con un programa no recursivo se puede eliminar esta actividad innecesaria. Sin embargo, en un procedimiento recursivo, el compilador es incapaz de identificar dichas variables y, en consecuencia, éstas son puestas y eliminadas de la pila para asegurar que no ocurre ningún problema.

Sin embargo, también se ha visto que en ocasiones, la solución recursiva es la vía más lógica y natural de resolver un problema. Es difícil que un programador pueda desarrollar la versión no recursiva de la solución al problema de las Torres de Hanoi a partir del planteamiento del problema. Se puede hacer un comentario simi-

lar en el caso del problema de convertir de prefija a postfija, donde la solución recursiva se desprende de manera directa de las definiciones. Una solución no recursiva que requiera de pilas es más difícil de desarrollar y está sujeta a más errores. Por lo tanto, existe un conflicto entre la eficacia de la máquina y la eficiencia del programador. Con el constante crecimiento de los costos de programación y la disminución de los costos de computación se ha llegado al punto en el que, en la mayoría de los casos, carece de valor el tiempo que emplea un programador para construir laboriosamente una solución no recursiva de un problema que puede resolverse de manera más natural por medio de la recursión. Por supuesto que un programador incompetente, pero sumamente diestro, puede llegar a una solución recursiva complicada para un problema simple que se puede resolver de manera directa mediante métodos no recursivos. (Ejemplo de esto es la función factorial e incluso la búsqueda binaria.) Sin embargo, cuando un programador competente identifica la solución recursiva como la más simple para resolver un problema, lo más probable que es no valga la pena invertir tiempo y esfuerzo para descubrir un método más eficaz.

No obstante, esto no siempre es así. Cuando se va a ejecutar un programa con mucha frecuencia (a menudo hay máquinas completas dedicadas a correr continuamente el mismo programa), de modo que la eficacia creciente en la velocidad de ejecución haga crecer significativamente el rendimiento efectivo total, entonces valdrá la pena invertir tiempo extra en la programación. Aún en tales casos, quizás sea mejor crear una versión no recursiva mediante la simulación y la transformación de la solución recursiva que tratar de crear ésta a partir del propio planteamiento del problema.

Para hacer esto con más eficiencia, es necesario escribir primero la rutina recursiva y después su versión simulada, incluyendo todas las pilas y las temporales. Una vez que se ha hecho esto, hay que eliminar todas las pilas y las variables superfluas. La versión final es una depuración del programa original y es, desde luego, más eficaz. Es evidente que la eliminación de las operaciones superfluas redundantes perfeccionan la eficacia del programa resultante. Sin embargo, toda transformación aplicada a un programa es una nueva abertura por la que se puede escurrir un error.

Cuando es imposible eliminar una pila de la versión no recursiva de un programa y cuando la versión recursiva no contiene parámetros o variables locales adicionales, la versión recursiva puede ser tan rápida o más rápida que la no recursiva con un buen compilador. Las Torres de Hanoi son ejemplo de un programa recursivo de este tipo. El factorial, cuya versión no recursiva no necesita pila, y el cálculo de los números de Fibonacci, que contiene una segunda llamada recursiva innecesaria (y tampoco necesita pila en la versión no recursiva), son ejemplos donde debe evitarse la recursión en una implantación práctica. En la sección 5.2 se examinará otro ejemplo de recursión eficaz (recorrido en orden de un árbol).

Otra cuestión que hay que recordar es que las llamadas explícitas a *pop*, *push* y *empty*, así como las verificaciones de desborde y subdesborde son bastante costosas. De hecho, a menudo pueden sobrepasar el costo de la sobrecarga ocasionada por la recursión. Por ello, para aumentar al máximo la eficacia real en tiempo de ejecución de una versión no recursiva, se deben remplazar estas llamadas por código en línea y

se deben eliminar las verificaciones de desborde y subdesborde cuando se sepá que se trabaja dentro de los límites del arreglo.

Las ideas y las transformaciones desarrolladas en la presentación de la función factorial y en el problema de las Torres de Hanoi se pueden aplicar a problemas más complejos cuya solución no recursiva no es tan aparente. El grado hasta el que se puede transformar una solución recursiva (real o simulada) en una solución directa depende del problema en particular y del ingenio del programador.

## EJERCICIOS

- 3.5.1. Corra la versión recursiva y la no recursiva de la función factorial de las secciones 3.2 y 3.4 y examine la cantidad de tiempo y espacio requerido cuando  $n$  aumenta de tamaño.  
3.5.2. Haga lo mismo que en el ejercicio 3.5.1, para el problema de las Torres de Hanoi.

## Colas y listas

Quando se habla de estructuras de datos, las colas suelen ser las más sencillas y más fáciles de entender. Una cola es una colección de elementos que se organizan en un orden específico. Los elementos se añaden al final de la cola y se eliminan del frente de la cola. La figura 4.1 ilustra una cola simple.

En este capítulo se presentan la cola y la cola de prioridad, dos importantes estructuras de datos que se usan a menudo para simular situaciones del mundo real. Los conceptos de pila y cola se prolongan después a una nueva estructura: la lista. Asimismo, se examinan varias formas de listas y sus operaciones asociadas y se presentan varias aplicaciones.

### 4.1. LA COLA Y SU REPRESENTACION SECUENCIAL

La *cola* es una colección ordenada de elementos de la que se pueden borrar elementos en un extremo (llamado el *frente* de la cola) o insertarlos en el otro (llamado el *final* de la cola).

La figura 4.1.a ilustra una cola que contiene los elementos *A*, *B* y *C*. *A* está en el frente de la cola y *C* en el final. En la figura 4.1.b se borró un elemento de la cola. Como los elementos sólo se pueden borrar desde el frente de la cola al eliminar *A*, *B* pasa a ser el nuevo frente. En la figura 4.1.c, cuando se insertaron los elementos *D* y *E*, fue necesario insertarlos desde el final de la cola.

Como *D* se insertó en la cola antes que *E*, será eliminada primero. El primer elemento insertado en una cola es el primero en ser eliminado. Por esta razón algunas veces las colas son denominadas como listas *fifo* (first-in, first-out [primero en entrar, primero en salir]) en oposición a la pila que es una lista *lifo* (last-in, first-out [último en entrar, primero en salir]). En el mundo real abundan ejemplos de cola. Ejemplos comunes de cola son la fila en el banco o en una parada de camiones, o un grupo de automóviles esperando en una caseta de peaje.

Hay tres operaciones rudimentarias que se pueden aplicar a una cola. La operación *insert(q, x)* inserta un elemento en el final de la cola *q*. La operación *x = remove(q)* borra un elemento del frente de la cola *q* y asigna su valor a *x*. La tercera operación, *empty(q)*, dará como resultado *false* o *true* si la cola tiene o no elementos.

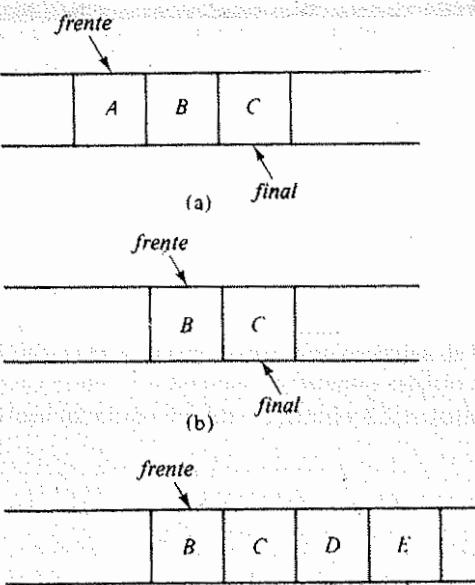


Figura 4.1.1 Una cola simple vista como un arreglo. Los elementos de la cola se almacenan en el arreglo, comenzando en la posición final. Algunas operaciones de cola están permitidas sólo si el arreglo no rebasa su capacidad máxima. Es posible obtener la cola de la figura 4.1.1 mediante la siguiente secuencia de operaciones: Supóngase que la cola está vacía al inicio.

```

insert(q, A);
insert(q, B);
insert(q, C); (Figure 4.1.1a)
x = remove(q); (Figure 4.1.1b; x is set to A)
insert(q, D);
insert(q, E); (Figure 4.1.1c)

```

La operación *insert* puede ejecutarse siempre debido a que no hay límite para el número de elementos que puede contener una cola. La operación *remove*, sin embargo, sólo se puede aplicar en caso de que la cola no esté vacía; no es posible eliminar un elemento de una cola que no contiene ninguno. El resultado del intento ilegal de eliminar un elemento de una cola vacía se conoce como subdesborde. Desde luego que la operación *empty* se puede aplicar siempre.

#### La cola como un tipo de datos abstracto

La representación de una cola como un tipo de datos abstracto es directa. *eltype* se usa para denotar el tipo de elementos de la cola y para parametrizar el tipo de cola.

```

abstract typedef <<eltype>>, QUEUE(eltype);
abstract empty(q) ==> eltype* empty(QUEUE(eltype) q);
postcondition & empty == (len(q) == 0);

```

```

abstract eltype remove(q)
QUEUE(eltype) q;
precondition empty(q) == FALSE;
postcondition remove == first(q');
q == sub(q', 1, len(q') - 1);

abstract insert(q, elt)
QUEUE(eltype) q;
eltype elt;
postcondition q == q' + <elt>;

```

#### Implantación de colas en C

¿Cómo debe representarse una cola en C? Una posibilidad es usar un arreglo para guardar los elementos de la cola y usar dos variables, *front* y *rear*, para guardar las posiciones en el arreglo del último y el primer elementos de la cola dentro del arreglo. Es posible declarar una cola de enteros *q* mediante

```

#define MAXQUEUE 100
struct queue {
 int items[MAXQUEUE];
 int front, rear;
} q;

```

Por supuesto que el uso de un arreglo para guardar una cola introduce la posibilidad de desborde si la cola rebasa el tamaño del arreglo. Si se hace a un lado por el momento la posibilidad de desborde y subdesborde, puede implantarse la operación *insert(q, x)* mediante las instrucciones:

```
q.items[++q.rear] = x;
```

y la operación *x = remove(q)* por medio de

```
x = q.items[q.front++];
```

Al comienzo, *q.rear* es igual a  $-1$  y *q.front* es igual a  $0$ . La cola está vacía siempre que *q.rear < q.front*. El número de elementos de la cola en cualquier momento es igual al valor de *q.rear - q.front + 1*.

Examíñese qué ocurriría bajo esta representación. La figura 4.1.2 muestra un arreglo de cinco elementos usado para representar una cola (es decir, *MAXQUEUE* es igual a  $5$ ). Al inicio, la cola está vacía (figura 4.1.2a). En la figura 4.1.2b se insertaron los elementos *A*, *B* y *C*. En la figura 4.1.2c se eliminaron dos elementos y en la 4.1.2d se insertaron dos nuevos elementos *D* y *E*. El valor de *q.front* es  $2$  y el de *q.rear*  $4$ , por lo que sólo hay  $4 - 2 + 1 = 3$  elementos en la cola. Como el arreglo contiene cinco elementos, debe haber espacio para que la cola se expanda sin riesgo de desborde.

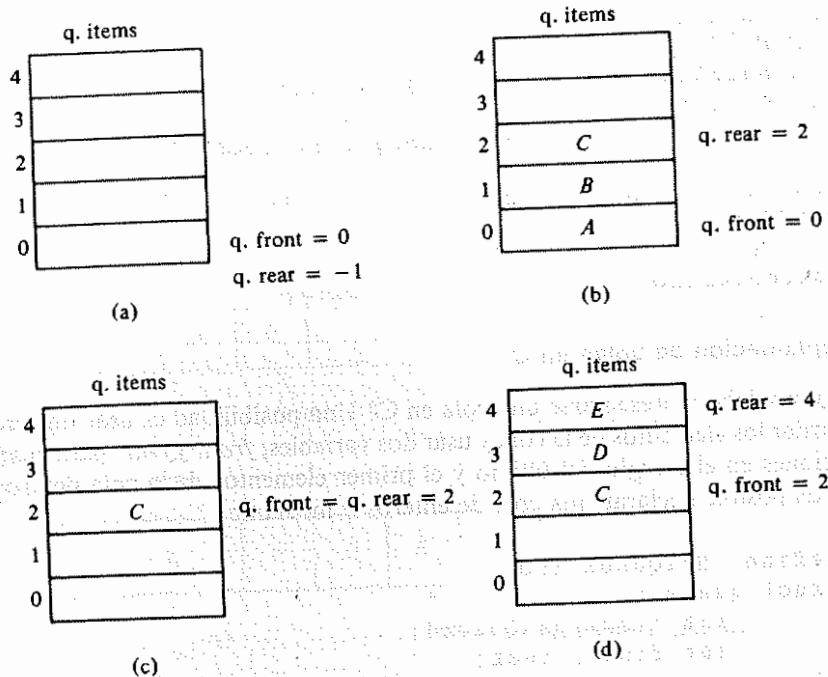


Figura 4.1.2

Sin embargo, para insertar  $F$  en la cola,  $q.\text{rear}$  debe incrementarse de 1 a 5 y  $q.\text{items}[5]$  debe colocarse al valor de  $F$ . Pero  $q.\text{items}$  es un arreglo de 5 elementos, por lo que no es posible hacer la inserción. Se puede caer en la absurda situación de que la cola esté vacía y que, aun así, se pueda insertar en ella un nuevo elemento (véase si se puede llegar a esta situación a través de una secuencia de inserciones y eliminaciones). Desde luego que la representación por medio de arreglos trazada anteriormente es inaceptable.

Una solución es modificar la operación *remove* de tal manera que cuando se elimine un elemento toda la cola se recorra al principio del arreglo. La operación  $x = \text{remove}(q)$  sería modificada entonces (y se haría a un lado una vez más la posibilidad de subdesborde) a

```

x = q.items[0];
for (i = 0; i < q.rear; i++)
 q.items[i] = q.items[i+1];
q.rear--;

```

La cola ya no necesitará un campo *front*, debido a que el elemento en la posición 0 del arreglo está siempre al frente de la misma. La cola vacía se representa por aquella en que *rear* es igual a  $-1$ .

Este método, sin embargo, es poco eficaz. Cada eliminación implica la transferencia de todos los elementos restantes de la cola. Si una cola contiene 500 o 1000 elementos, esto significa el pago de un precio muy alto. Además, la operación de eli-

minar un elemento de una cola entraña de manera lógica la manipulación de un solo elemento: aquel que esté en ese momento en el frente. La implantación de esta operación debe reflejar este hecho y no implicar un sinnúmero de operaciones extrañas (ver el ejercicio 4.1.3 para una opción más eficaz).

Otra solución consiste en observar el arreglo que guarda la cola como un círculo y no como una línea recta. Es decir, se puede imaginar que el primer elemento del arreglo (esto es, el elemento en la posición 0) sigue al último elemento del arreglo. Esto implica que aun cuando el último elemento esté ocupado, se puede insertar un nuevo valor detrás de él en el primer elemento del arreglo, siempre y cuando éste esté vacío.

A continuación se verá un ejemplo. Supóngase que una cola contiene tres elementos en las posiciones 2, 3, y 4 de un arreglo de cinco elementos. Esta es la si-

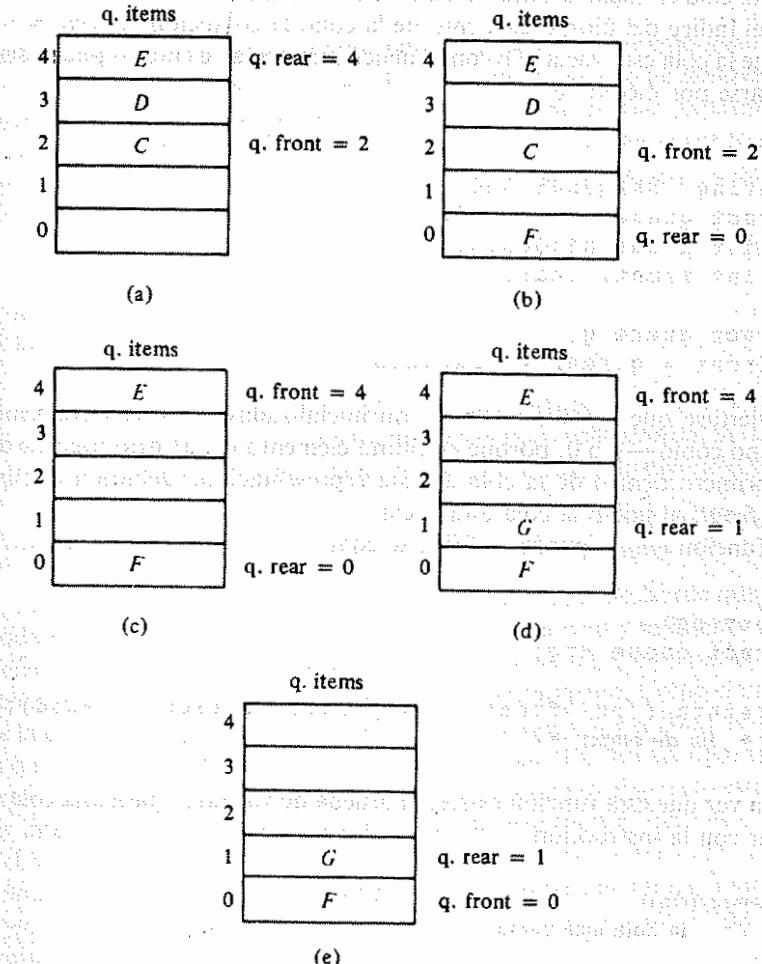


Figura 4.1.3

situación de la figura 4.1.2d, la que se repite en la figura 4.1.3a. Aunque el arreglo no está lleno, su último elemento está ocupado. Si se insertara ahora el elemento *F* en la cola, éste puede colocarse en la posición 0 del arreglo como se muestra en la figura 4.1.3b. El primer elemento de la cola está en *q.items[2]*, el que es seguido por *q.items[3]*, *q.items[4]* y *q.items[0]*. Las figuras 4.1.3c, d y e muestran el estado de la cola cuando eliminan en primer lugar los elementos *D* y *C*, luego se inserta *G* y por último se borra *E*.

Desafortunadamente, no es fácil determinar con esta representación si la cola está vacía. La condición *q.rear < q.front* deja de ser válida para la verificación de vacuidad de la cola. Las figuras 4.1.3b, c y d, por ejemplo, ilustran situaciones en las que la condición es verdadera aunque la pila no esté vacía.

Una manera de resolver este problema es establecer la regla convencional de que el valor de *q.front* es el índice del arreglo que precede de inmediato al primer elemento de la cola en lugar del índice del primer elemento mismo. Por lo tanto, como *q.rear* es el índice del último elemento de la cola, la condición *q.front == q.rear* implica que la cola está vacía. En consecuencia, una cola de enteros puede declararse e inicializarse por medio de

```
#define MAXQUEUE 100
struct queue {
 int items[MAXQUEUE];
 int front, rear;
};
struct queue q;
q.front = q.rear = MAXQUEUE-1;
```

Adviértase que *q.front* y *q.rear* son inicializadas como el último índice del arreglo y no como —1 o 0, porque el último elemento del arreglo precede de inmediato al primero dentro de la cola de esta representación. Debido a que *q.rear* es igual a *q.front*; al inicio la cola está vacía.

La función *empty* puede codificarse como

```
empty(pq)
struct queue *pq;
{
 return ((pq->front == pq->rear) ? TRUE : FALSE);
} /* fin de empty */
```

Una vez que esta función existe, la prueba de vacuidad para una cola se puede implantar con la instrucción

```
if (empty(&q))
 /* la cola está vacía */
else
 /* la cola no está vacía */
```

La operación *remove(q)* puede codificarse como

```
remove(pq)
struct queue *pq;
{
 if (empty(pq)) {
 printf("subdesborde en la cola");
 exit(1);
 } /* fin de if */
 if (pq->front == MAXQUEUE-1)
 pq->front = 0;
 else
 (pq->front)++;
 return (pq->items[pq->front]);
} /* fin de remove */
```

Adviértase que *pq* es ya un apuntador para una estructura de tipo *queue* (cola), por lo que el operador “*&*” no se usa para llamar a *empty* dentro de *remove*. Adviértase también que se debe actualizar *pq->front* antes de extraer un elemento.

Desde luego, una condición de subdesborde es significativa por lo general y sirve como señal para una nueva fase del proceso. Es probable que se desee usar una función *remvandtest*, cuyo encabezamiento es

```
remvandtest(pq, px, pund)
struct queue *pq;
int *px, *pund;
```

Si la cola no está vacía, esta rutina asigna *\*pund* a *FALSE* y *\*px* al elemento eliminado de la cola. Si la cola está vacía para que ocurra un subdesborde, la rutina asigna a *\*pund* el valor *TRUE*. El código de la rutina se queda como ejercicio para el lector.

La operación *insert*

La operación *insert* comprende una verificación de desborde que ocurre cuando todo el arreglo está ocupado por elementos de la cola y se intenta insertar otro. Por ejemplo considere la cola de la figura 4.1.4a. Hay tres elementos en la cola: *C*, *D* y *E* en *q.items[2]*, *q.items[3]* y *q.items[4]*, respectivamente. Como el último elemento de la cola ocupa el lugar *q.items[4]*, *q.rear* es igual a 4, y *q.front* es igual a 1 debido a que el primer elemento de la cola está en *q.items[2]*. En las figuras 4.1.4b y c se insertan en la cola los elementos *F* y *G*. En este momento el arreglo está lleno y cualquier intento por insertar otro elemento causaría desborde. Pero esto está indicado por el hecho de que *q.front* es igual a *q.rear*, lo que constituye precisamente la indicación de subdesborde. Bajo esta implantación parece que no hay manera de distinguir entre la cola vacía y la que está llena. Es evidente que una situación como ésta no resulta satisfactoria.

Una solución es sacrificar un elemento del arreglo y permitir que la cola pueda crecer únicamente hasta alcanzar el tamaño del arreglo menos uno. Así, cuando un

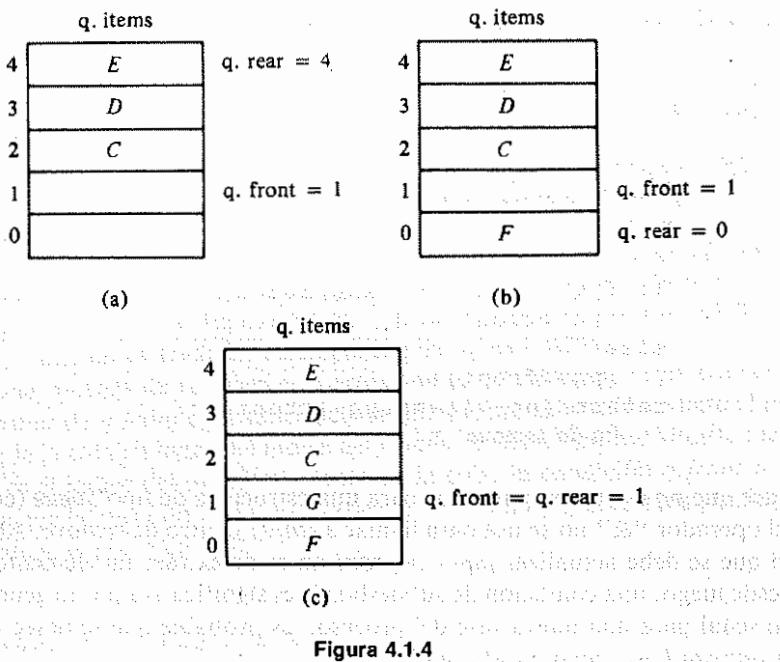


Figura 4.1.4

arreglo de 100 elementos se declara como una cola, ésta puede tener hasta 99 elementos. Si se intentara insertar el centésimo elemento, ocurriría un desbordamiento. Por lo tanto, la rutina *insert* puede escribirse como sigue:

```

insert(pq, x)
struct queue *pq;
int x;
{
 /* preparar el espacio para el nuevo elemento */
 if (pq->rear == MAXQUEUE-1)
 pq->rear = 0;
 else
 (pq->rear)++;
 /* se verifica si hay desborde */
 if (pq->rear == pq->front) {
 printf("desborde en la cola");
 exit(1);
 } /* fin de if */
 pq->items[pq->rear] = x;
 return;
} /* fin de insert */

```

La verificación de desborde o de *insert* ocurre después de ajustar *pq->rear*, mientras que en *remove* la verificación de subdesborde ocurre de inmediato al entrar la rutina, antes de que se actualice *pq->front*.

## La cola de prioridad

Tanto la pila como la cola son estructuras de datos cuyos elementos están ordenados con base en la secuencia en que se insertaron. La operación *pop* recupera el último elemento insertado, en tanto que la operación *remove* toma el primer elemento que se insertó. Si hay un orden intrínseco entre los elementos (por ejemplo, el orden alfabético o numérico), las operaciones de la pila o la cola lo ignoran.

La *cola de prioridad* es una estructura de datos en la que el ordenamiento intrínseco de los elementos determina los resultados de sus operaciones básicas. Hay dos tipos de cola de prioridad: la de prioridad ascendente y la de prioridad descendente. La *cola de prioridad ascendente* es una colección de elementos en la que pueden insertarse elementos de manera arbitraria y de la que puede eliminarse sólo el elemento menor. Si *apq* es una cola de prioridad ascendente, la operación *pqinsert(apq, x)* inserta el elemento *x* dentro de *apq* y la operación *pqmindelete(apq)* elimina el elemento mínimo de *apq* y regresa su valor.

La *cola de prioridad descendente* es similar, pero sólo permite la eliminación del elemento mayor. Las operaciones aplicables a una cola de este tipo, *dpq*, son *pqinsert(dpq, x)* y *pqmaxdelete(dpq)*. *pqinsert(dpq, x)* inserta el elemento *x* en *dpq* y es idéntica desde un punto de vista lógico a *pqinsert* en el caso de una cola de prioridad ascendente. *pqmaxdelete(dpq)* elimina el elemento máximo de *dpq* y regresa su valor.

La operación *empty(pq)* se aplica a ambos tipos de cola de prioridad y determina si la cola de prioridad está vacía. *pqmindelete* o *pqmaxdelete* sólo pueden aplicarse a colas de prioridad ocupadas (es decir, cuando *empty(pq)* es *FALSE*).

Una vez que se aplica *pqmindelete* para recuperar el elemento más pequeño de una cola de prioridad ascendente, se puede aplicar de nuevo para recuperar el siguiente elemento más pequeño, y así sucesivamente. Por esto, la operación recuperá, de manera sucesiva, elementos de una cola de prioridad en orden ascendente. (Sin embargo, cuando se inserta un elemento pequeño después de varias eliminaciones, la siguiente recuperación traerá ese pequeño elemento, que puede ser más pequeño que alguno previamente recuperado). De manera similar, *pqmaxdelete* recupera elementos de una cola de prioridad descendente en el mismo orden. Esto explica la designación de una cola de prioridad como ascendente o descendente.

Los elementos de una cola de prioridad no necesitan ser números o caracteres que se comparan de manera directa. Pueden ser estructuras complejas que están ordenadas en uno o más campos. Por ejemplo, las listas del directorio telefónico, que consisten en apellidos, nombres, direcciones y números telefónicos, están ordenadas por apellidos.

A veces el campo con el que se ordenan los elementos de una cola de prioridad ni siquiera es parte de los propios elementos; puede ser un valor externo especial, usado con el objetivo específico de ordenar la cola de prioridad. Por ejemplo, una pila puede verse como una cola de prioridad descendente cuyos elementos están ordenados por el tiempo de inserción. El último elemento insertado tiene el mayor valor de tiempo de inserción y es el único que puede recuperarse. De manera análoga, una cola puede verse como una cola de prioridad ascendente, cuyos elementos están ordenados de acuerdo

do con el tiempo de inserción. En ambos casos, el tiempo de inserción no es parte de los propios elementos, aunque se use para ordenar la cola de prioridad.

Se deja como ejercicio para el lector el desarrollo de la especificación de un ADT para una cola de prioridad. Ahora se revisarán las condiciones de implantación.

### Implantación con arreglo de una cola de prioridad

Como ya se vio, se puede implantar una cola y una pila en un arreglo de tal manera que cada eliminación o inserción implique el acceso sólo a un elemento simple del mismo. Desafortunadamente, esto no es posible en el caso de una cola de prioridad.

Supóngase que los  $n$  elementos de una cola de prioridad  $pq$  están en las posiciones 0 a  $n - 1$  de un arreglo  $pq.items$  de tamaño  $maxpq$  y que  $pq.rear$  es igual a la primera posición vacía del arreglo,  $n$ . Entonces  $pq.insert(pq, x)$  parecería aparentemente ser una operación bastante directa:

```
if (pq.rear >= maxpq) {
 printf("desborde en la cola de prioridad");
 exit(1);
} /* fin de if */
pq.items[pq.rear] = x;
pq.rear++;
```

Adviértase que con este método de inserción, los elementos de la cola de prioridad no se guardan de manera ordenada en el arreglo.

La implantación funciona bien cuando sólo hay inserciones. Supóngase, sin embargo, que se intenta la operación  $pqmindelete(pq)$  sobre una cola de prioridad ascendente. Esto produce dos consecuencias. Primera, todos los elementos del arreglo de  $pq.items[0]$  a  $pq.items[pq.rear - 1]$  deben examinarse para localizar el elemento más pequeño. Por lo tanto, una eliminación requiere accesar todos los elementos de la cola de prioridad.

Segunda, ¿cómo puede borrarse un elemento que esté en el centro del arreglo? Las eliminaciones en una pila o cola implican la remoción de un elemento de uno de los extremos y no requieren búsqueda alguna. La eliminación en una cola de prioridad, bajo esta implantación, requiere de ambas cosas: la búsqueda del elemento que debe eliminarse y la remoción de un elemento es el centro de un arreglo.

Hay varias soluciones a este problema, pero ninguna es enteramente satisfactoria:

Se puede colocar un indicador "vacío" en la posición borrada. Este indicador puede ser un valor no válido como elemento (por ejemplo,  $-1$  en una cola de prioridad de números no negativos), o cada elemento del arreglo puede contener un campo separado que indique si la posición está vacía. La inserción procede como antes, pero cuando  $pq.rear$  alcanza el valor de  $maxpq$ , los elementos del arreglo se compactan en el frente del mismo y  $pq.rear$  se reinicia como uno más que el número de elementos. Esta aproximación tiene varias desventajas. Prime-

ro, el proceso de búsqueda para localizar el elemento máximo o mínimo tiene que examinar todas las posiciones borradas del arreglo, además de los elementos reales de la cola de prioridad. Si se borraron muchos elementos pero no se ha llevado a efecto una compactación, la operación de eliminación necesitará muchos elementos más del arreglo que los que existen en la cola de prioridad. Segundo, de vez en cuando la inserción requiere accesar todas las posiciones del arreglo cuando se termina el espacio y comienza la compactación.

La operación de eliminación coloca una etiqueta de posición vacía como en la solución anterior, pero la inserción se modifica para insertar un nuevo elemento en la primera posición "vacía". La inserción implica entonces el acceso de todos los elementos del arreglo hasta alcanzar el primero que fue borrado. Esta disminución en eficacia de inserción es la desventaja principal para esta solución. Cada eliminación puede compactar el arreglo recorriendo una posición todos los elementos a partir del que se eliminó y disminuyendo después en 1 el valor de  $pq.rear$ . La inserción sigue sin cambiar. En promedio se recorre la mitad de todos los elementos de la cola de prioridad en cada eliminación, por lo que ésta resulta bastante ineficiente. Una posibilidad ligeramente mejor es recorrer todos los elementos que preceden al eliminado hacia delante o todos los que le suceden hacia atrás, dependiendo de qué grupo es menor. Esto requeriría guardar los indicadores  $front$  y  $rear$  y tratar el arreglo como una estructura circular, tal como se hizo con la cola.

- En lugar de guardar la cola de prioridad como un arreglo desordenado, hay que guardarla como un arreglo circular ordenado de la siguiente manera:

```
#define MAXPQ
struct pqueue {
 int items[MAXPQ];
 int front, rear;
};
struct pqueue pq;
```

$pq.front$  es la posición del elemento menor,  $pq.rear$  es mayor en 1 que la posición del más grande. La eliminación implica únicamente el incremento de  $pq.front$  (en el caso de una cola ascendente) o la disminución de  $pq.rear$  (en el caso de una cola descendente). Sin embargo, la inserción requiere la localización de la posición exacta del nuevo elemento para luego recorrer los elementos que le suceden o preceden (una vez más resulta útil, la técnica de corrimiento del grupo más pequeño). Este método traslada el trabajo de búsqueda y corrimiento de la operación de eliminación a la de inserción. Sin embargo, como el arreglo está ordenado, la búsqueda de la posición del nuevo elemento cuesta, en promedio, sólo la mitad de lo que costaría encontrar el elemento máximo o mínimo en un arreglo desordenado; asimismo, se puede usar la búsqueda binaria para reducir aún más el costo. También hay otras técnicas que dejan espacios en el arreglo entre los elementos de la cola de prioridad para permitir inserciones posteriores.

La implantación en C de *pqinsert*, *pqmindelete* y *pqmaxdelete* para la representación con arreglo de una cola de prioridad queda como ejercicio para el lector. La búsqueda en arreglos ordenados y desordenados se analiza posteriormente en la sección 7.1. En general, el uso de un arreglo no es un método eficaz para implantar una cola de prioridad. En la siguiente sección y en las secciones 6.3 y 7.3 se examinan implantaciones más eficaces.

## EJERCICIOS

- 4.1.1. Escriba la función *remvandtesi*(*pq*, *px*, *pund*) que hace a *\*pund* igual a *FALSE*, a *\*px* igual al elemento eliminado de una cola no vacía *\*pq* y a *\*pund* igual a *TRUE* si la cola está vacía.
- 4.1.2. ¿Qué conjunto de condiciones es necesario y suficiente para que una secuencia de operaciones *insert* y *remove* sobre una sola cola vacía deje la cola vacía sin causar subdesborde? ¿Qué conjunto de condiciones es necesario y suficiente para que tal secuencia deje sin cambios una cola no vacía?
- 4.1.3. Cuando un arreglo que guarda una cola no se considera circular, el texto sugiere que cada operación *remove* debe recorrer de manera descendente todos los elementos restantes de la cola. Otra posibilidad es posponer el corrimiento hasta que *rear* sea igual al último índice del arreglo. Cuando esto ocurre y se intenta efectuar una inserción, toda la cola se recorre de manera descendente, de modo que el primer elemento de la misma queda en la posición 0 del arreglo. ¿Qué ventajas tiene este método sobre el que recorre los elementos cada vez que se realiza una eliminación? ¿Cuáles son sus desventajas? Escribir otra vez las rutinas *remove*, *insert* y *empty* usando dicho método.
- 4.1.4. Demuestre cómo una secuencia de inserciones y eliminaciones puede causar desborde en una cola representada mediante un arreglo lineal al intentar insertar un elemento en una cola vacía.
- 4.1.5. Se puede evitar la eliminación de un elemento de la cola si se agrega un campo *qempty* a la representación de la misma. Muestre cómo puede hacerse lo anterior y escriba otra vez las rutinas de manipulación de colas por medio de tal representación.
- 4.1.6. ¿Cómo puede implantarse una cola de pilas? ¿Una pila de colas? ¿Una cola de colas? Escribir rutinas para implantar las operaciones apropiadas para cada una de estas estructuras de datos.
- 4.1.7. Muestre cómo implantar una cola de enteros en C usando un arreglo *queue*[100], donde *queue*[0] representa el frente de la cola, *queue*[1] el final y *queue*[2] hasta *queue*[99] se usan para contener los elementos de la cola. Mostrar cómo se inicializa tal arreglo para representar la cola vacía y escribir las rutinas *remove*, *insert* y *empty* para dicha implantación.
- 4.1.8. Muestre cómo implantar una cola en C, en la que cada elemento consiste en un número variable de enteros.
- 4.1.9. Una *cola doble* es un conjunto ordenado de elementos en el cual se pueden realizar inserciones y eliminaciones en cualquiera de los dos extremos. Llame a los extremos de la cola doble *left* y *right*. ¿Cómo puede representarse una cola doble como un arreglo en C? Escriba cuatro rutinas en C, *remvleft*, *remvright*, *insrtleft*, *insrtright*

para insertar y eliminar elementos en los extremos izquierdo y derecho de una cola doble. Asegúrese de que las rutinas funcionan correctamente para la cola doble vacía y detectan desborde y subdesborde.

- 4.1.10. Defina una *cola doble de entrada restringida* como una cola doble (vea ejercicio previo) para la que sólo son válidas las operaciones *remvleft*, *remvright* e *insrtleft*, y una *cola doble de salida restringida* como una cola doble para la cual sólo las operaciones *remvleft*, *insrtleft* e *insrtright* son válidas. Muestre cómo se puede usar cada una de ellas para representar tanto una cola como una pila.
- 4.1.11. El estacionamiento Scratchemup tiene un solo carril donde se guardan hasta 10 coches. Los coches entran por el extremo sur y salen por el extremo norte. Cuando un cliente llega por un coche que no está en el extremo norte, se sacan todos los coches que están más al norte de él, se saca éste del estacionamiento y se vuelven a meter los demás coches en el mismo orden en que estaban. Siempre que se va un coche, se recorren hacia delante todos los que estaban al sur de éste, de manera que todos los espacios vacíos siempre están en la parte sur del estacionamiento.  
Escriba un programa que lea un grupo de líneas de entrada. Cada línea de entrada contiene una 'E' para la entrada o una 'S' para la salida y un número de placas. Se supone que los coches llegan y salen en el orden especificado por la entrada. El programa deberá imprimir un mensaje cada vez que entra o sale un coche. Cada vez que entra un coche, el mensaje deberá especificar si hay o no un lugar para él en el estacionamiento. Si no lo hay, el coche espera hasta que lo haya o hasta que se lea una línea de salida para él. Cuando aparece un espacio disponible, se debe imprimir otro mensaje. Cuando un coche se va, el mensaje debe incluir el número de veces que se movió dentro del estacionamiento, incluyendo la salida pero no la entrada. Si el carro se retiró de la cola de espera, este número es 0.
- 4.1.12. Desarrolle una especificación de ADT para una cola de prioridad.
- 4.1.13. Implante una cola de prioridad ascendente y sus operaciones *pqinsert*, *pqmindelete* y *empty*, usando cada uno de los cuatro métodos presentados en el texto.
- 4.1.14. Muestre cómo se clasifica un conjunto de números de entrada por medio de una cola de prioridad y las operaciones *pqinsert*, *pqmindelete* y *empty*.

## 4.2. LISTAS LIGADAS

¿Cuáles son las desventajas de usar memoria secuencial para representar pilas y colas? La mayor desventaja, es que una cantidad de memoria fija permanece asignada a la pila o la cola aun cuando éstas usen una cantidad menor o incluso ninguna. Además, no puede asignar más memoria que la que indica esa cantidad fija, con lo que se introduce la posibilidad de desborde. Supóngase que un programa usa dos pilas implantadas en dos arreglos distintos: *s1.items* y *s2.items*. Supóngase además que cada uno de esos arreglos tiene 100 elementos. Entonces, a pesar de que están disponibles 200 elementos para las dos pilas, ninguna de ellas puede contener por separado más de 100. Aun cuando la primera pila contenga sólo 25, la segunda no podrá contener más de 100. Una solución a este problema es asignar un arreglo único *items* de 200 elementos. La primera pila ocupa de *items*[0], *items*[1], ..., *items*[*top1*], mientras que la segunda pila es colocada en el otro extremo del arreglo, ocupando *items*[199], *items*[198], ..., *items*[*top2*] en orden descendente. Así, cuando una pila no ocupa una parte de memoria, la otra puede hacerlo. Desde luego que se necesitan otras rutinas

distintas *pop*, *push* y *empty* para las dos pilas, ya que una crece cuando se incrementa *top1*, mientras que la otra lo hace cuando disminuye *top2*.

Desafortunadamente, aunque un esquema como éste permite compartir un área común, no existe una solución tan simple para el caso de tres o más pilas, ni siquiera para el de dos colas. En lugar de esto, se tienen que tomar en cuenta los topes y los fondos (o frentes y finales) de todas las estructuras que comparten un solo arreglo grande. Cada vez que el crecimiento de una estructura trata de violar la memoria que está usando otra, las estructuras vecinas deben recorrerse dentro del mismo arreglo para permitir el crecimiento.

En una representación secuencial, los elementos de la pila o cola están ordenados de manera implícita por el orden secuencial de memoria. Así, si *q.items[x]* representa un elemento de una cola, el elemento siguiente será *q.items[x + 1]* (o, *q.items[0]*, si *x* es igual a *MAXQUEUE - 1*). Supóngase que los elementos de una pila o cola se ordenaron de manera explícita, esto es, que cada elemento contenga en sí mismo la dirección del siguiente. Un ordenamiento explícito como ése, da origen a una estructura de datos como la de la figura 4.2.1, que se conoce como *lista lineal ligada*. Cada elemento de la lista se conoce como un *nodo* y contiene dos campos: uno de *información* y otro de *dirección siguiente*. El campo de información guarda el elemento real de la lista; el de dirección siguiente contiene la dirección del siguiente nodo de la lista. Una dirección de este tipo, que se usa para accesar un nodo en particular, se conoce como *apuntador*. La lista ligada completa se accesa desde un apuntador externo *list* que apunta al que contiene la dirección del primer nodo de la misma. (Por apuntador “externo” se entiende que es uno que no está incluido dentro del nodo. Por el contrario, su valor puede accesarlo de manera directa referenciando una variable). El campo de dirección siguiente del último nodo de la lista contiene un valor especial conocido como *null*, que no es una dirección válida. Este *apuntador nulo* se usa para señalar el final de la lista.

La lista que no contiene nodos se llama *lista vacía* o *lista nula*. El valor del apuntador externo, *list*, a una lista vacía es el apuntador nulo. Una lista puede inicializarse como la lista vacía mediante la operación *list = null*.

Ahora se introducirán algunas notaciones que pueden usarse en algoritmos (pero no en programas en C). Si *p* es un apuntador a un nodo, *node(p)* se refiere al nodo apuntado por *p*; *info(p)* se refiere a la porción de información del nodo, y *next(p)* se refiere a la porción de la dirección siguiente y es, por lo tanto, un apuntador. Así, si *next(p)* no es nulo (*null*), *info(next(p))* se refiere a la porción de información del nodo que sigue a *node(p)* en la lista.

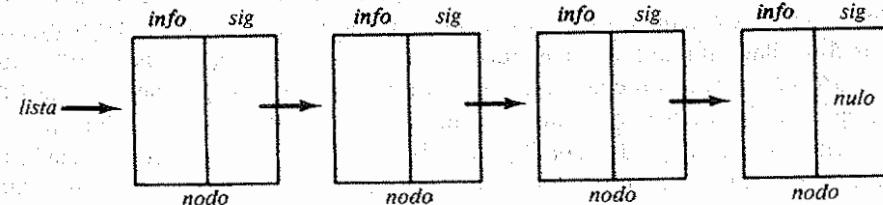


Figura 4.2.1 Una lista lineal ligada.

Antes de continuar con el análisis acerca de las listas ligadas, debe mencionarse que éstas se presentan de manera primaria como una estructura de datos (es decir, un método de implantación) en lugar de como un tipo de datos (esto es, como una estructura lógica con las operaciones originales definidas en forma precisamente). Por lo tanto, no se presenta aquí una especificación de ADT para listas ligadas. En la sección 9.1, se analizarán las listas como estructuras abstractas y se presentarán algunas operaciones originales para ellas.

En esta sección se presenta el concepto de lista ligada y se muestra la forma de usarla. En la siguiente sección se mostrará cómo implantar en C listas ligadas.

### Inserción y eliminación de nodos de una lista

Una lista es una estructura de datos dinámica. El número de nodos de una lista puede variar dramáticamente cuando se insertan o eliminan elementos. La naturaleza dinámica de una lista contrasta con la naturaleza estática de un arreglo, cuyo tamaño permanece constante.

Por ejemplo, supóngase que se da una lista de enteros, como se ilustra en la figura 4.2.2a. Se desea agregar el entero 6 al frente de la lista. Es decir, se desea cambiar la lista para que quede como la de la figura 4.2.2f. El primer paso es obtener un nodo donde alojar el entero adicional. Si una lista es una estructura que puede crecer y contraerse, debe haber algún mecanismo para obtener nodos vacíos que se puedan agregar a ésta. Adviértase que, a diferencia de un arreglo, una lista no tiene un conjunto suministrado con anterioridad de localidades de memoria en el que se puedan guardar elementos.

Supóngase la existencia de un mecanismo para obtener nodos vacíos. La operación

```
p = getnode();
```

obtiene un nodo vacío y pone los contenidos de una variable llamada *p* en la dirección de ese nodo. Entonces el valor de *p* es un apuntador a ese nodo recién asignado. La figura 4.2.2b muestra la lista y el nuevo nodo después de ejecutar la operación *getnode*. Los detalles del funcionamiento de dicha operación serán explicados en breve.

El paso siguiente es insertar el entero 6 dentro de la porción *info* del nuevo nodo asignado. Eso se hace mediante la operación

```
info(p) = 6;
```

El resultado de esta operación se muestra en la figura 4.2.2c.

Después de asignar el valor correspondiente a la porción *info* de *node(p)* es necesario hacer lo mismo con la porción *next* de dicho nodo. Como *node(p)* debe insertarse en el frente de la lista, el nodo que le sigue será el primer nodo vigente. Como la variable *list* contiene la dirección de este primer nodo, se puede agregar *node(p)* a la lista por medio de la operación

```
next(p) = list;
```

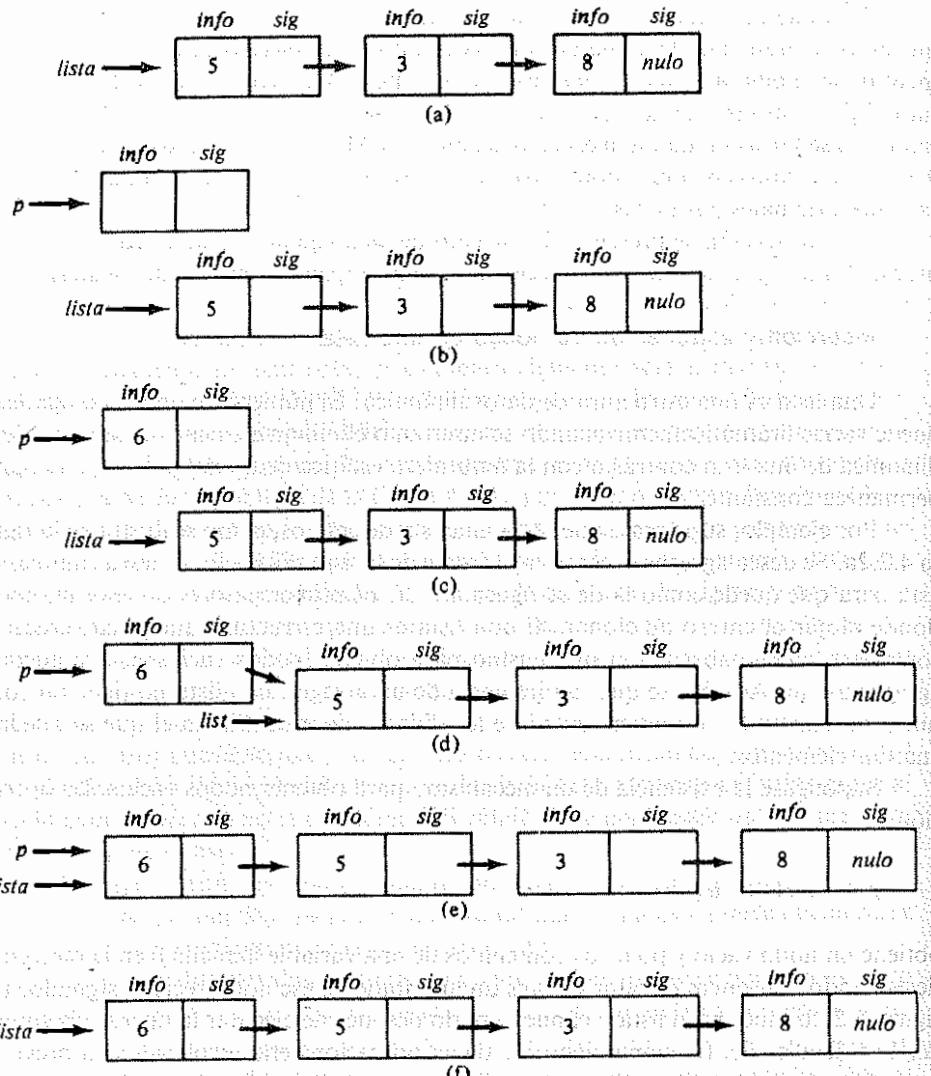


Figura 4.2.2 Añadiendo un elemento al frente de una lista.

Esta operación coloca el valor de *list* (que es la dirección del primer nodo de la lista) en el campo *next* de *node(p)*. La figura 4.2.2d ilustra el resultado de esta operación.

En este momento, *p* apunta a la lista con el elemento adicional incluido. Sin embargo, como *list* es el apuntador externo a la lista deseada, es necesario modificar su valor a la dirección del nuevo primer nodo. Esto puede hacerse ejecutando la operación

```
list = p;
```

que cambia el valor de *list* al valor de *p*. La figura 4.2.2e ilustra el resultado de esta operación. Adviértase que las figuras 4.2.2e y f son idénticas, excepto en que el valor de *p* no se muestra en la figura 4.2.2f. Esto es porque *p* se usa como variable auxiliar durante el proceso de modificación de la lista, pero su valor es irrelevante para el estado de la misma antes y después del proceso. Una vez que se ejecutan las operaciones anteriores, puede cambiarse el valor de *p* sin afectar la lista.

Al unir todos los pasos, se obtiene un algoritmo para agregar el entero 6 al frente de la lista *list*:

```
p = getnode();
info(p) = 6;
next(p) = list;
list = p;
```

Es obvio que el algoritmo puede generalizarse para que agregue cualquier objeto *x* al frente de la lista *list* si se remplaza la operación *info(p) = 6* por *info(p) = x*. Es necesario comprobar que el algoritmo *trabaja* en forma correcta aun cuando la lista esté inicialmente vacía (*list* == *null*).

La figura 4.2.3 ilustra el proceso de eliminación del primer nodo de una lista no vacía y el almacenamiento del valor de su campo *info* en la variable *x*. Las configuraciones inicial y final se muestran en las figuras 4.2.3a y f, respectivamente. El proceso mismo es casi opuesto al proceso en que se agrega un elemento en el frente de una lista. Para obtener la figura 4.2.3d de la figura 4.2.3a, se ejecutan las siguientes operaciones (cuyas acciones deben ser claras):

```
p = list;
list = next(p);
x = info(p);
```

(Figura 4.2.3b)  
 (Figura 4.2.3c)  
 (Figura 4.2.3d)

En este punto, el algoritmo ya realizó lo que supuestamente debía hacer: eliminar el primer nodo de la lista *list* y asignar a *x* el valor deseado. Sin embargo, el algoritmo no está completo aún. En la figura 4.2.3d, *p* todavía apunta al nodo que antes era el primero de la lista. Pero este nodo no se usa actualmente, pues ya no está en la lista y su información queda almacenada en *x*. (No se considera que el nodo está en la lista a pesar del hecho de que *next(p)* apunte a un nodo, ya que no hay manera de alcanzar *node(p)* desde el apuntador externo *list*.)

La variable *p* se usa como variable auxiliar durante el proceso de eliminación del primer nodo. Las configuraciones inicial y final de la lista no hacen referencia a *p*. De ahí que sea razonable esperar que *p* será usado para algún otro propósito, poco después de la ejecución de esta operación. Pero una vez que se cambia el valor de *p*, no hay manera de accesar el nodo, ya que ni un apuntador externo ni un campo *next* contienen su dirección. En consecuencia, el nodo es inútil en este momento y no puede volverse a usar, aun cuando esté ocupando una memoria valiosa.

Sería deseable contar con algún mecanismo para dejar disponible *node(p)* y volverlo a usar aun cuando se cambie el valor del apuntador *p*. La operación que hace esto es

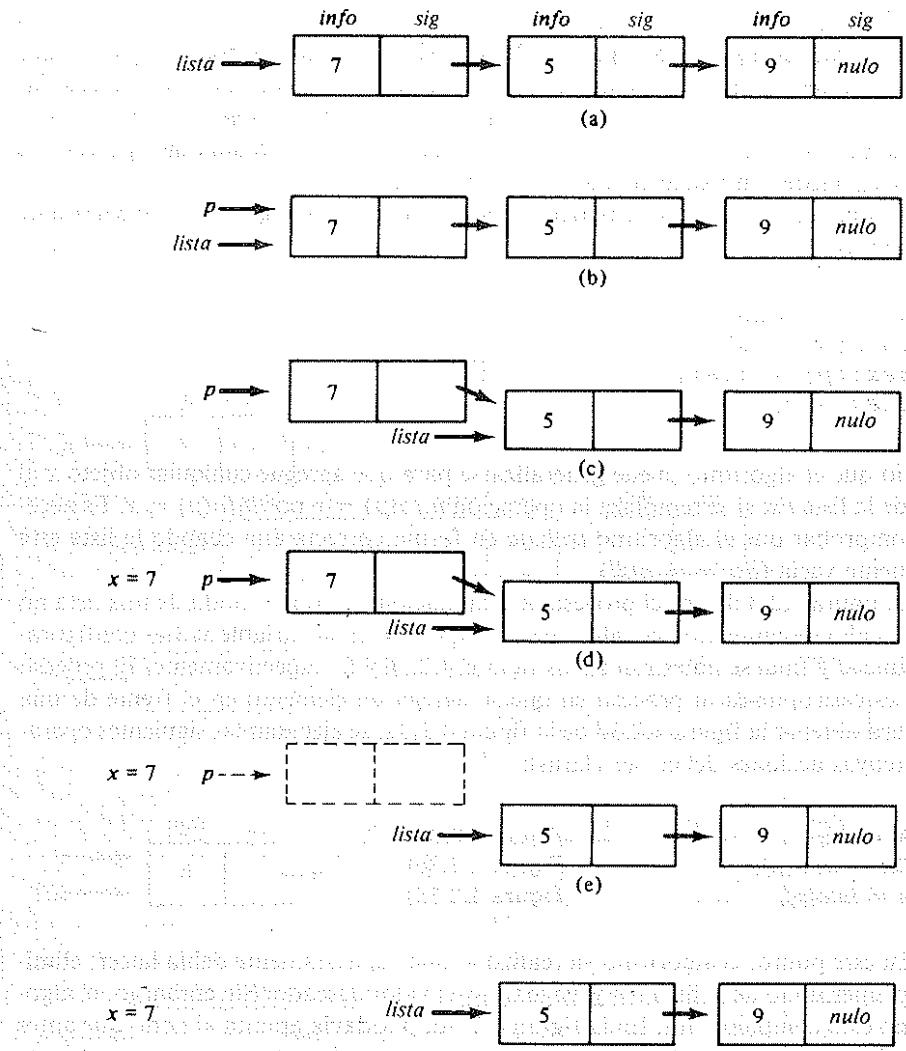


Figura 4.2.3 Eliminación de un nodo del frente de una lista.

Una vez que se ejecuta esta operación, es ilegal aludir a *node(p)*, debido a que el nodo ya no está localizado. Como el valor de *p* es un apuntador a un nodo que ha sido liberado, también es ilegal cualquier referencia a ese valor.

Sin embargo, se podría reubicar el nodo y se podría volver a asignar a *p* un apuntador mediante la operación *p = getnode()*. Adviértase que se dice que el nodo "podría ser" reubicado, ya que la operación *getnode* regresa un apuntador a algún

nodo que se acaba de asignar. No hay garantía de que este nuevo nodo sea el mismo que se acaba de liberar.

Otra manera de pensar en *getnode* y *freenode* es que *getnode* crea un nuevo nodo, mientras que *freenode* destruye un nodo. Desde este punto de vista, los nodos no se usan y se vuelven a usar, sino que se crean y se destruyen. Más adelante se verá un poco más acerca de las operaciones *getnode* y *freenode*, así como de los conceptos que éstas representan, pero antes se hará la siguiente observación interesante.

### Implantación ligada de pilas

La operación de agregar un elemento al frente de una lista ligada es bastante parecida a la de poner un elemento en una pila. En ambos casos se agrega un nuevo elemento como el único inmediatamente accesible de una colección. Una pila sólo se puede accesar a través de su elemento tope, y una lista sólo se puede accesar a su primer elemento desde el apuntador. De manera similar, la operación de eliminar el primer elemento de una lista ligada es análoga a la de sacar un elemento de la pila. En ambos casos, el único elemento inmediatamente accesible de una colección se elimina de ésta, y el siguiente se vuelve inmediatamente accesible.

De este modo se ha descubierto otra manera de implantar una pila. Una pila se puede representar por medio de una lista lineal ligada. El primer nodo de la lista es el tope de la pila. Si un apuntador externo *s* apunta a una lista ligada de este tipo, la operación *push(s, x)* se puede implantar mediante

```
p = getnode();
info(p) = x;
next(p) = s;
s = p;
```

La operación *empty(s)* no es más que la verificación de que *s* es igual al apuntador nulo (*null*). La operación *x = pop(s)* elimina el primer nodo de una lista no vacía y señala subdesborde si la lista está vacía:

```
if (empty(s)) {
 printf("subdesborde en la pila");
 exit(1);
} else {
 p = s;
 s = next(p);
 x = info(p);
 freenode(p);
} /* fin de if */
```

La figura 4.2.4a ilustra una pila implantada como una lista ligada, y la 4.2.4b, la misma pila después de que se le agregó otro elemento.

La ventaja de la implantación de pilas como listas es que todas las pilas que se usan en un programa pueden compartir la misma lista disponible. Cuando una pila

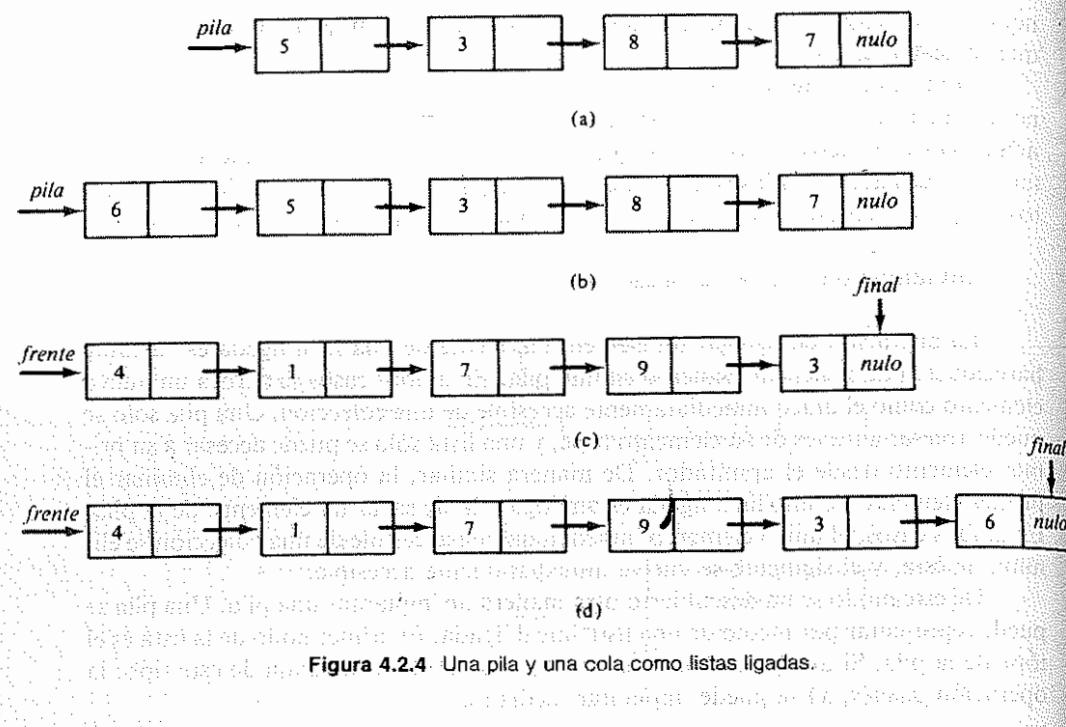


Figura 4.2.4. Una pila y una cola como listas ligadas.

necesita un nodo, lo puede obtener de dicha lista; cuando ya no lo necesita, lo regresa a la misma lista disponible. Cada pila puede crecer y contraerse a cualquier tamaño, siempre y cuando la cantidad total de espacio requerida por todas las pilas, en cualquier momento, sea menor que la cantidad de espacio disponible inicialmente para todas ellas. No se asignó con anterioridad espacio para ninguna pila en particular y ninguna pila usa espacio que no necesite. Además, otras estructuras de datos, como las colas dobles, pueden compartir también el mismo conjunto de nodos.

#### Las operaciones *getnode* y *freenode*

Ahora se retomará el análisis de las operaciones *getnode* y *freenode*. En un mundo idealizado y abstracto es posible postular un número infinito de nodos nuevos disponibles para uso de algoritmos abstractos. La operación *getnode* encuentra dicho nodo y lo deja disponible para el algoritmo. Otra posibilidad es considerar la operación *getnode* como una máquina que fabrica nodos y que nunca para. Cada vez que se llama a *getnode*, se presenta ante quien la llama con un nuevo nodo, que es distinto a todos los nodos previamente usados.

En un mundo ideal como ése, la operación *freenode* sería innecesaria para hacer que un nodo disponible pueda utilizarse otra vez. ¿Para qué usar un nodo de segunda mano cuando una simple llamada a *getnode* puede producir uno nuevo? El único daño que puede provocar un nodo sin uso es reducir el número de nodos disponibles; pero si hay un suplemento infinito de nodos, tal reducción es insignificante. En consecuencia, no hay razón para volver a usar un nodo.

Desafortunadamente, vivimos en un mundo real. Las computadoras no tienen una cantidad infinita de memoria y no tienen la capacidad de fabricar más para uso inmediato (por lo menos, no hasta ahora). Por lo tanto, hay un número finito de nodos disponibles y sólo se puede usar dicha cantidad en un momento dado. Si se desea usar más nodos que los existentes durante un periodo de tiempo determinado, algunos nodos tendrán que usarse otra vez. La función de *freenode* es hacer que un nodo que no se usa en su contexto actual quede disponible para su uso en un contexto diferente.

Puede pensarse que al principio hay un fondo común y finito de nodos vacíos, que el programador no puede accesar, excepto a través de las operaciones *getnode* y *freenode*. *getnode* elimina un nodo del fondo común mientras que *freenode* lo regresa. Como un nodo sin uso es tan bueno como cualquier otro, da igual qué nodo recupera *getnode* o en qué parte coloca un nodo *freenode*.

La forma más natural que puede tomar dicho fondo es la de una lista ligada que actúa como una pila. La lista se liga por medio del campo *next* en cada nodo. La operación *getnode* elimina el primer nodo de esta lista y lo deja disponible para su uso. La operación *freenode* agrega un nodo al frente de la lista, dejándolo disponible para su reubicación mediante la siguiente aplicación de *getnode*. La lista de nodos disponibles se denomina *lista disponible*.

¿Qué ocurre cuando la lista disponible está vacía? Esto significa que todos los nodos están en uso y es imposible asignar uno más. Si un programa llama a *getnode* cuando la lista está vacía, esto significa que la cantidad de memoria asignada a las estructuras de datos del programa en cuestión es muy pequeña. Por lo tanto, ocurre desborde. Esto es similar a la situación de una pila implantada en un arreglo, la que sobrepasa los límites del mismo.

Siempre que las estructuras de datos sean conceptos teóricos-abstratos en un mundo de espacio infinito, no hay posibilidad de desborde. Este sólo surge cuando se implantan dichas estructuras como objetos reales en un área finita.

Supóngase que un apuntador externo *avail* apunta a una lista de nodos disponibles. Entonces la operación

```
p = getnode();
```

se implanta como sigue:

```
if (avail == null) {
 printf("desborde");
 exit(1);
}
p = avail;
avail = next(avail);
```

Como la posibilidad de desborde da razón de la operación *getnode*, no es necesario mencionar ésta en la implantación de lista de *push*. Si una pila está a punto de desbordar la cantidad de nodos disponibles, la instrucción *p = getnode()* dentro de la operación *push* da lugar a un desborde.

La implantación de *freenode(p)* es directa:

```
next(p) = avail;
avail = p;
```

### Implantación ligada de colas

Ahora se examinará el modo de representar una cola como una lista ligada. Hay que recordar que en una cola los elementos se eliminan del frente y se insertan al final. Supóngase que el frente de la cola está representado por un apuntador al primer elemento de una lista y que otro apuntador ligado al último elemento de la lista representa el final de la cola, como se muestra en la figura 4.2.4c. La figura 4.2.4d ilustra la misma cola después de insertar un nuevo elemento.

Bajo la representación de lista, una cola *q* consiste en una lista y dos apuntadores *q.front* y *q.rear*. Las operaciones *empty(q)* y *x = remove(q)* son completamente análogas a *empty(s)* y *x = pop(s)*, con el apuntador *q.front* remplazando a *s*. Sin embargo, se debe poner mucha atención al eliminar el último elemento de una cola. En tal caso, a *q.rear* también se le asigna *null* (se le debe asignar el apuntador nulo), ya que en una cola vacía tanto *q.rear* como *q.front* deben ser *nulos*. Por lo tanto, el algoritmo para *x = remove(q)* es el siguiente:

```
if (empty(q)) {
 printf("subdesborde de la cola");
 exit(1);
}
p = q.front;
x = info(p);
q.front = next(p);
if (q.front == null)
 q.rear = null;
freenode(p);
return(x);
```

La operación *insert(q, x)* se implanta mediante

```
p = getnode();
info(p) = x;
next(p) = null;
if (q.rear == null)
 q.front = p;
else
 next(q.rear) = p;
q.rear = p;
```

¿Cuáles son las desventajas de representar una pila o cola mediante una lista ligada? Resulta evidente que en una lista ligada un nodo ocupa más espacio de memo-

ria que el que ocupa un elemento correspondiente en un arreglo, pues en una lista se necesitan dos campos de información para cada nodo (*next* e *info*), mientras que en la implantación con el arreglo sólo se necesita el elemento. Sin embargo, el espacio que para el nodo de una lista no duplica en general el espacio que se usa para un elemento de un arreglo, pues los elementos de tal lista consisten por lo general en estructuras con muchos subcampos. Por ejemplo, si cada elemento de una pila es una estructura que ocupa diez palabras, la adición de una onceava palabra que contendría un apuntador incrementaría el espacio requerido sólo en 10 porciento. Además, a veces es posible comprimir la información y el apuntador en una sola palabra, con lo que se gana espacio.

Otra desventaja es el tiempo adicional gastado en manejar la lista disponible. Cada adición y eliminación de un elemento de una pila o cola comprende una eliminación o una adición correspondiente a la lista disponible.

La ventaja de usar listas ligadas es que todas las listas y pilas de un programa tienen acceso a la misma lista libre de nodos. Los nodos que no usa una pila, puede usarlos otra, siempre y cuando el número total de nodos usado al mismo tiempo no sea mayor que el número total de nodos disponible.

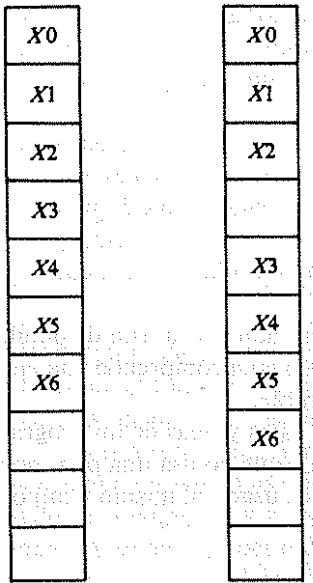
### La lista ligada como una estructura de datos

Las listas ligadas no sólo son importantes como un medio para implantar pilas y colas, sino como estructuras de datos por derecho propio. En una lista ligada, la forma de accesar un elemento es recorriendo la lista desde el principio. La implantación con arreglo permite accesar el *n*-ésimo elemento de un grupo por medio de una sola operación, mientras que en una implantación con lista se requiere de *n* operaciones. Es necesario pasar a través de los primeros *n* — 1 elementos antes de alcanzar el elemento *n*-ésimo de una lista, ya que no hay relación entre la localidad de memoria que ocupa un elemento de una lista y su posición en la misma.

Cuando se debe insertar o eliminar un elemento en medio de un grupo de otros elementos, la lista aventaja al arreglo. Por ejemplo, supóngase que se desea insertar un elemento *x* entre el tercero y cuarto elementos de un arreglo de tamaño 10 que en ese momento contiene siete elementos (*x[0]* hasta *x[6]*). Primero hay que mover una casilla los elementos 6 a 3, e insertar el nuevo elemento en la posición 3, recientemente desocupada. En la figura 4.2.5a se ilustra este proceso. En este caso, la inserción de un elemento propicia el movimiento de cuatro elementos, además de la inserción misma. Si el arreglo tuviera 500 o 1000 elementos, también tendría que moverse el enorme número de elementos correspondiente. De manera análoga, para eliminar un elemento de un arreglo sin dejar espacio, se deben recorrer una posición todos los elementos que están más allá de éste.

Por otra parte, supóngase que los elementos están almacenados como una lista. Si *p* apunta a un elemento de la lista, la inserción de un elemento nuevo después de *node(p)* implica la ubicación de un nodo, la inserción de la información y el ajuste de dos apuntadores. La cantidad de trabajo requerida es independiente al tamaño de la lista. En la figura 4.2.5b se ilustra esto:

Supóngase que *insafter(p, x)* denota la operación que inserta un elemento *x* dentro de una lista después de un nodo apuntado por *p*. Esta operación se implanta de la siguiente manera:



(a)

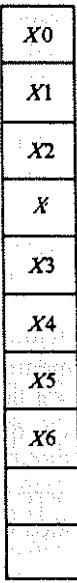


Figura 4.2.5

Supóngase que  $node(p)$  denota el apuntador al nodo que sigue a  $p$ . Si se quiere insertar un elemento en una lista lineal, se procede de la siguiente manera:

```

 q = getnode();
 info(q) = x;
 next(q) = next(p);
 next(p) = q;

```

Al finalizar, el nodo  $q$  apunta al elemento  $x$ , y el apuntador  $p$  apunta al nodo que sigue a  $x$ .

Un elemento puede insertarse sólo después de un nodo determinado, no antes. Esto es porque no hay manera de proceder de un nodo determinado a su antecesor en una lista lineal sin recorrer la lista desde el principio. Para insertar un elemento antes que  $node(p)$  es necesario cambiar primero el campo *next* de su antecesor para apuntar al nodo recién ubicado. Pero, al determinar  $p$ , no hay manera de encontrar ese antecesor. (Sin embargo, se puede producir el efecto de inserción de un elemento antes de  $node(p)$  insertando el elemento inmediatamente después de  $node(p)$  e intercambiando después la información *info(p)* con el campo *info* del sucesor recién creado. Al lector se le dejan los detalles.)

Análogamente, para eliminar un nodo de una lista lineal no basta con dar un apuntador a ese nodo. Esto es porque el campo *next* del antecesor del nodo debe cambiarse para que apunte al sucesor del nodo, y no hay manera directa de alcanzar el antecesor de un nodo determinado. Lo mejor que puede hacerse es eliminar un nodo que sigue a otro determinado. (Sin embargo, es posible salvar los contenidos del nodo siguiente, eliminar éste y luego remplazar los contenidos del nodo determinado con la información salvada. Esto produce el efecto de eliminar un nodo determinado a menos que éste sea el último de la lista.)

Supóngase que  $delafter(p, x)$  denota la operación de eliminar el nodo que sigue a  $node(p)$  y asignar su contenido a la variable  $x$ . Esta operación puede implantarse como sigue:

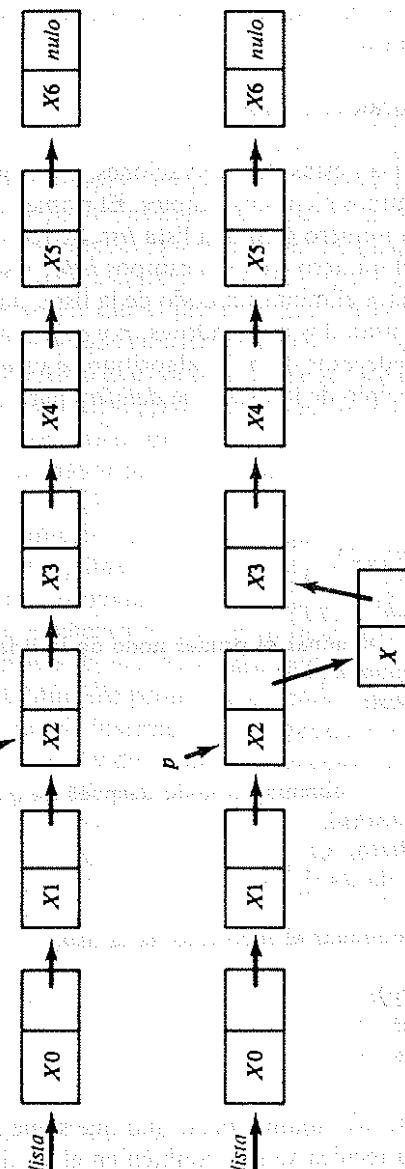


Figura 4.2.5 (cont.)

```

q = next(p);
x = info(q);
next(p) = next(q);
freenode(q);

```

El nodo liberado se coloca dentro de la lista disponible de manera que pueda usarse otra vez en el futuro.

### Ejemplos de operaciones con listas

Enseguida se ilustran estas dos operaciones, así como las operaciones *push* y *pop* para listas, con algunos ejemplos simples. El primer ejemplo consiste en borrar todos los incidentes del número 4 en una lista *list*. Se recorre la lista para buscar los nodos que contengan el número 4 en sus campos *info*, y se debe borrar cada uno de éstos de la lista. Pero para eliminar un nodo de la lista, hay que conocer a su predecesor. Por tal razón se usan dos apuntadores, *p* y *q*. *p* se usa para recorrer la lista y *q* apunta siempre al predecesor de *p*. El algoritmo se vale de la operación *pop* para eliminar nodos del principio de la lista y de *delafter* para eliminar nodos de la parte media.

```

q = null;
p = list;
while (p != null) {
 if (info(p) == 4) {
 if (q == null) {
 /* eliminar el primer nodo de la lista */
 x = pop(list);
 p = list;
 } else {
 /* eliminar el nodo después de q y mover p */
 p = next(p);
 delafter(q, x);
 } /* fin de if */
 } /* continuar el recorrido de la lista */
 q = p;
 p = next(p);
} /* fin de if */
} /* fin de while */

```

La práctica de usar dos apuntadores, uno que sigue al otro, es muy común al trabajar con listas. Esta técnica se usa también en el siguiente ejemplo. Supóngase que una lista *list* está ordenada de menor a mayor. A una lista de este tipo se la llama *lista ordenada*. Se desea insertar un elemento *x* en esta lista, en el lugar correspondiente. Para hacer esto, el algoritmo se vale de la operación *push* para agregar un nodo al frente de la lista y de la operación *insafter* para agregar uno en medio de la lista:

```

q = null;
for (p = list; p != null && x > info(p); p = next(p));
q = p;
/* En este momento, un nodo que contenga x deberá insertarse */
if (q == null) /* insertar x a la cabeza de la lista */
 push(list, x);
else
 insafter(q, x);

```

Esta operación, que es muy común, será denotada por *place(list, x)*.

Examíñese la eficacia de la operación *place*. ¿Cuántos nodos se acceden en promedio para insertar un elemento nuevo en una lista ordenada? Asúmase que la lista contiene *n* nodos. Entonces *x* puede insertarse en una de las *n* + 1 posiciones; esto es, se puede encontrar que *x* es menor que el primer elemento de la lista, que está entre el primero y el segundo,..... entre el (*n* - 1)-ésimo y el *n*-ésimo y que es mayor que este último. Si *x* es menor que el primero, *place* accesa sólo el primer nodo de la lista (aparte del nodo nuevo que contiene a *x*); esto es, determina de inmediato que *x* < *info(list)* e inserta un nodo que contiene *x* que se vale de *push*. Si *x* está entre el *k*-ésimo y el (*k* + 1)-ésimo elemento, *place* accesa los primeros *k* nodos; sólo después de verificar que *x* es menor que el contenido del (*k* + 1)-ésimo nodo se inserta *x* por medio de *insafter*. Si *x* es mayor que el *n*-ésimo elemento, se acceden todos los nodos *n*.

Supóngase ahora que también hay probabilidades de que *x* se inserte en cualquiera de las *n* + 1 posiciones posibles. (Si esto es cierto, puede decirse que la inserción es *aleatoria*.) Entonces la probabilidad de hacer la inserción en una posición particular es  $1/(n+1)$ . Si el elemento se inserta entre la posición *k*-ésima y la (*k* + 1)-ésima, el número de accesos es *k* + 1. Si se inserta después del *n*-ésimo elemento, el número de accesos es *n*. El número promedio de nodos accedidos, *A*, es la suma, en todas las posibles posiciones de inserción, de los productos de la probabilidad de insertar en una posición particular y del número de accesos requeridos para insertar un elemento en dicha posición. Así

$$A = \left( \frac{1}{n+1} \right) * 1 + \left( \frac{1}{n+1} \right) * 2 + \dots + \left( \frac{1}{n+1} \right) * (n-1) + \left( \frac{1}{n+1} \right) * n + \left( \frac{1}{n+1} \right) * n$$

Ahora  $1 + 2 + \dots + n = n * \frac{n+1}{2}$ . (Esto puede probarse fácilmente por medio de inducción matemática). Por lo tanto

$$A = \left( \frac{1}{n+1} \right) * (n * \frac{n+1}{2}) + \frac{n}{n+1} * \frac{n}{2} = \frac{n}{2} + \frac{n}{n+1}$$

Cuando  $n$  es grande,  $n/(n + 1)$  se acerca a 1, de manera que  $A$  es en forma aproximada  $n/2 + 1$  o  $(n + 2)/2$ . Para  $n$  grande,  $A$  está muy cercana a  $n/2$ , por lo que a menudo se dice que la operación de inserción aleatoria de un elemento en una lista ordenada requiere de casi  $n/2$  accesos de nodo en promedio.

### Implantación con lista de colas de prioridad

Una lista ordenada puede usarse para representar una cola de prioridad. Para la cola de prioridad ascendente se implanta la inserción (*pqinsert*) mediante la operación *place*, la que mantiene en orden la lista, y la eliminación del elemento mínimo (*pqmindelete*) mediante la operación *pop*, la que elimina el primer elemento de la lista. Una cola de prioridad descendente puede implantarse guardando la lista en orden descendente, en lugar de ascendente, o usando *remove* para implantar *pqmaxdelete*. Una cola de prioridad implantada como lista ordenada ligada necesita examinar un número promedio de casi  $n/2$  nodos para la inserción, pero un solo nodo para la eliminación.

Una lista no ordenada puede usarse también como cola de prioridad. Una lista de este tipo sólo necesita examinar un nodo para la inserción (implantando *pqinsert* por medio de *push* o *insert*), pero para la eliminación debe examinar los  $n$  elementos (recorrer la lista entera para encontrar el mínimo o máximo y luego eliminar ese nodo). Así, una lista ordenada es un tanto más eficaz que una no ordenada en la implantación de una cola de prioridad.

La ventaja de una lista sobre un arreglo en la implantación de una cola de prioridad es que en la primera no se necesita el recorrimiento de elementos. Se puede insertar un elemento en una lista sin mover otro, mientras que en un arreglo es imposible hacerlo a menos que se deje un espacio vacío. En las secciones 6.3 y 7.3 se examinan otras implantaciones de la cola de prioridad más eficaces.

### Nodos cabecera

A veces es deseable guardar un nodo extra en el frente de una lista. A tal nodo, que no representa un elemento de la lista, se le llama *nodo cabecera* o *cabecera de la lista*. Puede que la porción *info* del nodo cabecera no se use, como se observa en la figura 4.2.6a. Más a menudo, la porción *info* de dicho nodo se usa para guardar información global acerca de toda la lista. Por ejemplo, la figura 4.2.6b ilustra una lista en la que la porción *info* del nodo cabecera contiene el número de nodos (sin incluir la cabecera) de la lista. En una estructura de datos como ésta se necesita trabajar más para agregar o eliminar un elemento de la lista, debido a que hay que ajustar el conteo en la cabecera de la lista. Sin embargo, el número de elementos de la lista puede obtenerse de modo directo del nodo cabecera sin atravesar toda la lista.

Otro ejemplo del uso de nodos cabecera es el siguiente. Supóngase que en una fábrica se ensambla maquinaria en pequeñas unidades. Una máquina particular (número de inventario *A746*) podría estar compuesta de un número de partes diferentes (números *B841*, *K321*, *A087*, *J492*, *G593*). Este montaje estaría representado por una lista como la de la figura 4.2.6c, donde cada artículo de la lista representa un componente y el nodo cabecera todo el montaje.

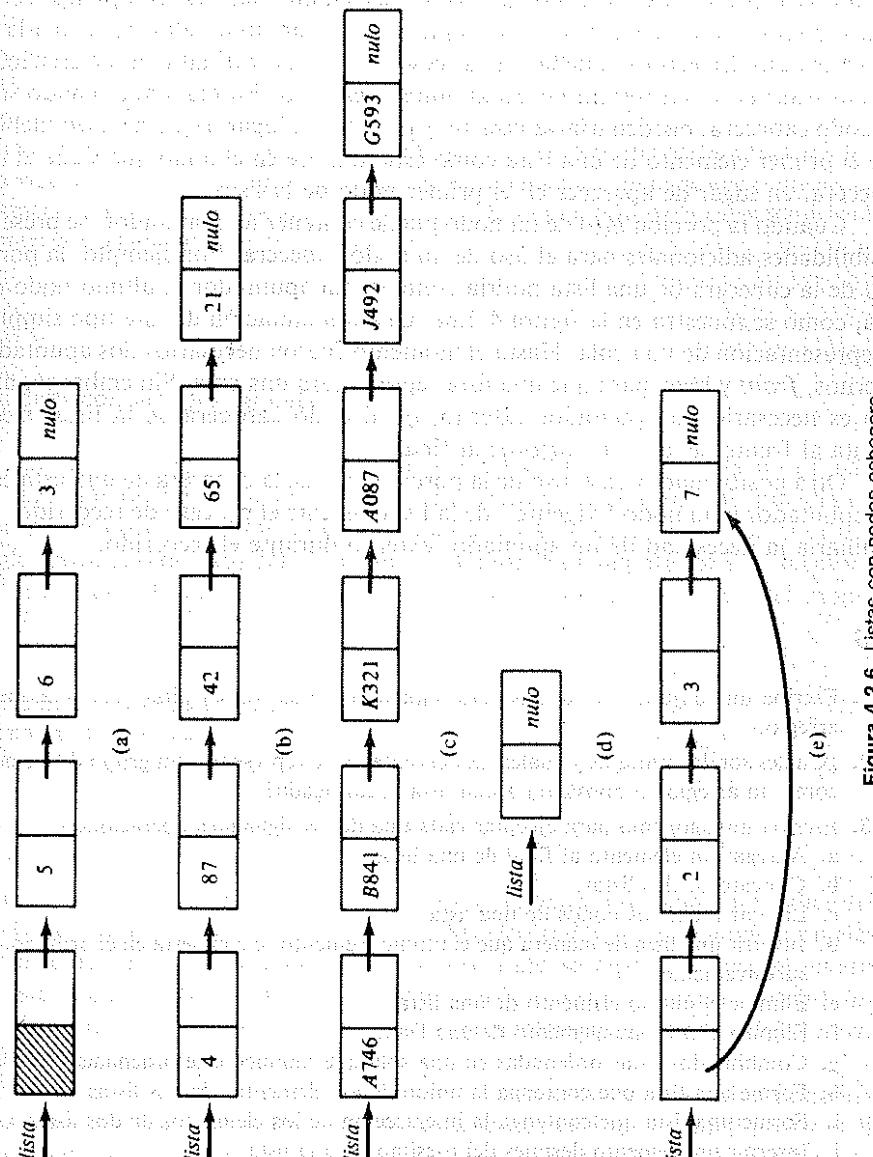


Figura 4.2.6 Listas con nodos cabecera.

La lista vacía ya no estaría representada por el apuntador nulo, sino por una lista con un nodo cabecera simple, como en la figura 4.2.6d.

Por supuesto que los algoritmos para operaciones como *empty*, *push*, *pop*, *insert* y *remove* se tienen que escribir de nuevo de manera que expliquen la presencia del nodo cabecera. Casi todas las rutinas se vuelven un poco más complejas, pero algunas, como *insert*, se vuelven más simples, ya que un apuntador externo a la lista nunca es nulo. El lector se queda con la reescritura de las rutinas como ejercicio. Las rutinas *insafter* y *delafter* no necesitan ningún cambio. En realidad, cuando se usa un nodo cabecera, pueden usarse *insafter* y *delafter* en lugar de *push* y *pop*, debido a que el primer elemento de una lista como ésta aparece en el nodo que sigue al nodo cabecera, en lugar de aparecer en el primer nodo de la lista.

Cuando la porción *in/o* de un nodo puede contener un apuntador, se presentan posibilidades adicionales para el uso de un nodo cabecera. Por ejemplo, la porción *info* de la cabecera de una lista podría contener un apuntador al último nodo de la lista, como se muestra en la figura 4.2.6e. Una implantación de este tipo simplifica la representación de una cola. Hasta el momento fueron necesarios dos apuntadores externos, *front* y *rear*, para que una lista representara una cola. Sin embargo, ahora sólo es necesario un apuntador externo, *q*, al nodo cabecera de la lista. *next(q)* apunta al frente de la cola e *info(q)* al final.

Otra posibilidad para el uso de la porción *info* de la cabecera de una lista es como apuntador a un nodo “vigente” de la lista durante el proceso de recorrido. Esto eliminaría la necesidad de un apuntador externo durante el recorrido.

## EJERCICIOS

- 4.2.1. Escriba un conjunto de rutinas para implantar varias colas y pilas dentro de un solo arreglo.
- 4.2.2. ¿Cuáles son las ventajas y cuáles las desventajas de representar un grupo de elementos como un arreglo en oposición a una lista lineal ligada?
- 4.2.3. Escriba un algoritmo para ejecutar cada una de las siguientes operaciones:
  - a. Agregar un elemento al final de una lista.
  - b. Concatenar dos listas.
  - c. Liberar todos los nodos de una lista.
  - d. Invertir una lista de manera que el último elemento se convierta en el primero, y así sucesivamente.
  - e. Elimine el último elemento de una lista.
  - f. Elimine el *n*-ésimo elemento de una lista.
  - g. Combine dos listas ordenadas en una sola que también esté ordenada.
  - h. Forme una lista que contenga la unión de los elementos de dos listas.
  - i. Forme una lista que contenga la intersección de los elementos de dos listas.
  - j. Insertar un elemento después del *n*-ésimo de una lista.
  - k. Elimine todos los segundos elementos de una lista.
  - l. Ponga los elementos de una lista en orden ascendente.
  - m. Dé como resultado la suma de los enteros de una lista.
  - n. Dé como resultado el número de elementos de una lista.
  - o. Mueva *node(p)* *n* posiciones hacia adelante de una lista.
  - p. Haga una segunda copia de una lista.

- 4.2.4. Escriba algoritmos para ejecutar cada una de las operaciones anteriores sobre un grupo de elementos en posiciones contiguas de un arreglo.
- 4.2.5. ¿Cuál es el número promedio de nodos accesados cuando se busca un elemento particular en una lista desordenada? ¿En una ordenada? ¿En un arreglo desordenado? ¿En uno ordenado?
- 4.2.6. Escriba algoritmos para *pqinsert* y *pqmindelete* para una cola de prioridad ascendente implantada como una lista desordenada y como una ordenada.
- 4.2.7. Escriba algoritmos para ejecutar cada una de las operaciones del ejercicio 4.2.3, suponiendo que cada lista contiene un nodo cabecera con el número de elementos de la lista.
- 4.2.8. Escriba un algoritmo que dé como resultado un apuntador a un nodo que contiene un elemento *x* en una lista con un nodo cabecera. El campo *info* de la cabecera deberá contener el apuntador que recorre la lista.

## 4.3. LISTAS EN C

### Implantación con arreglo de listas

¿Cómo pueden representarse listas lineales en C? Como una lista no es más que una colección de nodos, un arreglo de nodos se sugiere a sí mismo de inmediato. Sin embargo, es posible que los nodos no estén ordenados de acuerdo con el orden del arreglo; cada uno de ellos debe contener en sí mismo un apuntador a su sucesor. Así, un grupo de 500 nodos podría declararse como un arreglo *node* del modo siguiente:

```
#define NUMNODES 500
struct nodetype {
 int info, next;
};
struct nodetype node[NUMNODES];
```

En este esquema, un apuntador a un nodo se representa mediante un índice del arreglo. Un apuntador es un entero entre 0 y *NUMNODES* —1 que alude a un elemento particular del arreglo *node*. El apuntador nulo se representa por el entero —1. En esta implantación, la expresión *node[p]* en C, se usa para eludir *node(p)*; *info(p)* se alude mediante *node[p].info*, y *next(p)* son referenciadas por *node[p].next*. *null* está representado por —1.

Por ejemplo, supóngase que la variable *list* representa un apuntador a una lista. Si *list* tiene el valor 7, *node[7]* es el primer nodo de la lista, y *node[7].info* es el primer elemento de datos. El segundo nodo de la lista está dado por *node[7].next*. Supóngase que *node[7].next* es igual a 385. Entonces *node[385].info* es el segundo elemento de datos en la lista y *node[385].next* apunta al tercer nodo.

Los nodos de una lista pueden estar dispersos en todo el arreglo *node* en cualquier orden. Cada nodo lleva dentro de sí la ubicación de su sucesor hasta el último nodo de la lista, cuyo campo *next* contiene —1, que es el apuntador nulo. No hay relación entre los contenidos de un nodo y el apuntador a éste. El apuntador, *p*, a un

nodo sólo especifica qué elemento del arreglo *node* se referencia; *node[p].info* es quien representa la información contenida dentro de ese nodo.

La figura 4.3.1 ilustra una porción de un arreglo *node* que contiene cuatro listas ligadas. La lista *list1* comienza en *node[16]* y contiene los enteros 3, 7, 14, 6, 5, 37, 12. Los nodos que contienen estos enteros en su campo *info* están dispersos a lo largo del arreglo. El campo *next* de cada nodo contiene el índice dentro del arreglo del nodo que contiene el siguiente elemento de la lista. El último nodo de la lista es *node[23]*, el que contiene el entero 12 en su campo *info* y el apuntador nulo ( $-1$ ) en su campo *next*, para indicar que es el último de la lista.

De manera análoga, *list2* comienza en el *node[4]* y contiene los enteros 17 y 26; *list3* comienza en *node[11]* y contiene a 31, 19 y 32; y *list4* comienza en *node[3]* y contiene los enteros 1, 18, 13, 11, 4 y 15. Las variables *list1*, *list2*, *list3* y *list4* son enteros que representan apuntadores externos a las cuatro listas. Así, el hecho de que la variable *list2* tenga el valor 4 represente el hecho de que la lista a la cual apunta ésta comience en *node[4]*.

Al principio no se usa ningún nodo, ya que no se han formado aún las listas. En consecuencia, todos tienen que ser colocados en la lista disponible. Si la variable

|    | info | sig |
|----|------|-----|
| 0  | 26   | -1  |
| 1  | 11   | 9   |
| 2  | 5    | 15  |
| 3  | 1    | 24  |
| 4  | 17   | 0   |
| 5  | 13   | 1   |
| 6  |      |     |
| 7  | 19   | 18  |
| 8  | 14   | 12  |
| 9  | 4    | 21  |
| 10 |      |     |
| 11 | 31   | 7   |
| 12 | 6    | 2   |
| 13 |      |     |
| 14 |      |     |
| 15 | 37   | 23  |
| 16 | 3    | 20  |
| 17 |      |     |
| 18 | 32   | -1  |
| 19 |      |     |
| 20 | 7    | 8   |
| 21 | 15   | -1  |
| 22 |      |     |
| 23 | 12   | -1  |
| 24 | 18   | 5   |
| 25 |      |     |
| 26 |      |     |

Figura 4.3.1 Un arreglo de nodos que contiene cuatro listas ligadas.

global *avail* se usa para apuntar a la lista disponible, esta lista se puede organizar inicialmente de la siguiente manera:

```
avail = 0;
for (i = 0; i < NUMNODES-1; i++)
 node[i].next = i + 1;
node[NUMNODES-1].next = -1;
```

Los 500 nodos están ligados al principio en su orden natural, de manera que *node[i]* apunta a *node[i + 1]*. *node[0]* es el primer nodo de la lista disponible; *node[1]* el segundo, y así sucesivamente. *node[499]* es el último de la lista, ya que *node[499].next* es igual a  $-1$ . No hay otra razón para este orden inicial que la conveniencia. También se pudo haber puesto *node[0].next* como 499; *node[499].next* como 1, *node[1].next* como 498, y así sucesivamente, hasta asignar *node[249].next* a 250 y *node[250].next* a  $-1$ . El aspecto importante es que el ordenamiento resulta explícito dentro de los propios nodos y no como consecuencia de otra estructura subyacente.

Para el resto de las funciones de esta sección, puede suponerse que las variables *node* y *avail* son globales y que cualquier rutina puede, por lo tanto, usarlas.

Cuando se necesita un nodo para aplicarlo en una lista particular, éste se obtiene de la lista disponible. De manera análoga, cuando un nodo ya no se necesita, se devuelve a la lista disponible. Estas dos operaciones están implantadas por medio de las rutinas en C *getnode* y *freenode*; *getnode* es una función que elimina un nodo de la lista disponible y regresa un apuntador hacia él:

```
getnode()
{
 int p;
 if (avail == -1) {
 printf("desborde \n");
 exit(1);
 }
 p = avail;
 avail = node[avail].next;
 return(p);
} /* fin de getnode */
```

Si *avail* es igual a  $-1$  cuando se llama a esta función, no hay nodos disponibles. Esto significa que las estructuras de la lista de un programa particular rebasaron el espacio disponible.

La función *freenode* acepta un apuntador a un nodo y devuelve ese nodo a la lista disponible:

```
freenode(p)
int p;
{
}
```

```

node[p].next = avail;
avail = p;
return;
} /* fin de freenode */

```

Las operaciones originales para listas son versiones directas en C de los algoritmos correspondientes. La rutina *insafter* acepta un apuntador *p* a un nodo y un elemento *x* como parámetros. Primero se asegura que *p* no sea nulo y luego inserta *x* dentro del nodo que sigue al apuntado por *p*:

```

insafter(p, x)
int p, x;
{
 int q;
 if (p == -1) {
 printf("inserción no efectuada/n");
 return;
 }
 q = getnode();
 if (q == -1) {
 printf("inserción no efectuada/n");
 return;
 }
 node[q].info = x;
 node[q].next = node[p].next;
 node[p].next = q;
 return;
} /* fin de insafter */

```

La rutina *delafter(p, px)*, llamada mediante la instrucción *delafter(p, &p)*, elimina el nodo que sigue a *node(p)* y almacena su contenido en *x*:

```

delafter(p, px)
int p, *px;
{
 int q;
 if ((p == -1) || (node[p].next == -1)) {
 printf("eliminación no efectuada/n");
 return;
 }
 q = node[p].next;
 *px = node[q].info;
 node[p].next = node[q].next;
 freenode(q);
 return;
} /* fin de delaftter */

```

Antes de llamar a *insafter* hay que asegurarse de que *p* no es nulo. Antes de llamar a *delafter* hay que asegurarse de que ni *p* ni *node[p].next* lo son.

## Limitaciones de la implantación con arreglo

Como se vio en la sección 4.2, la noción de un apuntador permite construir y manipular listas ligadas de varios tipos. El concepto de apuntador introduce la posibilidad de agrupar una colección de bloques construidos, llamados nodos, dentro de estructuras flexibles. Alterando los valores de los apuntadores, los nodos pueden conectarse, desconectarse y agruparse otra vez en configuraciones que crecen y se contraen mientras progresan la ejecución de un programa.

En la implantación con arreglo, se establece un conjunto fijo de nodos mediante un arreglo al principio de la ejecución. Un apuntador a un nodo se representa por medio de la posición relativa del nodo dentro del arreglo. La desventaja de este enfoque es doble. Primero, con frecuencia no puede calcularse el número de nodos que se necesita cuando se escribe el programa. Los datos con los que se ejecuta el programa a menudo determinan el número de nodos necesario. Así, no importa cuántos elementos contiene el arreglo de nodos, siempre es posible que el programa se ejecute con datos que requieran una mayor cantidad.

La segunda desventaja del enfoque con arreglo es que cualquier número de nodos que se declare debe permanecer asignado al programa a lo largo de la ejecución. Por ejemplo, cuando se declaran 500 nodos de determinado tipo, se reserva la cantidad de memoria que éstos requieren para este propósito. Si el programa sólo usa 100, o incluso 10, en su ejecución, los nodos adicionales permanecen en reserva y la memoria que les corresponde no puede usarse para otro propósito.

La solución a este problema es permitir que los nodos sean *dinámicos* en lugar de estáticos. Es decir, siempre que se necesita un nodo, se reserva memoria para él, y cuando se deja de ocupar, se libera la memoria. Así, la memoria para los nodos que ya no están en uso, puede usarse en algún otro propósito. Tampoco se establece un límite predefinido sobre el número de nodos. Siempre que esté disponible la memoria suficiente para la tarea como un todo, se puede reservar parte de dicha memoria para su uso como nodo.

## Asignación y liberación de variables dinámicas

En las secciones 1.1, 1.2 y 1.3 se examinaron los apuntadores en el lenguaje C. Si *x* es un objeto, *&x* es un apuntador a *x*. Si *p* es un apuntador en C, *\*p* es el objeto al que apunta *p*. Se pueden usar apuntadores en C para ayudar a implantar listas ligadas dinámicas. Sin embargo, primero se analizará cómo se puede asignar y liberar memoria de manera dinámica y cómo se accesa en C la memoria dinámica.

En C puede crearse una variable apuntadora a un entero mediante la declaración:

```
int *p;
```

Una vez que se declara la variable *p* como apuntador a un tipo de objeto específico, se tiene la capacidad de crear un objeto de ese tipo específico de manera dinámica y asignar su dirección a *p*.

En C, esto puede hacerse llamando a la función `malloc(size)` de la biblioteca estándar. `malloc` asigna una porción de memoria de manera dinámica de *tamaño size* y da como resultado un apuntador a un elemento de tipo *char*. Considérese las declaraciones

```
extern char *malloc();
int *pi;
float *pr;
```

### Las instrucciones

Las instrucciones

```
pi = (int *) malloc(sizeof(int));
pr = (float *) malloc(sizeof(float));
```

crean de manera dinámica la variable entera *\*pi* y la de punto flotante *\*pr*. Estas variables se llaman *variables dinámicas*. En la ejecución de estas instrucciones, el operador `sizeof` da como resultado el *tamaño*, en bytes, de su operando. Este se usa para mantener la independencia de la máquina. `malloc` puede entonces crear un objeto de ese tamaño. Así, `malloc(sizeof(int))` asigna memoria para un entero, mientras que `malloc(sizeof(float))` lo hace para un número de punto flotante. `malloc` también da como resultado un apuntador a la memoria que asigna. Este apunta al primer byte (por ejemplo, carácter) de esa memoria y es de tipo *char\**. Para obligar a que este apuntador apunte a un entero o real, se usa el operador de cambio de tipo (*int\**) o (*float\**).

(El operador `sizeof` da como resultado un valor de tipo (*int*), mientras que la función `malloc` espera un parámetro de tipo *sin signo*. Para que el programa esté “limpio”, debe escribirse:

```
pi = (int *) malloc ((unsigned)(sizeof (int)));
```

Sin embargo, con frecuencia se omite el cambio de tipo sobre el operador `sizeof`.)

Para ejemplificar el uso de la función `malloc` y de los apuntadores, considérese las siguientes instrucciones:

```
1 int *p, *q;
2 int x;
3 int main()
4 {
5 p = (int *) malloc(sizeof(int));
6 *p = 3; /* Inicializa el espacio dinámico */
7 q = p;
8 *q = x;
9 printf("%d %d\n", *p, *q);
10 p = (int *) malloc(sizeof(int));
11 *p = 5; /* Reasigna el espacio dinámico */
12 printf("%d %d\n", *p, *q);
```

En la línea 3 se crea una variable entera y su dirección se pone en *p*. La línea 4 asigna a 3 al valor de esa variable. La 5 asigna a *q* la dirección de la misma. La instrucción de asignación en la línea 8 es perfectamente válida, ya que a una variable apuntador (*q*) se le está asignando el valor de otra (*p*). La figura 4.3.2a ilustra la situación después de la línea 5. Obsérvese que, a partir de ese momento, *\*p* y *\*q* se refieren a la misma variable. La línea 6, por lo tanto, imprime el contenido de dicha variable (que es 3) dos veces.

La línea 7 asigna valor 7 a una variable de tipo entera, *x*. La 8 cambia el valor de *\*q* a *x*. Sin embargo, como *p* y *q* son apuntadores a la misma variable, tanto *\*p* como *\*q* tienen el valor 7. En la figura 4.3.2b se ilustra esto. Por lo tanto, la línea 9 imprime dos veces el número 7.

La línea 10 crea una nueva variable entera y pone su dirección en *p*. En la figura 4.3.2c se ilustran los resultados. *\*p* se refiere ahora a la variable entera recién creada, a la que aún no se le asigna ningún valor. *q*, no ha sido cambiada; por lo tanto, el valor de *\*q* sigue siendo 7. Adviértase que *\*p* no se refiere a una variable simple específica; su valor cambia como el de *p*. La línea 11 asigna el valor 5 a esa variable recién creada, como se ilustra en la figura 4.3.2d, y la 12 imprime los valores 5 y 7.

La función `free` se usa en C para liberar memoria de una variable asignada de manera dinámica. La instrucción

```
free(p);
```

libera la memoria que apunta el apuntador *p*. La figura 4.3.2a ilustra la situación inicial. La figura 4.3.2b muestra la situación después de la ejecución de la instrucción *\*q = x;*. La figura 4.3.2c muestra la situación después de la ejecución de la instrucción *p = malloc(sizeof(int));*. La figura 4.3.2d muestra la situación después de la ejecución de la instrucción *\*p = 5;*.

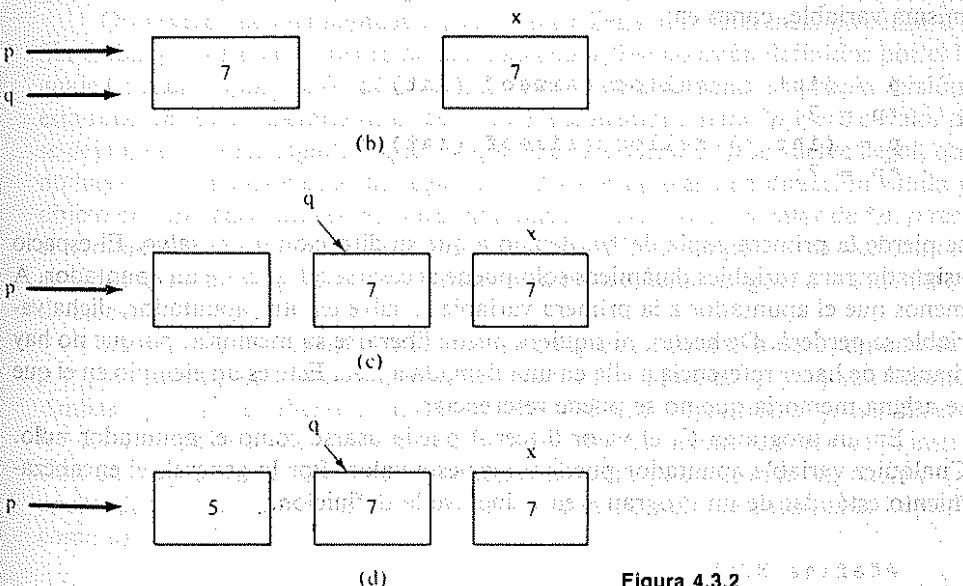


Figura 4.3.2

hace que cualquier referencia futura a la variable `*p` sea ilegal (a menos que, por supuesto, se asigne un nuevo valor a `p` mediante una instrucción de asignación o una llamada a `malloc`). Llamar a `free(p)` hace que la memoria ocupada por `*p` quede disponible para que se vuelva a usar si es necesario.

(Nota: La función `free` espera por omisión, un parámetro apuntador de tipo `char*`. Para que la instrucción sea “limpia” debe escribirse

```
free((char *) p);
```

Sin embargo, en la práctica se omite con frecuencia el cambio de tipo del parámetro.)

Para ilustrar el uso de la función `free`, considérese las siguientes instrucciones:

```
1 p = (int *) malloc (sizeof (int));
2 *p = 5;
3 q = (int *) malloc (sizeof (int));
4 *q = 8;
5 free(p);
6 p = q;
7 q = (int *) malloc (sizeof (int));
8 *q = 6;
9 printf("%d %d\n", *p, *q);
```

Se imprimen los valores 8 y 6. La figura 4.3.3a ilustra la situación después de la línea 4, donde se ha ubicado a `*p` y `*q` y se les ha asignado valor. La figura 4.3.3b muestra el efecto de la línea 5, en la que se ha liberado la variable a la que apunta `p`. La figura 4.3.3c ilustra la línea 6, en la que se cambia el valor de `p` para apuntar a la variable `*q`. En las líneas 7 y 8 se cambia el valor de `q` para apuntar a la variable recién creada a la que se da valor 6 en la línea 8 (figura 4.3.3d).

Adviértase que si se llama a `malloc` dos veces seguidas y se asigna su valor a la misma variable, como en:

```
p = (int *) malloc (sizeof (int));
*p = 3;
p = (int *) malloc (sizeof (int));
*p = 7;
```

se pierde la primera copia de `*p`, debido a que su dirección no se salvó. El espacio asignado para variables dinámicas sólo puede accesarse a través de un apuntador. A menos que el apuntador a la primera variable se salve en otro apuntador, dicha variable se perderá. De hecho, ni siquiera puede liberarse su memoria, porque no hay manera de hacer referencia a ella en una llamada a `free`. Este es un ejemplo en el que se asigna memoria que no se puede referenciar.

En un programa C, el valor 0 (cero) puede usarse como el apuntador nulo. Cualquier variable apuntador puede tomar este valor. Por lo general, el encabezamiento estándar de un programa en C incluye la definición

```
#define NULL 0
```

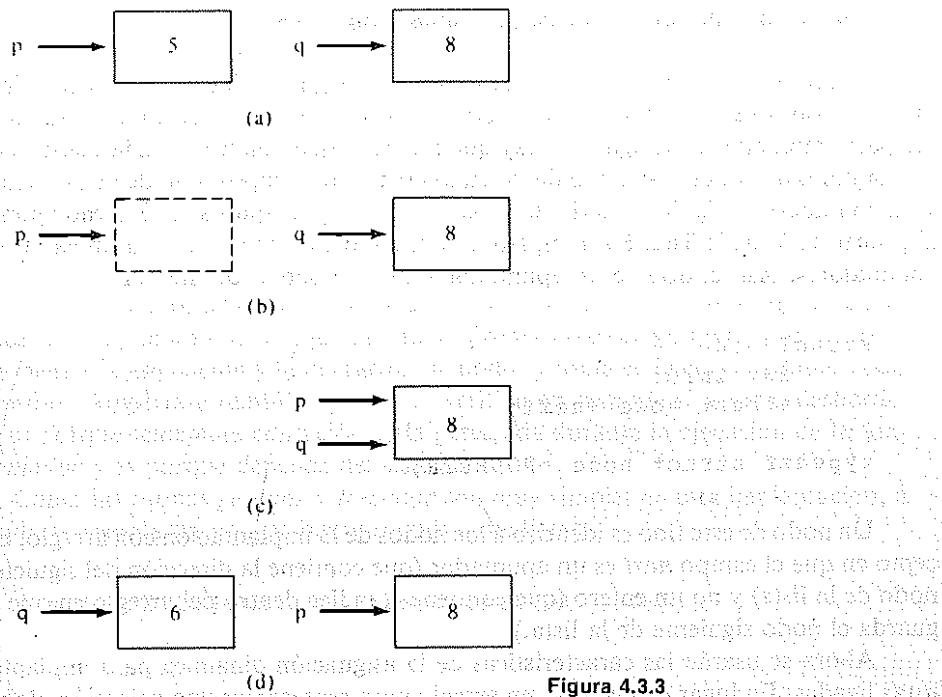


Figura 4.3.3

para permitir que el valor cero de un apuntador se escriba como `NULL`. Este valor de apuntador `NULL` no hace referencia a una localidad de memoria, sino que denota el apuntador que no apunta a nada. El valor `NULL` (cero) se puede asignar a cualquier variable apuntador `p`, luego de lo cual es ilegal una referencia a `*p`.

Obsérvese que una llamada a `free(p)` hace ilegal una referencia posterior a `*p`. Sin embargo, los efectos reales de una llamada a `free` no están definidos por el lenguaje C; cada implantación de C tiene la libertad de desarrollar su propia versión de esta función. En la mayoría de éstas, se libera la memoria para `*p`, pero el valor de `p` se deja intacto. Esto significa que aunque una referencia a `*p` se vuelve ilegal, quizás no haya manera de detectar la ilegalidad. El valor de `p` es una dirección válida y el objeto del tipo adecuado en esa dirección puede usarse como el valor de `*p`. `p` recibe el nombre de *apuntador inconexo*. Al programador le corresponde no usar nunca tal apuntador en un programa. Una buena costumbre es hacer `p` igual a `NULL` después de ejecutar `free(p)`.

Debe mencionarse otra característica peligrosa asociada a los apuntadores. Si `p` y `q` son dos apuntadores con el mismo valor, las variables `*p` y `*q` son idénticas. Ambas, `*p` y `*q`, se refieren al mismo objeto. Así, una asignación a `*p` cambia el valor de `*q`, a pesar de que ni `q` ni `*q` estén mencionadas explícitamente en la instrucción de asignación a `*p`. Al programador le corresponde tomar en cuenta “qué apuntadores apuntan a qué” y reconocer la ocurrencia de resultados implícitos como los mencionados.

## Listas ligadas por medio de variables dinámicas

Ahora que ya se cuenta con la capacidad de asignar y liberar de manera dinámica una variable, a continuación se verá cómo pueden usarse las variables dinámicas para implantar listas ligadas. Hay que recordar que una lista ligada consiste en un conjunto de nodos, cada uno de los cuales tiene dos campos: uno de información y un apuntador al siguiente nodo de la lista. Además, un apuntador externo apunta al primer nodo de la lista. Para implantar apuntadores a una lista, se usan variables apuntadoras. Así, el tipo de un apuntador y de un nodo se define por

```
struct node {
 int info;
 struct node *next;
};

typedef struct node *NODEPTR;
```

Un nodo de este tipo es idéntico a los nodos de la implantación con arreglo, excepto en que el campo *next* es un apuntador (que contiene la dirección del siguiente nodo de la lista) y no un entero (que contiene el índice dentro del arreglo en que se guarda el nodo siguiente de la lista).

Ahora se usarán las características de la asignación dinámica para implantar listas ligadas. En lugar de declarar un arreglo para representar una colección global de nodos, éstos se asignan y liberan cuando es necesario. Así se elimina la necesidad de una colección declarada de nodos.

Si se declara

```
NODEPTR p;
```

entonces el comando *malloc* devolverá un apuntador a memoria dinámica que apunta a un espacio de memoria que contiene una estructura de tipo *node*. El resultado es que si se ejecuta la instrucción

```
p = (NODEPTR) malloc(sizeof(struct node));
```

se obtiene un apuntador *p* que apunta a la memoria dinámica que contiene la estructura de tipo *node*. La ejecución de la instrucción

```
free(p);
```

libera la memoria dinámica que contiene la estructura de tipo *node* apuntada por *p*. La ejecución de la instrucción

```
NODEPTR p;
```

devuelve el valor *NULL*, que indica que no hay memoria dinámica asignada a *p*.

Adviértase que *malloc* regresa la dirección de la memoria dinámica que se asignó, pero no coloca la dirección de un nodo disponible en *p*. Ahora se presenta la función *getnode*:

```
NODEPTR getnode()
{
 NODEPTR p;
 p = (NODEPTR) malloc(sizeof(struct node));
 return(p);
}
```

Adviértase que *sizeof* se aplica a un tipo de estructura y regresa el número de bytes requerido para toda la estructura.

La ejecución de la instrucción

```
freenode(p);
```

deberá regresar el nodo cuya dirección está en *p* a la memoria disponible. Ahora se presenta la rutina *freenode*:

```
freenode(p)
NODEPTR p;
{
 free(p);
}
```

El programador no debe ocuparse del manejo de la memoria disponible. Ya no se necesita el apuntador *avail* (que apunta al primer nodo disponible), pues el sistema gobierna la asignación y la liberación de nodos y toma en cuenta el primer nodo disponible. Adviértase también que no se verifica en *getnode* si ocurrió desborde. Esto es porque semejante condición sería detectada durante la ejecución de la función *malloc* y es porque depende del sistema.

Como las rutinas *getnode* y *freenode* son muy simples en esta implantación, a menudo se reemplazan por las instrucciones inmediatas

```
p = (NODEPTR) malloc(sizeof(struct node));
```

y

```
free(p);
```

A continuación se presentan los procedimientos *insafter(p, x)* y *delafter(p, px)* por medio de la implantación dinámica de una lista ligada. Supóngase que *list* es una variable apuntador que apunta al primer nodo de una lista (si lo hay) y es igual a *NULL* en caso de que la lista esté vacía.

```
insafter(p, x)
NODEPTR p;
int x;
{
 NODEPTR q;
 if (p == NULL) {
 printf("inserción no efectuada");
 exit(1);
 }
 q = getnode();
 q->info = x;
 q->next = p->next;
 p->next = q;
} /* fin de insafter */

delafter(p, px)
NODEPTR p;
int *px;
{
 if (p == NULL) {
 printf("eliminación no efectuada");
 exit(1);
 }
 if (*px == p) {
 *px = p->next;
 free(p);
 }
}
```

```

NODEPTR q;
if ((p == NULL) || (p -> next == NULL)) {
 printf("supresión nula/n");
 exit(1);
}
q = p -> next;
*px = q -> info;
p -> next = q -> next;
freenode(q);
} /* fin de delafter */

```

Obsérvese la sorprendente similitud entre las rutinas anteriores y las de la implantación con arreglo, las que se presentaron previamente en esta sección. Ambas son implantaciones de los algoritmos de la sección 4.2. De hecho, la única diferencia entre las dos versiones es la manera en que se hace referencia a los nodos.

### Colas como listas en C

Para ilustrar con más claridad la forma en que se usan las implantaciones con listas en C, se presentan rutinas en C para la manipulación de una cola representada como una lista lineal. A manera de ejercicio, se le dejan al lector las rutinas para manipular una pila y una cola de prioridad. Para propósitos de comparación se muestran tanto la implantación con arreglo como la dinámica. Puede suponerse que *struct node* y *NODEPTR* fueron declarados igual que arriba. Una cola se representa como una estructura:

#### Implantación con arreglo

```

struct queue {
 int front, rear;
};
struct queue q;

```

*front* y *rear* son apuntadores al primero y al último nodos de una cola representada como una lista. *font* y *rear*, que igualan al apuntador nulo, representan la cola vacía. La función *empty* necesita verificar sólo uno de esos apuntadores, pues en una cola no vacía ni *front* ni *rear* serán *NULL*.

```

empty(pq)
struct queue *pq;
{
 return ((pq->front ==
 -1) ? TRUE: FALSE);
} /* fin de empty */

```

La rutina para insertar un elemento dentro de una cola puede escribirse como sigue:

```

insert(pq, x)
struct queue *pq;
int x;
{
 int p;
 p = getnode();
 node[p].info = x;
 node[p].next = -1;
 if (pq->rear == -1)
 pq->front = p;
 else
 node[pq->rear].next = p;
 pq->rear = p;
} /* fin de insert */

```

La función *remove* elimina el primer elemento de la cola y devuelve su valor:

```

remove(pq)
struct queue *pq;
{
 int p, x;
 NODEPTR p;
 int x;
 if (empty(pq)) {
 printf
 ("subdesborde de la cola\n");
 exit(1);
 }
 p = pq->front;
 x = node[p].info;
 pq->front = node[p].next;
 if (pq->front == -1)
 pq->rear = -1;
 freenode(p);
 return(x);
} /* fin de remove */

```

### Ejemplos de operaciones con listas en C

Enseguida se verán algunas operaciones con lista un poco más complejas implantadas en C. Se ha observado que la implantación dinámica es, por lo general, mejor que la implantación con arreglo. Por ello, la mayoría de los programadores de C la usan para implantar listas. A partir de este momento habrá que apegarse a la implantación dinámica de listas ligadas, aunque también puede aludirse a la implantación con arreglo cuando es apropiado.

Ya se definió con anterioridad la operación *place(list, x)* donde *list* apunta a una lista lineal ordenada y *x* es un elemento que debe insertarse en la posición correspondiente de la lista. Hay que recordar que esta operación se usa para implantar la

operación *pqinsert* para insertar en una cola de prioridad. Supóngase que ya se ha implantado la operación *push* para pilas. El código para implantar la operación *place* es:

```
place(plist, x)
NODEPTR *plist;
int x;
{
 NODEPTR p, q;
 q = NULL;
 for (p = *plist; p != NULL && x > p->info; p = p->next)
 q = p;
 if (q == NULL) /* insertar x a la cabeza de la lista */
 push(plist, x);
 else
 insafter(q, x);
} /* fin de place */
```

Adviértase que *plist* debe declararse como un apuntador al apuntador de la lista, ya que el valor del apuntador externo a la lista cambia si *x* se inserta al frente de la misma por medio de la rutina *push*. La rutina anterior se llamaría por medio de la instrucción *place(&list, x)*.

Como segundo ejemplo, hay que escribir una función *insend(plist, x)* para insertar el elemento *x* al final de una lista *list*:

```
insend(plist, x)
NODEPTR *plist;
int x;
{
 NODEPTR p, q;
 p = getnode();
 p->info = x;
 p->next = NULL;
 if (*plist == NULL)
 *plist = p;
 else {
 /* búsqueda del último nodo */
 for (q = *plist; q->next != NULL; q = q->next)
 ;
 q->next = p;
 } /* fin de if */
} /* fin de insend */
```

Ahora se presenta una función *search(list, x)* que da como resultado un apuntador a la primera ocurrencia de *x* en la lista *list* y el apuntador *NULL* si no existe *x* en la misma:

```
NODEPTR search(list, lx)
NODEPTR list;
int lx;
```

```
int x;
{
 NODEPTR p;
 for (p = list; p != NULL; p = p->next)
 if (p->info == x)
 return (p);
 /* x no está en la lista */
 return (NULL);
} /* fin de search */
```

La siguiente rutina elimina todos los nodos cuyo campo *info* contenga el valor *x*:

```
remvx(plist, x)
NODEPTR *plist;
int x;
{
 NODEPTR p, q;
 int y;
 q = NULL;
 p = *plist;
 while (p != NULL) {
 if (p->info == x) {
 if (q == NULL) { /* eliminar el primer nodo de la lista */
 freenode(*plist);
 *plist = p;
 }
 else
 delaftter(q, &y);
 }
 else
 /* avanzar al siguiente nodo de la lista */
 q = p;
 p = p->next;
 } /* fin de if */
} /* fin de remvx */
```

### Listas no enteras y no homogéneas

Por supuesto, un nodo de una lista no debe representar por fuerza a un entero. Por ejemplo, para representar una pila de cadenas de caracteres mediante una lista ligada, se necesitan nodos que contengan cadenas de caracteres en sus campos *info*. A los nodos de ese tipo, que usan la implantación de asignación dinámica, se los puede declarar por medio de:

```
struct node {
 /* ... */
 char info[100];
};
```

```

 struct node *next;
}

```

Una aplicación particular puede requerir de nodos que contengan más de un elemento de información. Por ejemplo, cada nodo de estudiante en una lista de estudiantes puede contener la siguiente información: nombre del estudiante, número de identificación del colegio, dirección, índice de calificaciones y especialidad. Los nodos para una aplicación como ésta pueden declararse como sigue:

```

struct node {
 char name[30];
 char id[9];
 char address[100];
 float gpinde;
 char major[20];
 struct node *next;
};

```

Para manipular listas que contengan cada tipo de nodos, hay que escribir un conjunto de rutinas distintas en C.

Para representar listas no homogéneas (aquellas que contienen nodos de tipos diferentes), puede usarse una unión. Por ejemplo,

```

#define INTGR 1
#define FLT 2
#define STRING 3
struct node {
 int etype /* etype será INTGR, FLT o STRING
 dependiendo del tipo del
 elemento correspondiente. */;
 union {
 int ival;
 float fval;
 char *pval; /* apuntador a una cadena */
 } element;
 struct node *next;
};

```

define un nodo cuyos elementos pueden ser números enteros o de punto-flotante o cadenas, dependiendo del valor correspondiente de *etype*. Como una unión siempre es lo suficientemente grande para guardar su componente mayor, se pueden usar las funciones *sizeof* y *malloc* para asignar memoria al nodo. Por lo tanto, las funciones *getnode* y *freenode* permanecen intactas. Por supuesto, al programador le corresponde usar de manera apropiada los componentes de un nodo. En resumen en el resto de esta sección puede suponerse que una lista ligada es declarada para que tenga sólo elementos homogéneos (por lo que las uniones son innecesarias). En la sección

9.1 se examinan listas no homogéneas, incluyendo listas que contienen otras listas y listas recursivas.

### Comparación de la implantación dinámica y con arreglo de listas

Resulta ilustrativo examinar las ventajas y las desventajas de las implantaciones dinámica y con arreglo de listas ligadas. La principal desventaja de la implantación dinámica es que puede consumir más tiempo recurrir al sistema para asignar y liberar memoria que manipular una lista disponible manejada por el programador. Su mayor ventaja es que no se reserva con anticipación un conjunto de nodos que será usado por un grupo particular de listas.

Por ejemplo, supóngase que un programa usa dos tipos de listas: listas de enteros y listas de caracteres. En la implantación con arreglo se asignarían de inmediato dos arreglos de tamaño fijo. Cuando un grupo de listas sobrepasa su arreglo, el programa no puede proseguir. En la representación dinámica, dos tipos de nodos se definen al principio, pero no se asigna memoria a las variables hasta que se necesite. Cuando esto ocurre, se llama al sistema para que proporcione los nodos. Toda memoria que no se usa en un tipo de nodo, puede usarse en otro. Así, no ocurre desborde si siempre que haya memoria disponible para los nodos presentes de verdad en la lista.

Otra ventaja de la implantación dinámica es que una referencia a *\*p* no implica el cálculo de la dirección necesario para calcular la dirección de *node[p]*. Para calcular la dirección de *node[p]*, se deben agregar los contenidos de *p* a la dirección base del arreglo *node*, mientras que la dirección de *\*p* está determinada de manera directa por los contenidos de *p*.

### Implantación de nodos cabecera

Al final de la sección anterior se introdujo el concepto de nodos cabecera que pueden contener información global acerca de la lista, tal como su longitud o un apuntador al nodo actual o último de la lista. Cuando el tipo de datos de los contenidos de la cabecera es idéntico al de los contenidos de los nodos de la lista, puede implantarse la cabecera sólo como otro nodo al principio de la lista.

También es posible declarar los nodos cabecera como variables independientes del conjunto de los nodos de la lista. Esto es particularmente útil cuando la información que contiene la cabecera es diferente a los datos de los nodos de la lista. Por ejemplo, considérese el siguiente conjunto de declaraciones:

```

struct node {
 char info;
 struct node *next;
};
struct charstr {
 int length;
 struct node *firstchar;
};
struct charstr s1, s2;

```

Las variables *s1* y *s2* de tipo *charstr* son nodos cabecera para una lista de caracteres. La cabecera contiene el número de caracteres de la lista (*length*) y un apuntador a la misma (*firstchar*). Así, *s1* y *s2* representan cuerdas de caracteres de longitud variable. A manera de ejercicio, se pueden escribir rutinas para concatenar dos cadenas de caracteres de este tipo o para extraer una subcadena de tal cadena.

## EJERCICIOS

- 4.3.1. Implante las rutinas *empty*, *push*, *pop* y *popandtest* por medio de la implantación con arreglo y la implantación de memoria dinámica de una pila ligada.
- 4.3.2. Implante las rutinas *empty*, *insert* y *remove* mediante una implantación de memoria dinámica de una cola ligada.
- 4.3.3. Implante las rutinas *empty*, *pqinsert* y *pqmindelete* mediante una implantación de memoria dinámica de una cola de prioridad ligada.
- 4.3.4. Escriba rutinas en C para implantar las operaciones del ejercicio 4.2.3 por medio de las implantaciones con arreglo y de memoria dinámica de una lista ligada.
- 4.3.5. Escriba una rutina en C para intercambiar el *n*-ésimo y el *m*-ésimo elemento de una lista.
- 4.3.6. Escriba una rutina *inssub*(*l1*, *i1*, *l2*, *i2*, *len*) para insertar elementos de la lista *l2*; comenzar con el *i2*-ésimo y continuar para *len* elementos dentro de la lista *l1* empezando en la posición *i1*. Ningún elemento de la lista *l1* debe eliminarse ni remplazarse. Si *i1* > *length(l1)* + 1 (donde *length(l1)* denota el número de nodos de la lista *l1*) o si *i2* + *len* - 1 > *length(l2)*, o si *i1* < 1, o si *i2* < 1, imprime un mensaje de error. La lista *l2* deberá permanecer intacta.
- 4.3.7. Escriba una función *search*(*l*, *x*) que acepte un apuntador *l* a una lista de enteros y un entero *x* y que dé como resultado un apuntador a un nodo que contenga *x*, en caso de que exista, o en caso contrario el apuntador nulo. Escriba otra función, *srchinsrt*(*l*, *x*), que agregue *x* a *l* si no lo encuentra y que siempre regrese un apuntador al nodo que contiene *x*.
- 4.3.8. Escriba un programa en C para leer un grupo de líneas de entrada, cada una de las cuales debe contener una palabra. Imprima cada palabra que aparezca en la entrada y el número de veces que esto ocurre.
- 4.3.9. Suponga que una cadena de caracteres se representa mediante una lista de caracteres simples. Escriba un conjunto de rutinas para manipular tales listas de la siguiente manera (en adelante *l1*, *l2* y *list* son apuntadores a un nodo cabecera de una lista que representa una cadena de caracteres, *str* es un arreglo de caracteres, e *i1* e *i2* son enteros):
  - a. *strcnvcl*(*str*) para convertir la cadena de caracteres *str* a una lista. Esta función da como resultado un apuntador a un nodo cabecera.
  - b. *strcnvlc*(*list*, *str*) para convertir una lista a una cadena de caracteres.
  - c. *strpsl*(*l1*, *l2*) para ejecutar la función *strpos* de la sección 1.2 sobre dos cadenas de caracteres representadas por listas. Esta función da como resultado un entero.
  - d. *strrvfyl*(*l1*, *l2*) para determinar la primera posición de la cadena representada por *l1* que no está contenida en la representada por *l2*. Esta función da como resultado un entero.
  - e. *strsbstr*(*l1*, *i1*, *i2*) para ejecutar la función *substr* de la sección 1.2 sobre una cadena de caracteres representada por la lista *l1* y los enteros *i1*, *i2*. Esta función da

como resultado un apuntador al nodo cabecera de una lista que representa una cadena de caracteres, que es la subcadena deseada. La lista *l1* permanece intacta.

- f. *strpsbl*(*l1*, *i1*, *i2*, *l2*) para ejecutar una seudoinstrucción *substr* para una lista *l1*. Los elementos de la lista *l2* remplazarán los *i2* elementos de *l1*, comenzando en la posición *i1*. La lista *l2* permanecerá intacta.
- g. *strcmpl*(*l1*, *l2*) para comparar dos cadenas de caracteres representadas por listas. Esta función da como resultado -1 si la cadena de caracteres representada por *l1* es menor que la representada por *l2*, 0 si son iguales y 1 si la representada por *l1* es mayor.

- 4.3.10. Escriba una función *binsrch* que acepte dos parámetros: un arreglo de apuntadores a un grupo de números ordenados y un número único. La función debe usar la búsqueda binaria (véase sección 3.1) para dar como resultado un apuntador al número si éste está en el grupo. Cuando esto no es así, regresará el valor *NULL*.
- 4.3.11. Suponga que se desea formar *N* listas, donde *N* es una constante. Declare un arreglo de apuntadores *list* mediante:

```
#define N ...
struct node { ...
 int info;
 struct node *next;
};
typedef struct node *NODEPTR;
NODEPTR list [N];
```

Lea dos números de cada línea de entrada, en los que el primero será el índice de la lista en la que debe colocarse el segundo número en orden ascendente. Cuando no haya más líneas de entrada, imprima todas las listas.

## 4.4. UN EJEMPLO: SIMULACION POR MEDIO DE LISTAS LIGADAS

Una de las aplicaciones más útiles de las colas, las colas de prioridad y las listas ligadas es la *simulación*. Un programa de simulación tiene el propósito de modelar una situación del mundo real con el objeto de aprender algo de ella. Cada objeto y acción de la situación real tiene su contraparte en el programa. Si la simulación es exacta, esto es, si el programa refleja con éxito el mundo real, el resultado del programa deberá reflejar el resultado de las acciones que se están simulando. Así, es posible entender qué ocurre en una situación sin tener que observar en realidad su ocurrencia.

Veamos un ejemplo. Suponga que hay un banco con cuatro cajeros. Un cliente entra al banco en un instante específico (*t1*) y desea hacer una transacción con algún cajero. Se espera que la transacción tome cierto tiempo (*t2*) antes de completarse. Cuando un cajero está libre, procesa la transacción del cliente de inmediato, el que abandonará el banco una vez que se completa la transacción en el tiempo *t1* + *t2*. El tiempo que empleó el cliente en el banco es exactamente el mismo que duró la transacción (*t2*).

Sin embargo, es posible que ningún cajero esté libre; que todos estén dando servicio a clientes que llegaron antes. En ese caso, en cada ventanilla de cajero hay

una fila de espera. La cola para un cajero en particular puede constar de una sola persona —la que esté realizando una transacción en ese momento— o puede ser una cola muy larga. El cliente se dirige a la cola más corta y espera a que los clientes anteriores completen sus transacciones y abandonen el banco. En ese momento, el cliente puede comenzar a tratar sus asuntos. Deja el banco  $t_2$  unidades de tiempo después de haber alcanzado el frente de la cola. En este caso el tiempo empleado será  $t_2$  más el tiempo de espera en la cola.

Con semejante sistema, se desea calcular el tiempo promedio invertido por cliente en el banco. Una manera de hacerlo es pararse en la puerta del banco y preguntarle a los clientes que salen el momento en que llegaron, anotar el momento de su partida y restar el primero del segundo para luego tomar el promedio de todos los clientes. Sin embargo, esto sería poco práctico. No se podría asegurar si algún cliente abandonó el banco de manera inadvertida. Además, es dudoso que la mayoría de los clientes recuerde el tiempo exacto de su llegada.

En lugar de ello, se debe escribir un programa para simular las acciones de los clientes. Cada parte de la situación del mundo real tiene su parte análoga en el programa. La acción del mundo real de un cliente que llega se modela mediante una entrada de datos. Cada que llega un cliente se conocen dos cosas: el tiempo de llegada y la duración de la transacción (pues, supuestamente, todo cliente que llega sabe lo que desea hacer en el banco). Así, la entrada de datos para cada cliente consiste en un par de números: el tiempo (en minutos desde que abre el banco) en que llega el cliente y el tiempo (de nuevo en minutos) necesario para la transacción. Los pares de datos se ordenan de acuerdo con el tiempo de llegada ascendente. Así, se asume por lo menos una línea de entrada.

Las cuatro filas del banco se representan por medio de cuatro colas. Cada nodo de la cola representa un cliente que espera en una fila, y el nodo del frente representa el cliente que es atendido por un cajero.

Supóngase que en un momento determinado las cuatro filas contienen un número específico de clientes. ¿Qué debe ocurrir para que se altere el estado de las filas? Ya sea que un nuevo cliente entre al banco, en cuyo caso una de las filas tendrá un cliente adicional, o que el primer cliente de una de las filas complete su transacción, en cuyo caso la fila tendrá uno menos. Así, hay un total de cinco acciones (un cliente que entra, más cuatro posibilidades de que uno salga) que pueden cambiar el estado de las filas. Cada una de esas cinco acciones se denomina como *evento*.

### El proceso de simulación

La simulación se realiza encontrando el siguiente evento y efectuando el cambio en las colas que refleje el cambio ocasionado por dicho evento en las filas del banco. Para mantenerse informado de los eventos, el programa usa una cola de prioridad ascendente, llamada *lista de eventos*. Esta lista contiene a lo sumo cinco nodos, cada uno de los cuales representa la siguiente ocurrencia de uno de los cinco tipos de evento. Así, la lista de eventos contiene un nodo que representa al siguiente cliente que llega y cuatro nodos que representan a cada uno de los cuatro clientes, en la cabeza de una fila, que terminan su transacción y abandonan el banco. Por supuesto, es posible que una o más líneas del banco estén vacías, o que se cierran las

puertas del banco por ese día, de manera que no entren más clientes. En tales casos, la lista de eventos contiene menos de cinco nodos.

A un nodo evento que representa la llegada de un cliente se le llama *nodo de llegada*, y a uno que representa la salida, *nodo de salida*. En cada punto de la simulación, es necesario saber qué evento ocurrirá enseguida. Por esta razón, la lista de eventos se ordena con base en el tiempo ascendente de ocurrencia de un evento para que el primer nodo de evento de la lista represente el siguiente en ocurrir. Así, la lista de eventos es una cola de prioridad ascendente representada por una lista ligada ordenada.

El primer evento que ocurre es la llegada del primer cliente. La lista de eventos, por lo tanto, se inicializa mediante la lectura de la primera línea de entrada y la ubicación de un nodo de llegada, que representa la primera llegada de un cliente a la misma. Al principio, por supuesto, las cuatro colas para los cajeros están vacías. La simulación prosigue de la siguiente manera: se elimina el primer nodo de la lista de eventos y se hacen los cambios que dicho evento ocasiona en las colas. Como se verá adelante, esos cambios también pueden causar eventos adicionales que deben colocarse en la lista de eventos. El proceso de eliminación del primer nodo de la lista de eventos y la actualización de los cambios que eso conlleva, se repite hasta que la lista de eventos está vacía.

Cuando se elimina un nodo de llegada de la lista de eventos, se coloca un nodo que representa al cliente que llega a la fila más corta. Si ese cliente es el único en una fila, en la lista de eventos se coloca también un nodo que represente la salida del cliente, ya que éste está en el frente de la fila. Al mismo tiempo, se lee la línea de entrada siguiente y en la lista de eventos se coloca un nodo de llegada que represente al siguiente cliente que llega. En la lista de eventos siempre habrá sólo un nodo de llegada (siempre que no se agote la entrada, momento en el que ya no llegan más clientes), ya que tan pronto como se elimina un nodo de llegada de la lista de eventos, se agrega otro a la misma.

Cuando se elimina un nodo de salida de la lista de eventos, el nodo que representa al cliente que se va, se elimina del frente de una de las cuatro colas. En ese momento, se calcula la cantidad de tiempo que empleó ese cliente en el banco y se agrega al total. Al final de la simulación se divide ese total entre el número de clientes para obtener el tiempo promedio empleado por cada uno. Una vez que se borra un nodo que representa a un cliente del frente de la fila, el siguiente cliente (si lo hay) avanza para recibir el servicio del cajero correspondiente y se agrega a la lista de eventos un nodo de salida para el siguiente.

Este proceso continúa hasta que la lista de eventos esté vacía, momento en el cual se calcula el tiempo promedio y se imprime. Advírtase que, por si misma, la lista de eventos no refleja ninguna circunstancia del mundo real. Se usa como parte del programa para controlar todo el proceso. Una simulación como ésta, que actúa cambiando la situación simulada en respuesta a la aparición de uno de varios eventos, se conoce como *simulación conducida por eventos*.

### Estructuras de datos

Ahora se examinarán las estructuras de datos necesarias para este programa. Los nodos de las colas representan clientes y deben, en consecuencia, tener campos

que representen el tiempo de llegada y la duración de la transacción, además del campo *next* para ligar los nodos de una lista. Los nodos de la lista de eventos representan eventos y, por lo tanto, deben contener el tiempo en que ocurre el evento, el tipo de evento y cualquier tipo de información relacionada con el mismo, así como un campo *next*. Así, parecería que se necesitan dos depósitos de nodos diferentes para los dos tipos de nodos diferentes. Dos tipos de nodos diferentes ocasionarían dos rutinas *getnode* y *freenode* y dos conjuntos de rutinas diferentes para la manipulación de listas. Para evitar este molesto conjunto de rutinas duplicadas, se tratará de usar un solo tipo de nodo tanto para eventos como para clientes.

Se puede declarar un depósito para los nodos y un tipo apuntador como sigue:

```
struct node {
 int time;
 int duration;
 int type;
 struct node *next;
};

typedef struct node *NODEPTR;
```

En un nodo para cliente, *time* es el tiempo de llegada del cliente y *duration* es la duración de la transacción. *type* no se usa en los nodos para clientes. *next* es un apuntador para ligar los elementos de la cola. En los nodos para eventos se usa *time* para guardar el tiempo de ocurrencia del evento; *duration* se usa para la duración de la transacción de un cliente que llega en un nodo de llegada, pero no se usa en los nodos de partida. *type* es un entero entre -1 y 3, dependiendo de si el evento es una llegada (*type* == -1) o una salida de la línea 0, 1, 2 o 3 (*type* == 0, 1, 2, o 3). *next* guarda un apuntador que liga la lista de eventos.

Las cuatro colas que representan las filas para los cajeros se declaran como un arreglo mediante la declaración:

```
struct queue {
 NODEPTR front, rear;
 int num;
} q[4];
```

La variable *q[i]* representa una cabecera para la *i*-ésima cola de cajero. El campo *num* de una cola contiene el número de clientes de la misma.

Una variable *evlist* apunta al frente de la lista de eventos. Una variable *tottime* se usa para mantenerse informado sobre el tiempo total empleado por todos los clientes y *count* cuenta el número de clientes que han pasado por el banco. Estas variables se usarán al final de la simulación para calcular el tiempo promedio empleado por los clientes del banco. Para almacenar por un tiempo la porción de información de un nodo se usa una variable auxiliar, *auxinfo*. Estas variables se declaran por medio de

```
NODEPTR evlist;
float count, tottime;
struct node auxinfo;
```

### El programa de simulación

La rutina principal inicializa todas las listas y colas y elimina repetidamente el nodo siguiente de la lista de eventos para conducir la simulación hasta que la lista de eventos esté vacía. La lista de eventos se ordena incrementando los valores del campo *time*. El programa llama a *place(&evlist, &auxinfo)* para insertar un nodo cuya información está determinada por *auxinfo* en el lugar apropiado de la lista de eventos. La rutina principal llama también a *popsub(&evlist, &auxinfo)* para eliminar el primer nodo de la lista de eventos y colocar su información en *auxinfo*. Esta rutina es equivalente a la función *pop*. Estas rutinas deben, por supuesto, modificarse de manera apropiada, a partir de los ejemplos proporcionados en la última sección, para poder tratar este tipo de nodo particular. Adviértase que *evlist*, *place* y *popsub* no son más que una implantación particular de una cola de prioridad ascendente y de las operaciones *pqinsert* y *pqmindelete*. Una representación más eficiente de una cola de prioridad (como la que se presenta en las secciones 6.3 y 7.3) permitirían que el programa operase con más eficacia.

El programa principal llama también a las funciones *arrive* y *depart*, las que efectúan los cambios causados por una llegada y una salida en las colas y la lista de eventos. En especial, la función *arrive(atime, dur)* refleja la llegada de un cliente en el tiempo *atime* con una transacción de duración *dur*, y la función *depart(qindx, dtime)* refleja la salida del primer cliente de la cola *q[qindx]* en el tiempo *dtime*. Los códigos de estas rutinas se proporcionarán en breve.

```
#include <STDIO.H>
#define NULL 0
struct node {
 int duration, time, type;
 struct node *next;
};
typedef struct node *NODEPTR;
struct queue {
 NODEPTR front, rear;
 int num;
};
struct queue q[4];
struct node auxinfo;
NODEPTR evlist;
int atime, dtime, dur, qindx;
float count, tottime;
main()
{
 /* inicializaciones */
 evlist = NULL;
 count = 0;
 tottime = 0;
 for (qindx = 0; qindx < 4; qindx++) {
 q[qindx].num = 0;
 }
}
```

```

q[qindx].front = NULL;
q[qindx].rear = NULL;
} /* fin de for */
/* inicializar la lista de eventos con la primera llegada */
printf("proporcionar tiempo y duración\n");
scanf("%d %d", &auxinfo.time, &auxinfo.duration);
auxinfo.type = -1; /* una llegada */
place(&evlist, &auxinfo);

/* Efectuar la simulación mientras la lista de eventos no se vacíe */
while (evlist != NULL) {
 popsub(&evlist, &auxinfo);
 /* verificar si el siguiente evento es una llegada o una salida */
 if (auxinfo.type == -1) {
 /* una llegada */
 atime = auxinfo.time;
 dur = auxinfo.duration;
 arrive(atime, dur);
 } else {
 /* una salida */
 qindx = auxinfo.type;
 dtime = auxinfo.time;
 depart(qindx, dtime);
 }
}
printf("El tiempo promedio es %.2f", tottime/count);
} /* fin de main */

```

La rutina *arrive(atime, dur)* modifica las colas y la lista de eventos para reflejar una nueva llegada en el tiempo *atime* con una transacción de duración *dur*. Esta rutina inserta un nuevo nodo al final de la cola más corta mediante una llamada a la función *insert(&q[j], &auxinfo)*. La rutina *insert* debe modificarse de manera apropiada para tratar el tipo de nodo de este ejemplo y también debe incrementarse en 1 *q[j].num*. Cuando un cliente es el único en la cola, se agrega un nodo que representa su salida o llegada a la lista de eventos por medio de la función *place(&evlist, &auxinfo)*. Después se lee el par de datos siguiente (si existe) y se coloca un nodo de llegada en la lista de eventos para remplazar la llegada que se acaba de procesar. Si no hay más entradas, la función regresa sin agregar un nuevo nodo de llegada y el programa procesa los nodos (de salida) restantes en la lista de eventos.

```

arrive(atime, dur)
int atime, dur;
{
 int i, j, small;
 /* localizar la cola más corta */

```

```

j = 0;
small = q[0].num;
for (i = 1; i < 4; i++) {
 if (q[i].num < small) {
 small = q[i].num;
 j = i;
 }
}
/* La cola j es la más corta. Insertar un nuevo nodo de cliente */
auxinfo.time = atime;
auxinfo.duration = dur;
auxinfo.type = j;
insert(&q[j], &auxinfo);
/* Verificar si es el único nodo en la cola. De ser así, el nodo de salida del cliente deberá colocarse en la lista de eventos */
if (q[j].num == 1) {
 auxinfo.time = atime + dur;
 place(&evlist, &auxinfo);
}
/* Si aún existen algunos datos de entrada, leer el siguiente par de datos y colocar una llegada en la lista de eventos */
printf("proporcionar tiempo\n");
if (scanf("%d", &auxinfo.time) != EOF) {
 printf("indique la duración\n");
 scanf("%d", &auxinfo.duration);
 auxinfo.type = -1;
 place(&evlist, &auxinfo);
}
/* fin de if */
} /* fin de arrive */

```

La rutina *depart(qindx, dtime)* modifica la cola *q[qindx]* y la lista de eventos para reflejar la salida del primer cliente de la cola en el tiempo *dtime*. El cliente se elimina de la cola con una llamada a *remove(&q[qindx], &auxinfo)*, la que debe modificarse de manera adecuada para tratar el tipo de nodo de este ejemplo y también debe disminuirse en 1 el campo *num* de la cola. El nodo de salida del siguiente cliente de la cola (si lo hay) remplaza al nodo de salida que apenas se eliminó de la lista de eventos.

```

depart(qindx, dtime)
int qindx, dtime;
{
 NODEPTR p;
 remove(&q[qindx], &auxinfc);
 tottime = tottime + (dtime - auxinfo.time);
 count++;
 /* Si hay algunos otros clientes en la cola, colocar la salida del siguiente cliente en la lista de eventos después de calcular su tiempo de partida */

```

```

if (q[qindx].num > 0) {
 p = q[qindx].front;
 auxinfo.time = dtime + p->duration;
 auxinfo.type = qindx;
 place(&evlist, &auxinfo);
} /* fin de if */
} /* fin de depart */

```

Los programas de simulación son variados en el uso de estructuras de lista. Se invita al lector a explorar el uso de C para la simulación y el uso de lenguajes de simulación de propósito especial.

## EJERCICIOS

- 4.4.1. En el programa del texto de simulación del banco, un nodo de salida de la lista de eventos representa al mismo cliente del primer nodo de una cola de clientes. ¿Es posible usar un solo nodo para un cliente que recibe servicio? Volver a escribir el programa del texto de manera que se use un solo nodo. ¿Existe alguna ventaja si se emplean dos nodos?
- 4.4.2. El programa del texto usa el mismo tipo de nodos tanto para clientes como para eventos. Escribir de nuevo el programa usando dos tipos de nodos diferentes para esos dos propósitos. ¿Esto ahorra espacio?
- 4.4.3. Revise el programa de simulación del banco, presentado en el texto, para determinar la longitud promedio de las cuatro filas.
- 4.4.4. Modifique el programa de simulación del banco para calcular las desviaciones estándar del tiempo que invierte un cliente en el banco. Escriba otro programa que simule una sola cola para los cuatro cajeros, en el que el cliente que está a la cabeza de una fila va al siguiente cajero disponible. Compare la media y la desviación estándar de los dos métodos.
- 4.4.5. Modifique el programa de simulación del banco de manera que siempre que la longitud de una de las filas excede a la de otra en más de dos, el último cliente de la fila más larga pase al final de la más corta.
- 4.4.6. Escriba un programa en C para simular un sistema de computadoras de usuarios múltiples de la siguiente manera: cada usuario tiene una ID (identificación) única y desea realizar un número de transacciones en la computadora. Sin embargo, en un momento dado, la computadora sólo puede procesar una transacción. Cada línea de entrada representa un solo usuario y contiene el ID de éste seguido del tiempo de comienzo y una serie de enteros que representan la duración de cada una de las transacciones. La entrada se clasifica de manera creciente de acuerdo con el tiempo de inicio, y todas las duraciones y los tiempos están proporcionados en segundos. Supóngase que cada usuario solicita tiempo para una transacción hasta que se ha completado la transacción anterior y que la computadora procesa transacciones con la política de el primero en llegar, el primero en ser atendido. El programa deberá simular el sistema e imprimir un mensaje que contenga la ID del usuario y el tiempo en que comienza y termina una transacción. Al final de la simulación debe imprimir el tiempo promedio de espera para una transacción. (El tiempo de espera es la cantidad del tiempo desde que se solicita la transacción hasta que ésta comienza.)

- 4.4.7. ¿Qué partes del programa de la simulación del banco serían transformadas si la cola de prioridad para los eventos se implantara como un arreglo o una lista no ordenada? ¿Cómo podrían modificarse?

- 4.4.8. Muchas simulaciones no simulan eventos proporcionados por datos de entrada, sino por eventos generados de acuerdo con cierta distribución de probabilidad. Los siguientes ejercicios explican cómo. La mayoría de las computadoras tienen una función *rand(x)* que genera un número aleatorio. (El nombre y los parámetros de la función varían de un sistema a otro. *rand* se usa sólo como ejemplo.) *x* se inicializa a un valor llamado *semilla*. La instrucción *x = rand(x)* reinicializa el valor de la variable *x* a un número real aleatorio uniforme entre 0 y 1. Esto quiere decir que si se ejecuta la instrucción un número suficiente de veces y se eligen dos intervalos de longitud igual entre 0 y 1, caerán en ellos la misma cantidad de valores sucesivos de *x* de manera aproximada. Así, la probabilidad de que un valor de *x* caiga en un intervalo de longitud *l* <= 1 es igual a *l*. Encuentre el nombre de la función que genera números aleatorios en el sistema y verifique que lo siguiente sea cierto.

Dado un generador de números aleatorios *rand*, considérese las siguientes instrucciones:

```

x = rand(x);
y = (b-a)*x + a

```

Muestre que, si se dan dos intervalos cualesquier de igual longitud dentro del intervalo de *a* a *b* y si las instrucciones se repiten con suficiente frecuencia, el mismo número de valores sucesivos de *y* cae en cada uno de los intervalos en forma aproximada. Muestre que si *a* y *b* son enteros, la parte entera de los valores sucesivos de *y* es igual a cada entero entre *a* y *b* — 1 en un número de veces aproximadamente igual. Se dice que la variable *y* es una *variable aleatoria distribuida uniformemente*. ¿Cuál es el promedio de los valores de *y* en términos de *a* y *b*?

Reescriba la simulación del banco que se presentó en el texto suponiendo que la duración de la transacción está distribuida de manera uniforme entre 1 y 15. Cada par de datos representa un cliente que llega y sólo contiene el tiempo de llegada. Al leer una línea de entrada, genere una duración para la transacción del cliente calculando el valor sucesivo de acuerdo con el método que se acaba de esbozar.

- 4.4.9. Se dice que los valores sucesivos de *y*, generados por las siguientes instrucciones, están *normalmente distribuidos*. (En realidad, casi están normalmente distribuidos, pero la aproximación es realmente buena.)

```

float x[15];
float m, s, sum, y;
int i;
/*
 Aquí se colocan las instrucciones para dar los
 valores iniciales de s, m y el arreglo x
*/
while (/* aquí se ubica la condición de terminación */) {
 sum = 0;
 for (i = 0; i < 15; i++) {
 x[i] = rand(x[i]);
 sum = sum + x[i];
 } /* end for */
 y = s * (sum - 7.5) / sqrt(1.25) + m;
}

```

```

 /* Aquí se colocan las instrucciones que emplean el valor de y */
} /* fin de while */

```

Verifique que el promedio de los valores de  $y$  (la media de la distribución) es igual a  $m$  y que la desviación estándar es igual a  $s$ .

Cierta fábrica produce artículos de acuerdo con el siguiente proceso: se debe ensamblar y pulir un artículo. El tiempo de ensamblaje se distribuye de manera uniforme entre 100 y 300 segundos, y el de pulido con una media de 20 segundos y una desviación estándar de 7 segundos (pero se descartan los valores por debajo de 5). Después de que se ensambla un artículo, hay que usar una pulidora y un trabajador no puede comenzar a ensamblar el artículo siguiente hasta que no esté pulido el que acaba de montar. Hay diez trabajadores, pero una sola pulidora. Si ésta no está disponible, los trabajadores que han terminado de ensamblar sus artículos deben esperar. Calcule el tiempo promedio de espera por artículo mediante una simulación. Haga lo mismo suponiendo que hay dos o tres máquinas pulidoras.

## 4.5. OTRAS ESTRUCTURAS DE LISTA

Aunque una lista lineal ligada es una estructura de datos útil, tiene varios defectos. En esta sección se presentan otros métodos para organizar una lista y se muestra cómo usar éstos para superar dichos defectos.

### Listas circulares

Luego de que determina un apuntador  $p$  a un nodo en una lista lineal, resulta imposible alcanzar algún nodo que preceda a  $node(p)$ . Si se recorre una lista, hay que preservar el apuntador externo a la lista para poder referenciar ésta de nuevo.

Suponga que se hace un pequeño cambio a la estructura de una lista lineal de manera que el campo *next* en el último nodo contenga un apuntador al primer nodo en lugar del apuntador nulo. Dicha lista se llama *lista circular* y se ilustra en la figura 4.5.1. En una lista como ésa es posible alcanzar cualquier punto de la lista desde otro cualquiera. Si se empieza en un nodo determinado y se recorre toda la lista, se terminará al final en el punto de partida.

Obsérvese que una lista circular no tiene un “primer” o “último” nodo natural. Por lo tanto, hay que establecer un primer y último nodo por convención. Una convención útil es dejar el apuntador externo a la lista circular apuntando al último nodo y dejar que el nodo siguiente sea el primero, como se ilustra en la figura 4.5.2. Si  $p$  es un apuntador externo a una lista circular, esta convención permite accesar el último nodo de la lista referenciando a  $node(p)$  y el primer nodo de la lista referenciando  $node(next(p))$ . Esta convención proporciona la ventaja de poder agregar o

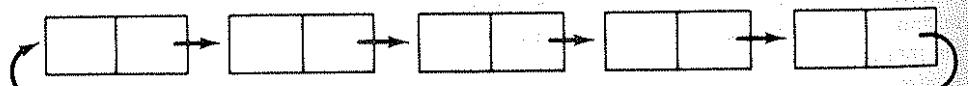


Figura 4.5.1 Una lista circular.

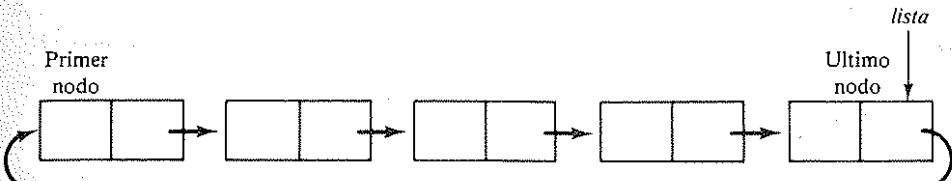


Figura 4.5.2 Primero y último nodo de una lista circular.

eliminar elementos de manera adecuada, ya sea desde el frente o desde el final de la lista. También se establece la convención de que el apuntador nulo representa una lista circular vacía.

### La pila como una lista circular

Una lista circular puede usarse para representar una pila o cola. Sea *stack* un apuntador al último nodo de una lista circular, y adóptese la convención de que el primer nodo es el tope de la pila. Una pila vacía se representa mediante la lista nula. La siguiente es una función en C para determinar si la pila está vacía. A *empty(&stack)* llama a esta función.

```

empty(pstack)
NODEPTR *pstack;
{
 return ((*pstack == NULL) ? TRUE : FALSE);
} /* fin de empty */

```

La siguiente es una función en C para poner un entero  $x$  dentro de una pila. La función *push* llama a una función *empty*, la que verifica si su parámetro es *NULL*. La llamada se realiza mediante *push(&stack, x)*, donde *stack* es un apuntador a una lista circular que actúa como pila.

```

push(pstack, x)
NODEPTR *pstack;
int x;
{
 NODEPTR p;
 p = getnode();
 p->info = x;
 if (empty(pstack) == TRUE)
 *pstack = p;
 else
 p->next = (*pstack) -> next;
 (*pstack) -> next = p;
} /* fin de push */

```

Adviértase que la rutina *push* es algo más compleja para las listas circulares que para las listas lineales.

La función *pop* en C para una pila implantada como lista circular llama a la función *freenode* presentada con anterioridad. *pop* es llamada por *pop(&stack)*.

```
pop(pstack)
NODEPTR *pstack;
{
 int x;
 NODEPTR p;
 if (empty(pstack) == TRUE) {
 printf("subdesborde de la pila\n");
 exit(1);
 } /* fin de if */
 p = (*pstack) -> next;
 x = p->info;
 if (p == *pstack)
 /* sólo un nodo en la pila */
 *pstack = NULL;
 else
 (*pstack) -> next = p->next;
 freenode(p);
 return(x);
} /* fin de pop */
```

### La cola como una lista circular

Es más fácil representar una cola como una lista circular que como una lineal. Como lista lineal, una cola se especifica por medio de dos apuntadores, uno al frente de la lista y otro al final. Sin embargo, mediante una lista circular, se puede especificar una cola a través de un apuntador simple *q* a dicha lista. *node(q)* es el final de la cola y el siguiente nodo es su frente.

La función *empty* es la misma que para pilas. La rutina *remove(pq)* llamada por *remove(&q)* es idéntica a *pop*, excepto en que todas las referencias a *pstack* son remplazadas por *pq*, un apuntador a *q*. La rutina C *insert* se llama por la instrucción *insert(&q, x)* y se puede codificar como sigue:

```
insert(pq, x)
NODEPTR *pq;
int x;
{
 NODEPTR p;
 p = getnode();
 p->info = x;
 if (empty(pq) == TRUE)
 *pq = p;
 else
 p->next = (*pq) -> next;
 (*pq) -> next = p;
```

```
*pq = p;
return;
} /* fin de insert */
```

Adviértase que *insert(&q, x)* es equivalente al código:

```
push(&q, x);
q = q->next;
```

Esto es, para insertar un elemento al final de una cola circular, se debe insertar éste en el frente de la cola y el apuntador a la lista circular debe recorrerse un elemento hacia delante, por lo que el nuevo elemento se convierte en el del final.

### Operaciones primitivas en listas circulares

La rutina *insafter(p, x)*, que inserta un nodo que contiene *x* después de *node(p)*, es similar a la rutina correspondiente para listas lineales presentadas en la sección 4.3. Sin embargo, es necesario modificar ligeramente la rutina *delafter(p, x)*. Si se mira la rutina correspondiente para listas lineales presentada en la sección 4.3, debe advertirse una consideración adicional en el caso de una lista circular. Supóngase que *p* apunta al único nodo de la lista. En una lista lineal, *next(p)* sería nulo en este caso, lo que invalida la eliminación. Sin embargo, en una lista circular, *next(p)* apunta a *node(p)*, de tal manera que *node(p)* se sigue a sí mismo. La pregunta en este caso, es si es deseable o no eliminar *node(p)* de la lista. Es improbable que se desee esto, ya que la operación *delafter* se invoca de manera normal cuando los apuntadores de los dos nodos están determinados de manera que uno sigue al otro y que se desee eliminar el segundo. Para listas circulares, *delafter* se implanta mediante la implantación dinámica con nodos como sigue:

```
delafter(p, px)
NODEPTR p;
int *px;
{
 NODEPTR q;
 if ((p == NULL) || (p == p->next)) {
 /* la lista está vacía o contiene un solo nodo */
 printf("eliminación no efectuada\n");
 return;
 } /* fin de if */
 q = p->next;
 *px = q->info;
 p->next = q->next;
 freenode(q);
 return;
} /* fin de delafter */
```

Obsérvese, sin embargo, que *insafter* no puede usarse para insertar un nodo que siga al último de una lista circular y que *delafter* no se puede usar para eliminar el último nodo de una lista circular. En ambos casos, el apuntador externo a la lista debe modificarse para apuntar al nuevo último nodo. Es posible modificar rutinas para aceptar *list* como parámetro adicional y cambiar su valor cuando sea necesario. (El parámetro actual en la rutina de llamada tendría que ser *&list*, ya que su valor se cambió.) Una alternativa es escribir rutinas distintas *insend* y *dellast* para esos casos. (*insend* es idéntica a la operación *insert* para una cola implementada como una lista circular.) La rutina de llamada sería responsable de determinar qué rutina llamar. Otra posibilidad es dar a la rutina de llamada la responsabilidad de ajustar el apuntador externo *list* si es necesario. El análisis de esas posibilidades se le quedan al lector.

Si se maneja una lista propia de nodos disponibles (por ejemplo, al usar la implantación por arreglos), también es más fácil liberar una lista circular completa que liberar una lista lineal. En el caso de una lista lineal, ésta deberá recorrerse por completo, ya que los nodos se colocan, uno a uno, en la lista disponible. En el caso de una lista circular, se puede escribir una rutina *freelist* que libere de manera efectiva una lista completa con el simple hecho de volver a arreglar los apuntadores. Esto se deja como ejercicio para el lector.

De manera análoga, se puede escribir una rutina *concat(&list1, &list2)* que concatene dos listas, esto es, que agregue la lista circular apuntada por *list2* al final de la apuntada por *list1*. Si se usan listas circulares, esto puede hacerse sin recorrer ninguna de las listas:

```

NODEPTR concat(NODEPTR *plist1, NODEPTR *plist2)
{
 NODEPTR p;
 if (*plist2 == NULL)
 return;
 if (*plist1 == NULL) {
 *plist1 = *plist2;
 return;
 }
 p = (*plist1) -> next;
 (*plist1) -> next = (*plist2) -> next;
 (*plist2) -> next = p;
 *plist1 = *plist2;
 return;
} /* fin de concat */

```

### El problema de Josephus

Considérese un problema que puede resolverse de manera directa mediante una lista circular. El problema se conoce como el problema de Josephus y postula a un grupo de soldados rodeados por una abrumadora fuerza enemiga. No hay espe-

ranza de victoria sin refuerzos, y sólo hay un caballo disponible para el escape. Los soldados están de acuerdo en pactar para determinar cuál de ellos debe escapar y pedir ayuda. Forman un círculo y se escoge un número *n* de un sombrero. También se escoge uno de sus nombres de un sombrero. Se empieza a contar en el sentido de las manecillas del reloj empezando por el soldado cuyo nombre fue elegido alrededor del círculo. Cuando el conteo llega a *n*, ese soldado se elimina del círculo y el conteo comienza de nuevo con el soldado siguiente. El proceso continúa de modo que cada vez que el conteo alcanza a *n* se elimina un soldado del círculo. Ningún soldado eliminado del círculo se vuelve a contar. El último que queda toma el caballo y escapa. Dado un número *n*, el orden de los soldados en el círculo y el soldado a partir del cual comienza el conteo, el problema consiste en el orden en el que se eliminarán los soldados del círculo y el soldado que escapa.

La entrada del programa es el número *n* y una lista de nombres, que es el orden en el sentido de las manecillas del reloj de los soldados del círculo, comenzando por el soldado a partir del cual comienza el conteo. La última línea de entrada contiene la cadena "END", que indica el final de la entrada. El programa deberá imprimir los nombres en el orden en que se eliminan del círculo y el del soldado que escapa.

Por ejemplo, supóngase que *n* = 3 y que hay cinco soldados llamados *A*, *B*, *C*, *D* y *E*. Hay que contar tres soldados a partir de *A*, de manera que *C* es el primero en ser eliminado. Luego se comienza con *D*, se cuenta *D*, *E* y *A*, y se elimina *A*. Después se cuenta *B*, *D*, y *E* (*C* ya se eliminó) y por último *B*, *D* y *B*, de tal manera que es *D* el que escapa.

Desde luego que en la resolución de este problema es natural usar una lista circular en la que cada nodo represente un soldado como estructura de datos. Es posible alcanzar cualquier nodo a partir de otro si se cuenta alrededor del círculo. Para representar la eliminación de un soldado del círculo, se borra un nodo de la lista circular. Al final, cuando sólo resta un nodo en la lista, el resultado está determinado.

Un esbozo del programa podría ser el siguiente:

```

read(n);
read(name);
while (name != END) {
 insertar el nombre en la lista circular;
 read(name);
} /* fin de while */
while (Existe más de un nodo en la lista) {
 contar n-1 nodos en la lista;
 imprimir el nombre del n-ésimo nodo;
 eliminar el n-ésimo nodo;
} /* fin de while */
imprimir el nombre del único nodo en la lista;

```

Supóngase que se ha declarado un conjunto de nodos igual que el anterior, excepto en que el campo *info* contiene una cadena de caracteres (un arreglo de caracteres) en lugar de un entero. Supóngase también por lo menos un nombre en la entrada. El programa usa las rutinas *insert*, *delafter* y *freenode*. Las rutinas *insert* y *delafter* se deben modificar, ya que la porción de información del nodo es una cadena de carac-

teres. La asignación de una cadena de caracteres variable a otra se realiza mediante un ciclo. El programa también se vale de una función *eqstr(str1, str2)*, la que da como resultado *TRUE* si *str1* es idéntica a *str2* y *FALSE* en caso contrario. La codificación de dicha rutina se deja como ejercicio al lector.

```

josephus()
{
 char *end = "end";
 char name[MAXLEN];
 int i, n;
 NODEPTR list = NULL;
 printf("proporcionar n/n");
 scanf("%d", &n);
 /* leer los nombres, colocándolos al final de la lista */
 printf("Proporcione los nombres\n");
 scanf("%s", name);
 /* construcción de la lista */
 while (!eqstr(name, end)) {
 insert(&list, name);
 scanf("%s", name);
 } /* fin de while */
 printf("El orden en que se eliminan los soldados es: %n");
 /* Continuar contando mientras exista más de un nodo en la lista */
 while (list != list->next) {
 for (i = 1; i < n; i++)
 list = list->next;
 /* list->next apunta al n-ésimo nodo */
 delafters(list, name);
 printf("%s\n", name);
 } /* fin de while */
 /* imprimir el único nombre en la lista y liberar su nodo */
 printf("el soldado que escapa es: %s", list->info);
 freeNode(list);
} /* fin de josephus */

```

#### Nodos cabecera

Supóngase que se desea recorrer una lista circular. Esto puede hacerse ejecutando de forma repetida  $p = p -> next$ , donde  $p$  es en el inicio un apuntador al principio de la lista. Sin embargo, como la lista es circular, no se puede saber cuándo se ha recorrido toda la lista sin que otro apuntador, *list*, apunte al primer nodo de la lista y sin que se verifique la condición  $p == list$ .

Un método alterno sería colocar un nodo cabecera como el primer nodo de una lista circular. Esta cabecera de lista puede reconocerse por medio de un valor espe-

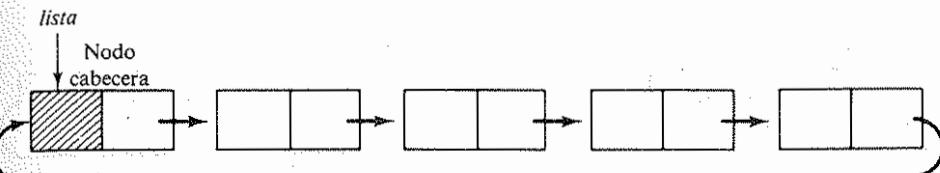


Figura 4.5.3 Una lista circular con un nodo cabecera.

cial en su campo *info*, que no puede ser un contenido válido como nodo de la lista en el contexto del problema, o puede contener un indicador que la marque como cabecera. La lista se puede recorrer por medio de un solo apuntador, deteniendo el recorrido al encontrar el nodo cabecera. El apuntador externo a la lista apunta al nodo cabecera, como se muestra en la figura 4.5.3. Esto significa que no es muy fácil agregar un nodo al final de una lista circular como se podría hacer si el apuntador externo apuntara al último nodo de la misma. Por supuesto, es posible mantener un apuntador al último nodo de la lista circular aun cuando se use un nodo cabecera.

Cuando se usa un apuntador externo estacionario a una lista circular, además del apuntador para el recorrido, no es necesario que el nodo cabecera contenga un código especial aunque se pueda usar casi de la misma manera que un nodo cabecera de una lista lineal para contener información global acerca de la lista. El final del recorrido se señalaría mediante la igualdad del apuntador del recorrido y el apuntador estacionario externo.

#### Suma de enteros positivos largos mediante listas circulares

Ahora se presentará una aplicación de las listas circulares con nodos cabecera. El hardware de muchas computadoras sólo permite enteros de una longitud máxima específica. Supóngase que se desea representar enteros positivos de longitud arbitraria y escribir una función que dé como resultado la suma de dos de esos enteros.

Para sumar dos enteros de ese tipo se recorren sus dígitos de derecha a izquierda y se agregan los dígitos correspondientes, así como el posible acarreo de la suma de los dígitos previos. Esto sugiere la representación de enteros largos mediante el almacenamiento de sus dígitos, de derecha a izquierda, en una lista para que el primer nodo de la lista contenga el dígito menos significativo (el de la extrema derecha), y el último, el más significativo (el de la extrema izquierda). Sin embargo, para ahorrar espacio, se guardan cinco dígitos en cada nodo. (Se usan variables de enteros largos para poder guardar en cada nodo números tan grandes como 99999. El tamaño máximo de un entero depende de la implantación; en consecuencia, tendrían que modificarse las rutinas para guardar números más pequeños en cada nodo). El conjunto de nodos se puede declarar mediante:

```

struct node {
 long int info;
 struct node *next;
};
typedef struct node *NODEPTR;

```

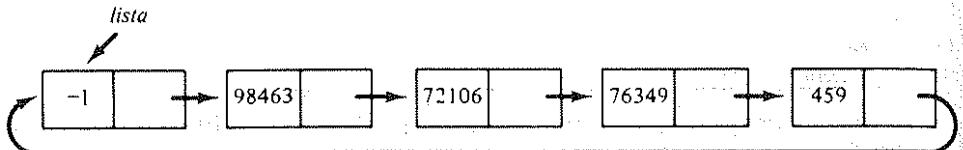


Figura 4.5.4 Un entero grande como una lista circular.

Ya que se desea recorrer las listas durante la suma pero también se desea restaurar posteriormente los apuntadores de la lista a sus valores originales, se usan listas circulares con nodos cabecera. El nodo cabecera se distingue por su campo *info* con valor -1. Por ejemplo, el entero 459763497210698463 se representa mediante la lista ilustrada en la figura 4.5.4.

Ahora se escribirá una función *addint* que acepta apuntadores a dos de esas listas que representan enteros, crea una lista que representa la suma de los enteros y da como resultado un apuntador a la lista suma. Ambas listas se recorren en paralelo y se suman cinco dígitos a la vez. Si la suma de dos números de cinco dígitos es  $x$ , los cinco dígitos de menor orden de  $x$  se pueden extraer mediante la expresión  $x \% 100000$ , que proporciona el residuo de  $x$  dividido entre 100000. El acarreo se puede calcular mediante la división de enteros  $x / 100000$ . Cuando se alcanza el final de una lista, dicho acarreo se propaga a los dígitos restantes de la otra lista. La función sigue y usa las rutinas *getnode* e *insafter*.

```

NODEPTR addint(p, q)
NODEPTR p, q;
{
 long int hunthou = 100000L;
 long int carry, number, total;
 NODEPTR s;
 /* Asignar a p y q los nodos siguientes a los nodos cabeceras */
 p = p->next;
 q = q->next;
 /* preparar un nodo cabecera para la suma */
 s = getnode();
 s->info = -1;
 s->next = s;
 /* al principio no hay acarreo */
 carry = 0;
 while (p->info != -1 && q->info != -1) {
 /* sumar la información de los dos nodos */
 total = p->info + q->info + carry;
 /* Determinar los cinco dígitos de menor
 orden de la suma e insertarlos en la lista */
 number = total % hunthou;
 insafter(s, number);
 /* avance los Traversals */
 p = p->next;
 q = q->next;
 }
}

```

```

s = s->next;
p = p->next;
q = q->next;
/* determinar si existe acarreo */
carry = total / hunthou;
} /* fin de while */
/* a partir de aquí, pueden quedar todavía nodos en alguna de
 las listas de entrada */
while (p->info != -1) {
 total = p->info + carry;
 number = total % hunthou;
 insafter(s, number);
 carry = total / hunthou;
 s = s->next;
 p = p->next;
} /* fin de while */
while (q->info != -1) {
 total = q->info + carry;
 number = total % hunthou;
 insafter(s, number);
 carry = total / hunthou;
 s = s->next;
 q = q->next;
} /* fin de while */
/* verificar si hay un acarreo adicional de los primeros
 cinco dígitos */
if (carry == 1) {
 insafter(s, carry);
 s = s->next;
} /* fin de if */
/* s apunta al último nodo de la suma. s->next */
/* apunta el nodo cabecera de la lista sum */
return(s->next);
} /* fin de addint */

```

### Listas doblemente ligadas

Aunque una lista ligada de manera circular tiene ventajas sobre una lista lineal, aún presenta algunos inconvenientes. No se puede recorrer tal lista en sentido inverso ni se puede borrar un nodo de una lista circular ligada, dando sólo un apuntador al mismo. En los casos en que se requieran esas facilidades, la estructura de datos apropiada es una *lista doblemente ligada*. Cada nodo de una lista de ese tipo contiene dos apuntadores: uno a su antecesor y otro a su sucesor. En realidad, en el contexto de listas doblemente ligadas, los términos antecesor y sucesor carecen de significado, ya que la lista es completamente simétrica. Las listas doblemente ligadas pueden ser lineales o circulares y pueden o no contener un nodo cabecera como se ilustra en la figura 4.5.5.

Se puede considerar que los nodos de una lista doblemente ligada consisten en tres campos: un campo *info* que contiene la información almacenada en el nodo, y los campos *left* y *right* que contienen apuntadores a los nodos de cada lado. Es posible declarar un conjunto de tales nodos, los que se valen de un arreglo o de una implantación dinámica, mediante:

#### Implantación de nodos

```
struct nodetype {
 int info;
 int left, right;
};
```

```
struct nodetype node[NUMNODES];
```

```
typedef struct node *NODEPTR;
```

#### Implantación dinámica

```
struct node {
```

```
int info;
```

```
struct node *left, *right;
```

}

Obsérvese que la lista disponible para tal conjunto de nodos en la implantación con arreglo no necesita estar doblemente ligada, ya que no se recorre de manera bidireccional. La lista disponible se puede ligar mediante el apuntador *left* o *right*. Por supuesto, hay que escribir las rutinas apropiadas *getnode* y *freenode*.

Ahora se presentan rutinas para operar sobre listas doblemente ligadas circulares. Una propiedad conveniente de dichas listas es que si *p* es un apuntador a algún nodo, donde se deja a *left(p)* como abreviatura de *node[p].left* o a *p -> left* y *right(p)* como abreviatura de *node[p].right* o *p -> right*, se tendrá

$$\text{left}(\text{right}(p)) = p = \text{right}(\text{left}(p))$$

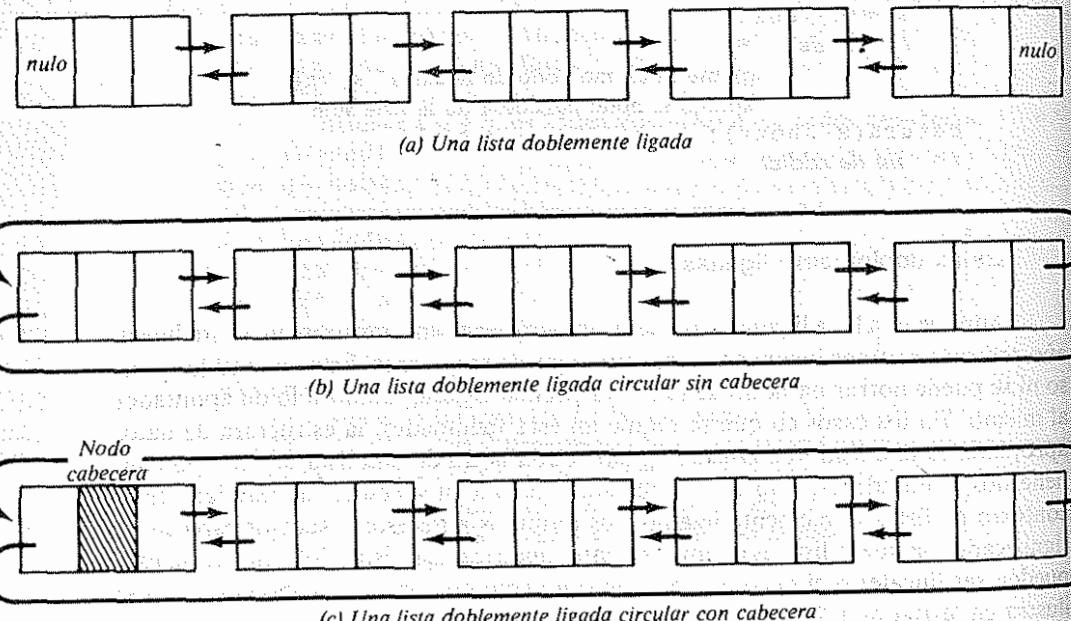


Figura 4.5.5 Listas doblemente ligadas.

Una operación que puede ejecutarse sobre listas doblemente ligadas, pero no sobre listas ligadas ordinarias, es la de eliminar un nodo determinado. La siguiente rutina C elimina el nodo apuntado por *p* de una lista doblemente ligada y almacena sus contenidos en *x* por medio de la implantación dinámica. La rutina es llamada por *delete(p, &x)*

```
delete(p, px)
NODEPTR p;
int *px;
{
 NODEPTR q, r;
 if (p == NULL) {
 printf("eliminación no efectuada\n");
 return;
 } /* fin de if */
 *px = p->info;
 q = p->left;
 r = p->right;
 q->right = r;
 r->left = q;
 freenode(p);
 return;
} /* fin de delete */
```

La rutina *insertright* inserta un nodo con campo de información *x* a la derecha de *node(p)* en una lista doblemente ligada:

```
insertright(p, x)
NODEPTR p;
int x;
{
 NODEPTR q, r;
 if (p == NULL) {
 printf("eliminación no efectuada\n");
 return;
 } /* fin de if */
 q = getnode();
 q->info = x;
 r = p->right;
 r->left = q;
 q->right = r;
 q->left = p;
 p->right = q;
 return;
} /* fin de insertright */
```

Al lector se le queda como ejercicio una rutina similar *insertleft* para insertar un nodo con campo de información *x* a la izquierda de *node(p)* en una lista doblemente ligada.

Cuando la eficacia de espacio es importante, un programa no puede darse el lujo de tener dos apuntadores para cada elemento de la lista. Existen varias técnicas para comprimir los apuntadores izquierdo y derecho de un nodo en un solo campo. Por ejemplo, un campo de apuntador simple *ptr* para cada nodo puede contener la suma de apuntadores de sus vecinos izquierdo y derecho. (Aquí, se presume que los apuntadores se representan de un modo tal que es posible realizar aritmética con ellos de manera rápida. Por ejemplo, los apuntadores representados mediante índices de un arreglo se pueden sumar y restar. Aunque en C no es legal sumar dos apuntadores, muchos compiladores permitirán tal aritmética de apuntadores. Dados dos apuntadores externos, *p* y *q*, a dos nodos adyacentes de manera que *p* == *left(q)*, *right(q)* puede calcularse como *ptr(q) - p* y *left(p)* como *ptr(p) - q*. Dados *p* y *q*, es posible eliminar cualquier nodo y reinicializar su apuntador al nodo antecesor o sucesor. También es posible insertar un nodo a la izquierda de *node(p)* o la derecha de *node(q)*, o insertar un nodo entre *node(p)* y *node(q)* y reinicializar *p* o *q* al nodo recién insertado. Mediante un esquema de este tipo, siempre es crucial mantener dos apuntadores externos a dos nodos adyacentes de la lista.

### Suma de enteros largos mediante listas doblemente ligadas

Para ilustrar el uso de las listas doblemente ligadas, considérese ampliar la implantación con lista de enteros largos para incluir enteros positivos y negativos. El nodo cabecera de una lista circular que representa un entero largo contiene una indicación que señala si el entero es positivo o negativo.

Para sumar un entero positivo y uno negativo, hay que restar el de menor valor absoluto del de mayor valor y darle al resultado el signo del último. Por lo tanto, se necesita algún método para averiguar cuál de los dos enteros representados como listas circulares tiene el mayor valor absoluto.

El primer criterio que se puede usar para identificar un entero con el mayor valor absoluto es el de la longitud de los enteros (suponiendo que éstos no contienen ceros importantes). La lista con más nodos representa el entero con valor absoluto mayor. Sin embargo, el conteo real del número de nodos implica un recorrido extra de la lista. En lugar de contar el número de nodos, el conteo podría guardarse como parte del nodo cabecera y referenciarse cuando fuese necesario.

Sin embargo, si las dos listas tienen el mismo número de nodos, el entero de mayor valor absoluto será aquél cuyo dígito más significativo sea mayor. Si los primeros dígitos son iguales en ambos enteros, habrá que recorrer la lista del dígito más significativo al menos significativo para determinar qué número es mayor. Obsérvese que este recorrido es en dirección opuesta al que se usa para sumar o restar dos enteros. Como se debe tener la capacidad de recorrer las listas en ambas direcciones, se usan listas doblemente ligadas para representar enteros de este tipo.

Considérese el formato del nodo cabecera. Además de un apuntador izquierdo y uno derecho, la cabecera debe tener la longitud de la lista y una indicación del signo del número. Esas dos porciones de información pueden combinarse en un entero simple cuyo valor absoluto sea la longitud de la lista y cuyo signo sea el del número que se está representando. Sin embargo, si se hiciera así, se destruiría la posibilidad de identificar el nodo cabecera por medio del examen del signo de su campo *info*.

Cuando se representó un entero positivo como una lista circular ligada, un campo *info* de —1 indicaba un nodo cabecera. Con la nueva representación, sin embargo, un nodo cabecera puede contener un campo *info*, como 5, que es válido para cualquier otro nodo de la lista.

Hay varias maneras de remediar este problema. Una es agregar otro campo a cada nodo para indicar si es o no un nodo cabecera. Tal campo podría contener el valor lógico *TRUE* si el nodo es cabecera, y *FALSE* en caso contrario. Esto significa, por supuesto, que cada nodo requeriría más espacio. Una alternativa sería eliminar el contador del nodo cabecera y un campo *info* —1 indicaría un número positivo y un campo *info* —2 uno negativo. Un nodo cabecera sería identificado entonces por su campo *info* negativo. Sin embargo, eso incrementaría el tiempo necesario para comparar dos números, pues sería necesario contar el número de nodos de cada lista. Tales relaciones de espacio/tiempo son comunes en computación, por lo que se tiene que tomar una decisión acerca de qué eficacia debe sacrificarse y cuál debe retenerse.

En nuestro caso, escogemos una tercera opción, que es retener un apuntador externo a la cabecera de la lista. Un apuntador *p* puede identificarse como apuntador a la cabecera si es igual al apuntador externo original; en caso contrario *node(p)* no es un nodo cabecera.

La figura 4.5.6 indica un nodo muestra y la representación de tres enteros como listas doblemente ligadas. Obsérvese que los dígitos menos significativos están a la derecha de la cabecera y que los contadores de los nodos cabecera no incluyen el propio nodo cabecera.

Mediante la representación anterior se presenta una función *compabs* que compara el valor absoluto de dos enteros representados como listas doblemente ligadas. Sus dos parámetros son apuntadores a las cabeceras de la lista y su resultado es 1 si el primero tiene el mayor valor absoluto, —1 si lo tiene el segundo y 0 si el valor absoluto de dos enteros es igual.

```
compabs(p, q)
NODEPTR p, q;
{
 NODEPTR r, s;
 /* comparando el tamaño de los enteros */
 if (abs(p->info) > abs(q->info))
 return(1);
 if (abs(p->info) < abs(q->info))
 return(-1);
 /* ambos tienen el mismo tamaño */
 r = p->left;
 s = q->left;
 /*.... Recorrido de la lista a partir del dígito más significativo */
 while (r != p) {
 if (r->info > s->info)
 return(1);
 if (r->info < s->info)
 return(-1);
 }
}
```

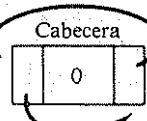
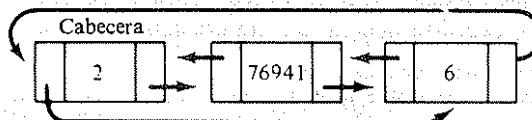
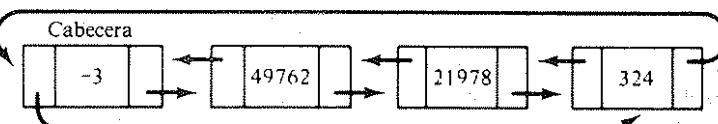
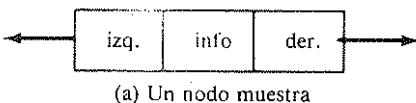


Figura 4.5.6 Enteros con listas doblemente ligadas.

```

 r = r->left;
 s = s->left;
 } /* fin de while */
 /* los valores absolutos son iguales */
 return(0);
} /* fin de compabs */

```

Ahora puede escribirse una función *addif* que acepte dos apuntadores a listas doblemente ligadas, las que representan enteros largos de signos diferentes, donde el valor absoluto del primero no es menor que el del segundo, y que dé como resultado un apuntador a una lista que representa la suma de los enteros. Por supuesto, hay que tener cuidado de eliminar los ceros no significativos de la suma. Para hacerlo, hay que mantener un apuntador *zeroptr* al primer nodo de un conjunto consecutivo de ceros a la izquierda y un indicador *zeroflag* que es *TRUE* si y sólo si el último nodo de la suma generada hasta el momento es 0.

En esta función, *p* apunta al número con mayor valor absoluto y *q* al número con menor valor absoluto. Los valores de esas variables no cambian. Para recorrer

la lista se usan variables auxiliares *pptr* y *qptr*. La suma se forma en una lista apuntada por la variable *r*.

```

NODEPTR addif(p, q)
NODEPTR p, q;
{
 int count;
 NODEPTR pptr, qptr, r, s, zeroptr;
 long int hunthou = 100000L;
 long int borrow, diff;
 int zeroflag;
 /* inicialización de variables */
 count = 0;
 borrow = 0;
 zeroflag = FALSE;
 /* generar un nodo cabecera para la suma */
 r = getnode();
 r->left = r;
 r->right = r;
 /* Recorrido de las dos listas */
 pptr = p->right;
 qptr = q->right;
 while (qptr != q) {
 diff = pptr->info - borrow - qptr->info;
 if (diff >= 0)
 borrow = 0;
 else {
 diff = diff + hunthou;
 borrow = 1;
 }
 /* fin de if */
 /* generar un nuevo nodo e insertarlo */
 /* a la izquierda del nodo cabecera de sum */
 insertleft(r, diff);
 count += 1;
 /* prueba para el nodo cero */
 if (diff == 0) {
 if (zeroflag == FALSE)
 zeroptr = r->left;
 zeroflag = TRUE;
 }
 else
 zeroflag = FALSE;
 pptr = pptr->right;
 qptr = qptr->right;
 } /* fin de while */
 /* Recorrido del resto de la lista p */
 while (pptr != p) {
 diff = pptr->info - borrow;
 if (diff >= 0)
 borrow = 0;
 else
 borrow = 1;
 /* generar un nuevo nodo e insertarlo */
 /* a la izquierda del nodo cabecera de sum */
 insertleft(r, diff);
 count += 1;
 }
}

```

```

 else {
 diff = diff + hunthou;
 borrow = 1;
 } /* fin de if */
 insertleft(r, diff);
 count += 1;
 if (diff == 0) {
 if (zeroflag == FALSE)
 zeroptr = r->left;
 zeroflag = TRUE;
 }
 else
 zeroflag = FALSE;
 pptr = pptr->right;
} /* fin de while */
if (zeroflag == TRUE) /* eliminar ceros iniciales */
 while (zeroptr != r) {
 s = zeroptr;
 zeroptr = zeroptr->right;
 delete(s, &diff);
 count -= 1;
 } /* fin de if ... de while */
/* insertar el tamaño y el signo en el nodo cabecera */
if (p->info > 0)
 r->info = count;
else
 r->info = -count;
return(r);
} /* fin de adddiff */

```

También se puede escribir una función *addsame* para sumar dos números de signo igual. Esta es muy similar a la función *addint* de la implantación anterior, excepto en que trata con una lista doblemente ligada y en que tiene que mantener actualizado el número de nodos de la suma.

Por medio de esas rutinas, se puede escribir una versión nueva de *addint* que sume dos enteros representados por listas doblemente ligadas.

```

NODEPTR addint(p, q)
NODEPTR p, q;
{
 /* verificar si los enteros son del mismo signo */
 if (p->info * q->info > 0)
 return(addsame(p, q));
 /* determinar cuál tiene el mayor valor absoluto */
 if (compabs(p, q) > 0)
 return(adddiff(p, q));
 else
 return(adddiff(q, p));
} /* fin de addint */

```

## EJERCICIOS

- 4.5.1. Escriba un algoritmo y una rutina C para ejecutar cada una de las operaciones del ejercicio 4.2.3 para listas circulares. ¿Qué es más eficaz de las listas circulares comparado con las lineales? ¿Qué es menos eficaz?
- 4.5.2. Escriba otra vez la rutina *place* de la sección 4.3 para insertar un nuevo elemento de una lista circular ordenada.
- 4.5.3. Escriba un programa para resolver el problema de Josephus usando un arreglo en lugar de una lista circular. ¿Por qué es más eficaz una lista circular?
- 4.5.4. Considere la siguiente variación del problema de Josephus. Un grupo de personas está en un círculo y cada una de ellas escoge un entero positivo. Se escoge un entero positivo  $n$  y uno de los nombres. Se comienza con la persona cuyo nombre fue elegido, se cuenta alrededor del círculo en el sentido de las manecillas del reloj y se elimina la persona  $n$ -ésima. Para continuar el conteo, se usa el entero positivo elegido por la persona recién eliminada. Cada vez que se elimina una persona, se usa el número de ésta para eliminar a la siguiente. Por ejemplo, supóngase que las cinco personas son  $A, B, C, D$  y  $E$ , que éstas eligieron los enteros 3, 4, 6, 2 y 7 respectivamente y que el primer número elegido es 2. Entonces, si se comienza con  $A$ , el orden con el que se elimina a las personas del círculo es  $B, A, E, C$ , dejando a  $D$  como el último del círculo. Escriba un programa que lea un grupo de líneas de entrada. Cada línea de entrada, excepto la primera y la última, contiene un nombre y un entero positivo escogido por esa persona. El orden de los nombres de los datos es el orden en el sentido de las manecillas de reloj de las personas del círculo, y el conteo debe comenzar con el primer nombre de la entrada. La primera línea de entrada contiene el número de personas del círculo. La última contiene sólo un entero positivo simple que representa el conteo inicial. El programa imprime el orden en que se eliminan las personas del círculo.
- 4.5.5. Escriba una función *multint*( $p, q$ ) en C para multiplicar dos enteros largos representados individualmente por listas circulares ligadas.
- 4.5.6. Escriba un programa para imprimir el número 100é-simo de Fibonacci.
- 4.5.7. Escriba un algoritmo y una rutina C para ejecutar cada una de las operaciones del ejercicio 4.2.3 para listas doblemente ligadas. ¿Cuáles son más eficaces, las de las listas doblemente ligadas o las de ligadura simple? ¿Cuáles son menos eficaces?
- 4.5.8. Suponga que un campo de apuntador simple en cada nodo de una lista doblemente ligada contiene la suma de los apuntadores a los nodos antecesor y sucesor, como se describe en el texto. Dados los apuntadores  $p$  y  $q$  a dos nodos adyacentes de tal lista, escribir rutinas en C para insertar un nodo a la derecha de *node*( $q$ ), a la izquierda de *node*( $p$ ) y entre *node*( $p$ ) y *node*( $q$ ), modificando  $p$  para que apunte al nodo recién insertado. Escriba una rutina adicional para borrar *node*( $q$ ), reinicializando  $q$  como el sucesor del nodo.
- 4.5.9. Suponga que *first* y *last* son apuntadores externos al primero y al último nodo de una lista doblemente ligada representada igual que en el ejercicio 4.5.8. Escriba rutinas en C para implantar las operaciones del ejercicio 4.2.3 para una lista de este tipo.
- 4.5.10. Escriba una rutina *addsame* para sumar dos enteros largos del mismo signo representados por listas doblemente ligadas.
- 4.5.11. Escriba una función *multint*( $p, q$ ) en C, para multiplicar dos enteros largos representados por listas doblemente ligadas circulares.
- 4.5.12. ¿Cómo se puede representar un polinomio de tres variables ( $x, y$  y  $z$ ) por medio de una lista circular? Cada nodo debe representar un término y contener las potencias de

$x$ ,  $y$  y  $z$ , así como el coeficiente de dicho término. Escribir funciones en C que hagan lo siguiente:

- a. Sume dos polinomios de ese tipo.
- b. Multiplique dos polinomios de ese tipo.
- c. Calcule la derivada parcial de un polinomio semejante con respecto a cualquiera de sus variables.
- d. Evalúe un polinomio semejante para valores dados de  $x$ ,  $y$  y  $z$ .
- e. Divida un polinomio por otro de ese tipo, creando un polinomio cociente y un polinomio restante.
- f. Integre un polinomio semejante con respecto a cualquiera de sus variables.
- g. Imprima la representación de semejante polinomio.
- h. Dados cuatro polinomios de ese tipo:  $f(x, y, z)$ ,  $g(x, y, z)$ ,  $h(x, y, z)$  e  $i(x, y, z)$ , calcule el polinomio  $f(g(x, y, z), h(x, y, z) i(x, y, z))$ .

## Arboles

En este capítulo se analizará una estructura de datos que es útil en muchas aplicaciones: el árbol. Se definen varias formas de esta estructura de datos y se muestra cómo pueden representarse en C y cómo se pueden aplicar para resolver una amplia gama de problemas. Al igual que con las listas, los árboles se tratan principalmente como estructuras de datos y no como un tipo de datos. Es decir, el interés inicial es por la implantación más que por la definición matemática.

### 5.1. ARBOLES BINARIOS

Un **árbol binario** es un conjunto finito de elementos que o está vacío o está dividido en tres subconjuntos desarticulados. El primer subconjunto contiene un solo elemento llamado **raíz** del árbol. Los otros dos son en sí mismos árboles binarios, llamados **subárboles izquierdo** y **derecho** del árbol original. Un subárbol izquierdo o derecho puede estar vacío. Cada elemento de un árbol binario se llama **nodo** del árbol.

En la figura 5.1.1 se muestra un método convencional de dibujar un árbol binario. Este árbol consiste de nueve nodos y tiene a  $A$  como raíz. Su subárbol izquierdo tiene a  $B$  como raíz y su subárbol derecho a  $C$ . Esto queda señalado por las dos ramas que salen de  $A$ ; hacia  $B$  a la izquierda y hacia  $C$  a la derecha. La ausencia de ramas indica que es un subárbol vacío. Por ejemplo, tanto el subárbol izquierdo del árbol binario que tiene raíz en  $C$  como el subárbol derecho del árbol binario que tiene raíz en  $E$  están vacíos. Los árboles binarios con raíz en  $D$ ,  $G$ ,  $H$  e  $I$  tienen subárboles derecho e izquierdo vacíos.

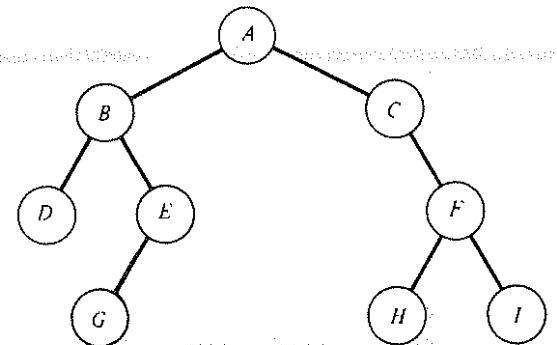
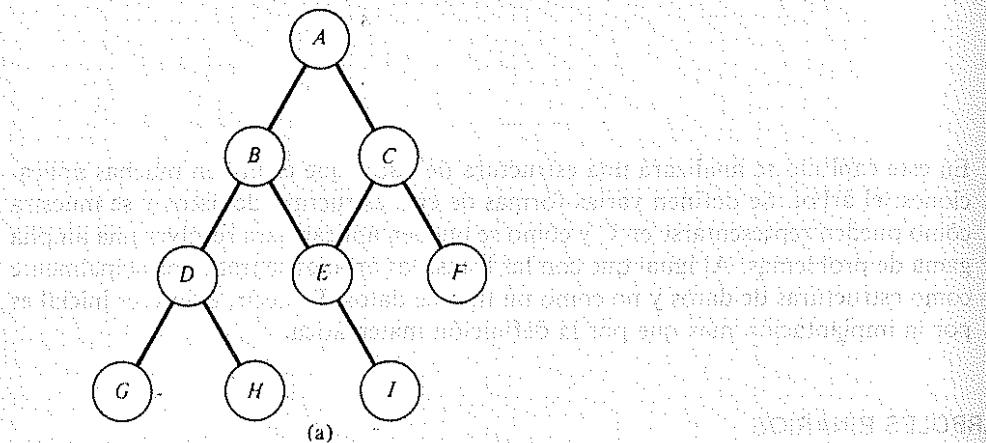


Figura 5.1.1. Un árbol binario.

La figura 5.1.2 ilustra algunas estructuras que no son árboles binarios. Hay que asegurarse de haber entendido por qué no lo son.

Si  $A$  es la raíz de un árbol binario y  $B$  es la raíz de su subárbol derecho o izquierdo entonces  $A$  se llama el *padre* de  $B$  y  $B$  el *hijo izquierdo* o *derecho* de  $A$ . Un nodo que no tiene hijos (como  $D$ ,  $G$ ,  $H$  o  $I$  de la figura 5.1.1) se llama *hoja*. El nodo



(a)

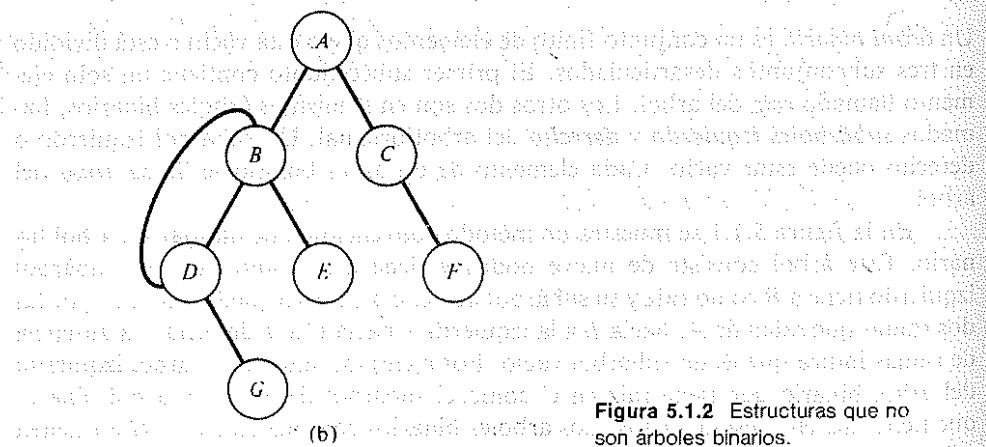


Figura 5.1.2. Estructuras que no son árboles binarios.

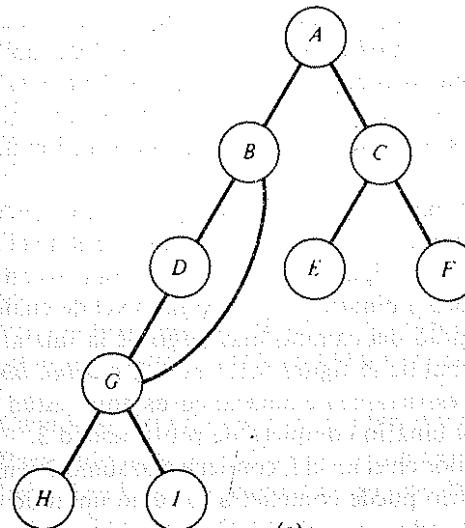


Figura 5.1.2 (cont.).

$n_1$  es un *ancestro* del nodo  $n_2$  (y  $n_2$  es un *descendiente* de  $n_1$ ) si  $n_1$  es el padre de  $n_2$  o el de algún ancestro de  $n_2$ . Por ejemplo, en el árbol de la figura 5.1.1,  $A$  es un ancestro de  $G$  y  $H$  es un descendiente de  $C$ , pero  $E$  no es ni ancestro ni descendiente de  $C$ . Un nodo  $n_2$  es un *descendiente izquierdo* de un nodo  $n_1$  si  $n_2$  es el hijo izquierdo de  $n_1$  o un descendiente del hijo izquierdo de  $n_1$ . Un *descendiente derecho* puede definirse de manera similar. Dos nodos son *hermanos* si son hijos derecho e izquierdo del mismo parente.

Aunque los árboles naturales crecen con sus raíces en la tierra y sus hojas en el aire, los científicos de la computación representan, casi a nivel universal, las estructuras de datos como árboles con la raíz en la cúspide y las hojas en el fondo. La dirección de la raíz hacia las hojas se llama “hacia abajo”, y de las hojas a la raíz “hacia arriba”. Ir de las hojas a la raíz se llama “trepar” a un árbol; e ir de la raíz a las hojas, “descender” de un árbol.

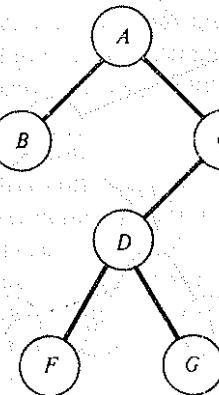


Figura 5.1.3. Un árbol estrictamente binario.

Si un nodo que no es una hoja de un árbol binario tiene subárboles izquierdo y derecho no-vacíos, el árbol se llama **árbol estrictamente binario**. Así, el árbol de la figura 5.1.3 es estrictamente binario, mientras que el de la figura 5.1.1, no lo es (porque los nodos C y E tienen, cada uno, un solo hijo). Un árbol estrictamente binario con  $n$  hojas tiene siempre  $2n - 1$  nodos. Al lector se le deja la comprobación de esto como ejercicio.

El **nivel** de un nodo en un árbol binario se define como sigue. La raíz del árbol tiene nivel 0 y el nivel de cualquier otro nodo del árbol es uno más que el nivel de su padre. Por ejemplo, en el árbol binario de la figura 5.1.1, el nodo E está en el nivel 2 y el H en el 3. La **profundidad** de un árbol binario es el máximo nivel de cualquier hoja del árbol. Esto es igual a la longitud del camino más largo de la raíz a cualquier hoja. Así, la profundidad del árbol de la figura 5.1.1 es 3. Un **árbol binario completo** de profundidad  $d$  es el árbol estrictamente binario cuyas hojas están en el nivel  $d$ . La figura 5.1.4 ilustra el árbol binario completo de profundidad 3.

Si un árbol binario contiene  $m$  nodos en el nivel  $l$ , contiene a lo sumo  $2m$  nodos en el nivel  $l + 1$ . Como un árbol binario puede contener a lo sumo un nodo en el nivel 0 (la raíz), contendrá a lo sumo  $2^l$  nodos en el nivel  $l$ . Un árbol binario completo de profundidad  $d$  es el árbol binario de profundidad  $d$  que contiene la cantidad exacta  $2^d$  nodos en cada nivel  $l$  entre 0 y  $d$ . (Esto equivale a decir que es el árbol binario de profundidad  $d$  que contiene la cantidad exacta  $2^d$  nodos en el nivel  $d$ ). El número total de nodos de un árbol binario completo de profundidad  $d$ ,  $tn$ , es igual a la suma del número de nodos de cada nivel entre 0 y  $d$ . Así,

$$tn = 2^0 + 2^1 + 2^2 + \dots + 2^d = \sum_{j=0}^{d-1} 2^j$$

Por inducción, se puede mostrar que esta suma es igual a  $2^{d+1} - 1$ . Como todas las hojas de un árbol de este tipo están en el nivel  $d$ , el árbol contiene  $2^d$  hojas y, en consecuencia,  $2^d - 1$  nodos que no son hojas.

De manera análoga, si se sabe el número de nodos,  $tn$ , de un árbol binario completo, se puede calcular su profundidad,  $d$ , por medio de la ecuación  $tn = 2^{d+1} - 1$ .

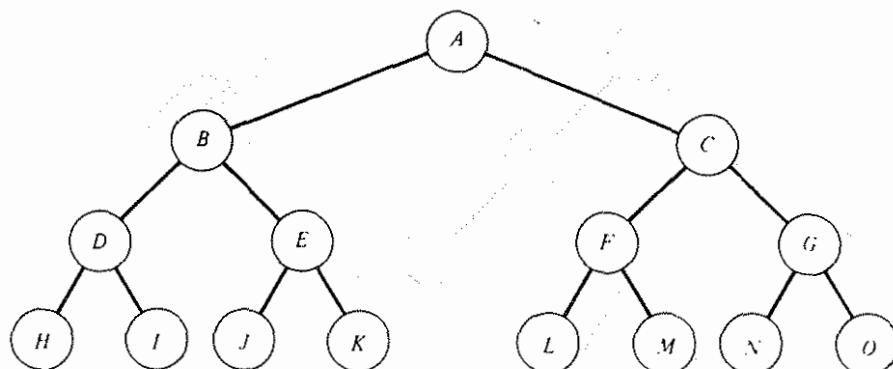


Figura 5.1.4 Un árbol binario completo de nivel 3.

$d$  es igual a 1 menos que el número de veces que hay que multiplicar 2 por sí mismo para alcanzar  $tn + 1$ . En matemáticas,  $\log_2 x$  se define como el número de veces que hay que multiplicar  $b$  por sí mismo para alcanzar  $x$ . Por lo tanto, se puede decir que, en un árbol binario completo,  $d$  es igual a  $\log_2(tn + 1) - 1$ . Por ejemplo, el árbol binario completo de la figura 5.1.4 contiene 15 nodos y es de profundidad 3. Adviértase que 15 es igual a  $2^3 + 1 - 1$  y 3 es igual a  $\log_2(15 + 1) - 1$ .  $\log_2 x$  es mucho más pequeño que  $x$  [por ejemplo,  $\log_2 1024$  es igual a 10 y  $\log_2 1000000$  es menor que 20]. La importancia de un árbol binario completo es que es el árbol binario con el número de nodos máximo para una profundidad determinada. Dicho de otro modo, aunque un árbol binario completo contenga muchos nodos, la distancia de la raíz a cualquier hoja (la profundidad del árbol) es relativamente pequeña.

Un árbol binario de profundidad  $d$  es un **árbol binario quasi-completo** si:

1. Cada hoja del árbol está en el nivel  $d$  o en el nivel  $d - 1$ .
2. Para todo nodo  $nd$  del árbol con un descendiente derecho en el nivel  $d$ , todos los descendientes izquierdos de  $nd$  que sean hojas también estarán en el nivel  $d$ .

El árbol estrictamente binario de la figura 5.1.5a no es quasi-completo, pues contiene hojas en los niveles 1, 2 y 3, con lo que viola la condición 1. El árbol estrictamente binario de la figura 5.1.5b satisface la condición 1, ya que todas las hojas están en el nivel 2 o en el 3. Sin embargo, la condición 2 no se satisface, pues A tiene un descendiente derecho en el nivel 3 (J), pero también tiene un descendiente izquierdo en el nivel 2 (E), el que es una hoja. El árbol estrictamente binario de la figura 5.1.5c satisface ambas condiciones, la 1 y la 2, y es, en consecuencia, un árbol binario quasi-completo. Aunque el árbol binario de la figura 5.1.5d es también un árbol binario quasi-completo, no es estrictamente binario, pues el nodo E tiene un hijo izquierdo pero no uno derecho. (Hay que observar que muchos textos se refieren a tales árboles como “árboles binarios completos” y no como “árboles binarios quasi-completos”. Otros textos usan el término “completo” o “completamente binario” para referirse al concepto que aquí se usa como “estrictamente binario”. Aquí se aplican los términos “estrictamente binario”, “completo” y “cuasi-completo” como se definen en el texto.)

Los nodos de un árbol binario quasi-completo pueden enumerarse de manera que se le asigne 1 a la raíz, al hijo izquierdo se le asigna el doble del número asignado a su padre y al derecho uno más que el doble del número asignado a su padre. Las figuras 5.1.5c y d ilustran esta técnica de enumeración. A cada nodo de un árbol binario quasi-completo se le asigna un número único que define su posición dentro del árbol.

Un árbol estrictamente binario quasi-completo con  $n$  hojas tiene  $2n - 1$  nodos, como ocurre con cualquier otro árbol estrictamente binario con  $n$  hojas. Un árbol binario quasi-completo con  $n$  hojas que no sea estrictamente binario tiene  $2n$  nodos. Hay dos árboles binarios quasi-completos distintos con  $n$  hojas, uno de los cuales es estrictamente binario y el otro no. Por ejemplo, los árboles de las figuras 5.1.5c y d son quasi-completos y tienen 5 hojas; sin embargo, el árbol de la figura 5.1.5c es estrictamente binario y el de la figura 5.1.5d no.

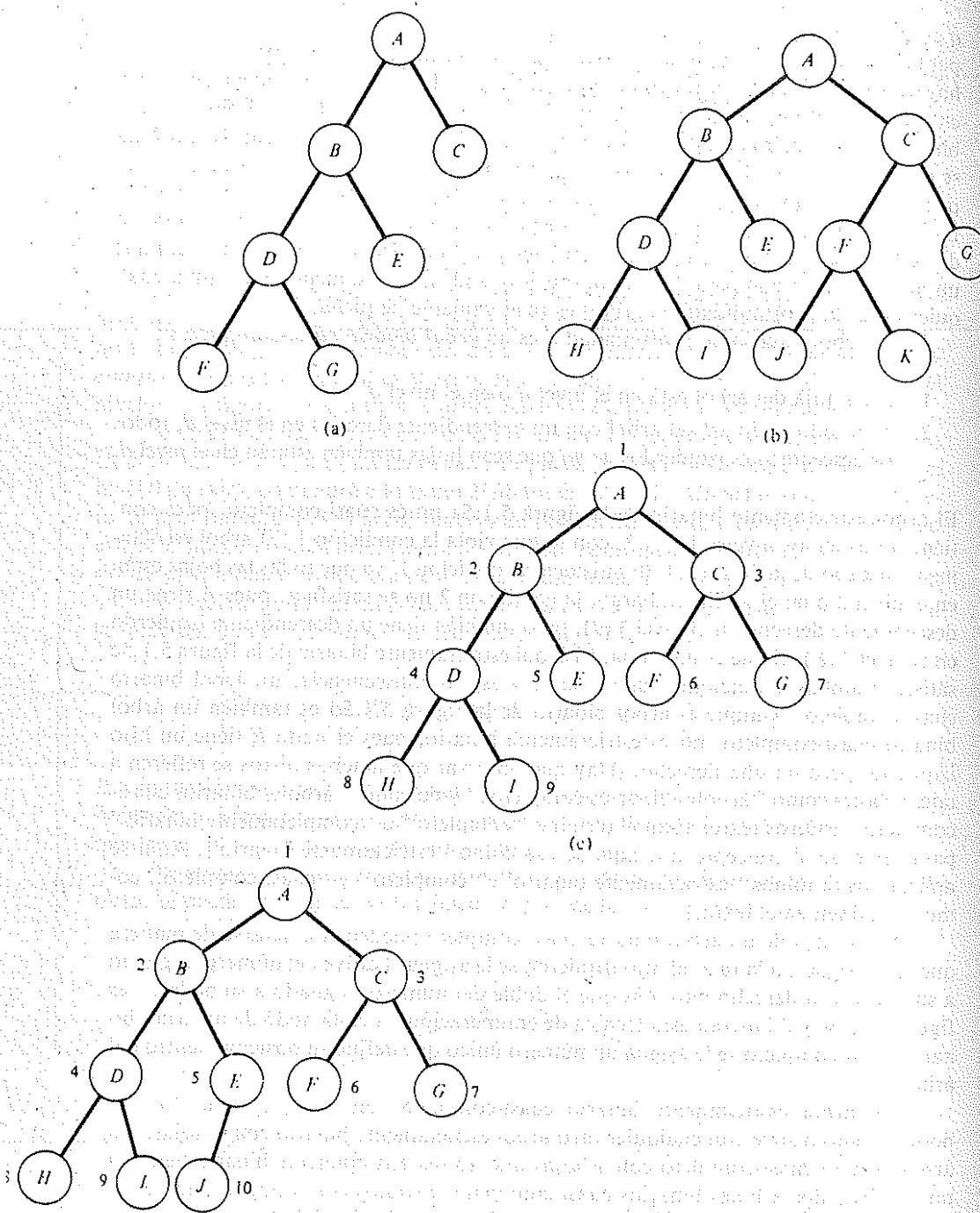


Figura 5.1.5 Enumeración de nodos para árboles binarios cuasi-completos.

Hay sólo un árbol binario cuasi-completo con  $n$  nodos. Ese árbol es estrictamente binario si y sólo si  $n$  es impar. Por lo tanto, el árbol de la figura 5.1.5c es el único árbol binario cuasi-completo con nueve nodos y es estrictamente binario porque 9 es impar, mientras que el de la figura 5.1.5d es el único árbol binario cuasi-completo con 10 nodos y no es estrictamente binario pues 10 es par.

Un árbol binario cuasi-completo de profundidad  $d$  es intermedio entre el árbol binario completo de profundidad  $d - 1$ , que contiene  $2^d - 1$  nodos, y el árbol binario completo de profundidad  $d$ , que contiene  $2^{d+1} - 1$  nodos. Si  $tn$  es el número total de nodos de un árbol binario cuasi-completo, su profundidad es el entero más grande que sea menor o igual que  $\log_2 tn$ . Por ejemplo, los árboles binarios cuasi-completos con 4, 5, 6 y 7 nodos tienen profundidad 2, y los árboles binarios cuasi-completos con 8, 9, 10, 11, 12, 13, 14 y 15 nodos tienen profundidad 3.

### Operaciones con árboles binarios

Existen varias operaciones primitivas que pueden aplicarse a árboles binarios. Si  $p$  es un apuntador a un nodo  $nd$  de un árbol binario, la función  $info(p)$  da como resultado los contenidos de  $nd$ . Las funciones  $left(p)$ ,  $right(p)$ ,  $father(p)$  y  $brother(p)$  dan como resultado apuntadores al hijo izquierdo, hijo derecho, padre y hermano de  $nd$ , respectivamente. Esas funciones dan como resultado el apuntador  $null$  si  $nd$  no tiene hijo izquierdo, hijo derecho, padre o hermano. Como consecuencia, las funciones lógicas  $isleft(p)$  e  $isright(p)$  dan como resultado *verdadero* (*true*) si  $nd$  es hijo izquierdo o derecho, respectivamente, de algún otro nodo del árbol, y *falso* (*false*) en caso contrario.

Obsérvese que las funciones  $isleft(p)$ ,  $isright(p)$  y  $brother(p)$  pueden implantarse mediante las funciones  $left(p)$ ,  $right(p)$  y  $father(p)$ . Por ejemplo,  $isleft$  puede implantarse como sigue:

```
q = father(p);
if (q == null)
 return(false); /* p apunta a la raíz */
if (left(q) == p)
 return(true);
return(false);
```

o, de manera más simple, como  $father(p) \&& == left(father(p)).isright$  puede implantarse de manera similar, o llamando a  $isleft$ .  $brother(p)$  puede implantarse mediante  $isleft$  o  $isright$  como sigue:

```
if (father(p) == null)
 return(null); /* p apunta a la raíz */
if (isleft(p))
 return(right(father(p)));
return(left(father(p)));
```

Las operaciones *maketree*, *setleft* y *setright* son útiles en la construcción de un árbol binario. *maketree(x)* crea un nuevo árbol binario que consiste en un solo nodo

con campo de información  $x$  y da como resultado un apuntador a dicho nodo.  $setleft(p, x)$  acepta un apuntador  $p$  a un nodo de un árbol binario sin hijo izquierdo. Esto crea un nuevo hijo izquierdo de  $node(p)$  con campo de información  $x$ .  $setright(p, x)$  es análoga a  $setleft$ , excepto en que crea un hijo derecho de  $node(p)$ .

### Aplicaciones de árboles binarios

Un árbol binario es una estructura de datos útil cuando en cada punto de un proceso hay que tomar una decisión de doble opción. Por ejemplo, supóngase que se desea encontrar todos los duplicados en una lista de números. Una manera de hacerlo es comparar cada número con todos los que lo preceden. Sin embargo, esto implica un enorme número de comparaciones.

El número de comparaciones puede reducirse mediante un árbol binario. El primer número de la lista se coloca en un nodo que se establece como la raíz de un árbol binario con subárboles izquierdo y derecho vacíos. Cada número sucesivo de la lista se compara después con el número que está en la raíz. Si hay coincidencia, se obtiene un duplicado. Si es más pequeño, se examina el subárbol izquierdo; si es mayor, se examina el subárbol derecho. Si el subárbol está vacío, el número no es un duplicado y se coloca en un nodo nuevo en esa posición del árbol. Si el subárbol no está lleno, el número se compara con los contenidos de la raíz del subárbol y se repite todo el proceso con el subárbol. A continuación se presenta un algoritmo para hacer esto.

```
/*
 * leer el primer número e insertarlo en un
 * árbol binario de un sólo nodo
 */
scanf ("%d", &number);
tree = maketree(number);
while (hay números rezagados en la entrada)
{
 scanf ("%d", &number);
 p = q = tree;
 while (number != info(p) && q != NULL)
 {
 p = q;
 if (number < info(p))
 q = left(p);
 else
 q = right(p);
 }
 /* fin de while */
 if (number == info(p))
 printf ("%d %s\n", number, "es un duplicado");
 /* insertar number a la derecha o izquierda de p */
 else if (number < info(p))
 setleft(p, number);
 else
 setright(p, number);
} /* fin de while */
}
```

La figura 5.1.6 ilustra el árbol construido a partir de los datos de entrada 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5.

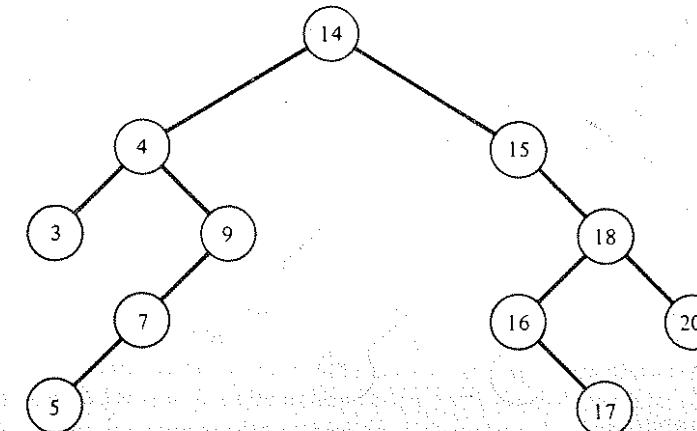


Figura 5.1.6: Un árbol binario construido para encontrar duplicados.

Otra operación común es *recorrer* un árbol binario; esto es, pasar a través del árbol, enumerando cada uno de sus nodos una vez. Quizá sólo se deseé imprimir los contenidos de cada nodo al enumerarlos, o procesar los nodos en otra forma. En cualquier caso, se habla de *visitar* cada nodo al enumerar éste.

El orden en que se visitan los nodos de una lista lineal es, de manera clara, del primero al último. Sin embargo, no hay tal orden lineal “natural” para los nodos de un árbol. Así, se usan diferentes ordenamientos para el recorrido en diferentes casos. Enseguida se definen tres de estos métodos de recorrido. En cada uno de ellos, no hay que hacer nada para recorrer un árbol binario vacío. Todos los métodos se definen en forma recursiva, de manera que el recorrido de un árbol binario implica la visita de la raíz y el recorrido de sus subárboles izquierdo y derecho. La única diferencia entre los métodos es el orden en que se ejecutan esas tres operaciones.

Para recorrer un árbol binario lleno en *preorden* (conocido también como *orden con prioridad a la profundidad* o *depth-first order*), se ejecutan las siguientes tres operaciones:

1. Visitar la raíz.
2. Recorrer el subárbol izquierdo en preorden.
3. Recorrer el subárbol derecho en preorden.

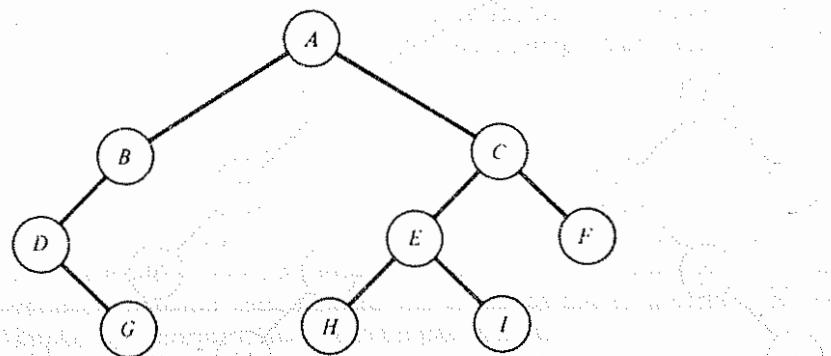
Para recorrer un árbol binario lleno en *orden* (u *orden simétrico*):

1. Recorrer el subárbol izquierdo en orden.
2. Visitar la raíz.
3. Recorrer el subárbol derecho en orden.

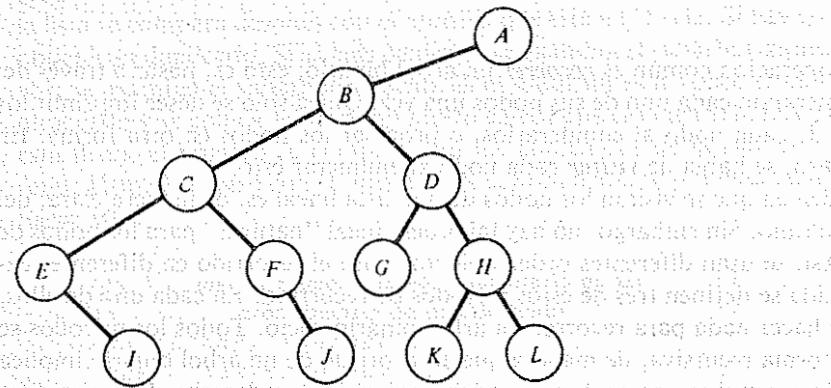
Para recorrer un árbol binario lleno en *postorden*:

1. Recorrer el subárbol izquierdo en postorden.
2. Recorrer el subárbol derecho en postorden.

### 3. Visitar la raíz.



Preorden: ABDGCEHF  
En orden: DGBAHEIF  
Postorden: GDBHIEFCA



Preorden: ABCEIFJDGHKL  
En orden: EICFJBGDKHLA  
Postorden: IEJFCGKLHDBA

Figura 5.1.7 Árboles binarios y sus recorridos.

La figura 5.1.7 ilustra dos árboles binarios y sus recorridos en preorden, orden y postorden.

Muchos algoritmos que usan árboles binarios proceden en dos fases. La primera construye un árbol binario y la segunda lo recorre. Como ejemplo de un algoritmo de este tipo, considérese el siguiente método de ordenamiento. Dada una lista de números en un archivo de entrada, se desea imprimirlos en orden ascendente. Luego de leer los números, se pueden insertar en un árbol binario como el de la figura 5.1.6. Sin embargo, a diferencia del algoritmo usado para encontrar duplicados, en este caso los valores repetidos se colocan en el árbol. Al comparar un número con los contenidos de un nodo del árbol, se toma una rama izquierda si el número es menor que los contenidos del nodo y una rama derecha si es mayor o igual que los contenidos del nodo. Así, la lista de entrada es:

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5

se produce el árbol binario de la figura 5.1.8.

Un árbol binario de este tipo tiene la propiedad de que todos los elementos del subárbol izquierdo de un nodo  $n$  son menores que los contenidos de  $n$  y todos los elementos del subárbol derecho  $n$  son mayores o iguales que los contenidos de  $n$ . Un árbol binario que tenga esta propiedad se llama **árbol de búsqueda binaria**. Si un árbol de búsqueda binaria se recorre en orden (izquierdo, raíz, derecho) y se imprimen los contenidos de cada nodo cuando se visita el mismo, los números se imprimen en orden ascendente. Hay que convencerse de que éste es el caso para el árbol de búsqueda binaria de la figura 5.1.8. Los árboles de búsqueda binarios y su uso en ordenamiento y búsquedas se analizan en las secciones 6.3 y 7.2.

Como otra aplicación de los árboles binarios, considérese el siguiente método de representación de una expresión que contiene operandos y operadores binarios por medio de un árbol estrictamente binario. La raíz del árbol estrictamente binario contiene un operador que se debe aplicar a los resultados de evaluar las expresiones representadas por los subárboles izquierdo y derecho. Un nodo que represente un operador no es una hoja, mientras que uno que represente un operando es una hoja. La figura 5.1.9 ilustra algunas expresiones y sus representaciones con árboles. (El carácter “\$” se usa una vez más para denotar la exponentiación.)

Ahora se verá qué ocurre cuando estos árboles de expresiones binarias se recorren. Recorrer un árbol en preorden significa que el operador (la raíz) precede a sus dos operandos (los subárboles). Así, un recorrido en preorden conduce a la expresión en forma prefija. (Para las definiciones de las formas prefija y postfija de

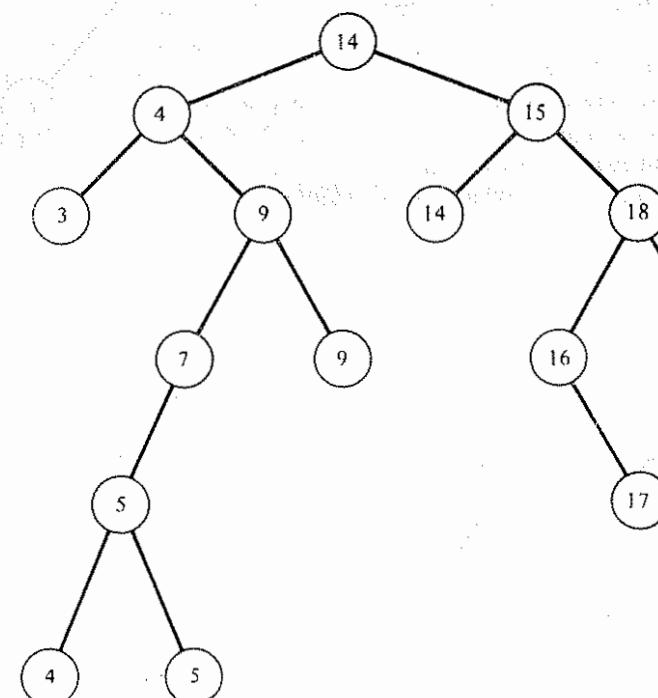
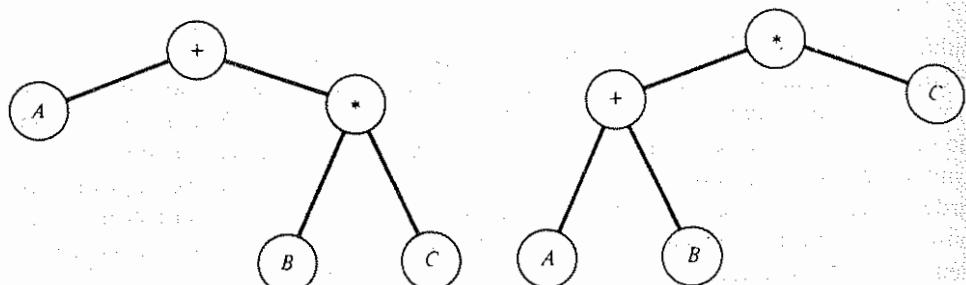
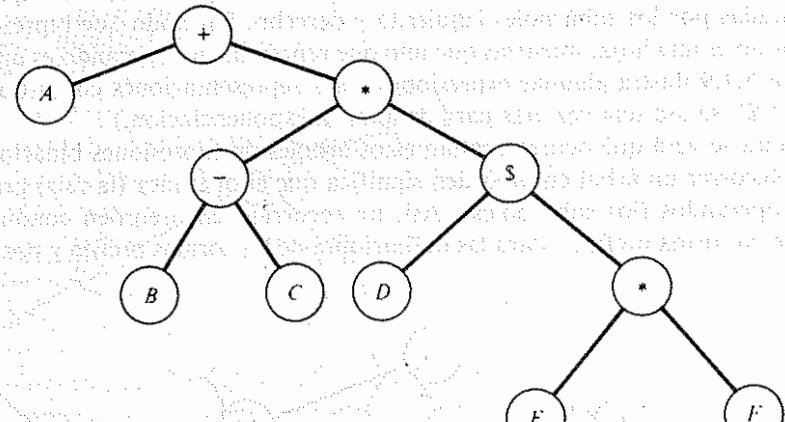


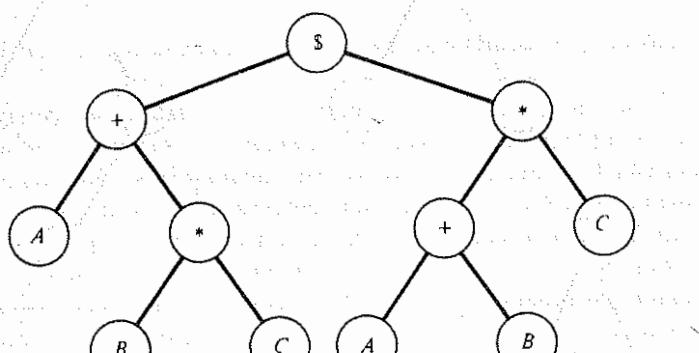
Figura 5.1.8 Un árbol binario construido para un ordenamiento.



(a)  $A + B * C$  (b)  $(A + B) * C$



(c)  $A + (B - C) * D\$ (E * F)$



(d)  $(A + B * C) \$ ((A + B) * C)$

Figura 5.1.9 Expresiones y sus representaciones mediante árboles binarios.

una expresión aritmética, ver las secciones 2.3 y 3.3). Recorrer los árboles binarios de la figura 5.1.9 da como resultado las formas prefijas

$$+ A * BC \quad (\text{Figura 5.1.9a})$$

$$* + ABC \quad (\text{Figura 5.1.9b})$$

$$+ A * - BC \$ D * EF \quad (\text{Figura 5.1.9c})$$

$$\$ + A * BC * + ABC \quad (\text{Figura 5.1.9d})$$

De manera similar, recorrer el árbol binario de una expresión en postorden coloca un operador después de sus dos operandos, de tal suerte que un recorrido en postorden produce la forma posfija de la expresión. Los recorridos en postorden de los árboles binarios de la figura 5.1.9 dan como resultado las formas posfijas

$$ABC * + \quad (\text{Figura 5.1.9a})$$

$$AB + C * \quad (\text{Figura 5.1.9b})$$

$$ABC - DEF * \$ * + \quad (\text{Figura 5.1.9c})$$

$$ABC * + AB + C * \$ \quad (\text{Figura 5.1.9d})$$

¿Qué ocurre si el árbol binario de una expresión se recorre en orden? Ya que la raíz (el operador) se visita después de los nodos del subárbol izquierdo y antes de los del derecho (los dos operandos), puede esperarse que un recorrido en orden conduzca a la forma infija de la expresión. De hecho, cuando se recorre el árbol binario de la figura 5.1.9a, se obtiene la expresión infija  $A + B * C$ . Sin embargo, el árbol de expresión binaria no contiene paréntesis, pues el orden de las operaciones se implica por la estructura del árbol. Así, una expresión cuya forma infija requiere paréntesis para modificar las reglas de prioridad convencionales no se puede recuperar mediante un simple recorrido en orden. Los recorridos en orden de los árboles de la figura 5.1.9 conducen a las expresiones

$$+ A * BC \quad (\text{Figura 5.1.9a})$$

$$A + B * C \quad (\text{Figura 5.1.9b})$$

$$A + B - C * D * E * F \quad (\text{Figura 5.1.9c})$$

$$A + B * C * A + B * C \quad (\text{Figura 5.1.9d})$$

que son correctas, excepto por los paréntesis.

## EJERCICIOS

- 5.1.1. Pruebe que la raíz de un árbol binario es un ancestro de todo nodo del árbol, excepto de ella misma.
- 5.1.2. Pruebe que un nodo de un árbol binario tiene a lo sumo un parente.

5.1.3. ¿Cuántos ancestros tiene un nodo de un árbol binario que esté en el nivel  $n$ ? Compruebe su respuesta.

5.1.4. Escriba algoritmos recursivos y no recursivos para determinar:

- el número de nodos de un árbol binario
- la suma de los contenidos de todos los nodos de un árbol binario
- la profundidad de un árbol binario

5.1.5. Escriba un algoritmo para determinar si un árbol binario es

- estrictamente binario
- completo
- cuasi-completo

5.1.6. Pruebe que un árbol estrictamente binario con  $n$  hojas contiene  $2n - 1$  nodos.

5.1.7. Dado un árbol estrictamente binario con  $n$  hojas, sea  $level(i)$  para  $i$  entre 1 y  $n$  igual al nivel de la  $i$ -ésima hoja. Pruebe que

$$\sum_{i=1}^n \frac{1}{2^{level(i)}} = 1$$

5.1.8. Pruebe que los nodos de un árbol estrictamente binario cuasi-completo con  $n$  hojas pueden enumerarse desde 1 hasta  $2n - 1$  de manera tal que el número asignado al hijo izquierdo del nodo numerado con  $i$  sea  $2i$  y el asignado al hijo derecho numerado con  $i$  sea  $2i + 1$ .

5.1.9. Dos árboles binarios son *similares* si ambos están vacíos, o no están vacíos, sus subárboles izquierdos son similares y los derechos también. Escriba un algoritmo para determinar si dos árboles binarios son similares.

5.1.10. Dos árboles binarios son *similarmente especulares* si ambos están vacíos o no vacíos, y el subárbol izquierdo de cada uno es reflejo del subárbol derecho del otro. Escriba un algoritmo para determinar si dos árboles binarios son similarmente especulares.

5.1.11. Escriba algoritmos para determinar si un árbol binario es o no similar y similarmente especular (ver el ejercicio previo) a algún subárbol de otro.

5.1.12. Desarrolle un algoritmo para encontrar duplicados en una lista de números sin usar un árbol binario. Si en la lista hay  $n$  números distintos, ¿cuántas veces se deben comparar dos números por uniformidad con su algoritmo? ¿Qué ocurre si todos los  $n$  números son iguales?

5.1.13. a. Escriba un algoritmo que acepte un apuntador a un árbol de búsqueda binaria y elimine el elemento más pequeño del árbol.  
b. Muestre cómo implantar una cola de prioridad ascendente (ver sección 4.1) como un árbol de búsqueda binaria. Presente algoritmos para las operaciones *pqinsert* y *pqmindelete* en un árbol de búsqueda binaria.

5.1.14. Escriba un algoritmo que acepte un árbol binario con el que se representa una expresión y que dé como resultado la versión *infija* de la expresión que contenga sólo aquellos paréntesis necesarios.

## 5.2. REPRESENTACIONES DE ARBOLES BINARIOS

En esta sección se examinarán varios métodos de implantación de árboles binarios en C y se presentarán rutinas que los construyen y recorren. También se presentan algunas aplicaciones adicionales de árboles binarios.

## Representación con nodos de árboles binarios

Al igual que en el caso de los nodos de una lista, los nodos de un árbol pueden implantarse como un arreglo de elementos o como asignación de una variable dinámica. Cada nodo contiene los campos *info*, *left*, *right* y *father*. Los campos *left*, *right* y *father* de un nodo apuntan a los nodos hijo izquierdo, hijo derecho y padre, respectivamente. Por medio de la implantación con arreglo, se puede declarar:

```
#define NUMNODES 500
struct nodetype {
 int info;
 int left;
 int right;
 int father;
};
struct nodetype node[NUMNODES];
```

Con esta representación, las operaciones *info(p)*, *left(p)*, *right(p)* y *father(p)* están implantadas mediante referencias a *node[p].info*, *node[p].left*, *node[p].right* y *node[p].father*, respectivamente. Las operaciones *isleft(p)*, *isright(p)* y *brother(p)* pueden implantarse en términos de las operaciones *left(p)*, *right(p)* y *father(p)*, como se describe en la sección precedente.

Para implantar *isleft* e *isright* de manera más eficaz, también se puede incluir dentro de cada nodo un indicador adicional *isleft*. El valor de este indicador es TRUE si el nodo es un hijo izquierdo y FALSE si no lo es. La raíz está identificada sólo por un valor NULL (0) en su campo *father*. Es común que el apuntador externo al árbol apunte a su raíz.

Otra posibilidad es que el signo del campo *father* sea negativo si el nodo es un hijo izquierdo o positivo si es un hijo derecho. El apuntador al padre de un nodo está dado entonces por el valor absoluto del campo *father*. Las operaciones *isleft* o *isright* necesitarán entonces examinar solamente el signo del campo *father*.

Para implantar *brother(p)* de manera más eficaz, se puede incluir también un campo *brother* adicional en cada nodo.

Una vez que se declara el arreglo de nodos, es posible crear una lista disponible ejecutando las siguientes instrucciones:

```
int avail, i;
{
 avail = 1;
 for (i=0; i < NUMNODES; i++)
 node[i].left = i + 1;
 node[NUMNODES-1].left = 0;
```

Las funciones *getnode* y *freenode* son directas y se dejan como ejercicio. Adviértase que la lista disponible no es un árbol binario sino una lista lineal cuyos nodos están

ligados por medio del campo *left*. Cada nodo de un árbol se toma, cada vez que se requiere, del recipiente que contiene los disponibles y se devuelve a dicho recipiente cuando deja de usarse. Esta representación se llama *representación con arreglo ligada* de un árbol binario.

De manera alterna, un nodo puede definirse por:

```
struct nodetype {
 int info;
 struct nodetype *izquierdo
 struct nodetype *derecho
 struct nodetype *father;
};

typedef struct nodetype *NODEPTR;
```

Las operaciones *info(p)*, *left(p)*, *right(p)* y *father(p)* se implantarían mediante referencias a *p -> info*, *p -> left*, *p -> right* y *p -> father*, respectivamente. Con esta implantación, no se necesita una lista disponible explícita. Las rutinas *getnode* y *freenode* asignan y liberan nodos llamando a las rutinas *malloc* y *free*. Esta representación se llama *representación con nodos dinámica*, de un árbol binario.

La representación con arreglo ligada y la representación con nodos dinámica son implantaciones de una *representación ligada abstracta* (llamada también *representación con nodos*) en la que los apuntadores explícitos ligan los nodos de un árbol binario.

Ahora se presentarán implantaciones en C de las operaciones con árboles binarios bajo la representación con nodos dinámica y al lector se le deja como ejercicio las implantaciones con arreglo ligadas. La función *maketree*, que asigna un nodo y lo inicializa como la raíz de un árbol binario de un solo nodo, puede escribirse como sigue:

```
NODEPTR maketree(x)
int x;
{
 NODEPTR p;
 p = getnode();
 p->info = x;
 p->left = NULL;
 p->right = NULL;
 return(p);
} /* fin de maketree */
```

La rutina *setleft(p,x)* inicializa un nodo con contenidos *x* como el hijo izquierdo de *node(p)*:

```
setleft(p, x)
NODEPTR p;
int x;
{
```

```
if (p == NULL)
 printf("inserción no efectuada \n");
else if (p->left != NULL)
 printf("inserción inválida \n");
else
 p->left = maketree(x);
} /* fin de setleft */
```

La rutina *setright(p,x)* para crear un hijo derecho de *node(p)* con contenidos *x* es similar y se deja como ejercicio al lector.

No siempre es necesario usar los campos *father*, *left* y *right*. Si un árbol se recorre siempre hacia abajo (de la raíz a las hojas), la operación *father* no se usa nunca; en ese caso, un campo *father* no es necesario. Por ejemplo, los recorridos en preorden, en orden y postorden no usan el campo *father*. De manera análoga, si un árbol se recorre siempre hacia arriba (de las hojas a la raíz), no se necesitan los campos *left* y *right*. Las operaciones *isleft* e *isright* pueden implantarse incluso sin usar los campos *left* y *right* si se usa un apuntador con signo en el campo *father* bajo la implantación con arreglo ligada, como se analizó con anterioridad; un hijo derecho se indica con un valor positivo de *father* y uno izquierdo con uno negativo. Desde luego que entonces hay que modificar las rutinas *maketree*, *setleft* y *setright* de manera apropiada para esas representaciones. Bajo la representación con nodos dinámica, se requiere un campo lógico *isleft* además de *father* si se desea implantar las operaciones *isright* e *isleft* y si los campos *left* y *right* no están presentes.

El siguiente programa usa un árbol de búsqueda binaria para encontrar números duplicados en un archivo de entrada en el que cada número está en una línea de entrada por separado. El programa es casi igual que el algoritmo de la sección 5.1. Como sólo se usan ligas hacia abajo, no se necesita el campo *father*.

```
struct nodetype {
 int info;
 struct nodetype *left;
 struct nodetype *right;
};

typedef struct nodetype *NODEPTR;

main()
{
 NODEPTR ptree;
 NODEPTR p, q;
 int number;
 scanf("%d", &number);
 ptree = maketree(number);
 while (scanf("%d", &number) != EOF) {
 p = q = ptree;
 while (number != p->info && q != NULL) {
 p = q;
 q =
```

```

if (number < p->info)
 q = p->izquierdo
else
 q = p->derecho
} /* fin de while */
if (number == p->info)
 printf("%d está duplicado\n", número);
else if (number < p->info)
 setleft(p, number);
else
 setright(p, number);
} /* fin de while */
} /* fin de main */

```

### Nodos internos y externos

Por definición, los nodos hoja no tienen hijos. Así, en la representación ligada de árboles binarios sólo se necesitan los apuntadores *left* y *right* para nodos que no sean hojas. A veces se usan dos conjuntos separados para nodos hojas y nodos que no lo sean. Los nodos que no son hojas contienen campos *info*, *left* y *right* (con frecuencia no hay información asociada a los nodos que no son hojas, de manera que el campo *info* es innecesario) y se asignan como registros dinámicos o como un arreglo de registros que se maneja usando una lista disponible. Los nodos hoja no contienen campos *left* ni *right* y se guardan como un arreglo *info* simple que se asigna en orden secuencial cuando se necesita (se asume que las hojas no se liberan nunca, lo que ocurre con frecuencia). Como alternativa, esos nodos se pueden asignar como variables dinámicas que contengan sólo un valor *info*. Esto ahorra una gran cantidad de espacio, pues las hojas representan con frecuencia la mayoría de los nodos en un árbol binario. Cada nodo (hoja o no) puede contener también un campo *father* si es necesario.

Cuando se hace esta distinción entre nodos hojas y nodos no hojas, los últimos se llaman *nodos internos* y los primeros *nodos externos*. La terminología también se usa con frecuencia, aun cuando se define un solo tipo de nodo. Por supuesto, un apuntador hijo dentro de un nodo interno debe ser identificado como apuntador a un nodo interno o externo. Hay dos maneras de hacer esto en C. Una técnica es declarar dos tipos diferentes de nodos y apuntadores y usar una unión para nodos internos, con cada alternativa conteniendo uno de los dos tipos de apuntadores. La otra técnica es retener un solo tipo de apuntador y un solo tipo de nodo, donde el nodo es una unión que contiene (si es interno) o no (si es externo) campos apuntadores izquierdo y derecho. Al final de esta sección se verá un ejemplo de esta última técnica.

### Representación con arreglo implícito de árboles binarios

Hay que recordar de la sección 5.1, que los  $n$  nodos de un árbol binario quasi-completo pueden enumerarse desde 1 hasta  $n$ , de manera que el número asignado al

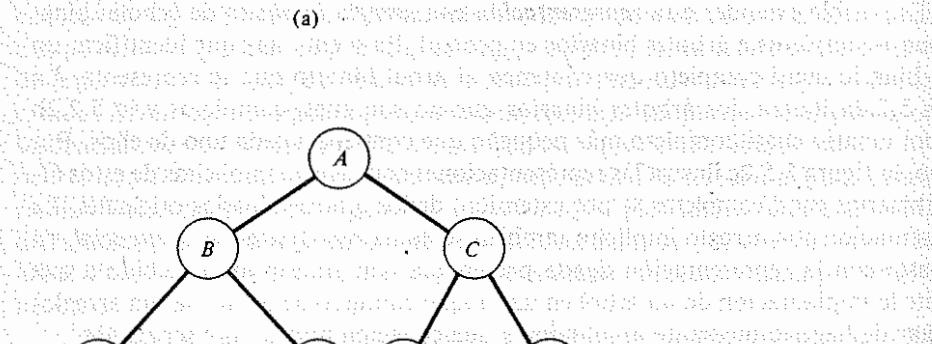
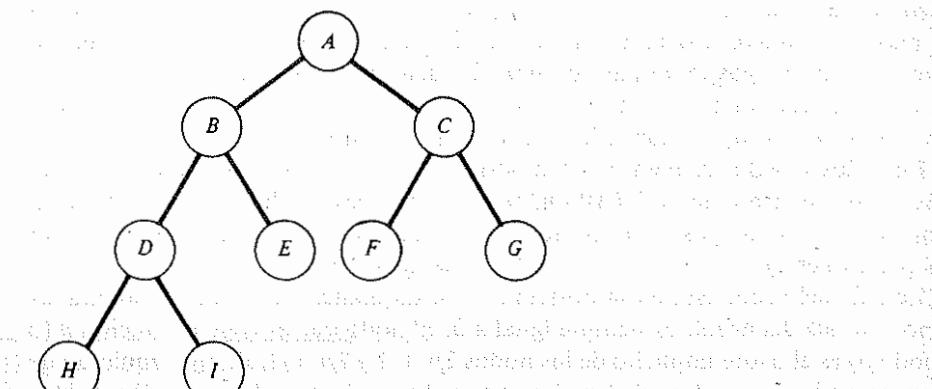
hijo izquierdo es dos veces el número asignado a su padre y el del hijo derecho es dos veces más 1 el número asignado a su padre. Se puede representar un árbol binario quasi-completo sin ligas *father*, *left* o *right*. En lugar de ello, los nodos pueden guardarse en un arreglo *info* de tamaño  $n$ . Aquí se alude al nodo de la posición  $p$  sólo como “*nodo p*”. *info[p]* guarda los contenidos de nodo  $p$ .

En C, los arreglos comienzan en la posición 0, por lo tanto, en lugar de enumerar los nodos del árbol desde 1 hasta  $n$ , se enumeran desde 0 hasta  $n - 1$ . A causa del corrimiento en una posición, los dos hijos de un nodo enumerado con  $p$  estarán en las posiciones  $2p + 1$  y  $2p + 2$ , en lugar de  $2p$  y  $2p + 1$ .

La raíz del árbol está en la posición 0, de tal manera que *tree*, el apuntador externo a la raíz del árbol, es siempre igual a 0. El nodo que está en la posición  $p$  (o sea, nodo  $p$ ) es el padre implícito de los nodos  $2p + 1$  y  $2p + 2$ . El hijo izquierdo de nodo  $p$  es nodo  $2p + 1$  y el derecho, nodo  $2p + 2$ . Así, la operación *left(p)* se implanta por medio de  $2 * p + 1$  y *right(p)* por medio de  $2 * p + 2$ . Dado un hijo izquierdo en la posición  $p$ , su hermano derecho está en la  $p + 1$ , y dado un hijo derecho en la posición  $p$ , su hermano izquierdo está en la  $p - 1$ . *father(p)* se implanta mediante  $(p - 1)/2$ .  $p$  apunta a un hijo izquierdo si y sólo si  $p$  es impar. Así, verificar si nodo  $p$  es un hijo izquierdo (la operación *isleft*) equivale a verificar que  $p \% 2$  no es igual a 0. La figura 5.2.1 muestra arreglos que representan los árboles binarios quasi-completos de las figuras 5.1.5c y d.

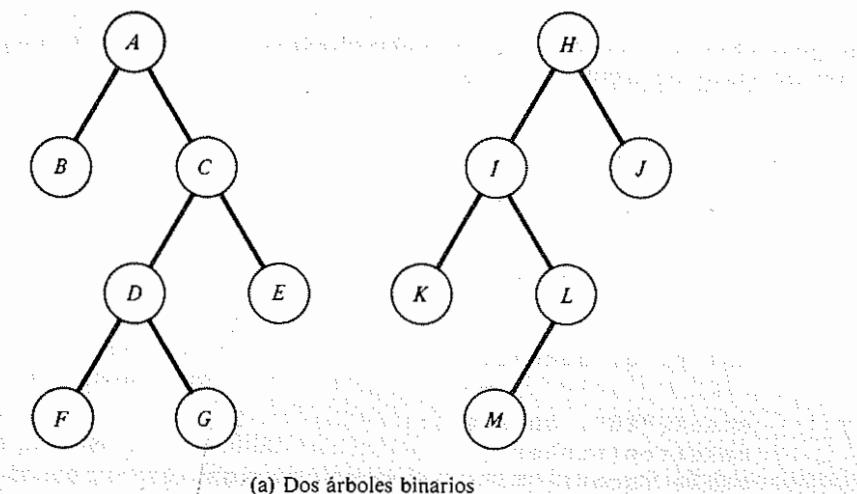
Es posible extender esta *representación con arreglo implícito* de árboles binarios quasi-completos a árboles binarios en general. Para ello, hay que identificar un árbol binario quasi-completo que contenga al árbol binario que se representa. La figura 5.2.2a ilustra dos árboles binarios que no son quasi-completos y la 5.2.2b el árbol binario quasi-completo más pequeño que contiene a cada uno de ellos. Por último, la figura 5.5.2c ilustra las representaciones con arreglo implícitas de estos árboles binarios quasi-completos y, por extensión, de los árboles binarios originales. La representación con arreglo implícito también se llama *representación secuencial*, en contraste con la representación ligada presentada con anterioridad, debido a que permite la implantación de un árbol en un bloque contiguo de memoria (un arreglo) en lugar de hacerlo mediante apuntadores que conecten nodos muy separados.

Con la representación secuencial, se asigna un elemento de un arreglo, ya sea que sirva o no para contener un nodo de un árbol. En consecuencia, hay que indicar los elementos del arreglo no usados como nodos del árbol *nulos* o que no existen. Esto se puede realizar mediante uno de dos métodos. Uno de ellos es asignar a *info[p]* un valor especial si nodo  $p$  es nulo. Este valor especial debería ser inválido como el contenido de información de un auténtico nodo del árbol. Por ejemplo, en un árbol que contenga números positivos, un nodo nulo puede indicarse mediante un valor negativo de *info*. Otra posibilidad es agregar a cada nodo un campo lógico indicador *used*. Cada nodo contiene entonces dos campos, *info* y *used*. Toda la estructura está contenida entonces en un nodo del arreglo. *used(p)*, implantada como *node[p].used*, es *TRUE* si el nodo  $p$  no es un nodo nulo y *FALSE* si lo es. *info(p)* se implanta mediante *node[p].info*. Este último método se usa para realizar la representación secuencial.

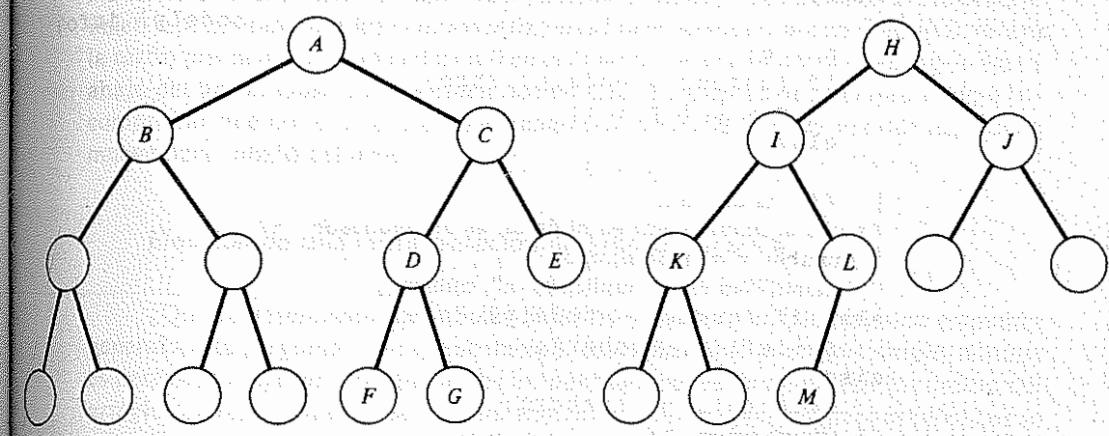


(b)

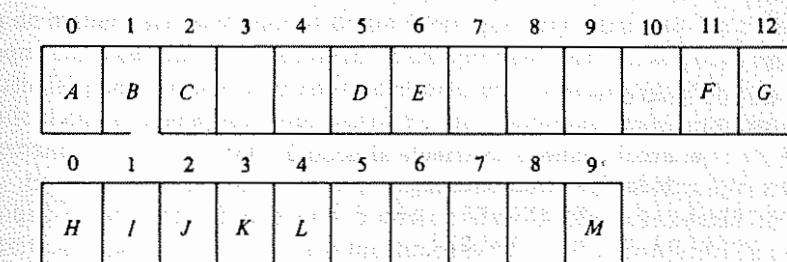
Figura 5.2.1



(a) Dos árboles binarios



(b) Extensiones cuasi-completas



(c) Representaciones por medio de arreglos

Figura 5.2.2

Ahora se presenta el programa para encontrar números duplicados en una lista de números de entrada, así como las rutinas *maketree* y *setleft*, usando la representación secuencial de árboles binarios.

```
#define NUMNODES 500
struct nodetype {
 int info;
 int used;
} node[NUMNODES];
main()
{
 int p, q, number;

 scanf("%d", &number);
 maketree(number);
 while (scanf("%d", &number) != EOF) {
 p = q = 0;
 while (q < NUMNODES && node[q].used && number != node[p].info) {
 p = q;
 if (number < node[p].info)
 q = 2 * p + 1;
 else
 q = 2 * p + 2;
 } /* fin de while */
 /* Si el número está en el árbol, es un duplicado */
 if (number == node[p].info)
 printf("%d es un duplicado\n", número);
 else if (number < node[p].info)
 setleft(p, number);
 else
 setright(p, number);
 } /* fin de while */
} /* fin de main */
maketree(x)
int x;
{
 int p;
 node[0].info = x;
 node[0].used = TRUE;
 /* el árbol contiene solamente al nodo 0 */
 /* todos los otros nodos están vacíos */
 for (p=1; p < NUMNODES; p++)
 node[p].used = FALSE;
} /* fin de maketree */

setleft(p, x)
int p, x;
{
```

```
 int q;
 q = 2 * p + 1; /* q es la posición del hijo izquierdo */
 if (q >= NUMNODES)
 error("desborde del arreglo");
 else if (node[q].used)
 error("inserción inválida");
 else {
 node[q].info = x;
 node[q].used = TRUE;
 } /* fin de if */
} /* fin de setleft */
```

La rutina para *setright* es similar.

Obsérvese que bajo esta implantación, la rutina *maketree* inicializa los campos *info* y *used* para representar un árbol con un solo nodo. Ya no es necesario que *maketree* dé como resultado un valor, ya que en esta representación el árbol binario único representado por los campos *info* y *used* tiene siempre raíz en el nodo 0. Esa es la razón por la que *p* se inicializa a 0 en la función principal antes de moverse hacia abajo del árbol. Obsérvese también que en esta representación, siempre se necesita comprobar que no se ha excedido el rango (*NUMNODES*) cada vez que alguien se mueve hacia abajo del árbol.

### Elección de una representación de árboles binarios

¿Qué representación de árboles binarios es preferible? No hay una respuesta general a esta pregunta. La representación secuencial es un poco más simple, aunque es necesario asegurarse de que todos los apuntadores están dentro de los límites del arreglo. La representación secuencial ahorra espacio en memoria para árboles que se sabe son quasi-completos, pues elimina la necesidad de los campos *left*, *right* y *father* e incluso no requiere del campo *used*. También es eficaz en cuanto a espacio para árboles que les faltan unos cuantos nodos para ser quasi-completos o cuando se eliminan de manera sucesiva, nodos de un árbol que originalmente fue quasi-completo, aunque entonces podría necesitarse un campo *used*. Sin embargo, la representación secuencial puede usarse sólo en un contexto en el que se requiera un solo árbol, o donde el número de árboles necesarios y cada uno de sus tamaños máximos se fija de antemano.

En contraste, la representación ligada requiere de campos *left*, *right* y *father* (aunque ya se ha visto que uno o dos de ellos pueden eliminarse en situaciones específicas), pero permite un uso mucho más flexible del conjunto de nodos. En la representación ligada, un nodo particular puede colocarse en cualquier localización y en cualquier árbol, mientras que en la representación secuencial un nodo se puede usar sólo si se necesita en una locación en un árbol específico. Además, en la representación con nodos dinámica, el número total de árboles y nodos está limitado exclusivamente por la cantidad de memoria disponible. Así, la representación ligada

es preferible en la situación dinámica general de muchos árboles de forma impredecible.

El programa para encontrar duplicados es una buena explicación de los compromisos implicados. El primer programa presentado utiliza la representación ligada de árboles binarios. Requiere de campos *left* y *right*, además de *info* (el campo *father* no se necesitó en ese programa). El segundo programa para encontrar duplicados, que utiliza la representación secuencial sólo requiere de un campo adicional, *used* (y éste también puede eliminarse si sólo se permiten números positivos en la entrada, de tal manera que un nodo nulo de un árbol pueda representarse mediante un valor *info* negativo específico). La representación secuencial puede usarse en este ejemplo, porque se requiere un solo árbol.

Sin embargo, el segundo programa podría no funcionar para tantos casos de entrada como el primero. Por ejemplo, supóngase que la entrada está en orden ascendente. Entonces el árbol formado por cualquiera de los programas tiene todos los subárboles izquierdos nulos (se invita al lector a verificar que ése sea el caso mediante la simulación del programa para tal entrada). En ese caso, los únicos elementos de *info* que están ocupados en la representación secuencial son 0, 2, 6, 14 y así sucesivamente (cada posición es el doble más dos de la anterior). Si el valor de *NUMNODES* se mantiene en 500, se pueden acomodar como máximo 16 números ascendentes distintos (el último estará en la posición 254). Esto puede contrastarse con el programa que usa la representación ligada, en el cual se pueden acomodar hasta 500 números distintos en orden ascendente antes de exceder el espacio. En el resto del texto, a menos que se indique otra cosa, se asume el uso de la representación ligada de un árbol binario.

### Recorridos de árboles binarios en C

Es posible implantar el recorrido de árboles binarios en C por medio de rutinas recursivas que reflejen las definiciones de recorrido. Las tres rutinas en C: *pretrav*, *intrav* y *posttrav*, imprimen los contenidos de un árbol binario en preorden, orden y postorden, respectivamente. El parámetro de cada rutina es un apuntador al nodo raíz de un árbol binario. Se usa la representación dinámica de los nodos para un árbol binario:

```
pretrav(tree)
NODEPTR tree;
{
 if (tree != NULL) {
 printf("%d\n", tree->info); /* visitando la raíz */
 pretrav(tree->left); /* recorriendo el subárbol izquierdo */
 pretrav(tree->right); /* recorriendo el subárbol derecho */
 } /* fin de if */
} /* fin de pretrav */

intrav(tree)
NODEPTR tree;
{

```

```
if (tree != NULL) {
 intrav(tree->left); /* recorriendo el subárbol izquierdo */
 printf("%d\n", tree->info); /* visitando la raíz */
 intrav(tree->right); /* recorriendo el subárbol derecho */
} /* fin de if */
} /* fin de intrav */

posttrav(tree)
NODEPTR tree;
{
 if (tree != NULL) {
 posttrav(tree->left); /* recorriendo el subárbol izquierdo */
 posttrav(tree->right); /* recorriendo el subárbol derecho */
 printf("%d\n", tree->info); /* visitando la raíz */
 } /* fin de if */
} /* fin de posttrav */

```

Se invita al lector a simular la acción de estas rutinas sobre los árboles de las figuras 5.1.7 y 5.1.8.

Por supuesto, las rutinas pueden escribirse de manera no recursiva para ejecutar la inserción y la eliminación necesarias de manera explícita. Por ejemplo, la siguiente es una rutina no recursiva para recorrer un árbol binario en orden:

```
#define MAXSTACK 100

intrav2(tree)
NODEPTR tree;
{
 struct stack {
 int top;
 NODEPTR item[MAXSTACK];
 } s;
 NODEPTR p;

 s.top = -1;
 p = tree;
 do {
 /* descendiendo por las ramas izquierdas tanto como sea posible,
 * y guardando los apuntadores a los nodos en el camino */
 while (p != NULL) {
 push (s, p);
 p = p->left;
 } /* fin de while */
 /* verificar si se terminó */
 if (!empty(s)) {
 /* en este punto el subárbol izquierdo está vacío */
 p = pop(s);
 printf("%d\n", p->info); /* visitando la raíz */
 }
 } while (s.top > -1);
}
```

```

 p = p->right; /* recorriendo el subárbol derecho */
} /* fin de if */
} while (!empty(s) && p != NULL);
} /* fin de intrav2 */

```

Se deja como ejercicio al lector la escritura de rutinas para recorrer un árbol binario en postorden y preorden, así como recorridos no recursivos, usando la representación secuencial.

*intrav* e *intrav2* representan un excelente contraste entre una rutina recursiva y su opuesta, la no recursiva. Si se ejecutan ambas, la recursiva *intrav* se ejecuta en general con mucha más rapidez que la no recursiva *intrav2*. Esto se opone a la aceptada “sabiduría popular” de que la recursión es más lenta que la iteración. La causa primaria de la ineficacia de *intrav2*, tal como está escrita, son las llamadas a *push*, *pop* y *empty*. Aun cuando el código para esas funciones esté insertado en línea dentro de *intrav2*, ésta será más lenta que *intrav* a causa de las verificaciones de desbordamiento y desbordamiento negativo, a menudo innecesarias, que se incluyen en ese código.

Sin embargo, aun cuando se eliminan las verificaciones mencionadas, ¡*intrav* es más rápida que *intrav2* si se usa un compilador que ejecute el proceso recursivo con eficacia! La eficacia de la recursividad está dada en este caso por un número de factores:

- No hay recursividad “extra” como sucede al calcular los números de Fibonacci, donde se vuelven a calcular por separado  $f(n - 2)$  y  $f(n - 1)$ , aunque el valor de  $f(n - 2)$  se utiliza para encontrar el de  $f(n - 1)$ .
- La pila de recursión no puede eliminarse por completo como ocurre cuando se calcula la función factorial. Así, el apilamiento y desapilamiento de la recursión interconstruida es más eficaz que la versión programada. (En muchos sistemas, el apilamiento puede realizarse mediante el incremento del valor de un registro que apunte al tope de la pila y moviendo todos los parámetros dentro de una nueva área de datos con un simple movimiento de bloques. El programa controlado de apilamiento, tal y como se implantó, requiere de asignaciones e incrementos individuales.)
- No hay parámetros o variables locales extraños, como los hay, por ejemplo, en algunas versiones de la búsqueda binaria. El manejo automático de la pila para recursión no inserta en la pila más variables que las necesarias.

En los casos en que la recursión no implica tal sobrecarga, como en un recorrido en orden se aconseja al programador usar la recursión de manera directa.

Las rutinas para recorridos presentadas se derivan directamente de las definiciones de los métodos de recorrido. Esas definiciones están planteadas en términos de los hijos izquierdo y derecho de un nodo y no hacen referencia al padre. Por tal razón, ambas rutinas, la recursiva y la no recursiva, no requieren de un campo *father* y no lo aprovechan aun cuando está presente. Como se verá más adelante, la presencia de un campo *father* permite desarrollar algoritmos de recorridos no recursivos sin usar una pila. Sin embargo, primero se examina una técnica para la elimi-

nación de la pila en un recorrido no recursivo, aun cuando no esté disponible un campo *father*.

### Arboles binarios hebrados

Recorrer un árbol binario es una operación común, y sería útil encontrar un método más eficaz para implantar el recorrido. Examíñese la función *intrav2* para descubrir la razón por la cual se necesita una pila. La pila se vacía cuando *p* es igual a *NULL*. Esto ocurre en uno de dos casos. En uno, se abandona el ciclo *while* después de haberlo ejecutado una o más veces. Esto implica que el programa ha recorrido hacia abajo ramas izquierdas hasta encontrar un apuntador *NULL*, poniendo un apuntador a cada nodo cuando los pasa. Así, el elemento tope de la pila es el valor de *p* antes de que éste se convierta en *NULL*. Si se guarda un apuntador auxiliar *q* un paso atrás de *p*, el valor de *q* puede usarse de manera directa sin necesidad de extraerlo de la pila.

El otro caso en el que *p* es *NULL* es aquél en el que el ciclo *while* se salta por completo. Esto ocurre después de alcanzar un nodo con un subárbol derecho vacío, cuando se ejecuta la instrucción *p = p->right*, y se regresa para repetir el cuerpo del ciclo *do while*. En este punto, ya se habría perdido el camino si no fuese por la pila cuyo tope apunta al nodo cuyo subárbol izquierdo se acaba de recorrer. Sin embargo, supóngase que un nodo con un subárbol derecho vacío contiene en su campo *right* un apuntador al nodo que estaría en el tope de la pila de ese punto del algoritmo (esto es, un apuntador a su sucesor en orden) en lugar de contener un apuntador *NULL*. Entonces ya no se necesitaría la pila, pues el último nodo visitado durante el recorrido de un subárbol izquierdo apunta directamente a su sucesor en orden. Un apuntador de este tipo se conoce como *hebra* y debe diferenciarse de un apuntador en el árbol que se usa para ligar un nodo a su subárbol derecho o izquierdo.

La figura 5.2.3 muestra los árboles binarios de la figura 5.1.7 con hebras que remplaza a apuntadores nulos *NULL* en los nodos con subárboles derechos vacíos. Las hebras están dibujadas con línea punteada para diferenciarlas de los apuntadores al árbol. Adviértase que el nodo de la extrema derecha de cada árbol tiene aún un apuntador derecho *NULL*, ya que no tiene sucesor de en orden. Los árboles de este tipo se llaman árboles binarios *enhebrados a la derecha*.

Para implantar un árbol binario enhebrado a la derecha mediante la implantación dinámica con nodos de un árbol binario, se incluye en cada nodo un campo lógico extra, *rthread*, para indicar si su apuntador derecho es o no es una hebra. Para ser consistentes, también se asigna *TRUE* al campo *rthread* del nodo del árbol a la extrema derecha (esto es, el último nodo en el recorrido en orden del árbol), aunque su campo *right* permanece como *NULL*. En consecuencia, un nodo se define como sigue (hay que recordar que se está suponiendo que no existe ningún campo *father*):

```

struct nodetype {
 int info;
 struct nodetype *left;
 struct nodetype *right;
 /* apuntador al hijo izquierdo */
 /* apuntador al hijo derecho */
};

```

```

 int rthread;
 }

 typedef struct nodetype *NODEPTR;

```

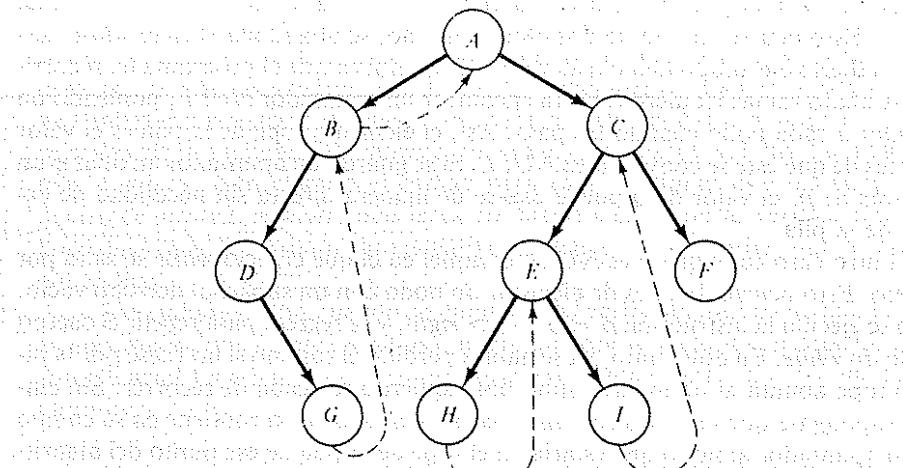


Figura 5.2.3 Árboles binarios enhebrados a la derecha.

Ahora se presenta una rutina para implantar el recorrido en orden de un árbol binario enhebrado a la derecha.

```

intrav3(tree)
NODEPTR tree;
{
 NODEPTR p, q;
 p = tree;
 do {
 q = NULL;
 while (p != NULL) { /* recorrido de la rama izquierda */
 q = p;
 p = p->left;
 } /* fin de while */
 if (q != NULL) {
 printf("%d\n", q->info);
 p = q->right;
 while (q->rthread && p != NULL) {
 printf("%d\n", p->info);
 q = p;
 p = p->right;
 } /* fin de while */
 } /* fin de if */
 } while (q != NULL)
} /* fin de intrav3 */

```

En un árbol binario enhebrado a la derecha, el sucesor de en orden de cualquier nodo puede encontrarse con eficacia. Dicho árbol también puede construirse de manera simple. A continuación se presentan las rutinas *maketree*, *setleft* y *setright*. Supóngase que cada nodo tiene campos *info*, *left*, *right* y *rthread*.

```

NODEPTR maketree(x)
int x;
{
 NODEPTR p;
 p = getnode();
 p->info = x;
 p->left = NULL;
 p->right = NULL;
 p->rthread = TRUE;
 return(p);
} /* fin de maketree */

setleft(p, x)
NODEPTR p;
int x;
{

```

```

NODEPTR q;

if (p == NULL)
 error("inserción no efectuada");
else if (p->left != NULL)
 error("inserción no válida");

else {
 q = getnode();
 q->info = x;
 p->left = q;
 q->left = NULL;
 /* El sucesor en orden de node (q) es node (p) */
 q->right = p;
 q->rthread = TRUE;
} /* fin de if */
} /* fin de setleft */

setright(p, x);

NODEPTR p;
int x;
{
 NODEPTR q, r;

 if (p == NULL)
 error("inserción no efectuada");
 else if (!p->rthread)
 error("inserción inválida");
 else {
 q = getnode();
 q->info = x;
 /* Guardar el sucesor en orden de node (p) */
 r = p->right;
 p->right = q;
 p->rthread = FALSE;
 q->left = NULL;
 /* El sucesor en orden de node (q) es el sucesor previo de */
 /* node (p) */
 q->right = r;
 q->rthread = TRUE;
 } /* fin de else */
} /* fin de setright */

```

En la implantación con arreglo ligada, una hebra puede representarse mediante un valor negativo de `node[p].right`. El valor absoluto de `node[p].right` es el índice de sucesor en orden de `node[p]` en el arreglo `node`. El signo de `node[p].right` indica si su valor absoluto representa una hebra (*menos*) o un apuntador a un subárbol no vacío (*más*). Bajo esta implantación, la rutina siguiente recorre un árbol binario enhebrado a la derecha en orden. A manera de ejercicio se dejan las rutinas *makeTree*, *setleft* y *setright* para las representaciones con arreglo ligadas.

```

intrav4(tree)
int tree;
{
 int p, q;
 p = tree;
 do {
 /* Descender por el árbol manteniendo el eslabón q detrás de p */
 q = 0;
 while (p != 0) {
 q = p;
 p = node[p].left;
 } /* fin de while */
 if (q != 0) { /* Verificar si se terminó */
 printf("%d\n", node[q].info);
 p = node[q].right;
 while (p < 0) {
 q = -p;
 printf("%d\n", node[q].info);
 p = node[q].right;
 } /* fin de while */
 } /* fin de if */
 /* Recorrer el subárbol derecho */
 } while (q != 0);
} /* fin de intrav4 */

```

En la representación secuencial de árboles binarios, el campo *used* indica las hebras por medio de valores positivos o negativos. Si *i* representa un nodo con un hijo derecho, `node[i].used` es igual a 1, y su hijo derecho está en  $2 * i + 2$ . Sin embargo, si *i* representa un nodo sin hijo derecho, `node[i].used` contiene el número negativo con valor absoluto igual al índice de su sucesor en orden. (Obsérvese que el uso de números negativos permite distinguir un nodo con un hijo derecho de un nodo cuyo sucesor en orden sea la raíz del árbol.) Si *i* es el nodo a la extrema derecha del árbol, de tal manera que no tiene sucesor en orden, `node[i].used` puede contener el valor especial + 2. Si *i* no representa un nodo, `node[i].used` es 0. Se deja como ejercicio al lector la implantación de algoritmos de recorrido para esta representación.

Un árbol binario *enhebrado a la izquierda* puede definirse de manera similar coino el árbol en el que cada apuntador izquierdo *NULL* se altera para contener una hebra hacia el nodo antecesor en orden. Un árbol binario *enhebrado* se puede definir entonces como el árbol binario que está tanto a la derecha como a la izquierda. Sin embargo, los árboles enhebrados a la izquierda no tienen las ventajas que los enhebrados a la derecha.

También pueden definirse los árboles binarios *prehebrados* a la derecha y a la izquierda, en los que los apuntadores derechos e izquierdos de nodos con valor *NULL* son sustituidos respectivamente por los sucesores y antecesores de preorden del nodo correspondiente. Un árbol binario prehebrado a la derecha puede ser recorrido de manera eficaz en preorden sin usar una pila. Un árbol binario enhebrado a la derecha también puede recorrer en preorden sin usar una pila. Los algoritmos de recorrido se dejan como ejercicio para el lector.

## Recorrido por medio de un campo *father*

Cuando cada árbol de nodos contiene un campo *father*, no se necesitan ni hebras ni pila para el recorrido no recursivo. En lugar de ello, cuando el proceso de recorrido alcanza un nodo hoja se puede usar el campo *father* para ascender por atrás al árbol. Cuando se alcanza *node(p)* desde un hijo izquierdo, su subárbol derecho aún debe recorrerse; por lo tanto, el algoritmo procede hacia *right(p)*. Cuando se alcanza *node(p)* desde su hijo derecho, ya se han recorrido sus dos subárboles y el algoritmo retrocede hacia *father(p)*. La siguiente rutina implanta este proceso para el recorrido en orden.

```

intrav5(tree)
NODEPTR tree;
{
 NODEPTR p, q;
 q = NULL; p = tree;
 do {
 while (p != NULL) {
 q = p;
 p = p->left;
 } /* fin de while */
 if (q != NULL) {
 printf("%d\n", q->info);
 p = q->right;
 } /* fin de if */
 while (q != NULL && p == NULL) {
 do {
 /* node (q) no tiene hijo derecho. Regresar hasta que
 se encuentre un hijo izquierdo o la raíz del árbol */
 p = q;
 q = p->father;
 } while (!isleft(p) && q != NULL);
 if (q != NULL) {
 printf("%d\n", q->info);
 p = q->right;
 } /* fin de if */
 } /* fin de while */
 } while (q != NULL);
} /* fin de intrav5 */

```

Obsérvese que se escribió *isleft(p)* en lugar de *p -> isleft* porque no es necesario un campo *isleft* para determinar si *node(p)* es un hijo izquierdo o derecho; se puede verificar simplemente si el nodo es el hijo izquierdo del padre.

En este recorrido en orden se visita un nodo [*printf("%d\n", q -> info)*] cuando se reconoce al hijo izquierdo como *NULL* o cuando se alcanza éste después de regresar del hijo izquierdo. Los recorridos en preorden y postorden son similares

excepto en que, en preorden, se visita un nodo sólo cuando se alcanza descendiendo del árbol y en postorden, cuando el hijo derecho es *NULL* o cuando se alcanza después de regresar del hijo derecho. Los detalles se quedan como ejercicio para el lector.

El recorrido que usan los apuntadores *father* al regresar es menos eficaz en cuanto a tiempo que el recorrido de un árbol enhebrado. Una hebra apunta en forma directa al sucesor de un nodo, mientras que en un árbol no enhebrado quizás tenga que seguirse una serie de apuntadores *father* para alcanzar ese sucesor. No es fácil comparar la eficacia concerniente al tiempo del recorrido basado en la pila con el del recorrido basado en el padre, pues el primero incluye la sobrecarga de apilamiento y desapilamiento.

Este algoritmo de regreso sugiere también una técnica de recorrido no recursivo sin pila para árboles no enhebrados, aun cuando no existe el campo *father*. La técnica es simple: sólo hay que invertir el apuntador *son* cuando se desciende del árbol para que pueda usarse para encontrar un camino de regreso para ascender. En este camino de regreso, se restaura al apuntador su valor original.

Por ejemplo, en *intrav5*, se puede introducir una variable *f* para guardar un apuntador al padre de *node(q)*. Las instrucciones:

```

q = p;
p = p->left;

```

en el primer ciclo *while* pueden remplazarse por

```

f = q;
q = p;
p = p->left;
if (p != NULL)
 q->left = f;

```

Esto modifica el apuntador izquierdo de *node(q)* para que apunte al padre de *node(q)* cuando se avanza a la izquierda en el camino de descenso [advírtase que *p* apunta al hijo izquierdo de *node(q)*, de tal manera que no se pierde el camino]. La instrucción,

```

p = q->right;

```

que aparece dos veces, puede remplazarse por

```

p = q->right;
if (p != NULL)
 q->right = f;

```

para modificar de manera similar el apuntador derecho de *node(q)* de manera que apunte a su padre cuando se avanza a la derecha en el camino de descenso. Por último, las instrucciones

```

p = q;
q = p->father;

```

en el ciclo interno *do-while* pueden remplazarse por

```
p = q;
q = f;
if (q != NULL && isleft(p)) {
 f = left(q);
 left(q) = p;
}
else {
 f = right(q);
 right(q) = p;
} /* fin de if */
```

para seguir un apuntador modificado que asciende por el árbol y restituye el valor del mismo para que apunte a su hijo derecho o izquierdo según se necesite.

Sin embargo, ahora se requiere de un campo *isleft*, ya que la operación *isleft* no puede implantarse mediante un campo *father* que no existe. Este algoritmo tampoco puede usarse en un ambiente de usuarios múltiples si varios usuarios necesitan accesar al árbol simultáneamente. Cuando un usuario recorre el árbol y modifica temporalmente los apuntadores, otro usuario no podrá usarlo como una estructura coherente. Se necesita algún tipo de mecanismo de cierre para asegurar que nadie más use el árbol mientras se invierten los apuntadores.

### Arboles binarios heterogéneos

Con frecuencia, la información contenida en los diferentes nodos de un árbol binario no es del mismo tipo. Por ejemplo, en la representación de una expresión binaria con operandos numéricos constantes, tal vez se desee usar un árbol binario cuyas hojas contengan números pero cuyos nodos que no son hojas contengan caracteres que representen operadores. La figura 5.2.4 ilustra un árbol binario de este tipo. Para representar dicho árbol binario en C, se puede usar una unión que represente la porción de información del nodo. Por supuesto, cada nodo del árbol debe contener en sí un campo que indique el tipo de objeto que contiene su campo *info*.

```
#define OPERATOR 0
#define OPERAND 1
struct nodetype {
 short int utype; /* OPERATOR or OPERAND */
 union {
 char chinfo;
 float numinfo;
 } info;
 struct nodetype *left;
 struct nodetype *right;
};
typedef struct nodetype *NODEPTR;
```

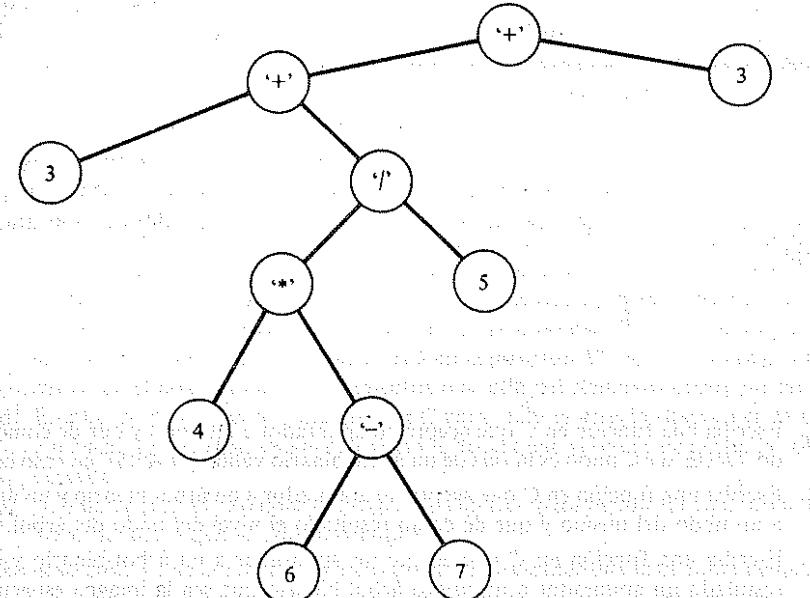


Figura 5.2.4 Árbol binario que representa a  $3 + 4 * (6 - 7)/5 + 3$ .

Escríbese ahora una función en C, de nombre *evalbintree*, que acepte un apuntador a un árbol de este tipo y dé como resultado el valor de la expresión representada por el árbol. La función evalúa de manera recursiva los subárboles izquierdo y derecho y luego aplica el operador de la raíz a los dos resultados. Aquí se usa la función auxiliar *oper(symb, opnd1, opnd2)* presentada en la sección 2.3. El primer parámetro de *oper* es un carácter que representa un operador y los dos últimos parámetros son números reales que son los operandos. La función *oper* regresa el resultado de aplicar el operador a los dos operandos.

```
float evalbintree (tree)
NODEPTR tree;
{
 float opnd1, opnd2;
 char symb;

 if (tree->utype == OPERAND) /* la expresión es un operando único */
 return (tree->numinfo);
 /* tree->utype == OPERATOR */
 /* evaluar el subárbol izquierdo */
 opnd1 = evalbintree(tree->left);
 /* evaluar el subárbol derecho */
 opnd2 = evalbintree(tree->right);
 return (oper(symb, opnd1, opnd2));
}
```

```

symbol = tree->chinfo; /* extraer el operador */
/* aplicarlo y regresar el resultado */
evalbintree = oper(symbol, opnd1, opnd2);
} /* fin de evalbintree */

```

En la sección 9.1 se analizan métodos adicionales de implantación de estructuras ligadas que contienen elementos heterogéneos. Obsérvese también que, en este ejemplo, todos los nodos operando son hojas y todos los nodos operadores son no-hojas.

## EJERCICIOS

- 5.2.1. Escriba una función en C que acepte un apuntador a un nodo y que dé como resultado *TRUE* si ese nodo es la raíz de un árbol binario válido y *FALSE* en caso contrario.
- 5.2.2. Escriba una función en C que acepte un apuntador a un árbol binario y un apuntador a un nodo del mismo y que dé como resultado el nivel del nodo del árbol.
- 5.2.3. Escriba una función en C que acepte un apuntador a un árbol binario y dé como resultado un apuntador a un nuevo árbol binario que sea la imagen especular (esto es, todos los subárboles derechos son ahora subárboles izquierdos y viceversa) del primero.
- 5.2.4. Escriba funciones en C que conviertan un árbol binario implantado mediante la representación ligada con arreglo con sólo un campo *father* (en el que el hijo izquierdo del campo *father* contiene el valor negativo del apuntador a su padre y el hijo derecho de dicho padre contiene un apuntador a su padre) a su representación que usa campos *left* y *right* y viceversa.
- 5.2.5. Escriba un programa en C para realizar el siguiente experimento: la generación de 100 números aleatorios. Cada vez que se genere un número, insértelo en un árbol de búsqueda binaria inicialmente vacío. Una vez que se inserten los 100 números, imprima el nivel de la hoja con nivel máximo y el de la hoja con nivel mínimo. Repita este proceso 50 veces. Imprima una tabla con un conteo acerca de cuántas de las 50 veces hubo diferencia entre el nivel de hoja máxima y mínima de 0, 1, 2, 3, etcétera.
- 5.2.6. Escriba rutinas en C para recorrer un árbol binario en postorden y preorden.
- 5.2.7. Implante el recorrido en orden, *maketree*, *setleft* y *setright* para árboles binarios enhebrados a la derecha bajo la representación secuencial.
- 5.2.8. Escriba funciones en C para crear un árbol binario determinado a:
  - a. los recorridos en preorden y en orden de ese árbol
  - b. los recorridos en preorden y postorden de ese árbol
 Cada función debe aceptar dos cadenas de caracteres como parámetros. El árbol creado debe contener un solo carácter en cada nodo.
- 5.2.9. La solución al problema de las Torres de Hanoi, para  $n$  discos (ver secciones 3.3 y 3.4), puede representarse mediante un árbol binario completo de nivel  $n - 1$  como sigue:
  - a. Sea que la raíz del árbol represente un movimiento del disco tope del poste *frompeg* al poste *topeg*. (Se desconoce la identificación de los discos que se están moviendo, puesto que hay un solo disco que se puede mover [el del tope] de un poste a otro.) Si *nd* es un nodo hoja (en un nivel menor que  $n - 1$ ) que representa el

movimiento del disco del tope del poste *x* al poste *y*, sea *z* el tercer poste que no es ni el origen ni el destino del nodo *nd*. Entonces *left(nd)* representa un movimiento del disco tope del poste *x* al poste *z* y *right(nd)* uno del poste *z* al poste *y*. Dibujar árboles de solución muestra cómo se describió previamente para  $n = 1, 2, 3$  y  $4$  y mostrar que un recorrido en orden de tal árbol produce la solución al problema de las Torres de Hanoi.

- b. Escriba un procedimiento recursivo en C que acepte un valor  $n$  y genere y recorra el árbol como se discutió previamente.
  - c. Debido a que el árbol está completo, se puede guardar en un arreglo de tamaño  $2^n - 1$ . Muestre que los nodos del árbol se pueden almacenar en el arreglo de tal manera que un recorrido secuencial del mismo produzca el recorrido en orden del árbol como sigue: la raíz del árbol está en la posición  $2^{n-1} - 1$ ; para cualquier nivel *j*, el primer nodo en ese nivel está en la posición  $2^{n-1-j} - 1$  y cada nodo sucesivo en el nivel *j* está  $2^{n-j}$  elementos más allá del elemento previo de ese nivel.
  - d. Escriba un programa no recursivo en C para crear el arreglo descrito en la parte c y mostrar que el paso secuencial a través del arreglo produce de hecho la solución deseada.
  - e. ¿Cómo se podría extender el programa anterior para incluir en cada nodo el número del disco que se está moviendo?
- 5.2.10. En la sección 4.5 se introdujo un método para representar una lista doblemente ligada con un solo campo apuntador en cada nodo manteniendo su valor como la “*o*” exclusiva de los apuntadores al antecesor y sucesor del nodo. Algo similar puede hacerse con un árbol binario si se mantiene en cada nodo un campo como la “*o*” exclusiva de apuntadores al nodo *father* e hijo *izquierdo* [llámese a este campo *fleft(p)*] y otro campo en el nodo como la “*o*” exclusiva de los apuntadores a los nodos *father* e hijo *derecho* [llámese a este campo *fright(p)*].
- a. Dados *father(p)* y *fleft(p)*, muestre cómo calcular *left(p)*.  
Dados *father(p)* y *fright(p)*, muestre cómo calcular *right(p)*.
  - b. Dados *fleft(p)* y *left(p)*, muestre cómo calcular *father(p)*.  
Dados *fright(p)* y *right(p)*, muestre cómo calcular *father(p)*.
  - c. Supóngase que un nodo contiene sólo campos *info*, *fleft*, *fright* e *isleft*. Escriba algoritmos para los recorridos en preorden, en orden y postorden de un árbol binario, dado un apuntador externo a la raíz del árbol sin usar una pila y sin modificar ningún campo.
  - d. ¿Puede eliminarse el campo *isleft*?
- 5.2.11. El índice de un libro de texto consta de términos principales colocados en orden alfabético. Cada uno de ellos va acompañado por un conjunto de números de página y un conjunto de subtérminos. Los subtérminos están impresos en líneas sucesivas que siguen al término principal y dispuestos en orden alfabético dentro del término principal. Cada subtérmino va acompañado de un conjunto de números de página. Diseñe una estructura de datos para representar un índice como el descrito y escriba un programa en C para imprimir un índice de datos como sigue. Cada línea de entrada comienza con una *m* (término principal) o una *s* (subtérmino). Una línea *m* contiene una *m* seguida de un término principal, seguido por un entero *n* (0 es posible), seguido por *n* números de página en que aparece el término principal. Una línea *s* es similar excepto en que contiene un subtérmino en lugar de un término principal. Las líneas de entrada no aparecen en orden particular, excepto por el hecho de que cada subtérmino se considera un subtérmino del último término principal que lo precede. Puede haber muchas líneas de entrada para un solo término principal o subtérmino (todos los números de página que aparecen en cualquier línea para un término particular deben imprimirse con el mismo).

El índice debe imprimirse en una línea con un término seguido por todas las páginas en las que aparece dicho término en orden ascendente. Los términos principales deben imprimirse en orden alfabético. Los subtérminos, que también deben aparecer en orden alfabético, van inmediatamente después de su término principal. Los subtérminos deben colocarse a cinco columnas del término principal.

El conjunto de términos principales debe organizarse como un árbol binario. Cada nodo del árbol contiene (además de los apuntadores izquierdo y derecho y el término principal) apuntadores a otros dos árboles binarios. Uno de éstos representa el conjunto de números de página en las que aparece el término principal; el otro representa el conjunto de subtérminos del mismo. Cada nodo de un árbol binario que representa un subtérmino contiene (además de los apuntadores izquierdo y derecho y el propio subtérmino) un apuntador a un árbol binario que representa el conjunto de números de páginas en las que aparece el subtérmino.

- 5.2.12.** Escriba una función en C para implantar un método de ordenamiento como el de la sección 5.1 que use un árbol de búsqueda binaria.
- 5.2.13.** a. Implete una cola de prioridad ascendente usando un árbol de búsqueda binaria por medio de implantaciones en C de los algoritmos *pqinsert* y *pqmindelete*, igual que en el ejercicio 5.1.13. Modificar las rutinas para contar el número de nodos del árbol accesados.  
 b. Use un generador de números aleatorios para verificar la eficacia de la implantación de una cola de prioridad como sigue. Primero, cree una cola de prioridad con 100 elementos mediante la inserción de 100 números aleatorios en un árbol de búsqueda binaria inicialmente vacío. Después, llame a *pqmindelete* e imprima el número de nodos del árbol accesados para encontrar el elemento mínimo, genere un nuevo número aleatorio y llame a *pqinsert* para insertar el nuevo número aleatorio e imprima el número de nodos del árbol accesados en la inserción. Adviértase que después de llamar a *pqinsert*, el árbol aún contiene 100 elementos. Repita el proceso de eliminar/imprimir/generar/insertar/imprimir mil veces. Advertir que el número de nodos accesados en la eliminación tiende a disminuir, mientras que el número de nodos accesados en la inserción tiende a crecer. Explique dicho comportamiento.

### 5.3. UN EJEMPLO: EL ALGORITMO DE HUFFMAN

Supóngase que se tiene un alfabeto de  $n$  símbolos y un largo mensaje compuesto con símbolos del mismo. Se desea codificar el mensaje como una larga cadena de bits (un bit es 0 o 1) mediante la asignación de una cadena de bits como código para cada símbolo del alfabeto y la concatenación de los códigos individuales de los símbolos que conforman el mensaje para producir la codificación del mismo. Por ejemplo, supóngase que el alfabeto consta de los cuatro símbolos *A*, *B*, *C* y *D* y que se asignan códigos a estos símbolos como sigue:

| Símbolo  | Código |
|----------|--------|
| <i>A</i> | 010    |
| <i>B</i> | 100    |
| <i>C</i> | 000    |
| <i>D</i> | 111    |

El mensaje *ABACCCA* se codificaría entonces como 01010001000000111010. Una codificación de este tipo es ineficaz, debido a que se usan tres bits para cada símbolo, por lo que se necesitan 21 bits para todo el mensaje. Supóngase que se asigna un código de dos bits a cada símbolo, de la siguiente manera:

| Símbolo  | Código |
|----------|--------|
| <i>A</i> | 00     |
| <i>B</i> | 01     |
| <i>C</i> | 10     |
| <i>D</i> | 11     |

Entonces el código para el mensaje sería 00010010101100, que requiere únicamente 14 bits. Por tanto, se desea encontrar un código que reduzca al mínimo la longitud del mensaje codificado.

Examine otra vez el ejemplo anterior. Cada una de las letras *B* y *D* aparecen una sola vez en el mensaje, mientras que la letra *A* aparece tres veces. Si se escoge un código de tal manera que se le asigne a la *A* una cadena de bits más corta que las letras *B* y *D*, la longitud del mensaje codificado será más pequeña. Esto ocurre porque el código corto (que representaría a la letra *A*) aparece con más frecuencia que el largo. De hecho, los códigos pueden asignarse como sigue:

| Símbolo  | Código |
|----------|--------|
| <i>A</i> | 0      |
| <i>B</i> | 110    |
| <i>C</i> | 10     |
| <i>D</i> | 1110   |

Mediante este código el mensaje *ABACCCA* se codifica como 0110010101110, el que ocupa sólo 13 bits. En mensajes muy largos que contienen símbolos que aparecen con muy poca frecuencia, los ahorros son sustanciales. Por lo general, los códigos no se construyen sobre la base de la frecuencia de caracteres en un mensaje aislado, sino sobre la base de su frecuencia dentro de todo un conjunto de mensajes. El mismo conjunto de códigos se usa entonces para cada mensaje. Por ejemplo, si los mensajes consisten en palabras del inglés, podría usarse la frecuencia conocida de ocurrencia de las letras del alfabeto de la lengua inglesa, aunque la frecuencia relativa de las letras de un solo mensaje no es necesariamente la misma.

Si se usan códigos de longitud variable, el código para un símbolo no puede ser un prefijo del código para otro. Para descubrir por qué, supóngase que el código para el símbolo *x*, *c(x)*, es un prefijo del código para otro símbolo *y*, *c(y)*. Entonces, cuando se encuentra *c(x)* al examinar de izquierda a derecha, no queda claro si *c(x)* presenta a *x* o si es la primera parte de *c(y)*.

En el caso del ejemplo anterior, la decodificación procede examinando una cadena de caracteres de izquierda a derecha. Cuando se encuentra un 0 como primer bit, el símbolo es *A*; en caso contrario, el símbolo será *B*, *C* o *D*, y se examina el bit siguiente. Si el segundo bit es un 0, el símbolo es una *C*; en caso contrario, tiene que ser una *B* o una *D* y debe examinarse el tercer bit. Si el tercer bit es un 0, el símbolo es una *B*, y si es un 1, es una *D*. Tan pronto como se ha identificado el primer símbolo, se repite el proceso comenzando con el siguiente bit para encontrar el segundo símbolo.

Esto sugiere un método para desarrollar un esquema de codificación óptima, dada la frecuencia de ocurrencia de cada símbolo en un mensaje. Encontrar los dos símbolos que aparecen con menos frecuencia. En el ejemplo citado, éstos son *B* y *D*. El último bit de sus códigos es diferente: 0 para *B* y 1 para *D*. Combinar estos dos símbolos dentro de uno solo, *BD*, cuyo código represente el conocimiento de que el símbolo es una *B* o una *D*. La frecuencia de ocurrencia de este nuevo símbolo es la suma de las frecuencias de los símbolos que lo constituyen. Entonces, la frecuencia de *BD* es dos. Hay ahora tres símbolos: *A* (con frecuencia 3), *C* (con frecuencia 2) y *BD* (con frecuencia 2). Se eligen de nuevo los dos símbolos con menor frecuencia: *C* y *BD*. Los últimos bits de sus códigos difieren otra vez el uno del otro: 0 para *C* y 1 para *BD*. Los dos símbolos se combinan entonces en uno solo, *CBD*, con frecuencia 4. Ahora sólo restan dos símbolos: *A* y *CBD*. Estos se combinan en un solo símbolo *ACBD*. El último bit de sus códigos para *A* y *CBD* difieren uno del otro: 0 para *A* y 1 para *CBD*.

El símbolo *ACBD* contiene todo el alfabeto; a éste se le asigna la cadena de bits nula de longitud 0 como código. Al principio de la decodificación, antes de que ningún bit haya sido examinado, es seguro que todo símbolo está contenido en *ACBD*. A los dos símbolos que componen *ACBD* (*A* y *CBD*) se les asignan los códigos 0 y 1, respectivamente. Si se encuentra un 0, el símbolo codificado es una *A*; si se encuentra un 1, es una *C* una *B* o una *D*. De manera análoga, se asigna a los símbolos que constituyen *CBD* (*C* y *BD*) los códigos 10 y 11, respectivamente. El primer bit indica que el símbolo es uno de los que constituyen a *CBD*; el segundo, si se trata de *C* o de *BD*. Despues se asignan los códigos 110 y 111 a los que componen *BD* (*B* y *D*). Mediante este proceso, se asignan códigos más cortos a los símbolos que aparecen con más frecuencia que a los que aparecen menos.

La acción de combinar dos símbolos en uno sugiere el uso de un árbol binario. Cada nodo del árbol representa un símbolo, y cada hoja, un símbolo del alfabeto original. La figura 5.3.1a muestra el árbol binario construido por medio del ejemplo previo. Cada nodo de la ilustración contiene un símbolo y su frecuencia. La figura 5.3.1b muestra el árbol binario construido mediante este método para el alfabeto y la tabla de frecuencias de la figura 5.3.1c. Tales árboles se llaman *árboles de Huffman* por el descubridor de este método de codificación.

Ya que está elaborado el árbol de Huffman, puede construirse el código para cualquier símbolo del alfabeto comenzando en la hoja que represente ese símbolo y subiendo hacia la raíz. El código se inicializa a *null*. Cada vez que se asciende a una rama izquierda, se agrega 0 al principio del código y cada vez que se asciende a una derecha, se agrega 1 al principio del código.

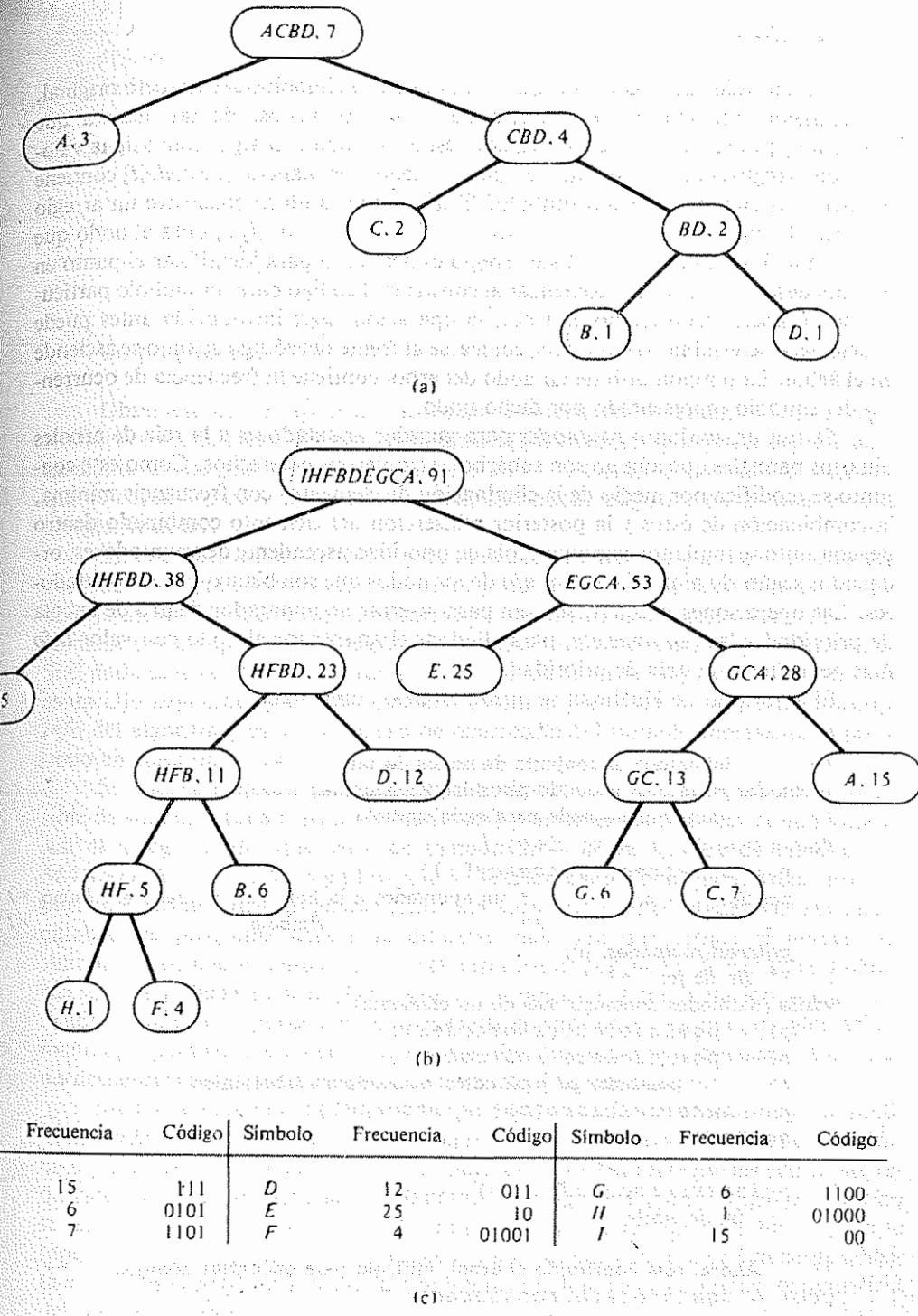


Figura 5.3.1 Árboles de Huffman

## El algoritmo de Huffman

Las entradas del algoritmo son  $n$ , el número de símbolos del alfabeto original, y  $frequency$ , un arreglo del tamaño de  $n$  por lo menos, de tal manera que  $frequency[i]$  es la frecuencia relativa del  $i$ -ésimo símbolo. El algoritmo asigna valores a un arreglo  $code$  del tamaño de  $n$  por lo menos, de manera que  $code[i]$  contiene el código asignado al  $i$ -ésimo símbolo. El algoritmo también construye un arreglo  $position$  del tamaño de  $n$  por lo menos, por lo que  $position[i]$  apunta al nodo que representa el  $i$ -ésimo símbolo. Este arreglo es necesario para identificar el punto en el árbol desde el que se va a comenzar al construir el código para un símbolo particular del alfabeto. Construido el árbol, la operación  $isleft$  introducida antes puede usarse para determinar si 0 o 1 debe colocarse al frente del código cuando se asciende en el árbol. La porción  $info$  de un nodo del árbol contiene la frecuencia de ocurrencia del símbolo representado por dicho nodo.

Se usa un conjunto  $rootnodes$  para guardar apuntadores a la raíz de árboles binarios parciales que aún no son subárboles izquierdos o derechos. Como este conjunto se modifica por medio de la eliminación de elementos con frecuencia mínima, la combinación de éstos y la posterior reinserción del elemento combinado dentro del conjunto se implanta como una cola de prioridad ascendente de apuntadores, ordenados según el valor del campo  $info$  de los nodos que son blanco de los apuntadores. Las operaciones  $pqinsert$ , se usan para insertar un apuntador dentro de la cola de prioridad y las  $pqmindelete$ , para eliminar el apuntador al nodo con valor  $info$  más pequeño de la cola de prioridad.

El algoritmo de Huffman se puede esbozar como sigue:

```
/*
 * inicializar el conjunto de nodos de raíz
 * rootnodes = la cola vacía de prioridad ascendente;
 * construir un nodo para cada símbolo
 */
for (i = 0; i < n; i++) {
 p = maketree(frequency[i]);
 position[i] = p; /* un apuntador a la hoja que contiene el i-ésimo
 símbolo */
 pqinsert(rootnodes, p);
} /* fin de for */
while (rootnodes contenga más de un elemento) {
 p1 = pqmindelete(rootnodes);
 p2 = pqmindelete(rootnodes);
 /* combinar p1 y p2 como ramas de un árbol único */
 p = maketree(info(p1) + info(p2));
 setleft(p, p1);
 setright(p, p2);
 pqinsert(rootnodes, p);
} /* fin de while */

/* Ahora, está construido el árbol; utilícelo para encontrar códigos */
root = pqmindelete(rootnodes);
for (i = 0; i < n; i++) {
 p = position[i];
 ...
}
```

```
code[i] = la cadena de bits nula;
while (p != root) {
 /* ascender por el árbol */
 if (isleft(p))
 code[i] = 0 seguido de code[i];
 else
 code[i] = 1 seguido de code[i];
 p = father(p);
} /* fin de while */
} /* fin de for */
```

## Un programa en C

Obsérvese que el árbol de Huffman es estrictamente binario. Así, si hay  $n$  símbolos en el alfabeto, el árbol de Huffman (que tiene  $n$  hojas) puede representarse mediante un arreglo de nodos de tamaño  $2n - 1$ . Como la cantidad de memoria que se necesita para el árbol es conocida, ésta puede asignarse de antemano en un arreglo de nodos.

En la construcción del árbol y la obtención de los códigos, sólo es necesario guardar una liga de cada nodo a su padre y una indicación de cuándo cada nodo es un hijo derecho o izquierdo; los campos  $left$  y  $right$  son innecesarios. Así, cada nodo contiene tres campos:  $father$ ,  $isleft$  y  $freq$ .  $father$  es un apuntador al padre del nodo. Si el nodo es la raíz, su campo  $father$  es  $NULL$ . El valor de  $isleft$  es  $TRUE$  si el nodo es un hijo izquierdo, y  $FALSE$  en caso contrario.  $freq$  (que corresponde al campo  $info$  del algoritmo) es la frecuencia de ocurrencia del símbolo representado por el nodo en cuestión.

Se reserva memoria para el arreglo  $node$  con base en el número máximo posible de símbolos (una constante  $maxsyms$ ) y no en el número real de símbolos  $n$ . Así el arreglo  $node$ , que sería de tamaño  $2n - 1$ , se declara de tamaño  $2 * MAXSYMS - 1$ . Esto significa que se desperdicia un poco de espacio. Por supuesto,  $n$  mismo puede asignarse como una constante y no como una variable, pero entonces el programa deberá modificarse cada vez que difiera el número de símbolos. Los nodos también se pueden representar por medio de variables dinámicas sin desperdiciar espacio. No obstante, se presenta la implantación con arreglo ligada. (También se puede dar como entrada el valor de  $n$  y asignar arreglos del tamaño apropiado usando  $malloc$  de manera dinámica durante la ejecución. Entonces no desperdiciaría espacio usando una implantación con arreglo.)

Mediante la implantación con arreglo ligada pueden reservarse desde  $node[0]$  hasta  $node[n - 1]$  para las hojas que representan los  $n$  símbolos originales del alfabeto y  $node[n]$  hasta  $node[2 * n - 2]$  para los  $n - 1$  nodos que no son hojas requeridos por el árbol estrictamente binario. Esto significa que el arreglo  $position$  no se requiere como guía para los nodos hoja que representan los  $n$  símbolos, pues se sabe que el nodo que contiene el  $i$ -ésimo símbolo de entrada (donde  $i$  va desde 0 hasta  $n - 1$ ) es  $node[i]$ . Si se usara la representación dinámica con nodos, se requeriría del arreglo  $position$ .

El siguiente programa codifica un mensaje por medio del algoritmo de Huffman. La entrada consta de un número  $n$ , que es el número de símbolos del alfabeto, seguido por un conjunto de  $n$  pares, cada uno de los cuales consiste en un símbolo y su frecuencia relativa. El programa construye primero una cadena  $alph$ , que consta de todos los símbolos del alfabeto, y un arreglo de código tal que  $code[i]$  es el código del  $i$ -ésimo símbolo en  $alph$ . Después, el programa imprime cada carácter, su frecuencia relativa y su código.

Como el código se construye de derecha a izquierda, hay que definir una estructura *codetype* como sigue:

```
#define MAXBITS 50
struct codetype {
 int bits[MAXBITS];
 int startpos;
};
```

*MAXBITS* es el número máximo de bits permitidos en un código. Si un código *cd* es nulo, *cd.startpos* es igual a *MAXBITS*. Cuando se agrega a *cd* un bit *b* a la izquierda, *cd.startpos* decrece en 1 y a *cd.bits[cd.startpos]* se le asigna *b*. Cuando se completa un código *cd*, los bits del código están en las posiciones *cd.startpos* hasta *MAXBITS* - 1, inclusive.

Un aspecto importante es saber cómo organizar la cola de prioridad de los nodos raíces. En el algoritmo, esta estructura de datos se representó como una cola de prioridad de nodos con apunadores. La implantación de la cola de prioridad mediante una lista ligada, como en la sección 4.2, requeriría de un nuevo conjunto de nodos, cada uno de los cuales guardará un apuntador al nodo raíz y un campo *next*. Por fortuna, no se usa el campo *father* de un nodo raíz, por lo que puede emplearse para ligar todos los nodos raíz dentro de una lista. El apuntador *rootnodes* podría apuntar al primer nodo raíz de la lista. La lista misma puede estar ordenada o no, dependiendo de la implantación de *pqinsert* y *pqmindelete*.

En el siguiente programa se hace uso de esta técnica, la que implanta el algoritmo que se acaba de presentar.

```
#define MAXBITS 50
#define MAXSYMB 50
#define MAXNODES 99 /* MAXNODES es igual a 2*MAXSYMB-1 */
struct codetype {
 int bits[MAXBITS];
 int startpos;
};
struct nodetype {
 int freq;
 int father; /* si node[p] no es el nodo raíz
 father señala al nodo padre; si
 lo es, father apunta */
};
```

```
/* al nodo raíz siguiente
 en la cola de prioridad
*/
int isleft;
};

main()
{
 struct codetype cd, code[MAXSYMB];
 struct nodetype node[MAXNODES];
 int i, k, n, p, p1, p2, root, rootnodes;
 char symb, alph[MAXSYMB];
 /* introducir el alfabeto y las frecuencias */
 for (i = 0; i < MAXSYMB; i++)
 alph[i] = ' ';
 rootnodes = 0;
 scanf("%d", &n);
 for (i = 0; i < n; i++) {
 scanf("%s %d", &symb, &node[i].freq);
 pqinsert(rootnodes, i);
 alph[i] = symb;
 } /* fin de for */
 /* ahora se construyen los árboles */
 for (p = n; p < 2*n-1; p++) {
 /* p apunta al siguiente nodo disponible. Obtener los nodos
 raíz p1 y p2 con las menores frecuencias */
 p1 = pqmindelete(rootnodes);
 p2 = pqmindelete(rootnodes);
 /* asignar left(p) a p1 y right(p) a p2 */
 node[p1].father = p;
 node[p1].isleft = TRUE;
 node[p2].father = p;
 node[p2].isleft = FALSE;
 node[p].freq = node[p1].freq + node[p2].freq;
 pqinsert(rootnodes, p);
 } /* fin de for */
 /* ahora hay un solo nodo izquierdo */
 /* con un campo father vacío */
 root = pqmindelete(rootnodes);
 /* extraer los códigos por medio del árbol */
 for (i = 0; i < n; i++) {
 /* inicializar code[i] */
 cd.startpos = MAXBITS;
 /* ascender por el árbol */
 p = i;
 while (p != root) {
 --cd.startpos;
 if (node[i].isleft)
 cd.bits[cd.startpos] = 0;
```

```

 else
 cd.bits[cd.startpos] = 1;
 p = node[p].father;
 } /* fin de while */
 for (k = code[i].startpos; k < MAXBITS; k++)
 code[i].bits[k] = cd.bits[k];
 code[i].startpos = cd.startpos;
} /* fin de for */
/* impresión de resultados */
for (i = 0; i < n; i++) {
 printf("\n%c %d ", alph[i], nodes[i].freq);
 for (k = code[i].startpos; k < MAXBITS; k++)
 printf("%d", code[i].bits[k]);
 printf("\n");
} /* fin de for */
} /* fin de main */

```

Al lector se le deja el código de la rutina *encode(alph, code, msge, bitcode)*. Este procedimiento acepta la cadena *alph*, el arreglo *code* construido en el programa anterior y un mensaje *msge*, y asigna a *bitcode* la cadena de bits que codifica el mensaje.

Dada la codificación de un mensaje y el árbol de Huffman usado en la construcción del código, el mensaje original puede recobrarse como sigue. Hay que comenzar en la raíz del árbol. Cada vez que se encuentre un 0, hay que mover hacia abajo una rama izquierda, y cada vez que se encuentre un 1, hay que moverse hacia abajo una rama derecha. Repetir el proceso anterior hasta encontrar una hoja. El siguiente carácter del mensaje original es el símbolo que corresponde a esa hoja. Buscar si se puede codificar 1110100010111011 mediante el árbol de Huffman de la figura 5.3.1b.

Para ello, es necesario ir de la raíz del árbol hacia abajo, hasta sus hojas. Esto significa que en lugar de los campos *father* e *isleft*, se necesitan dos campos *left* y *right* para guardar los hijos derecho e izquierdo de un nodo particular. El cálculo de los campos *left* y *right*, a partir de los campos *father* e *isleft*, es directo. Otra posibilidad es construir los valores *left* y *right* de manera directa a partir de la información de frecuencia para los símbolos del alfabeto y mediante una aproximación similar a la usada en la asignación del valor de *father*. (Por supuesto, para que los árboles sean idénticos, los pares símbolo/frecuencia se deben presentar en el mismo orden en cualquiera de los dos métodos.) Al lector se le dejan como ejercicio estos algoritmos, así como el algoritmo de decodificación.

## EJERCICIOS

- 5.3.1. Escriba en C una función *encode(alph, code, msge, bitcode)*. La función acepta la cadena *alph*, el arreglo *code* producido por el programa *findcode* en el texto y un mensaje *msge*. El procedimiento asigna a *bitcode* la codificación de Huffman de dicho mensaje.
- 5.3.2. Escriba en C una función *decode(alph, left, right, bitcode, msge)*, en donde *alph* sea la cadena producida por el programa *findcode* en el texto, *left* y *right* sean arreglos usa-

dos para representar un árbol de Huffman y *bitcode* sea una cadena de bits. La función asigna a *msge* la decodificación de Huffman de *bitcode*.

- 5.3.3. Implante la cola de prioridad *rootnodes* como una lista ordenada. Escribir rutinas *pqinsert* y *pqmindelete* apropiadas.
- 5.3.4. ¿Es posible tener dos árboles de Huffman diferentes para un conjunto de símbolos con determinadas frecuencias? Pruebe que sólo existe un árbol como ése o dé un ejemplo en el que haya dos de éstos.
- 5.3.5. Defina el *árbol binario de Fibonacci de orden n* como sigue: Si  $n = 0$  o  $n = 1$ , el árbol consta de un solo nodo. Si  $n > 1$ , el árbol consta de una raíz, con el árbol de Fibonacci de orden  $n - 1$  como subárbol izquierdo y el árbol de Fibonacci de orden  $n - 2$  como subárbol derecho.
  - a. Escriba una función en C que proporcione un apuntador al árbol binario de Fibonacci de orden  $n$ .
  - b. ¿Tal árbol es estrictamente binario?
  - c. ¿Cuál es el número de hojas en el árbol de Fibonacci de orden  $n$ ?
  - d. ¿Cuál es la profundidad del árbol de Fibonacci de orden  $n$ ?
- 5.3.6. Dado un árbol binario *t*, su *extensión* se define como el árbol binario *e(t)* formado de *t* mediante la adición de una hoja a cada apuntador izquierdo y derecho *NULL* en *t*. Las hojas nuevas se llaman nodos *externos*, y los nodos originales (que ahora son nodos no-hoja) se llaman nodos *internos*. *e(t)* se llama un *árbol binario extendido*.
  - a. Pruebe que un árbol binario extendido es estrictamente binario.
  - b. Si *t* tiene  $n$  nodos, ¿cuántos nodos tiene *e(t)*?
  - c. Pruebe que todas las hojas de un árbol binario extendido son nodos recién agregados.
  - d. Escriba una rutina en C que extienda un árbol binario *t*.
  - e. Pruebe que un árbol estrictamente binario con más de un nodo es una extensión de un solo árbol binario.
  - f. Escriba una función en C que acepte un apuntador a un árbol estrictamente binario *t1* que contenga más de un nodo y elimine nodos de *t1* creando un árbol binario *t2* de tal manera que  $t1 = e(t2)$ .
  - g. Muestre que el árbol binario completo de orden  $n$  es la  $n$ -ésima extensión del árbol binario que consiste en un solo nodo.
- 5.3.7. Dado un árbol estrictamente binario *t* en el que las  $n$  hojas están etiquetadas como nodos del 1 hasta  $n$ , sea *level(i)* el nivel del nodo *i* y *freq(i)* un entero asignado al nodo *i*. Definir la *longitud de camino con pesos* de *t* como la suma de  $freq(i) * level(i)$  sobre todas las hojas de *t*.
  - a. Escriba una rutina en C para calcular la longitud de camino con peso, dados los campos *freq* y *father*.
  - b. Muestre que el árbol de Huffman es el árbol estrictamente binario con longitud de camino con peso mínima.

## 5.4. REPRESENTACION DE LISTAS COMO ARBOLES BINARIOS

Hay varias operaciones que pueden ejecutarse con una lista de elementos. Entre estas operaciones están la suma de un nuevo elemento en el frente o en el final de la lista, la eliminación del primer o último elemento de una lista, la recuperación del  $k$ -ésimo elemento o del último elemento de la lista, la inserción de un elemento seguido o precedido por un elemento dado, la eliminación de un elemento dado y la

eliminación del antecesor o el sucesor de un elemento dado. La construcción de una lista con elementos dados es una operación adicional que se usa con frecuencia.

De acuerdo con la representación que se elija para una lista, algunas operaciones pueden o no realizarse con grados variables de eficacia. Por ejemplo, una lista se puede representar mediante elementos sucesivos en un arreglo o como nodos en una estructura ligada. La inserción de un elemento que sigue a un elemento dado es más o menos eficaz en una lista ligada (lo que implica modificaciones a unos cuantos apuntadores, además de la inserción real) pero relativamente ineficaz en un arreglo (pues implica el movimiento de todos los elementos subsecuentes una posición dentro del arreglo). Sin embargo, encontrar el elemento  $k$ -ésimo de una lista es mucho más eficaz en un arreglo (implica sólo el cálculo de un desplazamiento) que en una estructura ligada (la que requiere pasar a lo largo de los  $k - 1$  primeros elementos). De manera análoga, no es posible eliminar un elemento específico de una lista lineal ligada no doble si sólo se da un apuntador a dicho elemento, y sólo es posible hacerlo de manera poco eficaz en una lista ligada circular no doble (si se recorre toda la lista para alcanzar el elemento previo y después se ejecuta la eliminación). La misma operación, sin embargo, es muy eficaz en una lista doblemente ligada (lineal o circular).

En esta sección se presenta una representación con árboles de una lista lineal en la cual las operaciones para encontrar el  $k$ -ésimo elemento de la lista y eliminar un elemento específico son un tanto eficaces. Mediante esta representación, también se puede construir una lista con elementos determinados. También se considera en forma breve la operación de insertar un solo elemento nuevo.

Como se ilustra en la figura 5.4.1, una lista puede representarse mediante un árbol binario. La figura 5.4.1a muestra una lista en el formato ligado regular, mientras que las figuras 5.4.1b y c muestran dos árboles binarios que representan dicha lista. Los elementos de la lista original se representan por medio de hojas del árbol (las que aparecen en la figura como cuadrados), mientras que los nodos no hoja del árbol (que aparecen en la figura como círculos) están presentes como parte de la estructura interna del árbol. Asociado a cada nodo hoja están los contenidos del correspondiente elemento de la lista. Asociado a cada nodo no hoja hay un contador que representa el número de hojas del subárbol izquierdo del nodo. (Aunque este conteo se puede calcular de la estructura del árbol, se guarda como un elemento de datos para no tener que volver a calcular su valor cada vez que se necesita). Los elementos de la lista en su secuencia original están asignados a las hojas del árbol en la secuencia de en orden de las hojas. Advierta que en la figura 5.4.1 varios árboles binarios pueden representar la misma lista.

### Hallazgo del elemento $k$ -ésimo

Para justificar el uso de tantos nodos extra para representar una lista, se presenta un algoritmo para encontrar el elemento  $k$ -ésimo de una lista representada mediante un árbol. Sea  $tree$  un apuntador a la raíz del árbol y  $lcount(p)$  el conteo asociado al nodo no hoja apuntado por  $p$ ;  $lcount(p)$  es el número de hojas del árbol que tiene raíz en  $node(left(p))$ . El siguiente algoritmo emplea la variable  $find$  para apuntar a la hoja que contiene el elemento  $k$ -ésimo de la lista.

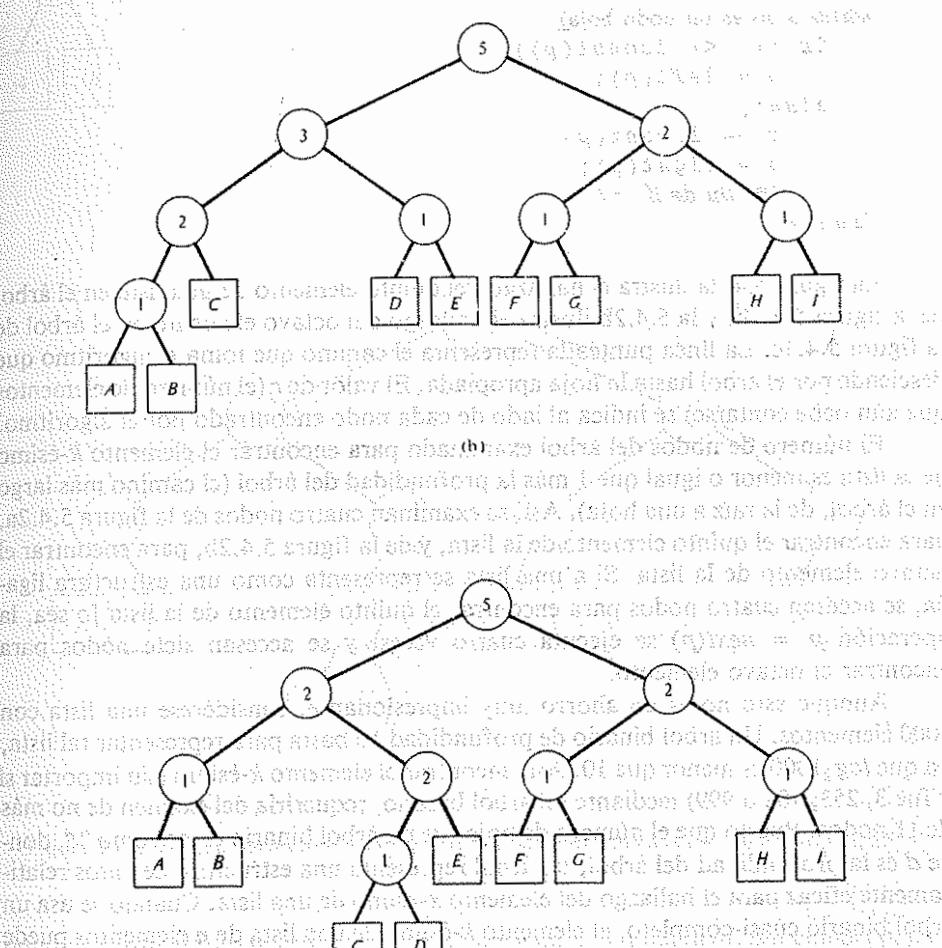
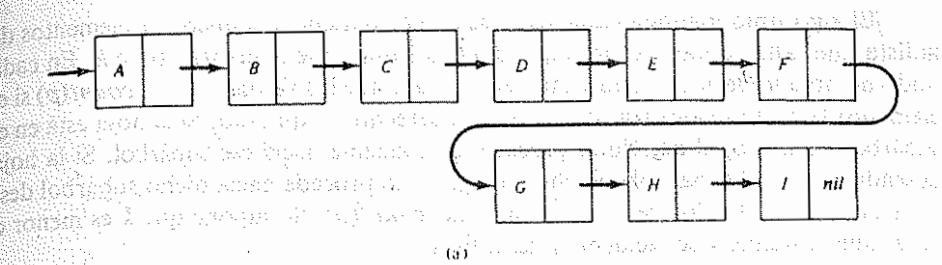


Figura 5.4.1 Una lista y dos árboles binarios correspondientes.

El algoritmo mantiene una variable  $r$  que contiene el número de elementos de la lista que falta contar. Al principio del algoritmo,  $r$  se inicializa como  $k$ . En cada nodo no hoja  $node(p)$ , el algoritmo determina desde los valores de  $r$  y  $lcount(p)$  si el elemento  $k$ -ésimo se localiza en el subárbol derecho o izquierdo. Si la hoja está en el subárbol izquierdo, el algoritmo procede directamente hacia ese subárbol. Si la hoja deseada está en el subárbol derecho, el algoritmo procede hacia dicho subárbol después de reducir el valor de  $r$  por el valor de  $lcount(p)$ . Se supone que  $k$  es menor o igual que el número de elementos de la lista.

```

r = k;
p = tree;
while p no es un nodo hoja
 if (r <= lcount(p))
 p = left(p);
 else {
 r -= lcount(p);
 p = right(p);
 } /* fin de if */
find = p;

```

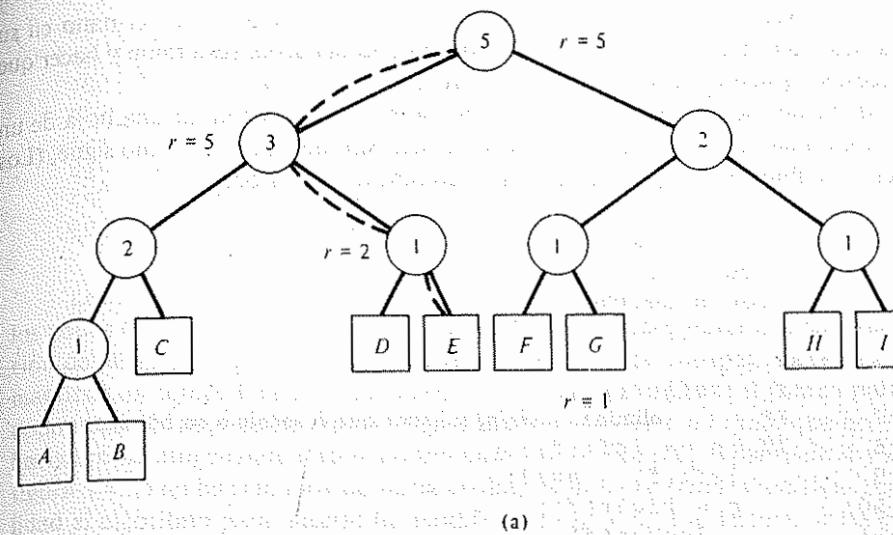
La figura 5.4.2a ilustra el hallazgo del quinto elemento de una lista en el árbol de la figura 5.4.1b, y la 5.4.2b ilustra el hallazgo del octavo elemento en el árbol de la figura 5.4.1c. La línea punteada representa el camino que toma el algoritmo que desciende por el árbol hasta la hoja apropiada. El valor de  $r$  (el número de elementos que aún debe contarse) se indica al lado de cada nodo encontrado por el algoritmo.

El número de nodos del árbol examinado para encontrar el elemento  $k$ -ésimo de la lista es menor o igual que 1 más la profundidad del árbol (el camino más largo en el árbol, de la raíz a una hoja). Así, se examinan cuatro nodos de la figura 5.4.2a, para encontrar el quinto elemento de la lista, y de la figura 5.4.2b, para encontrar el octavo elemento de la lista. Si a una lista se representa como una estructura ligada, se acceden cuatro nodos para encontrar el quinto elemento de la lista [o sea, la operación  $p = next(p)$  se ejecuta cuatro veces] y se acceden siete nodos para encontrar el octavo elemento.

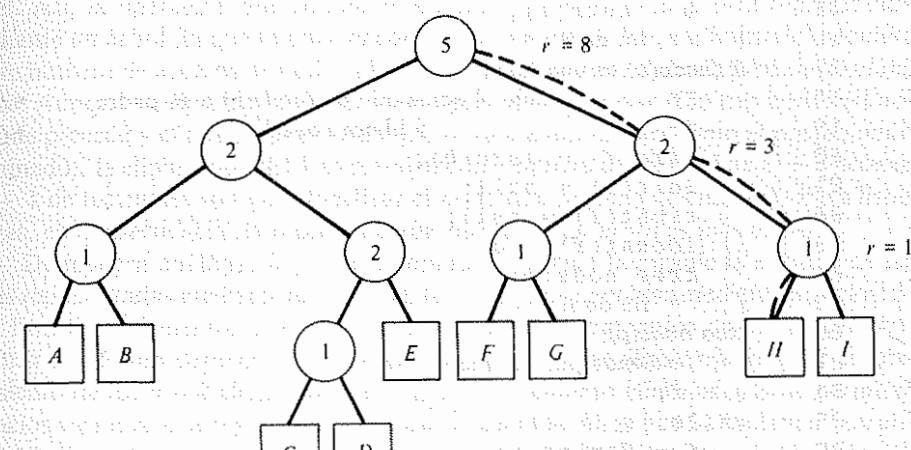
Aunque esto no es un ahorro muy impresionante, considérese una lista con 1000 elementos. Un árbol binario de profundidad 10 basta para representar tal lista, ya que  $\log_2 1000$  es menor que 10. Así, encontrar el elemento  $k$ -ésimo (sin importar si  $k$  fue 3, 253, 708 o 999) mediante tal árbol binario, requeriría del examen de no más de 11 nodos. Puesto que el número de hojas de un árbol binario crece como  $2^d$ , donde  $d$  es la profundidad del árbol, tal árbol representa una estructura de datos relativamente eficaz para el hallazgo del elemento  $k$ -ésimo de una lista. Cuando se usa un árbol binario quasi-completo, el elemento  $k$ -ésimo de una lista de  $n$  elementos puede encontrarse en  $\log_2 n + 1$  accesos de nodos como máximo, mientras que si se usara una lista lineal ligada se requerirían  $k$  accesos.

#### Eliminación de un elemento

¿Cómo puede eliminarse un elemento de una lista representada por un árbol? La eliminación misma es relativamente fácil. Implica sólo la asignación de *null* un



(a)



(b)

Figura 5.4.2 Hallazgo del  $n$ -ésimo elemento de una lista representada por medio de un árbol.

apuntador izquierdo ó derecho en el padre de la hoja eliminada *dl*. Sin embargo, para habilitar los accesos subsecuentes, hay que modificar los conteos en todos los ancestros de *dl*. La modificación consiste en reducir *lcount* en 1 en cada nodo *nd* del que *dl* fue un descendiente izquierdo, ya que el número de hojas del subárbol izquierdo de *nd* es uno menos. Al mismo tiempo, si el hermano de *dl* es una hoja, se puede trasladar hacia arriba del árbol para tomar el lugar de su padre. Por lo tanto,

se puede mover ese nodo hacia arriba, aún más lejos, si no tiene hermano en su nueva posición. Esto puede reducir la profundidad del árbol resultante y hacer que los accesos siguientes sean un poco más eficaces.

Se puede, por lo tanto, presentar un algoritmo para eliminar una hoja de un árbol apuntada por  $p$  (y en consecuencia un elemento de una lista) como sigue. (Los números de línea a la izquierda son para referencias posteriores.)

```

1 if (p == tree) {
2 tree = null;
3 free node(p);
4 }
5 else {
6 f = father(p);
7 /* ... eliminar node(p) y hacer que b señale a su hermano */
8 if (p == left(f)) {
9 left(f) = null;
10 b = right(f);
11 --lcount(f);
12 }
13 else {
14 right(f) = null;
15 b = left(f);
16 } /* fin de if */
17 if (node(b) es una hoja) {
18 /* traslade el contenido de (node(b)) a su padre */
19 /* y libere node(b) */
20 info(f) = info(b);
21 left(f) = null;
22 right(f) = null;
23 lcount(f) = 0;
24 free node(b);
25 } /* fin de if */
26 free node(p);
27 /* ascienda por el árbol */
28 q = f;
29 while (q != tree) {
30 f = father(q);
31 if (q == left(f)) {
32 /* la hoja eliminada fue un descendiente */
33 /* izquierdo de node(f) */
34 --lcount(f);
35 b = right(f);
36 }
37 else
38 b = left(f);
39 /* node(b) es el hermano de node(q) */
40 if (b == null & node(q) es una hoja) {
41 /* ... traslade a su padre los contenidos */
42 /* de node(q) y libérello */
43 info(f) = info(q);
44 right(f) = null;
45 lcount(f) = 0;
46 free node(q);
47 } /* fin de if */
48 q = f;
49 } /* fin de while */
50 } /* fin de else */

```

```

43 left(f) = null;
44 right(f) = null;
45 lcount(f) = 0;
46 free node(q);
47 } /* fin de if */
48 q = f;
49 } /* fin de while */
50 } /* fin de else */

```

La figura 5.4.3 ilustra los resultados de este algoritmo para un árbol en el cual los nodos  $C$ ,  $D$  y  $B$  se eliminan en ese orden. Hay que asegurarse que se sigue la acción del algoritmo en esos ejemplos. Obsérvese que, por consistencia, el algoritmo mantiene un conteo 0 en los nodos hoja, aunque no se requiere el conteo para tales nodos. Adviértase también que el algoritmo nunca mueve un nodo que no es hoja hacia arriba, aun cuando esto se podría hacer. (Por ejemplo, el padre de  $A$  y  $B$  en la figura 5.4.3b no ha sido movido hacia arriba). Aunque se puede modificar con facilidad el algoritmo para hacerlo (la modificación se le deja al lector), se ha evitado por razones que serán evidentes en breve.

Este algoritmo de eliminación implica la inspección de dos nodos hacia arriba (el ancestro del nodo que se está eliminando y el hermano de ese ancestro) en cada nivel. Así, la operación que elimina el  $k$ -ésimo elemento de una lista representada mediante un árbol (la que implica encontrar el elemento y luego eliminarlo) requiere de un número de accesos de nodos más o menos igual a tres veces la profundidad del árbol. Aunque la eliminación en una lista ligada exige accesar sólo tres nodos (el nodo que precede y el que sigue al nodo que se va a eliminar, así como el propio nodo eliminado), la eliminación del  $k$ -ésimo elemento requiere de un total de  $k + 2$  accesos ( $k - 1$  de los cuales son para localizar el nodo que precede al  $k$ -ésimo). Para listas largas, en consecuencia, la representación mediante un árbol es más eficaz.

De manera análoga, se puede comparar en forma favorable la eficacia de las listas representadas mediante árboles con las listas representadas mediante arreglos. Si una lista de  $n$  elementos se guarda en los primeros  $n$  elementos de un arreglo, encontrar el  $k$ -ésimo elemento requiere un solo acceso, pero eliminarlo exige el desplazamiento de los  $n - k$  elementos que seguían al elemento eliminado. Si se permiten espacios vacíos en el arreglo, de manera que la eliminación se pueda implantar con eficacia (colocando un indicador en la posición del arreglo que ocupa el elemento eliminado, en lugar de recorrer todos los elementos que le suceden), el hallazgo del elemento  $k$ -ésimo requiere como mínimo de  $k$  accesos en el arreglo. La razón es que ya no es posible conocer la posición en el arreglo del  $k$ -ésimo elemento de la lista, ya que pueden existir espacios entre los elementos del arreglo. (Debe observarse, sin embargo, que si el orden de los elementos de la lista es irrelevante, el elemento  $k$ -ésimo de un arreglo puede eliminarse eficazmente remplazándolo por el elemento en la posición  $n$  [el último elemento] y ajustando el conteo a  $n - 1$ . Sin embargo, es improbable que se desee eliminar el  $k$ -ésimo elemento de una lista en la que el orden sea irrelevante, pues entonces ya no tendría importancia el  $k$ -ésimo elemento sobre los demás.)

Insertar un nuevo  $k$ -ésimo elemento dentro de una lista representada por un árbol [entre el  $(k - 1)$  y el  $k$ -ésimo previo] es también una operación relativamente

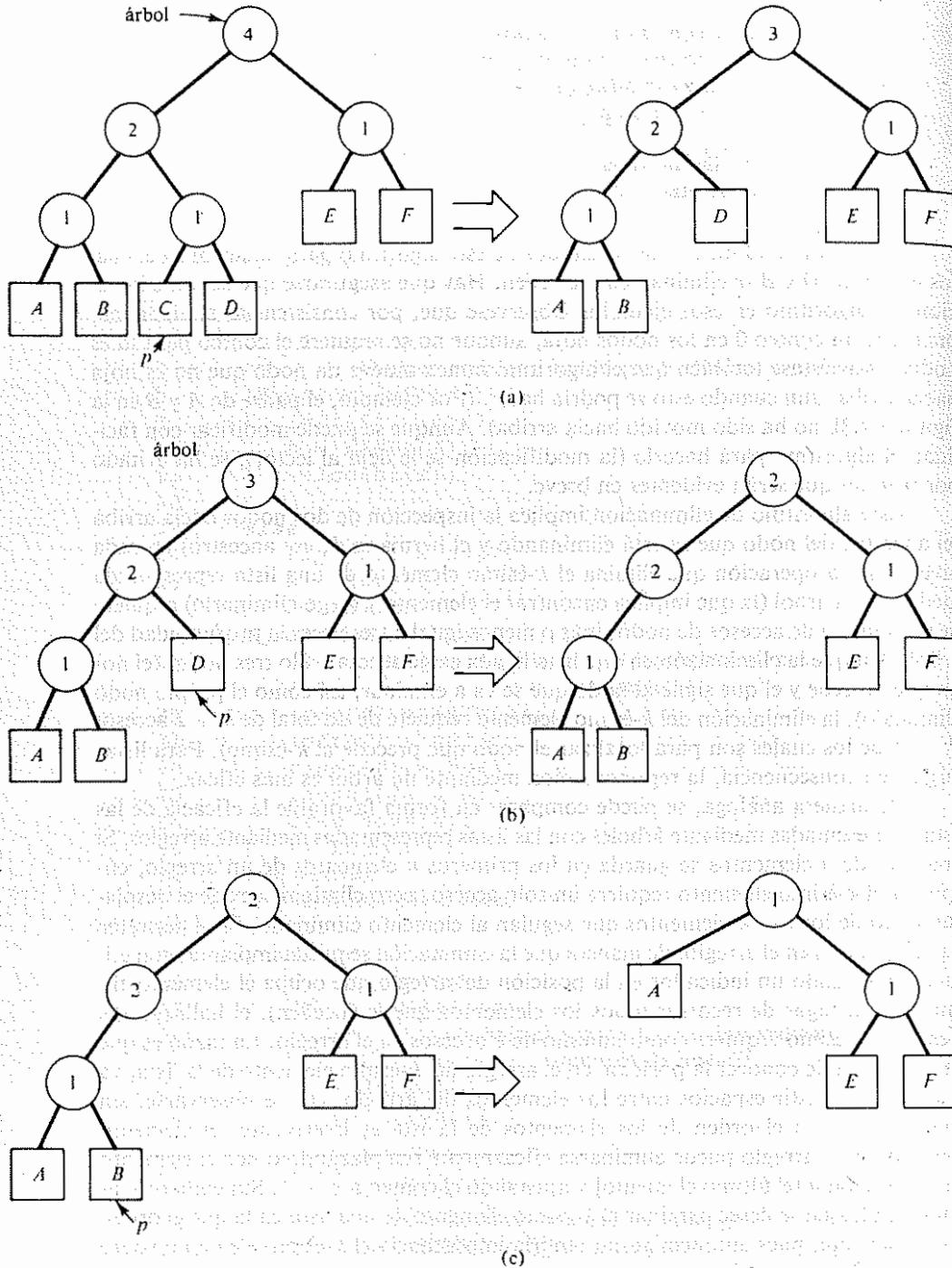


Figura 5.4.3 El algoritmo de eliminación.

eficaz. La inserción consiste en localizar el  $k$ -ésimo elemento, remplazarlo por un nuevo nodo que no es hoja que tenga una hoja con el nuevo elemento como hijo izquierdo y una hoja que contenga al antiguo  $k$ -ésimo elemento como hijo derecho y ajustar conteos apropiados entre sus ancestros. Al lector se le dejan los detalles. (Sin embargo, agregar de manera repetida un nuevo  $k$ -ésimo elemento con este método provoca que el árbol quede muy desequilibrado, ya que la rama que contiene el elemento  $k$ -ésimo se vuelve larga y desproporcionada en relación a las demás. Esto significa que la eficacia del hallazgo del elemento  $k$ -ésimo no es tan fabuloso como lo sería en un árbol balanceado, en el que todos los caminos tienen más o menos la misma longitud. Se invita al lector a encontrar una estrategia de “balanceo” para aliviar este problema. A pesar de este problema, si se hacen las inserciones en el árbol de manera aleatoria, de tal manera que sea igualmente probable insertar un elemento en cualquier posición dada, el árbol resultante permanece por lo regular balanceado y el hallazgo del elemento  $k$ -ésimo sigue siendo eficaz.)

### Implantación de listas representadas por medio de árboles en C

Las implantaciones de los algoritmos de búsqueda y eliminación en C son directas por medio de la representación ligada de árboles binarios. Sin embargo, una representación de este tipo requiere de campos *info*, *lcount*, *father*, *left* y *right* para cada nodo del árbol, mientras que un nodo de una lista sólo necesita campos *info* y *next*. Aunado al hecho de que la representación por medio de un árbol requiere de casi dos veces más nodos que la representación por medio de una lista ligada, este requerimiento de espacio puede hacer que la representación por medio de un árbol sea poco práctica. Se podría, por supuesto, utilizar nodos externos que contengan sólo un campo *info* (y quizás un campo *father*) para las hojas y nodos internos que contengan campos *lcount*, *father*, *left* y *right* para los nodos no hojas. Aquí no se considera esta posibilidad.

En la representación secuencial de un árbol binario, los requerimientos de espacio están lejos de ser tan fabulosos. Si se asume que no se necesitan inserciones una vez que se construye el árbol y que se conoce el tamaño inicial de la lista, se puede apartar un arreglo para guardar la representación de la lista como un árbol estrictamente binario quasi-completo. En esa representación los campos *father*, *left* y *right* son innecesarios. Como se mostrará más adelante, siempre es posible construir una representación de una lista por medio de un árbol binario quasi-completo.

Ya que se construyó el árbol, los únicos campos requeridos son *info*, *lcount* y un campo que indique cuando un elemento del arreglo representa un nodo existente o uno eliminado. También, como ya se observó antes, *lcount* sólo es necesario para nodos del árbol que no sean hojas, por lo que podría usarse una estructura con el campo *lcount* o el campo *info*, dependiendo de si el nodo es o no una hoja. Se deja esa posibilidad como ejercicio al lector. También es posible eliminar la necesidad del campo *used* con un cierto costo de eficacia en tiempo (ver ejercicios 5.4.4 y 5.4.5). Se asumen las siguientes definiciones y declaraciones (suponer 100 elementos en la lista):

```

#define MAXELTS 1001 /* ... Número máximo de elementos en lista */
#define NUMNODES 2*MAXELTS - 1 /* ... Número de nodos en la lista */
#define BLANKS 100 /* ... Número de espacios blancos */
struct nodetype {
 char info[20];
 int lcount; /* ... Número de hijos de un nodo no hoja */
 int used; /* ... Indica si el nodo es una hoja o no */
} node[NUMNODES];

```

Un nodo no hoja se puede reconocer mediante un valor *info* igual a *BLANKS*. *father(p)*, *left(p)* y *right(p)* pueden implantarse de manera regular como  $(p - 1)/2$ ,  $2 * p + 1$  y  $2 * p + 2$ , respectivamente.

A continuación se presenta una rutina en C para encontrar el *k*-ésimo elemento. Usar la rutina de biblioteca *strcmp*, la que da como resultado 0 si dos cadenas son iguales.

```

/* No se devuelven los espacios blancos que aparecen en node[0].info */
findelement(k)
int k;
{
 /* Se intenta mover el apuntador al elemento que se encuentra en la i-ésima posición. Si se obtiene una cadena vacía, se considera que el elemento deseado es el que aparece en el apuntador. */
 int p, r;
 if (k <= 0) /* ... Si el número deseado es menor o igual que el primero, se considera que el elemento deseado es el que aparece en el apuntador. */
 r = k;
 else /* ... Si el número deseado es mayor que el primero, se considera que el elemento deseado es el que aparece en el apuntador más uno. */
 r = k + 1;
 while (strcmp(node[p].info, BLANKS) != 0) /* ... Si el elemento deseado es una hoja, se sigue buscando. */
 if (r <= node[p].lcount) /* ... Si el elemento deseado es un hermano de la izquierda, se sigue buscando. */
 p = p * 2 + 1; /* ... Se actualiza el apuntador para que apunte al hermano de la izquierda. */
 else /* ... Si el elemento deseado es un hermano de la derecha, se sigue buscando. */
 r = node[p].lcount;
 p = p * 2 + 2; /* ... Se actualiza el apuntador para que apunte al hermano de la derecha. */
 /* fin de if */
 return(p);
} /* fin de findelement */

```

La rutina en C para eliminar la hoja apuntada por *p* mediante la representación secuencial es un poco más sencilla que el algoritmo correspondiente presentado con anterioridad. Se pueden ignorar todas las asignaciones de *null* (lineas 2, 9, 14, 21, 22, 43 y 44) debido a que no se usan apunadores. También se pueden ignorar las asignaciones de 0 a un campo *lcount* (lineas 23 y 45), pues dicha asignación es parte de la conversión de un nodo no hoja a hoja y en esta representación en C no se usa el campo *lcount* para nodos hojas. Un nodo puede reconocerse como hoja (lineas 17 y 39) por un blanco como valor *info*, y el apuntador *b* como *null* (línea 39) por un valor *FALSE* para *node[b].used*. La liberación de un nodo (lineas 3, 26 y 46) se realiza haciendo su campo *used* igual a *FALSE*. La rutina utiliza la rutina de biblioteca *strcpy(s,t)*, la que asigna la cadena *t* a la cadena *s* y la rutina *strcmp* para verificar si dos cadenas son iguales.

```

delete(p) /* ... Elimina el elemento que apunta a través del apuntador p */
int p;
{
 /* ... Se verifica si el elemento que apunta a través del apuntador p es una hoja */
 int b, f, q;
 if (p == 0) /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 node[p].used = FALSE; /* ... Se establece el campo used en FALSE */
 else {
 f = (p - 1) / 2; /* ... Se calcula el apuntador al padre */
 if (p % 2 != 0) { /* ... Si el elemento que apunta a través del apuntador p es un hermano de la izquierda */
 b = 2 * f + 2; /* ... Se calcula el apuntador al hermano de la derecha */
 --node[f].lcount; /* ... Se actualiza el campo lcount */
 } /* ... Si el elemento que apunta a través del apuntador p es un hermano de la derecha */
 else /* ... Si el elemento que apunta a través del apuntador p es un hermano de la izquierda */
 b = 2 * f + 1; /* ... Se calcula el apuntador al hermano de la izquierda */
 if (strcmp(node[b].info, BLANKS) != 0) /* ... Si el elemento deseado es una hoja */
 strcpy(node[f].info, node[b].info); /* ... Se copia la información del hermano de la izquierda al padre */
 node[b].used = FALSE; /* ... Se establece el campo used en FALSE */
 } /* ... fin de if */
 node[p].used = FALSE; /* ... Se establece el campo used en FALSE */
 /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 if (node[p].used) /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 node[p].info[0] = '\0'; /* ... Se establece el campo info en blanco */
 /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 while (q != 0) /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 if (f = (q - 1) / 2) /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 if (q % 2 != 0) { /* ... Si el elemento que apunta a través del apuntador p es un hermano de la izquierda */
 b = 2 * f + 2; /* ... Se calcula el apuntador al hermano de la derecha */
 } /* ... Si el elemento que apunta a través del apuntador p es un hermano de la derecha */
 else /* ... Si el elemento que apunta a través del apuntador p es un hermano de la izquierda */
 b = 2 * f + 1; /* ... Se calcula el apuntador al hermano de la izquierda */
 if (!node[b].used && strcmp(node[q].info, BLANKS) != 0) /* ... Si el elemento deseado es una hoja */
 strcpy(node[f].info, node[q].info); /* ... Se copia la información del hermano de la izquierda al padre */
 node[q].used = FALSE; /* ... Se establece el campo used en FALSE */
 } /* ... fin de if */
 q = f; /* ... Se actualiza el apuntador para que apunte al hermano de la izquierda */
 } /* ... fin de if */
 /* ... Si el elemento que apunta a través del apuntador p es una hoja */
 /* ... Si el elemento que apunta a través del apuntador p es una hoja */
} /* ... fin de delete */

```

Nuestro uso de la representación secuencial explica la razón para no trasladar un nodo no hoja sin hermano más arriba del árbol durante la eliminación. En la representación secuencial, dicho proceso de traslación hacia arriba implica copiar los contenidos de todos los nodos al subárbol dentro del arreglo, en tanto que si se usa la representación ligada implica la modificación de un solo apuntador.

## Construcción de una lista representada mediante un árbol

Ahora se toma otra vez el postulado de que, dada una lista de  $n$  elementos, es posible construir un árbol estrictamente binario quasi-completo que represente la lista. Ya se observó en la sección 5.1 que es posible construir un árbol estrictamente binario quasi-completo con  $n$  hojas y  $2^d - 1$  nodos. Las hojas de un árbol de este tipo ocupan los nodos numerados de  $n - 1$  a  $2^d - 2$ . Si  $d$  es el entero más pequeño, de manera que  $2^d$  sea mayor o igual que  $n$  (esto es, si  $d$  es igual al entero más pequeño que es mayor o igual que  $\log_2 n$ ),  $d$  es igual a la profundidad del árbol. El número asignado al primer nodo en el nivel del fondo del árbol es  $2^d - 1$ . A los primeros elementos de la lista se les asignan los nodos del  $2^d - 1$  al  $2^d - 2$  y al resto (si lo hay), los nodos numerados de  $n - 1$  a  $2^d - 2$ . En la construcción de un árbol que representa una lista con  $n$  elementos, se pueden asignar elementos a los campos *info* de las hojas del árbol en esa secuencia y asignar una cadena de blancos a los campos *info* de nodos no hojas, numerados de 0 a  $n - 2$ . También es muy simple inicializar el campo *used* como *true* en todos los nodos numerados de 0 a  $2^d - 2$ .

Inicializar los valores del arreglo *lcount* es más difícil. Se pueden usar dos métodos: uno que implica más tiempo y otro que implica más espacio. En el primer método se inicializan con 0 todos los campos *lcount*. Luego, se asciende por el árbol, por turnos, de cada hoja a la raíz. Cada vez que se alcanza un nodo desde su hijo izquierdo, se agrega 1 a su campo *lcount*. Después de ejecutar este proceso para cada hoja, todos los valores de *lcount* quedan asignados apropiadamente. La siguiente rutina usa este método para construir un árbol de una lista de datos de entrada:

```
buildtree(n)
int n;
{
 int d, f, i, p, power, size;
 /* calcular la profundidad del árbol d y el valor de 2d */
 d = 0;
 power = 1;
 while (power < n) {
 ++d;
 power *= 2;
 } /* fin de while */
 /* asignar los elementos de la lista, inicializar las etiquetas empleadas e inicializar el campo lcount con 0 en todos los nodos que no son hojas */
 size = 2*n - 1;
 for (i = power-1; i < size; i++) {
 scanf("%d", &node[i].info);
 node[i].used = TRUE;
 } /* fin de for */
 for (i=n-1; i < power-1; i++){
 scanf("%s", node[i].info);
 node[i].used = TRUE;
 } /* fin de for */
}
```

```
for (i=0; i < n-1; i++) {
 node[i].used = TRUE;
 node[i].lcount = 0;
 strcpy(node[i].info, BLANKS);
} /* fin de for */
/* colocar valores en los campos lcount */
for (i=n-1; i < size; i++) {
 /* seguir la ruta de cada hoja a la raíz */
 p = i;
 while (p != 0) {
 f = (p-1) / 2;
 if (p % 2 != 0)
 ++node[f].lcount;
 p = f;
 } /* fin de while */
} /* fin de for */
} /* fin de buildtree */
```

El segundo método usa un campo adicional, *rcount*, en cada nodo para guardar el número de hojas en el subárbol derecho de cada nodo no hoja. Este campo, así como el campo *lcount*, se hace igual a 1 en cada nodo no hoja que sea padre de dos hojas. Si  $n$  es impar, de tal manera que hay un nodo [numerado  $(n - 3)/2$ ] que es el padre de una hoja y de una no hoja, se asigna 2 a *lcount* en ese nodo y 1 a *rcount*.

El algoritmo avanza luego sobre los elementos restantes del arreglo en orden inverso, asignando a *lcount* en cada nodo a la suma de *lcount* y *rcount* en el hijo izquierdo del nodo y a *rcount* la suma de *lcount* y *rcount* en el hijo derecho del nodo. Se deja al lector la implantación en C de esta técnica. Obsérvese que *rcount* puede implantarse como un arreglo local en *buildtree* y no como un campo en cada nodo, ya que sus valores dejan de usarse una vez que se termina de construir el árbol.

Este segundo método tiene la ventaja de visitar una vez cada nodo no hoja para calcular de manera directa su valor *lcount* (y *rcount*). El primer método visita cada nodo que no es hoja una vez para cada una de sus hojas descendientes, agregando 1 a *lcount* cada vez que se encuentra que la hoja es un descendiente izquierdo. Para compensar esta ventaja, el segundo método requiere un campo extra *rcount*, mientras que el primero no requiere campos extra.

## Revisión del problema de Josephus

El problema de Josephus de la sección 4.5 es un ejemplo perfecto de la utilidad de la representación de una lista mediante un árbol binario. En ese problema fue necesario encontrar en forma repetida el  $m$ -ésimo elemento siguiente de una lista y luego eliminarlo. Esas son operaciones que se pueden ejecutar con eficacia en listas representadas por árboles.

Si *size* es igual al número de elementos presentes en una lista, la posición del  $m$ -ésimo nodo siguiente al nodo que se acaba de eliminar en la posición *k* se da por  $1 + (k - 2 + m) \% \text{size}$ . (Aquí se asume que el primer nodo de la lista está en la posi-

ción 1, no en la 0.) Por ejemplo, a una lista con cinco elementos se le elimina el tercero y se desea encontrar el cuarto después del elemento eliminado,  $size = 4$ ,  $k = 3$  y  $m = 4$ . Entonces  $k - 2 + m$  es igual a 5 y  $(k - 2 + m) \% size$  es 1, por lo que el cuarto elemento, a partir del elemento eliminado, está en la posición 2. (Después de eliminar el elemento 3, se cuentan los elementos 4, 5, 1 y 2). Se puede por lo tanto, escribir una función en C *follower* para encontrar el  $m$ -ésimo nodo que sigue a un nodo en la posición  $k$  que acaba de ser eliminado y asignar a  $k$  su posición. La rutina llama a la rutina *findelement* presentada con anterioridad.

```

follower(size, m, pk)
{
 int size, m, *pk;
 int j, d;
 j = k - 2 + m;
 *pk = (j % size) + 1;
 return(findelement(*pk));
} /* fin de follower */
El siguiente programa en C implanta el algoritmo de Josephus mediante un árbol como representación de lista. El programa recibe un número de personas en el círculo (n), un entero $count$ (m) y los nombres de las personas del círculo en orden, empezando con la persona a partir de la cual comienza el conteo. Las personas del círculo se cuentan en orden y la persona en la que se alcanza el conteo de entrada abandona el círculo. Luego se inicia el conteo otra vez a partir de 1, comenzando con la siguiente persona. El programa imprime el orden en el que las personas abandonan el círculo. En la sección 4.5 se presentó un programa para hacer esto por medio de una lista circular en la que se accesan $(n - 1) * m$ nodos una vez que se construye la lista inicial. El siguiente algoritmo accesa menos nodos que $(n - 1) * \log_2 n$ ya que se construyó el árbol.

main()
{
 int k, m, n, p, size;
 struct nodetype node[NUMNODES];
 scanf("%d%d", &n, &m);
 buildtree(n);
 k = n + 1; /* de inicio hemos "eliminado" a la (n+1)-ésima persona */
 for (size = n; size > 2; --size) {
 /* repetir hasta que sólo quede una persona */
 p = follower(size, m, &k);
 printf("%d\n", node[p].info);
 delete(p);
 }
}

```

```
 /* fin de for */
 printf("%d", node[0].info);
 /* fin de main */
```

## EJERCICIOS

- 5.4.1. Pruebe que un árbol estrictamente binario cuasi-completo se asigna el número  $2^n$  al nodo de la extrema izquierda en el nivel  $n$ .
  - 5.4.2. Pruebe que la extensión (ver ejercicio 5.3.5) de un árbol binario cuasi-completo es cuasi-completo.
  - 5.4.3. ¿Para qué valores de  $n$  y  $m$  es más rápida de ejecutar la solución al problema de Josephus proporcionada en esta sección que la de la sección 4.5? ¿Por qué?
  - 5.4.4. Explique cómo se puede eliminar la necesidad del campo *used* si se opta por no mover hacia arriba una hoja recién creada que no tenga hermano durante la eliminación.
  - 5.4.5. Explique cómo eliminar la necesidad del campo *used* si se hace *lcount* igual a  $-1$  en un nodo no hoja que se convierte a hoja y coloca blancos en *info* en un nodo eliminado.
  - 5.4.6. Escriba en C la rutina *buildtree*, en la cual cada nodo sea visitado una sola vez mediante un arreglo *rcount* como el descrito en el texto.
  - 5.4.7. Muestre cómo representar una lista ligada como un árbol binario cuasi-completo en el que cada elemento de la lista está representado por un nodo del árbol. Escribir una función en C que señale con un apuntador al elemento  $k$ -ésimo de una lista de este tipo.

## 5.5. ARBOLES Y SUS APLICACIONES

En esta sección se examinan los árboles generales y sus representaciones. También se investigan algunos de sus usos en la solución de problemas.

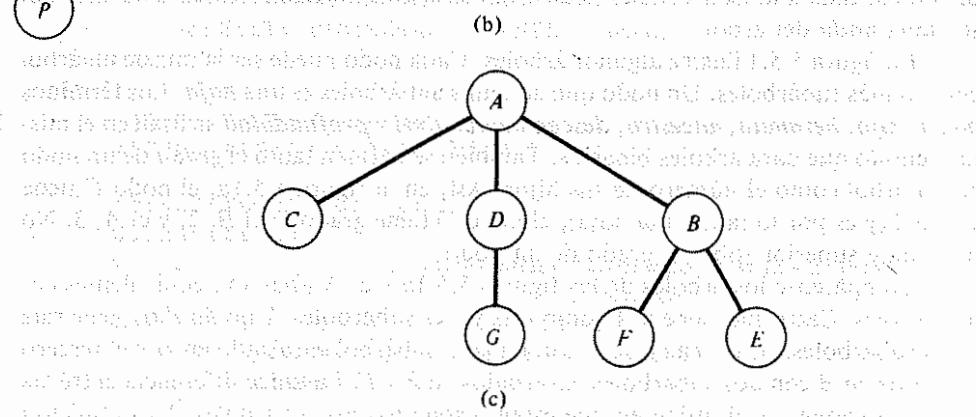
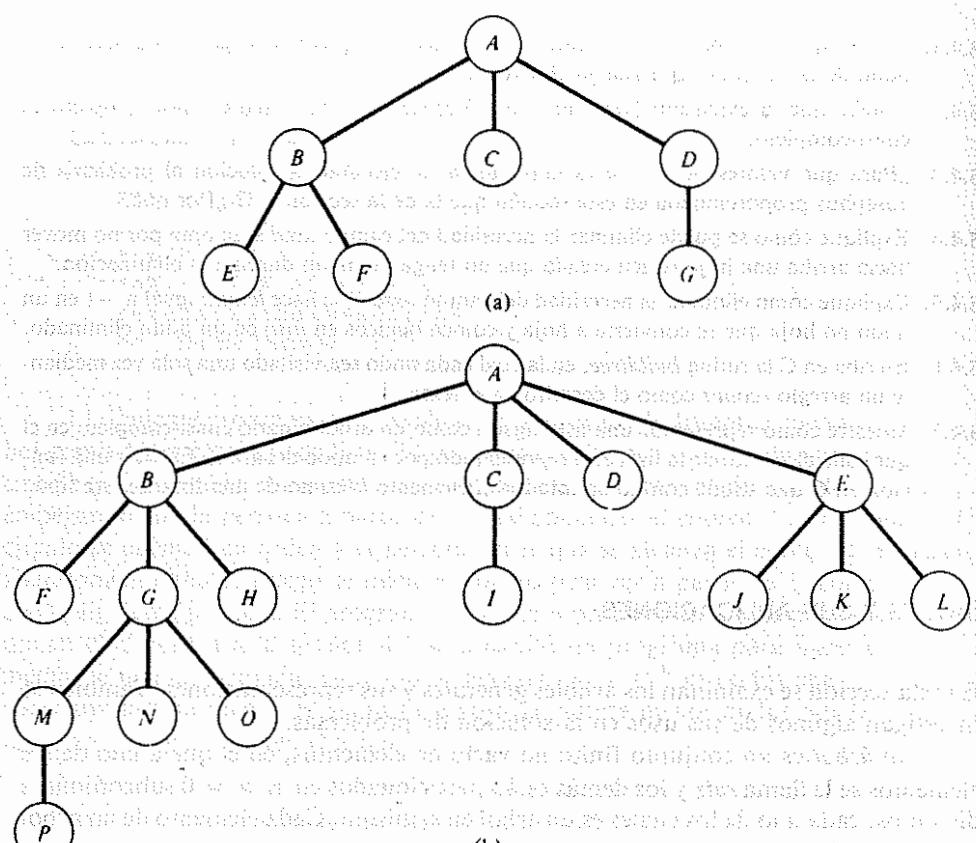
Un **árbol** es un conjunto finito no vacío de elementos, en el que a uno de los elementos se le llama *raíz* y los demás están particionados en  $m > 0$  subconjuntos disjuntos, cada uno de los cuales es un árbol en sí mismo. Cada elemento de un árbol se llama *nodo* del árbol.

La figura 5.5.1 ilustra algunos árboles. Cada nodo puede ser la raíz de un árbol con 0 o más subárboles. Un nodo que no tenga subárboles es una *hoja*. Los términos *padre*, *hijo*, *hermano*, *ancestro*, *descendiente*, *nivel* y *profundidad* se usan en el mismo sentido que para árboles binarios. También se definen tanto el *grado* de un nodo en un árbol como el número de sus hijos. Así, en la figura 5.5.1a, el nodo C tiene grado 0 (y es por lo tanto una hoja); el nodo D tiene grado 1; el B, 2, y el A, 3. No hay límite superior sobre el grado de un nodo.

Compárense los árboles de las figuras 5.5.1a y c. Ambos son equivalentes como árboles. Cada uno tiene a  $A$  como raíz y tres subárboles. Uno de ellos tiene raíz  $C$  sin subárboles, otro tiene raíz  $D$  con un solo subárbol enraizado en  $G$  y el tercero tiene raíz en  $B$  con dos subárboles enraizados en  $E$  y  $F$ . La única diferencia entre las dos ilustraciones es el orden en que están dispuestos los subárboles. La definición de un árbol no hace distinción entre los subárboles de un árbol general, a diferencia de

un árbol binario en el que se hace distinción entre sus subárboles izquierdo y derecho.

Un **árbol ordenado** se define como un árbol en el cual los subárboles de cada nodo forman un conjunto ordenado. En un árbol ordenado se puede hablar del primero, segundo o último hijo de un nodo particular. Al primer hijo de un nodo de un



**Figura 5.5.1** Ejemplos de árboles.

árbol ordenado se le llama con frecuencia el **hijo mayor** del nodo, y al último hijo, el **hijo menor**. Aunque los árboles de la figura 5.5.1a y c son equivalentes como árboles no ordenados, ambos son diferentes como árboles ordenados. En el resto de este capítulo se usa la palabra “árbol” para referirse a un “árbol ordenado”. Un **bosque** es un conjunto ordenado de árboles ordenados.

Surge la pregunta de si un árbol binario es un árbol. Todo árbol binario, excepto el vacío, es de hecho un árbol. Sin embargo no todo árbol es binario. Un nodo de un árbol puede tener más de dos hijos y un nodo de un árbol binario no. Incluso un árbol cuyos nodos tengan como máximo dos hijos no es por fuerza un árbol binario. Esto es porque un solo hijo de un árbol general no se designa como hijo “izquierdo” o “derecho”, mientras que en un árbol binario todo hijo tiene que ser un hijo “izquierdo” o “derecho”. En realidad, aunque un árbol binario no vacío es un árbol, las designaciones de izquierdo y derecho no tienen significado dentro del contexto de un árbol (excepto quizás para ordenar los dos subárboles de aquellos nodos con dos hijos). Un árbol binario no vacío es un árbol en donde cada uno de sus nodos tiene como máximo dos subárboles que tienen la designación agregada de “izquierdo” o “derecho”.

### Representaciones de árboles en C

¿Cómo puede representarse en C un árbol ordenado? De inmediato acuden a la mente dos posibilidades: se puede declarar un arreglo de nodos del árbol o se puede asignar una variable dinámica para cada nodo creado. Sin embargo, ¿cuál debe ser la estructura de cada nodo individual? En la representación de un árbol binario, cada nodo contiene un campo de información y dos apuntadores a sus dos hijos. Pero, ¿cuántos apuntadores debe contener el nodo de un árbol? El número de hijos de un nodo es variable y puede ser tan grande o pequeño como se desee. Si se declara de manera arbitraria

```

#define MAXSONS 20
struct treenode {
 int info;
 struct treenode * padre;
 struct treenode * hijos[MAXSONS];
};

```

se restringe entonces el número de hijos de un nodo a un máximo de 20. Aunque en muchos casos basta esto, a veces es necesario crear un nodo de manera dinámica con 21 o 100 hijos. Peor que esta posibilidad remota es el hecho de que se reserven veinte unidades de memoria aun cuando un nodo pueda tener en realidad 1 o 2 (o incluso 0) hijos. Esto es un tremendo despilfarro de espacio.

Otra posibilidad es ligar todos los hijos de un nodo en una lista lineal. Así, el conjunto de nodos disponibles (mediante la implantación con arreglo) puede declararse como sigue:

Arboles

```

#define MAXNODES 500

struct treenode {
 int info;
 int father;
 int son;
 int next;
};

struct treenode node[MAXNODES];

```

*node[p].son apunta al hijo mayor de node[p], y node[p].next apunta al siguiente hermano más joven de node[p].*

De manera alternativa, un nodo puede declararse como variable dinámica:

```

struct treenode { /* aquí resuelvo tres cosas: nombre, tipo y dirección de memoria */
 int info; /* no tiene que ser nro, pero es más intuitivo */
 struct treenode *father; /* dirección de dirección, porque es apuntador */
 struct treenode *son; /* "heredero" o "descendiente" */
 struct treenode *next;
};

typedef struct treenode *NODEPTR;

```

Si todos los recorridos son de un nodo a sus hijos, puede omitirse el campo *father*. La figura 5.5.2 ilustra las representaciones de los árboles de la figura 5.5.1 en estos métodos cuando no se necesita el campo *father*.

Incluso cuando es necesario accesar el padre de un nodo, puede omitirse el campo *father* colocando un apuntador al padre en el campo *next* del hijo más joven, en lugar de dejarlo como *null*. Puede usarse entonces un campo lógico adicional para indicar si el campo *next* apunta a un hijo “real” o al padre. De manera alternativa (en la implantación con arreglo de nodos), los contenidos del campo *next* pueden tener índices tanto negativos como positivos. Un valor negativo indicaría que el campo *next* apunta al padre del nodo y no a su hermano, y el valor absoluto del campo *next* produce el apuntador real. Esto es similar a la representación de hebras en árboles binarios. Por supuesto, en cada uno de estos dos últimos métodos, se requerirá recorrer la lista de los hijos hacia el más joven, en un nodo dado, para accesar al padre del nodo.

Si se considera que *son* está en correspondencia con el apuntador *left* de un nodo de un árbol binario y que *next* está en correspondencia con su apuntador *right*, este método representa en realidad un árbol ordenado general mediante un árbol binario. Se puede dibujar este árbol binario como el árbol original con una inclinación de 45 grados y con todas las ligas padre-hijo eliminadas, excepto aquellas entre un nodo y su hijo mayor y con las ligas agregadas entre cada nodo y su hermano menor próximo. La figura 5.5.3 ilustra los árboles binarios que corresponden a los árboles de la figura 5.5.1.

En realidad, un árbol binario puede usarse para representar un bosque entero, pues el apuntador *next* de la raíz de un árbol puede usarse para apuntar al siguiente árbol del bosque. La figura 5.5.4 ilustra un bosque y su árbol binario correspondiente.

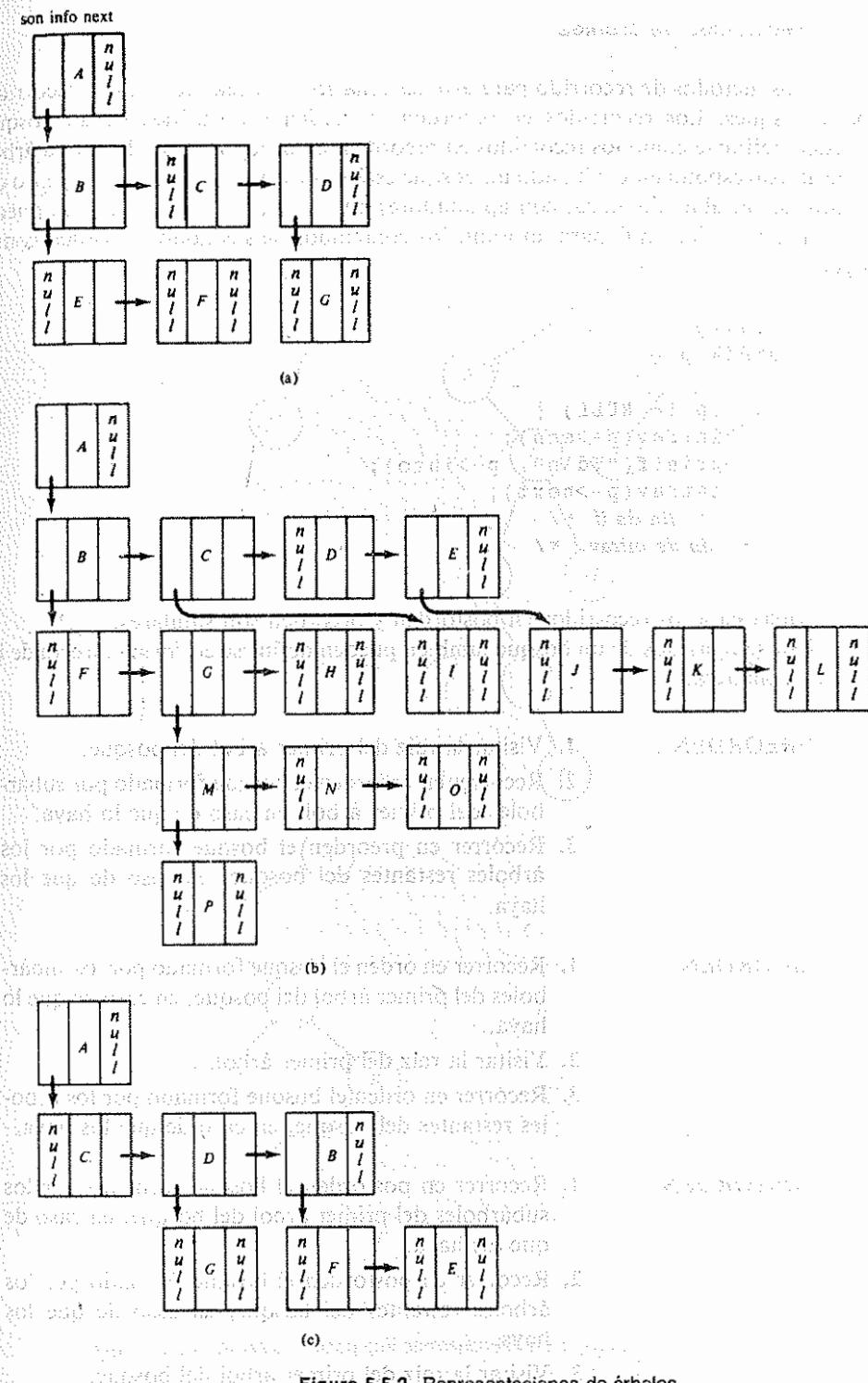


Figura 5.5.2 Representaciones de árboles.

## Recorridos de árboles

Los métodos de recorrido para árboles binarios inducen métodos de recorrido para bosques. Los recorridos en preorden, en orden y postorden de un bosque pueden definirse como los recorridos en preorden, en orden y postorden de su árbol binario correspondiente. Cuando un bosque está representado como un conjunto de nodos de variable dinámica, con apunadores *next* y *son* igual que antes, se puede escribir una rutina en C para imprimir los contenidos de sus nodos en orden como sigue:

```
intrav(p)
NODEPTR p;
{
 if (p != NULL) {
 intrav(p->son);
 printf("%d\n", p->info);
 intrav(p->next);
 } /* fin de if */
} /* fin de intrav */
```

Las rutinas para los recorridos en postorden y preorden son similares.

Estos recorridos de un bosque también pueden definirse en forma directa de la siguiente manera:

### PREORDEN

1. Visitar la raíz del primer árbol del bosque.
2. Recorrer en preorden el bosque formado por subárboles del primer árbol, en caso de que lo haya.
3. Recorrer en preorden el bosque formado por los árboles restantes del bosque, en caso de que los haya.

### EN ORDEN

1. Recorrer en orden el bosque formado por los subárboles del primer árbol del bosque, en caso de que lo haya.
2. Visitar la raíz del primer árbol.
3. Recorrer en orden el bosque formado por los árboles restantes del bosque, en caso de que los haya.

### POSTORDEN

1. Recorrer en postorden el bosque formado por los subárboles del primer árbol del bosque, en caso de que los haya.
2. Recorrer en postorden el bosque formado por los árboles restantes del bosque, en caso de que los haya.
3. Visitar la raíz del primer árbol del bosque.

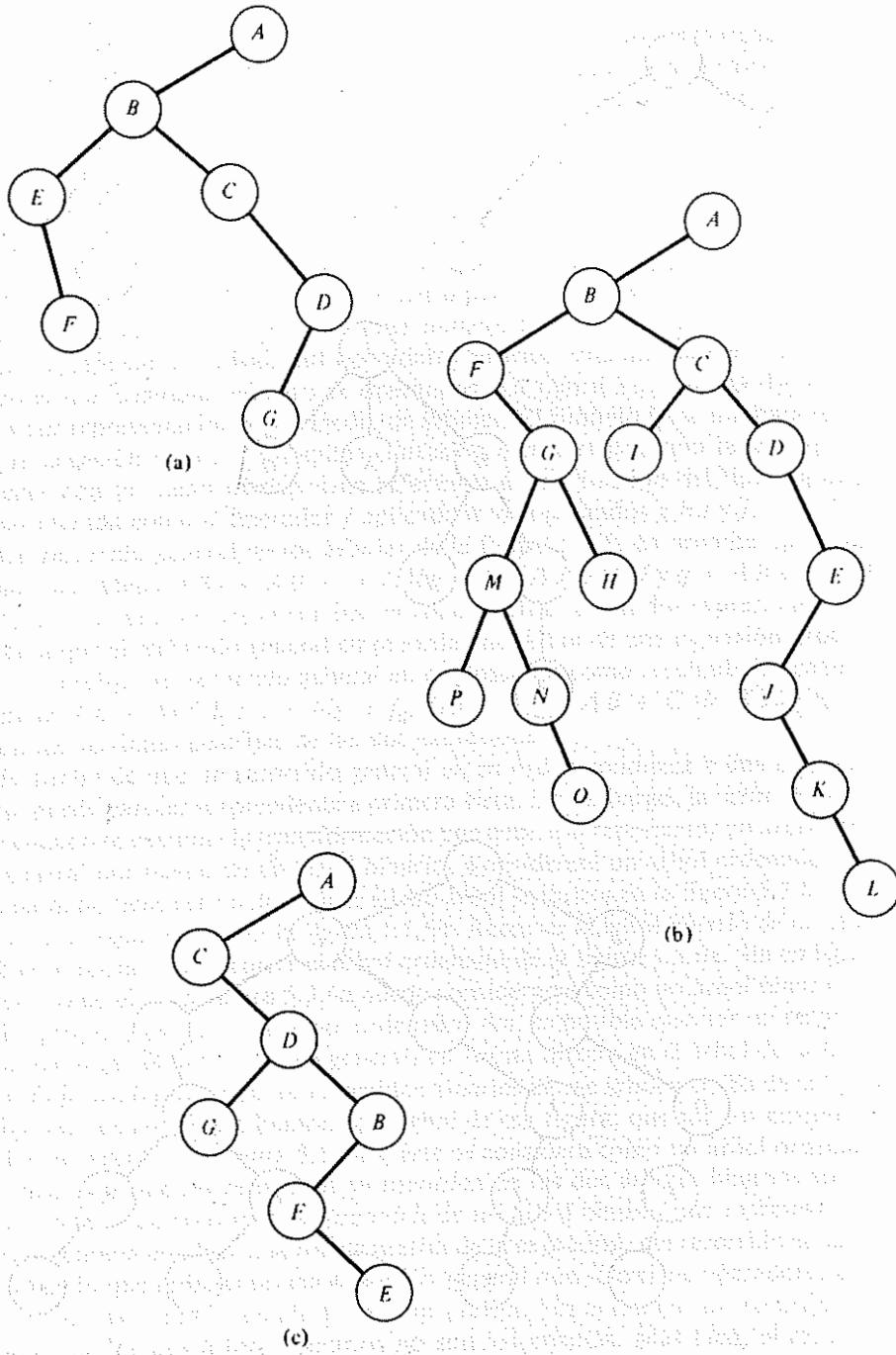


Figura 5.5.3 Árboles binarios que corresponden a los árboles de la figura 5.5.1.

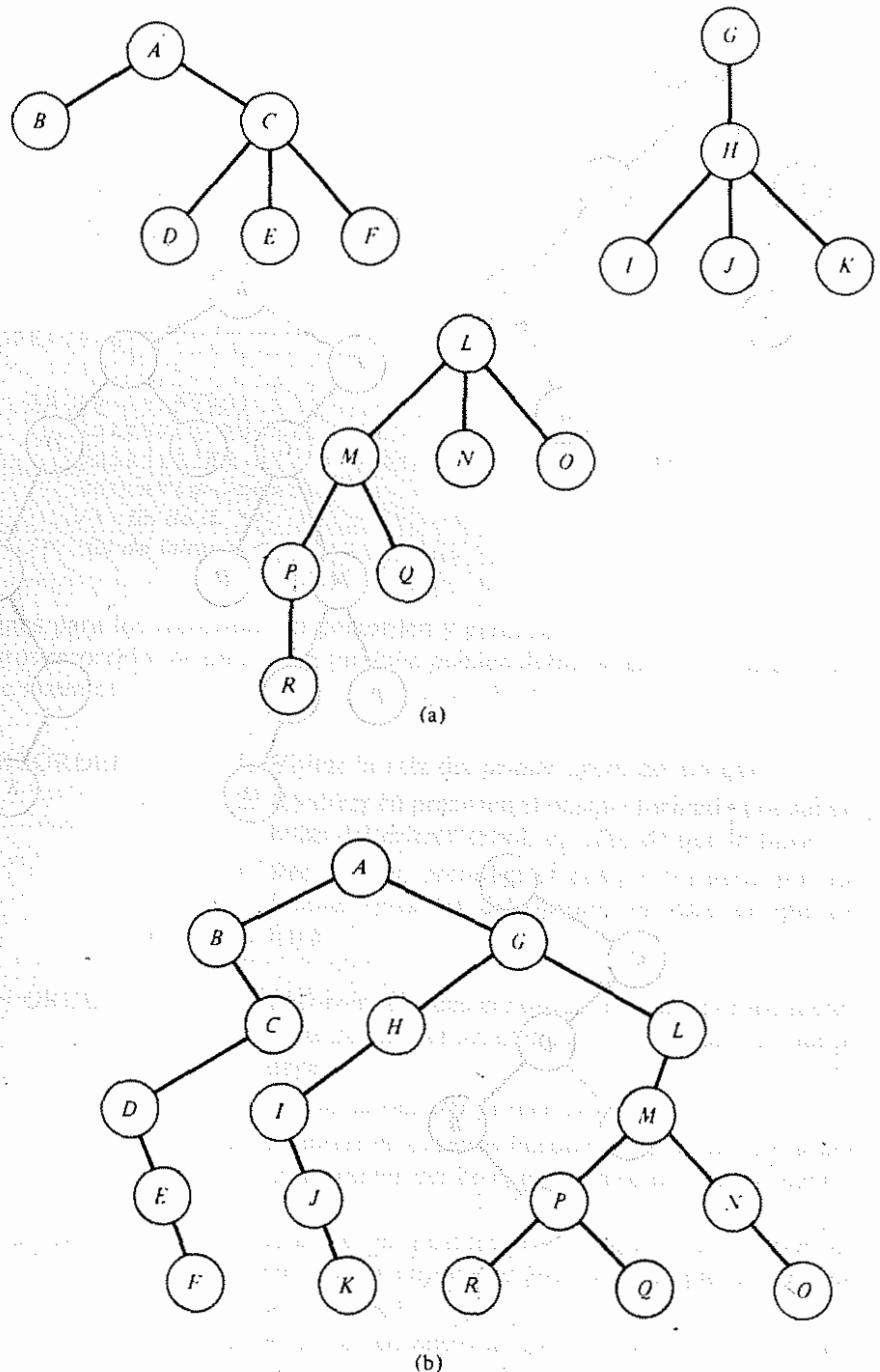


Figura 5.5.4 Un bosque y su árbol binario correspondiente.

Los nodos del bosque de la figura 5.5.4a pueden agruparse en preorden como *ABCDEFGHIJKLMPRQNO*, en orden como *BDEFCAIJKHGRPQMNOL* y en postorden como *FEDCBKJIHRQPONMLGA*. Denomínese a un recorrido de un árbol binario como *recorrido binario* y a un recorrido de un árbol general ordenado como *recorrido general*.

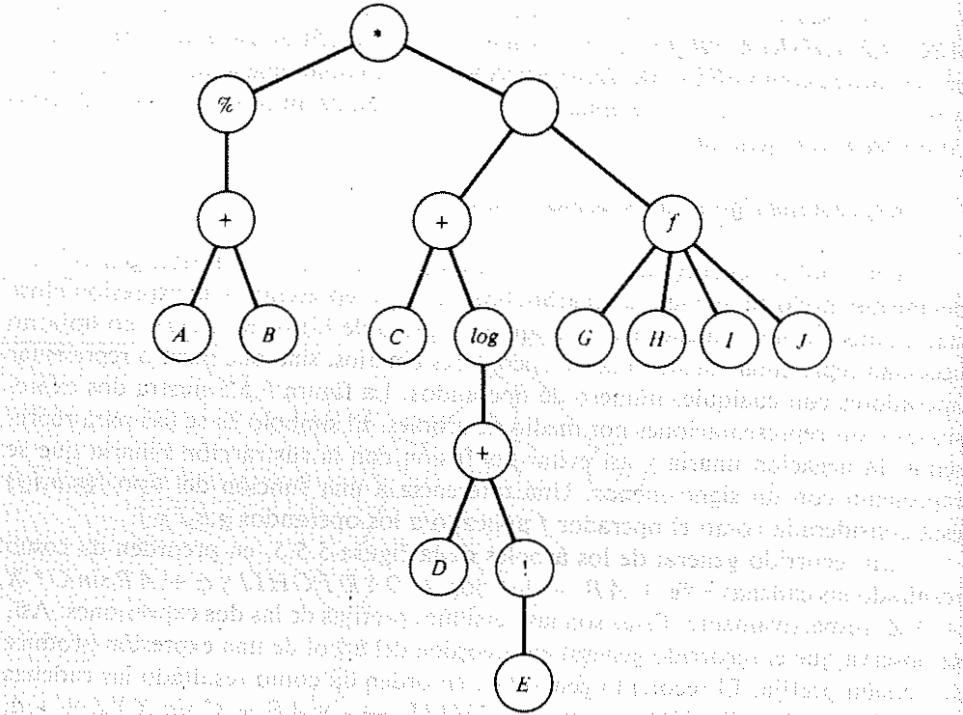
### Expresiones generales como árboles

Un árbol ordenado puede usarse para representar una expresión general casi del mismo modo en que se usa un árbol binario para representar una expresión binaria. Como un nodo puede tener cualquier número de hijos, los nodos no hoja no necesitan representar exclusivamente operadores binarios, sino que pueden representar operadores con cualquier número de operandos. La figura 5.5.5 ilustra dos expresiones y sus representaciones por medio de árboles. El símbolo  $\%$  se usa para representar la negación unaria y así evitar confusión con la sustracción binaria que se representa con un signo menos. Una referencia a una función del tipo  $f(g,h,i,j)$  está considerada como el operador  $f$  aplicado a los operandos  $g, h, i$  y  $j$ .

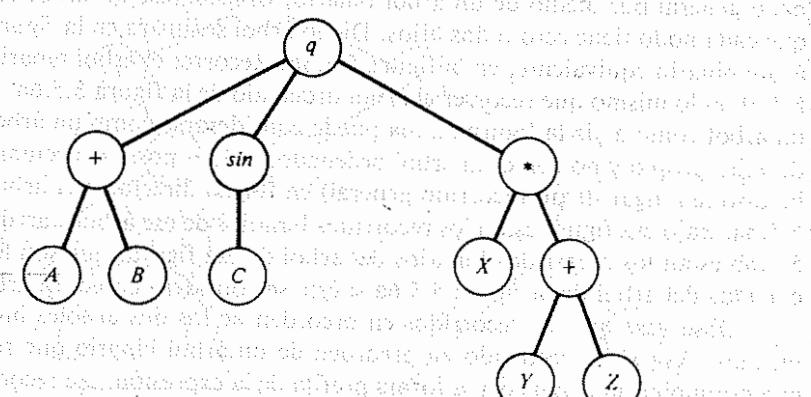
Un recorrido general de los árboles de la figura 5.5.5, en preorden da como resultado las cadenas  $* \% + AB - + C \log + D ! E FGHIJ$  y  $q + AB \sin C * X + Y Z$ , respectivamente. Estas son las versiones prefijas de las dos expresiones. Así, se observa que el recorrido general en preorden del árbol de una expresión produce su versión prefija. El recorrido general en orden da como resultado las cadenas respectivas  $AB + \% CDE ! + \log + GHIJF - * y AB + C \sin XYZ + * q$ , que son las versiones postfijas de las dos expresiones.

El hecho de que un recorrido general en orden conduzca a una expresión postfija puede parecer sorprendente a primera vista. Sin embargo, la razón de ello se aclara cuando se examina la transformación que ocurre al representar un árbol ordenado general por medio de un árbol binario. Considérese un árbol ordenado en el que cada nodo tiene cero o dos hijos. Dicho árbol se ilustra en la figura 5.5.6a, y su árbol binario equivalente, en la figura 5.5.6b. Recorrer el árbol binario de la figura 5.5.6b es lo mismo que recorrer el árbol ordenado de la figura 5.5.6a. Sin embargo, un árbol como el de la figura 5.5.6a puede considerarse como un árbol binario por derecho propio y no como un árbol ordenado. Así, es posible ejecutar un recorrido binario (en lugar de un recorrido general) en forma directa en el árbol de la figura 5.5.6a. Bajo esa figura están los recorridos binarios de ese árbol y arriba de la figura 5.5.6b están los recorridos binarios del árbol de esa figura, que son los mismos recorridos del árbol de la figura 5.5.6a si éste se considera como un árbol ordenado.

Obsérvese que los recorridos en preorden de los dos árboles binarios son los mismos. Así, si un recorrido en preorden de un árbol binario que representa una expresión binaria conduce a la forma prefija de la expresión, ese recorrido en un árbol ordenado que representa una expresión general que sólo tiene operadores binarios también da como resultado la versión prefija. Sin embargo, los recorridos en postorden de los dos árboles binarios no son los mismos. Más bien, el recorrido binario en orden del segundo (que es el mismo recorrido general en orden del primero si éste se considera un árbol ordenado) es igual al recorrido binario en postorden del primero. Así, el recorrido general en orden de un árbol ordenado que representa una



(a)  $-(A + B) * (C + \log(D + E!)) - f(G, H, I, J)$



(b)  $q(A + B, \sin(C), X * Y + Z)$

**Figura 5.5.5** Representación por medio de un árbol de una expresión aritmética.

expresión binaria es equivalente al recorrido binario en postorden del árbol binario que representa a esa expresión, y produce la versión postfixa.

### Evaluación del árbol de una expresión

Supóngase que se desea evaluar una expresión cuyos operandos son todos constantes numéricas. Tal expresión puede representarse en C con un árbol cada uno de cuyos nodos se declara mediante:

```
#define OPERATOR 0
#define OPERAND 1
struct treenode {
 short int utype; /* OPERATOR u OPERAND */
 union {
 char operator[10];
 float val;
 } info;
 struct treenode *son;
 struct treenode *next;
};
typedef treenode *NODEPTR;
```

Como ya se ilustró antes, los apuntadores *son* y *next* se usan para ligar los nodos de un árbol. Como un nodo contiene información que puede ser un número (operando) o una cadena de caracteres (operador), la porción de información del nodo es una componente unión de la estructura.

Se desea escribir una función en C, *evaltree(p)*, que acepte un apuntador a un árbol de ese tipo y dé como resultado el valor de la expresión representada por el árbol. La rutina *evalbintree* presentada en la sección 5.2 ejecuta una función similar para expresiones binarias. *evalbintree* utiliza una función *oper* que acepta un símbolo de operador y dos operandos numéricos y regresa el resultado numérico de la aplicación del operador a los dos operandos. Sin embargo, en el caso de una expresión general no puede usarse una función de este tipo, ya que el número de operandos (y por consiguiente el número de argumentos) varía con el operador. Por lo tanto, se introduce una nueva función *apply(p)*, la que acepta un apuntador al árbol de una expresión que contenga un solo operador y sus operandos numéricos y regresa el resultado de aplicar dicho operador a sus operandos. Por ejemplo, el resultado de llamar a la función *apply(p)*, con parámetro *p* apuntando al árbol de la figura 5.5.7 es 24. Si la raíz del árbol que se transfiere a *evaltree* representa un operador, cada uno de sus subárboles se remplaza por nodos del árbol que representan los resultados numéricos de su evaluación para que la función *apply* pueda ser llamada. Al evaluar la expresión, los nodos del árbol que representan operandos se liberan y los nodos operador se convierten en nodos operando.

En seguida se presenta un procedimiento recursivo *replace* que acepta un apuntador al árbol de una expresión y remplaza el árbol por el nodo de un árbol que contiene el resultado numérico de la evaluación de dicha expresión.

Algunas de las estrategias más comunes para evaluar una expresión en notación polaca inversa son:

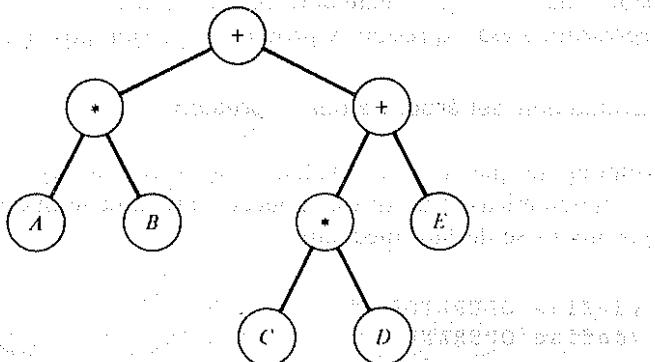


Diagrama (a)

Preorden:  $+ * AB + * CDE$   
En orden:  $A * B + C * D + E$   
Postorden:  $AB * CD * E + +$

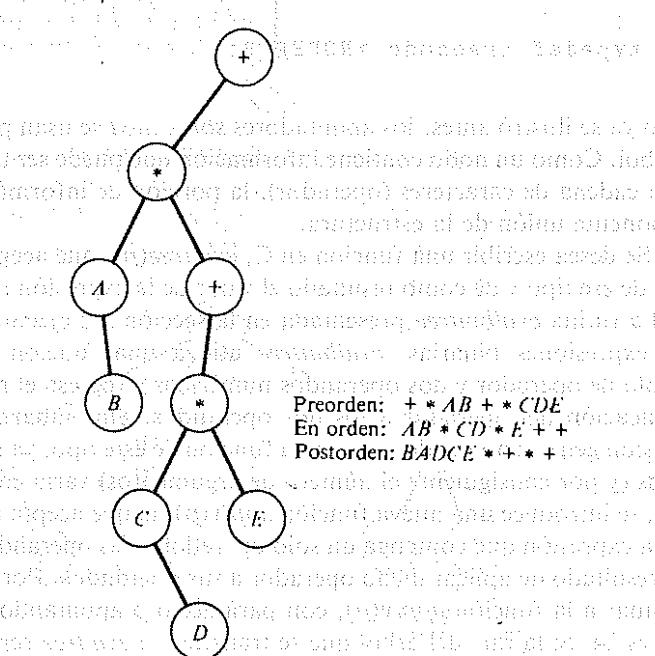


Diagrama (b)

Preorden:  $+ * AB + * CDE$   
En orden:  $AB * CD * E + +$   
Postorden:  $BADCE * + * +$

hijo      próximo      etiqueta      operador/val

| hijo | próximo | etiqueta | operador/val |
|------|---------|----------|--------------|
| nulo |         | oprtr    | *            |

hijo      próximo      etiqueta      operador/val

| hijo | próximo | etiqueta | operador/val |
|------|---------|----------|--------------|
| nulo |         | opnd     | 4            |

hijo      próximo      etiqueta      operador/val

| hijo | próximo | etiqueta | operador/val |
|------|---------|----------|--------------|
| nulo | nulo    | opnd     | 6            |

Figura 5.5.7 Árbol de una expresión.

```

replace(p)
NODEPTR p;
{
 float value;
 NODEPTR q, r;
 if (p->utype == OPERATOR) { /* como el operador es el tipo de dato */
 /* el árbol tiene un operador */ /* se aplica la operación */
 /* como su raíz */ /* los resultados se devuelven */
 q = p->son; /* el resultado se reemplaza en el nodo */
 while (q != NULL) {
 /* reemplazar cada uno de los subárboles */
 /* con operandos */
 replace(q);
 q = q->next;
 } /* fin de while */
 /* aplicar el operador de la raíz a los */
 /* operandos de los subárboles */
 value = apply(p);
 /* reemplace el operador por el resultado */
 p->utype = OPERAND;
 p->val = value;
 /* liberar todos los subárboles */
 q = p->son;
 p->son = NULL;
 }
}

```

```

 while (q != NULL) {
 r = q;
 q = q->next;
 free(r);
 } /* fin de while */
} /* fin de if */
} /* fin de replace */

```

La función *evaltree* puede escribirse ahora como sigue:

```

float evaltree(p)
NODEPTR p;
{
 NODEPTR q;
 replace(p);
 return(p->val);
 free(p);
} /* fin de evaltree */

```

Después de llamar a *evaltree(p)*, se destruye el árbol y el valor de *p* deja de tener significado. Este es el caso de un *apuntador indefinido*, en el que una variable apuntador contiene la dirección de una variable liberada. Los programadores de C que usan variables dinámicas deben ser cuidadosos para reconocer tales apuntadores y no usarlos posteriormente.

### Construcción de un árbol

En la construcción de un árbol se usan con frecuencia varias operaciones. Enseguida se presentan algunas de ellas y sus implantaciones en C. En la representación en C, se asume que los apuntadores *father* no se necesitan, por lo que no se usa el campo *father* y el apuntador *next* del nodo más joven es *null*. Si éste no fuese el caso, las rutinas serían un poco más complejas y menos eficaces.

La primera operación que se analiza es *setsons*. Esta operación acepta un apuntador al nodo de un árbol que no tenga hijos y una lista lineal de nodos ligados a través del campo *next*. *setsons* establece los nodos en la lista como hijos del nodo del árbol. La rutina C para implantar esta operación es directa (se usa la implantación de memoria dinámica):

```

setsons(p, list)
NODEPTR p, list;
{
 /* p señala un nodo del árbol; lista de hijos */
 /* a una lista de nodos ligados por medio */
 /* de sus campos next */
 if (p == NULL) {
 printf("inserción no válida\n");
 exit(1);
 }
 /* fin de if */
}

```

```

} /* fin de if */
if (p->son != NULL) {
 printf("inserción no válida\n");
 exit(1);
} /* fin de if */
p->son = list;
/* fin de setsons */

```

Otra operación común es *addson(p,x)*, en la que *p* apunta al nodo de un árbol y se desea agregar un nodo que contenga *x* como el hijo más joven de *node(p)*. La rutina en C para implantar *addson* se presenta a continuación. La rutina llama a la función auxiliar *getnode*, que asigna un nodo y regresa un apuntador al mismo.

```

addson(p, x)
NODEPTR p;
int x;
{
 NODEPTR q; /* apuntador */
 if (p == NULL) {
 printf("no se puede realizar la inserción\n");
 exit(1);
 } /* fin de if */
 /* el apuntador q recorre la lista de hijos de p */
 /* q es un nodo detrás de p */
 r = p->son;
 while (q != NULL) {
 r = q;
 q = q->next;
 } /* fin de while */
 /* En este punto, r señala al hijo más joven de p,
 o es nulo si p no tiene hijos */
 q = getnode();
 q->info = x;
 q->next = NULL;
 if (r == NULL) /* * p no tiene hijos */
 p->son = q;
 else
 r->next = q;
} /* fin de addson */

```

Obsérvese que para agregar un nuevo hijo a un nodo, se tiene que recorrer la lista de hijos existentes. Como agregar un hijo es una operación común, con frecuencia se usa una representación que realice esta operación con más eficacia. En esta representación alternativa, la lista de hijos se ordena del menor al mayor y no a la inversa. Así, *son(p)* apunta al hijo menor de *node(p)* y *next(p)* a su siguiente hermano mayor que él. En esta representación, la rutina *addson* puede escribirse como sigue:

```

addson(p, x)
NODEPTR p;
int x;
{
 NODEPTR q;

 if (p == NULL) {
 printf("inserción no válida \n");
 exit(1);
 } /* fin de if */
 if ((q = getnode()) == NULL) {
 q->info = x; // si el espacio es nulo, se asigna el espacio que se ha liberado
 q->next = p->son; // q ahora es el hijo de p
 p->son = q;
 } /* fin de addson */
}

```

## EJERCICIOS

- 5.5.1. ¿Cuántos árboles con  $n$  nodos existen?
- 5.5.2. ¿Cuántos árboles con  $n$  nodos y nivel máximo  $m$  existen?
- 5.5.3. Pruebe que si se apartan en cada nodo de un árbol general  $m$  campos apuntadores para apuntar a un máximo de  $m$  hijos y si el número de nodos del árbol es  $n$ , el número de campos apuntadores a hijos que son nulos es  $n * (m - 1) + 1$ .
- 5.5.4. Si se representa un bosque por medio de un árbol binario como en el texto, mostrar que el número de ligas derechas nulas es 1 más que el número de nodos no hojas del bosque.
- 5.5.5. Defina el *orden breadth-first* (primero por amplitud) de los nodos de un árbol general como la raíz seguida por todos los nodos del nivel 1, seguidos de todos los nodos del nivel 2 y así sucesivamente. En cada nivel deben ordenarse los nodos de manera que los hijos del mismo padre aparezcan en el mismo orden en que aparecen en el árbol, y si  $n_1$  y  $n_2$  tienen padres diferentes,  $n_1$  aparece antes que  $n_2$  si el padre de  $n_1$  aparece antes que el padre de  $n_2$ . Extender la definición a un bosque. Escriba un programa en C que recorra un bosque representado como un árbol binario en orden primero por amplitud.
- 5.5.6. Considere el siguiente método de transformación de un árbol general,  $gt$ , en un árbol estrictamente binario,  $bt$ . Cada nodo de  $gt$  está representado por una hoja de  $bt$ . Si  $gt$  consta de un solo nodo,  $bt$  consta de un solo nodo. En caso contrario  $bt$  consta de un nuevo nodo raíz, un subárbol izquierdo  $lt$  y un subárbol derecho  $rt$ .  $lt$  es el árbol estrictamente binario formado de manera recursiva a partir del subárbol mayor de  $gt$ , y  $rt$  es el árbol estrictamente binario formado de manera recursiva a partir de  $gt$  sin su subárbol mayor. Escribir una rutina en C para convertir un árbol general en un árbol estrictamente binario.
- 5.5.7. Escriba en C una función *compute*, que acepte un apuntador a un árbol que represente una expresión con operandos constantes y dé el resultado de evaluar la expresión sin destruir el árbol.
- 5.5.8. Escriba un programa en C para convertir una expresión infija a una expresión de árbol. Supóngase que todos los operadores no binarios preceden a sus operandos. Sea entonces la representación de la expresión de entrada como sigue: un operando está representado por el carácter 'N' seguido por un número, un operador por el carácter 'T' seguido

por un carácter que representa al operador, y una función por el carácter 'F' seguido por el nombre de la función.

- 5.5.9. Considere las definiciones de expresión, término y factor determinadas al final de la sección 3.2. Asignada una cadena de letras, signos más, asteriscos y paréntesis que formen una expresión válida, se puede formar un *árbol de sintaxis* para la cadena. Tal árbol se ilustra en la figura 5.5.8 para la cadena "(A + B) \* (C + D)". Cada nodo de un árbol de ese tipo representa una subcadena y contiene una letra (E para denotar expresión, T para término, F para factor o S para símbolo) y dos enteros. El primero es la posición dentro de la cadena de entrada donde comienza la subcadena representada por ese nodo; el segundo es la longitud de la subcadena. (La subcadena representada por cada nodo se muestra debajo del nodo de la figura.) Las hojas son todas nodos S y representan símbolos simples de la entrada original. La raíz del árbol debe ser un nodo E. Los hijos de cualquier nodo N que no sea S representan las subcademas que componen el objeto gramatical representado por N.

Escribir una rutina en C que acepte una cadena como ésta y construya un árbol de sintaxis para la misma.

## 5.6. UN EJEMPLO: ARBOLES DE JUEGO

Una aplicación de los árboles se encuentra en los juegos y un participante es la computadora. Esta aplicación se ilustra escribiendo un programa en C para determinar el "mejor" movimiento en el juego del "gato", a partir de una posición determinada del tablero.

Supóngase que hay una función *evaluate* que acepta una posición del tablero y una indicación de un jugador (X o O) y da como resultado el valor numérico que representa cuán "buena" parece ser la posición para ese jugador (cuanto más grande sea el valor que dé como resultado *evaluate*, mejor será la posición). Por supuesto, una posición ganadora tendrá el valor más grande posible y una posición perdedora el menor valor posible. Un ejemplo de una función de evaluación como ésa para el "gato" es el número de renglones, columnas y diagonales restantes abiertas para un jugador menos el número de las mismas para su oponente (excepto que el valor 9 sería el resultado para la posición que gana y -9 para la que pierde). Esta función no "prevé" todas las posibles posiciones del tablero que podrían resultar de la posición real; sólo evalúa una posición estática del mismo.

Dada una posición del tablero, el mejor movimiento siguiente está determinado por la consideración de todos los movimientos posibles y las posiciones resultantes. El movimiento seleccionado será aquel que resulte en la posición del tablero con mayor evaluación. Tal análisis, sin embargo, no conduce por fuerza al mejor movimiento. La figura 5.6.1 ilustra una posición y los cinco posibles movimientos que puede hacer X desde la misma. Aplicando la función de evaluación que se acaba de describir a las cinco posiciones resultantes, se llega a los valores mostrados. Cuatro movimientos producen la misma evaluación máxima, aunque tres de ellos son sin duda inferiores al cuarto. (La cuarta posición produce una victoria segura para X, mientras que los otros tres pueden empatar con O). En realidad, el movimiento que produce la evaluación menor es tan bueno o mejor que los movimientos que producen una evaluación más alta. La función de evaluación estática, por consiguiente, no

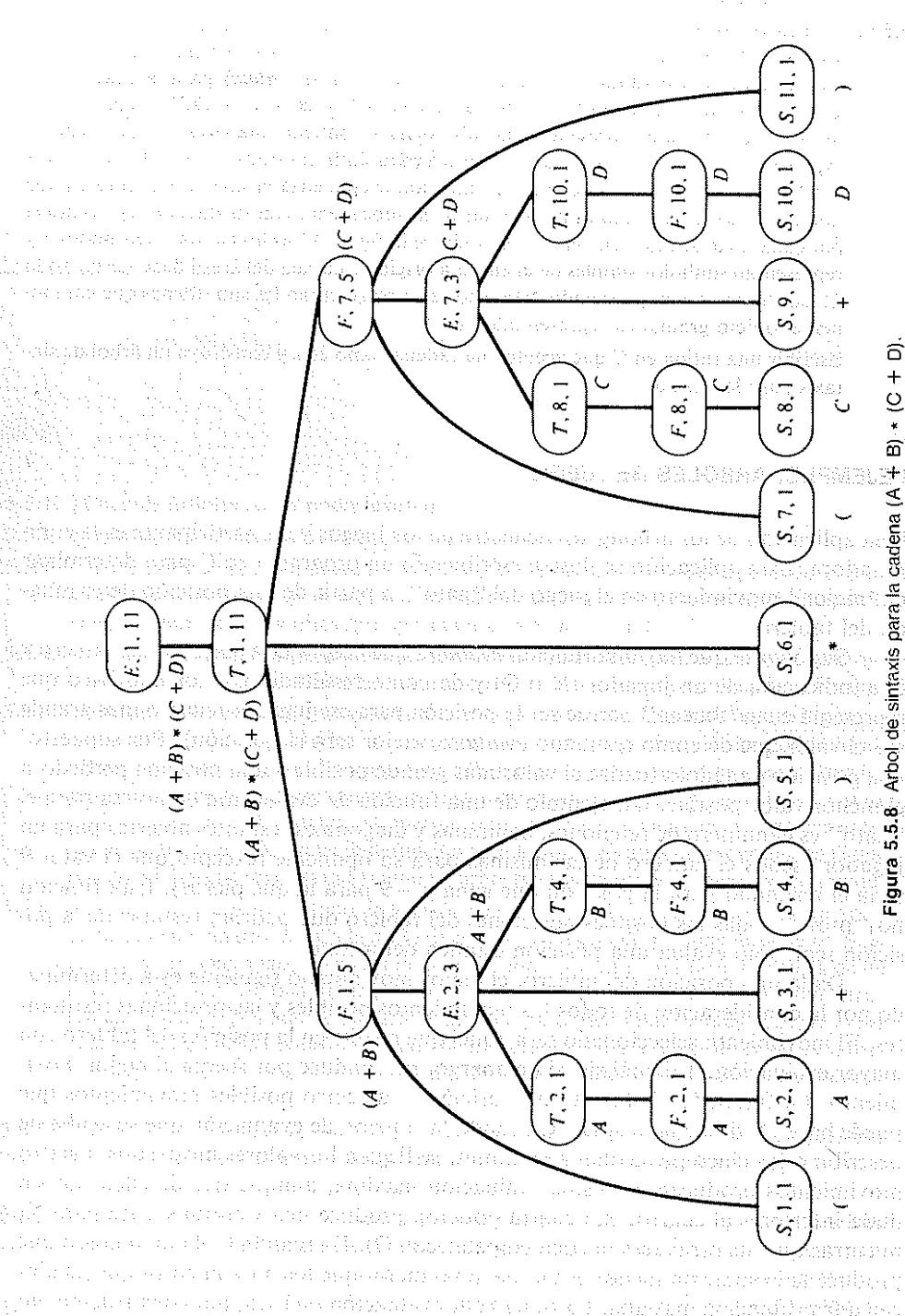


Figura 5.5.8 Árbol de sintaxis para la cadena  $(A + B) * (C + D)$ .

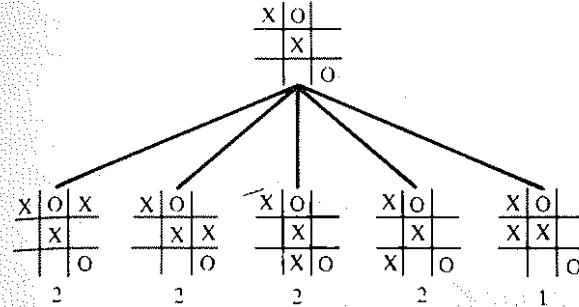


Figura 5.6.1

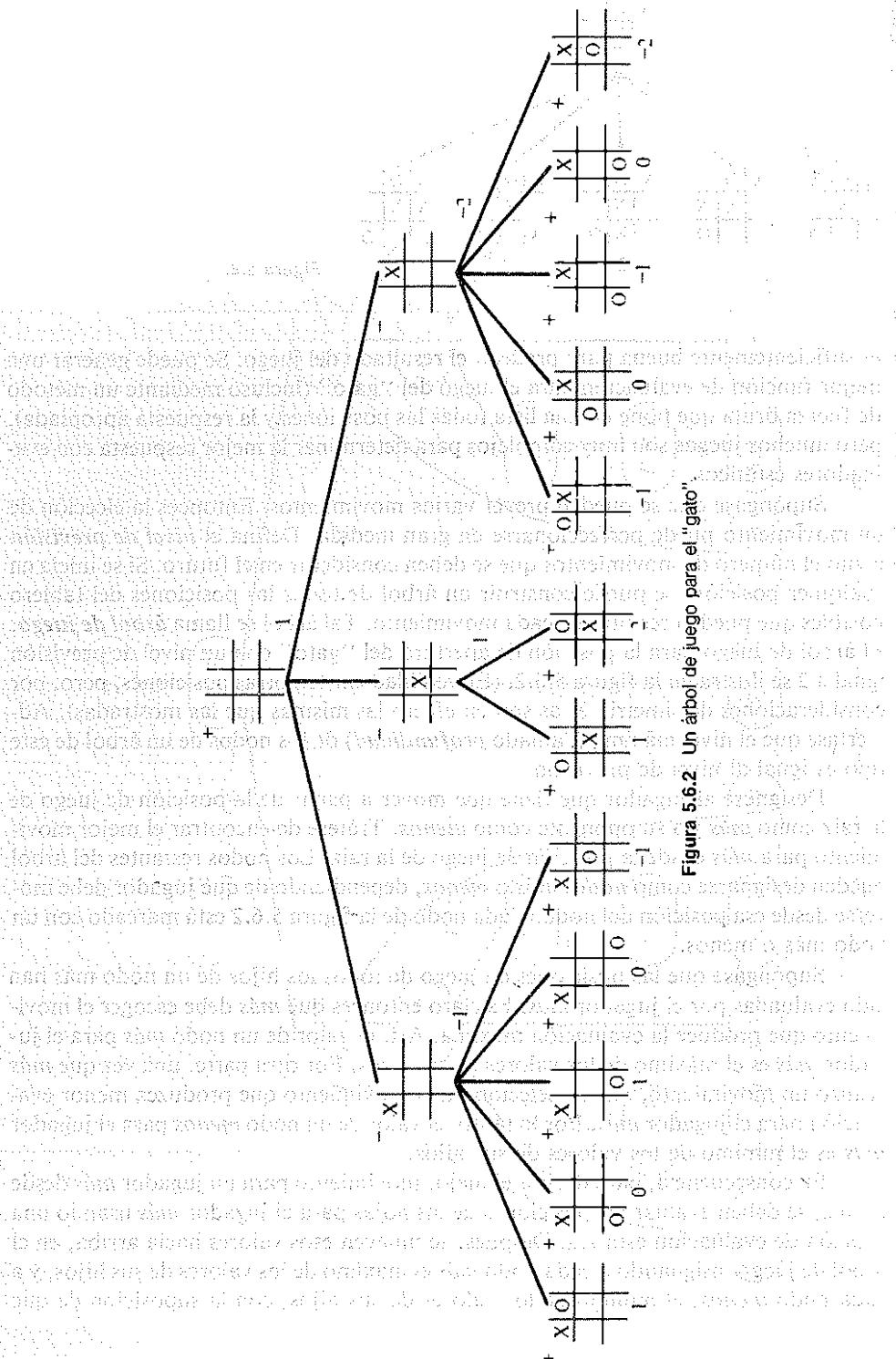
es suficientemente buena para predecir el resultado del juego. Se puede generar una mejor función de evaluación para el juego del "gato" (incluso mediante un método de fuerza bruta que pone en una lista todas las posiciones y la respuesta apropiada), pero muchos juegos son muy complejos para determinar la mejor respuesta con evaluadores estáticos.

Supóngase que se pueden prever varios movimientos. Entonces la elección de un movimiento puede perfeccionarse en gran medida. Defina el *nivel de previsión* como el número de movimientos que se deben considerar en el futuro. Si se inicia en cualquier posición, se puede construir un árbol de todas las posiciones del tablero posibles que pueden resultar de cada movimiento. Tal árbol se llama *árbol de juego*. El árbol de juego para la posición de apertura del "gato" con un nivel de previsión igual a 2 se ilustra en la figura 5.6.2. (En realidad existen otras posiciones, pero, por consideraciones de simetría éstas son en efecto las mismas que las mostradas). Adviéntase que el nivel máximo (llamado *profundidad*) de los nodos de un árbol de este tipo es igual al nivel de previsión.

Designese al jugador que tiene que mover a partir de la posición de juego de la raíz como *más* y a su oponente como *menos*. Trátese de encontrar el mejor movimiento para *más* desde la posición de juego de la raíz. Los nodos restantes del árbol pueden designarse como *nodos más* o *menos*, dependiendo de qué jugador debe moverse desde esa posición del nodo. Cada nodo de la figura 5.6.2 está marcado con un nodo más o menos.

Supóngase que las posiciones de juego de todos los hijos de un nodo más han sido evaluadas por el jugador *más*. Es claro entonces que *más* debe escoger el movimiento que produce la evaluación máxima. Así, el valor de un nodo *más* para el jugador *más* es el máximo de los valores de sus hijos. Por otra parte, una vez que *más* realizó un movimiento, *menos* seleccionará el movimiento que produzca menor evaluación para el jugador *más*. Por lo tanto, el valor de un nodo *menos* para el jugador *más* es el mínimo de los valores de sus hijos.

En consecuencia, para decidir el mejor movimiento para un jugador *más* desde la raíz, se deben evaluar las posiciones de las hojas para el jugador *más* usando una función de evaluación estática. Después, se mueven esos valores hacia arriba, en el árbol de juego, asignando a cada nodo *más* el máximo de los valores de sus hijos, y a cada nodo *menos*, el mínimo de los valores de sus hijos, con la suposición de que



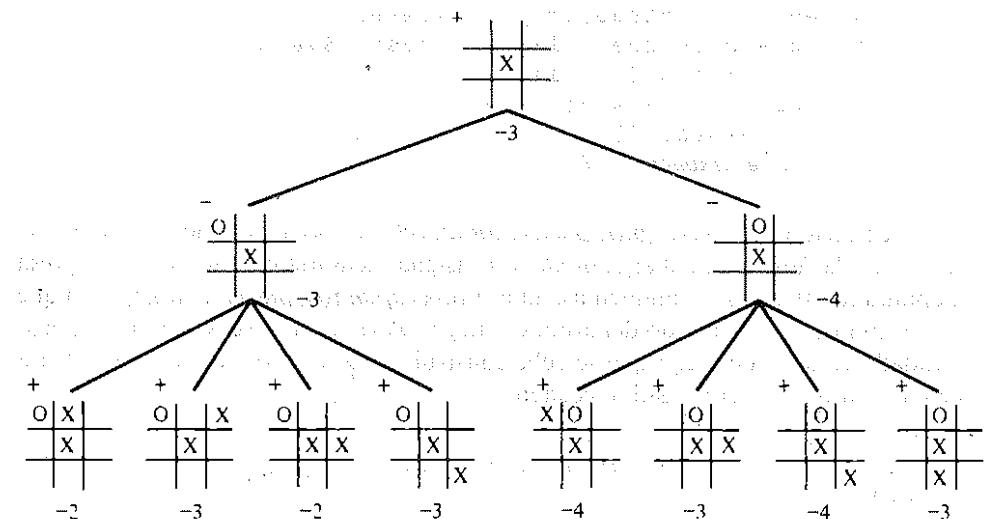
menos va a elegir el movimiento que es peor para más. El valor asignado a cada nodo de la figura 5.6.2 mediante este proceso se indica de inmediato debajo del nodo.

El movimiento que más debe seleccionar, dada la posición del tablero en el nodo raíz, es aquel que eleva al máximo su valor. Así, el movimiento de apertura para X debe ser en el cuadrado medio, como se ilustra en la figura 5.6.2. La figura 5.6.3 ilustra la determinación de la mejor respuesta de O. Adviéntase que la designación de "más" y "menos" depende de a quien se le está determinando su movimiento. Así, en la figura 5.6.2, X está designado como más, mientras que en la 5.6.3 es O. En la aplicación de la función de evaluación estática a una posición del tablero, se calcula el valor de la posición para cualquier jugador designado con más. Este método se llama el método *minimax*, ya que, al subir al árbol, las funciones máximo y mínimo se aplican de manera alterna.

El mejor movimiento para un jugador desde una posición dada, puede determinar construyendo primero el árbol de juego y aplicando una función de evaluación estática a las hojas. Entonces esos valores se mueven hacia arriba aplicando el mínimo y el máximo en los nodos menos y más, respectivamente. Cada nodo del árbol de juego tiene que incluir una representación del tablero y una indicación de si el nodo es un nodo más o menos. Por lo tanto, los nodos pueden declararse por medio de:

```
struct nodetype {
 char board[3][3];
 int turn;
 struct nodetype *hijo;
 struct nodetype *próximo;
};

typedef struct nodetype *NODEPTR;
```



**Figura 5.6.3** Computación de la respuesta del jugador O.

*p -> board[i][j]* tiene el valor 'X', 'O' o ' ', dependiendo de si el cuadrado en el renglón *i* y columna *j* de ese nodo está ocupado por alguno de los jugadores o está desocupado. *p -> turn* tiene el valor +1 o -1, dependiendo de si el nodo es un nodo más o menos, respectivamente. Los dos campos restantes de un nodo se usan para indicar la posición del nodo en el árbol. *p -> son* apunta al hijo mayor del nodo y *p -> next* a su siguiente hermano más joven. Supóngase que la declaración anterior es global, que la lista disponible de nodos se ha establecido y que se han escrito las rutinas *getnode* y *freenode* adecuadas.

La función *nextmove* (*brd*, *player*, *looklevel*, *newbrd*) en C determina el mejor movimiento siguiente. *brd* es un arreglo de 3 por 3 que representa la posición actual del tablero; *player* es 'X' o 'O', dependiendo de qué movimiento se esté calculando (adviértase que en el "gato" el valor de *player* puede calcularse a partir de *brd*, de manera que ese parámetro no es tan necesario), y *looklevel* es el nivel de previsión usado en la construcción del árbol. *newbrd* es un parámetro de salida que representa la mejor posición del tablero que *player* puede alcanzar desde la posición *brd*.

*nextmove* usa dos rutinas auxiliares: *buildtree* y *bestbranch*. La función *buildtree* construye el árbol de juego y da como resultado un apuntador a su raíz. La función *bestbranch* calcula el valor de dos parámetros de salida: *best*, que es un apuntador al nodo del árbol que representa el mejor movimiento, y *value*, que es la evaluación de dicho movimiento mediante la técnica "minimax".

```
nextmove(brd, looklevel, player, newbrd)
char brd[3][3], newbrd[3][3];
int looklevel;
char player;
{
 NODEPTR ptree, best;
 int i, j, value;
 ptree = buildtree(brd, looklevel);
 bestbranch(ptree, player, &best, &value);
 for (i=0; i < 3; ++i)
 for (j=0; j < 3; ++j)
 newbrd[i][j] = best->board[i][j];
} /* fin de nextmove */
```

La función *nextmove* (*brd*, *player*, *looklevel*, *newbrd*) en C determina el mejor movimiento y usa la función auxiliar *getnode*, que asigna memoria para un nodo y regresa un apuntador al mismo. También usa una rutina *expand(p, plevel, depth)*, en la que *p* es un apuntador a un nodo del árbol de juego, *plevel* es su nivel y *depth* es la profundidad del árbol de juego que se debe construir. *expand* produce el subárbol con raíz en *p* para la profundidad adecuada.

```
NODEPTR buildtree(brd, looklevel)
char brd[3][3];
int looklevel;
{
```

```
NODEPTR ptree;
int i, j;
/* crear la raíz del árbol e inicializarla */
ptree = getnode();
for (i=0; i < 3; ++i)
 for (j=0; j < 3; ++j)
 ptree->board[i][j] = brd[i][j];
/* la raíz es un nodo más por definición */
ptree->turn = 1;
ptree->son = NULL;
ptree->next = NULL;
/* creación del resto del árbol de juegos */
expand(ptree, 0, looklevel);
return(ptree);
/* fin de buildtree */
```

*expand* puede implantarse mediante la generación de todas las posiciones del tablero que pueden obtenerse de la posición del tablero apuntada por *p* y el establecimiento de ésta como los hijos de *p* en el árbol de juego. Después *expand* se llama a sí misma de manera recursiva usando esos hijos como parámetros hasta que se alcanza la profundidad deseada. *expand* usa una función auxiliar *generate*, que acepta una posición del tablero *brd* y da como resultado un apuntador a una lista de nodos que contiene las posiciones del tablero que pueden obtenerse a partir de *brd*. Esta lista está ligada por medio del campo *next*. La escritura de *generate* se deja como ejercicio para el lector.

```
expand(p, plevel, depth)
NODEPTR p;
int plevel, depth;
{
 NODEPTR q;
 if (plevel < depth) {
 /* p no es el nivel máximo */
 q = generate(p->board);
 p->son = q;
 while (q != NULL) {
 /* recorrer la lista de nodos */
 if (p->turn == 1)
 q->turn = -1;
 else
 q->turn = 1;
 q->son = NULL;
 expand(q, plevel+1, depth);
 q = q->next;
 } /* fin de while */
 } /* fin de if */
} /* fin de expand */
```

Ya que se creó el árbol de juego, *bestbranch* evalúa los nodos del mismo. Cuando un apuntador a una hoja se transfiere a *bestbranch*, ésta llama a una función *evaluate* que evalúa estáticamente la posición del tablero de esa hoja para el jugador cuyo movimiento se está determinando. El código de *evaluate* queda como ejercicio. Cuando se transfiere a *bestbranch* un apuntador a un nodo no hoja, la rutina se llama a sí misma de manera recursiva en cada uno de sus hijos y después asigna el máximo de los valores de sus hijos al nodo no hoja si éste es un nodo *más*, o el mínimo, si es un nodo *menos*. *bestbranch* también se encarga de no perder de vista qué hijo produjo este valor máximo o mínimo.

Si  $p->turn$  es  $-1$ , el nodo apuntado por  $p$  es un nodo *menos* y se le debe asignar el mínimo de los valores asignados a sus hijos. Sin embargo, si  $p->turn$  es  $+1$ , el nodo apuntado por  $p$  es un nodo *más* y su valor deberá ser el máximo de los valores a los hijos del nodo. Si  $\min(x,y)$  es el mínimo de  $x$  y  $y$  y  $\max(x,y)$  su máximo, entonces,  $\min(x,y) = -\max(-x, -y)$  (se invita a probar la afirmación anterior como ejercicio trivial). Así, el máximo o el mínimo correctos pueden encontrarse como sigue: en el caso de un nodo *más*, calcular el máximo; en el caso de un nodo *menos*, calcular el máximo de los negativos de los valores y cambiar el signo del resultado. Estas ideas están incorporadas en *bestbranch*. Los parámetros de salida *\*pbest* y *\*pvalue* son, respectivamente, un apuntador a aquel hijo de la raíz del árbol que aumenta al máximo su valor y el valor del hijo que ha sido asignado ahora a la raíz.

```

NODEPTR bestbranch(NODEPTR pnd, char player, NODEPTR *pbest, int *pvalue);
{
 NODEPTR p, pbest2;
 int val;

 if (pnd->son == NULL) {
 /* pnd es una hoja */
 *pvalue = evaluate(pnd->tablero, jugador);
 *pbest = pnd;
 }
 else {
 /* el nodo es una hoja, recorrer la lista de hijos */
 p = pnd->son;
 bestbranch(p, player, pbest, pvalue);
 *pbest = p
 if (pnd.turn == -1)
 *pvalue = -*pvalue;
 p = p->next;
 while (p != NULL) {
 bestbranch(p, player, &pbest2, &val);
 if (pnd->turn == -1)
 val = -val;
 if (val > *pvalue) {
 *pvalue = val;
 *pbest = p;
 }
 p = p->next;
 }
 }
}

```

```

if (val > *pvalue) {
 *pvalue = val;
 *pbest = p;
}
/* fin de if */
p = p->next;
}
/* fin de while */
if (pnd->turn == -1)
 *pvalue = -*pvalue;
}
/* fin de if */
}
/* fin de bestbranch */

```

## EJERCICIOS

- 5.6.1. Examine las rutinas de esta sección y determinar si todos los parámetros son realmente necesarios. ¿Cómo se pueden revisar las listas de parámetros?
- 5.6.2. Escriba rutinas en C *generate* y *evaluate* como las descritas en el texto.
- 5.6.3. Vuelva a escribir los programas de esta sección y de la anterior bajo la implantación en la que cada nodo del árbol incluye un campo *father* que contiene un apuntador a su padre. ¿Bajo qué implantación son más eficaces estos programas?
- 5.6.4. Escriba versiones no recursivas de las rutinas *expand* y *bestbranch* proporcionadas en el texto.
- 5.6.5. Modifique la rutina *bestbranch* del texto de modo que los nodos del árbol sean liberados una vez que ya no se necesitan.
- 5.6.6. Combine el proceso de construir el árbol de juego y evaluar sus nodos en un proceso único, de tal manera que no se requiera contar con el árbol de juego completo y se pueda liberar sus nodos cuando no sean necesarios.
- 5.6.7. Modifique el programa del ejercicio previo de manera que si la evaluación de un nodo *menos* es mayor que el mínimo de los valores de los hermanos mayores de su padre, el programa no se preocupa en extender los hermanos más jóvenes de ese nodo *menos*, y si la evaluación de un nodo *más* es menor que el máximo de los valores que los hermanos mayores de su padre, el programa no se preocupa en extender los hermanos más jóvenes de ese nodo *más*. Este método se llama *alfa-beta minimax*. Explicar por qué es correcto.
- 5.6.8. El juego de *kalah* se juega como sigue: dos jugadores tienen cada uno 7 orificios, seis de ellos se llaman *pits* y el séptimo se llama *kalah*. Están dispuestos de acuerdo con el siguiente diagrama.

Jugador 1

K P P P P P  
P P P P P K

Jugador 2

Al inicio hay seis piedras en cada "pit" y ninguna en *kalah*, por lo que la posición de apertura se ve como sigue:

Los jugadores se turnan y cada turno consiste en uno o más movimientos. Para hacer un movimiento, el jugador elige uno de sus pits no vacíos. Las piedras se eliminan de ese pit y se distribuyen en el sentido de las manecillas del reloj, dentro de los pits y de la "kalah" del jugador correspondiente (la kalah del oponente se salta) una piedra por orificio hasta que no queda ninguna. Por ejemplo, si el jugador 1 es el primero en mover, un movimiento de apertura posible daría como resultado la siguiente posición del tablero:

1 7 7 7 7 7 0  
6 6 6 6 6 0

Si la última piedra de un jugador va a parar en su propia "kalah", el jugador puede hacer otro movimiento. Si va a parar en uno de sus pits vacíos, esa piedra y las del oponente que estén en el "pit" opuesto se eliminan y se colocan en la "kalah" del jugador. El juego termina cuando algún jugador ya no tiene piedras en sus "pits". En ese momento se colocan todas las piedras del oponente en la "kalah" del oponente y termina el juego. El ganador es aquél que tenga más piedras en su "kalah". Escriba un programa que acepte la posición de un tablero de "kalah" y una indicación de quién está en turno y produzca el mejor movimiento para ese jugador.

- 5.6.9. ¿Cómo pueden modificarse las ideas del programa para el juego de "gato" con el propósito de calcular el mejor movimiento en un juego que contenga un elemento de azar, como el "backgammon"?
- 5.6.10. ¿Por qué se han podido programar las computadoras para jugar de manera perfecta el "gato" pero no el ajedrez o las damas?
- 5.6.11. El juego de *nim* se juega cómo sigue: se colocan cierto número de palillos en una pila. Se alternan dos jugadores para eliminar uno o dos palillos de la pila. El jugador que elimina el último es el perdedor. Escriba una función en C para determinar el mejor movimiento en "nim".

## Ordenamiento

La búsqueda y el ordenamiento están entre los ingredientes más comunes de los sistemas de programación. En la primera sección de este capítulo, discutimos algunas de las consideraciones globales que involucra el ordenamiento. En el resto del capítulo examinamos algunas de las técnicas de ordenamiento más comunes y las ventajas o desventajas que tienen unas con respecto a otras. En el capítulo siguiente trataremos acerca de la búsqueda y algunas de sus aplicaciones.

### 6.1. ANTECEDENTES GENERALES

El concepto de un conjunto ordenado de elementos tiene un impacto considerable en nuestra vida diaria. Considérese, por ejemplo, el proceso de encontrar un número de teléfono en un directorio. Este proceso, llamado *búsqueda*, se simplifica de manera considerable por el hecho de que los nombres estén registrados por orden alfabético en el directorio. Imagínese el problema que tendría al intentar localizar un número de teléfono si los nombres aparecieran según el orden en que los clientes solicitaron su servicio a la compañía telefónica. En tal caso, los nombres también podrían haber sido registrados en orden aleatorio. El proceso de búsqueda se simplifica porque los registros están clasificados en orden alfabético en vez de cronológico. O considérese el caso de una persona buscando un libro en una biblioteca. Como los libros están acomodados en un orden específico (Librería del Congreso, Sistema Dewey, etc.), a cada libro se le asigna una posición específica respecto a los otros y puede ser recuperado en una cantidad razonable de tiempo (si está ahí). O considérese un conjunto de

números ordenados en forma secuencial en la memoria de una computadora. Como veremos en el capítulo siguiente, por lo general es más fácil encontrar un elemento particular si el conjunto de números se guarda clasificado según un orden. En general, un conjunto de artículos se guarda de manera ordenada con el fin de producir un informe (para simplificar la recuperación manual de información, como en un directorio telefónico o en la estantería de una biblioteca) o para hacer más eficiente el acceso a los datos en una máquina.

Presentamos ahora alguna terminología básica. Un *archivo* de tamaño  $n$  es una secuencia de  $n$  elementos  $r[0], r[1], \dots, r[n - 1]$ . Cada elemento en el archivo se llama un *registro*. (Los términos archivo y registro no se están usando aquí para referirnos a una estructura de datos específica, como en la terminología en C. En lugar de ello, los usamos en un sentido más general). A cada registro  $r[i]$  está asociada una llave,  $k[i]$ . Por lo regular (pero no siempre) la llave es un subcampo del registro entero. Se dice que el archivo está *ordenado de acuerdo a la llave*, si  $i < j$  implica que  $k[i]$  precede a  $k[j]$  para algún ordenamiento de las llaves. En el ejemplo del directorio telefónico, el archivo consta de todas las entradas del libro. Cada entrada es un registro. La llave de acuerdo a la cual está ordenado el archivo es el campo de nombres del registro. Además, cada registro contiene campos para la dirección y el número de teléfono.

Un ordenamiento se puede clasificar como *interno* si los registros que se están ordenando están en la memoria principal, o *externo* si algunos de los registros que se están ordenando están en el almacenamiento auxiliar. Restringimos nuestra atención a los ordenamientos internos.

Es posible que de dos registros de un archivo tengan la misma llave. Una técnica de ordenamiento se llama *estable* si para todos los registros  $i$  y  $j$  tales que  $k[i]$  sea igual a  $k[j]$ , si  $r[i]$  precede a  $r[j]$  en el archivo original, entonces  $r[i]$  también precede a  $r[j]$  en el archivo ordenado. Es decir, un ordenamiento estable mantiene los registros con llaves iguales en el mismo orden relativo en el que estaban antes del ordenamiento.

Un ordenamiento ocurre ya sea sobre los mismos registros o sobre una tabla auxiliar de apuntadores. Por ejemplo, considérese la figura 6.1.1a en la cual se muestra un archivo de cinco registros. Si se ordena el archivo en orden ascendente de

Llaves, Otros campos

|            | 4 | DDD |
|------------|---|-----|
| Registro 1 | 1 | AAA |
| Registro 2 | 2 | BBB |
| Registro 3 | 1 | AAA |
| Registro 4 | 5 | EEE |
| Registro 5 | 3 | CCC |

(a) Archivo original,

|            | 1 | AAA |
|------------|---|-----|
| Registro 1 | 2 | BBB |
| Registro 2 | 3 | CCC |
| Registro 3 | 4 | DDD |
| Registro 4 | 5 | EEE |
| Registro 5 | 5 | EEE |

(b) Archivo ordenado

Figura 6.1.1 Ordenamiento de registros reales.

acuerdo a las llaves numéricas mostradas, el archivo que resulta es como el de la figura 6.1.1b. En este caso los propios registros han sido ordenados.

Supóngase, sin embargo, que la cantidad de datos almacenada en cada uno de los registros en el archivo de la figura 6.1.1a es tan grande que la sobrecarga que implica mover los datos reales es prohibitiva. En ese caso, puede usarse una tabla auxiliar de apuntadores de manera que esos apuntadores sean movidos en lugar de los datos reales, como se muestra en la figura 6.1.2. (Esto se llama *ordenamiento por dirección*.) La tabla en el centro es el archivo y la de la izquierda es la tabla inicial de apuntadores. La entrada en la posición  $j$  en la tabla de apuntadores apunta al registro  $j$ . Durante el proceso de ordenamiento, las entradas en la tabla de apuntadores se ajustan de tal manera que la tabla final es como la que se muestra en la figura de la derecha. Al inicio, el primer apuntador señalaba hacia la primera entrada en el archivo; al terminar el primer apuntador señala a la cuarta entrada de la tabla. Obsérvese que no se mueve ninguna de las entradas originales del archivo. En muchos de los programas en este capítulo ilustramos técnicas de ordenamiento de registros reales. La extensión de estas técnicas para ordenamiento por dirección es rectilínea y se dejará como ejercicio al lector. (En realidad, ordenamos sólo las llaves en los ejemplos de este capítulo en aras de obtener simplicidad; dejamos al lector la modificación de los programas para ordenar registros completos.)

Dada la relación entre búsqueda y ordenamiento, la primera cuestión que se debe plantear en cualquier aplicación es si debería o no ordenarse un archivo. En ocasiones, buscar un elemento particular en un conjunto implica menos trabajo que ordenar primero el conjunto completo para después extraer el elemento deseado. Por otra parte, si se requiere del uso frecuente del archivo con el propósito de recuperar elementos específicos, podría ser más eficiente ordenarlo primero. Esto ocurre porque la sobrecarga que implican las búsquedas sucesivas puede exceder en mucho a la que implica ordenar antes el archivo para luego recuperar del mismo los elementos correspondientes. Así, no puede decirse que es más eficiente ordenar o no ordenar. El programador debe tomar una decisión basándose en circunstancias individuales. Una vez tomada la decisión de ordenar, tienen que tomarse otras decisiones, incluyendo qué debe ser ordenado y cuáles métodos deben usarse. No hay un método de ordenamiento que sea universalmente superior a todos los otros. El programador

Tabla original de apuntadores Archivo Tabla ordenada de apuntadores

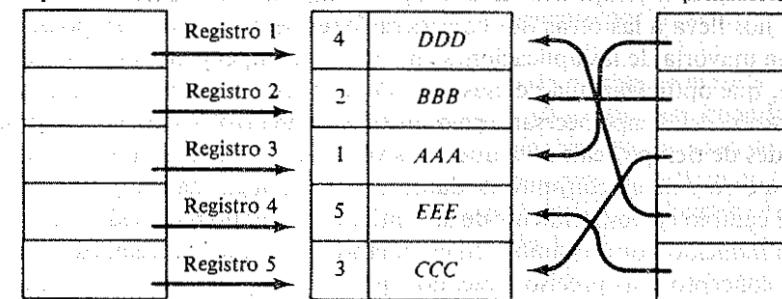


Figura 6.1.2 Ordenamiento usando una tabla auxiliar de apuntadores.

debe examinar el problema de manera cuidadosa, así como los resultados deseados, antes de decidir estas cuestiones tan importantes.

### Consideraciones de eficiencia

Como veremos en este capítulo, hay un gran número de métodos que pueden usarse para ordenar un archivo. El programador debe estar enterado de varias consideraciones de eficiencia que se interrelacionan y con frecuencia entran en conflicto para hacer una elección inteligente del método de ordenamiento más apropiado a un problema particular. Tres de las consideraciones más importantes, entre esas, incluyen la cantidad de tiempo que debe invertir un programador para codificar un programa particular de ordenamiento, la cantidad de tiempo de máquina necesaria para que el programa corra y la cantidad de espacio necesaria para el programa.

Si un archivo es pequeño, por lo regular ocurre que técnicas de ordenamiento complejas, diseñadas para minimizar los requerimientos de espacio y tiempo, son peores o mejores sólo por un margen pequeño en el logro de eficiencia que métodos más simples y, por lo general, menos eficientes. De manera similar, si un programa particular de ordenamiento debe correrse sólo una vez y hay suficiente tiempo y espacio de máquina para ello, sería ridículo que un programador invirtiera días investigando los mejores métodos para obtener el último gramo de eficiencia. En tales casos, la cantidad de tiempo que debe invertir el programador es, de manera primordial, para la consideración de determinar qué método de ordenamiento usar. Sin embargo hay que tener mucho cuidado. El tiempo de programación nunca es una excusa válida para usar un programa incorrecto. Un ordenamiento que se corra una sola vez puede darse el lujo de una técnica ineficiente, pero no de una incorrecta. Los datos presumiblemente ordenados pueden usarse en una aplicación en la que la suposición de datos ordenados sea crucial.

Sin embargo, un programador debe ser capaz de reconocer el hecho de que un ordenamiento particular sea ineficiente y de justificar su uso en una situación particular. Muy a menudo, los programadores toman la vía fácil y programan un ordenamiento ineficiente, que se incorpora después a un sistema mayor en el cual dicho ordenamiento es un componente llave. Los diseñadores y planeadores del sistema se sorprenden después de lo inadecuado de su creación. Para maximizar su eficiencia, un programador tiene que conocer un amplio rango de técnicas de ordenamiento así como sus ventajas y desventajas, de tal manera que cuando surja la necesidad de un ordenamiento pueda suministrar la más apropiada para la situación particular.

Esto nos lleva a las otras dos consideraciones de eficiencia: tiempo y espacio. Como en la mayoría de las aplicaciones en computación, el programador tiene, con frecuencia, que optimizar una de esas consideraciones en detrimento de la otra. En la consideración del tiempo necesario para ordenar un archivo de tamaño  $n$  no tratamos con unidades de tiempo reales, ya que éstas variarán de una máquina a otra, de un programa a otro y de un conjunto de datos a otro. En lugar de ello, estamos interesados en el cambio correspondiente de la cantidad de tiempo requerido para ordenar un archivo inducido por un cambio en el tamaño del mismo,  $n$ . Veamos si podemos hacer este concepto más preciso. Decimos que  $y$  es *proporcional* a  $x$  si la relación entre  $y$  y  $x$  es tal que la multiplicación de  $x$  por una constante multiplica a  $y$  por la

misma constante de proporcionalidad = 1. Así, si  $y$  es proporcional a  $x$ , duplicar  $x$  duplicará a  $y$ , y multiplicar  $x$  por 10 multiplicará  $y$  por 10. De manera similar, si  $y$  es proporcional a  $x$ , duplicar  $x$  multiplicará a  $y$  por 4 constante de proporcionalidad = 2 y multiplicar  $x$  por 10 multiplicará  $y$  por 100.

Con frecuencia, no medimos la eficiencia en cuanto a tiempo de un ordenamiento por el número de unidades de tiempo requeridas sino por el número de operaciones críticas ejecutadas. Ejemplos de tales operaciones críticas son comparaciones de llaves (es decir, las comparaciones de las llaves de dos registros en el archivo para determinar cuál es mayor), movimientos de registros o apuntadores a registros, o intercambio de dos registros. Las operaciones críticas elegidas son aquellas que toman más tiempo. Por ejemplo, una comparación de llaves puede ser una operación compleja, en especial si las propias llaves son largas o el orden entre ellas no es trivial. Así, una comparación de llaves requiere mucho más tiempo que, digamos, un simple incremento de una variable índice en una iteración *for*. También, el número de operaciones simples requerido es, por lo regular, proporcional al número de comparaciones de llaves. Por esta razón, el número de comparaciones de llave es una medida útil de la eficiencia en tiempo de un ordenamiento.

Hay dos maneras de determinar los requerimientos de tiempo de un ordenamiento, ninguna de las cuales produce resultados que sean aplicables a todos los casos. Un método es hacer el análisis matemático y a veces intrincado de varios casos (por ejemplo, mejor caso, peor caso y caso promedio). El resultado de este análisis es con frecuencia una fórmula que da el tiempo promedio (o número de operaciones) requerido por un ordenamiento particular como función del tamaño del archivo  $n$ . (En realidad, los requerimientos de tiempo de un ordenamiento dependen de otros factores más que del tamaño del archivo; sin embargo, aquí nos ocupamos sólo de la dependencia del tamaño del archivo.) Suponer un análisis matemático de ese tipo de un programa de ordenamiento particular que lleva a la conclusión de que el programa toma un tiempo de ejecución de  $0.01n^2 + 10n$  unidades. Las columnas primera y cuarta de la figura 6.1.3 muestran el tiempo necesario para el ordenamiento de varios valores de  $n$ . Se notará que para valores pequeños de  $n$ , la cantidad  $10n$  (tercera columna de la figura 6.1.3) sobrepasa la cantidad  $0.01n^2$  (segunda columna). Esto ocurre porque la diferencia entre  $n^2$  y  $n$  es pequeña para valores pequeños de  $n$  y está más que compensada por la diferencia entre 10 y 0.01. Así, para valores pequeños de  $n$ , un incremento de  $n$  por el factor 2 (por ejemplo de 50 a 100) aumenta el tiempo necesario para el ordenamiento por el mismo factor 2 aproximadamente (de 525 a 1100). De manera similar, un incremento de  $n$  por el factor 5 (por ejemplo, de 10 a 50) aumenta el tiempo necesario para el ordenamiento por el factor 5, más o menos (de 101 a 525).

Sin embargo, cuando  $n$  se hace más grande, la diferencia entre  $n^2$  y  $n$  crece tan rápido que compensa al final la diferencia entre 10 y 0.01. Así, cuando  $n$  es igual a 1000 los dos términos contribuyen de la misma manera a la cantidad de tiempo necesaria para el programa. Cuando  $n$  se hace aún más grande, el término  $0.01n^2$  rebasa el término  $10n$  y la contribución de  $10n$  se vuelve casi insignificante. Así, para valores grandes de  $n$ , un incremento de  $n$  por el factor 2 (por ejemplo, de 50000 a 100000) resulta en un aumento del tiempo de ordenamiento cercano a 4 (de 25.5 millones a 101 millones) y un incremento de  $n$  en el factor 5 (por ejemplo, de 10000 a

50000) lo incrementa en aproximadamente un factor de 25 (de 1.1 millones a 25.5 millones). En realidad, cuando  $n$  se vuelve cada vez mayor, el tiempo de ordenamiento se vuelve cada vez más proporcional a  $n^2$ , como se ilustra claramente en la última columna de la figura 6.1.3. Así, para  $n$  grande, el tiempo requerido por el ordenamiento es casi proporcional a  $n^2$ . Por supuesto, para valores pequeños de  $n$ , el ordenamiento puede exhibir un comportamiento muy diferente (como en la figura 6.1.3), situación que tiene que ser tomada en cuenta en el análisis de su eficiencia.

#### Notación O

Introducimos alguna terminología y una nueva notación para captar el concepto de una función que se vuelve proporcional a otra cuando crece. En el ejemplo previo, se dice que la función  $0.01n^2 + 10n$  es “del orden de” la función  $n^2$ , porque cuando  $n$  se hace grande, la función se vuelve casi proporcional a  $n^2$ .

Para ser precisos, dadas dos funciones  $f(n)$  y  $g(n)$ , decimos que  $f(n)$  es *del orden de*  $g(n)$  o que  $f(n)$  es  $O(g(n))$  si existen enteros positivos  $a$  y  $b$  tales que  $f(n) \leq a * g(n)$  para todo  $n \geq b$ . Por ejemplo, si  $f(n) = n^2 + 100n$  y  $g(n) = n^2$ ,  $f(n)$  es  $O(g(n))$ , ya que  $n^2 + 100n$  es menor o igual que  $2n^2$  para todo  $n$  mayor o igual que 100. En este caso  $a$  es igual a 2 y  $b$  es igual a 100. Del mismo modo  $f(n)$  es también  $O(n^3)$ , dado que  $n^2 + 100n$  es menor o igual que  $2n^3$  para todo  $n$  mayor o igual que 8. Dada una función  $f(n)$ , puede haber muchas funciones  $g(n)$  tales que  $f(n)$  sea  $O(g(n))$ .

Si  $f(n)$  es  $O(g(n))$ ,  $f(n)$  se vuelve “finalmente” (es decir, para  $n \geq b$ ) en forma permanente menor o igual que algún múltiplo de  $g(n)$ . En un sentido estamos diciendo que  $f(n)$  está acotada por arriba mediante  $g(n)$  o que  $f(n)$  es una función “menor” que  $g(n)$ . Otra manera formal de decir esto es que  $f(n)$  está *asintóticamente acotada* por  $g(n)$ . Otra interpretación es que  $f(n)$  crece más despacio que  $g(n)$ , dado que, en proporción (es decir a partir del factor  $a$ ),  $g(n)$  se vuelve al final más grande.

Es fácil mostrar que si  $f(n)$  es  $O(g(n))$  y  $g(n)$  es  $O(h(n))$ , entonces  $f(n)$  es  $O(h(n))$ . Por ejemplo,  $n^2 + 100n$  es  $O(n^2)$  y  $n^2$  es  $O(n^3)$  (para ver esto hacer  $a$  y  $b$  iguales a 1): en consecuencia  $n^2 + 100n$  es  $O(n^3)$ . Esta propiedad se llama *propiedad transitiva*.

Obsérvese que si  $f(n)$  es una función constante [es decir,  $f(n) = c$  para todo  $n$ ],  $f(n)$  es  $O(1)$ , dado que haciendo  $a$  igual a  $c$  y  $b$  igual a 1, tenemos que  $c \leq c * 1$  para

| $n$     | $a = 0.01n^2$ | $a + b = 10n$ | $a + b = n^2$ | $a + b = n^3$ |
|---------|---------------|---------------|---------------|---------------|
| 10      | 1             | 100           | 101           | 1.01          |
| 50      | 25            | 500           | 525           | 0.21          |
| 100     | 100           | 1,000         | 1,100         | 0.11          |
| 500     | 2,500         | 5,000         | 7,500         | 0.03          |
| 1,000   | 10,000        | 10,000        | 20,000        | 0.02          |
| 5,000   | 250,000       | 500,000       | 300,000       | 0.01          |
| 10,000  | 1,000,000     | 100,000       | 1,100,000     | 0.014         |
| 50,000  | 25,000,000    | 500,000       | 25,500,000    | 0.01          |
| 100,000 | 100,000,000   | 1,000,000     | 101,000,000   | 0.01          |
| 500,000 | 2,500,000,000 | 5,000,000     | 2,505,000,000 | 0.01          |

Figura 6.1.3

todo  $n \geq 1$ . (De hecho, el valor de  $b$  o  $n$  es irrelevante, puesto que el valor de una función constante es independiente de  $n$ .)

También es fácil mostrar que la función  $c * n$  es  $O(n^k)$  para cualesquier constantes  $c$  y  $k$ . Para verlo, nótese simplemente que  $c * n$  es menor o igual que  $c * n^k$  para todo  $n \geq 1$  (es decir, hacer  $a = c$ ,  $b = 1$ ). También es obvio que  $n^k$  es  $O(n^{k+j})$  para todo  $j \geq 0$  (usar  $a = 1$ ,  $b = 1$ ). De la misma manera podemos mostrar que si  $f(n)$  y  $g(n)$  son ambas  $O(h(n))$ , la nueva función  $f(n) + g(n)$  es también  $O(h(n))$ . Todos esos hechos juntos pueden usarse para mostrar que si  $f(n)$  es un polinomio con término principal de orden  $k$  [es decir,  $f(n) = c_1 * n^k + c_2 * n^{k-1} + \dots + c_k * n + c_{k+1}$ ],  $f(n)$  es  $O(n^{k+j})$  para todo  $j \geq 0$ .

Aunque una función puede ser asintóticamente acotada por muchas otras [como por ejemplo,  $10n^2 + 37n + 153$  es  $O(n^2)$ ,  $O(10n^2)$ ,  $O(37n^2 + 10n)$  y  $O(0.05n^3)$ ], por lo regular buscamos una cota asintótica que sea un sólo término con coeficiente principal igual a 1 y que sea “tan cercana” como sea posible. Así diríamos que  $10n^2 + 37n + 153$  es  $O(n^2)$ , aunque también esté asintóticamente acotada por muchas otras funciones. De manera igual nos gustaría encontrar una función  $g(n)$  tal que  $f(n)$  fuese  $O(g(n))$  y  $g(n)$  fuese  $O(f(n))$ . Si  $f(n)$  es una constante o un polinomio, esto puede hacerse siempre usando su término principal con coeficiente 1. Para funciones más complejas, sin embargo, no siempre es posible encontrar un ajuste fijo como ese.

Una función importante en el estudio de la eficiencia de un algoritmo es la función logarítmica. Recordar que  $\log_m n$  es el valor  $x$  para el cual  $m^x$  es igual a  $n$ .  $m$  se llama la *base* del logaritmo. Considerar las funciones  $\log_m n$  y  $\log_k n$ . Sea  $xm$  el  $\log_m n$  y  $xk$  el  $\log_k n$ . Entonces

$$m^{xm} = n \quad \text{y} \quad k^{xk} = n$$

de manera que

$$m^{xm} = k^{xk}$$

Tomando  $\log_m$  en ambos lados,

$$xm = \log_m(k^{xk})$$

Ahora se puede mostrar fácilmente que  $\log_z(x^y)$  es igual a  $y * \log_z x$  para toda  $x$ ,  $y$  y  $z$ , de manera que la última ecuación se puede volver a escribir como (recuerde que  $xm = \log_m n$ )

$$\log_m n = xk * \log_m k$$

o como (recordar que  $xk = \log_k n$ )

$$\log_m n = (\log_m k) * \log_k n$$

Así,  $\log_m n$  y  $\log_k n$  son múltiplos constantes cada una de la otra.

Es fácil mostrar que si  $f(n) = c * g(n)$ , donde  $c$  es una constante,  $f(n)$  es  $O(g(n))$  [en realidad ya habíamos mostrado que esto es cierto para la función  $f(n) = n^k$ ]. Así  $\log_m n$  es  $O(\log_k n)$  y  $\log_k n$  es  $O(\log_m n)$  para toda  $m$  y  $k$ . Como cada función logarítmica es del orden de cualquier otra, por lo regular omitimos la base

cuando hablamos de funciones de orden logarítmico y decimos que todas ellas son  $O(\log n)$ .

Los siguientes planteamientos establecen un orden jerárquico de funciones:

$c$  es  $O(1)$  para toda constante  $c$ .

$c$  es  $O(\log n)$ , pero  $\log n$  no es  $O(1)$ .

$c * \log_n n$  es  $O(\log n)$  para cualesquier constantes  $c, k$ .

$c * \log_k n$  es  $O(n)$ , pero  $n$  no es  $O(\log n)$ .

$c * n^k$  es  $O(n^k)$  para cualesquier constantes  $c, k$ .

$c * n^k$  es  $O(n^{k+j})$ , pero  $n^{k+j}$  no es  $O(n^k)$ .

$c * n * \log_k n$  es  $O(n \log n)$  para cualesquier constantes  $c, k$ .

$c * n * \log_k n$  es  $O(n^2)$ , pero  $n^2$  no es  $O(n \log n)$ .

$c * n^j * \log_k n$  es  $O(n^j \log n)$  para cualesquier constantes  $c, j, k$ .

$c * n^j * \log_k n$  es  $O(n^{j+1})$ , pero  $n^{j+1}$  no es  $O(n^j \log n)$ .

$c * n^j * (\log_k n)^l$  es  $O(n^j (\log n)^l)$  para cualesquier constantes  $c, j, k$  y  $l$ .

$c * n^j * (\log_k n)^l$  es  $O(n^{j+1})$  pero  $n^{j+1}$  no es  $O(n^j (\log n)^l)$ .

$c * n^j * (\log_k n)^l$  es  $O(n^j (\log n)^{l+1})$  pero  $n^j (\log n)^{l+1}$  no es  $O(n^j (\log n)^l)$ .

$c * n^k$  es  $O(d^n)$  pero  $d^n$  no es  $O(n^k)$  para cualesquier constantes  $c$  y  $k$ , y  $d > 1$ .

La jerarquía de funciones establecida por lo anterior, con cada función de orden menor que la que le sigue, es  $c, \log n, (\log n)^k, n, n, (\log n)^k, n^k, n^k, (\log n)^l, n^{k+1}$  y  $d^n$ .

Las funciones que son  $O(n^k)$  para alguna  $k$  se dice que son de orden *polinomial*, mientras que las funciones que son  $O(d^n)$  para alguna  $d > 1$  pero no  $O(n^k)$  para cualquier  $k$  se dice que son de *orden exponencial*.

La distinción entre funciones de orden polinomial y exponencial es muy importante. Incluso una función de orden exponencial pequeño, como  $2^n$ , crece mucho más que cualquier función de orden polinomial como  $n^k$ , sin importar el tamaño de  $k$ . Como ilustración de la rapidez con que crece una función de orden exponencial, considérese que  $2^{10}$  es igual a 1024 pero  $2^{100}$  (es decir,  $1024^{10}$ ) es mayor que el número formado por un 1 seguido de 30 ceros. El menor  $k$  para el cual  $10^k$  excede a  $2^{10}$  es 4, pero el menor  $k$  para el cual  $100^k$  excede a  $2^{100}$  es 16. Cuando  $n$  se hace más grande, se necesitan valores de  $k$  mayores para que  $n^k$  no quede rezagado de  $2^n$ . Para cualquier  $k$  fijo,  $2^n$  finalmente se hace de manera permanente más grande que  $n^k$ .

Dado el increíble grado de crecimiento de las funciones de orden exponencial, los problemas que requieren de algoritmos con tiempo exponencial para su solución se consideran *intratables* con el equipo de cómputo actual; es decir, problemas de ese tipo no pueden resolverse con precisión excepto en los casos más simples.

## Eficiencia de un ordenamiento

Con este concepto de “es de orden de” para un ordenamiento, podemos comparar varias técnicas y clasificarlas como “buenas” o “malas” en términos generales. Podría esperarse descubrir el ordenamiento “óptimo” (que sea  $O(n)$  sin importar el contenido o el orden de la entrada). Sin embargo, desafortunadamente puede mostrarse que no existe tal ordenamiento que sea útil siempre. La mayoría de los ordenamientos clásicos que consideraremos tienen requerimientos de tiempo que van de  $O(n \log n)$  a  $O(n^2)$ . En el primero, la multiplicación del tamaño del archivo por 100 multiplicará el tiempo de ordenamiento por algo menos que 200; en el último, la multiplicación del tamaño del archivo por 100 multiplica el tiempo de ordenamiento por un factor de 10000. La figura 6.1.4 muestra la comparación de  $n \log n$  con  $n^2$  para un rango de valores de  $n$ . Puede verse en la figura que para  $n$  grande, cuando se incrementa  $n$ ,  $n^2$  aumenta a un ritmo mucho más rápido que  $n \log n$ . Sin embargo, no debería seleccionarse una técnica de ordenamiento por el simple hecho de ser  $O(n \log n)$ . La relación del tamaño  $n$  del archivo y de los otros términos que constituyen el tiempo de ordenamiento real deben conocerse. En particular, los términos que juegan un papel insignificante para valores grandes de  $n$  pueden tener un papel dominante para valores pequeños de  $n$ . Todos estos resultados deben considerarse antes de que pueda hacerse una selección inteligente del ordenamiento.

Un segundo método para determinar los requerimientos de tiempo de una técnica de ordenamiento es correr en realidad el programa y medir su eficiencia (ya sea midiendo las unidades de tiempo absolutas o el número de operaciones ejecutadas). Para usar tales resultados en la medición de la eficiencia de un ordenamiento, la prueba tiene que correrse en “muchos” archivos muestra. Aun cuando se recojan tales estadísticas, la aplicación de ese ordenamiento a un archivo específico puede no producir resultados que sigan el patrón general. Atributos peculiares del archivo en cuestión pueden hacer que la velocidad del ordenamiento se desvíe de manera significativa. En los ordenamientos de las secciones siguientes daremos una explicación intuitiva de porqué un ordenamiento particular se clasifica como  $O(n^2)$  o  $O(n \log n)$ ; dejamos el análisis matemático y la verificación sofisticada de datos empíricos como un ejercicio para el lector ambicioso.

En muchos casos el tiempo necesario para un ordenamiento depende de la secuencia original de los datos. Para algunos ordenamientos, los datos de entrada que

| $n$             | $n \log_{10} n$   | $n^2$                |
|-----------------|-------------------|----------------------|
| $1 \times 10^1$ | $1.0 \times 10^1$ | $1.0 \times 10^2$    |
| $5 \times 10^1$ | $8.5 \times 10^1$ | $2.5 \times 10^3$    |
| $1 \times 10^2$ | $2.0 \times 10^2$ | $1.0 \times 10^4$    |
| $5 \times 10^2$ | $1.3 \times 10^3$ | $2.5 \times 10^5$    |
| $1 \times 10^3$ | $3.0 \times 10^3$ | $1.0 \times 10^6$    |
| $5 \times 10^3$ | $1.8 \times 10^4$ | $2.5 \times 10^7$    |
| $1 \times 10^4$ | $4.0 \times 10^4$ | $1.0 \times 10^8$    |
| $5 \times 10^4$ | $2.3 \times 10^5$ | $2.5 \times 10^9$    |
| $1 \times 10^5$ | $5.0 \times 10^5$ | $1.0 \times 10^{10}$ |
| $5 \times 10^5$ | $2.8 \times 10^6$ | $2.5 \times 10^{11}$ |
| $1 \times 10^6$ | $6.0 \times 10^6$ | $1.0 \times 10^{12}$ |
| $5 \times 10^6$ | $3.3 \times 10^7$ | $2.5 \times 10^{13}$ |
| $1 \times 10^7$ | $7.0 \times 10^7$ | $1.0 \times 10^{14}$ |

Figura 6.1.4 Una

comparación de  $n \log n$  y  $n^2$  para varios valores de  $n$ .

estén casi ordenados pueden terminar de ordenarse en tiempo  $O(n)$ , mientras que los que están en orden inverso requieren de un tiempo  $O(n^2)$ . Para otros ordenamientos, el tiempo requerido es  $O(n \log n)$  sin importar el orden original de los datos. Así, si tenemos algún conocimiento acerca de la secuencia original de los datos podemos tomar una decisión más inteligente acerca de qué método de ordenamiento elegir. Por otra parte, si no tenemos tal conocimiento podemos desear seleccionar un ordenamiento basado en el peor caso posible o en el caso "promedio". En cualquier caso, el único comentario general que puede hacerse acerca de técnicas de ordenamiento es que no existe una técnica general "mejor" de ordenamiento. La elección de una de ellas debe depender necesariamente de las circunstancias específicas.

Una vez que ha seleccionado una técnica de ordenamiento particular, el programador debe proceder a hacer el programa tan eficiente como sea posible. En muchas aplicaciones de programación es necesario sacrificar eficiencia para obtener claridad. Con los ordenamientos, la situación es por lo general opuesta. Una vez que el programa para el ordenamiento ha sido escrito y probado, la meta principal del programador es mejorar su velocidad aunque se vuelva menos legible. La razón para esto es que un ordenamiento puede ser responsable de la mayor parte de la eficiencia de un programa, de manera que cualquier perfeccionamiento en el tiempo de ordenamiento afecta de modo significativo la eficiencia global. Otra razón es que un ordenamiento se usa por lo general con bastante frecuencia, por lo que un pequeño aumento en su velocidad de ejecución ahorra una gran cantidad de tiempo de computación. Normalmente, es una buena idea eliminar los llamados a funciones, en especial si éstos se encuentran en ciclos internos, y remplazarlos por el código de la función en línea, dado que el mecanismo llamada-retorno de un lenguaje puede ser muy costoso en términos de tiempo. También, una llamada a una función puede implicar la asignación de memoria a variables locales, una actividad que requiere a veces una llamada al sistema operativo. En muchos de los programas no se hace así para no oscurecer el objetivo del programa con enormes bloques de código.

Las restricciones de espacio son por lo general menos importantes que las consideraciones de tiempo. Una razón para ello es que para la mayor parte de los programas de ordenamiento, la cantidad de espacio requerida es más próxima a  $O(n)$  que a  $O(n^2)$ . Una segunda razón es que si se requiere de más espacio, éste puede casi siempre encontrarse en la memoria auxiliar. Un ordenamiento ideal es un *ordenamiento en el lugar* cuyos requerimientos de espacio adicional son  $O(1)$ . Es decir, un ordenamiento en el lugar manipula los elementos que deben ser ordenados dentro del espacio de la lista o arreglo que contiene la entrada original no ordenada. Cualquier espacio adicional requerido está en la forma de un número constante de localidades (tales como las variables individuales declaradas de un programa), sin importar el tamaño del conjunto que ha de ser ordenado.

Por lo general, la expectativa de relación entre tiempo y espacio se mantiene para los algoritmos de ordenamiento: aquellos programas que requieren menos tiempo requieren más espacio y viceversa. Sin embargo, existen algoritmos muy diestros que usan tanto tiempo como espacio mínimos; es decir, son ordenamientos en el lugar y  $O(n \log n)$ . Sin embargo, éstos pueden requerir más tiempo del programador para desarrollar y verificar. Tienen también constantes de proporcionalidad más altas que muchos ordenamientos que usan más espacio o que tienen orden de

tiempo mayor y, por lo tanto, requieren más tiempo para ordenar conjuntos pequeños.

En las secciones restantes investigamos algunas de las técnicas más populares de ordenamiento e indicamos algunas de sus ventajas y desventajas.

## EJERCICIOS

- 6.1.1. Elija una técnica de ordenamiento con la que se esté familiarizado.
  - a. Escribir un programa para el ordenamiento.
  - b. ¿Es estable dicho ordenamiento?
  - c. Determinar los requerimientos de tiempo de ordenamiento como una función del tamaño del archivo, de manera matemática y empírica.
  - d. ¿De qué orden es el ordenamiento?
  - e. ¿A partir de qué tamaño del archivo comienza el término más dominante a eclipsar a los otros?
- 6.1.2. Muestre que la función  $(\log_m n)^k$  es  $O(n)$  para todo  $m$  y  $k$  pero  $n$  no es  $O((\log n)^k)$  para cualquier  $k$ .
- 6.1.3. Suponga que un cierto requerimiento de tiempo está dado por la fórmula  $a * n^2 + b * n * \log_2 n$ , donde  $a$  y  $b$  son constantes. Contestar las siguientes preguntas probando los resultados de manera matemática y escribiendo un programa para validar dichos resultados de manera empírica.
  - a. ¿Para qué valores de  $n$  (expresado en términos de  $a$  y  $b$ ) el primer término domina al segundo?
  - b. ¿Para qué valor de  $n$  (en términos de  $a$  y  $b$ ) los dos términos son iguales?
  - c. ¿Para qué valores de  $n$  (expresado en términos de  $a$  y  $b$ ) el segundo término no domina al primero?
- 6.1.4. Muestre que cualquier proceso que ordene un archivo puede extenderse para encontrar todos los duplicados en el archivo.
- 6.1.5. Un *árbol de decisión para ordenamiento* es un árbol binario que representa un método de ordenamiento basado en comparaciones. La figura 6.1.5 ilustra tal árbol de decisión para un archivo de tres elementos. Cada hoja de un árbol de ese tipo representa una comparación entre dos elementos. Cada hoja representa un archivo ordenado por completo. Una rama izquierda de una no hoja indica que la primera llave fue más pequeña que la segunda; una rama derecha indica que fue más grande. (Suponemos que todos los elementos del archivo tienen llaves distintas.) Por ejemplo, el árbol de la figura 6.1.5 representa un ordenamiento de tres elementos  $x[0]$ ,  $x[1]$ ,  $x[2]$  que procede como sigue:  
 Comparar  $x[0]$  con  $x[1]$ . Si  $x[0] < x[1]$ , comparar  $x[1]$  con  $x[2]$ , y si  $x[1] < x[2]$ , el tipo de ordenamiento del archivo es  $x[0], x[1], x[2]$ ; en otro caso si  $x[0] > x[2]$  el tipo de ordenamiento será  $x[0], x[2], x[1]$ , y si  $x[0] > x[1]$ , el tipo de ordenamiento será  $x[2], x[0], x[1]$ . Si  $x[0] > x[1]$ , proceder de manera similar hacia abajo del subárbol derecho.
  - a. Muestre un árbol de decisión para ordenamiento que nunca haga comparaciones redundantes (es decir, que nunca compare  $x[i]$  y  $x[j]$  si se conoce la relación entre ellos). Tiene  $n!$  hojas.
  - b. Demuestre que la profundidad de tal árbol de decisión es al menos  $\log_2(n!)$ .
  - c. Demuestre que  $n! \geq (n/2)^{n/2}$ , de tal manera que la profundidad de dicho árbol es  $O(n \log n)$ .

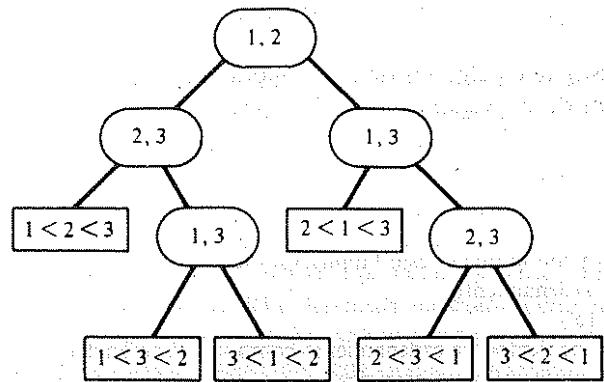


Figura 6.1.5 Un árbol de decisión para un archivo de tres elementos.

d. Explique por qué esto prueba que cualquier método de ordenamiento que use comparaciones en un archivo de tamaño  $n$  tiene que hacer al menos  $O(n \log n)$  comparaciones.

- 6.1.6. Dado un árbol de decisión para ordenamiento de un archivo como en el ejercicio previo, mostrar que si el archivo contiene algunos elementos iguales, el resultado de la aplicación del árbol al archivo (donde se toma una rama izquierda o una derecha cuando dos elementos sean iguales) es un archivo ordenado.
- 6.1.7. Extienda el concepto de un árbol de decisión binario de los ejercicios previos a un árbol ternario que incluya la posibilidad de igualdad. Se desea determinar cuáles elementos del archivo son iguales, en adición al orden de los distintos elementos del mismo. ¿Cuántas comparaciones son necesarias?
- 6.1.8. Demuestre que si  $k$  es el menor entero mayor o igual a  $n + \log_2 n - 2$ , son necesarias y suficientes  $k$  comparaciones para encontrar el primero y segundo elementos más grandes de un conjunto de  $n$  elementos distintos.
- 6.1.9. ¿Cuántas comparaciones se requieren para encontrar el elemento mayor y el menor de un conjunto de  $n$  elementos distintos?
- 6.1.10. Muestre que la función  $f(n)$  definida por
- $$f(1) = 1$$
- $$f(n) = f(n - 1) + 1/n \text{ para } n > 1$$
- es  $O(\log n)$ .

## 6.2. ORDENAMIENTOS DE INTERCAMBIO

### Ordenamiento de burbuja

El primer ordenamiento que presentamos es quizás el más ampliamente conocido entre los estudiantes que se inicián en la programación: el *ordenamiento de burbuja*. Una de las características de este ordenamiento es que es fácil de entender y programar. Aunque, entre todos los ordenamientos que consideraremos, es probable que sea el menos eficiente.

En cada uno de los ejemplos subsecuentes,  $x$  es un arreglo de enteros de los cuales los  $n$  primeros deben ser ordenados de manera que  $x[i] \leq x[j]$  para  $0 \leq i < j < n$ . Es fácil extender este simple formato a uno que se use en el ordenamiento de  $n$  registros, cada uno con una llave de subcampo  $k$ .

La idea básica subyacente en el ordenamiento de burbuja es pasar a través del archivo varias veces en forma secuencial. Cada paso consiste en la comparación de cada elemento en el archivo con su sucesor ( $x[i]$  con  $x[i + 1]$ ) y el intercambio de los dos elementos si no están en el orden correcto. Considérese el siguiente archivo:

25 57 48 37 12 92 86 33

En el primer paso se hacen las siguientes comparaciones:

|                   |             |                |
|-------------------|-------------|----------------|
| $x[0]$ con $x[1]$ | (25 con 57) | no intercambio |
| $x[1]$ con $x[2]$ | (57 con 48) | intercambio    |
| $x[2]$ con $x[3]$ | (57 con 37) | intercambio    |
| $x[3]$ con $x[4]$ | (57 con 12) | intercambio    |
| $x[4]$ con $x[5]$ | (57 con 92) | no intercambio |
| $x[5]$ con $x[6]$ | (92 con 86) | intercambio    |
| $x[6]$ con $x[7]$ | (92 con 33) | intercambio    |

Así, después del primer paso, el archivo está en el siguiente orden:

25 48 37 12 57 86 33 92

Obsérvese que después del primer paso, el elemento mayor (en este caso 92) está en la posición correcta dentro del arreglo. En general,  $x[n - i]$  estará en su posición adecuada después de la iteración  $i$ . El método se llama ordenamiento de burbuja porque cada número “burbujea” con lentitud hacia su posición correcta. Despues del segundo paso el archivo será:

25 37 12 48 57 33 86 92

Obsérvese que ahora 86 encontró su lugar en la segunda posición mayor. Como cada iteración coloca un nuevo elemento en su posición correcta, un archivo de  $n$  elementos requiere de no más de  $n - 1$  iteraciones.

El conjunto completo de iteraciones es el siguiente:

|                                |    |    |    |    |    |    |    |    |
|--------------------------------|----|----|----|----|----|----|----|----|
| iteración 0 (archivo original) | 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| iteración 1                    | 25 | 48 | 37 | 12 | 57 | 86 | 33 | 92 |
| iteración 2                    | 25 | 37 | 12 | 48 | 57 | 33 | 86 | 92 |
| iteración 3                    | 25 | 12 | 37 | 48 | 33 | 57 | 86 | 92 |
| iteración 4                    | 12 | 25 | 37 | 33 | 48 | 57 | 86 | 92 |
| iteración 5                    | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |

|             |    |    |    |    |    |    |    |    |
|-------------|----|----|----|----|----|----|----|----|
| iteración 6 | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |
| iteración 7 | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |

Sobre la base de la discusión anterior podríamos proceder a codificar el ordenamiento de burbuja. Sin embargo, existen modificaciones obvias para perfeccionar el método anterior. Primero, como los elementos en posiciones mayores o iguales a  $n - i$  están en la posición adecuada después de la iteración  $i$ , no deben ser considerados en las iteraciones siguientes. Así, en el primer paso se hacen  $n - 1$  comparaciones, en el segundo  $n - 2$  y en el  $(n - 1)$ -ésimo sólo una comparación (entre  $x[0]$  y  $x[1]$ ). En consecuencia, el proceso se agiliza a medida que avanza a lo largo de pasos sucesivos.

Hemos mostrado que son suficientes  $n - 1$  pasos para ordenar un archivo de tamaño  $n$ . Sin embargo, en el ejemplo precedente el archivo muestra de ocho elementos fue ordenado tras cinco iteraciones, haciendo innecesarias las dos últimas. Para eliminar pasos innecesarios debemos ser capaces de detectar el hecho de que el archivo ya esté ordenado. Pero eso es una tarea sencilla, dado que en un archivo ordenado no se hacen intercambios en ningún paso. Llevando el registro de si fueron hechos o no intercambios en un paso dado, puede determinarse si son necesarios pasos futuros. En este método, el paso final no hace intercambios si el archivo puede ordenarse en menos de  $n - 1$  pasos.

Usando estas mejoras, presentamos una rutina bubble (burbuja) que acepta dos variables  $x$  y  $n$ .  $x$  es un arreglo de números y  $n$  un entero que representa el número de elementos que deben ser ordenados ( $n$  puede ser menor que el número de elementos de  $x$ ).

```

bubble (x, n)
int x[], n;
{
 int hold, j, pass;
 int switched = TRUE;
 /* El ciclo externo controla el número de pasos */
 for (pass = 0; pass < n-1 && switched == TRUE; pass++) {
 switched = FALSE; /* Al principio no se han
 realizado intercambios en este paso */
 /* El ciclo interno maneja los pasos individuales */
 for (j = 0; j < n-pass-1; j++)
 if (x[j] > x[j+1]) {
 /* se intercambian los elementos
 no ordenados en caso de necesidad */
 switched = TRUE;
 hold = x[j];
 x[j] = x[j+1];
 x[j+1] = hold;
 } /* fin de if */
 } /* fin de for */
} /* fin de bubble */

```

¿Qué se puede decir acerca de la eficiencia del ordenamiento de burbuja? En el caso del ordenamiento que no incluye las dos mejoras esbozadas antes, el análisis es simple. Hay  $n - 1$  pasos y  $n - 1$  comparaciones en cada uno de ellos. Así, el número total de comparaciones es  $(n - 1) * (n - 1) = n^2 - 2n + 1$ , que es  $O(n^2)$ . Por supuesto, el número de intercambios depende del orden original del archivo. Sin embargo, el número de intercambios no puede ser mayor que el número de comparaciones. Es probable que sea el número de intercambios y no el de comparaciones el que más tiempo consuma en la ejecución del programa.

Veamos cómo afectan las mejoras introducidas la velocidad del ordenamiento de burbuja. El número de comparaciones en la iteración  $i$  es  $n - i$ . Así, si hay  $k$  iteraciones el número total de comparaciones será  $(n - 1) + (n - 2) + (n - 3) + \dots + (n - k)$ , que es igual a  $(2kn - k^2 - k)/2$ . Se puede mostrar que el número promedio de iteraciones,  $k$ , es  $O(n)$ , de manera que la fórmula completa sigue siendo  $O(n^2)$ , aunque el multiplicador constante es menor que antes. Sin embargo, hay una sobrecarga adicional ocasionada por la prueba y la inicialización de la variable *switched* (una por paso) y la asignación a la misma del valor *TRUE* (una vez por intercambio).

La única característica redentora del ordenamiento de burbuja, es que requiere de poco espacio adicional (un registro adicional para guardar el valor temporal para el intercambio y algunas variables enteras simples) y que es  $O(n)$  en el caso de un archivo ordenado en su totalidad (o casi ordenado en su totalidad). Esto se concluye de observar que sólo es necesario un paso de  $n - 1$  comparaciones (y ningún intercambio), para establecer que un archivo ordenado lo está.

Hay otras maneras de perfeccionar el ordenamiento de burbuja. Una de ellas es observar que el número de pasos necesarios para ordenar un archivo es la distancia más larga a la que un número debe moverse "hacia abajo" en el arreglo. En nuestro ejemplo, el número 33, que comienza en la posición 7 del arreglo encuentra al final la posición 2, después de cinco iteraciones. El ordenamiento de burbuja puede acelerarse haciendo que pasos sucesivos tomen direcciones opuestas de manera que los elementos menores se muevan con rapidez al frente del archivo en la misma forma que lo hacen los mayores hacia la parte posterior. Esto reduce el número requerido de pasos. Esta versión se deja como ejercicio al lector.

### Quicksort

El siguiente ordenamiento que consideramos es el *ordenamiento por intercambio de partición* (o *quicksort*). Sea  $x$  un arreglo y  $n$  el número de elementos en el arreglo que debe ser ordenado. Elegir un elemento  $a$  de una posición específica en el arreglo (por ejemplo,  $a$  puede elegirse como el primer elemento tal que  $a = x[0]$ ). Suponer que los elementos de  $x$  están separados de manera que  $a$  está colocado en la posición  $j$  y se cumplan las siguientes condiciones:

1. Cada uno de los elementos en las posiciones de 0 a  $j - 1$  es menor o igual que  $a$ .
2. Cada uno de los elementos en las posiciones  $j + 1$  a  $n - 1$  es mayor o igual que  $a$ .

Obsérvese que si se cumplen esas dos condiciones para una  $a$  y  $j$  particulares,  $a$  es el  $j$ -ésimo menor elemento de  $x$ , de manera que  $a$  se mantiene en la posición  $j$  cuando el arreglo está ordenado en su totalidad. (Demostrar este hecho como ejercicio.) Si se repite el proceso anterior con los subarreglos que van de  $x[0]$  a  $x[j - 1]$  y de  $x[j + 1]$  a  $x[n - 1]$  y con todos los subarreglos creados mediante el proceso en iteraciones sucesivas, el resultado final será un archivo ordenado.

Ilustremos el quicksort con un ejemplo. Si un arreglo inicial está dado por

25 57 48 37 12 92 86 33

Al principio, el primer elemento (25) se coloca en su posición correcta, el arreglo resultante es

12 25 57 48 37 92 86 33

En este punto, 25 está en la posición que le corresponde en el arreglo ( $x[1]$ ; cada elemento debajo de esa posición (12) es menor o igual a 25, y cada elemento arriba de esa posición (57, 48, 37, 92, 86 y 33) es mayor o igual que 25. Como 25 está en su posición final, el problema original se descompuso en el problema de ordenar los dos subarreglos

(12) y (57, 48, 37, 92, 86, 33)

No hay que hacer nada para ordenar el primero de esos subarreglos; un archivo de un elemento ya está ordenado. Para ordenar el segundo subarreglo se repite el proceso y se vuelve a subdividir el subarreglo. El arreglo completo puede ser visto ahora como

12 25 (48, 37, 33) 57 (92, 86)

y las repeticiones posteriores conducen a

|    |    |      |     |    |    |      |     |
|----|----|------|-----|----|----|------|-----|
| 12 | 25 | (37  | 33) | 48 | 57 | (92  | 86) |
| 12 | 25 | (33) | 37  | 48 | 57 | (92  | 86) |
| 12 | 25 | 33   | 37  | 48 | 57 | (92  | 86) |
| 12 | 25 | 33   | 37  | 48 | 57 | (86) | 92  |
| 12 | 25 | 33   | 37  | 48 | 57 | 86   | 92  |

Obsérvese que el último arreglo está ordenado.

En este momento se debe haber notado que el "quicksort" puede ser definido de manera conveniente como un procedimiento recursivo. Queremos esbozar un algoritmo  $quick(lb, ub)$  para ordenar todos los elementos de un arreglo  $x$  que están entre las posiciones  $lb$  y  $ub$  ( $lb$  es el límite inferior del arreglo y  $ub$  el superior) como sigue:

```

if (lb >= ub)
 return;
 /* ... */
partition(x, lb, ub, j); /* ... */
 /*
 * Dividir los elementos del
 * subarreglo de tal manera
 * que uno de ellos (posiblemente
 * x[lb]) esté ahora en x[j]
 * (j es un parámetro de salida), y:
 * 1. x[i] <= x[j] para lb <= i < j
 * 2. x[i] >= x[j] para j < i <= ub
 * x[j] está ahora en su posición
 * final
 */
 /* Ordenar el subarreglo en forma
 * recursiva entre las posiciones lb y j - 1
 */
 /* Ordenar el subarreglo en forma
 * recursiva entre las posiciones j + 1 y ub
 */

```

Ahora hay dos problemas. Tenemos que producir un mecanismo para implantar la *partición* y un método para implantar el proceso completo de manera no recursiva.

El objeto de la *partición* es permitir a un elemento específico encontrar su posición correspondiente con respecto a los otros en el subarreglo. Nótese que la manera en que se ejecuta esta partición es irrelevante para el método de ordenamiento. Todo lo que requiere el ordenamiento es que los elementos estén particionados de manera apropiada. En el ejemplo anterior, los elementos en cada uno de los dos subarchivos se mantienen en el mismo orden relativo en el que aparecen en el archivo original. Sin embargo, un método de particionar como ese es un tanto ineficiente para implantarse.

La manera de efectuar la partición de modo eficiente es la siguiente: Sea  $a = x[lb]$  el elemento cuya posición final se busca. (No se gana eficiencia apreciable seleccionando el primer elemento del subarreglo como aquel que está insertado dentro de su posición adecuada; lo único que esto hace es lograr que la escritura de algunos programas sea más fácil.) Se inicializan dos apuntadores *up* y *down* como el límite superior e inferior del subarreglo, respectivamente. En cualquier punto de la ejecución, cada elemento en una posición arriba de *up* es mayor o igual que  $a$  y cada elemento en una posición debajo de *down* es menor o igual que  $a$ . Los apuntadores *up* y *down* se mueven uno hacia otro de la siguiente manera.

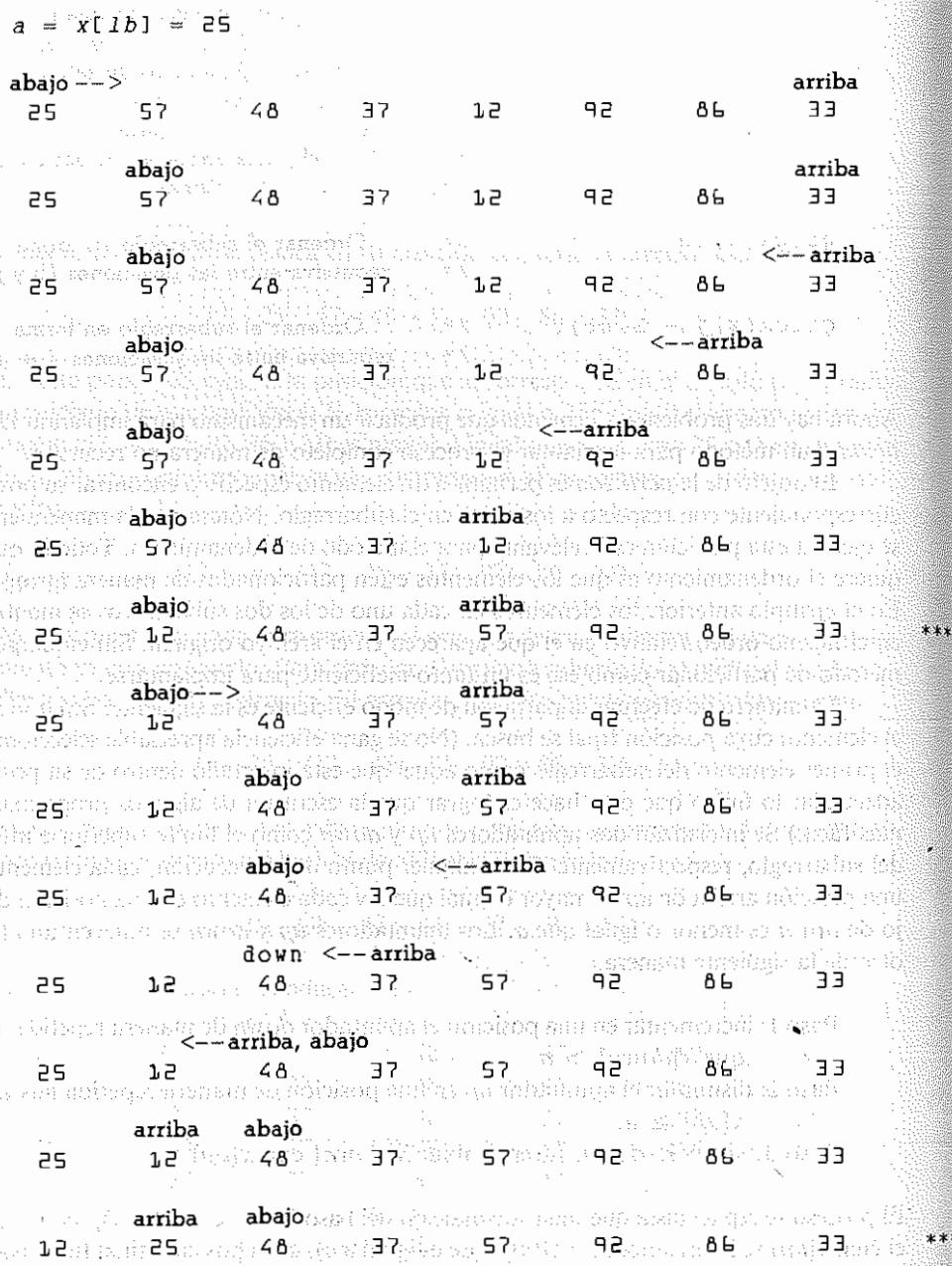
Paso 1: incrementar en una posición el apuntador *down* de manera repetida hasta que  $x[down] > a$ .

Paso 2: disminuir el apuntador *up* en una posición de manera repetida hasta que  $x[up] \geq a$ .

Paso 3: si  $up < down$ , intercambiar  $x[down]$  con  $x[up]$ .

El proceso se repite hasta que falla la condición del paso 3 ( $up <= down$ ), momento en el cual  $x[up]$  se intercambia con  $x[lb]$  (que es igual a  $a$ ), cuya posición final fue buscada, y se hace  $j$  igual a  $up$ .

Ilustramos este proceso con el archivo muestra, mostrando las posiciones de *up* y *down* cuando son ajustadas. La dirección en la que se examina el archivo está indicada por medio de una flecha en el apuntador que se está moviendo. Tres asteriscos en una línea indican que se está realizando un intercambio.



En este punto, 25 está en la posición adecuada (posición 1), todo elemento a su izquierda es menor o igual que él y todo elemento a su derecha es mayor o igual que él. Podemos proceder ahora a ordenar los dos subarreglos (12) y (48 37 57 92 86 33) aplicando el mismo método.

Este algoritmo particular puede implantarse mediante el siguiente procedimiento.

```

partition (x, lb, ub, pj)
int x[], lb, ub, *pj;
{
 int a, down, temp, up;

 a = x[lb]; /* a es el elemento cuya posición final se busca */
 up = ub;
 down = lb;
 while (down < up) {
 while (x[down] <= a && down < ub)
 down++; /* recorrer el límite superior del arreglo */
 while (x[up] > a)
 up--; /* recorrer el límite superior del arreglo */
 if (down < up) {
 /* intercambiar x[down] y x[up] */
 temp = x[down];
 x[down] = x[up];
 x[up] = temp;
 } /* fin de if */
 } /* fin de while */
 x[lb] = x[up];
 x[up] = a;
 *pj = up;
} /* fin de partition */

```

Obsérvese que si *k* es igual a *ub* — *lb* + 1 de tal manera que estemos rearreglando un subarreglo de tamaño *k*, la rutina usa *k* comparaciones de llaves (de *x[down]* con *a* y *x[up]* con *a*) para ejecutar la partición.

La rutina se puede hacer un tanto más eficiente eliminando algunas de las verificaciones redundantes. Se deja como ejercicio hacerlo.

Aunque el algoritmo recursivo para el quicksort es relativamente claro en términos de qué lleva a cabo y cómo lo hace, es deseable evitar la sobrecarga de llamadas a rutinas en programas como los de ordenamiento, en los cuales la eficiencia de la ejecución es una consideración significativa. Las llamadas recursivas a *quick* pueden eliminarse en forma fácil usando una pila como en la sección 3.4. Una vez que ha sido ejecutada *partition*, ya no se necesitan los parámetros actuales de *quick*, excepto para calcular los argumentos de las dos llamadas recursivas siguientes. Así, en lugar de poner en la pila los parámetros actuales en cada llamada recursiva, podemos calcular y poner en la pila los nuevos parámetros para cada una de las dos llamadas recursivas. Bajo este enfoque, la pila contiene en cada punto los límites superior e inferior de todos los subarreglos

que aún tienen que ser ordenados. Además, como la segunda llamada recursiva precede de inmediato al regreso al programa de llamada (como en el problema de las Torres de Hanoi) puede ser eliminada por completo y remplazada por un salto. Por último, como el orden en que se hacen las dos llamadas recursivas no afecta la corrección del algoritmo, elegimos en cada caso poner en la pila el subarreglo mayor y procesar de inmediato el más corto. Como explicamos, esta técnica mantiene el tamaño de la pila al mínimo.

Ahora podemos codificar un programa para implantar el quicksort. Como en el caso de *bubble*, los parámetros son el arreglo *x* y el número de elementos de *x* que deseamos ordenar, *n*. La rutina *push* pone en la pila a *lb* y *ub*, *popsub* los elimina de la misma y *empty* determina si la pila está vacía.

```
#define MAXSTACK... /* tamaño máximo de la pila */
quicksort(x, n)
int x[], n;
{
 int i, j;
 struct bndtype {
 int lb;
 int ub;
 } newbnds;
 /* ... La pila es utilizada por las funciones pop, push y empty */
 struct {
 int top;
 struct bndtype bounds[MAXSTACK];
 } stack;
 stack.top = -1;
 newbnds.lb = 0;
 newbnds.ub = n-1;
 push (&stack, &newbnds);
 /* Repetir mientras exista algún subarreglo */
 /* desordenado en la pila */
 while (!empty(&stack)) {
 popsub(&stack, &newbnds);
 while (newbnds.ub > newbnds.lb) {
 /* procesamiento del siguiente subarreglo */
 partition(x, newbnds.lb, newbnds.ub, &j);
 /* colocar en la pila el subarreglo mayor */
 if (j-newbnds.lb > newbnds.ub-j) {
 /* colocar en la pila el subarreglo inferior */
 i = newbnds.ub;
 newbnds.ub = j-1;
 push(&stack, &newbnds);
 /* procesar el subarreglo superior */
 newbnds.lb = j+1;
 newbnds.ub = i;
 }
 else {
 /* colocar en la pila el subarreglo superior */
 i = newbnds.lb;
```

```
newbnds.lb = j+1;
push(&stack, &newbnds);
/* procesar al subarreglo inferior */
newbnds.lb = i;
newbnds.ub = j-1;
}
/* fin de if */
}
/* fin de while */
}
/* fin de quicksort */
```

Para mayor eficiencia, las rutinas *empty*, *partition*, *popsub* y *push* deben insertarse en línea. Seguir la acción de *quicksort* sobre el archivo muestra,

Nótese que elegimos usar *x[lb]* como el elemento alrededor del cual particionar cada subarchivo por conveniencia en el procedimiento *partición*, pero cualquier otro elemento podría haber sido escogido. El elemento alrededor del cual realizamos la partición se llama *pivot*. Incluso no es necesario que *pivot* sea un elemento del subarchivo; *partición* puede escribirse con el encabezamiento

```
partition(lb, ub, x, j, pivot)
```

para particionar los elementos desde *x[lb]* hasta *x[ub]* de manera que todos los elementos entre *x[lb]* y *x[j-1]* sean menores que *pivot* y todos los elementos entre *x[j]* y *x[ub]* mayores o iguales a *pivot*. En ese caso, el propio elemento *x[j]* está incluido en el segundo subarchivo (dado que no está necesariamente incluido en la posición que le corresponde), de modo que la segunda llamada recursiva a *quick* es *quick(x, j + 1, ub)* en lugar de *quick(x, j + 1, ub)*.

Han sido encontradas varias maneras de elegir el valor de *pivot* para mejorar la eficiencia del quicksort garantizando subarchivos más balanceados. La primer técnica que se usa es la mediana del primero, último y elemento medio del subarchivo que debe ser ordenado (es decir, la mediana de *x[lb]*, *x[ub]* y *x[lb + ub]/2*) como valor del pivot. Esta mediana de tres valores está más cerca de la mediana del subarchivo que está siendo particionado que *x[lb]*, de manera que los dos subarchivos resultantes de la partición se asemejan más en cuanto a tamaño. En este método el valor de pivot es un elemento del archivo, de manera que *quick(x, j + 1, ub)* puede usarse como la segunda llamada recursiva.

Un segundo método, llamado *ordenamiento* por promedios utiliza *x[lb]* o la media de tres elementos como pivot cuando se partitiona el archivo original, pero añade el código en *partition* para calcular el término medio (los promedios) de los dos subarchivos que se están creando. En las particiones siguientes, se usan como un valor de pivot los promedios de cada subarchivo que fueron calculados cuando se crearon los mismos. Una vez más, este promedio está más cerca de la media del subarchivo que *x[lb]* y da como resultado archivos más balanceados. El promedio no es por fuerza un elemento del archivo, de manera que *quick(x, j, ub)* tiene que usarse como la segunda llamada recursiva. El código para encontrar el promedio no requiere de comparaciones de claves adicionales pero agrega alguna sobrecarga extra.

Otra técnica, llamada *Bsort*, usa como pivot el elemento medio de un subarchivo. Durante la partición se intercambian *x[up]* con *x[up + 1]* si *x[up] > x[up + 1]*,

cuando se disminuye el valor del apuntador *up*. Siempre que aumente el valor del apuntador *down*, se intercambia  $x[\text{down}]$  con  $x[\text{down} - 1]$  si  $x[\text{down}] < x[\text{down} - 1]$ . Siempre que se intercambien  $x[\text{up}]$  y  $x[\text{down}]$  se intercambian  $x[\text{up}]$  con  $x[\text{up} + 1]$  si  $x[\text{up}] > x[\text{up} + 1]$  y  $x[\text{down}]$  se intercambia con  $x[\text{down} - 1]$  si  $x[\text{down}] < x[\text{down} - 1]$ . Esto garantiza que  $x[\text{up}]$  sea siempre el elemento menor en el subarchivo derecho (de  $x[\text{up}]$  a  $x[\text{ub}]$ ) y que  $x[\text{down}]$  sea siempre el elemento mayor en el subarchivo izquierdo (de  $x[\text{lb}]$  a  $x[\text{down}]$ ).

Esto permite dos optimizaciones: Si no se requirieron intercambios entre  $x[\text{up}]$  y  $x[\text{up} + 1]$  durante la partición, se sabe que el subarchivo derecho estaba ordenado y no necesita ser puesto en la pila y si no se requirieron intercambios entre  $x[\text{down}]$  y  $x[\text{down} - 1]$  es el subarchivo izquierdo el que estaba ordenado y no necesita ser puesto en la pila. Lo anterior es similar a la técnica de mantener una etiqueta en el ordenamiento de burbuja para detectar que no ocurrieron intercambios durante un paso completo y que no son necesarios pasos adicionales. Segundo, se sabe que un subarchivo de tamaño 2 está ordenado y no tiene que ser puesto en la pila. Un subarchivo de tamaño 3 puede ser ordenado de manera directa con una sola comparación y un posible intercambio (entre los dos primeros elementos en un subarchivo izquierdo y entre los dos últimos en un subarchivo derecho). Ambas optimizaciones en el Bsort reducen el número de subarchivos que deben procesarse.

### Eficiencia del quicksort

¿Cuán eficiente es el quicksort? Supóngase que el tamaño del archivo,  $n$ , es una potencia de 2 o sea  $n = 2^m$ , de manera que  $m = \log_2 n$ . Supóngase también que la posición adecuada del pivot vienen a ser siempre la mitad exacta del subarreglo. En ese caso se harán más o menos  $n$  comparaciones en el primer paso (en realidad  $n - 1$ ), después del cual el archivo estará dividido en dos subarchivos cada uno de tamaño  $n/2$ , aproximadamente. Para cada uno de esos dos subarchivos se harán  $n/2$  comparaciones y se formarán un total de cuatro archivos cada uno de tamaño  $n/4$ . Cada uno de esos archivos requerirá de  $n/4$  comparaciones produciendo un total de  $n/8$  subarchivos. Después de dividir los subarchivos  $m$  veces, hay  $n$  archivos de tamaño 1. Así, el número total de comparaciones para el ordenamiento completo será muy próximo a:

$$n + 2 * (n/2) + 4 * (n/4) + 8 * (n/8) + \dots + n * (n/n)$$

O lo que es lo mismo, el resultado es:

$$n + n + n + n + \dots + n \quad (m \text{ terms})$$

comparaciones. Hay  $m$  términos porque el archivo se subdivide  $m$  veces. Así, el número total de comparaciones es  $O(n * m)$  o  $O(n \log n)$  recuérdese que  $m = \log_2 n$ . De esta manera, si el archivo satisface las propiedades descritas, el quicksort es  $O(n \log n)$ , lo que es de relativa eficiencia.

Para el quicksort no modificado en el cual se usa como valor del pivot  $x[\text{lb}]$  este análisis supone que el arreglo original y todos los subarreglos resultantes están desordenados, de manera que el valor del pivot  $x[\text{lb}]$  encuentra siempre su posición

adecuada en el medio del subarreglo. Supóngase que no se cumplen las condiciones anteriores y que el arreglo original está ordenado (o casi ordenado). Si, por ejemplo,  $x[\text{lb}]$  está en la posición correcta, el archivo original se divide en dos partes de tamaño 0 y  $n - 1$ . Si este proceso continúa, se ordena un total de  $n - 1$  subarchivos, el primero de tamaño  $n$ , el segundo de tamaño  $n - 1$ , el tercero de tamaño  $n - 2$  y así sucesivamente. Suponiendo  $k$  comparaciones para redisponer un archivo de tamaño  $k$ , el número total de comparaciones para ordenar el archivo completo es

$$n + (n - 1) + (n - 2) + \dots + 2$$

que es  $O(n^2)$ . De manera similar, si se ordena el archivo original en orden descendente la posición final de  $x[\text{lb}]$  es  $ub$  y se vuelve a dividir el archivo en dos subarchivos muy desbalanceados (tamaño  $n - 1$  y 0). Así, el quicksort no modificado tiene la propiedad al parecer absurda de trabajar mejor para archivos "completamente desordenados" que para archivos completamente ordenados. La situación es precisamente la opuesta para el ordenamiento de burbuja, la cual trabaja mejor para archivos ordenados y peor para archivos desordenados.

Es posible acelerar el quicksort para archivos ordenados eligiendo un elemento *aleatorio* para cada subarchivo como valor del pivot. Si se sabe que un archivo está casi ordenado, esto podría ser una buena estrategia (aunque, en ese caso, escoger el elemento medio como pivot sería aún mejor). Sin embargo, si no se sabe nada acerca del archivo, una estrategia de ese tipo no modifica el comportamiento del peor caso, dado que es posible (aunque improbable) que el elemento aleatorio escogido cada vez sea el elemento menor de cada subarchivo. Como cosa práctica, son más comunes los archivos ordenados que un buen generador de números aleatorios que de casualidad elija en forma repetida el elemento menor.

El análisis para el caso en el cual el tamaño del archivo no es una potencia de 2 es similar pero un poco más complejo; los resultados, sin embargo, permanecen iguales. No obstante, se puede mostrar que, en promedio (sobre todos los archivos de tamaño  $n$ ), el quicksort hace más o menos  $1.386 n \log_2 n$  comparaciones, incluso en su versión no modificada. En situaciones prácticas, el quicksort es con frecuencia el ordenamiento disponible más rápido a causa de su pequeña sobrecarga y su comportamiento  $O(n \log n)$  promedio.

Si se usa la técnica de la mediana de tres elementos, el quicksort puede ser  $O(n \log n)$  aun si el archivo está ordenado (suponiendo que la *partition* deja los subarchivos ordenados). Sin embargo, hay archivos patológicos en los cuales los elementos medio, primero y último de cada subarchivo son siempre los tres menores o mayores. En tales casos, el quicksort sigue siendo  $O(n^2)$ . Por fortuna dichos casos son raros.

El ordenamiento por promedios es  $O(n \log n)$  siempre que los elementos del archivo estén distribuidos de manera uniforme entre el mayor y el menor. Una vez más, distribuciones raras pueden hacerlo  $O(n^2)$  pero esto es menos probable que el peor de los otros métodos. Para archivos aleatorios el ordenamiento por promedios no ofrece ninguna reducción considerable en los intercambios o comparaciones en re-

lación con el quicksort estándar. Su sobrecarga, al calcular el promedio, demanda mucho más tiempo de CPU que el quicksort estándar. Para un archivo que se sabe está casi ordenado, el ordenamiento por promedios proporciona una reducción significativa de los intercambios y comparaciones. Sin embargo, la sobrecarga en el cálculo del promedio lo hace más lento que el quicksort, a menos que el archivo esté ordenado casi por completo.

El Bsort requiere mucho menos tiempo que el quicksort o el ordenamiento por promedios sobre archivos de entrada ordenados o casi ordenados, aunque requiere de más comparaciones e intercambios que el ordenamiento por promedios para entradas casi ordenadas (pero el ordenamiento por promedios tiene una sobrecarga significativa para encontrar el promedio). Se requiere de menos comparaciones pero más intercambios que el ordenamiento por promedios y más intercambios y comparaciones que el quicksort para entradas ordenadas de manera aleatoria. Sin embargo sus requerimientos de CPU son mucho menores que los del ordenamiento por promedios, aunque algo mayores que los del quicksort para entradas aleatorias.

Así, el Bsort se puede recomendar si se sabe que la entrada está casi ordenada o si estamos dispuestos a aceptar incrementos moderados en el tiempo de ordenamiento promedio para evitar incrementos muy grandes en el tiempo de ordenamiento del peor caso. Se puede recomendar el ordenamiento por promedios sólo en caso de entradas que se sabe están casi ordenadas y el quicksort estándar para entradas aleatorias probables o si el tiempo de ordenamiento promedio tiene que ser tan rápido como sea posible. En la sección 6.5 presentamos una técnica que es más rápida que el Bsort y el ordenamiento por promedios sobre archivos que estén casi ordenados.

Los requerimientos de espacio para el quicksort dependen del número de llamadas recursivas anidadas o del tamaño de la pila. Es claro que la pila no puede crecer nunca más allá del número de elementos que hay en el archivo original. El crecimiento de la pila por debajo de  $n$  depende del número de subarchivos generados y de sus tamaños. El tamaño de la pila se contiene un poco si siempre se pone en la misma el subarreglo más grande y se aplica la rutina al más pequeño de los dos. Esto garantiza que todos los subarreglos que resultaron más pequeños sean subdivididos primero que los que resultaron más grandes dando el efecto neto de tener menos elementos en la pila en cualquier momento dado. La razón para esto es que un subarreglo más pequeño, será dividido menos veces que uno más grande. Por supuesto, el subarreglo más largo será procesado y subdividido al final, pero esto ocurrirá después que los subarreglos pequeños hayan sido ordenados y en consecuencia eliminados de la pila.

Otra ventaja del quicksort es la localidad de referencia. Es decir, en un corto periodo de tiempo todos los accesos en el arreglo se hacen en una de dos porciones un tanto pequeñas (un subarchivo o una porción del mismo). Esto asegura eficiencia en el ambiente de la memoria virtual, donde páginas de datos se intercambian de manera constante entre la memoria interna y la externa. La localidad de referencia resulta en menos requerimientos de cambios de páginas para un programa particular. Un estudio de simulación mostró que en tal ambiente, el quicksort usa menos recursos de espacio y tiempo que cualquier otro ordenamiento considerado.

## EJERCICIOS

- 6.2.1. Pruebe que el número de pasos necesario en el ordenamiento por burbuja del texto, antes de que el archivo esté ordenado (sin incluir el último paso, que detecta el hecho de que el archivo está ordenado), es igual a la distancia más larga que tiene que ser movido un elemento de un índice mayor a uno menor.
- 6.2.2. Vuelva a escribir la rutina *bubble* de manera que pasos sucesivos avancen en direcciones opuestas.
- 6.2.3. Pruebe que, en el ordenamiento del ejercicio previo, si dos elementos no son intercambiados durante dos pasos consecutivos en direcciones opuestas, ellos están en su posición final.
- 6.2.4. Un ordenamiento por *conteo* se ejecuta como sigue. Declárese un arreglo *count* y a *count[i]* igual al número de elementos que sean menores que *x[i]*. Después colóquese *x[i]* en la posición *count[i]* de un arreglo de salida. (Cúdese la posibilidad de elementos iguales.) Escribir una rutina para ordenar un arreglo *x* de tamaño *n* usando este método.
- 6.2.5. Suponga que un archivo contiene enteros entre *a* y *b* con muchos números repetidos varias veces. Un *ordenamiento por distribución* procede como sigue. Declare un arreglo *number* de tamaño *b - a + 1*, y a *number[i - a]* igual al número de veces que aparece el entero *i* en el archivo; después vuélvase a poner los valores en el archivo de acuerdo a lo anterior. Escribir una rutina para ordenar un arreglo *x* de tamaño *n* que contenga enteros entre *a* y *b* mediante este método.
- 6.2.6. El *ordenamiento por transposición par-impar* procede de la siguiente manera. Pase a través del archivo varias veces. En el primer paso, compare *x[i]* con *x[i + 1]* para todo *i* impar. En el segundo paso comparar *x[i]* con *x[i + 1]* para todo *i* par. Cada vez que *x[i] > x[i + 1]* intercambiarlas. Continuar alternando en esta forma hasta que el archivo esté ordenado.
  - a. ¿Cuál es la condición para culminar el ordenamiento?
  - b. Escriba una rutina en C para implantar el ordenamiento.
  - c. ¿Cuál es, en promedio, la eficiencia de este ordenamiento?
- 6.2.7. Vuelva a escribir el programa para el quicksort comenzando con el algoritmo recursivo y aplicando los métodos del capítulo 3 para producir una versión no recursiva.
- 6.2.8. Modifique el programa para el quicksort del texto de manera que si un subarreglo es pequeño, se use el ordenamiento de burbuja. Determine en forma empírica con la computadora cuán pequeño debería ser el subarreglo, de manera que esta estrategia mixta sea más eficiente que el quicksort ordinario.
- 6.2.9. Modifique *partition* de manera que el valor medio de  $x[lb], x[ub]$  y  $x[ind]$  (donde  $ind = (ub + lb)/2$ ) se use para partitionar el arreglo. ¿En cuáles casos está el quicksort usando este método más eficiente que la versión del texto? ¿En cuáles es menos eficiente?
- 6.2.10. Implante la técnica del ordenamiento por promedios. *partition* debe usar el promedio del subarchivo que se está partitionado, calculado cuando fue creado el subarchivo, como valor de pivot y debe calcular el promedio de cada uno de los dos subarchivos que crea. Cuando se ponen en la pila los límites superior e inferior de un subarchivo, también se debe poner su media.
- 6.2.11. Implante la técnica de Bsort. El elemento medio de cada archivo debe usarse como pivot, el último elemento del subarchivo izquierdo que se está creando debe mante-

nerse como el mayor en el subarchivo izquierdo y el primer elemento del subarchivo derecho debe mantenerse como el menor en el subarchivo derecho. Se deben usar dos bits para registrar cuál de los dos subarchivos está ordenado al final de partición. Un subarchivo ordenado no requiere ser procesado aún más. Si un subarchivo tiene 3 o menos elementos, se ordena enseguida por medio de un intercambio simple, cuando mucho.

- 6.2.12. a. Vuelva a escribir las rutinas para el ordenamiento de burbuja y el quicksort como se presentaron en el texto y para los ordenamientos de los ejercicios de manera que se lleve el registro del número real de comparaciones e intercambios hechos.
- b. Escriba un generador de números aleatorios (o use uno existente si la instalación lo tiene) que genere enteros entre 0 y 999.
- c. Usando el generador del inciso b, genere varios archivos de tamaño 10, 100 y 1000. Aplique las rutinas de ordenamiento de la parte a para medir los requerimientos de tiempo de cada uno de los ordenamientos aplicados a cada uno de los archivos.
- d. Mida los resultados del inciso c y compárelos con los valores teóricos presentados en esta sección. ¿Coinciden? Si no, explicar. En particular, redispone los archivos de manera que estén ordenados por completo y en orden inverso y ver cómo se comportan los ordenamientos con esas entradas.

### 6.3. ORDENAMIENTOS POR SELECCIÓN Y CON ARBOLES

Un *ordenamiento por selección* es uno en el cual se selecciona elementos sucesivos en orden y se colocan en sus posiciones correspondientes. Los elementos de la entrada pueden haber sido reprocesados para hacer posible la selección ordenada. Cualquier ordenamiento por selección puede conceptualizarse con el siguiente algoritmo general que usa una cola de prioridad descendente (recuérdese que *pqinsert* inserta el elemento mayor de una cola de prioridad y *pqmaxdelete* lo recupera).

```
Asignar dpq a la cola de prioridad descendente vacía;
/* preprocesar los elementos del arreglo de entrada */
/* insertándolos en la cola de prioridad */
for (i = 0; i < n; i++)
 pqinsert(dpq, x[i]);
/* Seleccionar cada elemento sucesivo en orden */
for (i = n - 1; i >= 0; i--)
 x[i] = pqmaxdelete(dpq);
```

Este algoritmo es llamado de *ordenamiento por selección general*.

Examinamos ahora varios ordenamientos por selección diferentes. Hay dos características que distinguen a un ordenamiento por selección específico. Una de ellas es la estructura de datos usada para implantar la cola de prioridad. La segunda es el método usado para implantar el algoritmo general. Una estructura de datos particular puede permitir una optimización significativa del algoritmo de ordenamiento por selección general.

Nótese también que el algoritmo general puede modificarse para usar una cola de prioridad ascendente *apq* en lugar de *dpq*. El segundo ciclo que implanta la fase de selección sería modificado de la siguiente manera:

```
for (i = 0; i < n; i++)
 x[i] = pqmindelete(apq);
```

#### Algoritmo por selección directa

El *ordenamiento por selección directa*, u *ordenamiento inverso*, implanta la cola de prioridad descendente como un arreglo desordenado. El arreglo de entrada *x* se usa para guardar la cola de prioridad, eliminando así la necesidad de espacio adicional. El ordenamiento por selección directa es, en consecuencia, un ordenamiento en el lugar. Más aún, como el arreglo de entrada *x* es el propio arreglo desordenado que representará la cola de prioridad descendente, la entrada ya está en el formato adecuado haciendo innecesaria la fase de preprocesamiento.

Por lo tanto el ordenamiento por selección directa consiste en su totalidad de una fase de selección en la cual el mayor de los elementos entre los restantes, *large*, se coloca de manera repetida en su posición correcta, *i*, al final del arreglo. Para hacer esto, se intercambian *large* y *x[i]*. La cola de prioridad con *n* elementos al inicio se reduce en un elemento después de cada selección. Tras *n* — 1 selecciones está ordenado el arreglo completo. Así, el proceso de selección tiene que hacerse de *n* — 1 a 1 en lugar de hasta 0. La siguiente función en C implanta la selección directa:

```
selectsort(x, n)
int x[], n;
{
 int i, indx, j, large;
 for (i = n - 1; i > 0; i--) {
 /* colocar el número mayor de entre x[0] hasta x[i] */
 /* en large y su índice en indx */
 large = x[0];
 indx = 0;
 for (j = 1; j <= i; j++)
 if (x[j] > large) {
 large = x[j];
 indx = j;
 } /* fin de for... if */
 x[indx] = x[i];
 x[i] = large;
 } /* fin de for */
} /* fin de selectsort */
```

El análisis de la selección directa es simple. En el primer paso se efectúan *n* — 1 comparaciones, en el segundo *n* — 2, y así sucesivamente. En consecuencia hay un total de

$$(n - 1) + (n - 2) + (n - 3) + \cdots + 1 = n * (n - 1)/2$$

comparaciones, que es  $O(n^2)$ . El número de intercambios es siempre *n* — 1 (a no ser que se agregue una verificación para prevenir el intercambio de un elemento consigo

mismo). Sólo se requiere un poco de memoria adicional (para guardar unas pocas variables temporales). El ordenamiento puede ser categorizado, en consecuencia, como  $O(n^2)$ . Aunque es más rápido que el ordenamiento de burbuja. No hay mejora si el archivo de entrada está desordenado u ordenado, dado que la prueba se completa sin considerar la composición del archivo. A pesar de ser fácil de codificar, es improbable que se use el ordenamiento por selección directa salvo en los archivos para los cuales  $n$  es pequeño.

También es posible implantar un ordenamiento representando la cola de prioridad descendente mediante un arreglo ordenado. De manera interesante, esto conduce a un ordenamiento que consta de una fase de preprocessamiento en la que se forma un arreglo ordenado de  $n$  elementos. La fase de selección es, en consecuencia superflua. Este ordenamiento se presenta en la sección 6.4 como el *ordenamiento por inserción simple*; no es un ordenamiento de selección porque no se requiere de esta última.

### Ordenamiento por árboles binarios

En el resto de esta sección ilustramos algunos métodos de ordenamiento por selección que representan una cola de prioridades mediante un árbol binario. El primer método es el *ordenamiento con árbol binario* de la sección 5.1 que usa un árbol de búsqueda binaria. Se aconseja al lector revisar dicha sección antes de proseguir.

El método implica el examen de la cada elemento del archivo de entrada y el colocar al mismo en la posición apropiada dentro de un árbol binario. Para encontrar la posición adecuada de un elemento  $y$ , se toma en cada nodo una rama izquierda o derecha dependiendo de si  $y$  es menor que el elemento que está en el nodo o mayor o igual a él. Una vez que cada elemento de entrada está en su posición adecuada en el árbol, se puede recuperar el archivo ordenado mediante un recorrido en orden del árbol. Presentamos un algoritmo para este ordenamiento, modificándolo para acomodar la entrada como un arreglo preexistente. Convertir el algoritmo a una rutina en C es sencillo.

```
/* establecer el primer elemento como raíz */
tree = maketree(x[0]);
/* repetir para cada elemento sucesivo */
for (i = 1; i < n; i++) {
 y = x[i];
 q = tree;
 p = q;
 /* recorra el árbol hacia abajo hasta alcanzar una hoja */
 while (p != null) {
 q = p;
 if (y < info(p))
 p = left(p);
 else
 p = right(p);
 } /* fin de while */
}
```

```
if (y < info(q))
 setleft(q,y);
else
 setright(q,y);
} /* fin de for */
/* Se ha construido el árbol recorrerlo en orden */
intrav(tree);
```

Para convertir el algoritmo en una rutina que ordene un arreglo es necesario revisar *intrav* de tal manera que en la visita a un nodo se coloquen los contenidos del mismo en la posición siguiente del arreglo original.

En realidad, el árbol de búsqueda binaria representa una cola de prioridad ascendente, como se describió en los ejercicios 5.1.13 y 5.2.13. La construcción del árbol representa la fase de preprocessamiento y el recorrido la fase de selección del algoritmo de ordenamiento por selección general.

Por lo regular, la extracción del elemento mínimo (*pqmindelete*) de una cola de prioridad, representada mediante un árbol de búsqueda binaria, implica descender por la parte izquierda del árbol desde la raíz. En realidad, ese es el primer paso del proceso en un recorrido en orden. Sin embargo, como una vez construido el árbol no se insertan en el mismo nuevos elementos y el elemento mínimo no tiene que ser eliminado en realidad, el recorrido en orden implica de manera eficiente el proceso de selección sucesiva.

La eficiencia relativa de este método depende del orden original de los datos. Si el arreglo original está ordenado por completo (u ordenado en orden inverso), el árbol resultante aparece como una secuencia sólo de ligas derechas (o izquierdas), como en la figura 6.3.1. En ese caso la inserción del primer nodo no requiere comparación alguna, el segundo nodo requiere de dos, el tercero de tres, y así sucesivamente. Así, el número total de comparaciones es

$$2 + 3 + \dots + n = n * (n + 1)/2 - 1$$

que es  $O(n^2)$ .

Por otra parte, si en el arreglo original los datos están organizados de tal manera que la mitad de los números anteriores a uno dado,  $a$ , en el arreglo sean menores que  $a$  y la otra mitad mayores que  $a$ , resultan árboles balanceados como los de la figura 6.3.2. En tal caso la profundidad del árbol binario resultante es el menor entero  $d$ , mayor o igual que  $\log_2(n + 1) - 1$ . El número de nodos en cualquier nivel  $l$  (con posible excepción del último) es  $2^l$  y el número de comparaciones necesarias para colocar un nodo en el nivel  $l$  (excepto cuando  $l = 0$ ) es  $l + 1$ . Así, el número total de comparaciones está entre

$$d + \sum_{l=1}^{d-1} 2^l * (l + 1) \text{ y } \sum_{l=1}^d 2^l * (l + 1)$$

Se puede mostrar (los lectores inclinados hacia las matemáticas podrían estar interesados en probar este hecho como ejercicio) que las sumas resultantes son  $O(n \log n)$ .

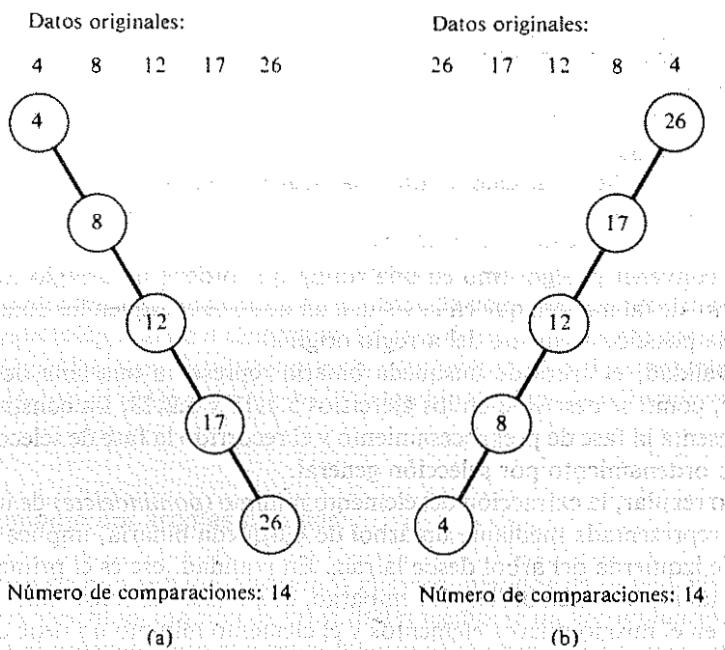


Figura 6.3.1

Por fortuna, se puede mostrar que si todo ordenamiento posible de la entrada se considera tan probable, resultan con mayor frecuencia árboles balanceados que no balanceados. El tiempo promedio de ordenamiento para un ordenamiento de árbol binario es por lo tanto  $O(n \log n)$ , aunque la constante de proporcionalidad es mayor en promedio que en el mejor de los casos. Sin embargo, en el peor caso (entrada ordenada), el ordenamiento de árbol binario es  $O(n^2)$ . Por supuesto, una vez que el árbol ha sido creado, se emplea tiempo en recorrerlo. Si se añaden hebras

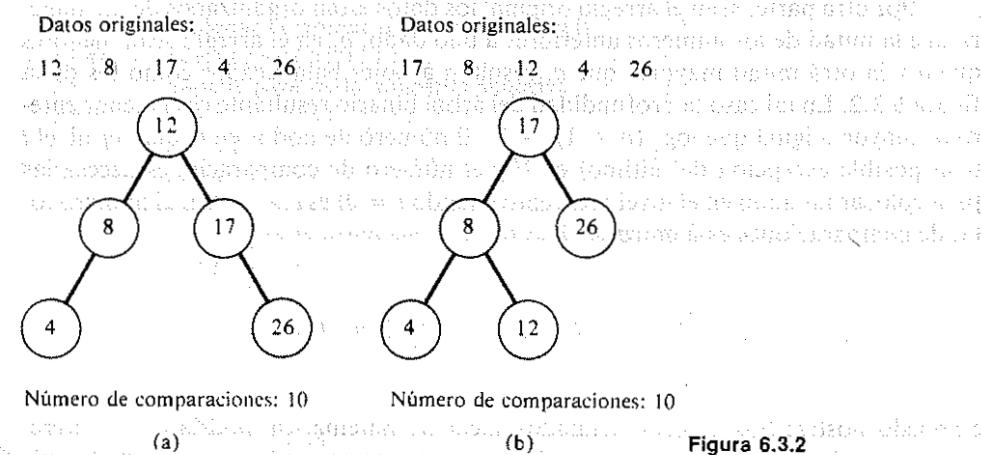


Figura 6.3.2

al árbol cuando éste se crea, se reduce el tiempo de recorrido y elimina la necesidad de una pila (implícita en la recursión o explícita en un recorrido en orden no recursivo).

Este ordenamiento requiere que sea reservado un nodo en el árbol para cada elemento del arreglo. Dependiendo del método usado para implantar el árbol, se puede requerir de espacio para apuntadores al mismo y hebras, si existen. Este requerimiento adicional de espacio, unido a la pobre eficiencia en cuanto a tiempo  $O(n^2)$  para entradas ordenadas o que estén en orden inverso, representa la desventaja primaria del ordenamiento por árbol binario.

### Heapsort

Las desventajas del ordenamiento de árbol binario pueden remediarse mediante el **heapsort**, un ordenamiento en el lugar que requiere de sólo  $O(n \log n)$  operaciones sin importar el orden de la entrada. Definir un **heap descendente** (también llamado **max heap** o **árbol parcialmente ordenado descendente**) de tamaño  $n$  como un árbol binario quasi-completo de  $n$  nodos tal que, el contenido de cada nodo sea menor o igual que el de su padre. Si se usa la representación secuencial de un árbol binario quasi-completo, esta condición se reduce a la desigualdad

$$\text{info}[j] \leq \text{info}[(j-1)/2] \quad \text{for } 0 \leq ((j-1)/2) < j \leq n - 1$$

De esta definición queda claro de un heap descendente que la raíz del árbol (o el primer elemento del arreglo) contiene el elemento mayor en el heap. Nótese también que cualquier camino de la raíz a una hoja (o de hecho cualquier camino en el árbol que incluya no más de un nodo en cada nivel) es una lista ordenada en orden descendente. También es posible definir un **heap ascendente** (o **min heap**) como un árbol binario quasi-completo tal que los contenidos de cada nodo sean mayores o iguales que los contenidos de su padre. En un heap ascendente, la raíz contiene el elemento menor del heap y cualquier camino de la raíz a una hoja es una lista ordenada de manera ascendente.

Un heap permite una implantación muy eficiente de una cola de prioridad. Recordar de la sección 4.2 que una lista ordenada que contenga  $n$  elementos permite que la inserción en una cola de prioridad (*pqinsert*) se implanta usando un promedio de accesos de nodos aproximado a  $n/2$  y la eliminación del máximo o mínimo (*pqmindelete* o *pqmaxdelete*) usando sólo un nodo de acceso. Así, una secuencia de  $n$  inserciones y  $n$  eliminaciones de una lista ordenada como la que necesita un ordenamiento por selección, requiere de  $O(n^2)$  operaciones. Aunque la inserción en una cola de prioridad usando un árbol de búsqueda binaria podría necesitar de sólo unos pocos accesos de nodo como  $\log_2 n$ , podría requerir hasta  $n$  accesos si el árbol está desbalanceado. Así, un ordenamiento por selección usando un árbol de búsqueda binaria podría requerir también  $O(n^2)$  operaciones, aunque en promedio sólo se necesitan  $O(n \log n)$ .

Como veremos, un heap permite que tanto la inserción como la eliminación se implanten en  $O(n \log n)$  operaciones. Así, un ordenamiento por selección que consista de  $n$  inserciones y  $n$  eliminaciones puede implantarse usando un heap en  $O(n$

$\log n$ ) operaciones, aun en el peor de los casos. Una ganancia adicional es que el propio heap se puede implantar dentro del arreglo de entrada  $x$ , usando la implantación secuencial de un árbol binario quasi-completo. El único espacio adicional requerido es para variables del programa. El heapsort es, por lo tanto, un ordenamiento  $O(n \log n)$  en el lugar.

### El heap como una cola de prioridad

Implantemos ahora una cola de prioridad descendente usando un heap descendente. Suponer que  $dpq$  es un arreglo que representa de manera implícita un heap descendente de tamaño  $k$ . Como la cola de prioridad está contenida en los elementos 0 a  $k - 1$  del arreglo, agregamos  $k$  como un parámetro de las operaciones de inserción y eliminación. Entonces la operación  $pqinsert(dpq, k, elt)$  puede implantarse insertando simplemente  $elt$  dentro de su posición correcta en la lista descendente formada por el camino que va de la raíz del  $heap(dpq[0])$  a la hoja  $dpq[k]$ . Una vez ejecutada  $pqinsert(dpq, k, elt)$ ,  $dpq$  se convierte en un heap de tamaño  $k + 1$ .

La inserción se hace recorriendo el camino desde la posición vacía  $k$  hasta la posición 0 (la raíz), buscando el primer elemento mayor o igual que  $elt$ . Cuando ha sido encontrado dicho elemento, se inserta  $elt$  precediéndolo de inmediato en la trayectoria (es decir  $elt$  se inserta como su hijo). Como cada elemento menor que  $elt$  se pasa durante el recorrido, se desvía un nivel hacia abajo en el árbol para hacer espacio a  $elt$ . (Este recorrimiento es necesario porque estamos usando la representación secuencial en lugar de la representación ligada del árbol. Un nuevo elemento no puede ser insertado entre dos elementos existentes sin recorrer algunos.)

Esta operación de inserción en el heap se llama también operación *siftup* dado que  $elt$  examina su camino hacia arriba en el árbol. El algoritmo siguiente implanta  $pqinsert(dpq, k, elt)$ :

```

s = k;
f = (s - 1)/2; /* f es el padre de s */
while (s > 0 && dpq[f] < elt) {
 dpq[s] = dpq[f];
 s = f; /* avanzar hacia arriba por el árbol */
 f = (s - 1)/2;
} /* fin de while */
dpq[s] = elt;

```

La inserción es, de manera clara,  $O(\log n)$  dado que un árbol binario quasi-completo con  $n$  nodos tiene  $\log_2 n + 1$  niveles y se acceden a lo sumo un nodo por nivel.

Examinamos ahora como implantar  $pqmaxdelete(dpq, k)$  para un heap descendente de tamaño  $k$ . Primero definimos  $subtree(p, m)$ , donde  $m$  es mayor que  $p$ , como el subárbol (del heap descendente) que tiene raíz en la posición  $p$  con los elementos  $dpq[p]$  a  $dpq[m]$ . Por ejemplo,  $subtree(3, 10)$  consta de la raíz  $dpq[3]$  y sus dos hijos  $dpq[7]$  y  $dpq[8]$ .  $subtree(3, 17)$  consta de  $dpq[3], dpq[7], dpq[8], dpq[15], dpq[16]$  y  $dpq[17]$ . Si  $dpq[i]$  está incluido en  $subtree(p, m)$ ,  $dpq[2 * i + 1]$  está

incluido si y sólo si  $2 * i + 1 <= m$  y  $dpq[2 * i + 2]$  está incluido si y sólo si  $2 * i + 2 = m$ . Si  $m$  es menor que  $p$ ,  $subtree(p, m)$  se define como el árbol vacío.

Para implantar  $pqmaxdelete(dpq, k)$ , notamos que el máximo de los elementos, está siempre en la raíz de un heap descendente de  $k$ -elementos. Cuando se elimina dicho elemento, los  $k - 1$  elementos restantes tienen que ser redistribuidos de las posiciones 1 a  $k - 1$  a las posiciones 0 a  $k - 2$  de manera que el segmento del arreglo resultante de  $dpq[0]$  a  $dpq[k - 2]$  siga siendo un heap descendente. Sea  $adjust-heap(root, k)$  la operación de reacomodar los elementos  $dpq[root + 1]$  a  $dpq[k]$  dentro de  $dpq[root]$  hasta  $dpq[k - 1]$  de manera que  $subtree(root, k - 1)$  forme un heap descendente. Entonces  $pqmaxdelete(dpq, k)$  para un heap descendente de  $k$ -elementos puede implantarse mediante:

```

p = dpq[0];
adjustheap(0, k - 1);
return(p);

```

En un heap descendente, no sólo la raíz es el elemento mayor del árbol, sino que un elemento en *cualquier* posición  $p$  tiene que ser el elemento mayor de  $subtree(p, k)$ . Ahora,  $subtree(p, k)$  consta de tres grupos de elementos: su raíz,  $dpq[p]$ ; su subárbol izquierdo,  $subtree(2 * p + 1, k)$ ; y su subárbol derecho,  $subtree(2 * p + 2, k)$ .  $dpq[2 * p + 1]$ , el hijo izquierdo de la raíz, es el elemento mayor del subárbol izquierdo y  $dpq[2 * p + 2]$ , el hijo derecho de la raíz, es el elemento mayor del subárbol derecho. Cuando la raíz  $dpq[p]$  se elimina, se mueve hacia arriba el elemento más grande de esos dos para tomar su lugar como el nuevo elemento mayor de  $subtree(p, k)$ . Entonces, el subárbol con raíz en la posición del elemento mayor movido hacia arriba tiene que ser reajustado a su vez.

Definamos  $largeson(p, m)$  como el hijo mayor de  $dpq[p]$  dentro de  $subtree(p, m)$ . Puede implantarse como:

```

s = 2 * p + 1;
if (s + 1 <= m && x[s] < x[s + 1])
 s = s + 1;
/* verificar si se está fuera de límites */
if (s > m)
 return(-1);
else
 return(s);

```

Entonces,  $adjustheap(root, k)$  puede implantarse de manera recursiva por medio de:

```

f = root;
s = largeson(f, k - 1);
if (s >= 0 && dpq[k] < dpq[s]) {
 dpq[f] = dpq[s];
 adjustheap(s, k);
}
else
 dpq[f] = dpq[k];

```

A continuación presentamos una versión iterativa de *adjustheap*. El algoritmo usa una variable temporal *kvalue* para guardar el valor de *dpq[k]*:

```

f = root;
kvalue = dpq[k];
s = largeson(f, k - 1);
while (s >= 0 && kvalue < dpq[s]) {
 dpq[f] = dpq[s];
 f = s;
 s = largeson(f, k - 1);
}
dpq[f] = kvalue;

```

Obsérvese que recorremos una trayectoria del árbol desde la raíz hacia una hoja recorriendo hacia arriba todos los elementos mayores que *dpq[k]* una posición e insertando *dpq[k]* en la posición correspondiente. De nuevo, el corrimiento es necesario porque estamos usando la representación secuencial en lugar de la implantación ligada de un árbol. El procedimiento de ajuste se llama con frecuencia operación *siftdown* porque *dpq[k]* examina su camino desde la raíz hacia abajo en el árbol.

Este algoritmo de eliminación en un heap también es  $O(\log n)$ , dado que hay  $\log_2 n + 1$  niveles en el árbol y se acceden a lo sumo dos nodos en cada nivel. Sin embargo, la sobrecarga por corrimiento y cálculo de *largeson* es significativa.

### Ordenamiento usando un heap

El heapsort no es más que una implantación del algoritmo de ordenamiento por selección general usando el arreglo de entrada *x* como un heap que represente una cola de prioridad descendente. La fase de reprocesamiento crea un heap de tamaño *n* usando la operación *siftdown* y la fase de selección redistribuye los elementos del heap en el orden en que los elimina de la cola de prioridad usando la operación *siftdown*. En ambos casos los ciclos no necesitan incluir el caso en que *i* es igual a 0, dado que *x[0]* es ya una cola de prioridad de un elemento y el arreglo está ordenado una vez que los elementos de *x[1]* hasta *x[n - 1]* están en las posiciones que les corresponden.

```

/* Crear la cola de prioridad; antes de cada interacción en el ciclo */
/* la cola de prioridad está formada por los elementos desde x[0] */
/* hasta x[i-1]. Cada iteración agrega a x[i] a la cola */
for (i = 1; i < n; i++)
 pqinsert(x, i, x[i]);
/* seleccionar cada elemento sucesivo en orden */
for (i = n - 1; i > 0; i--)
 x[i] = pqmaxdelete(x, i + 1);

```

La figura 6.3.3 ilustra la creación de un heap de tamaño 8 del archivo original.

25 57 48 37 12 92 86 33

Las líneas punteadas en esa figura indican un elemento que está recorrido hacia abajo en el árbol.

La figura 6.3.4 ilustra el ajuste del heap donde *x[0]* se selecciona de manera repetida, se coloca en la posición que le corresponde y en el arreglo y se reajusta el heap, hasta que todos los elementos heap son procesados. Notar que después de “eliminar” un elemento del heap, dicho elemento permanece en el arreglo; sólo que se ignora en el procesamiento subsecuente.

### El procedimiento heapsort

Presentamos ahora un procedimiento para heapsort con todos los subprocedimientos (*pqinsert*, *pqmaxdelete*, *adjustheap* y *largeson*) expandidos en línea e integrados para mayor eficiencia.

```

heapsort (x, n)
int x[], n;
{
 int i, elt, s, f, ivalue;
 /* etapa de procesamiento; crear el heap inicial */
 for (i = 1; i < n; i++) {
 elt = x[i];
 /* pqinsert(x, i, elt) */
 s = i;
 f = (s-1)/2;
 while (s > 0 && x[f] < elt) {
 x[s] = x[f];
 s = f;
 f = (s-1)/2;
 } /* fin de while */
 x[s] = elt;
 } /* fin de for */
 /* etapa de selección; eliminar de manera repetida a x[0],
 * insertarlo en la posición correcta y ajustar el heap */
 for (i = n-1; i > 0; i--) {
 /* pqmaxdelete(x, i+1) */
 ivalue = x[i];
 x[i] = x[0];
 f = 0;
 /* s = largeson (0, i-1) */
 if (i == 1)
 s = -1;
 else
 s = 1;
 if (i > 2 && x[2] > x[1])
 s = 2;
 while (s >= 0 && ivalue < x[s]) {
 x[f] = x[s];
 f = s;
 s = largeson(f, s-1);
 }
 }
}

```

```

/* s = largeson(f, i-1) */
s = 2*f+1;
if (s+1 <= i-1 && x[s] < x[s+1])
 s = s+1;
if (s > i-1)
 s = -1;
} /* fin de while */
x[f] = ivalue;
/* fin de for */
} /* fin de heapsort */

```

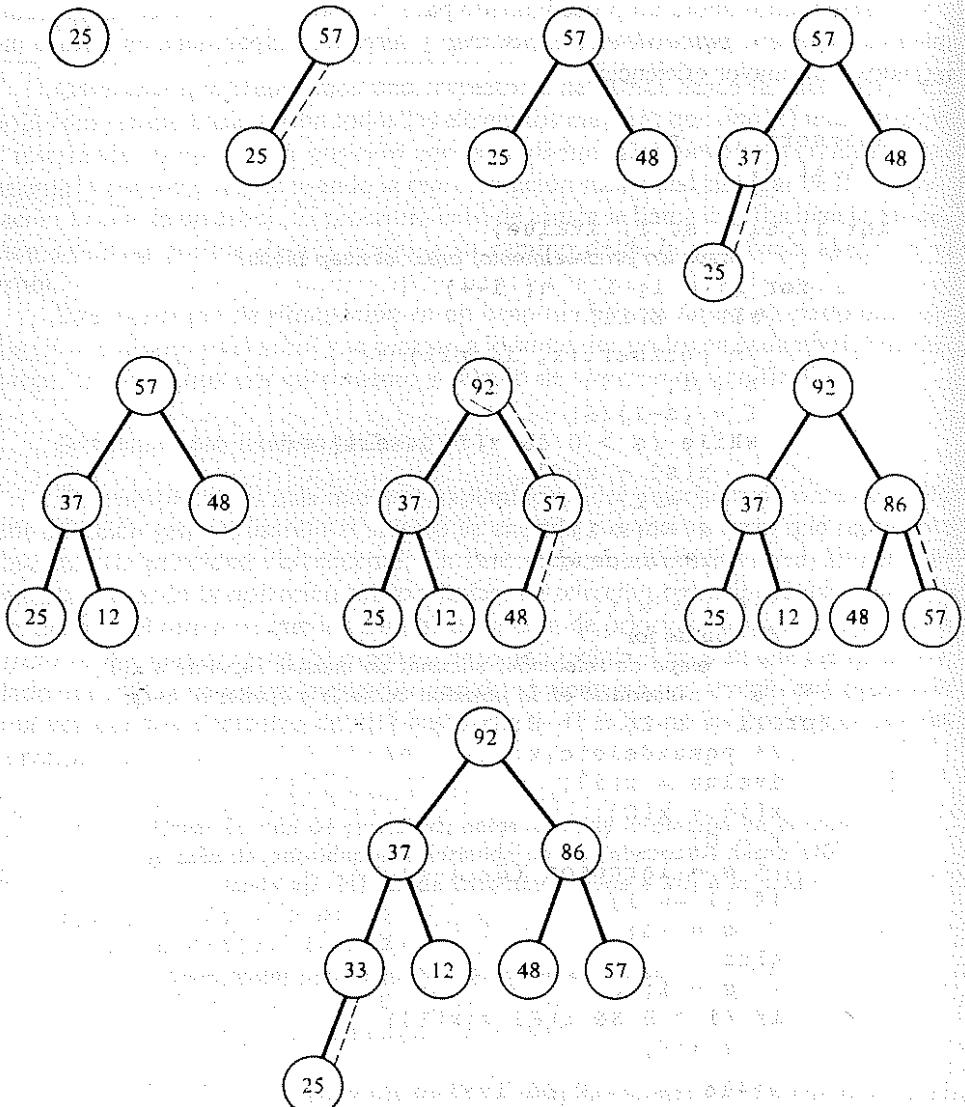


Figura 6.3.3 Creación de un heap de tamaño 8.

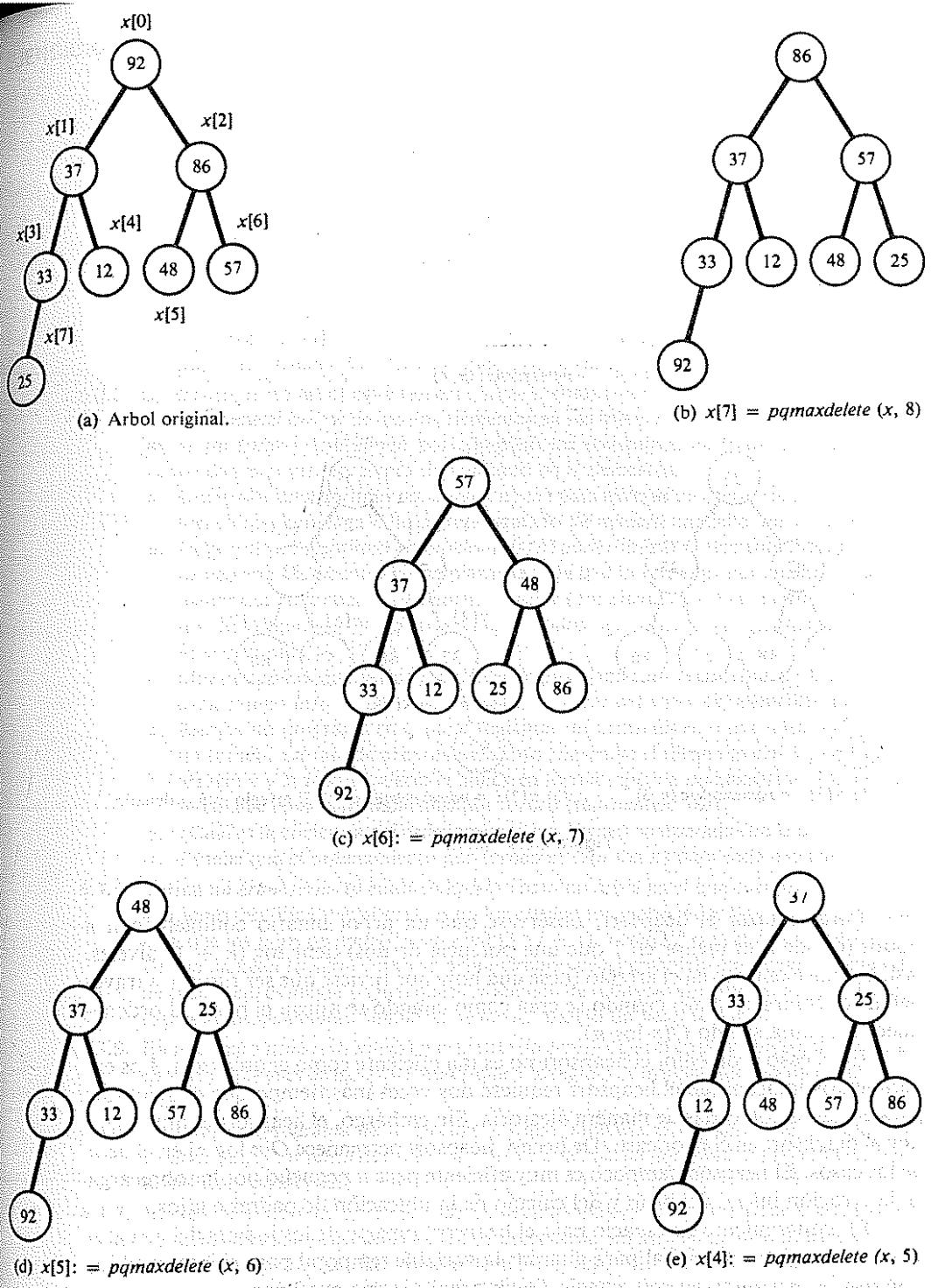


Figura 6.3.4 Ajuste de un heap.

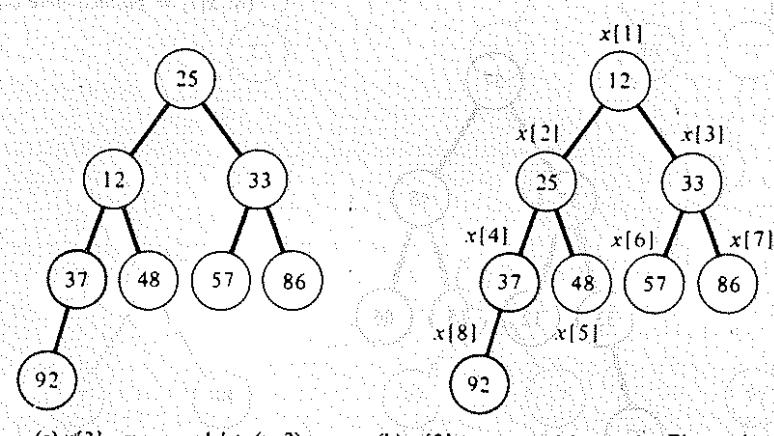
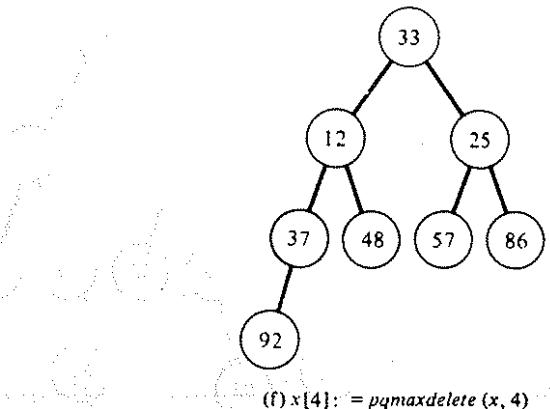
(h)  $x[2] := pqmaxdelete(x, 2)$ . El arreglo está ordenado.

Figura 6.3.4 (cont.)

Para analizar el heapsort, obsérvese que un árbol binario completo con  $n$  nodos (donde  $n$  es menor en 1 que una potencia de dos) tiene  $\log(n + 1)$  niveles. Así, si cada elemento en el arreglo fuese una hoja que tuviese que ser filtrada a través del árbol completo tanto cuando se crea como cuando se ajusta el heap, el ordenamiento seguiría siendo  $O(n \log n)$ .

En el caso promedio, el heapsort no es tan eficiente como el quicksort. Los experimentos indican que el heapsort requiere dos veces más tiempo que el quicksort para entradas colocadas de manera aleatoria. Sin embargo, el heapsort es muy superior al quicksort en el peor caso. De hecho, heapsort permanece  $O[n \log n]$  en el peor de los casos. El heapsort tampoco es muy eficiente para  $n$  pequeño por la sobrecarga de la creación inicial del heap y del cálculo de la ubicación de padres e hijos.

El requerimiento de espacio para el heapsort (aparte de los índices del arreglo) es sólo un registro adicional para guardar la variable temporal para el intercambio, si se usa la implantación con arreglo de un árbol binario quasi-completo.

6.3.1. Explique por qué el ordenamiento por selección directa es más eficiente que el ordenamiento de burbuja.

6.3.2. Considere el siguiente *ordenamiento por selección cuadrática*: Dividir los  $n$  elementos del archivo en  $\sqrt{n}$  grupos de  $\sqrt{n}$  elementos cada uno. Encuentre el elemento mayor de cada grupo e insértelo dentro de un arreglo auxiliar. Encuentre el elemento mayor de este arreglo auxiliar. Este es el elemento mayor del archivo. Despues reemplace este elemento en el arreglo por el siguiente elemento más grande del grupo del cual venia el primero. Encuentre otra vez el elemento más grande del arreglo auxiliar. Este es el segundo elemento más grande del archivo. Repita el proceso hasta que el archivo haya sido ordenado. Escriba una rutina en C para implantar un ordenamiento por selección cuadrática de manera tan eficiente como sea posible.

6.3.3. Un *torneo* es un árbol estrictamente binario quasi-completo en el cual cada nodo no hoja contiene el mayor de los dos elementos en sus hijos. Así, los contenidos de las hojas de un torneo determinan por completo los contenidos de todos sus nodos. Un torneo con  $n$  hojas representa un conjunto de  $n$  elementos.

- Desarrolle un algoritmo  $pqinsert(t, n, elt)$  para agregar un nuevo elemento  $elt$  a un torneo que contenga  $n$  hojas representado de manera implícita por un arreglo  $t$ .
- Desarrolle un algoritmo  $pqmaxdelete(t, n)$  para eliminar el elemento máximo de un torneo con  $n$  elementos al remplazar la hoja que lo contenga por cualquier valor o menor que el de cualquier elemento posible (por ejemplo,  $-1$  en un torneo de enteros no negativos) y luego reajustar todos los valores en el camino de esa hoja a la raíz.
- Muestre cómo simplificar  $pqmaxdelete$  guardando un apuntador a cada hoja en cada campo *info* de un nodo no hoja, en lugar del valor del elemento real.
- Escriba un programa en C para implantar un ordenamiento por selección usando un torneo. La fase de preprocessamiento construye el torneo inicial a partir de un arreglo  $x$  y la fase de selección aplica en forma repetida  $pqmaxdelete$ . Tal método se conoce como ordenamiento por torneo (*tournament sort*).
- ¿Cuál es la eficiencia del ordenamiento por torneo comparada con la del heapsort?
- Pruebe que el ordenamiento por torneo es  $O(n \log n)$  para toda entrada.

6.3.4. Defina un *árbol ternario quasi-completo* como un árbol en el que cualquier nodo tiene a lo sumo tres hijos y en el cual los nodos pueden ser enumerados de 0 a  $n - 1$ , de manera que los hijos de  $node[i]$  sean  $node[3 * i + 1]$ ,  $node[3 * i + 2]$  y  $node[3 * i + 3]$ . Defina un *heap ternario* como un árbol ternario quasi-completo en el cual el contenido de cada nodo es mayor o igual que el contenido de todos sus descendientes. Escriba una rutina de ordenamiento similar al heapsort usando un heap ternario.

6.3.5. Escriba una rutina *combine(x)* que acepte un arreglo  $x$  en el cual los subárboles con raíz en  $x[1]$  y  $x[2]$  sean heaps y que modifique el arreglo  $x$  de manera que represente un heap único.

6.3.6. Reescriba el programa de la sección 5.3 que implanta el algoritmo de Huffman de manera que el conjunto de nodos raíz forme una cola de prioridad implantada mediante un heap ascendente.

6.3.7. Escriba un programa en C que use un heap ascendente para intercalar  $n$  archivos de entrada, cada uno ordenado de manera ascendente, en un solo archivo de salida. Cada nodo del heap contiene un número de archivo y un valor. El valor se usa como llave mediante la cual se organiza el heap. Al inicio, se lee un valor de cada archivo y los  $n$  valores forman un heap ascendente, con el número de archivo del que venía cada valor

y que se ha mantenido junto a ese valor en el nodo. El valor más pequeño está entonces en la raíz del heap y es la salida, tomando su lugar el siguiente valor de su archivo asociado. Ese valor, junto con su número de archivo asociado, se intercala en la posición que le corresponde en el heap y se saca el nuevo valor raíz. Este proceso de salida / entrada/intercalación se repite hasta que no reste ningún valor de entrada.

- 6.3.8. Desarrolle un algoritmo usando un heap de  $k$  elementos para encontrar los  $k$  números mayores en un gran archivo de  $n$  números desordenados.

## 6.4. ORDENAMIENTOS POR INSERCIÓN

### Inserción simple

Un *ordenamiento por inserción* es uno en que reordena un conjunto de registros insertando registros en un archivo ordenado existente. Un ejemplo de *ordenamiento por inserción simple* es el siguiente procedimiento:

```
insertsort(x, n)
int x[], n;
{
 int i, k, y;
 /* Al inicio x[0] puede pensarse como un archivo ordenado de
 un elemento. Después de cada repetición en el siguiente
 ciclo, los elementos desde x[0] hasta x[k] están en orden
 */
 for (k = 1; k < n; k++) {
 /* insertar x[k] en el archivo ordenado */
 y = x[k];
 /* bajar una posición todos los elementos mayores que y */
 for (i = k-1; i >= 0 && y < x[i]; i--)
 x[i+1] = x[i];
 /* insertar y en la posición correcta */
 x[i+1] = y;
 } /* fin de for */
} /* fin de insertsort */
```

Como observamos al principio de la sección 6.3, el ordenamiento por inserción simple puede verse como un ordenamiento por selección general en el cual la cola de prioridad se implanta como un arreglo ordenado. Sólo es necesaria la fase de preprocesamiento en la que se insertan los elementos en la cola de prioridad; una vez que éstos han sido insertados, ya están ordenados, de manera que la selección deja de ser necesaria.

Si el archivo inicial está ordenado, se hace una sola comparación en cada paso, de manera que el ordenamiento es  $O(n)$ . Si el archivo está inicialmente ordenado en orden inverso, el ordenamiento es  $O(n^2)$ , dado que el número total de comparaciones es

$$(n - 1) + (n - 2) + \dots + 3 + 2 + 1 = (n - 1) * (n / 2)$$

que es  $O(n^2)$ . Sin embargo, el ordenamiento por inserción simple es, por lo regular, aún mejor que el ordenamiento de burbuja. Mientras más próximo esté un archivo a ser ordenado, más eficiente se volverá la inserción simple. El número promedio de *comparaciones* en el ordenamiento por inserción simple (considerando todas las posibles permutaciones del arreglo de entrada) es también  $O(n^2)$ . Los requerimientos de espacio para este ordenamiento comprenden sólo una variable temporal  $y$ .

La velocidad del ordenamiento se puede mejorar un tanto, usando una búsqueda binaria (ver las secciones 3.1, 3.2 y 7.1) para encontrar la posición adecuada de  $x[k]$  en el archivo ordenado  $x[0], \dots, x[k - 1]$ . Esto reduce el número total de comparaciones de  $O(n^2)$  a  $O(n \log n)$ . Sin embargo, aun cuando la posición correcta  $i$  para  $x[k]$  sea encontrada en  $O(\log n)$  pasos, cada uno de los elementos  $x[i + 1], \dots, x[k - 1]$  tiene que ser movido una posición. Esta última operación, ejecutada  $n$  veces, requiere de  $O(n^2)$  remplazamientos. No obstante, la técnica de búsqueda binaria no perfecciona, por lo tanto, los requerimientos de tiempo globales para el ordenamiento, de manera significativa.

Se puede hacer otra mejora al ordenamiento por inserción simple usando *inserción en lista*. En este método hay un arreglo *link* de apuntadores, uno para cada uno de los elementos del arreglo original. En un principio  $link[i] = i + 1$  para  $0 \leq i < n - 1$  y  $link[n - 1] = -1$ . Así, el arreglo se puede imaginar como una lista lineal señalada por un apuntador externo *first* inicializado a 0. Para insertar el  $k$ -ésimo elemento se recorre la lista ligada hasta encontrar la posición adecuada a  $x[k]$ , o hasta que se alcance el final de la lista. En ese momento  $x[k]$  puede insertarse en la lista con sólo ajustar los apuntadores a la misma sin recorrer ningún elemento del arreglo. Esto reduce el tiempo requerido por la inserción pero no el tiempo necesario para la búsqueda de la posición adecuada. Los requerimientos de espacio se incrementan también a causa de un arreglo *link* suplementario. El número de comparaciones sigue siendo  $O(n^2)$ , aunque el número de remplazamientos en el arreglo *link* es  $O(n)$ . El ordenamiento por inserción en lista puede verse como un ordenamiento por selección general en el cual la cola de prioridad está representada por una lista ordenada. Una vez más, no se necesita selección dado que los elementos están ordenados tan pronto como se completa la fase de preprocesamiento o inserción. Se deja como ejercicio al lector programar los ordenamientos por inserción binaria y por inserción en lista.

Ambos ordenamientos, por inserción simple y por selección directa, son más eficientes que el ordenamiento de burbuja. El ordenamiento por selección requiere de menos asignaciones que el ordenamiento por inserción pero más comparaciones. Así, se recomienda el ordenamiento por selección para archivos pequeños cuando los registros son grandes, de manera que la asignación no sea costosa, pero las llaves sean simples, de manera que la comparación sea barata. Si se presenta la situación inversa, se recomienda el ordenamiento por inserción. Si la entrada está inicialmente en una lista ligada, se recomienda la inserción en lista aun cuando los registros sean grandes, dado que no se requieren movimientos de datos (de manera opuesta a la modificación de apuntadores).

Por supuesto, el heapsort y el quicksort son, ambos, más eficientes que la inserción o selección para  $n$  grande. El punto de equilibrio es cercano a 20-30 para el quicksort; para tamaños con menos de 30 elementos, usar ordenamiento por in-

serción; para más de 30 usar quicksort. Una aceleración útil del quicksort usa ordenamiento por inserción para cualquier subarchivo de tamaño menor que 20. Para el heapsort el punto de equilibrio con el ordenamiento por inserción es aproximado a 60-70.

### Shell sort

Se puede lograr un perfeccionamiento más significativo del ordenamiento por inserción simple que el que se alcanza con la inserción binaria o en lista usando el *Shell sort* (*ordenamiento por disminución de incremento*), nombrado así en honor a su descubridor. Este método ordena subarchivos separados del archivo original. Estos subarchivos contienen todo elemento  $k$ -ésimo del archivo original. El valor de  $k$  se llama un *incremento*. Por ejemplo, si  $k$  es 5, se ordena primero el subarchivo que contiene a  $x[0], x[5], x[10]$ . De esta manera se ordenan cinco subarchivos, y cada uno contiene la quinta parte de los elementos del archivo original. Estos son (leyendo a través de)

Subarchivo 1  $\rightarrow x[0] \quad x[5] \quad x[10]$   
 Subarchivo 2  $\rightarrow x[1] \quad x[6] \quad x[11]$   
 Subarchivo 3  $\rightarrow x[2] \quad x[7] \quad x[12]$   
 Subarchivo 4  $\rightarrow x[3] \quad x[8] \quad x[13]$   
 Subarchivo 5  $\rightarrow x[4] \quad x[9] \quad x[14]$

El  $i$ -ésimo elemento del  $j$ -ésimo subarchivo es  $x[(i-1)*5 + j - 1]$ . Si se elige un incremento diferente  $k$ , se dividen los  $k$  subarchivos de manera que el  $i$ -ésimo elemento del  $j$ -ésimo subarchivo sea  $x[(i-1)*k + j - 1]$ .

Después de ordenar los primeros  $k$  subarchivos (en general por inserción simple), se elige un nuevo valor menor de  $k$  y se vuelve a particionar el archivo en un nuevo conjunto de subarchivos. Cada uno de esos subarchivos más grandes se ordena y se repite el proceso una vez más con un valor aún más pequeño de  $k$ . Al final, el valor de  $k$  se hace 1, de tal manera que se ordena el subarchivo que contiene al archivo completo. Al principio de todo el proceso se fija una secuencia decreciente de incrementos. El último valor de dicha secuencia tiene que ser 1.

Por ejemplo, si el archivo original es

25 57 48 37 12 92 86 33

y se elige la secuencia (5, 3, 1), se ordenan los siguientes subarchivos en cada iteración:

( $x[0], x[5]$ )

( $x[1], x[6]$ )

( $x[2], x[7]$ )

( $x[3]$ )

( $x[4]$ )

segunda iteración (incremento = 3)

( $x[0], x[3], x[6]$ )

( $x[1], x[4], x[7]$ )

( $x[2], x[5]$ )

tercera iteración (incremento = 1)

( $x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]$ )

La figura 6.4.1 ilustra el Shell sort en este archivo muestra. Las líneas debajo de cada arreglo unen elementos individuales de los subarchivos separados. Cada uno de los subarchivos se ordena usando el ordenamiento por inserción simple.

Presentamos abajo una rutina para implantar el Shell sort. Dicha rutina requiere además de los parámetros  $x$  y  $n$ , un arreglo *incrmts*, que contiene los incrementos decrecientes del ordenamiento y *numinc*, el número de elementos en el arreglo *incrmts*.

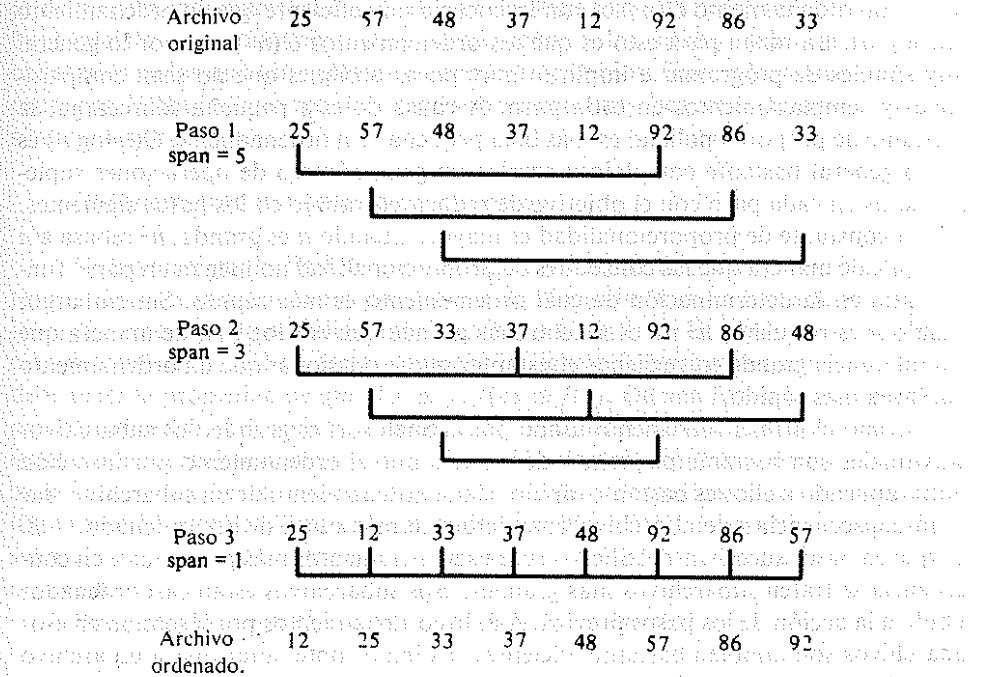


Figura 6.4.1

```

shellsort(x, n, incrmts, numinc)
int x[], n, incrmts[], numinc;
{
 int incr, j, k, span, y;
 for (incr = 0; incr < numinc; incr++) {
 /* span es el tamaño de los incrementos */
 span = incrmts[incr];
 for (j = span; j < n; j++) {
 /* Insertar el elemento x[j] en la posición adecuada */
 /* en el subarchivo correspondiente */
 y = x[j];
 for (k = j - span; k >= 0 && y < x[k]; k -= span)
 x[k + span] = x[k];
 x[k + span] = y;
 } /* fin de for */
 } /* fin de shellsort */
}

```

Asegurarse de entender la acción del programa anterior sobre el archivo muestra de la figura 6.4.1. Obsérvese que en la última iteración, donde  $span = 1$ , el ordenamiento se reduce a una inserción simple.

La idea que hay detrás de Shell sort es simple. Ya observamos que el ordenamiento por inserción simple es muy eficiente para un archivo que esté casi ordenado. También es importante darse cuenta de que cuando el tamaño del archivo,  $n$ , es pequeño, un ordenamiento  $O(n^2)$  es con frecuencia más eficiente que un ordenamiento  $O(n \log n)$ . La razón para esto es que los ordenamientos  $O(n^2)$  son por lo general muy simples de programar e implican muy pocas acciones que no sean comparaciones y remplazamientos en cada paso. A causa de esta pequeña sobrecarga, la constante de proporcionalidad es más bien pequeña. Un ordenamiento  $O(n \log n)$  es por lo general bastante complejo y emplea un gran número de operaciones suplementarias en cada paso con el objetivo de reducir el trabajo en los pasos siguientes. Así, su constante de proporcionalidad es mayor. Cuando  $n$  es grande,  $n^2$  rebasa a  $n * \log(n)$ , de manera que las constantes de proporcionalidad no juegan un papel fundamental en la determinación de cuál ordenamiento es más rápido. Sin embargo, cuando  $n$  es pequeño,  $n^2$  no es mucho más grande que  $n * \log(n)$ , de manera que una diferencia grande entre dichas constantes puede ocasionar que un ordenamiento  $O(n^2)$  sea más rápido.

Como el primer incremento usado por el Shell sort es grande, los subarchivos individuales son bastante pequeños, de manera que el ordenamiento por inserción simple aplicado a ellos es bastante rápido. Cada ordenamiento de un subarchivo trae como consecuencia que el archivo completo esté más cerca de ser ordenado. Así, aunque en pasos sucesivos del Shell sort se usen incrementos más pequeños y en consecuencia se traten subarchivos más grandes, esos subarchivos están casi ordenados debido a la acción de los pasos previos. Así, los ordenamientos por inserción en esos subarchivos son también bastante eficientes. Es importante notar que si un archivo está ordenado en forma parcial usando un incremento  $k$  y se ordena de manera sucesiva usando un incremento  $j$ , el archivo permanece parcialmente ordenado en el incremento  $k$ . Es decir, ordenamientos parciales subsecuentes no interfieren con los anteriores.

El análisis de la eficiencia del Shell sort está involucrado con las matemáticas y más allá del alcance de este libro. Los requerimientos reales de tiempo para un ordenamiento específico dependen del número de elementos en el arreglo *incrmts* y de sus valores reales. Un requerimiento intuitivamente claro es que los elementos de *incrmts* deberían ser primos relativos (es decir, que no tengan divisores comunes distintos de 1). Esto garantiza que iteraciones sucesivas entremezclen subarchivos de manera que el archivo completo esté en realidad casi ordenado cuando *span* es igual a 1 en la última iteración.

Se ha mostrado que el orden del Shell sort puede aproximarse por  $O(n(\log n)^2)$  si se usa una secuencia adecuada de incrementos. Para otra serie de incrementos, se puede probar que el tiempo de ejecución es  $O(n^{1.5})$ . Datos empíricos indican que el tiempo de ejecución es de la forma  $a * n^b$ , donde  $a$  está entre 1.1 y 1.7 y  $b$  es más o menos 1.26 o de la forma  $c * n * (\ln(n))^2 - d * n * \ln(n)$ , donde  $c$  está cercano a 0.3 y  $d$  está entre 1.2 y 1.75. En general el Shell sort se recomienda para archivos de tamaño moderado de algunos cientos de elementos.

Knuth recomienda la elección de incrementos como sigue: definir una función  $h$  de manera recursiva que tal que  $h(1) = 1$  y  $h(i + 1) = 3 * h(i) + 1$ . Sea  $x$  el entero menor tal que  $h(x) \geq n$ , y sea *numinc*, el número de incrementos, igual a  $x - 2$  e *incrmts[i]* igual a  $h(\text{numinc} - i + 1)$  para  $i$  desde 1 hasta *numinc*.

Se puede usar una técnica similar al Shell sort para perfeccionar el ordenamiento de burbuja. En la práctica, una causa importante de la ineficiencia en el ordenamiento de burbuja no es el número de comparaciones sino el de intercambios. Si se usa una serie de incrementos para definir subarchivos a ser ordenados por burbuja de manera individual, como en el caso de Shell sort, los ordenamientos por burbuja iniciales se harán sobre archivos pequeños y los últimos sobre archivos casi ordenados en los que son necesarios pocos intercambios. Este ordenamiento modificado por burbuja, que requiere una sobrecarga muy pequeña, funciona bien en situaciones prácticas.

### Ordenamiento por cálculo de dirección

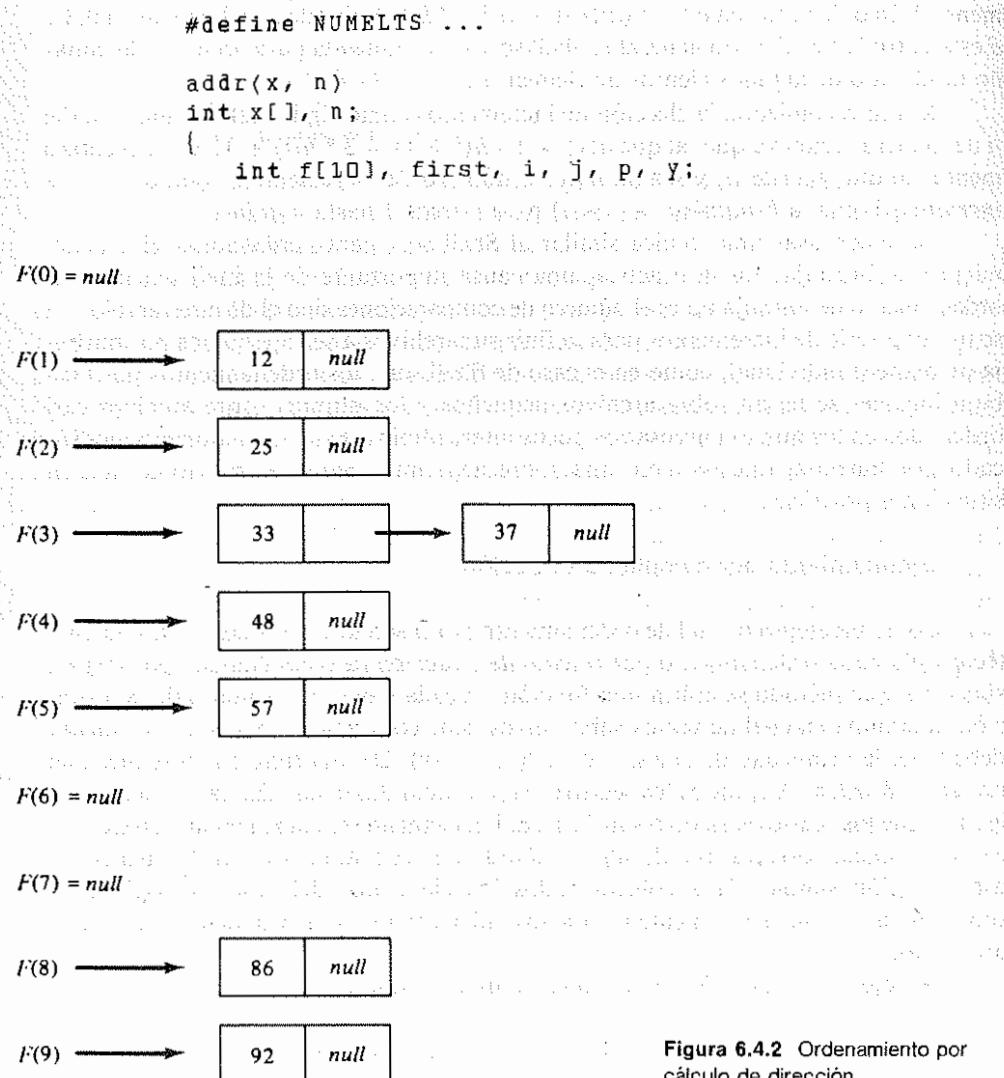
Como un ejemplo final de ordenamiento por inserción, considérese la siguiente técnica llamada ordenamiento por *cálculo de dirección* (a veces llamada por *dispersión*). En este método se aplica una función  $f$  a cada llave. El resultado de esta función determina en cuál de varios subarchivos debe colocarse el registro. La función debe tener la propiedad de que si  $x \leq y$ ,  $f(x) \leq f(y)$ . De una función tal se dice que *preserva el orden*. Así, todos los registros en un subarchivo tendrán llaves menores o iguales que los registros en otro subarchivo. Un elemento se coloca en un subarchivo en la secuencia correcta usando algún método de ordenamiento: con frecuencia se usa inserción simple. Tras colocar todos los elementos del archivo original en subarchivos, se pueden concatenar dichos subarchivos para producir el resultado ordenado.

Por ejemplo, considérese de nuevo el archivo muestra

25 57 48 37 12 92 86 33

Crearemos diez subarchivos, uno para cada uno de los diez primeros dígitos posibles. Al inicio, cada uno de esos subarchivos está vacío. Se declara un arreglo de apunadores  $f[10]$ , donde  $f[i]$  apunta al primer elemento en el archivo cuyo primer dígito es  $i$ . Después de examinar el primer elemento (25), éste se coloca dentro del archivo encabezado por  $f[2]$ . Cada uno de los subarchivos se guarda como una lista ligada ordenada de los elementos del arreglo original. Después de procesar cada uno de los elementos del archivo original, los subarchivos serán como los de la figura 6.4.2.

Presentamos una rutina para implantar el ordenamiento por cálculo de dirección. La rutina supone un arreglo de números de dos dígitos y usa el primer dígito de cada número para asignar ese número a un subarchivo.



```

struct {
 int info;
 int next;
} node[NUMELTS];

/* size available list */
int avail = 0;
for (i = 0; i < n-1; i++) {
 node[i].next = i+1;
}
node[n-1].next = -1;
/* inicializar apunadores */
for (i = 0; i < 10; i++) {
 f[i] = -1;
}
for (i = 0; i < n; i++) {
 /* insertar cada elemento, en forma sucesiva, en el
 * subarchivo respectivo utilizando inserción en lista */
 y = x[i];
 first = y/10; /* Encontrar el primer dígito de un
 * número de dos dígitos */
 /* Buscar en la lista ligada */
 place(&f[first], y); /* place inserta y en la posición adecuada en la lista */
 /* f[first] apunta ahora a la lista ligada a la que apunta f[first] */
} /* fin de for */
/* copiar los últimos números en el arreglo x */
i = 0;
for (j = 0; j < 10; j++) {
 p = f[j];
 while (p != -1) {
 x[i++] = node[p].info;
 p = node[p].next;
 } /* fin de while */
} /* fin de for */
} /* fin de addr */

```

Los requerimientos de espacio del ordenamiento por cálculo de dirección son aproximadamente  $2 * n$  (usados por el arreglo *node*) más algunos nodos cabecera y variables temporales. Obsérvese que si los datos originales se dan en la forma de una lista ligada en lugar de un arreglo secuencial, no es necesario guardar el arreglo *x* y la estructura ligada *node* a la vez.

Para evaluar los requerimientos de tiempo del ordenamiento, obsérvese lo siguiente: Si los  $n$  elementos originales están distribuidos uniformemente de manera aproximada sobre los  $m$  subarchivos y el valor de  $n/m$  es más o menos 1, el tiempo de ordenamiento está próximo a  $O(n)$ , dado que la función asigna cada elemento a su archivo correspondiente y se requiere poco trabajo extra para colocar el elemento dentro del propio subarchivo. Por otra parte, si  $n/m$  es mucho mayor que 1, o si el archivo original no está distribuido de manera uniforme sobre los  $m$  subarchivos, se requiere un trabajo significativo para insertar un elemento en el archivo adecuado y el tiempo es, en consecuencia, cercano a  $O(n^2)$ .

## EJERCICIOS

6.4.1. El *ordenamiento por inserción de dos vías* es una modificación del ordenamiento por inserción simple como sigue: Se aparta un arreglo de salida separado de tamaño  $n$ . Este arreglo de salida actual igual que una estructura circular como en la sección 4.1.  $x[0]$  se coloca en el elemento medio del arreglo. Una vez que un grupo contiguo de elementos está en el arreglo, se hace espacio para un nuevo elemento recorriendo todos los elementos menores un paso a la izquierda o todos los elementos mayores un paso a la derecha. La elección de la dirección en la cual se deben recorrer los elementos depende de cuál dirección causaría la menor cantidad de corrimientos. Escribir una rutina en C para implantar esta técnica.

6.4.2. El *ordenamiento por inserción de intercalación* procede de la siguiente manera:

Paso 1: Para todo  $i$  par entre 0 y  $n - 2$ , comparar  $x[i]$  con  $x[i + 1]$ . Colocar el mayor en la posición siguiente de un arreglo *large* y el más pequeño en la siguiente posición de un arreglo *small*. Si  $n$  es impar, colocar  $x[n - 1]$  en la última posición del arreglo *small*. (*Large* es de tamaño  $ind$ , donde  $ind = (n - 1)/2$ ; *small* es de tamaño  $ind$  o  $ind + 1$ , dependiendo de si  $n$  es par o impar.)

Paso 2: Ordenar el arreglo *large* usando inserción por intercalación de manera recursiva. Siempre que un elemento *large*[ $j$ ] se mueva a *large*[ $k$ ], *small*[ $j$ ] también se mueve a *small*[ $k$ ]. (Al final de este paso,  $large[i] \leq large[i + 1]$  para todo  $i$  menor que  $ind$  y  $small[i] \leq large[i]$  para todo  $i$  menor o igual que  $ind$ .)

Paso 3: Copiar *small*[0] y todos los elementos de *large* desde  $x[0]$  a  $x[ind]$ .

Paso 4: Definir el entero  $núm[i]$  como  $(2^{i+1} + (-1)^i)/3$ . Comenzando con  $i = 0$  y procediendo con 1 mientras  $núm[i] \leq (n/2) + 1$ , insertar los elementos de *small*[ $núm[i + 1]$ ] hasta *small*[ $núm[i] + 1$ ] en *x* por turno, usando inserción binaria. (Por ejemplo si  $n = 20$ , los valores sucesivos de  $núm$  son  $núm[0] = 1$ ,  $núm[1] = 1$   $núm[2] = 3$ ,  $núm[3] = 5$  y  $núm[4] = 11$ , que es igual a  $(n/2) + 1$ . Así, los elementos de *small* se insertan en el siguiente orden: *small*[2], *small*[1]; después *small*[4], *small*[3]; después *small*[9], *small*[8], *small*[7], *small*[6], *small*[5]. En este ejemplo no hay *small*[10].)

Escribir una rutina en C para implantar esta técnica.

6.4.3. Modifique el quicksort de la sección 6.2 de manera que use un ordenamiento por inserción simple cuando un subarchivo tenga un tamaño por debajo de  $s$ . Determine de manera experimental qué valor de  $s$  debería usarse para obtener una eficiencia máxima.

6.4.4. Pruebe que si un archivo se ordena en forma parcial, usando un incremento  $j$  en el Shell sort, se mantiene parcialmente ordenado en dicho incremento aun después de ser ordenado para otro incremento  $k$ .

6.4.5. Explique por qué es deseable escoger todos los incrementos del Shell sort de manera que sean primos relativos.

6.4.6. ¿Cuál es el número de comparaciones e intercambios (en términos del tamaño del archivo  $n$ ) ejecutados con cada uno de los métodos de ordenamiento (del  $a$  al  $j$ ) para los siguientes archivos:

- i. un archivo ordenado.
  - ii. un archivo ordenado en orden inverso (es decir de mayor a menor)
  - iii. un archivo en el cual los elementos  $x[0]$ ,  $x[2]$ ,  $x[4]$ , son los menores y están ordenados de cierta forma y los elementos  $x[1]$ ,  $x[3]$ ,  $x[5]$ , ... son los mayores y están en orden inverso (es decir,  $x[0]$  es el menor  $x[1]$  es el mayor,  $x[2]$  es el siguiente más pequeño,  $x[3]$  es el siguiente más grande, y así de manera sucesiva).
  - iv. un archivo en el cual  $x[0]$  hasta  $x[ind]$  (donde  $ind = (n - 1)/2$ ) son los menores y están ordenados y en el cual  $x[ind + 1]$  hasta  $x[n - 1]$  son los mayores y están en orden inverso.
  - v. un archivo en el cual  $x[0]$ ,  $x[2]$ ,  $x[4]$ , ... son los elementos menores en orden y  $x[1]$ ,  $x[3]$ ,  $x[5]$ , ... son los mayores en orden.
- a. el ordenamiento por inserción simple
  - b. el ordenamiento por inserción usando búsqueda binaria
  - c. el ordenamiento por inserción en lista
  - d. el ordenamiento por inserción de dos vías del ejercicio 6.4.1.
  - e. el ordenamiento por inserción de intercalación del ejercicio 6.4.2
  - f. el Shell sort usando incrementos 2 y 1
  - g. el Shell sort usando incrementos 3, 2 y 1
  - h. el Shell sort usando incrementos 8, 4, 2 y 1
  - i. el Shell sort usando incrementos 7, 5, 3 y 1
  - j. el ordenamiento por cálculo de dirección presentado en el texto

6.4.7. En qué circunstancias se recomendaría el uso de cada uno de los siguientes ordenamientos respecto a los otros:

- a. el Shell sort de esta sección
- b. el heapsort de la sección 6.3
- c. el quicksort de la sección 6.2

6.4.8. Determinar cuál de los siguientes ordenamientos es más eficiente:

- a. el ordenamiento por inserción simple de esta sección
- b. el ordenamiento por selección directa de la sección 6.3
- c. el ordenamiento de burbuja de la sección 6.2

## 6.5. ORDENAMIENTOS POR INTERCALACION Y DE BASE

### Ordenamientos por intercalación

Intercalación es el proceso de combinar dos o más archivos ordenados en un tercer archivo ordenado. Un ejemplo de una rutina que acepta dos arreglos ordenados *a* y *b* de  $n_1$  y  $n_2$  elementos respectivamente, intercalándolos dentro de un tercer arreglo *c* que contiene  $n_3$  elementos, es el siguiente:

```
mergearr(a, b, c, n1, n2, n3)
int a[], b[], c[], n1, n2, n3;
{
 int apoint, bpoint, cpoint;
 int alimit, blimit, climit;
```

```

alimit = n1-1;
blimit = n2-1;
climit = n3-1;
if (n1+n2 != n3) {
 printf("incompatibilidad en los límites del arreglo/n");
 exit(1);
} /* fin de if */
/* apoint y bpoint son indicadores de que tanto se ha avanzado en los arreglos a y b */
apoint = 0;
bpoint = 0;
for (cpoint = 0; apoint <= alimit && bpoint <= blimit;
 cpoint++)
{
 if (a[apoint] < b[bpoint])
 c[cpoint] = a[apoint++];
 else
 c[cpoint] = b[bpoint++];
}
while (apoint <= alimit)
 c[cpoint++] = a[apoint++];
while (bpoint <= blimit)
 c[cpoint++] = b[bpoint++];
} /* fin de mergesort */

```

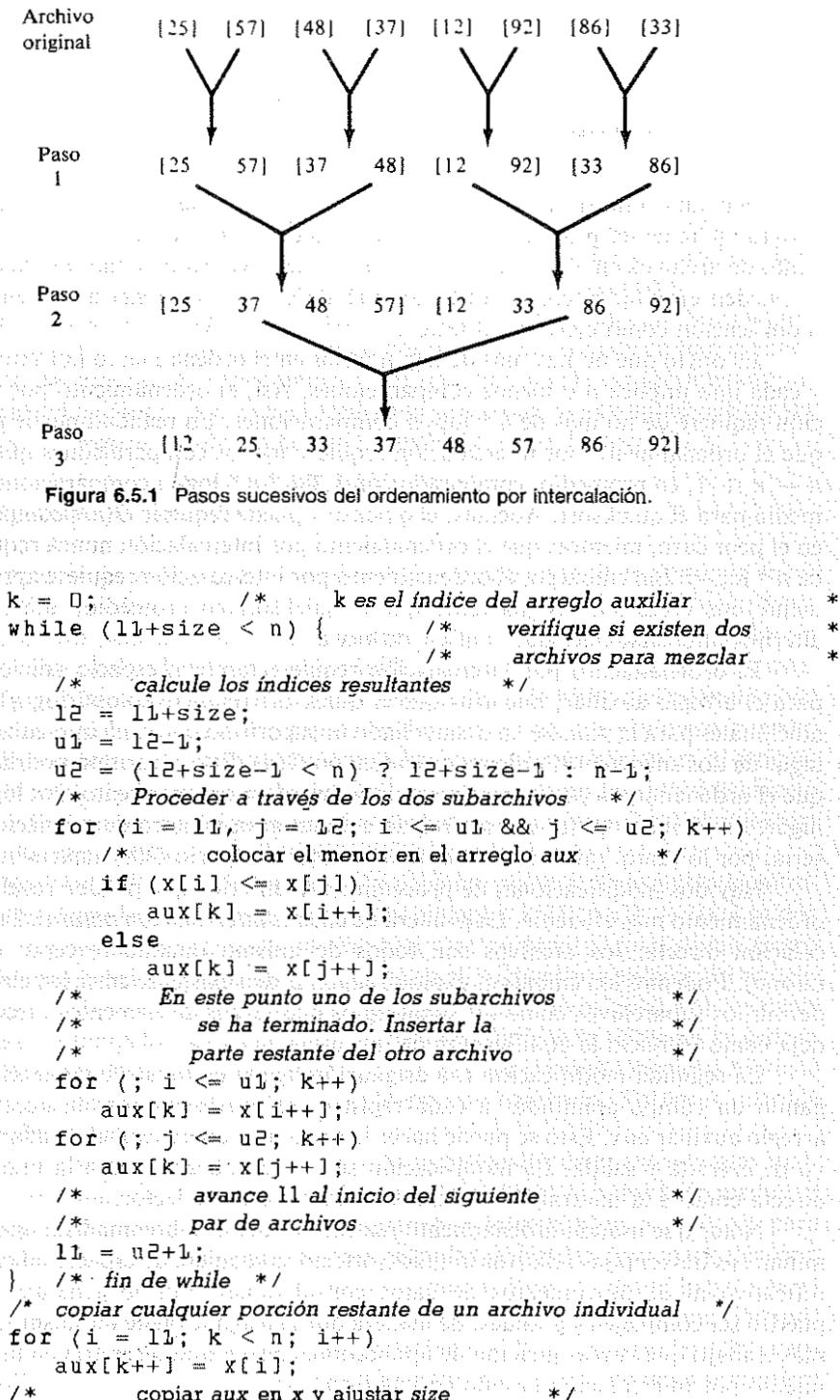
Podemos usar esta técnica para ordenar un archivo de la siguiente manera. Dividir el archivo en  $n$  subarchivos de tamaño 1 e intercalar pares adyacentes (inconexos) de archivos. Entonces tenemos más o menos  $n/2$  archivos de tamaño 2. Repetir el proceso hasta que sólo reste un archivo de tamaño  $n$ . La figura 6.5.1 ilustra cómo opera este proceso en un archivo muestra. Cada archivo individual está entre corchetes.

Presentamos una rutina para implantar la descripción anterior de un *ordenamiento por intercalación directa*. Se requiere de un arreglo auxiliar *aux* de tamaño  $n$  para guardar los resultados de intercalar dos subarreglos de *x*. La variable *size* contiene el tamaño de los subarreglos que se están intercalando. Como en todo momento, los dos archivos que están fusionándose son subarreglos de *x*, se requieren límites superiores e inferiores para indicar los subarchivos de *x* que se están intercalando. *l1* y *u1* representan los límites inferior y superior del primer archivo y *l2*, *u2* los del segundo. *i* y *j* se usan para hacer referencia a elementos de los archivos fuente que se están intercalando y *k* indexa el archivo destino *aux*. La rutina se presenta a continuación:

```

#define NUMELTS ...
mergesort(x, n)
int x[], n;
{
 int aux[NUMELTS], i, j, k, l1, l2, size, u1, u2;
 size = 1; /* intercalar archivos de tamaño 1 */
 while (size < n) {
 l1 = 0; /* inicializar el límite inferior del primer archivo */

```



```

for (i = 0; i < n; i++)
 x[i] = aux[i];
size *= 2;
} /* fin de while */
} /* fin de mergesort */

```

Hay una deficiencia en el procedimiento anterior que es fácil de remediar: si el programa ha de ser práctico para arreglos grandes. En lugar de intercalar cada conjunto de archivos en el arreglo auxiliar *aux* y luego volver a copiar *aux* dentro de *x*, se pueden ejecutar intercalaciones alternas de *x* a *aux* y de *aux* a *x*. Dejamos esta modificación como ejercicio al lector.

Es obvio que no hay más de  $\log_2 n$  pasos en el ordenamiento por intercalación y cada uno implica  $n$  o menos comparaciones. Así, el ordenamiento por intercalación requiere de no más de  $n * \log_2 n$  comparaciones. En realidad puede mostrarse que el ordenamiento por intercalación requiere menos comparaciones que  $n * \log_2 n - n + 1$ , en promedio, comparado con  $1.386 * n * \log_2 n$  comparaciones en promedio para el quicksort. Además, el quicksort puede requerir  $O(n^2)$  comparaciones en el peor caso, mientras que el ordenamiento por intercalación nunca requiere más de  $n * \log_2 n$ . Sin embargo, el ordenamiento por intercalación requiere aproximadamente dos veces más asignaciones que el quicksort en promedio, aun cuando se alternen intercalaciones de *x* a *aux* y de *aux* a *x*.

El ordenamiento por intercalación requiere también espacio adicional  $O(n)$  para el arreglo auxiliar, mientras que el quicksort requiere sólo  $O(\log n)$  espacios adicionales para la pila. Se ha desarrollado un algoritmo para una intercalación en el lugar de dos subarreglos ordenados en tiempo  $O(n)$ . Este algoritmo podría permitir que el ordenamiento por intercalación se volviese un ordenamiento  $O(n \log n)$  en el lugar. Sin embargo, esta técnica requiere de un gran número de asignaciones y no sería, por lo tanto, tan práctica como encontrar el espacio  $O(n)$  suplementario.

Hay dos modificaciones de procedimiento anterior que pueden resultar en un ordenamiento más eficiente. La primera de ellas es *intercalación natural*. En la intercalación directa, los archivos son todos del mismo tamaño (excepto quizás el último). Podemos sin embargo, explotar algún orden existente entre los elementos y definir los subarchivos como los subarreglos más largos de elementos crecientes. Se deja como ejercicio la codificación de tal rutina.

La segunda modificación usa asignación ligada en lugar de secuencial. Agregando un campo apuntador a cada registro, se puede eliminar la necesidad del arreglo auxiliar *aux*. Esto se puede hacer ligando de manera explícita cada subarchivo de entrada y salida. La modificación se puede aplicar tanto a la intercalación directa como a la natural. Lo dejamos como ejercicio al lector.

Notar que usando ordenamiento por intercalación sobre una lista ligada se eliminan sus desventajas relativas al quicksort: no se requiere de espacio adicional significativo ni de movimientos de datos considerables. Por lo general, los datos pueden ser complejos y grandes, de manera que la asignación de los mismos requiere más trabajo que la reasignación de apuntadores que es aún necesaria en un ordenamiento por intercalación basado en una lista.

El ordenamiento por intercalación también puede ser presentado de manera bastante natural como un proceso recursivo en el cual las dos mitades del arreglo se ordenan primero en forma recursiva usando ordenamiento por intercalación y una vez ordenadas se unen por intercalación. Para los detalles, ver el ejercicio 6.5.1 y el 6.5.2. Ambos, el ordenamiento por intercalación y el quicksort, son métodos que implican la división del archivo en dos partes, ordenando ambas por separado y juntándolas luego. En el ordenamiento por intercalación la división es trivial (se toman simplemente las dos mitades) y la unión es difícil (intercalar los dos archivos ordenados). En el quicksort, la división es difícil (particionar) y la unión es trivial (las dos mitades y el pivot forman de manera automática un arreglo ordenado).

El ordenamiento por inserción puede considerarse como un caso especial del ordenamiento por intercalación en el cual las dos mitades constan de un solo elemento y el resto del arreglo. El ordenamiento por selección puede ser considerado un caso especial del quicksort en el cual el archivo se partitiona en una mitad que consta sólo del mayor elemento y otra mitad que contiene el resto del arreglo.

### El algoritmo de Cook-Kim

Con frecuencia se sabe que un archivo está casi ordenado salvo por unos pocos elementos. O se puede saber que un archivo de entrada es probable que esté ordenado. Para archivos pequeños que estén casi ordenados o para archivos ordenados, la inserción simple es el ordenamiento más rápido considerando comparaciones y asignaciones que se ha encontrado. Para archivos grandes o archivos que estén un poco desordenados el ordenamiento más rápido es el quicksort usando el elemento medio como pivot. (Si se consideran sólo las comparaciones, el más rápido es el ordenamiento por intercalación.) Sin embargo, existe otro algoritmo híbrido descubierto por Cook y Kim que es más rápido que el ordenamiento por inserción y el quicksort usando el elemento medio para entradas casi ordenadas.

El algoritmo de Cook-Kim opera de la siguiente manera: Se examina la entrada para pares desordenados de elementos (por ejemplo,  $x[k] > x[k + 1]$ ). Los dos elementos en un par no ordenado se eliminan y se agregan al final de un nuevo arreglo. Se examina el siguiente par que consta en el predecesor y sucesor del par eliminado, tras la eliminación de un par desordenado. El arreglo original, con los pares desordenados eliminados, está ahora de cierta forma ordenado. Después se ordena el arreglo de pares desordenados usando el quicksort por elemento medio si contiene más de 30 elementos y por inserción simple en caso contrario. Entonces, los dos arreglos se intercalan.

El algoritmo de Cook-Kim explota más lo ordenado de la entrada que cualquier otro ordenamiento y es mucho mejor que el quicksort por elemento medio, el ordenamiento por inserción, el ordenamiento por intercalación o el Bsort para entradas casi ordenadas. Sin embargo, para entradas generadas de manera aleatoria el algoritmo de Cook-Kim es menos eficiente que el Bsort (y desde luego que el quicksort o el ordenamiento por intercalación). En consecuencia, el quicksort por elemento medio, el ordenamiento por intercalación o el Bsort, son preferibles cuando son probables archivos grandes de entrada ordenados aunque también se requiere un comportamiento aleatorio de la entrada.

## Ordenamiento por base

El siguiente método de ordenamiento que consideramos se llama *ordenamiento por base*. Este ordenamiento se basa en los valores de los dígitos presentes en las representaciones posicionales de los números que se están ordenando. Por ejemplo, el número 235 en notación decimal se escribe como un 2 en las centenas, un 3 en las decenas y un 5 en las unidades. El mayor de dos enteros de igual longitud en esa notación se puede determinar de la siguiente manera: Comenzar en el dígito más significativo y avanzar hacia los menos significativos siempre y cuando los dígitos correspondientes de ambos números coincidan. El número con el dígito mayor en la primera posición en que ambos números no coinciden será entonces el mayor. Por supuesto, si todos los dígitos de ambos números coinciden, los números son iguales.

Podemos escribir una rutina de ordenamiento basada en el método anterior. Usando la base decimal, por ejemplo, los números se pueden repartir en diez grupos basados en sus dígitos más significativos. (Por sencillez, suponemos que todos los números tienen la misma cantidad de dígitos, agregando ceros no significativos si es necesario). Así, cualquier elemento en el grupo “0” es menor que todo elemento en el grupo “1”, cuyos elementos son, todos menores que cualquier elemento en el grupo “2” y así sucesivamente. Entonces podemos ordenar dentro de los grupos individuales basados en el siguiente dígito más significativo. Repetimos este proceso hasta que cada subgrupo haya sido subdividido de manera que los dígitos menos significativos estén ordenados. En este momento, el archivo original ha sido ordenado. (Observese que la división de un subarchivo en grupos con el mismo dígito en una posición dada es similar a la operación *partition* en el quicksort, en la cual un subarchivo se divide en dos grupos basado en la comparación con un elemento particular.) Este método se llama a veces el *ordenamiento por intercambio de base*; su programación se deja al lector como ejercicio.

Consideremos una alternativa al método anterior. De la anterior discusión parece ser que hay una gran cantidad de contabilidad involucrada en la constante subdivisión de los archivos y la distribución de sus contenidos en los subarchivos basada en dígitos particulares. Con seguridad, sería más fácil que pudiéramos procesar el archivo completo como un todo en lugar de tratar con muchos archivos individuales.

Suponer que llevamos a cabo las siguientes acciones en el archivo para cada dígito comenzando con el menos significativo y terminando con el más significativo. Tómese cada número en el orden en que aparece en el archivo, y colóquese dentro de una de diez colas, dependiendo del valor del dígito que está siendo procesado. Después póngase cada cola en el archivo original comenzando con la de los números de dígito 0 y terminando con la de números de dígito 9. Cuando se haya ejecutado lo anterior para cada dígito, comenzando con el menos significativo y terminando con el más significativo, el archivo está ordenado. Este método de ordenamiento se llama *ordenamiento por base*.

Observese que este esquema de ordenamiento ordena primero los dígitos menos significativos. Así, cuando todos los números estén ordenados en un dígito más significativo, aquellos que tienen el mismo dígito en esa posición pero dígitos diferentes en una posición menos significativa ya están ordenados en la posición menos

significativa. Esto permite procesar el archivo completo sin subdividir los archivos y sin perder de vista dónde comienza y termina cada subarchivo. La figura 6.5.2 ilustra este ordenamiento aplicando al archivo muestra

25 57 48 37 12 92 86 33

Asegúrese de que se pueden seguir las acciones descritas en los dos pasos de la figura 6.5.2.

En consecuencia podemos esbozar un algoritmo para ordenar del modo antes descrito como sigue:

```
for (k = dígito menos significativo; k <= dígito más significativo; k++) {
 for (i = 0; i < n; i++) {
 y = x[i];
 j = k-ésimo dígito de y;
 colocar y al final de queue[j];
```

Archivo original

25 57 48 37 12 92 86 33

Colas basadas en dígitos menos significativos.

|           | Frente | Final |
|-----------|--------|-------|
| queue [0] |        |       |
| queue [1] |        |       |
| queue [2] | 12     | 92    |
| queue [3] | 33     |       |
| queue [4] |        |       |
| queue [5] | 25     |       |
| queue [6] | 86     |       |
| queue [7] | 57     | 37    |
| queue [8] | 48     |       |
| queue [9] |        |       |

Después del primer paso:

12 92 33 25 86 57 37 48

Colas basadas en el dígito más significativo.

|           | Frente | Final |
|-----------|--------|-------|
| queue [0] |        |       |
| queue [1] | 12     |       |
| queue [2] | 25     |       |
| queue [3] | 33     | 37    |
| queue [4] | 48     |       |
| queue [5] | 57     |       |
| queue [6] |        |       |
| queue [7] |        |       |
| queue [8] | 86     |       |
| queue [9] | 92     |       |

Archivo ordenado: 12 25 33 37 48 57 86 92

Figura 6.5.2 Ilustración del ordenamiento por base.

```

 } /* fin de for */
 for (qu = 0; qu < 10; qu++)
 colocar los elementos de queue[qu] en la siguiente posición
 secuencial de x;
} /* fin de for */

```

Presentamos ahora un programa para implantar el ordenamiento anterior sobre números de  $m$ -dígitos. Con el objetivo de ahorrar una gran cantidad de trabajo en el procesamiento de las colas (en especial en el paso donde regresamos los elementos de la cola al archivo original) escribimos el programa usando asignación ligada. Si la entrada inicial a la rutina es un arreglo, se convierte primero dicha entrada en una lista lineal ligada; si la entrada original ya está en formato ligado, este paso no es necesario y de hecho, se ha ahorrado espacio. Esta es la misma situación que en la rutina *addr* (ordenamiento por cálculo de dirección) de la sección 6.4. Como en programas previos, no hacemos llamadas internas a rutinas sino que realizamos su acción en el lugar.

```

#define NUMELTS ...
radixsort(x, n)
int x[], n;
{
 int front[10], rear[10];
 struct {
 int info;
 int next;
 } node[NUMELTS];
 int exp, first, i, j, k, p, q, y;
 /* inicializar la lista ligada */
 for (i = 0; i < n-1; i++) {
 node[i].info = x[i];
 node[i].next = i+1;
 }
 /* fin de for */
 node[n-1].info = x[n-1];
 node[n-1].next = -1;
 first = 0; /* first es la cabeza de la lista ligada */
 for (k = 1; k < 5; k++) {
 /* suponer que se tienen números de cuatro dígitos */
 for (i = 0; i < 10; i++) {
 /* inicializar las colas */
 rear[i] = -1;
 front[i] = -1;
 }
 /* fin de for */
 /* procesar cada elemento de la lista */
 while (first != -1) {
 p = first;
 first = node[first].next;
 y = node[p].info;

```

```

 /* extraer el k-ésimo dígito */
 exp = power(10, k-1);
 /* elevar 10 a la k-ésima potencia */
 /* (k-1)-ésima potencia */

 j = (y/exp)%10;
 /* insertar y en queue[j] */
 q = rear[j];
 if (q == -1)
 front[j] = p;
 else
 node[q].next = p;
 rear[j] = p;
 }
 /* fin de while */
 /* En este punto cada registro está en la cola adecuada
 de acuerdo con el dígito k. Formar ahora una lista simple a partir
 de todos los elementos de la cola. Encontrar el primero */
 for (j = 0; j < 10 && front[j] == -1; j++)
 ;
 first = front[j];
 /* encadenar las colas restantes */
 while (j <= 9) { /* verificar si se terminó */
 /* encontrar el siguiente elemento */
 for (i = j+1; i < 10 && front[i] == -1; i++)
 ;
 if (i <= 9) {
 p = i;
 node[rear[j]].next = front[i];
 }
 /* fin de if */
 j = i;
 }
 /* fin de while */
 node[rear[p]].next = -1;
 }
 /* fin de for */
 /* reescribir el arreglo original */
 for (i = 0; i < n; i++) {
 x[i] = node[first].info;
 first = node[first].next;
 }
 /* fin de for */
}
/* fin de radixsort */

```

Los requerimientos de tiempo del ordenamiento por base dependen del número de dígitos ( $m$ ) y del número de elementos en el archivo ( $n$ ). Como el ciclo externo  $for (k = 1; k = m; k++)$  se recorre  $m$  veces (una para cada dígito) y ciclo interno  $n$  veces (una para cada ordenamiento del archivo), el ordenamiento es más o menos  $O(m * n)$ . Así, el ordenamiento es razonablemente eficiente si el número de dígitos en las llaves no es demasiado grande. Deberá observarse, sin embargo, que muchas máquinas tienen facilidades en el hardware para ordenar dígitos de un número (en especial si están en binario) mucho más rápido de lo que pueden ejecutar una comparación de dos llaves completas. Por lo tanto no es razonable comparar la estimación  $O(m * n)$  con algunos de los otros resultados a los que llegamos en este capítulo. Nótese también que si las llaves son densas (es decir, si casi todo número

que sea una posible llave lo es en realidad),  $m$  se aproxima a  $\log n$  de manera que  $O(m * n)$  se aproxima a  $O(n \log n)$ . El ordenamiento requiere de espacio para almacenar apuntadores al frente y al final de las colas, además de un campo extra en cada registro que se usa como apuntador en las listas ligadas. Si el número de dígitos es grande, es a veces más eficiente ordenar el archivo aplicando primero el ordenamiento por base a los dígitos más significativos y usando luego inserción directa en el archivo redispuesto. En casos en que la mayoría de los registros en el archivo tengan dígitos más significativos diferentes, este proceso elimina pasos innecesarios para los dígitos menos significativos.

## EJERCICIOS

- 6.5.1. Escriba un algoritmo para una rutina  $merge(x, lb1, ub2)$  que suponga que  $x[lb1]$  hasta  $x[ub1]$  y  $x[ub1 + 1]$  hasta  $x[ub2]$  están ordenados e intercalar a ambos en  $x[lb1]$  hasta  $x[ub2]$ .

- 6.5.2. Considere la siguiente versión recursiva del ordenamiento por intercalación que usa la rutina  $merge$  del ejercicio previo. La rutina se llama al inicio mediante  $msort2(x, 0, n - 1)$ . Reescriba la rutina eliminando la recursión y simplificando. ¿Cómo difiere la rutina resultante de la del texto?

```
msort2(x, lb, ub)
{
 if (lb != ub) {
 mid = (ub+lb)/2;
 msort2(x, lb, mid);
 msort2(x, mid+1, ub);
 merge(x, lb, mid, ub);
 } /* fin de if */
} /* fin de msort2 */
```

- 6.5.3. Sea  $a(l1, l2)$  el número promedio de comparaciones necesarias para intercalar dos arreglos ordenados de longitud  $l1$  y  $l2$  respectivamente, donde los elementos de los arreglos están elegidos al azar entre  $l1 + l2$  elementos.

- ¿Cuáles son los valores de  $a(l1, 0)$  y  $a(0, l2)$ ?
  - Muestre que para  $l1 > 0$  y  $l2 > 0$ ,  $a(l1, l2)$  es igual a  $(l1/(l1 + l2)) * (1 + a(l1 - 1, l2)) + (l2/(l1 + l2)) * (1 + a(l1, l2 - 1))$ . (Sugerencia: exprese el número promedio de comparaciones en términos del número promedio de comparaciones tras la primera comparación).
  - Muestre que  $a(l1, l2)$  es igual a  $(l1 * l2 * (l1 + l2 + 2)) / ((l1 + 1) * (l2 + 1))$ .
  - Verifique la fórmula del inciso c para dos arreglos, uno de tamaño 2 y otro de tamaño 1.
- 6.5.4. Considere el siguiente método para intercalar dos arreglos  $a$  y  $b$  en un arreglo  $c$ : Realizar una búsqueda binaria para  $b[0]$  en el arreglo  $a$ . Si  $b[0]$  está entre  $a[i]$  y  $a[i + 1]$  poner  $a[i]$  hasta  $a[i]$  en el arreglo  $c$ , luego poner  $b[0]$  en  $c$ . Después ejecutar una búsqueda binaria para  $b[1]$  en el subarreglo  $a[i + 1]$  a  $a[la]$  (donde  $la$  es el número de elementos del arreglo  $a$ ) y repetir el proceso. Repetir este procedimiento para todo elemento del arreglo  $b$ .

- Escribir una rutina en C que implante este método.
  - ¿En qué casos este método es más eficiente que el método del texto? ¿En cuáles es menos eficiente?
- 6.5.5. Considere el siguiente método (llamado *intercalación binaria*) de intercalar dos arreglos ordenados  $a$  y  $b$  en un arreglo  $c$ : Sean  $la$  y  $lb$  el número de elementos en  $a$  y  $b$ , respectivamente y suponer que  $la \geq lb$ . Dividir  $a$  en, más o menos,  $lb + 1$  subarreglos iguales. Comparar  $b[0]$  con el elemento menor del segundo subarreglo de  $a$ . Si  $b[0]$  es menor encontrar  $a[i]$  tal que  $a[i] \leq b[0] \leq a[i + 1]$  mediante búsqueda binaria en el primer subarreglo. Ponga todos los elementos del primer subarreglo a partir de  $a$  inclusive  $a[i]$  en  $c$  y luego ponga  $b[0]$  en  $c$ . Repetir este proceso con  $b[1], b[2], \dots, b[j]$ , donde  $b[j]$  es mayor que el menor elemento del segundo subarreglo. Ponga los elementos restantes del primer subarreglo y el primer elemento del segundo en  $c$ . Despues comparar  $b[j]$  con el elemento menor del tercer subarreglo de  $a$  y así de manera sucesiva.
- Escriba un programa para implantar la intercalación binaria.
  - Muestre que si  $la = lb$ , la intercalación binaria actúa como la intercalación descrita en el texto.
  - Muestre que si  $lb = 1$ , la intercalación binaria actúa como la intercalación del ejercicio previo.
- 6.5.6. Determine el número de comparaciones (como función de  $m$  y  $n$ ) que se ejecutan al intercalar dos archivos ordenados  $a$  y  $b$  de tamaño  $n$  y  $m$  respectivamente, para cada uno de los siguientes métodos de intercalación, con cada uno de los siguientes conjuntos de archivos ordenados.
- Métodos de intercalación:
- el método de intercalación presentado en el texto
  - la intercalación del ejercicio 6.5.4
  - la intercalación binaria del ejercicio 6.5.5.
- Conjuntos de archivos:
- $m = n$  y  $a[i] < b[i] < a[i + 1]$  para toda  $i$
  - $m = n$  y  $a[n] < b[1]$
  - $m = n$  y  $a[n/2] < b[1] < b[m] < a[(n/2) + 1]$
  - $n = 2 * m$  y  $a[i] < b[i] < a[i + 1]$  para toda  $i$  entre  $0$  y  $m - 1$
  - $n = 2 * m$  y  $a[m + i] < b[i] < a[n + i + 1]$  para toda  $i$  entre  $0$  y  $m - 1$
  - $n = 2 * m$  y  $a[2 * i] < b[i] < a[2 * i + 1]$  para toda  $i$  entre  $0$  y  $m - 1$
  - $m = 1$  y  $b[0] = a[n/2]$
  - $m = 1$  y  $b[0] < a[0]$
  - $m = 1$  y  $a[m] < b[0]$
- 6.5.7. Genere dos archivos aleatorios de tamaño 100 e intercálelos usando cada uno de los métodos del ejercicio previo, sin perder de vista el número de comparaciones realizadas. Haga lo mismo para dos archivos de tamaño 10 y dos de tamaño 1000. Repita el experimento 10 veces. ¿Qué indican los resultados acerca de la eficiencia promedio de los métodos de intercalación?
- 6.5.8. Escriba una rutina que ordene un archivo aplicando primero el ordenamiento por base a los  $r$  dígitos más significativos (donde  $r$  es una constante dada) y luego una inserción directa para ordenar el archivo completo. Esto elimina pasos excesivos en dígitos de menor orden que pueden no ser necesarios.
- 6.5.9. Escriba un programa que imprima todos los conjuntos de seis enteros positivos  $a1, a2, a3, a4, a5$  y  $a6$ , de manera que



ejemplo, en un archivo de nombres y direcciones, si se usa el estado (provincia) como llave para una búsqueda particular, es probable que no sea único, dado que puede haber dos registros con el mismo estado dentro del archivo. Una llave de ese tipo se llama *llave secundaria*. Algunos de los algoritmos que presentamos suponen llaves únicas; otros permiten llaves duplicadas. Cuando se adopte un algoritmo para una aplicación particular, el programador debe saber si las llaves son únicas y asegurarse de que el algoritmo seleccionado es el adecuado.

Un *algoritmo de búsqueda* es un algoritmo que acepta un argumento *a* y trata de encontrar un registro cuya llave sea *a*. El algoritmo puede dar como resultado el registro entero o, lo que es más común, un apuntador a dicho registro. Es posible que la búsqueda de un argumento particular en una tabla no sea exitosa, es decir, que no exista registro en la tabla que tenga como llave ese argumento. En tal caso el algoritmo puede dar como resultado un "registro nulo" especial o un apuntador nulo. Si la búsqueda es infructuosa, con mucha frecuencia, es deseable agregar un nuevo registro con dicho argumento como llave. Un algoritmo que haga esto se llama algoritmo de *búsqueda e inserción*. A una búsqueda exitosa se le llama con frecuencia una *recuperación*.

A una tabla de registros en la cual se usa una llave para recuperación se le llama con frecuencia *tabla de búsqueda o diccionario*.

En algunos casos es deseable insertar un registro con una llave primaria *key* en un archivo sin buscar primero otro registro con la misma llave. Tal situación podría surgir si ya se determinó que no existe un registro tal en el archivo. En discusiones posteriores investigamos y comentamos acerca de la relativa eficiencia de varios algoritmos. En tales casos el lector debe observar si los comentarios se refieren a una búsqueda, a una inserción o a una búsqueda e inserción.

Obsérvese que no hemos dicho nada acerca de la forma en la cual está organizada la tabla o el archivo. Puede ser un arreglo de registros, una lista ligada, un árbol o incluso un diagrama. Dado que distintas técnicas de búsqueda pueden ser adecuadas para organizaciones de tablas diferentes, con frecuencia se diseña una tabla teniendo en mente una técnica de búsqueda específica. La tabla puede estar contenida en su totalidad en la memoria, en la memoria auxiliar o estar dividida en ambas. Es claro que son necesarias diferentes técnicas de búsqueda bajo esas distintas suposiciones. La búsqueda en la cual toda la tabla está de manera frecuente en la memoria principal se llama *búsqueda interna*, mientras que la búsqueda en la que la mayor parte de la tabla está en la memoria auxiliar se llama *búsqueda externa*. Como en el ordenamiento, nos concentraremos de manera primordial en la búsqueda interna; sin embargo mencionaremos algunas técnicas de búsqueda externa cuando están muy relacionadas con los métodos que estudiamos.

### El diccionario como un tipo de datos abstracto

Una tabla de búsqueda o diccionario puede representarse como un *ADT* (tipo de datos abstracto). Primero suponemos dos tipos de declaraciones de los tipos de llave y registros y una función que extrae la llave de un registro del mismo. También definimos un registro nulo para representar una búsqueda infructuosa.

```

typedef KEYTYPE ... /* un tipo de llave */
typedef RECTYPE ... /* un tipo de registro */
RECTYPE nullrec = ... /* un registro "nulo" */

KEYTYPE keyfunct(r)
RECTYPE r;
{ ... }

```

Podemos entonces representar el tipo de datos abstracto *table* como un simple conjunto de registros. Este es nuestro primer ejemplo de un *ADT* definido como un conjunto y no como una secuencia. Usamos la notación [*eltype*] para denotar un conjunto de objetos de tipo *eltype*. La función *inset(s, elt)* da como resultado *verdadero* si *elt* está en el conjunto *s* y *falso* en caso contrario. La operación de conjuntos *x — y* denota el conjunto *x* eliminando de él todos los elementos de *y*.

```

abstract typedef [rectype] TABLE (RECTYPE);

abstract member(tbl,k)
TABLE(RECTYPE) tbl;
KEYTYPE k;
postcondition if(existe un r en tbl tal que keyfunct(r) == k)
 then member = TRUE
 else member = FALSE

abstract RECTYPE search(tbl,k)
TABLE(rectype) tbl;
keytype k;
postcondition (not member(tbl, k) && (search == nullrec)
 || (member(tbl,k) && keyfunct(search) == k));

abstract inset(tbl,r)
TABLE(RCTYPE) tbl;
RECTYPE r;
precondition member(tbl,keyfunct(r)) == FALSE
postcondition inset(tbl,r);
 (tbl - [r]) == tbl; /* con tbl dividido en dos partes */

abstract delete(tbl, k)
TABLE(RCTYPE) tbl;
KEYTYPE k;
postcondition tbl == (tbl - [search(tbl,k)]);
 (tbl - [keyfunct(k)]) == tbl; /* con tbl dividido en dos partes */

```

Como no se presume que exista relación entre los registros o sus llaves asociadas, la tabla que especificamos se llama *tabla desordenada*. Aunque una tabla como esa permite que los elementos sean recuperados con base en los valores de sus llaves, los elementos no pueden recuperarse en un orden específico. Hay ocasio-

nes en las que, además de las facilidades que proporciona una tabla desordenada, también es necesario recuperar elementos basándose en algún ordenamiento de sus llaves. Una vez establecido un ordenamiento entre las llaves, se hace posible la referencia al primer elemento de una tabla, al último y al sucesor de un elemento dado. Una tabla que cuente con estas facilidades adicionales se llama una *tabla ordenada*. El ADT para una tabla ordenada debe especificarse como una secuencia para indicar el ordenamiento de los registros y no como un conjunto. Dejamos la especificación de ADT como ejercicio al lector.

### Notación algorítmica

La mayoría de las técnicas presentadas en este capítulo se presentan como algoritmos en lugar de programas en C. La razón para ello es que una tabla puede representarse en una amplia variedad de formas. Por ejemplo, una tabla (llaves más registros) organizada como un arreglo podría declararse mediante:

```
#define TABLESIZE 1000
typedef KEYTYPE ...
typedef RECTYPE ...
struct {
 KEYTYPE k;
 RECTYPE r;
} table[TABLESIZE];
```

o como dos arreglos separados:

```
KEYTYPE k[TABLESIZE];
RECTYPE r[TABLESIZE];
```

En el primer caso la  $i$ -ésima llave sería referida como  $table[i].k$ ; en el segundo como  $k[i]$ .

De manera similar, para una tabla organizada como una lista, podría usarse la representación dinámica de una lista o la representación con arreglo de una lista. En el primer caso la llave del registro apuntado por un apuntador  $p$  sería referida como  $node[p].k$ ; en el último, como  $p \rightarrow k$ .

Sin embargo, las técnicas para buscar en esas tablas son muy similares. Así, con el objeto de liberarnos de la necesidad de elegir una representación específica, adoptamos la convención algorítmica de hacer referencia a la llave  $i$ -ésima como  $k(i)$  y a la llave del registro apuntado por  $p$  como  $k(p)$ . De igual forma, hacemos referencia al registro correspondiente como  $r(i)$  o  $r(p)$ . De esta manera podemos concentrar nuestra atención en los detalles de la técnica en lugar de los de la implantación.

### Búsqueda secuencial

La forma más simple de búsqueda es la *búsqueda secuencial*. Esta búsqueda es aplicable a una tabla organizada, ya sea como un arreglo o como una lista ligada. Supongamos que  $k$  es un arreglo de  $n$  llaves, de  $k(0)$  a  $k(n - 1)$  y  $r$  un arreglo de

registros de  $r(0)$  a  $r(n - 1)$  de tal manera que  $k(i)$  es la llave de  $r(i)$ . (Obsérvese que estamos usando la notación algorítmica,  $k(i)$  y  $r(i)$  como se describió de manera previa.) Supongamos también que  $key$  es un argumento de búsqueda. Queremos obtener el entero  $i$  más pequeño tal que  $k(i)$  sea igual a  $key$  si existe tal  $i$  y  $-1$  en caso contrario. El algoritmo para hacerlo es el siguiente:

```
for (i = 0; i < n; i++)
 if (key == k(i))
 return(i);
return (-1);
```

El algoritmo examina cada llave en turno; al encontrar una que coincida con el argumento de la búsqueda, da como resultado su índice (que actúa como apuntador a su registro). Si ninguna coincide el resultado es  $-1$ .

Este algoritmo puede modificarse con facilidad para agregar un registro  $rec$  con llave  $key$  a la tabla si  $key$  aún no está en la misma. La última instrucción se modifica como sigue:

```
k(n) = key; /* insertar la nueva llave */
r(n) = rec; /* y el registro */
n++; /* incrementar el tamaño de la tabla */
return(n - 1);
```

Obsérvese que si se hacen inserciones usando sólo el algoritmo modificado anterior, dos registros no pueden tener la misma llave. Cuando este algoritmo se implanta en C, debemos asegurarnos que el incremento de  $n$  no haga que su valor exceda el límite superior del arreglo. Para usar la búsqueda secuencial con inserción en un arreglo, debe haberse asignado con anterioridad memoria suficiente para el mismo.

Un método de búsqueda aún más eficiente involucra la inserción de la llave del argumento al final del arreglo antes de comenzar la búsqueda, garantizando así que la llave será encontrada.

```
k(n) = key;
for (i = 0; key != k(i); i++)
;
if (i < n)
 return(i);
else
 return(-1);
```

Para una búsqueda e inserción, la instrucción *if* completa se remplaza por

```
if (i == n)
 r(n++) = rec;
return(i);
```

La llave extra insertada al final del arreglo se llama un *centinela*.

Almacenar una tabla como una lista ligada tiene la ventaja de que el tamaño de la tabla puede aumentar la manera dinámica cuando sea necesario. Supongamos que la tabla está organizada como una lista lineal ligada apuntada por *table* y ligada mediante un campo apuntador *next*. Entonces, suponiendo *k*, *r*, *key* y *rec* como antes, la búsqueda secuencial con inserción para una lista ligada puede escribirse de la siguiente manera:

```
q = null;
for (p = table; p != null && k(p) != key; p = next(p))
 q = p; /* lo que significa que k(p) == KEY */
if (p != null)
 return(p);
/* insertar un nuevo nodo */
s = getnode();
k(s) = key;
r(s) = rec;
next(s) = null;
if (q == null)
 table = s;
else
 next(q) = s;
return(s);
```

La eficiencia de la búsqueda en una lista puede perfeccionarse mediante la misma técnica que acabamos de sugerir para un arreglo. Se puede agregar un nodo centinela que contenga la llave del argumento al final de la lista antes de comenzar la búsqueda de manera que la condición en la iteración *for* sea la condición simple *k(p) != key*. Sin embargo, el método del centinela requiere de guardar un apuntador externo adicional al último nodo de la lista. Dejamos al lector los detalles adicionales (como, por ejemplo, qué ocurre con el nodo recién agregado cuando se encuentra la llave dentro de la lista).

La eliminación de un registro de una tabla almacenada como un arreglo desordenado se implanta remplazando el registro a ser eliminado por el último registro del arreglo y reduciendo el tamaño de la tabla en 1. Si el arreglo está ordenado de alguna manera (aun si el orden no es según las llaves), este método no puede usarse, y en promedio la mitad de los elementos en el arreglo tiene que ser recorrida. (¿Por qué?) Si la tabla está almacenada como una lista ligada, es muy eficiente eliminar un elemento sin tener en cuenta si está o no ordenada.

### Eficiencia de la búsqueda secuencial

¿Cuán eficiente es una búsqueda secuencial? Examinemos el número de comparaciones hechas por una búsqueda secuencial cuando se busca una llave dada. Suponemos que no se realizan inserciones ni eliminaciones, de manera que estamos buscando en una tabla de tamaño constante, *n*. El número de comparaciones depende del lugar de la tabla donde aparece el registro que tiene la llave del argumento. Si

el registro es el primero en la tabla, se realiza una sola comparación; si el registro es el último, se necesitan *n* comparaciones. Si es igualmente probable que el argumento aparezca en cualquier posición dada en la tabla, una búsqueda exitosa haría (en el promedio)  $(n + 1)/2$  comparaciones y, una infructuosa *n* comparaciones. En cualquier caso, el número de comparaciones es  $O(n)$ .

Sin embargo, es normal el caso en que algunos argumentos se presentan con más frecuencia que otros al algoritmo de búsqueda. Por ejemplo, en los archivos de la secretaría de asuntos escolares de una escuela, es más probable que se requieran los registros de alumnos de último año que piden sus calificaciones para ingresar a estudios de postgrado o de alumnos de primer año a los que se les están actualizando sus promedios de bachillerato que los registros de alumnos del segundo y tercer años. De manera similar, es más probable que los registros de personas que no respetan la ley o evasores de impuestos sean recuperados de los archivos de la oficina de tránsito o de Hacienda que los de un ciudadano que respeta la ley. (Como veremos más adelante en este capítulo, estos ejemplos no son realistas dado que es improbable que sea usada la búsqueda secuencial en archivos tan largos; pero por el momento, supongamos que se está usando la búsqueda secuencial.) Entonces, si se colocan al principio del archivo los registros que se acceden con mayor frecuencia, el número promedio de comparaciones se reduce de manera considerable, ya que los registros más accesados pueden recuperarse en un menor tiempo.

Sea *p(i)* la probabilidad de que el registro *i* sea recuperado. (*p(i)* es un número entre 0 y 1 tal que si se hacen *m* recuperaciones del archivo,  $m * p(i)$  serán de *r(i)*.) Supongamos también que  $p(0) + p(1) + \dots + p(n - 1) = 1$ , de manera que no haya posibilidad de que una llave del argumento no se encuentre en la tabla. Entonces el número promedio de comparaciones en la búsqueda del registro es

$$p(0) + 2 * p(1) + 3 * p(2) + \dots + n * p(n - 1)$$

Es claro que, este número se minimiza si

$$p(0) \geq p(1) \geq p(2) \geq \dots \geq p(n - 1)$$

(¿Por qué?) Así, dado un archivo extenso y estable, el reordenamiento del mismo según la probabilidad de recuperación en orden decreciente alcanza un mayor grado de eficiencia cada vez que se busca en el archivo.

Una estructura de lista es preferible a un arreglo si se tienen que ejecutar muchas inserciones y eliminaciones en una tabla. Sin embargo, aun en una lista sería mejor mantener la relación

$$p(0) \geq p(1) \geq \dots \geq p(n - 1)$$

para asegurar una búsqueda secuencial eficiente. Esto puede hacerse con mayor facilidad si se inserta en la lista un nuevo elemento en la posición que le corresponde. Si *prob* es la probabilidad de un registro con una llave dada sea el argumento de búsqueda, ese registro debería insertarse entre los registros *r(i)* y *r(i + 1)* donde *i* es tal que se cumple:

$$p(i) \geq prob \geq p(i+1)$$

Por supuesto, este método implica que se guarda un campo extra  $p$  con cada registro o que  $p$  pueda ser computado basándose en alguna otra información del registro.

### Reordenamiento de una lista para alcanzar máxima eficiencia de búsqueda

Desafortunadamente, es raro que se conozcan de antemano las probabilidades  $p(i)$ . Aunque es común que ciertos registros sean recuperados con mayor frecuencia que otros, es casi imposible identificar con anticipación dichos registros. También, la probabilidad de que un determinado registro sea recuperado puede cambiar a través del tiempo. Para usar el ejemplo de la escuela dado antes, un estudiante inicia como alumno de primer año (con alta probabilidad de que su registro sea recuperado) y luego se convierte en alumno de segundo y tercer años (con poca probabilidad de ser recuperado) antes de convertirse en alumno del último año (otra vez con alta probabilidad de ser recuperado). Así, sería de gran ayuda tener un algoritmo que reordenara de manera continua la tabla, de tal forma que los registros que se acceden con mayor frecuencia estuvieran al frente y los que se acceden con menor frecuencia al final.

Hay dos métodos de búsqueda para realizar lo anterior. Uno de ellos se conoce como método de *moverse-al-frente* y es eficiente sólo en el caso de una tabla organizada como una lista. En este método, siempre que una búsqueda es exitosa (es decir, cuando el argumento coincide con la llave de un registro dado), el registro recuperado se elimina de su localización actual en la lista y se coloca a la cabeza de la misma.

El otro método se llama *transposición*, en el cual un registro recuperado se intercambia con el registro que lo precede de manera inmediata. Presentamos un algoritmo para implantar el método de transposición en una tabla almacenada en forma de lista ligada. El algoritmo da como resultado un apuntador al registro recuperado o el apuntador nulo si no se encuentra el registro. Como antes,  $key$  es el argumento de búsqueda,  $k$  y  $r$  son las tablas de llaves y registros.  $table$  es un apuntador al primer nodo de la lista.

```

q = s = null; /* q se encuentra un paso detrás de p; */
/* s está dos pasos detrás de p */
for (p = table; p != null && k(p) != key; p = next(p)) {
 s = q;
 q = p;
} /* fin de for */
if (p == null)
 return (p);
/* Se ha encontrado el registro en la posición p. */
/* Transponer los registros apuntados por p y q. */
if (q == null)
 /* la llave está en la primera posición de la tabla. */

```

```

/*
 * No se requiere la transposición
 */
return(p);
/* transponer node(q) y node (p). */
next(q) = next(p);
next(p) = q;
(s == null) ? table = p : next(s) = p;
return (p);

```

Obsérvese que las dos instrucciones *if* en el algoritmo anterior pueden combinarse dentro de la instrucción simple *if*( $p == null || q == null$ ) *return* ( $p$ ), para ser más concisos. Dejamos como ejercicio al lector la implantación del método de transposición para un arreglo y el método de moverse-al-frente.

Ambos métodos están basados en el fenómeno observado de que un registro que ha sido recuperado tiene probabilidad de ser recuperado de nuevo. Adelantando dichos registros hacia al frente de la tabla, las recuperaciones subsecuentes serán más eficientes. La racionalidad del método de moverse-al-frente consiste en que, ya que el registro tiene probabilidad de ser recuperado de nuevo, se coloque en la posición de la tabla desde la cual dicha recuperación sea más eficiente. Sin embargo, el contraargumento para el método de transposición es que una sola recuperación no implica aún que el registro será recuperado con frecuencia; colocándolo al frente de la tabla se reduce la eficiencia de la búsqueda para todos los otros registros que antes le precedían. Adelantando el registro una sola posición cada vez que es recuperado, aseguramos que avance al frente de la lista sólo si se recupera con frecuencia.

Se ha mostrado que el método de transposición es más eficiente para un gran número de búsquedas con una distribución de probabilidad que no cambie. Sin embargo, el método de moverse-al-frente da mejores resultados para un número pequeño o medio de búsquedas y responde con mayor rapidez ante un cambio en la distribución de probabilidad. Tiene también un mejor comportamiento en el peor de los casos que el de transposición. Por esta razón, es preferible el método de moverse-al-frente en la mayoría de las situaciones prácticas que involucran la búsqueda secuencial.

Si se requiere un gran número de búsquedas con una distribución de probabilidad que no cambie, lo mejor puede ser una estrategia mezclada: usar para las primeras  $s$  búsquedas el método de moverse-al-frente para organizar la lista de manera rápida en una buena secuencia y luego cambiar al método de transposición para obtener un comportamiento aún mejor. El valor exacto de  $s$  para optimizar la eficiencia global depende de la longitud de la lista y de la exacta distribución de probabilidad de acceso.

Una ventaja del método de transposición sobre el método de moverse-al-frente es que se puede aplicar de manera eficiente a tablas almacenadas en forma de arreglo tanto como a tablas estructuradas como listas. La transposición de dos elementos en un arreglo es una operación muy eficiente, mientras que mover un elemento de en medio de un arreglo al frente implica (en promedio) mover la mitad del arreglo. (Sin embargo, en este caso el número promedio de movimientos no es tan grande, dado que el elemento que debe ser movido con más frecuencia, viene de la porción superior del arreglo.)

## Búsqueda en una tabla ordenada

Si la tabla se almacena en orden ascendente o descendente de las llaves de los registros, pueden usarse varias técnicas para mejorar la eficiencia de la búsqueda. Esto es cierto en especial si la tabla es de tamaño fijo. Una ventaja obvia de la búsqueda en un archivo ordenado se tiene cuando la llave del argumento no está presente en el archivo. En el caso de un archivo desordenado, se necesitan  $n$  comparaciones para detectar este hecho. En el caso de un archivo ordenado, suponiendo que las llaves argumentos están distribuidas de manera uniforme sobre el rango de llaves en el archivo, se necesitan (en promedio) sólo  $n/2$  comparaciones. Esto ocurre porque sabemos que una llave está faltando en un archivo ordenado de manera ascendente tan pronto como encontramos una llave que sea mayor que el argumento.

Supóngase que es posible reunir un gran número de peticiones de recuperación antes de que alguna de ellas sea procesada. Por ejemplo, en muchas aplicaciones la respuesta a una petición de información puede diferirse al día siguiente. En tal caso, se pueden reunir todas las peticiones de un día específico y la búsqueda real puede hacerse durante la noche cuando no están entrando nuevas peticiones. Si tanto la tabla como la lista de peticiones están ordenadas, la búsqueda secuencial puede llevarse a cabo para ambas a la vez. Así, no es necesario recorrer la tabla entera para cada petición de búsqueda. En realidad, si hay muchas de esas peticiones distribuidas uniformemente sobre toda la tabla, cada petición requerirá sólo unas cuantas consultas (si el número de peticiones es menor que el número de entradas de la tabla) o quizás una sola comparación (si el número de peticiones es mayor que el número de entradas de la tabla). En situaciones de ese tipo, la búsqueda secuencial es quizás el mejor método a utilizar.

### La búsqueda secuencial indexada

Hay otra técnica para perfeccionar la eficiencia de la búsqueda en un archivo ordenado, pero involucra un incremento en la cantidad de espacio requerido. Este método se llama *método de búsqueda secuencial indexada*. Se aparta una tabla auxiliar, llamada *índice* además del propio archivo ordenado. Cada elemento en el *índice* consta de una llave *kindex* y un apuntador al registro del archivo que corresponde a *kindex*. Los elementos en el índice tanto como los elementos en el archivo, tienen que estar ordenados de acuerdo a las llaves. Si el índice es un octavo del tamaño del archivo, cada octavo registro del archivo tiene que estar representado en el índice. Esto se ilustra en la figura 7.1.1.

El algoritmo usado para buscar en un archivo secuencial indexado es de forma directa. Sean  $r$ ,  $k$  y  $key$  definidas como antes,  $kindex$  un arreglo de llaves del índice y  $pindex$  un arreglo de apuntadores dentro del índice que apuntan a los registros reales

| <i>k</i><br>(Llave) | <i>r</i><br>(Registro) |
|---------------------|------------------------|
| 8                   | 12                     |
| 14                  |                        |
| 26                  |                        |
| 38                  |                        |
| 72                  |                        |
| 115                 |                        |
| 306                 |                        |
| 321                 |                        |
| 329                 |                        |
| 387                 |                        |
| 409                 |                        |
| 512                 |                        |
| 540                 |                        |
| 567                 |                        |
| 583                 |                        |
| 592                 |                        |
| 602                 |                        |
| 611                 |                        |
| 618                 |                        |
| 741                 |                        |
| 798                 |                        |
| 811                 |                        |
| 814                 |                        |
| 876                 |                        |

**Indice**

**kindex pindex**

**Figura 7.1.1** Un archivo secuencial indexado.

del archivo. Suponemos que el archivo está ordenado como un arreglo, que  $n$  es el tamaño del archivo y que  $\text{idxsze}$  es el tamaño del índice.

```

 for (i = 0; i < idxsize && kindex(i) <= key; i++)
 ;
 lowlim = (i == 0) ? 0 : pindex(i - 1);
 hilim = (i == idxsize) ? n - 1 : pindex(i) - 1;
 for (j = lowlim; j <= hilim && k(j) != key; j++)
 ;
 return ((j > hilim) ? -1 : j);
 }
}

```

Obsérvese que en el caso de registros múltiples con la misma llave, el algoritmo anterior no da como resultado un apuntador al primero de tales registros en todos los casos.

Lá ventaja real del método secuencial indexado es que los elementos de la tabla pueden examinarse de manera secuencial si todos los registros del archivo tienen que ser accedidos, sin embargo el tiempo de búsqueda de un elemento en particular se

reduce en forma considerable. Se ejecuta una búsqueda secuencial en el índice, que es menor que la tabla. Una vez que se ha encontrado la posición correcta en el índice se ejecuta una segunda búsqueda secuencial sobre una porción menor de la propia tabla de registros.

El uso de un índice es aplicable a una tabla ordenada almacenada tanto como una lista ligada, que como un arreglo. Usar una lista ligada implica una gran sobre-carga de espacio para apuntadores, aunque las inserciones y eliminaciones pueden ejecutarse con mucha mayor facilidad.

Si la tabla es tan grande que incluso el uso de un índice no alcanza suficiente eficiencia (ya sea porque el índice es extenso con el objetivo de reducir la búsqueda secuencial en la tabla o que el índice es pequeño de manera que las llaves adyacentes del índice están muy alejadas una de otra en la tabla), se puede usar un índice secundario. El índice secundario actúa como un índice al índice primario que apunta a las entradas de la tabla secuencial. Esto se ilustra en la figura 7.1.2.

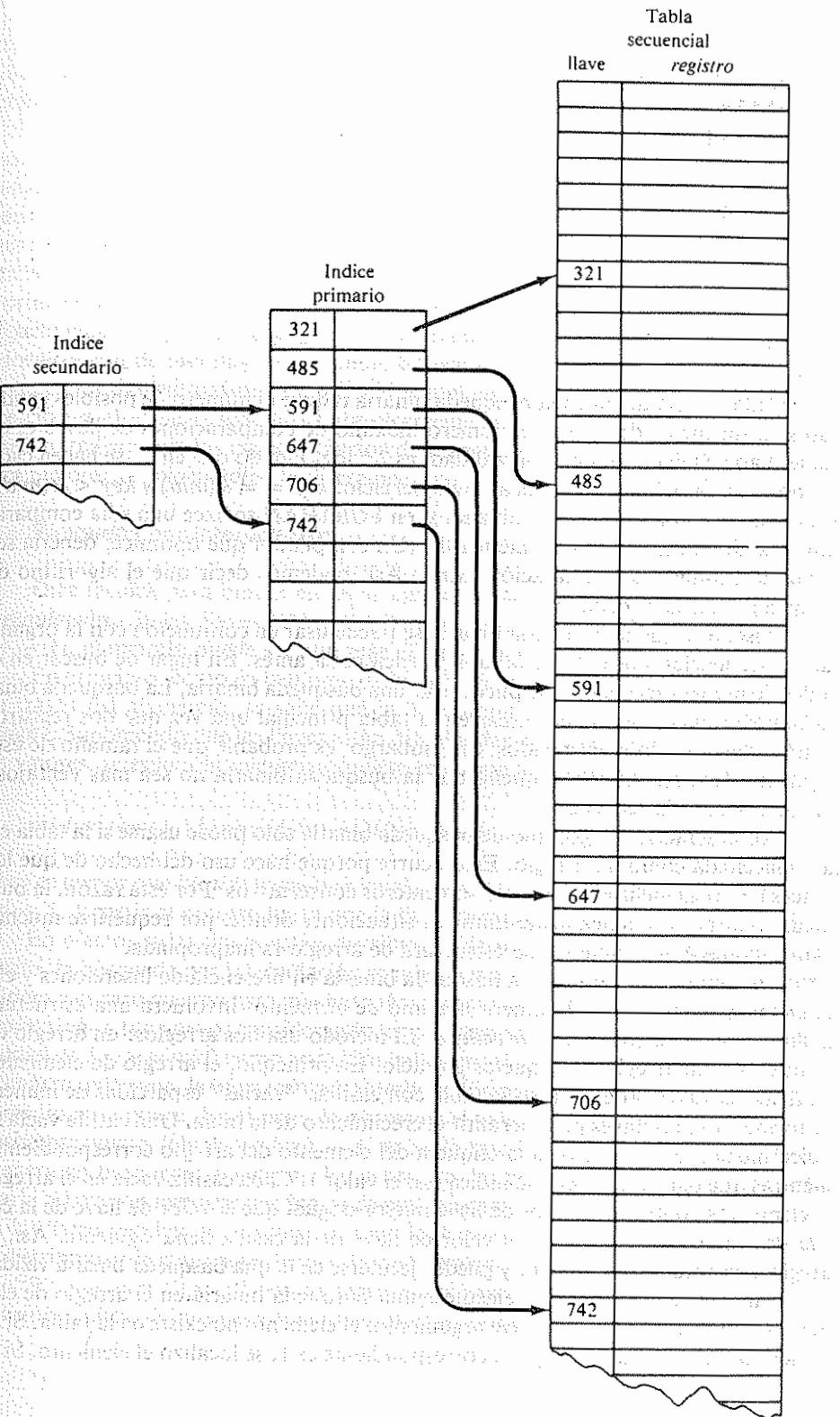
Las eliminaciones de una tabla secuencial indexada se pueden hacer con mayor facilidad etiquetando las entradas eliminadas. En la búsqueda secuencial a lo largo de la tabla, se ignoran las entradas eliminadas. Obsérvese que si se elimina un elemento, incluso si su llave está en el índice, nada tiene que hacerse al índice; sólo se etiqueta la entrada de la tabla original.

La inserción en una tabla secuencial indexada es más difícil, dado que puede no haber espacio entre dos entradas de la tabla ya existentes, necesitándose así un corrimiento de un gran número de elementos de la tabla. Sin embargo, si ha sido etiquetado un elemento cercano en la tabla cuando se eliminó, se necesita recorrer sólo unos pocos y escribir sobre el elemento eliminado. Esto puede requerir de una alteración del índice si se recorre un elemento apuntado por un elemento del índice. Un método alternativo es mantener un área de desborde en alguna otra localización y ligarla a cualquier registro insertado. Sin embargo, esto requeriría un campo apuntador extra en cada registro de la tabla original. Dejamos como ejercicio la exploración de esas posibilidades.

## La búsqueda binaria

El método de búsqueda más eficiente en una tabla secuencial sin usar índices o tablas auxiliares es el de búsqueda binaria. El lector debería estar familiarizado con esta técnica de búsqueda de las secciones 3.1 y 3.2. Básicamente, se compara el argumento con la llave del elemento medio de la tabla. Si son iguales, la búsqueda termina con éxito; en caso contrario, se busca de manera similar en la mitad superior o inferior de la tabla.

En el capítulo 3 observamos que la búsqueda binaria puede definirse de la mejor manera en forma recursiva. Como resultado fueron presentados una definición recursiva, un algoritmo recursivo y un programa recursivo para la búsqueda binaria. Sin embargo, la sobrecarga asociada a la recursividad puede hacerla inapropiada para su uso en situaciones prácticas en las cuales la eficiencia es una consideración primordial. En consecuencia, presentamos la siguiente versión no recursiva del algoritmo de búsqueda binaria:



**Figura 7.1.2** Uso de un índice secundario.

```

low = 0;
hi = n - 1;
while (low <= hi) {
 mid = (low + hi)/2;
 if (key == k(mid))
 return(mid);
 if (key < k(mid))
 hi = mid - 1;
 else
 low = mid + 1;
} /* fin de while */
return(-1);

```

Cada comparación en la búsqueda binaria reduce el número de posibles candidatos en un factor de 2. Así, el número máximo de comparaciones de llaves es de manera aproximada  $\log_2 n$ . (En realidad, es  $2 * \log_2 n$  dado que en C, se hacen cada vez dos comparaciones de llaves a través del ciclo:  $key == k(mid)$  y  $key < k(mid)$ ). Sin embargo, en lenguaje ensamblador o en FORTRAN se hace una sola comparación usando la instrucción aritmética IF. (Un compilador que optimice, debería ser capaz de eliminar la comparación extra.) Así, podemos decir que el algoritmo de búsqueda binaria es  $O(\log n)$ .

Obsérvese que la búsqueda binaria se puede usar en conjunción con la organización secuencial indexada de la tabla mencionada antes. En lugar de buscar en el índice de manera secuencial, se puede usar una búsqueda binaria. La búsqueda binaria también puede usarse al buscar en la tabla principal una vez que dos registros frontera hayan sido identificados. Sin embargo, es probable que el tamaño de este segmento de tabla sea tan pequeño que la búsqueda binaria no sea más ventajosa que una búsqueda secuencial.

Por desgracia, el algoritmo de búsqueda binaria sólo puede usarse si la tabla está almacenada como un arreglo. Esto ocurre porque hace uso del hecho de que los índices de los elementos del arreglo son enteros consecutivos. Por esta razón, la búsqueda binaria es prácticamente inútil en situaciones donde, por requerirse muchas eliminaciones e inserciones, una estructura de arreglo es inapropiada.

Un método para utilizar la búsqueda binaria en presencia de inserciones y eliminaciones si se conoce el número máximo de elementos involucra una estructura de datos conocida como *lista de relleno*. El método usa dos arreglos: un arreglo de elementos y un arreglo de etiquetas paralelo. En principio, el arreglo de elementos contiene las llaves ordenadas de la tabla con casillas "vacías" espaciadas de manera uniforme entre las llaves para permitir el crecimiento de la tabla. Una casilla vacía se indica mediante un valor 0 en la etiqueta del elemento del arreglo correspondiente, mientras que una casilla llena se indica por el valor 1. Cada casilla vacía en el arreglo de elementos contiene un valor de llave mayor o igual que el valor de llave de la casilla llena previa y menor que el valor de llave de la casilla llena siguiente. Así, el arreglo completo está ordenado, y puede ejecutarse en él una búsqueda binaria válida.

Para buscar un elemento ejecútense una búsqueda binaria en el arreglo de elementos. Si no se encuentra la llave argumento, el elemento no existe en la tabla. Si se encuentra y el valor de la etiqueta correspondiente es 1, se localizó el elemento. Si el

valor de la etiqueta correspondiente es 0, verifíquese si la casilla llena previa contiene la llave argumento. Si es así, se encontró el elemento, si no, no existe tal elemento en la tabla.

Para insertar un elemento, localícese primero su posición. Si la posición está vacía insértese el elemento en la misma, haga el valor de su etiqueta igual a 1 e igualense los contenidos de todas las posiciones vacías contiguas previas a los contenidos del elemento lleno previo y todos los contenidos de las posiciones vacías contiguas siguientes al elemento insertado dejando sus etiquetas iguales a 0. Si la posición está llena, recórranse todos los elementos siguientes una posición hacia delante hasta la primera posición vacía (escribiendo sobre la primera posición vacía y poniendo su etiqueta igual a 1) para hacer lugar al nuevo elemento. La eliminación sólo involucra la localización de una llave y el cambio del valor de su etiqueta asociada a 0. Por supuesto las desventajas de este método son el corrimiento que debe hacerse en la inserción y el espacio limitado para el crecimiento. De manera periódica puede desearse redistribuir los espacios vacíos de manera uniforme a lo largo del arreglo para mejorar la velocidad en la inserción.

### Búsqueda por interpolación

Otra técnica para buscar en un arreglo ordenado es la llamada *búsqueda por interpolación*. Si las llaves están distribuidas de manera uniforme entre  $k(0)$  y  $k(n - 1)$ , el método puede ser aun más eficiente que la búsqueda binaria.

En principio, como en la búsqueda binaria,  $low$  se hace 0 y  $high$  se hace  $n - 1$ , y a través del algoritmo, se sabe que la llave argumento  $key$  está entre  $k(low)$  y  $k(high)$ . Suponiendo que las llaves están distribuidas de manera uniforme entre esos dos valores, se esperaría que  $key$  estuviese en forma aproximada en la posición

$$mid = low + (high - low) * ((key - k(low)) / (k(high) - k(low)))$$

Si  $key$  es menor que  $k(mid)$  haga  $high$  igual a  $mid - 1$ ; si es mayor, haga  $low$  igual a  $mid + 1$ . Repetir el proceso hasta que la llave haya sido encontrada o  $low > high$ .

En efecto, si las llaves están distribuidas de manera uniforme a lo largo del arreglo, la búsqueda por interpolación requiere un promedio de  $\log_2(\log_2 n)$  comparaciones y es raro que requiera muchas más, comparado con la búsqueda binaria que requiere  $\log_2 n$  (de nuevo, considerando las dos comparaciones para igualdad y desigualdad de  $key$  y  $k(mid)$  como una). Sin embargo, si las llaves no están distribuidas de manera uniforme, la búsqueda por interpolación puede tener un comportamiento promedio muy pobre. En el peor de los casos, el valor de  $mid$  puede ser de manera consistente igual a  $low + 1$  o  $high - 1$ , en cuyo caso la búsqueda por interpolación degenera en búsqueda secuencial. En contraste, las comparaciones en la búsqueda binaria nunca son mayores que  $\log_2 n$  de manera aproximada. En situaciones prácticas, las llaves tienden con frecuencia a agruparse en torno a ciertos valores y no están distribuidas de manera uniforme. Por ejemplo, hay más nombres que comienzan con "S" que con "Q", y por lo tanto habrá más Suárez y menos Quiroz. En situaciones tales, la búsqueda binaria es muy superior a la de interpolación.

Una variación de la búsqueda por interpolación, llamada *búsqueda por interpolación robusta* (o *búsqueda rápida*), intenta remediar el pobre comportamiento práctico de la búsqueda por interpolación a la vez que extiende su ventaja sobre la búsqueda binaria a distribuciones no uniformes de las llaves. Esto se hace estableciendo un valor *gap* de manera que *mid* — *low* y *high* — *mid* sean siempre mayores que *gap*. Al inicio, se hace *gap* igual a  $\sqrt{high - low + 1}$ . *Probe* se hace igual a  $low + (high - low) * ((key - k(low)) / (k(high) - k(low)))$ , y *mid* se hace igual a  $\min(hight - gap, \max(probe, low + gap))$  (donde *mín* y *máx* dan como resultado el mínimo y máximo respectivos de dos valores). Es decir, garantizamos que la siguiente posición usada para la comparación (*mid*) esté como mínimo a *gap* posiciones de los extremos del intervalo, donde *gap* es como mínimo la raíz cuadrada del intervalo. Cuando se encuentra que la llave argumento está restringida al más pequeño de los dos intervalos, de *low* a *mid* y de *mid* a *high*, se iguala *gap* a la raíz cuadrada del tamaño del nuevo intervalo. Sin embargo, si la llave argumento está en el mayor de los dos intervalos, se duplica el valor de *gap*, aunque nunca se permite que sea mayor que la mitad del tamaño del intervalo. Esto garantiza escapar de un agrupamiento extenso de valores de llaves similares.

El número esperado de comparaciones para la búsqueda de interpolación robusta para una distribución aleatoria de llaves es  $O(\log \log n)$ . Esto es superior a la búsqueda binaria. En una lista de aproximadamente 40 000 nombres, la búsqueda binaria requiere un promedio en forma aproximada de 16 comparaciones de llaves. A causa del agrupamiento de los nombres en situaciones prácticas, la búsqueda por interpolación requiere de 134 comparaciones promedio en un experimento real, mientras que la búsqueda por interpolación robusta requiere sólo 12.5. En una lista distribuida de manera uniforme de aproximadamente 40 000 elementos — $\log_2(40\,000)$  es alrededor de 3.9— la búsqueda por interpolación robusta requiere 6.7 comparaciones promedio. (Debe observarse que el tiempo de computación extra requerido por la búsqueda por interpolación robusta puede ser sustancial pero se ignora en estos hallazgos.) El peor caso para la búsqueda por interpolación robusta es  $(O(\log n)^2)$  comparaciones, que es más grande que para la búsqueda binaria, pero mucho mejor que el  $O(n)$  de la búsqueda de interpolación normal.

Sin embargo, en la mayoría de las computadoras, los cálculos requeridos por la búsqueda por interpolación son muy lentas, dado que involucran aritmética con las llaves y multiplicaciones y divisiones complicadas. La búsqueda binaria sólo requiere aritmética en índices enteros y división entre 2 que puede ejecutarse de manera eficiente recorriendo un bit a la derecha. Así, los requerimientos computacionales de la búsqueda por interpolación, ocasionan con frecuencia que ésta se ejecute más despacio que la búsqueda binaria aun cuando la primera requiere menos comparaciones.

## EJERCICIOS

- 7.1.1. Modifique los algoritmos de búsqueda e inserción de esta sección de manera que se conviertan en algoritmos de actualización. Si un algoritmo encuentra un *i* tal que *key* sea igual a *k(i)*, cambiar el valor de *r(i)* por *rec*.

- 7.1.2. Implante los algoritmos de búsqueda secuencial y de búsqueda secuencial e inserción en C para las representaciones con arreglo y con lista ligada.
- 7.1.3. Compare la eficiencia de la búsqueda en una tabla secuencial ordenada de tamaño *n* y la búsqueda en una tabla desordenada del mismo tamaño para la llave *key*
- si no está presente un registro con la llave *key*
  - si hay un registro con la llave *key* y se busca sólo uno
  - si hay más de un registro con la llave *key* y sólo se desea encontrar el primero
  - si hay más de un registro con la llave *key* y se desea encontrar a todos
- 7.1.4. Asuma que una tabla ordenada se almacena como una lista circular con dos apunadores externos: *table* y *other*. *table* apunta siempre al nodo que contiene el registro con la menor llave. *other* es al principio igual a *table* pero se hace apuntar al registro recuperado cada vez que se ejecuta una búsqueda. Si la búsqueda es infructuosa, *other* se hace igual a *table*. Escribir una rutina en C *search(table, other, key)* que implante este método y dé como resultado un apuntador al registro recuperado o un apuntador nulo si la búsqueda resulta infructuosa. Explicar cómo manteniendo el apuntador *other* puede reducir el número promedio de comparaciones en una búsqueda.
- 7.1.5. Considere una tabla ordenada implantada como un arreglo o como una lista doblemente ligada de manera que se pueda hacer una búsqueda secuencial en la tabla hacia adelante o hacia atrás. Suponer que un solo apuntador *p* apunta al último registro recuperado de manera exitosa. La búsqueda comienza siempre en el registro apuntado por *p* pero puede proseguir en cualquier dirección. Escribir una rutina *search(table, p, key)* para el caso de un arreglo y una lista doblemente ligada que recupere un registro con llave *key* y modifique *p* de acuerdo a ello. Demostrar que las comparaciones de llaves en ambos casos, si se localiza o no la llave, son las mismas que en el caso del ejercicio previo, en el cual la tabla puede ser examinada en una sola dirección pero el proceso de examen puede comenzar en uno de dos puntos.
- 7.1.6. Considere un programador que escribe el siguiente programa:

```
if (c1)
 if (c2)
 if (c3)
 ...
 if (cn)
 { enunciado }
```

donde *c<sub>i</sub>* es una condición que puede ser verdadera o falsa. Note que la reubicación de las condiciones en orden diferente da como resultado un programa equivalente, dado que [instrucción] sólo se ejecuta si todas las *c<sub>i</sub>* son verdaderas. Asumir que *time(i)* es el tiempo necesario para evaluar la condición *c<sub>i</sub>* y que *prob(i)* es la probabilidad de que *c<sub>i</sub>* sea verdadera. ¿En qué orden deberían ser redispuestas las condiciones para hacer el programa más eficiente?

- 7.1.7. Modifique la búsqueda secuencial indexada de manera que dé como resultado el primer registro de la tabla en caso de que haya varios con la misma llave.
- 7.1.8. Considere la siguiente implantación en C de un archivo secuencial indexado:

```
#define INDEXSIZE 100;
#define TABLESIZE 1000;
```

```

struct indxtype {
 int kindex;
 int pindex;
};

struct tabletype {
 int k;
 int r;
 int flag;
};

struct isfiletype {
 struct indxtype indx[INDXSIZ];
 struct tabletype table[TABLESIZE];
};

struct isfiletype isfile;

```

Escriba una rutina en C, *create(isfile)* que inicialice un archivo tal de datos de entrada. Cada línea de entrada contiene una llave y un registro. La entrada está ordenada de acuerdo a las llaves en orden ascendente. Cada entrada de índice corresponde a diez entradas de la tabla. *flag* se hace igual a *VERDADERO* en una entrada de la tabla que esté ocupada e igual a *FALSO* en una que esté desocupada. Dos de cada diez entradas de la tabla se dejan desocupadas para permitir el crecimiento futuro.

- 7.1.9. Dado un archivo secuencial indexado como en el ejercicio previo, escriba una rutina en C *search(isfile, key)* para imprimir el registro en el archivo con la llave *key* si hay uno y una indicación de que no lo hay si no existe un registro con dicha llave. (¿Cómo puede asegurar que una búsqueda infructuosa es tan eficiente como sea posible?) También, escriba rutinas *insert(isfile, key, rec)* para insertar un registro *rec* con llave *key* y *delete(isfile, key)* para eliminar un registro con llave *key*.
- 7.1.10. Considere la siguiente versión de la búsqueda binaria, que supone que las llaves están contenidas en  $k(1)$  a  $k(n)$  y que el elemento especial  $k(0)$  es menor que cualquier llave posible:

```

mid = n/2;
len = (n - 1)/2;
while (key != k(mid)) {
 if (key < k(mid))
 mid -= len/2;
 else
 mid += len/2;
 if (len == 0)
 return(-1);
 len /= 2;
} /* fin de while */
return(mid);

```

Demuestre que este algoritmo es correcto. ¿Cuáles son las ventajas y/o desventajas de este método con respecto al método presentado en el texto?

- 7.1.11. El siguiente algoritmo de búsqueda en un arreglo ordenado que se conoce como *búsqueda de Fibonacci* a causa del uso de los números de Fibonacci. (Para una definición de los números de Fibonacci y de la función *fib* ver la sección 3.1.)

```

for (j = 1; fib(j) < n; j++)
 ;
mid = n - fib(j - 2) + 1;
f1 = fib(j - 2);
f2 = fib(j - 3);
while (key != k(mid)) {
 if (mid < 0 || key > k(mid)) {
 if (f1 == 1)
 return(-1);
 mid += f2;
 f1 -= f2;
 f2 = f1;
 } else {
 if (key < k(mid))
 if (f2 == 0)
 return(-1);
 mid -= f2;
 t = f1 - f2;
 f1 = f2;
 f2 = t;
 } /* fin de if */
 }
 return(mid);
}

```

Explique cómo trabaja este algoritmo. Confrontar el número de comparaciones de llaves con el número que realiza la búsqueda binaria. Modificar la porción inicial de este algoritmo de manera que calcule los números de Fibonacci de manera eficiente, en lugar de buscarlos en una tabla o computarlos cada vez de nuevo.

- 7.1.12. Modifique la búsqueda binaria del texto de manera que en el caso de una búsqueda infructuosa, dé como resultado el índice *i* tal que  $k(i) < key < k(i + 1)$ . Si  $key < k(0)$ , dé como resultado  $-1$  y si  $key > k(n - 1)$ , da  $n - 1$ . Hacer lo mismo para los métodos de búsqueda de los ejercicios 7.1.10 y 7.1.11.

## 7.2. BUSQUEDA EN ARBOLES

En la sección 7.1 discutimos operaciones de búsqueda en un archivo que estaba organizado bien como un arreglo o como una lista. En esta sección consideraremos varias maneras de organizar archivos como árboles y algunos algoritmos de búsqueda asociados.

En las secciones 5.1 y 6.3 presentamos un método de usar un árbol binario para almacenar un archivo con el objetivo de hacer más eficiente el ordenamiento del archivo. En ese método, todos los descendientes izquierdos de un nodo con llave *key* tenían llave menor que *key* y todos los descendientes derechos tenían llaves mayores o iguales que *key*. El recorrido en orden de tal árbol binario, producía el archivo en orden ascendente de las llaves.

Un árbol así, puede usarse como un árbol de búsqueda binaria. Usando notación de árbol binario, el algoritmo para la búsqueda de la llave *key* en un árbol de ese tipo es como sigue (suponemos que cada nodo contiene cuatro campos: *k*, que

guarda el valor de la llave del registro,  $r$ , que guarda el propio registro y  $left$  y  $right$  que son apuntadores a los subárboles):

```
p = tree;
while (p != null && key != k(p))
 p = (key < k(p)) ? left(p) : right(p);
return(p);
```

La eficiencia del proceso de búsqueda puede perfeccionarse usando un centinela como en la búsqueda secuencial. Un nodo centinela con un apuntador externo por separado apuntándole, permanece asignado con el árbol. Todos los apuntadores al árbol  $left$  o  $right$  que no apunten a otro nodo del árbol apuntan ahora a ese centinela en lugar de ser iguales al apuntador *nulo*. Cuando se ejecuta una búsqueda, se inserta primero la llave argumento en el nodo centinela, garantizando así que será encontrada en el árbol. Esto permite que el encabezamiento del ciclo de búsqueda escriba como  $while (key != k(p))$  sin el riesgo de un ciclo infinito. Tras abandonar el ciclo, si  $p$  es igual al apuntador externo al centinela, la búsqueda fue infructuosa; en caso contrario  $p$  apunta al nodo deseado. Dejamos al lector el algoritmo real.

Obsérvese que la búsqueda binaria de la sección 7.1 en realidad usa un arreglo ordenado como un árbol de búsqueda binaria implícito. El elemento medio del arreglo puede pensarse como la raíz del árbol, la mitad inferior del arreglo (cuyos elementos son menores que el elemento medio) puede considerarse el subárbol izquierdo y la mitad superior (cuyos elementos son mayores que el elemento medio) puede considerarse el subárbol derecho.

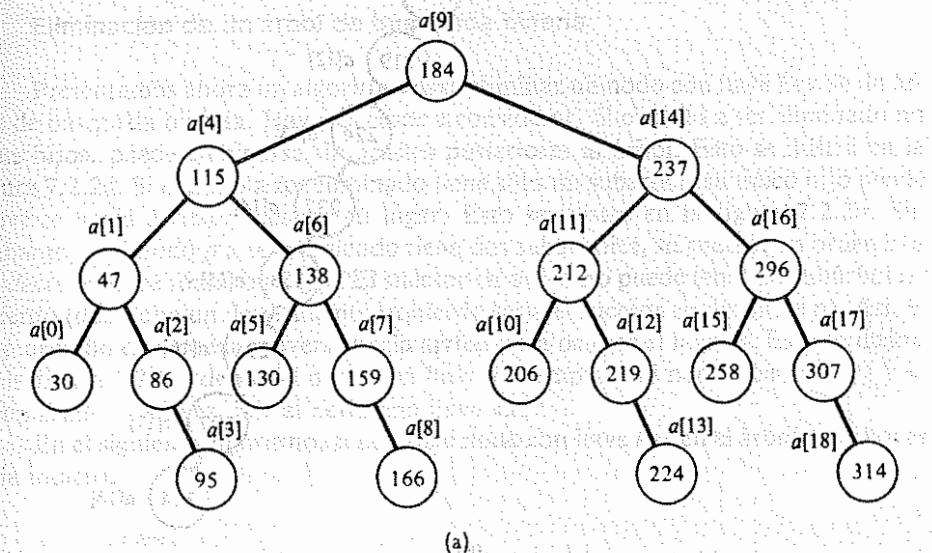
Un arreglo ordenado puede producirse a partir de un árbol de búsqueda binaria recorriendo el árbol en orden e insertando cada elemento en forma secuencial dentro del arreglo al ser visitado. Por otra parte, hay muchos árboles de búsqueda binaria que corresponden a un arreglo ordenado dado. Viendo el elemento medio del arreglo como la raíz y los elementos restantes como subárboles izquierdo y derecho de manera recursiva, se llega a un árbol de búsqueda binaria relativamente balanceado (ver la figura 7.2.1a). Viendo el primer elemento del arreglo como la raíz de un árbol y cada elemento sucesivo como el hijo derecho de su antecesor resulta un árbol binario muy desbalanceado (vea figura 7.2.1b).

La ventaja de usar un árbol de búsqueda binaria sobre el uso de un arreglo es que un árbol permite que las operaciones de inserción, eliminación y búsqueda se ejecuten con eficiencia. Si se usa un arreglo, una inserción o eliminación requieren mover casi la mitad de los elementos del mismo. (¿Por qué?) Por otra parte, la inserción o eliminación en un árbol de búsqueda requieren de sólo unos cuantos ajustes de apuntadores.

#### Inserción en un árbol de búsqueda binaria

El siguiente algoritmo realiza una búsqueda en un árbol de búsqueda binaria e inserta un nuevo registro si la búsqueda resulta infructuosa. (Suponemos la existencia de una función *maketree* que construye un árbol binario consistente en un solo nodo cuyo campo de información se transfiere como argumento y da como resulta-

|     |         |
|-----|---------|
| 30  | $a[0]$  |
| 47  | $a[1]$  |
| 86  | $a[2]$  |
| 95  | $a[3]$  |
| 115 | $a[4]$  |
| 130 | $a[5]$  |
| 138 | $a[6]$  |
| 159 | $a[7]$  |
| 166 | $a[8]$  |
| 184 | $a[9]$  |
| 206 | $a[10]$ |
| 212 | $a[11]$ |
| 219 | $a[12]$ |
| 224 | $a[13]$ |
| 237 | $a[14]$ |
| 258 | $a[15]$ |
| 296 | $a[16]$ |
| 307 | $a[17]$ |
| 314 | $a[18]$ |



(a)

Figura 7.2.1: Un arreglo ordenado y dos de sus representaciones con árboles binarios.

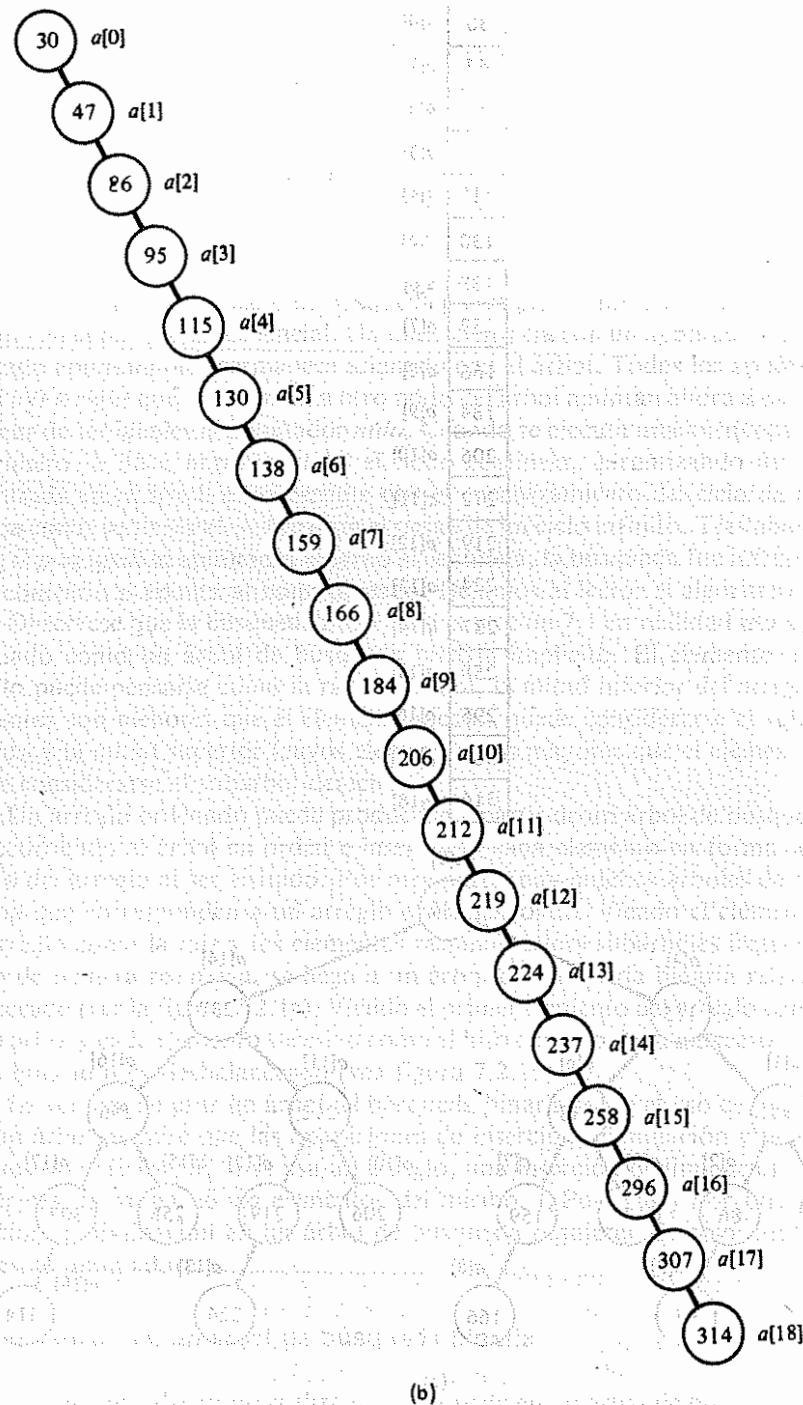


Figura 7.2.1 (Cont.)

do un apuntador al árbol. Esta función se describió en la sección 5.1. Sin embargo, en nuestra versión particular, suponemos que *maketree* acepta dos argumentos, un registro y una llave.)

```

q = null;
p = tree;
while (p != null) {
 if (key == k(p))
 return(p);
 q = p;
 if (key < k(p))
 p = left(p);
 else
 p = right(p);
} /* fin de while */
v = maketree(rec, key);
if (q == null)
 tree = v;
else
 if (key < k(q))
 left(q) = v;
 else
 right(q) = v;
return(v);

```

Obsérvese que después de insertar un nuevo registro, el árbol conserva la propiedad de estar ordenado en un recorrido en orden.

### Eliminación de un árbol de búsqueda binaria

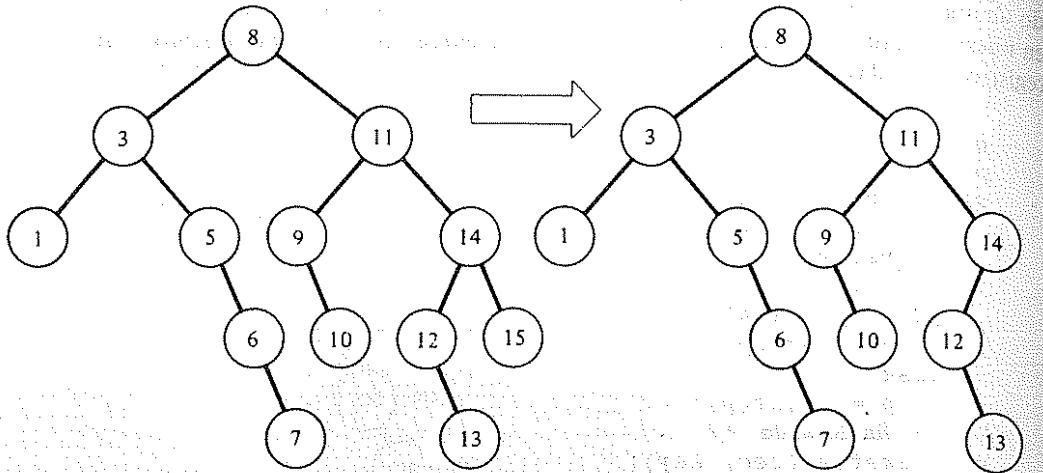
Presentamos ahora un algoritmo para eliminar un nodo con llave *key* de un árbol de búsqueda binaria. Hay tres casos a considerar. Si el nodo a ser eliminado no tiene hijos, puede eliminarse sin ajustes posteriores al árbol. Esto se ilustra en la figura 7.2.2a. Si el nodo a ser eliminado tiene sólo un subárbol, su único hijo puede moverse hacia arriba y ocupar su lugar. Esto se ilustra en la figura 7.2.2b. Sin embargo, si el nodo *p* a ser eliminado tiene dos subárboles, su sucesor en orden *s* (*o predecessor*) debe tomar su lugar. El sucesor en orden no puede tener un subárbol izquierdo (dado que un descendiente izquierdo sería el sucesor en orden de *p*). Así, el hijo derecho de *s* puede moverse hacia arriba para ocupar el lugar *s*. Esto se ilustra en la figura 7.2.2c, donde el nodo con llave 12 remplaza al nodo con llave 11 y es remplazado, a su vez, por el nodo con llave 13.

En el siguiente algoritmo, si no existe nodo con llave *key* en el árbol, el árbol se deja intacto.

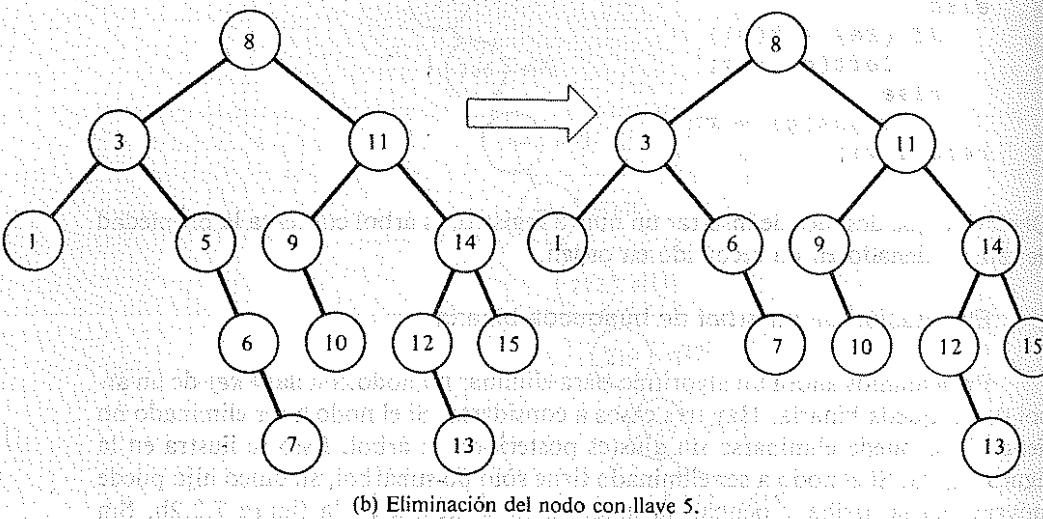
```

p = tree;
q = null;
/* buscar el nodo con la llave key, apuntar dicho nodo
 con p y señalar a su padre con q, si existe
*/

```

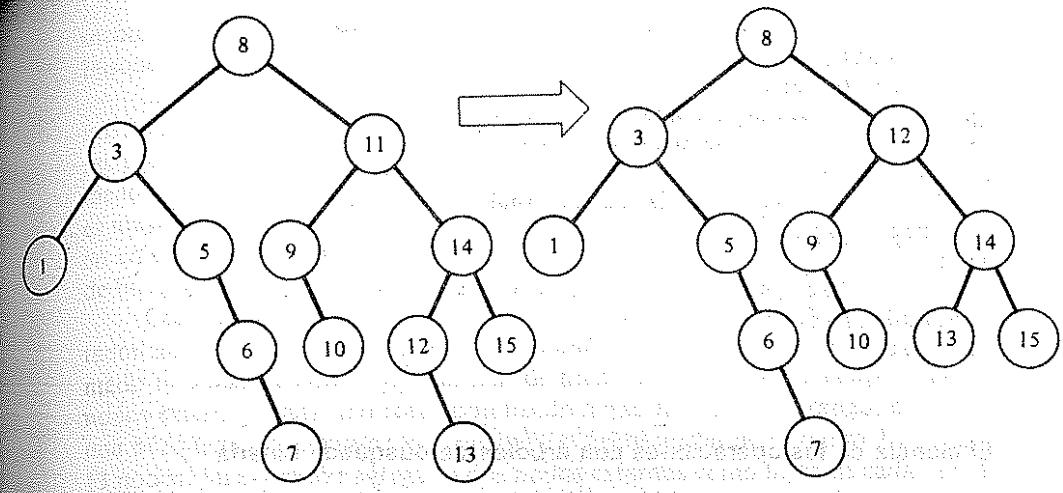


(a) Eliminación del nodo con llave 15.



(b) Eliminación del nodo con llave 5.

Figura 7.2.2 Eliminación de nodos de un árbol de búsqueda binaria.



(c) Eliminación del nodo con llave 11.

Figura 7.2.2 (Cont.)

```

/*
 * asignar a la variable rp el nodo que reemplazará
 * a node (p).
 */
/* Los primeros dos casos: el nodo que se eliminará, a lo más
 * tiene un hijo */
if (left(p) == null)
 rp = right(p);
else {
 if (right(p) == null)
 rp = left(p);
 else {
 /*
 * al sucesor inorder de p y a f el padre de rp
 * f = P;
 * rp = right(p);
 * s = left(rp); /* s es siempre el hijo izquierdo de rp
 * while (s != null) {
 * f = rp;
 * rp = s;
 * s = left(rp);
 * } /* fin de while */
 * en este punto, rp es el sucesor inorder de p
 * if (f != p) {
 * /* p no es el padre de rp y rp == left(f)
 * left(f) = right(rp);
 * /* eliminar node(rp) de su posición actual y
 * /* reemplazarlo con el hijo derecho de node (rp)
 * /* node (rp) toma el lugar de node(p). La razón
 * /* es que si se hace right(rp) = right(p); una vez
 * /* eliminado el nodo p, el sucesor de p es el
 * /* siguiente. Sin embargo, para mantener la
 * /* consistencia, se debe asignar al hijo izquierdo de node(rp) un valor tal
 */
}

```

```

while (p != null && k(p) != key) {
 q = p;
 p = (key < k(p)) ? left(p) : right(p);
} /* fin de while */
if (p == null)
 /* la llave no está en el árbol
 /* dejar el árbol sin modificar
return;

```

```

 /* que node(rp) tome el lugar de node(p) */
 left(rp) = left(p);
} /* fin de if */
/* insertar node(rp) en la posición antes
 ocupada por node(p) */
if (q == null)
 /* node(p) era la raíz del árbol */
 tree = rp;
else
 (p == left(q)) ? left(q) = rp : right(q) = rp;
freenode(p);
return;

```

### Eficiencia de las operaciones con árboles de búsqueda binaria

Como ya vimos en la sección 6.3 (ver figuras 6.3.1 y 6.3.2), el tiempo requerido para buscar en un árbol de búsqueda binaria varía entre  $O(n)$  y  $O(\log n)$ , dependiendo de la estructura del árbol. Si los elementos se insertan en el árbol usando el algoritmo de inserción anterior, la estructura del árbol depende del orden en que se inserten los registros. Si los registros se insertan en orden (u orden inverso), el árbol resultante contiene todas las ligaduras izquierdas (o derechas) nulas, de manera que la búsqueda en el árbol se reduce a una búsqueda secuencial. Sin embargo, si los registros se insertan de manera que la mitad de los que quedan después de un registro dado  $r$  con llave  $k$  tiene llaves menores que  $k$  y la mitad tienen llaves mayores que  $k$ , se alcanza un árbol balanceado en el cual son suficientes de manera aproximada  $\log n$  comparaciones de llave para recuperar un elemento. (De nuevo, debe observarse que el examen de un nodo en nuestro algoritmo de inserción requiere dos comparaciones: una para igualdad y otra para menor que. Sin embargo, en lenguaje de máquina y con algunos compiladores, ambas pueden combinarse en una sola.)

Si los registros se presentan en forma aleatoria (es decir, cualquier permutación de los  $n$ -elementos tiene la misma probabilidad) es más frecuente que resulten árboles balanceados que árboles que no lo son, de manera que, en promedio, el tiempo de búsqueda permanece en  $O(\log n)$ . Para ver esto, definamos la *longitud interna de camino*,  $i$ , de un árbol binario como la suma de los niveles de todos los nodos del árbol (obsérvese que el nivel de un nodo es la longitud del camino de la raíz al nodo). En el árbol inicial de la figura 7.2.2, por ejemplo,  $i$  es igual a 30 (1 nodo en el nivel 0, 2 en el nivel 1, 4 en el 2, 4 en el 3 y 2 en el 4:  $1 * 0 + 2 * 1 + 4 * 2 + 4 * 3 + 2 * 4 = 30$ ). Como el número de comparaciones requeridas para accesar un nodo en un árbol de búsqueda binaria es uno más que el nivel del nodo, el número promedio de comparaciones requeridas para una búsqueda exitosa en un árbol de búsqueda binaria con  $n$  nodos es igual a  $(i + n)/n$ , suponiendo igual probabilidad de acceso para todo nodo del árbol. Así, para el árbol inicial de la figura 7.2.2,  $(30 + 13)/13$ , o en forma aproximada 3.31, será el número de comparaciones requeridas para una búsqueda exitosa. Sea  $s_n$  igual al número promedio de comparaciones requeridas por una búsqueda exitosa en un árbol de búsqueda binaria aleatorio de  $n$  nodos en el cual hay igual probabilidad de que cualquiera de las  $n$  llaves sea el argu-

mento de búsqueda, y sea  $i_n$  la longitud interna de camino promedio de un árbol de búsqueda binaria aleatorio de  $n$  nodos. Entonces  $s_n$  es igual a  $(i_n + n)/n$ .

Sea  $u_n$  el número de comparaciones promedio requerido por una búsqueda infructuosa en un árbol de búsqueda binaria aleatorio de  $n$  nodos. Hay  $n + 1$  maneras posibles en que puede ocurrir una búsqueda infructuosa de la llave  $key$ :  $key$  es menor que  $k(1)$ ,  $key$  está entre  $k(1)$  y  $k(2)$ , ...,  $key$  está entre  $k(n - 1)$  y  $k(n)$  y  $key$  es mayor que  $k(n)$ . Ellas corresponden a los  $n + 1$  apuntadores nulos a subárboles en un árbol de búsqueda binaria de  $n$  nodos. (Se puede mostrar que cualquier árbol de búsqueda binaria con  $n$  nodos tiene  $n + 1$  apuntadores nulos.)

Considérese la *extensión* de un árbol binario formado remplazando cada apuntador izquierdo o derecho nulo por un apuntador a un nuevo nodo hoja aparte, llamado un *nodo externo*. La extensión de un árbol binario de  $n$  nodos tiene  $n + 1$  nodos externos, cada uno correspondiendo a uno de los  $n + 1$  rangos de llave para una búsqueda infructuosa. Por ejemplo, el árbol inicial de la figura 7.2.2 contiene 13 nodos. Su extensión agregaría dos nodos externos como hijos de cada una de las hojas que contienen 1, 7, 10, 13 y 15 y un nodo externo a las que contienen 5, 6, 9 y 12 para un total de 14 nodos externos. Se define la *longitud externa de camino*,  $e$ , de un árbol binario como la suma de los niveles de todos los nodos externos de su extensión. La extensión del árbol inicial de la figura 7.2.2 tiene 4 nodos externos en el nivel 3, 6 en el 4 y 4 en el 5, para una longitud externa de camino de 56. Obsérvese que el nivel de un nodo externo es igual al número de comparaciones llevadas a cabo en una búsqueda infructuosa de la llave en el rango representado por ese nodo externo. Entonces si  $e_n$  es la longitud externa de camino promedio de un árbol de búsqueda binaria aleatorio de  $n$  nodos,  $u_n = e_n/(n + 1)$ . (Con ello se supone que cada uno de los  $n + 1$  rangos de llave tiene igual probabilidad en una búsqueda infructuosa. En la figura 7.2.2 el número promedio de comparaciones para una búsqueda infructuosa es 56/14 o 4.0.) Sin embargo, se puede mostrar que  $e = i + 2n$  para cualquier árbol binario de  $n$  nodos (por ejemplo, en la figura 7.2.2,  $56 = 30 + 2 * 13$ ), de manera que  $e_n = i_n + 2n$ . Como  $s_n = (u_n + n)/n$  y  $u_n = e_n/(n + 1)$ , esto significa que  $s_n = ((n + 1)/n) u_n - 1$ .

El número de comparaciones requeridas para accesar una llave es uno más que el número requerido cuando el nodo fue insertado. Pero el número requerido para insertar una llave es igual al número requerido en una búsqueda infructuosa de esa llave antes de que fuese insertada. Así,  $s_n = 1 + (u_0 + u_1 + \dots + u_{n-1})/n$ . (Es decir, el número promedio de comparaciones para recuperar un elemento en un árbol de  $n$ -nodos es igual al número promedio en accesar cada uno de los primeros elementos hasta el  $n$ -ésimo, y el número para accesar el  $i$ -ésimo es igual a uno más que el número para insertar el  $i$ -ésimo o  $1 + u_{i-1}$ .) Combinando esto con la ecuación  $s_n = ((n + 1)/n) u_n - 1$  se obtiene

$$(n + 1)u_n = 2n + u_0 + u_1 + \dots + u_{n-1}$$

para toda  $n$ . Reemplazando  $n$  por  $n - 1$  tenemos

$$nu_{n-1} = 2(n - 1) + u_0 + u_1 + \dots + u_{n-2}$$

y restando de la ecuación previa tenemos

$$(n+1)u_n - nu_{n-1} = 2 + u_{n-1}$$

$$u_n = u_{n-1} + 2/(n+1)$$

Como  $u_1 = 1$  tenemos que

$$u_n = 1 + 2/3 + 2/4 + \dots + 2/(n+1)$$

y, por lo tanto, como  $s_n = ((n+1)/n)u_n - 1$ , qué

$$\text{entonces } s_n = 2((n+1)/n)(1 + 1/2 + 1/3 + \dots + 1/n) - 3.$$

Cuando  $n$  se hace grande,  $(n+1)/n$  es de manera aproximada 1 y se puede mostrar que  $1 + 1/2 + \dots + 1/n$  es en forma aproximada  $\log(n)$ , donde  $\log(n)$  se define como el logaritmo natural de  $n$  en el archivo de la biblioteca estándar de C *math.h*. Así,  $s_n$  puede ser aproximada (para  $n$  grande) por  $2 * \log(n)$ , que es igual a  $1.386 * \log_2 n$ . Esto significa que el tiempo de búsqueda promedio en un árbol de búsqueda binaria aleatorio es  $O(\log n)$  y requiere sólo 39% más comparaciones, en promedio, que en un árbol binario balanceado.

Como ya observamos, la inserción en un árbol de búsqueda binaria requiere el mismo número de comparaciones que el de una búsqueda infructuosa de la llave. La eliminación requiere el mismo número de comparaciones que la búsqueda de la llave a ser eliminada, aunque involucra trabajo adicional para encontrar el sucesor o antecesor en orden. Puede mostrarse que el algoritmo de eliminación que presentamos en realidad modifica el costo de búsqueda promedio subsecuente en el árbol. Es decir, un árbol aleatorio de  $n$ -llaves creado insertando  $n+1$  llaves y eliminando después una llave aleatoria tiene una longitud interna de camino inferior (y por lo tanto menor costo de búsqueda promedio) a la de un árbol aleatorio de  $n$ -llaves creado insertando  $n$  llaves. El proceso de eliminación de un nodo con un hijo remplazando al primero por el último, sin importar si el hijo es izquierdo o derecho, produce un árbol mejor que el promedio; un algoritmo de eliminación similar que sólo reemplace un nodo con un hijo por su hijo si este último es un hijo izquierdo (es decir, si su sucesor no está contenido en su subárbol) y en caso contrario remplace el nodo por su sucesor de en orden produce un árbol aleatorio, suponiendo que no se llevan a cabo inserciones adicionales. Este último algoritmo se llama *algoritmo de eliminación asimétrica*.

Sin embargo, de manera bastante extraña, cuando se hacen inserciones y eliminaciones adicionales usando el algoritmo de eliminación asimétrica, la longitud interna de camino y el tiempo de búsqueda disminuyen al inicio pero empiezan luego a elevarse en forma rápida otra vez. Para árboles que contengan más de 128 llaves, la longitud interna de camino se vuelve a fin de cuentas peor que para un árbol aleatorio, y para árboles con más de 2048 llaves, la longitud interna de camino se vuelve al final 50 por ciento peor que para un árbol aleatorio.

Un *algoritmo de eliminación simétrica* alternativo, que alterne la eliminación del predecesor y sucesor en orden (pero que siga remplazando un nodo con un hijo

por su hijo sólo cuando el predecesor o sucesor en orden respectivamente no estén contenidos en su subárbol), produce al final mejores árboles que los aleatorios después de mezclar inserciones y eliminaciones adicionales. Datos empíricos indican que se reduce la longitud de camino después de muchas inserciones y eliminaciones simétricas alternas en 88 por ciento de su valor aleatorio correspondiente en forma aproximada.

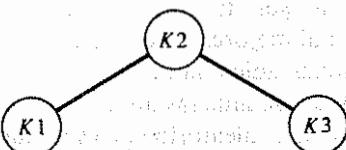
### Eficiencia de árboles de búsqueda binaria no uniforme

Todo lo anterior supone que hay igual probabilidad para cualquier llave en la tabla como argumento de búsqueda. Sin embargo, en la práctica real es común que se recuperen algunos registros con mayor frecuencia, otros con frecuencia moderada y otros casi nunca. Supóngase que los registros se insertan en el árbol de manera que los accesados con mayor frecuencia preceden a los accesados con menor frecuencia. Entonces los registros recuperados con mayor frecuencia estarán más cerca de la raíz del árbol, de manera que el tiempo promedio de búsqueda exitosa se reduce. (Por supuesto, esto supone que el reordenamiento de las llaves por frecuencia reducida de accesos no produce un árbol binario muy desbalanceado, dado que si lo hace, la ventaja por la reducción en el número de comparaciones para los registros accesados con mayor frecuencia podría no compensar la desventaja de incrementarse el número de comparaciones para la vasta mayoría de los registros.)

Si los elementos a ser recuperados forman un conjunto constante, sin inserciones o eliminaciones, puede valer la pena definir un árbol de búsqueda binaria que haga más eficientes las búsquedas subsecuentes. Por ejemplo, considerar los árboles de búsqueda binaria de la figura 7.2.3. Ambos árboles: el de la figura 7.2.3a y el de la 7.2.3b contienen tres elementos  $k_1, k_2$  y  $k_3$ , donde  $k_1 < k_2 < k_3$ , y son árboles de búsqueda binaria válidos para ese conjunto. Sin embargo, la recuperación de  $k_3$  requiere dos comparaciones en la figura 7.2.3a pero sólo una en la figura 7.2.3b. Por supuesto, existen aún otros árboles de búsqueda binaria válidos para este conjunto de llaves.

El número de comparaciones de llaves necesario para recuperar un registro es igual al nivel de ese registro en el árbol de búsqueda binaria más 1. Así, la recuperación de  $k_2$  requiere una comparación en el árbol de la figura 7.2.3a pero requiere tres comparaciones en el árbol de la figura 7.2.3b. Una búsqueda binaria infructuosa de un argumento que esté de manera inmediata entre dos llaves  $a$  y  $b$  requiere tantas comparaciones de llaves como el número máximo de comparaciones requeridas por una búsqueda exitosa para  $a$  o  $b$ . (¿Por qué?) Esto es igual a 1 más el máximo de los niveles de  $a$  o  $b$ . Por ejemplo, una búsqueda de la llave que está entre  $k_2$  y  $k_3$  requiere dos comparaciones de llaves en la figura 7.2.3a y tres comparaciones en la figura 7.2.3b, mientras que una búsqueda de una llave mayor de  $k_3$  requiere dos comparaciones en la figura 7.2.3a pero sólo una en la figura 7.2.3b.

Suponer que  $p_1, p_2$  y  $p_3$  son las probabilidades de que el argumento de búsqueda sea igual a  $k_1, k_2$  y  $k_3$  en forma respectiva. Suponer también que  $q_0$  es la probabilidad de que el argumento de búsqueda sea menor que  $k_1$ ,  $q_1$  es la probabilidad de que esté entre  $k_1$  y  $k_2$ ,  $q_2$  de que esté entre  $k_2$  y  $k_3$  y  $q_3$  de que sea mayor que  $k_3$ . Entonces  $p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3 = 1$ . El *número esperado* de comparaciones en una búsqueda es la suma de las probabilidades de que el argumento



(a) Número esperado de comparaciones.

$$2p_1 + p_2 + 2p_3 + 2q_0 + 2q_1 + 2q_2 + 2q_3$$



(b) Número esperado de comparaciones.

$$2p_1 + 3p_2 + p_3 + 2q_0 + 3q_1 + 3q_2 + q_3$$

Figura 7.2.3 Dos árboles de búsqueda binaria.

tenga un valor dado por el número de comparaciones requeridas para recuperar ese valor, donde la suma se toma sobre todos los valores que sean posibles argumentos de búsqueda. Por ejemplo, el número esperado de comparaciones en una búsqueda en el árbol de la figura 7.2.3a es

$$2p_1 + p_2 + 2p_3 + 2q_0 + 2q_1 + 2q_2 + 2q_3$$

y el número esperado de comparaciones en una búsqueda en el árbol de la figura 7.2.3b es

$$2p_1 + 3p_2 + p_3 + 2q_0 + 3q_1 + 3q_2 + q_3$$

Este número esperado de comparaciones puede usarse como una medida de cuán “bueno” es un árbol de búsqueda binaria en particular para un conjunto dado de llaves y un conjunto dado de probabilidades. Así, para las siguientes probabilidades a la izquierda, el árbol de la figura 7.2.3a es más eficiente; para las probabilidades de la derecha, es más eficiente el árbol de la figura 7.2.3b.

Probabilidad:  $p_1 = .1$  para clave  $k_1$ ,  $p_2 = .1$  para clave  $k_2$ ,  $p_3 = .1$  para clave  $k_3$ . Probabilidad:  $p_1 = .1$  para clave  $k_1$ ,  $p_2 = .3$  para clave  $k_2$ ,  $p_3 = .3$  para clave  $k_3$ .

$$\begin{aligned} q_0 &= .1 \\ q_1 &= .2 \\ q_2 &= .1 \\ q_3 &= .1 \end{aligned}$$

$$\begin{aligned} q_0 &= .1 \\ q_1 &= .1 \\ q_2 &= .1 \\ q_3 &= .2 \end{aligned}$$

Número esperado para 7.2.3a = 1.7

Número esperado para 7.2.3b = 2.4

Número esperado para 7.2.3a = 1.9

Número esperado para 7.2.3b = 1.8

### Arboles de búsqueda óptimos

Un árbol de búsqueda binaria que minimice el número esperado de comparaciones para un conjunto dado de llaves y probabilidades se llama **óptimo**. El más rápido algoritmo conocido para producir un árbol de búsqueda binaria óptimo es  $O(n^2)$  en el caso general. Esto es muy costoso a menos que el árbol se mantenga sin cambiar a lo largo de un número muy grande de búsquedas. En casos en los cuales todas las  $p(i)$  sean iguales a 0 (es decir, las llaves actúan sólo para definir valores de rangos con los que están asociados los datos, de manera que todas las búsquedas sean “infructuosas”), existe un algoritmo  $O(n)$  para crear un árbol de búsqueda binaria óptimo.

Sin embargo, aunque no existe un algoritmo eficiente para construir un árbol óptimo en el caso general, hay varios métodos de construcción de árboles cercanos a óptimos en tiempo  $O(n)$ . Suponer  $n$  llaves, de  $k(1)$  a  $k(n)$ . Sea  $p(i)$  la probabilidad de buscar la llave  $k(i)$ , y  $q(i)$  la probabilidad de una búsqueda infructuosa entre  $k(i-1)$  y  $k(i)$  (con  $q(0)$  la probabilidad de una búsqueda infructuosa de una llave menor que  $k(1)$ , y  $q(n)$  la probabilidad de una búsqueda infructuosa de una llave posterior a  $k(n)$ ). Definir  $s(i, j)$  como  $q(i) + p(i+1) + \dots + q(j)$ .

Un método llamado el **método de balanceo**, intenta encontrar un valor  $i$  que minimice el valor absoluto de  $s(0, i-1) - s(i, n)$  y establezca a  $k(i)$  como la raíz del árbol de búsqueda binaria, de  $k(1)$  a  $k(i-1)$  en su subárbol izquierdo y de  $k(i+1)$  a  $k(n)$  en su subárbol derecho. El proceso se aplica entonces de manera recursiva para construir los subárboles izquierdo y derecho.

La localización del valor de  $i$  para el cual  $abs(s(0, i-1) - s(i, n))$  se hace mínimo, puede realizarse de manera eficiente como sigue. Primero, definir un arreglo  $s0[n+1]$  de manera que  $s0[i]$  sea igual a  $s(0, i)$ . Esto puede hacerse inicializando  $s0[0]$  como  $q(0)$  y  $s0[j]$  como  $s0[j-1] + p(j) + q(j)$  para  $j$  de 1 a  $n$ . Una vez inicializado  $s0$ , se puede calcular  $s(i, j)$  para toda  $i$  y  $j$  como  $s0[j] - s0[i-1] - p(i)$  siempre que sea necesario. Definimos  $si(j)$  como  $s(0, j-1) - s(j, n)$ . Queremos minimizar  $abs(si(i))$ .

Tras inicializar  $s0$ , comenzamos el proceso de hallar un  $i$  para minimizar  $abs(s(0, i-1) - s(i, n))$ , o  $si(i)$ . Obsérvese que  $si$  es una función monótona creciente. Obsérvese también que  $si(0) = q(0) - 1$ , que es negativo, y  $si(n) = 1 - q(n+1)$  que es positivo. Revisar los valores de  $si(1), si(n), si(2), si(n-1), si(4), si(n-3), \dots, si(2^j), si(n+1-2^j)$  por turno hasta descubrir el primer  $si(2^j)$  positivo o el primer  $si(n+1-2^j)$  negativo. Si se encuentra primero un  $si(2^j)$  positivo, el  $i$  deseado que minimiza  $abs(si(i))$  está dentro del intervalo  $[2^{j-1}, 2^j]$ ; si se encuentra

primero un  $si(n + 1 - 2^j)$  negativo, el  $i$  deseado está en el intervalo  $[n + 1 - 2^j, n + 1 - 2^{j-1}]$ . En cualquier caso,  $i$  ha sido limitado a un intervalo de tamaño  $2^{j-1}$ . Dentro del intervalo, usar una búsqueda binaria para limitar a  $i$ . El efecto de duplicar el tamaño del intervalo garantiza que el proceso recursivo entero sea  $O(n)$ , mientras que si para comenzar se usara una búsqueda binaria en todo el intervalo  $[0, n]$ , el proceso sería  $O(n \log n)$ .

Un segundo método usado para construir árboles de búsqueda binaria cercanos al óptimo se llama el *método exhaustivo*. En lugar de construir el árbol del tope hacia abajo, como en el método de balanceo, el método exhaustivo construye el árbol de abajo hacia arriba. El método usa una lista lineal doblemente ligada en la cual cada elemento contiene cuatro apuntadores, un valor de llave y tres valores de probabilidad. Los cuatro apuntadores son apuntadores izquierdo y derecho a la lista usados para organizar la lista doblemente ligada y apuntadores a los subárboles izquierdo y derecho para tomar en cuenta los subárboles de búsqueda binaria que contienen llaves menores que, y mayores que el valor de llave en el nodo. Los tres valores de probabilidad son la suma de las probabilidades en el subárbol izquierdo, llamada la *probabilidad izquierda*, la probabilidad  $p(i)$  del valor de la llave del nodo  $k(i)$ , llamada la *probabilidad de llave* y la suma de las probabilidades en el subárbol derecho llamada *probabilidad derecha*. La *probabilidad total* de un nodo se define como la suma de sus probabilidades izquierda, derecha y de llave. Al inicio, hay  $n$  nodos en la lista. El valor de la llave en el nodo  $i$ -ésimo es  $k(i)$ , su probabilidad izquierda es  $q(i-1)$  su probabilidad derecha es  $q(i)$ , su probabilidad de llave es  $p(i)$ , y sus apuntadores a los subárboles izquierdo y derecho son *nulos*.

Cada iteración del algoritmo encuentra el primer nodo  $nd$  en la lista cuya probabilidad total es menor o igual que la de su sucesor (si no hay nodo que lo cumpla,  $nd$  se hace igual al último nodo de la lista). La llave en  $nd$  se convierte en la raíz de un subárbol de búsqueda binaria cuyos subárboles derecho e izquierdo son los subárboles derecho e izquierdo de  $nd$ .  $nd$  se elimina entonces de la lista. El apuntador al subárbol izquierdo de su sucesor (si lo hay) y el apuntador al subárbol derecho de su antecesor (si lo hay) se hacen apuntar al nuevo subárbol, la probabilidad izquierda de su sucesor y la probabilidad derecha de su antecesor se hacen iguales a la probabilidad total de  $nd$ . Este proceso se repite hasta que sólo quede un nodo en la lista. (Obsérvese que no es necesario comenzar con un completo recorrido de la lista desde el principio de la misma en cada iteración; sólo es necesario empezar a partir del segundo antecesor del nodo eliminado en la iteración previa.) Cuando sólo queda un nodo en la lista, su llave se coloca en la raíz del árbol de búsqueda binaria final, con los apuntadores derecho e izquierdo del nodo como apuntadores a los subárboles derecho e izquierdo de la raíz.

Otra técnica para reducir el tiempo promedio de búsqueda, cuando se conocen las probabilidades, es un *árbol de división*. Un árbol así contiene dos llaves en lugar de una en cada nodo. La primera, llamada *llave de nodo*, se verifica si es igual a la llave argumento. Si son iguales, la búsqueda culmina con éxito; si no, la llave argumento se compara con la segunda llave en el nodo, llamada la *llave de división*, para determinar si la búsqueda debe continuar en el subárbol izquierdo o derecho. Un tipo particular de árbol de división, llamado *árbol de división por medianas*, coloca como llave de nodo a la más frecuente de las llaves en el subárbol con raíz en ese nodo y coloca la llave de división igual a la mediana de las llaves en ese subárbol (es

dicho, la llave  $k$  tal que el mismo número de llaves en el subárbol sean menores que, y mayores que  $k$ ). Esto tiene la doble ventaja de garantizar un árbol balanceado y asegurar que las llaves más frecuentes se encuentren cerca de la raíz. Aunque los árboles de división por medianas requieren una llave extra en cada nodo, pueden construirse como árboles binarios casi completos implantados dentro de un arreglo, ahorrando el espacio de los apuntadores de árbol. Un árbol de división por medianas de un conjunto dado de llaves y frecuencias puede construirse en tiempo  $O(n \log n)$ , y una búsqueda en un árbol de ese tipo siempre requiere menos de  $\log_2 n$  visitas a nodos, aunque cada visita requiere dos comparaciones por separado.

### Árboles balanceados

Como observamos con anterioridad, si la probabilidad de buscar una clave en una tabla es la misma para todas las llaves, la búsqueda más eficiente se efectúa en un árbol binario balanceado. Desafortunadamente, el algoritmo de búsqueda e inserción presentado previamente no asegura que el árbol permanezca balanceado; el grado de balanceo depende del orden en que son insertadas las llaves en el árbol. Nos gustaría tener un algoritmo de búsqueda e inserción eficiente que mantenga el árbol de búsqueda como un árbol binario balanceado.

Definamos primero de manera más precisa la noción de un árbol "balanceado". La *altura* de un árbol binario es el nivel máximo de sus hojas (también se conoce a veces como la *profundidad* del árbol). Por conveniencia, la altura del árbol nulo se define como  $-1$ . Un *árbol binario balanceado* (a veces llamado *árbol AVL*) es un árbol binario en el cual las alturas de los dos subárboles de todo nodo difieren a lo sumo en 1. El *balance* de un nodo en un árbol binario se define como la altura de su subárbol izquierdo menos la altura de su subárbol derecho. La figura 7.2.4a ilustra un árbol binario balanceado. Cada nodo en un árbol binario balanceado tiene balance igual a 1,  $-1$  o 0, dependiendo de si la altura de su subárbol izquierdo es mayor que, menor que o igual a la altura de su subárbol derecho. En la figura 7.2.4a se muestra el balance de cada nodo.

Supóngase que tenemos un árbol binario balanceado y usamos el algoritmo de búsqueda e inserción precedente para insertar un nodo  $p$  en dicho árbol. Entonces el árbol resultante puede o no permanecer balanceado. La figura 7.2.4b ilustra todas las posibles inserciones que pueden hacerse en el árbol de la figura 7.2.4a. Cada inserción que produce un árbol binario balanceado se indica con una  $B$ . Las inserciones desbalanceadas se indican con una  $U$  y están enumeradas del 1 al 12. Es fácil ver que el árbol se vuelve desbalanceado si y sólo si el nodo recién insertado es un descendiente izquierdo de un nodo que tenía de manera previa un balance de 1 (esto ocurre en los casos  $U1$  hasta  $U8$  de la figura 7.2.4b) o si es un hijo derecho descendiente de un nodo que tenía de manera previa balance  $-1$  (casos del  $U9$  al  $U12$ ).

En la figura 7.2.4b, el ancestro más joven que se vuelve desbalanceado en cada inserción se indica mediante los números contenidos en tres de los nodos.

Examinemos algo más el subárbol con raíz en el ancestro más joven que se volverá desbalanceado como resultado de una inserción. Ilustramos el caso donde el balance de este subárbol fue de manera previa 1, dejando el otro caso como ejercicio al lector. La figura 7.2.5 ilustra este caso. Denominemos al nodo desbalanceado  $A$ .

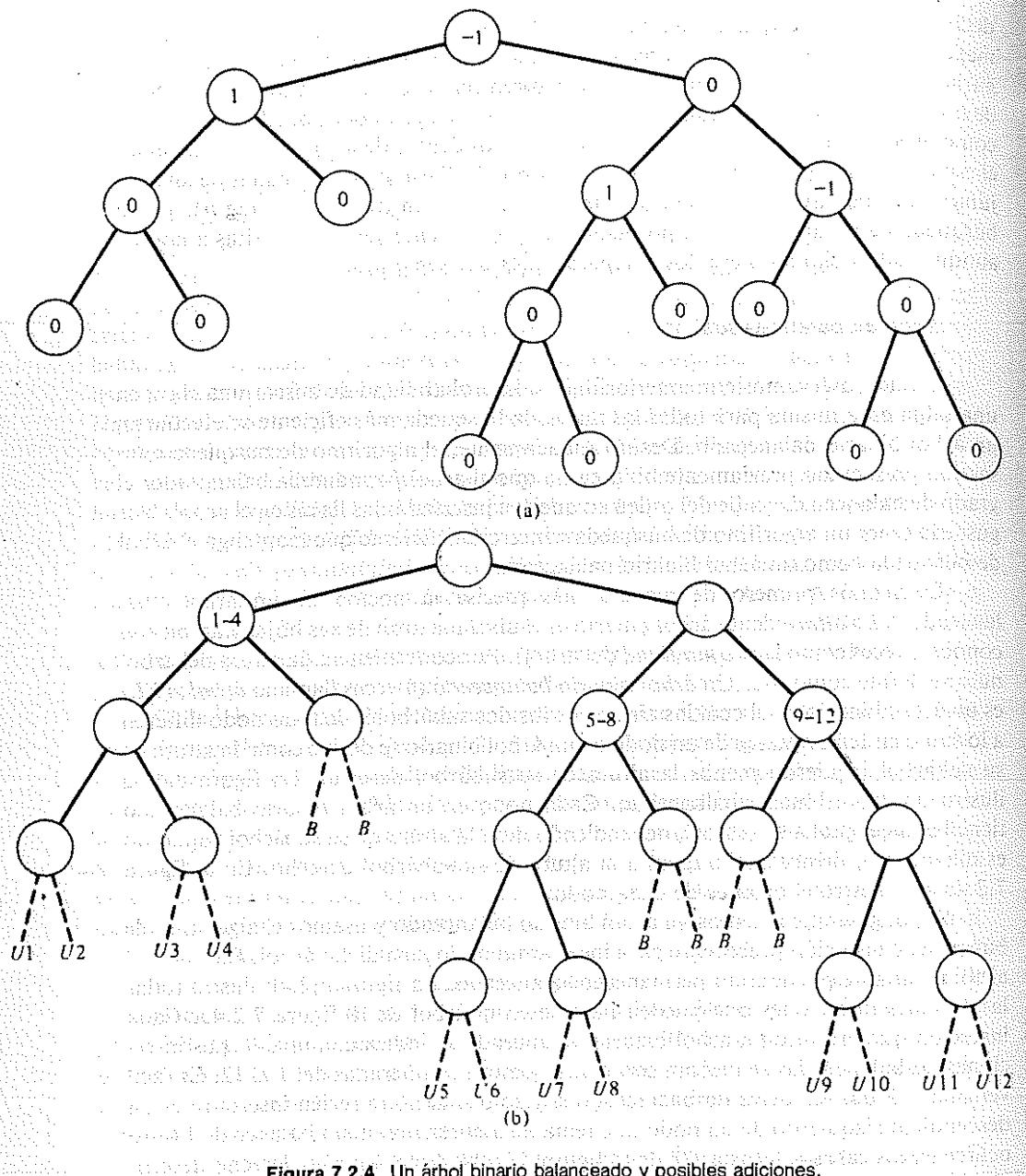


Figura 7.2.4 Un árbol binario balanceado y posibles adiciones.

Como  $A$  tenía un balance de 1, su subárbol izquierdo no era nulo; podemos en consecuencia designar a su hijo izquierdo como  $B$ . Dado que  $A$  es el ancestro más joven del nuevo nodo que se volverá desbalanceado, el nodo  $B$  debe haber tenido un balance de 0. (Se pide al lector probar la afirmación anterior como ejercicio). Así, el nodo  $B$  debe haber tenido (antes de la inserción) subárboles izquierdo y derecho

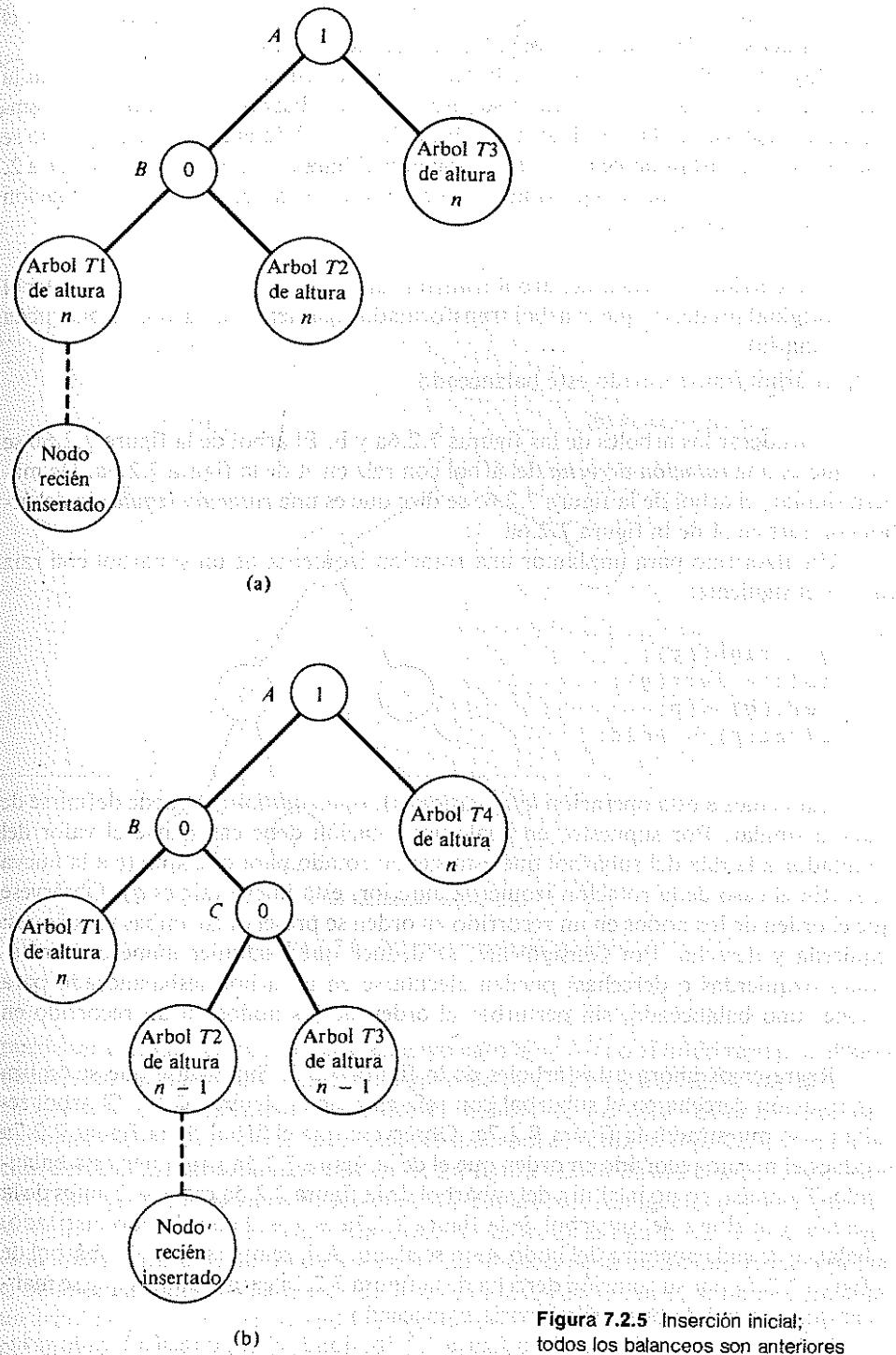


Figura 7.2.5 Inserción inicial; todos los balanceos son anteriores a la inserción.

de igual altura  $n$  donde es posible que  $n = -1$ .) Dado que el balance de  $A$  fue 1, el subárbol derecho de  $A$  debe también haber sido de altura  $n$ .

Hay ahora dos casos por considerar, ilustrados en la figura 7.2.5a y b. En la figura 7.2.5a el nodo recién creado se inserta en el subárbol izquierdo de  $B$ , cambiando el balance de  $B$  a 1 y el de  $A$  a 2. En la figura 7.2.5b el nodo recién creado se inserta en el subárbol derecho de  $B$ , cambiando el balance de  $B$  a  $-1$  y el de  $A$  a 2. Para que el árbol se mantenga balanceado es necesario realizar una transformación en el mismo de manera que

1. el recorrido en orden del árbol transformado sea el mismo que para el árbol original (es decir, que el árbol transformado siga siendo un árbol de búsqueda binaria)
2. el árbol transformado esté balanceado.

Considerar los árboles de las figuras 7.2.6a y b. El árbol de la figura 7.2.6b se dice que es una *rotación derecha* del árbol con raíz en  $A$  de la figura 7.2.6a. De manera similar, el árbol de la figura 7.2.6c se dice que es una *rotación izquierda* del árbol con raíz en  $A$  de la figura 7.2.6a.

Un algoritmo para implantar una rotación izquierda de un subárbol con raíz en  $p$  es el siguiente:

```
q = right(p);
hold = left(q);
left(q) = p;
right(p) = hold;
```

Llamemos a esta operación *leftrotation*( $p$ ). *rightrotation*( $p$ ) puede definirse de manera similar. Por supuesto, en cualquier rotación debe cambiarse el valor del apuntador a la raíz del subárbol que está siendo rotado para que apunte a la nueva raíz. (En el caso de la rotación izquierda anterior, esta nueva raíz es  $q$ .) Obsérvese que el orden de los nodos en un recorrido en orden se preserva en ambas rotaciones: izquierda y derecha. Por consiguiente, se deduce que cualquier número de rotaciones (izquierdas o derechas) pueden ejecutarse en un árbol desbalanceado para obtener uno balanceado, sin perturbar el orden de los nodos en un recorrido en orden.

Regresemos ahora a los árboles de la figura 7.2.5. Supóngase que se realiza una rotación derecha en el subárbol con raíz en  $a$  de la figura 7.2.5a. El árbol resultante se muestra en la figura 7.2.7a. Obsérvese que el árbol de la figura 7.2.7a produce el mismo recorrido en orden que el de la figura 7.2.5a y también está balanceado. También, como la altura del subárbol de la figura 7.2.5a era  $n + 2$  antes de la inserción y la altura del subárbol de la figura 7.2.7a es  $n + 2$  con el nodo insertado, el balance de cada ancestro del nodo  $A$  no se altera. Así, remplazando el subárbol de la figura 7.2.5a por su rotación derecha de la figura 7.2.7a garantizamos que se mantenga como árbol de búsqueda binaria balanceado.

Volvamos ahora al árbol de la figura 7.2.5b, donde el nodo recién creado se inserta en el subárbol derecho de  $B$ . Sea  $C$  el hijo derecho de  $B$ . (Hay tres casos:  $C$

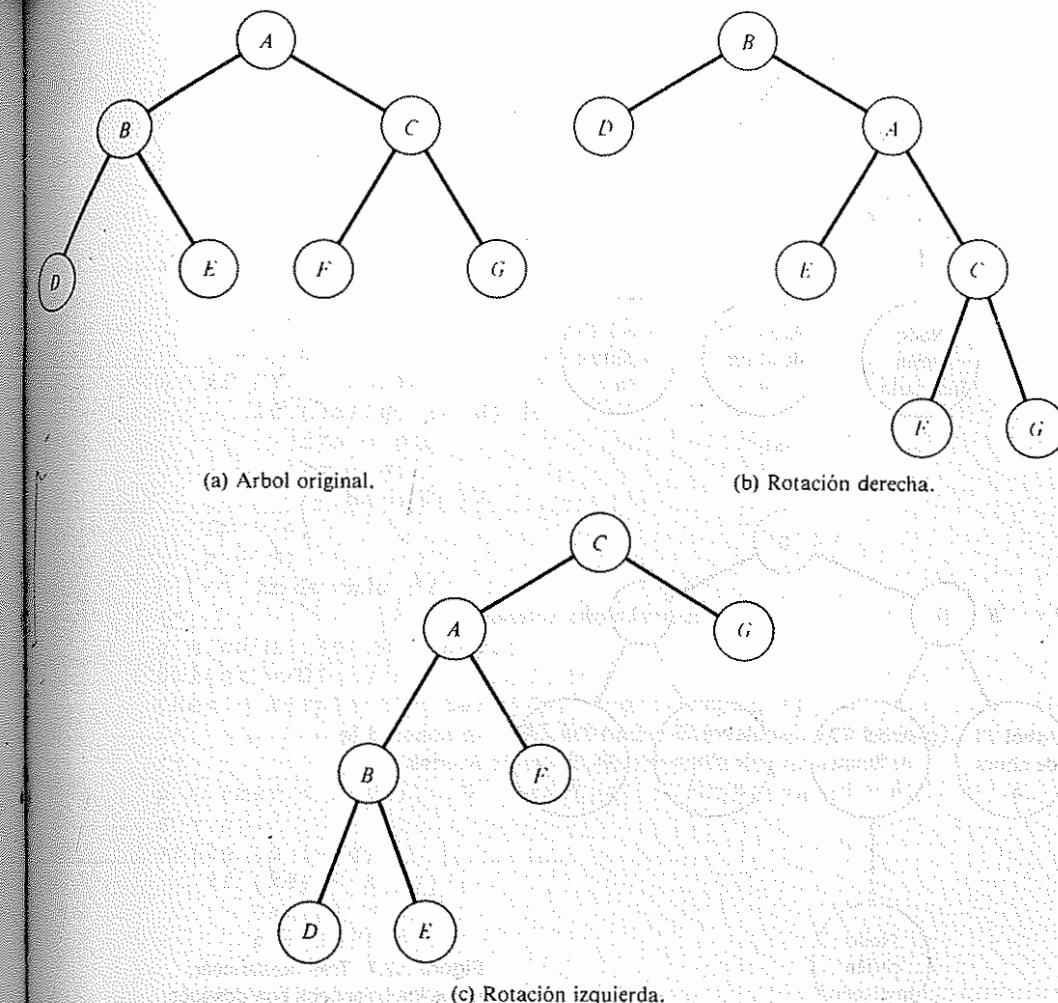
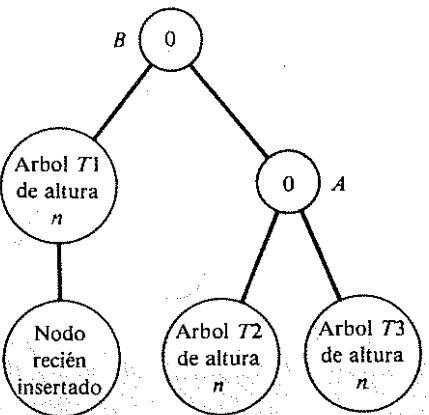
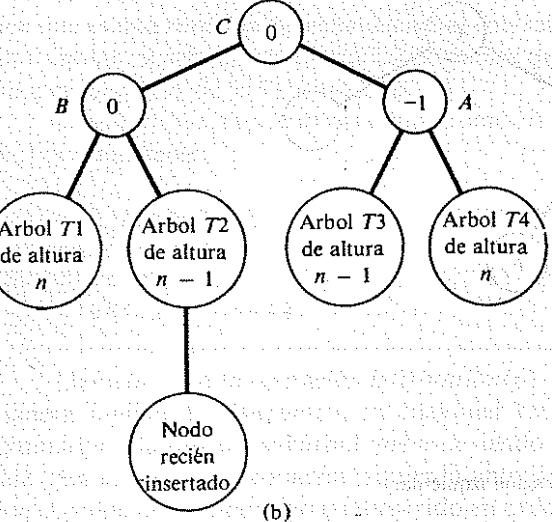


Figura 7.2.6 Rotación simple en un árbol.

puede ser el nodo recién insertado en cuyo caso  $n = -1$ , o el nodo recién insertado puede estar en el subárbol derecho o izquierdo de  $C$ . La figura 7.2.5b ilustra el caso en que está en el subárbol izquierdo; el análisis de los otros casos es análogo.) Supóngase que una rotación izquierda del subárbol con raíz en  $B$  precede a una rotación derecha del subárbol con raíz en  $A$ . La figura 7.2.7b ilustra el árbol resultante. Verificar que los recorridos en orden de los dos árboles son iguales y que el árbol de la figura 7.2.7b esté balanceado. La altura del árbol de la figura 7.2.7b es  $n + 2$ , que es la misma altura del árbol de la figura 7.2.5b antes de la inserción, de manera que el balance en todos los ancestros de  $A$  no cambia. Por lo tanto, seguimos teniendo un árbol de búsqueda balanceado remplazando el árbol de la figura 7.2.5b por el de la figura 7.2.7b, siempre que esto ocurra después de la inserción.



(a)



(b)

Figura 7.2.7 Tras rebalancear, todos los balanceos son después de la inserción.

Presentemos ahora un algoritmo para buscar e insertar en un árbol binario balanceado que no esté vacío. Cada nodo del árbol contiene cinco campos:  $k$  y  $r$ , que guardan la llave y el registro de manera respectiva,  $left$  y  $right$  que son apuntadores a los subárboles izquierdo y derecho de manera respectiva y  $bal$ , cuyo valor es 1, -1 o 0 dependiendo del balance del nodo. En la primera parte del algoritmo, si la llave deseada aún no se encuentra en el árbol, se inserta un nuevo nodo en el árbol de búsqueda binario, sin importar el balance. La primera fase también toma en cuenta al ancestro más joven, ya que puede desbalancearse tras la inserción. El algoritmo hace uso de la función *maketree* descrita con anterioridad y de las rutinas *rightrotation* y *leftrotation*, que aceptan un apuntador a la raíz de un subárbol y ejecutan la rotación deseada.

#### /\* PARTE I:

```

fp = null;
p = tree;
fya = null;
ya = p;
/*
ya apunta al ancestro más joven que puede
llegar a desbalancearse. fya señala al padre
de ya, y fp al padre de p
*/
while (p != null) {
 if (key == k(p)) return(p);
 q = (key < k(p)) ? left(p) : right(p);
 if (q != null) {
 if (bal(q) != 0) {
 fya = p;
 ya = q;
 }
 }
 fp = p;
 p = q;
}
/* fin de while */

/* insertar nuevo registro
q = maketree(rec, key);
bal(q) = 0;
(key < k(fp)) ? left(fp) = q : right(fp) = q;
el balance de todos los nodos entre node(ya) y node(q)
deberá alterarse, modificando su valor de 0
*/
p = (key < k(ya)) ? left(ya) : right(ya);
s = p;
while (p != q) {
 if (key < k(p)) {
 bal(p) = 1;
 p = left(p);
 } else {
 bal(p) = -1;
 p = right(p);
 }
}
/* fin de while */
/* PARTE II: Determinar si el árbol se encuentra
desbalanceado o no. Si lo está, q es el nodo
recién insertado, ya es su ancestro desbalanceado más joven,
fya es el parente de ya y s es el hijo de ya en la
dirección del desbalance
imbal = (key < k(ya)) ? 1 : -1;
if (bal(ya) == 0) {
 /*
 Se le ha agregado otro nivel al árbol
 El árbol permanece balanceado
 bal(ya) = imbal;
 return(q);
}
/* fin de if */

```

```

if (bal(ya) != imbal) {
 /* el nodo agregado se ha colocado en la */
 /* dirección opuesta del desbalance. */
 /* El árbol permanece balanceado */
 bal(ya) = 0;
 return(q);
} /* fin de if */
/* PARTE III: El nodo adicional a desbalanceado al árbol.
 Restablecer el balance efectuando la rotación requerida,
 ajustando después los valores de balance de los nodos involucrados
if (bal(s) == imbal) {
 /* ya y s se han desbalanceado en la misma dirección;
 observar figura 7.2.5a donde ya = a y s = b */
 p = s;
 if (imbal == 1)
 rightrotation(ya);
 else
 leftrotation(ya);
 bal(ya) = 0;
 bal(s) = 0;
}
else {
 /* ya y s se encuentran desbalanceados en direcciones
 opuestas; ver figura 7.2.5b */
 if (imbal == -1) {
 p = right(s);
 leftrotation(s);
 left(ya) = p;
 rightrotation(ya);
 }
 else {
 p = left(s);
 right(ya) = p;
 rightrotation(s);
 leftrotation(ya);
 } /* fin de if */
 /* ajustar el campo bal para los nodos involucrados */
 if (bal(p) == 0) {
 /* p fue un nodo insertado */
 bal(ya) = 0;
 bal(s) = 0;
 }
 else
 if (bal(p) == imbal) {
 /* ver las figuras 7.2.5b y 7.2.7b */
 bal(ya) = -imbal;
 bal(s) = 0;
 }
}
}

```

```

else {
 /* sólo que el nuevo nodo se insertó en f3 */
 bal(ya) = 0;
 bal(s) = imbal;
} /* fin de if */
bal(p) = 0;
} /* fin de if */
/* ajustar el apuntador del subárbol rotado */
if (fya == null)
 tree = p;
else
 (ya == right(fya)) ? right(fya) = p : left(fya) = p;
return(q);
}

```

La altura máxima de un árbol de búsqueda binaria balanceado es  $1.44 \log_2 n$ , de manera que una búsqueda en un árbol así nunca requiere más de 44 por ciento de comparaciones que las necesarias en un árbol balanceado de manera completa. En la práctica, los árboles de búsqueda binaria balanceados se comportan aún mejor, produciendo tiempos de búsqueda del orden de  $\log_2 n + 0.25$  para valores grandes de  $n$ . En promedio, se requiere una rotación en el 46.5 por ciento de las inserciones.

El algoritmo para eliminar un nodo de un árbol de búsqueda binaria balanceado conservando su balance, es aún más complejo. Mientras que la inserción requiere a lo sumo una rotación doble, la eliminación puede requerir una rotación (doble o simple) en cada nivel del árbol, o  $O(\log n)$  rotaciones. Sin embargo, en la práctica, se ha visto que sólo se requiere un promedio de 0.214 rotaciones (simples o dobles) por eliminación.

Los árboles de búsqueda binaria balanceados que hemos visto se llaman *árboles de altura balanceada* porque su altura se usa como criterio para el balanceo. Hay un gran número de formas diferentes para definir árboles balanceados. En un método, se define el *peso* del árbol como el número de nodos externos en el mismo (que es igual al número de apuntadores nulos). Si el cociente del peso del subárbol izquierdo de todo nodo entre el peso del subárbol con raíz en el nodo está entre alguna fracción  $a$  y  $1 - a$ , el árbol es un *árbol de pesos balanceados de razón a* o se dice que está en la clase *wb[a]*. Cuando una inserción o eliminación ordinaria en un árbol de clase *wb[a]* elimina al árbol de dicha clase, se usan rotaciones para restaurar la propiedad de pesos balanceados.

Otro tipo de árbol, llamado por Tarjan, un *árbol binario balanceado*, requiere que para todo nodo *nd*, la longitud del camino más largo de *nd* a un nodo externo sea a lo sumo dos veces la longitud del camino más corto de *nd* a un nodo externo. (Recuérdese que los nodos externos son nodos agregados al árbol en cada apuntador nulo.) De nuevo, se usan rotaciones para mantener el balance después de una inserción o eliminación. Los árboles balanceados de Tarjan tienen la propiedad de que, tras una eliminación o inserción, puede restaurarse el balance aplicando a lo sumo una rotación doble y una simple, en contraste con las posibles  $O(\log n)$  rotaciones tras la eliminación en un árbol de altura balanceada.

Los árboles balanceados también pueden usarse para una implantación eficiente de colas de prioridad (ver secciones 4.1 y 6.3.) La inserción de un nuevo elemento requiere a lo sumo  $O(\log n)$  pasos para encontrar la posición adecuada y  $O(1)$  pasos para accesar el elemento (siguiendo los apuntadores izquierdos hasta la hoja de la extrema izquierda) y  $O(\log n)$  ó  $O(1)$  pasos para eliminar esa hoja. Así, al igual que una cola de prioridad implantada usando un heap (sección 6.3), una cola de prioridad implantada usando un árbol balanceado puede ejecutar cualquier secuencia de  $n$  inserciones y eliminaciones mínimas en  $O(n \log n)$  pasos.

## EJERCICIOS

- 7.2.1. Escriba un algoritmo de inserción eficiente para un árbol de búsqueda binaria con el fin de insertar un nuevo registro cuya llave se sabe que no está en el árbol.
  - 7.2.2. Muestre que es posible obtener un árbol de búsqueda binaria en el cual existe sólo una hoja, aun cuando los elementos del árbol no se inserten obligatoriamente en orden ascendente o descendente.
  - 7.2.3. Verifique mediante simulación que si se presentan registros en orden aleatorio al algoritmo de búsqueda e inserción para un árbol binario, el número de comparaciones de llave es  $O(\log n)$ .
  - 7.2.4. Demuestre que no todo árbol de búsqueda binaria de  $n$  nodos tiene igual probabilidad (suponiendo que los nodos se insertan en orden aleatorio), y que los árboles balanceados son más probables que los árboles ‘rectilíneos’.
  - 7.2.5. Escriba un algoritmo para eliminar un nodo de un árbol binario que reemplace el nodo por su antecesor en orden, en lugar de por su sucesor en orden.
  - 7.2.6. Suponga que el tipo de nodo de un árbol de búsqueda binaria se define de la siguiente manera:
- ```

struct nodetype {
    int k;
    int r;
    struct nodetype *left;
    struct nodetype *right;
};
```
- Los campos *r* y *k* contienen el registro y la llave del nodo; *left* y *right* son apuntadores a los hijos del nodo. Escribir una rutina en C *sinsert(tree, key, rec)* para buscar e insertar un registro *rec* con llave *key* en un árbol de búsqueda binaria apuntado por *tree*.
- 7.2.7. Escriba una rutina en C, *sdelete(tree, key)* para buscar y eliminar un registro *record* con llave *key* de un árbol de búsqueda binaria implantado como en el ejercicio previo. Si se encuentra dicho registro, la función regresa el valor de su campo *r*; si no, regresa 0.
 - 7.2.8. Escriba una rutina en C *delete(tree, key1, key2)* para eliminar todos los registros con llaves (inclusive) entre *key1* y *key2* de un árbol de búsqueda binaria cuyos nodos están declarados como en los ejercicios previos.

7.2.9. Considere los árboles de búsqueda de la figura 7.2.8.

- ¿Cuántas permutaciones de los enteros del 1 al 7 producirían los árboles de búsqueda binaria de la figura 7.2.8a, b y c, de manera respectiva?
- ¿Cuántas permutaciones de los enteros del 1 al 7 producen árboles de búsqueda binaria similares a los de la figura 7.2.8a, b y c, de manera respectiva? (Ver el ejercicio 5.1.9.)
- ¿Cuántas permutaciones de los enteros del 1 al 7 producen árboles de búsqueda binaria con el mismo número de nodos en cada nivel como los árboles de la figura 7.2.8a, b y c de manera respectiva?
- Encuentre una asignación de probabilidades a los siete primeros números enteros positivos como argumentos de búsqueda que hagan a cada uno de los árboles de la figura 7.2.8a, b y c óptimos.

- 7.2.10. Muestre que el árbol de Fibonacci de orden $h + 1$ (ver ejercicio 5.3.5) es un árbol balanceado por altura con altura h y tiene menos nodos que cualquier otro árbol balanceado por altura con altura h .

7.3. ÁRBOLES DE BUSQUEDA GENERALES

Los árboles no binarios generales también se usan como tablas de búsqueda, en particular en la memoria externa. Existen dos grandes categorías de dichos árboles: árboles de búsqueda de accesos múltiples y árboles de búsqueda digitales. Examinaremos cada uno de ellos.

Árboles de búsqueda de accesos múltiples

En un árbol de búsqueda binaria, cada nodo *nd* contiene una sola llave y apunta a dos subárboles. Uno de esos subárboles contiene todas las llaves del árbol con raíz en *nd* que son menores que la llave en *nd* y el otro subárbol contiene todas las llaves en el árbol con raíz en *nd* que son mayores que (o iguales a) la llave en *nd*.

Podemos extender este concepto a un árbol de búsqueda general en el cual cada nodo contiene una o más llaves. Un *árbol de búsqueda de accesos múltiples de orden n* es un árbol general en el cual cada nodo tiene *n* o menos subárboles y contiene una llave menor que la cantidad de subárboles. Es decir, si un nodo tiene cuatro subárboles, contiene tres llaves. Además, si s_0, s_1, \dots, s_{m-1} son los *m* subárboles de un nodo que contiene llaves k_0, k_1, \dots, k_{m-2} en orden ascendente, todas las llaves en el subárbol s_0 son menores o iguales que k_0 , todas las llaves en el subárbol s_j (donde j está entre 1 y $m - 2$) son mayores que k_{j-1} y menores o iguales que k_j , y todas las llaves en el subárbol s_{m-1} son mayores que k_{m-2} . El subárbol s_j se llama el *subárbol izquierdo* de llave k_j y su raíz el *hijo izquierdo* de llave k_j . De manera similar s_j se llama el *subárbol derecho* y su raíz el *hijo derecho*, de llave k_{j-1} . Uno o más subárboles de un nodo pueden estar vacíos. (Algunas veces, se usa el término ‘árbol de búsqueda de accesos múltiples’ para hacer referencia a cualquier árbol no binario usado para la búsqueda, incluyendo los árboles digitales que presentaremos al final de esta sección. Sin embargo, nosotros usamos el término sólo para árboles que puedan contener llaves completas en cada nodo.)

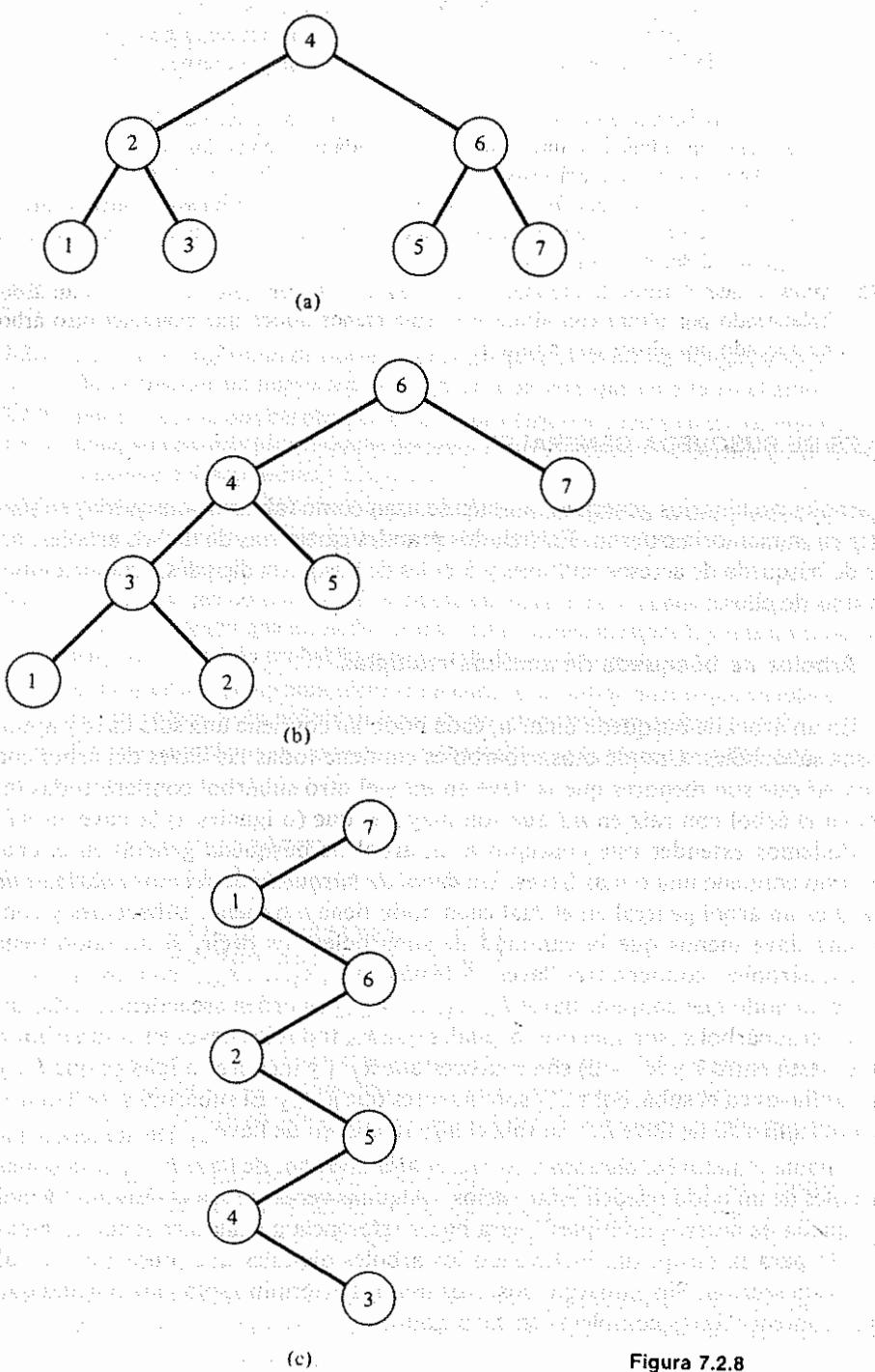


Figura 7.2.8

La figura 7.3.1 ilustra un árbol de búsqueda multivías. La 7.1.3a muestra uno de orden 4. Los ocho nodos de ese árbol se etiquetaron desde A hasta H . Los nodos A, D, E y G contienen el número máximo de subárboles, 4, y el número máximo de llaves, 3. Nodos de ese tipo se llaman *nodos llenos*. Sin embargo, algunos de los subárboles de los nodos D y E y todos los subárboles de nodos G están vacíos como se indica mediante flechas que parten de las posiciones correspondientes en los nodos. Los nodos B, C, F y H no están llenos y también contienen algunos subárboles vacíos. El primer subárbol de A contiene las llaves 6 y 10, ambas menores que 12, que es la primera llave de A . El segundo subárbol de A contiene 25 y 37, ambas mayores que 12 (que es la primera llave de A) y menor que 50 (la segunda llave de A). La tercera llave contiene 60, 70, 80, 62, 65 y 69, de los cuales todos están entre 50 (la segunda llave de A) y 85 (la tercera llave). Finalmente, el último subárbol de A contiene 100, 120, 150 y 110, todas mayores que 85 la última llave en el nodo A . De manera similar, cada subárbol de cualquier otro nodo contiene sólo llaves entre las dos llaves de ese nodo y sus ancestros.

La figura 7.3.1b ilustra un *árbol de búsqueda multivías de arriba abajo*. Tal árbol se caracteriza por la condición de que todo nodo lleno es una hoja. Adviértase que el árbol de la figura 7.3.1a no es “de arriba abajo”, dado que el nodo C lleno contiene un subárbol no vacío. Definir una *semihojas* como un nodo con un subárbol vacío al menos. En la figura 7.3.1a, los nodos del B al H son todos semihojas. En la figura 7.3.1b, los nodos del B al G y del I al R son semihojas. En un árbol de multivías de arriba abajo, una semihojas tiene que estar llena o bien ser una hoja.

La figura 7.3.1c es otro árbol de búsqueda de accesos múltiples de orden 3. No es de “arriba abajo” dado que hay cuatro nodos con sólo una llave y subárboles que no están por completo vacíos. Sin embargo, tiene la propiedad especial de ser *balanceado*. Es decir, todas sus semihojas están en el mismo nivel (3). Esto implica que todas las semihojas son hojas. Ni el árbol de la figura 7.3.1a (que tiene hojas en los niveles 1 y 2) ni el de la figura 7.3.1b (con hojas en los niveles 2, 3 y 4) son árboles de búsqueda de accesos múltiples balanceados. (Nótese que aunque un árbol de búsqueda binaria es un árbol de búsqueda de accesos múltiples de orden 2, un árbol de búsqueda binaria balanceado como se definió al final de la sección 7.2 no es por fuerza balanceado como árbol de búsqueda de accesos múltiples, dado que puede tener hojas en diferentes niveles.)

Búsqueda en un árbol de accesos múltiples

El algoritmo para buscar en un árbol de búsqueda multivías, sin tener en cuenta si es “de arriba a abajo” o no, balanceado o no, es directo. Cada nodo contiene un solo campo entero, un número variable de campos apuntadores y un número variable de campos llaves. Si $node(p)$ es un nodo, el campo entero $numtrees(p)$ es igual al número de subárboles de $node(p)$. $numtrees(p)$ siempre es menor o igual que el orden del árbol, n . Los campos apuntadores $son(p, 0)$ hasta $son(p, numtrees(p)-1)$ apuntan a los subárboles de $node(p)$. Los campos llaves $k(p, 0)$ hasta $k(p, numtrees(p)-2)$ son las llaves contenidas en $node(p)$ en orden ascendente. El subárbol al que apunta $son(p, i)$ (para i entre 1 y $numtrees(p)-2$ inclusive) contiene todas las llaves del árbol entre $k(p, i-1)$ y $k(p, i)$. $son(p, 0)$ apunta a un subárbol

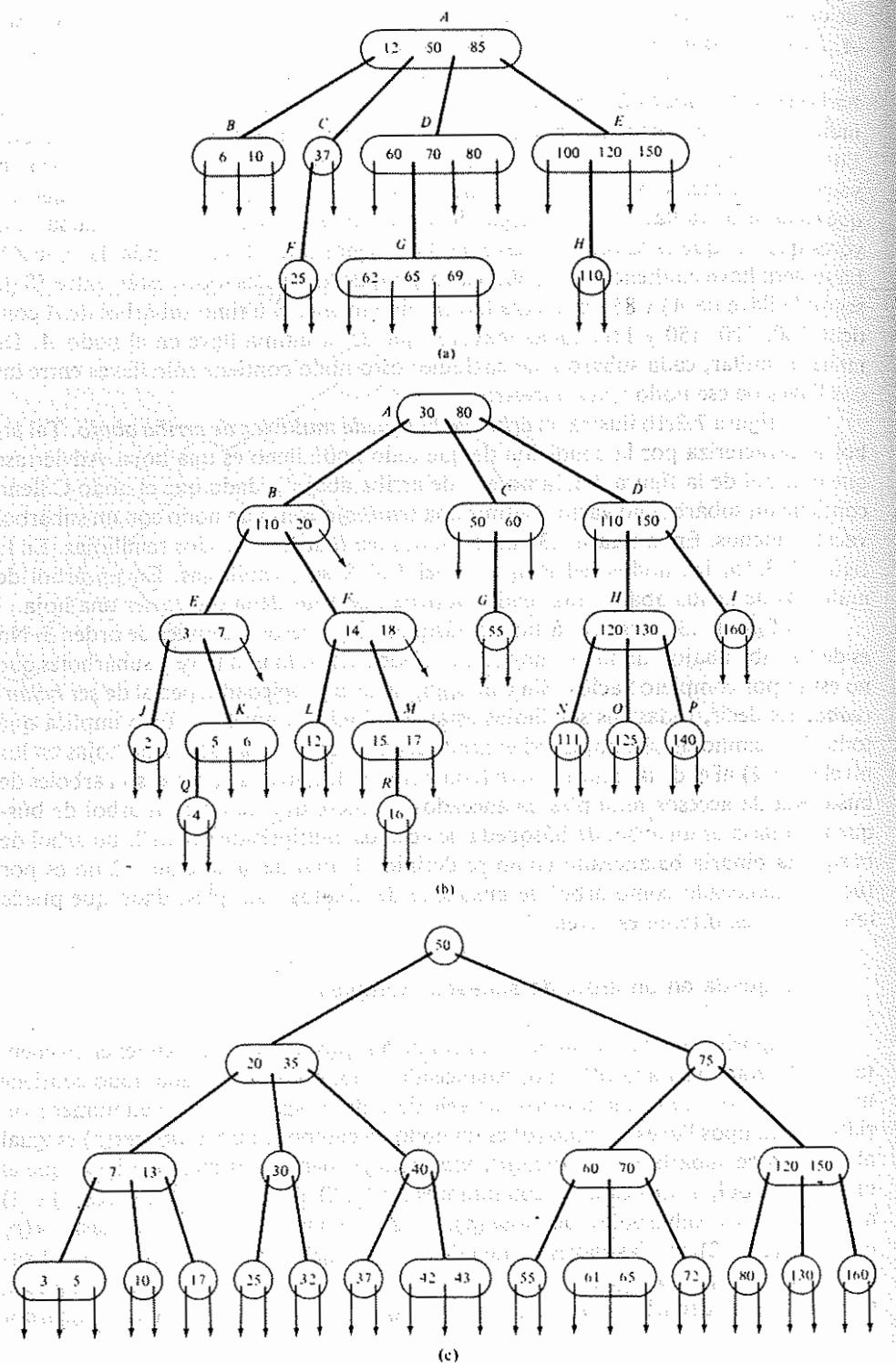


Figura 7.3.1 Árboles de búsqueda multivías.

que contiene sólo llaves menores que $k(p, 0)$ y $son(p, numtrees(p)-1)$ apunta a un subárbol que contiene sólo llaves mayores que $k(p, numtrees(p)-2)$.

También suponemos una función $nodeSearch(p, key)$ que da como resultado el menor entero j , tal que $key \leq k(p, j)$, o $numtrees(p)-1$ si key es mayor que todas las llaves en $node(p)$. (Discutiremos en breve cómo se implanta $nodeSearch$). El siguiente algoritmo recursivo es para una función $search(tree)$ que da como resultado un apuntador al nodo que contiene key (ó -1 [representando *nulo* 0] si no hay tal nodo en el árbol) y hace la variable global $position$ igual a la posición de key en ese nodo.

```
p = tree;
if (p == null) {
    position = -1;
    return(-1);
} /* fin de if */
i = nodeSearch(p, key);
if (i < numtrees(p) - 1 && key == k(p,i)) {
    position = i;
    return(p);
} /* fin de if */
return(search(son(p,i)));
```

Obsérvese que después de hacer i igual a $nodeSearch(p, key)$, insistimos en verificar que $i < numtrees(p) - 1$ antes de accesar $k(p, i)$. Esto es para evitar el uso de $k(p, numtrees(p) - 1)$ erróneo o no existente, en el caso que key sea mayor que todas las llaves en $node(p)$. Lo que sigue es una versión no recursiva del algoritmo anterior.

```
p = tree;
while (p != null) {
    /* Buscar el subárbol rotado en node(p) */
    i = nodeSearch(p, key);
    if (i < numtrees(p) - 1 && key == k(p,i)) {
        position = i;
        return(p);
    } /* fin de if */
    p = son(p,i);
} /* fin de while */
position = -1;
return(-1);
```

La función $nodeSearch$ es responsable de la localización de la llave más pequeña en un nodo mayor o igual que el argumento de búsqueda. La técnica más simple para hacer esto es una búsqueda secuencial a través del conjunto ordenado de llaves en el nodo. Si todas las llaves son de igual longitud fija, se puede usar también una búsqueda binaria para localizar la llave apropiada. La decisión de cuándo usar búsqueda secuencial o binaria depende del orden del árbol que determina cuántas

llaves tienen que ser buscadas. Otra posibilidad es organizar las llaves dentro del nodo como un árbol de búsqueda binaria.

Implantación de un árbol de accesos múltiples

Nótese que hemos implantado un árbol de búsqueda multivías de orden n usando nodos de hasta n hijos en lugar de como un árbol binario con apuntadores a hijo y hermano, como se esbozó en la sección 5.5 para árboles generales. La razón para esto es que en árboles de accesos múltiples, a diferencia de árboles en general, hay un límite para el número de hijos de un nodo y podemos esperar que la mayoría de los nodos estén tan llenos como sea posible. En un árbol general no había tal límite, y muchos nodos podían contener sólo uno o dos elementos. En consecuencia, la flexibilidad de permitir tantos o tan pocos elementos como fuera necesario y el ahorro de espacio cuando un nodo estaba casi vacío hacían meritoria la sobrecarga de los apuntadores a hermanos adicionales.

No obstante, cuando los nodos no están llenos, la implantación de árboles multivías que hicimos aquí usa una cantidad considerable de memoria. A pesar de este posible gasto de memoria, los árboles multivías se usan con bastante frecuencia para almacenar datos, en especial en un dispositivo externo de acceso directo como un disco. La razón para ello es que el acceso de cada nodo nuevo, durante una búsqueda, requiere de la lectura de un bloque de memoria del dispositivo externo. Esta operación de lectura es relativamente costosa en términos de tiempo a causa del trabajo mecánico involucrado en colocar el dispositivo de manera apropiada. Sin embargo, una vez colocado el dispositivo, la tarea de leer una gran cantidad de datos secuenciales es muy rápida. Esto significa que el tiempo total para leer un bloque de memoria (es decir un "nodo") es afectado sólo de manera mínima por su tamaño. Una vez leído y trasladado un nodo a la memoria interna de la computadora, el costo de su búsqueda a velocidad electrónica interna es minúsculo comparado con el costo de su lectura inicial en la memoria. Además, el almacenamiento externo es poco costoso de manera que una técnica que perfeccione la eficiencia en tiempo a expensas del espacio de almacenamiento externo es efectiva en costo real (es decir, en dinero). Por esta razón, los sistemas de memoria externa basados en árboles de búsqueda multivías intenta incrementar al máximo el tamaño de cada nodo, y árboles de orden 200 ó más no son poco comunes.

El segundo factor a considerar en la implantación de árboles de búsqueda multivías es el almacenamiento de los propios registros de datos. Como en cualquier sistema de memoria, los registros pueden ser almacenados con las llaves o lejos de ellas. La primera técnica requiere guardar registros completos dentro de los nodos del árbol, mientras que la segunda requiere guardar un apuntador al registro asociado con cada llave en un nodo. (Una tercera técnica involucra la duplicación de las llaves y guardar los registros sólo en las hojas. Este mecanismo se discute más tarde con mayor detalle cuando presentamos los B⁺-árboles.)

En general, quisiéramos guardar tantas llaves como sea posible en cada nodo. Para ver por qué esto es así, considerar dos árboles "de arriba a abajo" con 4000 llaves y profundidad mínima, uno de orden 5 y otro de orden 11. El árbol de orden-5 requiere 1000 nodos (de 4 llaves cada uno) para guardar las 4000 llaves, mientras que

el árbol del orden-11 sólo requiere 400 nodos (de 10 llaves cada uno). La profundidad del árbol de orden-5 es por lo menos 5 (nivel 0 con un nodo, nivel 1 con 5, nivel 2 con 25, nivel 3 con 125, nivel 4 con 625 y nivel 5 con las 219 restantes), mientras que la profundidad del árbol de orden-11 puede ser tan pequeña como 3 (nivel 0 con un nodo, nivel 1 con 11, nivel 2 con 121, nivel 3 con las 267 restantes). Así, para buscar en el árbol de orden-5 tienen que accederse 5 ó 6 nodos para la mayoría de las llaves y en el árbol de orden-11 sólo tienen que accederse 3 ó 4 nodos. Pero como observamos arriba, el acceso de un nodo es la operación más costosa en la búsqueda de la memoria externa, donde son usados con más frecuencia los árboles multivías. Así, un árbol de orden mayor conduce a un proceso de búsqueda más eficiente. La memoria real requerida por ambas situaciones es aproximadamente la misma, dado que, aunque se requieren menos nodos grandes para guardar un archivo de un tamaño dado, cuando el orden es grande, cada nodo es más grande.

Como el tamaño de un nodo está en general fijado por otros factores externos (por ejemplo, la cantidad de memoria que se lee físicamente de un disco en una operación), un árbol de orden mayor se obtiene guardando los registros fuera de los nodos del árbol. Aun cuando esto causa una lectura externa extra para obtener un registro después de que su llave ha sido localizada, guardar los registros dentro de un nodo reduce de manera típica el orden en un factor entre 5 y 40 (que es el rango típico del cociente del tamaño del registro al de la llave), de manera que el cambio no vale la pena.

Si se guarda un árbol de búsqueda multivías en memoria externa, un apuntador a un nodo es una dirección de memoria externa que especifica la posición de comienzo de un bloque de almacenamiento. El bloque que compone un nodo tiene que ser leído en la memoria interna antes de que puedan ser accesados cualquiera de los campos *numtrees*, *k* o *son*. Suponer que la rutina *directread(p, block)* lee un nodo en la dirección de la memoria externa *p* dentro de un registro de memoria interna *block*, suponer también que los campos *numtrees*, *k* y *son* en el registro se acceden mediante la notación similar en C *block.numtrees*, *block.k* y *block.son*. Suponer además que se modifica la función *nodesearch* para aceptar un bloque (interno) de memoria en lugar de un apuntador (es decir, se invoca mediante *nodesearch(block, key)* y no mediante *nodesearch(p, key)*). Entonces el siguiente es un algoritmo no recursivo para buscar en un árbol de búsqueda multivías almacenado en la memoria externa.

```
p = tree;
while (p != null) {
    directread(p, block);
    i = nodesearch(block, key);
    if (i < block.numtrees - 1 && key == block.son(i)) {
        position = i;
        return(p);
    } /* fin de if */
    p = block.son(i);
} /* fin de while */
position = -1;
return(-1);
```

El algoritmo también hace *block* igual al nodo en la dirección externa *p*. El registro asociado a *key* o un apuntador a él puede encontrarse en *block*. Nótese que *null*, tal y como se usa en este algoritmo, se refiere a una dirección de memoria externa nula en lugar del apuntador *nulo* de C.

Recorrido de un árbol multivías

Una operación común con una estructura de datos es el *recorrido*: acceso de todos los elementos de la estructura en una secuencia fija. Lo que sigue es un algoritmo recursivo *traverse(tree)* para recorrer un árbol multivías e imprimir sus llaves en orden ascendente.

```
if (tree != null) {
    nt = numtrees(tree);
    for (i = 0; i < nt - 1; i++) {
        traverse(son(tree, i));
        printf("%d", k(tree, i));
    } /* fin de for */
    traverse(son(tree, nt));
} /* fin de if */
```

En la implantación de la recursividad, tenemos que guardar una pila de apuntadores a todos los nodos en un camino comenzando con la raíz del árbol hasta el nodo que está siendo visitado en el momento.

Si cada nodo es un bloque de memoria externa y *tree* es la dirección de almacenamiento de memoria externa del nodo raíz, un nodo tiene que leerse en la memoria interna antes de que puedan ser accesados sus campos *son* o *k*. Así el algoritmo se convierte en:

```
if (tree != null) {
    directread(tree, block);
    nt = block.numtrees;
    for (i = 0; i < nt - 1; i++) {
        traverse(block.son(i));
        printf("%d", block.k(i));
    } /* fin de for */
    traverse(block.son(nt));
} /* fin de if */
```

donde *directread* es una rutina del sistema que lee un bloque de memoria en una dirección externa particular (*tree*) dentro del buffer de memoria interna (*block*). Esto requiere también guardar una pila de registros. Si *d* es la profundidad del árbol, se tienen que guardar en la memoria *d* + 1 registros.

De manera alternativa, casi todos los buffers pueden eliminarse si cada nodo contiene dos campos adicionales: un campo *father* apuntando a su padre y un campo *index* indicando cuál hijo del padre es el nodo. Entonces cuando el último subárbol de un nodo ha sido recorrido, el algoritmo usa el campo *father* del nodo para

acceder a su padre y su campo *index* y determinar cuál llave del nodo padre dar como salida y cuál subárbol del padre recorrer como paso siguiente. Sin embargo, esto requeriría numerosas lecturas para cada nodo y es probable que no valga la pena el ahorro de espacio de los buffers, especialmente debido a que un árbol de orden elevado con un número muy grande de llaves requiere una profundidad muy pequeña. (Como ya se ilustró, un árbol de orden 11 con 4000 llaves se puede acomodar muy bien con profundidad 3. Un árbol de orden 100 puede acomodar cerca de un millón de llaves con una profundidad de sólo 2.)

Otra operación común relacionada muy de cerca al recorrido es el *acceso secuencial directo*. Esto se refiere a accesar la llave que sigue a una llave cuya localización en el árbol es conocida. Supongamos que localizamos una llave *k1* mediante una búsqueda en el árbol y que ella está en la posición *k(n1, i1)*. Por lo general, el sucesor de *k1* puede encontrarse ejecutando la siguiente rutina *next(n1, i1)*. (*nullkey* es un valor especial que indica que la llave apropiada no puede ser encontrada.)

```
p = son(n1, i1 + 1);
q = null; /* ... q está un nodo detrás de p ... */
while (p != null) {
    q = p;
    p = son(p, 0);
} /* fin de while */
if (q != null)
    return(k(q, 0));
if (i1 < numtrees(n1) - 2)
    return(k(n1, i1 + 1));
return(nullkey);
```

Este algoritmo se basa en el hecho de que el sucesor de *k1* es la primera llave en el subárbol que sigue a *k1* en *node(n1)*, o si ese subárbol está vacío [*son(n1, i1 + 1)* es igual a *null*] y si *k1* no es la última llave en su nodo (*i1 < numtrees(n1) - 2*), el sucesor es la llave siguiente en *node(n1)*.

Sin embargo, si *k1* es la última llave en su nodo y si el subárbol que le sigue está vacío, su sucesor sólo puede ser encontrado regresándose en el árbol. Suponiendo campos *father* e *index* en cada nodo como se esbozó con anterioridad, se puede escribir un algoritmo completo para encontrar el sucesor de la llave en la posición *i1*, *sucesor(n1, i1)*, del nodo apuntado por *n1* de la siguiente manera:

```
p = son(n1, i1 + 1);
if (p != null && i1 < numtrees(n1) - 2)
    /* ... utilizar el algoritmo anterior ... */
    return(next(n1, i1));
f = father(n1);
i = index(n1);
while (f != null && i == numtrees(f) - 1) {
    i = index(f);
    f = father(f);
} /* fin de while */
```

```

if (f == null)
    return(NULLKEY);
return(k(f, i));

```

Por supuesto, quisiéramos evitar el regreso en el árbol siempre que sea posible. Como un recorrido que inicie en una llave es bastante común, el proceso de búsqueda inicial se modifica con frecuencia para retener en la memoria interna, todos los nodos en el camino de la raíz del árbol a la llave localizada. Entonces, si se tiene que regresar en el árbol, el camino a la raíz está disponible con facilidad. Como ya se observó, si esto se hace, no se necesitan los campos *father* e *index*.

Posteriormente en esta sección examinamos una adaptación especializada de un árbol de búsqueda multivías, llamado un B^+ -árbol, que no requiere una pila para el recorrido secuencial eficiente.

Inserción en un árbol de búsqueda de accesos múltiples

Ahora que hemos examinado cómo buscar y recorrer árboles de búsqueda multivías, hagámoslo con las técnicas de inserción para esas estructuras. Examinamos dos técnicas de inserción para árboles de búsqueda multivías. La primera es análoga a la inserción en un árbol de búsqueda binaria y da como resultado un árbol de búsqueda multivías de "arriba a abajo". La segunda es una nueva técnica de inserción y produce un tipo especial de árboles de búsqueda multivías balanceado. Es esta segunda técnica, o una leve variación de la misma, la que se usa con mayor frecuencia en sistemas de almacenamiento externo de archivos de acceso directo.

Por conveniencia, suponemos que no se permiten llaves duplicadas en el árbol, de manera que si se encuentra el argumento *key* no ocurren inserciones. Suponemos que el árbol no está lleno. El primer paso en ambos procedimientos de inserción es buscar el argumento *key*. Si se encuentra dicho argumento en el árbol, se obtiene como resultado un apuntador al nodo que contiene la llave y se asigna a la variable *position* su posición dentro del nodo, tal y como en el procedimiento de búsqueda presentado antes. Sin embargo, si no se encuentra el argumento *key*, se da como resultado un apuntador a la semioja *node(s)* que contendría la llave si estuviera presente, y *position* se hace igual al índice de la llave más pequeña en *node(s)* que sea mayor que el argumento *key* (es decir, la posición del argumento *key* si estuviera presente en el árbol). Si todas las llaves en *node(s)* son menores que el argumento *key*, *position* se hace igual a *numtrees(s)* - 1. Una variable *found* se hace igual al *verdadero* o *falso* dependiendo de si se encuentra o no la llave en el árbol.

La figura 7.3.2 ilustra el resultado de este procedimiento para un árbol de búsqueda de accesos múltiples balanceado "top-down" de orden-4 y varios argumentos de búsqueda. El algoritmo para *find* es directo:

```

q = null;
p = tree;
while (p != null) {
    i = nodesearch(p, key);
    q = p;
    if (i < numtrees(p) - 1 && key == k(p, i)) {

```

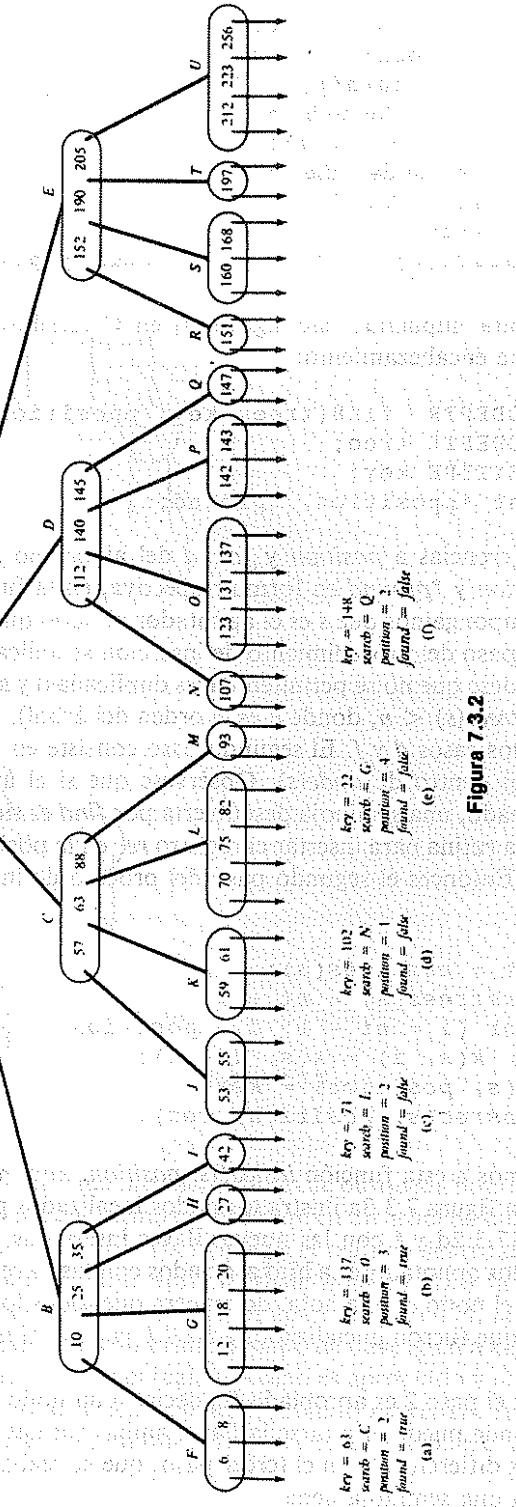


Figura 7.3.2

```

        found = TRUE;
        position = i;
        return(p); /* la llave se encontró en node(p) */
    } /* fin de if */
    P = son(p, i);
} /* fin de while */
found = FALSE;
position = i;
return(q); /* p es null. q apunta a una semihoya */

```

Para implantar este algoritmo en C escribiríamos una función *find* con el siguiente encabezamiento:

```

NODEPTR find(tree, key, pposition, pfound)
NODEPTR tree;
KEYTYPE key;
int *pposition, *pfound;

```

Las referencias a *position* y *found* del algoritmo se remplazan por referencias a **pposition* y **pfound* en forma respectiva, en la función en C.

Supongamos que *s* es el apuntador al nodo que da como resultado *find*. El segundo paso del procedimiento de inserción se aplica sólo si no se encuentra la llave (recuérdese que no se permiten llaves duplicadas) y si *node(s)* no está lleno (es decir, si *numtrees(s) < n*, donde *n* es el orden del árbol). En la figura 7.3.2 esto se aplica sólo a los casos *d* y *f*. El segundo paso consiste en la inserción de la nueva llave (y registro) dentro de *node(s)*. Obsérvese que si el árbol es “de arriba a abajo” o balanceado, una semihoya descubierta por *find* es siempre una hoja. Sea *insrec(p, i, rec)* una rutina para insertar el registro *rec* en la posición *i* de *node(p)* como es apropiado. Entonces el segundo paso del proceso de inserción puede describirse como sigue:

```

nt = numtrees(s);
numtrees(s) = nt + 1;
for (i = nt - 1; i > position; i--)
    k(s, i) = k(s, i - 1);
k(s, position) = key;
insrec(s, position, rec);

```

Llamamos a esta función *insleaf(s, position, key, rec)*.

La figura 7.3.3a ilustra los nodos localizados por el procedimiento *find* en las figuras 7.3.2d y f con las nuevas llaves insertadas. Obsérvese que no es necesario copiar los apuntadores a hijo asociados con las llaves que están siendo movidas, dado que el nodo es una hoja, de manera que todos los apuntadores son nulos. Suponemos que fueron inicializados a *NULL* cuando el nodo fue en el inicio agregado al árbol.

Si el paso 2 es apropiado (es decir, si un nodo hoja no lleno ha sido encontrado), donde puede insertarse la llave, ambas rutinas de inserción terminan. Las dos técnicas difieren sólo en el tercer paso, que se realiza cuando el procedimiento *find* localiza una semihoya llena.

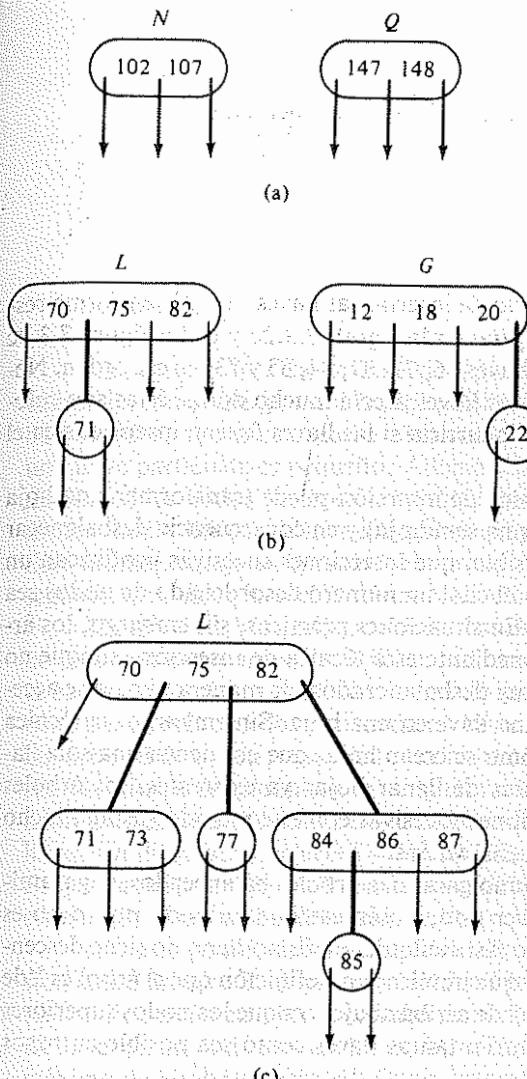


Figura 7.3.3

La primera técnica de inserción, que produce árboles de búsqueda multivías de “arriba a abajo”, imita la acción del algoritmo de inserción en un árbol de búsqueda binaria. Es decir, asigna un nuevo nodo, inserta la llave y el registro dentro del mismo y coloca el nuevo nodo como el hijo apropiado de *node(s)*. Usa la rutina *maketree(key, rec)* para asignar un nodo, hace a los *n* apuntadores en él, iguales a *NULL*, sus campos *numtrees* iguales a 2 y su primer campo de llave igual a *key*. *Maketree* llama después a *insrec* para insertar el registro como es apropiado y da como resultado final un apuntador al nodo recién asignado. Usando *maketree*, la rutina *insfull* para insertar la llave cuando la semihoya apropiada está llena puede implementarse de manera trivial como

```

p = maketree(key, rec);
son(s, position) = p;

```

Si se guardan en cada nodo campos *father* e *index*, las operaciones

```

father(p) = s;
index(p) = position;

```

se requieren también.

La figura 7.3.3b ilustra el resultado de insertar las llaves 71 y 22, en forma respectiva, en los nodos localizados por *find* en la figura 7.3.2c y e. La figura 7.3.3c ilustra inserciones sucesivas de las llaves 86, 77, 87, 84, 85 y 73, en ese orden. Nótese que el orden en el cual se insertan las llaves afecta mucho donde éstas son colocadas. Por ejemplo, considerar lo que ocurriría si las llaves fueran insertadas en el orden 85, 77, 86, 87, 73, 84.

Obsérvese también que esta técnica de inserción puede transformar una hoja en no-hoja (aunque permanece como una semihoya) y en consecuencia desbalancear el árbol multivías. Es por lo tanto posible, que inserciones sucesivas produzcan un árbol fuertemente desbalanceado y en el cual un número desordenado de nodos sea accesado para localizar ciertas llaves. En situaciones prácticas, sin embargo, los árboles de búsqueda multivías creados mediante esta técnica de inserción, aunque no balanceados en su totalidad, no son muy desbalanceados, de manera que no se acceden muchos nodos cuando se busca una llave en una hoja. Sin embargo, la técnica tiene una desventaja fundamental. Como se crean hojas que contienen una sola llave, y se pueden crear otras hojas antes de llenar hojas ya existentes, los árboles multivías creados por medio de inserciones sucesivas con este método gastan mucho más espacio en nodos hoja que están casi vacíos.

Aunque este método de inserción no garantiza árboles balanceados, sí garantiza árboles “de arriba abajo”. Para ver esto, obsérvese que un nodo nuevo no es creado a no ser que su padre esté lleno. Así cualquier nodo no-lleño no tiene descendientes y es, por lo tanto, una hoja, lo que implica por definición que el árbol es “de arriba abajo”. La ventaja de un árbol “de arriba abajo” es que los nodos superiores están llenos, de manera que se encuentren tantas llaves como sea posible en pasos cortos.

Antes de examinar la segunda técnica de inserción, juntamos todas las piezas de la primera técnica para formar un algoritmo completo de búsqueda e inserción para árboles de búsqueda multivías de “arriba abajo”.

```

if (tree == null) {
    tree = maketree(key, rec);
    position = 0;
    return (tree);
} /* fin de if */
s = find(tree, key, position, found);
if (found == TRUE)
    return (s);

```

```

if (numtrees(s) < n) {
    insleaf(s, position, key, rec);
    return(s);
} /* fin de if */
p = maketree(key, rec);
son(s, position) = p;
position = 0;
return (p);

```

Arboles-B

La segunda técnica de inserción para árboles de búsqueda multivías es más compleja. Sin embargo, esta complejidad se compensa por el hecho de crear árboles balanceados, de manera que el número máximo de nodos accesados para encontrar una llave particular es pequeño. Además, la técnica ofrece la ventaja adicional, de que todos los nodos (excepto la raíz) de un árbol creado mediante ella están por lo menos medio llenos, de manera que se desperdicia muy poco espacio de memoria. Esta última ventaja es la razón primaria por la cual se usa con mayor frecuencia la segunda técnica de inserción (o una variación de ella) en sistemas de archivos reales.

Un árbol de búsqueda multivías balanceado de orden n en el cual cada nodo-no raíz contiene al menos $n/2$ llaves se llama un **árbol-B de orden n** . (Obsérvese que la barra en $n/2$ denota división entera de manera que un árbol 1-B de orden 11 contiene por lo menos 5 llaves en cada nodo-no raíz, igual que un árbol-B de orden 10.) Un árbol-B de orden n también se conoce como **árbol $n-(n-1)$** o **árbol $(n-1)-n$** . (La raya fuera del paréntesis es un guion mientras que la de adentro es un signo menos.) Esto refleja el hecho de que cada nodo en el árbol tiene un máximo de $n-1$ llaves y n hijos. Así, un árbol 4-5 es un árbol-B de orden 5 como lo es un árbol 5-4. En particular, un árbol 2-3 (o 3-2) es el árbol-B trivial (es decir no binario) más elemental, como una o dos llaves y dos o tres hijos por nodo.

(En este punto, deberíamos decir algo acerca de la terminología. En la discusión sobre árboles-B, la palabra “orden” la usan de manera diferente autores diferentes. Es común encontrar el *orden* de un árbol-B definido como el número mínimo de llaves en un nodo no-raíz [es decir, $n/2$] y el *grado* de un árbol-B como el número máximo de hijos [es decir, n]. Otros autores usan “orden” para denotar el número máximo de llaves en un nodo [es decir, $n-1$]. Nosotros usamos *orden* de manera consistente para todos los árboles de búsqueda multivías para denotar el número máximo de hijos.)

Los dos primeros pasos de la técnica de inserción son los mismos para árboles-B que para árboles de “arriba abajo”. Primero, se usa *find* para encontrar la hoja en la cual se debe insertar la llave, y después si la hoja localizada no está llena, agregar la llave usando *insleaf*. Es el tercer paso, cuando la hoja localizada está llena, que difieren los métodos. En lugar de crear un nuevo nodo con una sola llave, dividir la hoja llena en dos: una hoja izquierda y una hoja derecha. Por simplicidad, suponer que n es impar. Las n llaves consistentes en las $n-1$ llaves en la hoja llena y la nueva llave a ser insertada, se dividen en tres grupos: las $n/2$ llaves menores que colocan en la hoja izquierda, las $n/2$ mayores en la hoja derecha y la del medio [tiene

que haber una dado que $2 * (n/2)$ es igual a $n - 1$ si n es impar] se coloca dentro del nodo padre si es posible (es decir, si el nodo padre no está lleno). Los dos apuntadores a cada lado de la llave insertada dentro del padre se hacen igual a las hojas derecha e izquierda recién creadas, de manera respectiva.

La figura 7.3.4 ilustra este proceso en un árbol-B de orden 5. La figura 7.3.4a muestra un subárbol de un árbol-B y la 7.3.4b muestra parte del mismo subárbol cuando éste se altera por la inserción de 382. La hoja de la extrema izquierda ya estaba llena, de manera que se dividen las cinco llaves 380, 382, 395, 406 y 412 en forma tal que 380 y 382 se colocan en una nueva hoja izquierda, 406 y 412 en una nueva hoja derecha, y la llave del medio, 395, avanza al nodo padre con apuntadores a las hojas izquierda y derecha a cada lado. No hay problema en colocar 395 en el nodo padre, dado que sólo contienen dos llaves y tiene espacio para cuatro.

La figura 7.3.4c muestra el mismo subárbol con la inserción de las llaves 518 primero y luego 508. (El mismo resultado se alcanzaría si fueran insertadas en orden inverso.) Es posible insertar 518 en forma directa en la hoja de la extrema derecha, dado que hay espacio para una llave adicional. Sin embargo, cuando llega 508, la hoja ya está llena. Las cinco llaves: 493, 506, 508, 511 y 518 se dividen de manera que las dos más pequeñas (493 y 506) se colocan en una nueva hoja izquierda, las dos mayores (511 y 518) en una nueva hoja derecha y la llave del medio (508) avance hacia el padre, que aún tiene lugar para acomodarla. Obsérvese que la llave que avanza hacia el padre siempre es la llave del medio, sin tomar en cuenta si llega primero o después que las otras.

Si el orden de un árbol-B es par, tienen que dividirse las $n - 1$ llaves (excluyendo la del medio) en dos grupos de tamaño desigual: uno de tamaño $n/2$ y otro de tamaño $(n - 1)/2$. [El segundo grupo siempre es de tamaño $(n - 1)/2$, sin tomar en cuenta si n es impar o par, dado que cuando n es impar $(n - 1)/2$ es igual a $n/2$.] Por ejemplo, si n es igual a 10, 10/2 (o 5) llaves están en un grupo, 9/2 (o 4) están en el otro y una llave avanza, para un total de 10. Estas pueden dividirse de manera que el grupo de mayor tamaño siempre esté en la hoja izquierda o en la derecha, o se pueden alternar las divisiones de manera que en una división, la hoja derecha contenga más llaves y en la siguiente la izquierda contenga más. En la práctica, no hay mucha diferencia entre ambas técnicas.

La figura 7.3.5 ilustra la polarización izquierda y derecha en un árbol-B de orden 4. Obsérvese que el elegir una rama derecha o izquierda determina cuál llave debe avanzarse hacia el padre.

Una base sobre la cual puede ser tomada la decisión en cuanto a dónde colocar más llaves en la hoja izquierda o derecha, es examinar los rangos de llaves bajo las dos posibilidades. En la figura 7.3.5b, utilizando una polarización izquierda, el rango de llaves del nodo izquierdo es de 87 a 102, o 15 y el rango de llaves en el nodo derecho es de 102 a 140 ó 38. En la figura 7.3.5c utilizando una polarización derecha, los rangos de llave son 13 (87 a 100) y 40 (100 a 140). Así, seleccionaríamos una polarización izquierda en este caso, dado que la probabilidad de que un nuevo nodo vaya a la izquierda o a la derecha es casi la misma, suponiendo que las llaves están distribuidas de manera uniforme.

Hasta ahora, en la discusión anterior supusimos que hay espacio en el padre para insertar el nodo del medio. ¿Qué ocurre si el nodo padre también está lleno?

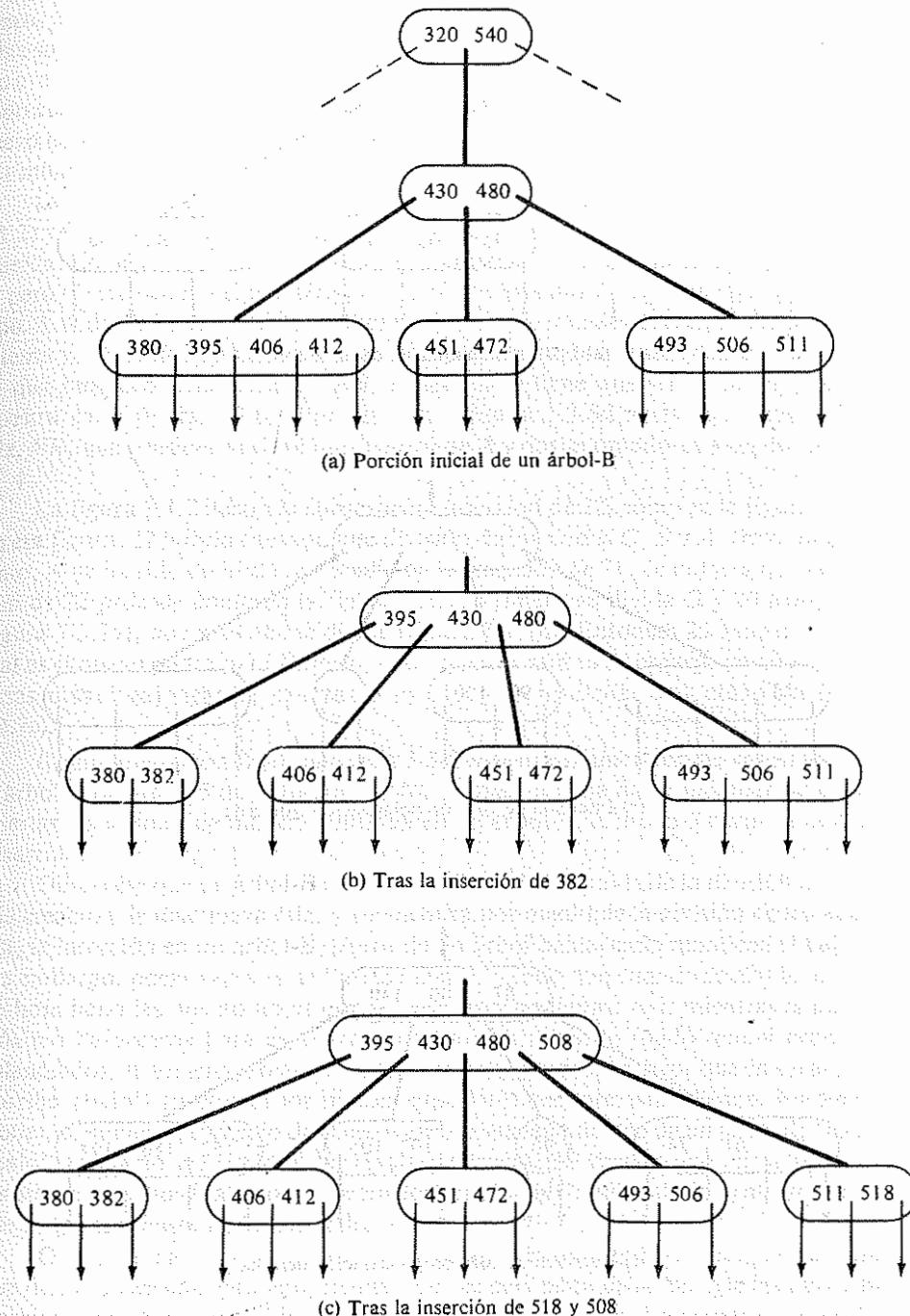


Figura 7.3.4

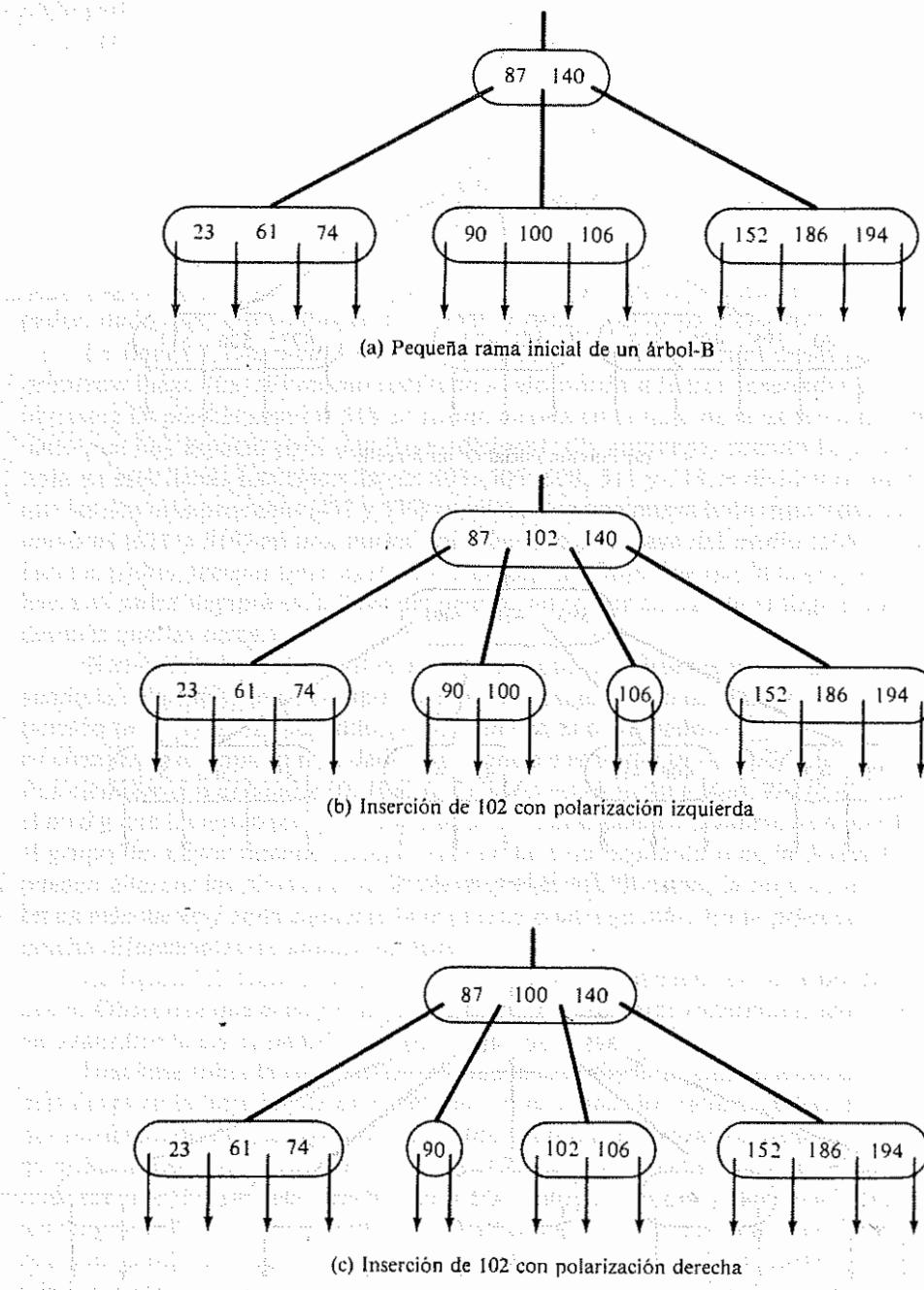


Figura 7.3.5

Por ejemplo, ¿qué pasa con la inserción de la figura 7.3.2c, si 71 tiene que insertarse dentro del nodo lleno L , y C , el padre de L , también está lleno? La solución es bastante simple. El nodo padre también se divide de la misma manera y su nodo del medio avanza a su padre. Este proceso continúa hasta que una llave sea insertada en un nodo con espacio, o se divide el propio nodo-raíz, A . Cuando esto ocurre, se crea un nuevo nodo raíz NR que contiene la llave que avanzó de la división de A y tiene a las dos mitades de A como hijos.

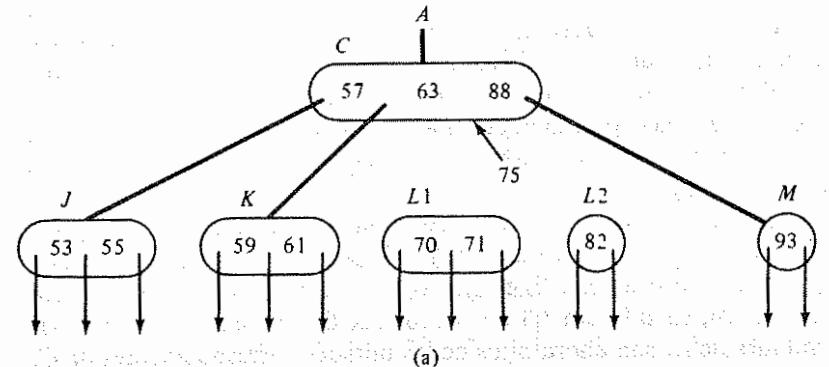
La figura 7.3.6 y la 7.3.7 ilustran este proceso con las inserciones de las figuras 7.3.2c e. En la figura 7.3.6a, se divide el nodo L . (Suponemos una polarización izquierda a lo largo de esta ilustración). El elemento medio (75) debería avanzar hacia C , pero C está lleno. Así, en la figura 7.3.6b, vemos que C también está siendo dividido. Las dos mitades de L , son ahora hijos de las mitades correspondientes de C . Setenta y cinco, que había avanzando hacia C , tiene que avanzar ahora al nodo raíz A , que tampoco tiene espacio. Así, el propio A tiene que ser dividido, como se muestra en la figura 7.3.6c. Por último, la figura 7.3.6d muestra un nuevo nodo raíz, NR , que contiene la clave que avanzó de A y dos apuntadores a las dos mitades de A .

La figura 7.3.7 ilustra la subsecuente inserción de 22, como en la figura 7.3.2e. En esa figura, 22 habría causado una división de los nodos G , B y A . Pero, mientras tanto, A ya ha sido dividido por medio de la inserción de 71, de manera que la inserción de 22 procede como en la figura 7.3.7. Primero se divide G y 20 avanza a B (figura 7.3.7a), que a su vez se divide (figura 7.3.7b). Entonces 25 avanza a $A1$, el cual es el nuevo parente para B . Pero, dado que $A1$ aún tiene espacio ya no son necesarias más divisiones. Se inserta 25 en $A1$ y la inserción está completa (figura 7.3.7c).

Como ilustración final, la figura 7.3.8 muestra la inserción de varias llaves en el árbol-B de orden-5 de la figura 7.3.4c. Valdría la pena hacer una lista de llaves e insertar las mismas de manera continua en un árbol-B de orden-5 para ver cómo se desarrolla.

Obsérvese que un árbol-B crece en profundidad a través de la división de la raíz y la creación de una nueva raíz, y en anchura por medio de la división de los nodos. Así, la inserción en un árbol-B dentro de un árbol balanceado mantiene el balance. Sin embargo, pocas veces es de “arriba abajo”, dado que cuando se divide un nodo no-hoja lleno las dos no-hojas que se crean son no-llenas. Así, mientras el número máximo de accesos para encontrar una llave es pequeño (dado que el árbol está balanceado), el número promedio de tales accesos puede ser mayor que en un árbol de “arriba abajo” en el cual los niveles superiores siempre están llenos. En simulaciones, el número promedio de accesos en la búsqueda dentro de un árbol de “arriba abajo” aleatorio en realidad ha resultado levemente menor que en la búsqueda dentro de un árbol-B aleatorio porque los árboles de “arriba abajo”, aleatorios son por lo general bastante balanceados.

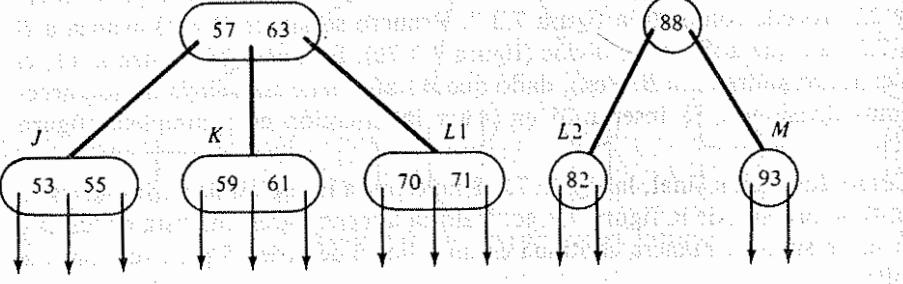
Otro punto a tomar en cuenta en un árbol-B es que las llaves antiguas (aquellas que fueron insertadas primero) tienden a estar más cerca de la raíz que las llaves más jóvenes, dado que han tenido más oportunidad de avanzar. Sin embargo, es posible que una llave permanezca en una hoja para siempre aun cuando con posterioridad se



(a)

Algoritmo de inserción en un 2-3 árbol. Se inserta la llave 100 en el nodo raíz A.

Algoritmo de inserción en un 2-3 árbol. Se inserta la llave 100 en el nodo raíz A.



(b)

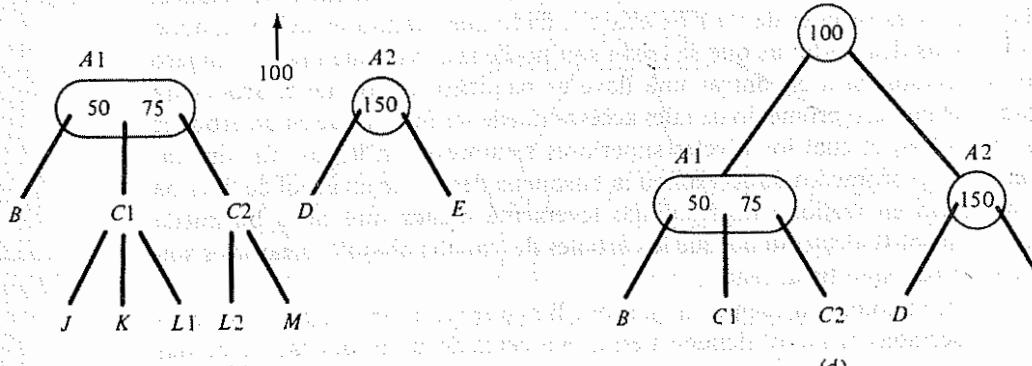
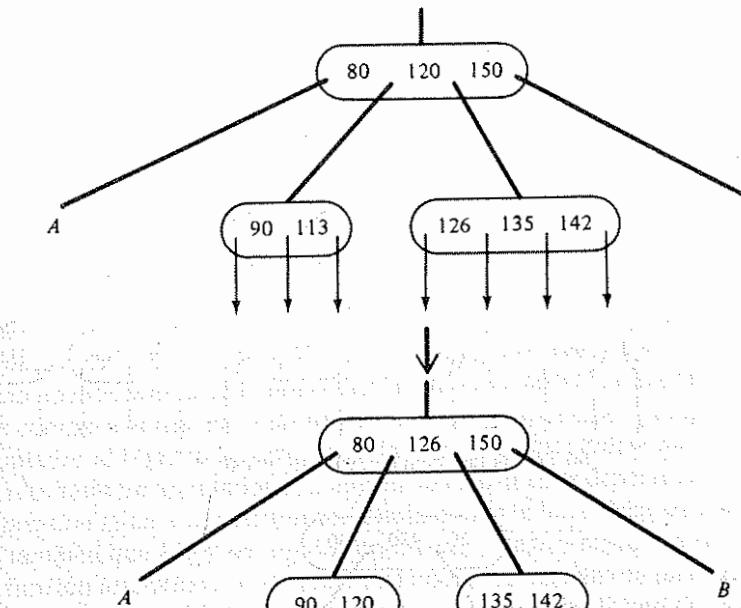
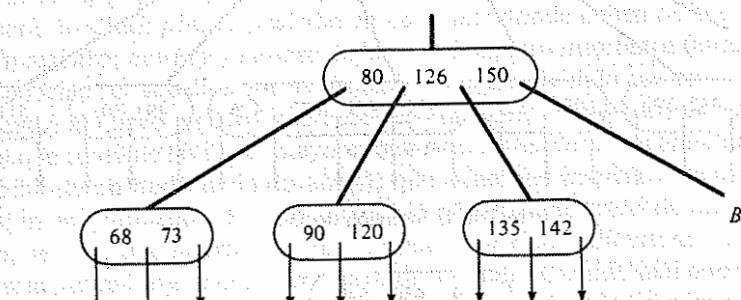


Figura 7.3.6



(a) Eliminación de la llave 113



(b) Eliminación de la llave 120 y consolidación

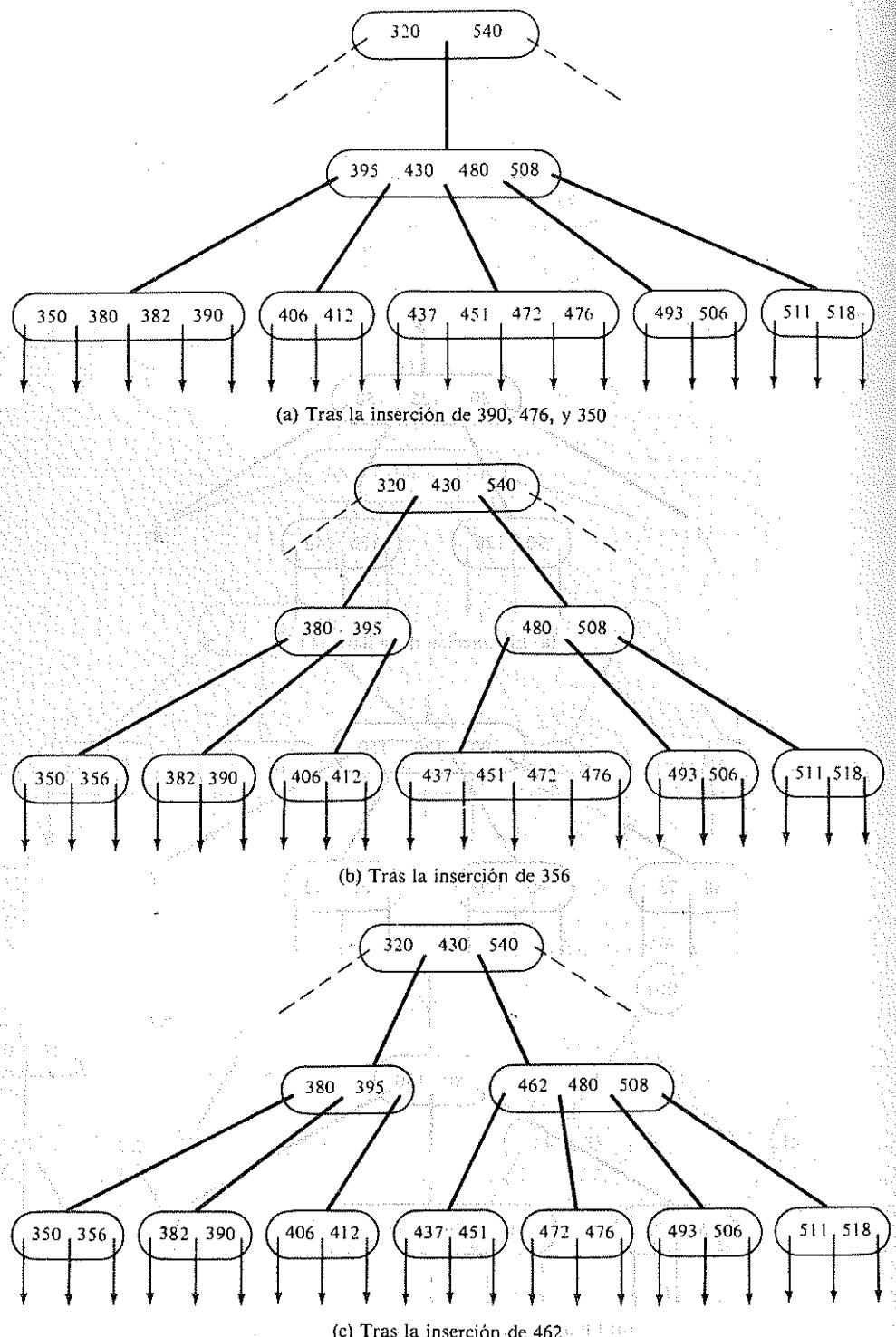


Figura 7.3.8

inserten en su nodo llaves mayores y menores. Esto es distinto a lo que ocurre con un árbol de “arriba abajo” en el cual una llave en un nodo ancestro tiene que ser más antigua que cualquier llave en un nodo descendiente.

Algoritmos para la inserción en un árbol-B

Como podría imaginarse, el algoritmo para la inserción en un árbol-B es bastante complicado. Para simplificar las cosas por el momento, asumiremos que podemos accesar un apuntador al padre de $node(nd)$ haciendo referencia a $father(nd)$ y la posición del apuntador nd en $node(father(nd))$ mediante $index(nd)$ de tal manera que $son(father(nd), index(nd))$ es igual a nd . (Esto se puede implantar en forma más directa adicionando a cada nodo campos $father$ e $index$, pero esta forma de actuar tiene complicaciones que discutimos en breve). Suponemos también que $r(p, i)$ es un apuntador al registro asociado a la llave $k(p, i)$. Recuérdese que la rutina $find$ da como resultado un apuntador a la hoja en la cual la llave debe insertarse y hace la variable *position* igual a la posición en la hoja donde la llave debe ser insertada. Recuérdese también que *key* y *rec* son los argumentos llave y registro a ser insertados.

La función $insert(key, rec, s, position)$ inserta un registro en un árbol-B. Se llama después de una llamada a $find$ si el parámetro de salida *found* de esta rutina es falso (es decir, la llave aún no está en el árbol), donde el parámetro *s* ha sido igualado al apuntador al nodo que regresó $find$ (es decir, el nodo donde deben insertarse *key* y *rec*). La rutina usa dos rutinas auxiliares que serán presentadas en breve. La primera de ellas, *split*, acepta cinco parámetros de entrada: *nd*, un apuntador al nodo que será dividido; *pos*, la posición en $node(nd)$ donde deben ser insertados una llave y un registro; *newkey* y *newrec*, la llave y el registro insertados (esta llave y este registro podrían ser aquellos que avanzan de un nodo dividido con anterioridad o la nueva llave y el nuevo registro insertados en el árbol); y *newnode*, un apuntador al subárbol que contiene las llaves mayores que *newkey* (es decir, un apuntador a la mitad derecha de un nodo dividido antes), que tiene que insertarse dentro del nodo dividido en ese momento. Para mantener el número adecuado de hijos dentro de un nodo, se tiene que insertar un nuevo apuntador hijo cada vez que un nuevo registro y una nueva llave se insertan dentro de un nodo. Cuando una nueva llave y un nuevo registro se insertan en una hoja, el apuntador al hijo se hace igual a *null*. Como una llave y un registro se insertan dentro de uno de los niveles superiores sólo cuando se divide un nodo en un nivel inferior, el apuntador al nuevo hijo que debe ser insertado (*newnode*) apuntará la mitad derecha del nodo que fue dividido en el nivel inferior. La mitad izquierda permanece intacta dentro del nodo del nivel inferior asignado antes. *split* dispone *newkey* y las llaves de $node(nd)$ de manera que el grupo de las $n/2$ llaves más pequeñas permanece en $node(nd)$, la llave del medio y el registro son asignados a los parámetros de salida *midkey* y *midrec*, y las llaves restantes se insertan dentro de un nodo nuevo, *node(nd2)*, donde *nd2* es también un parámetro de salida.

La segunda rutina, *insnode*, inserta una llave *newkey*, el registro *newrec* y el subárbol apuntado por *newnode* dentro de $node(nd)$ en la posición *pos* si hay espacio. Recordar que la rutina *maketree(key, rec)* crea un nodo nuevo que contiene sólo la llave *key*, el registro *rec* y todos los apuntadores iguales a *null*. *maketree* da como resultado un apuntador al nodo recién creado. Presentamos el algoritmo para *insert* usando las rutinas *split*, *insnode* y *maketree*.

INSERT

```

nd = s;
pos = position;
newnode = null; /* apuntador a la mitad derecha del nodo dividido */
newrec = rec; /* el registro que se va a insertar */
newkey = key; /* la llave que se va a insertar */
f = father(nd);
while (f != null && numtree(nd) == n) {
    split(nd, pos, newkey, newrec, newnode, nd2, midkey,
          midrec); /* se divide el nodo en dos partes: nd2 y nd1 */
    newnode = nd2;
    pos = index(nd);
    nd = f;
    f = father(nd);
    newkey = midkey;
    newrec = midrec;
} /* fin de while */
if (numtrees(nd) < n) {
    insnode(nd, pos, newkey, newrec, newnode);
    return;
} /* fin de if */
/* caso 1: f es igual a null y numtrees(nd) es igual a n, así que nd es */
/* una raíz completa; dividirla y crear una nueva raíz */
split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
tree = maketree(midkey, midrec);
son(tree, 0) = nd;
son(tree, 1) = nd2;
igualo y regresa al programa y sigue el algoritmo

```

El corazón de ese algoritmo es la rutina *split* que de hecho divide al nodo. La propia *split* usa una rutina auxiliar *copy(nd1, first, last, nd2)*, que hace a una variable local *numkeys* igual a *last-first + 1* y copia los campos de *k(nd1, first)* hasta *k(nd1, last)* dentro de *k(nd2, 0)* hasta *k(nd2, numkeys)*, los campos *r(nd1, first)* hasta *r(nd1, last)* (que contienen apuntadores a los registros reales) dentro de *r(nd2, 0)* hasta *r(nd2, numkeys)* y los campos *son(nd1, first)* hasta *son(nd1, last + 1)* dentro de *son(nd2, 0)* hasta *son(nd2, numkeys + 1)*. *copy* también hace *numtrees(nd2)* igual a *numtrees(nd) + 1*. Si *last < first*, *copy* hace *numtrees(nd2)* igual a 1 pero no cambia *ninguno* de los campos *k, r* o *son*. *split* también usa *getnode* para crear un nuevo nodo e *insnode* para insertar una nueva llave dentro de un nodo no-lleno.

Lo que sigue es un algoritmo para *split*. Las *n* llaves contenidas en *node(nd)* y *newkey* tienen que distribuirse de manera que las *n/2* menores se queden en *node(nd)*, las *(n - 1)/2* más grandes (que es igual a *n/2* si *n* es impar) sean colocadas en un nuevo nodo, *node(nd2)* y la llave media en *midkey*. Para evitar el volver a computar *n/2* cada vez, suponer que su valor ha sido asignado a la variable global *ndiv2*. El valor de entrada *pos* es la posición en *node(nd)* en la cual *newkey* sería colocada si hubiera espacio para ello.

```

SPILT
/* crear un nuevo nodo para la mitad derecha; mantener la primera */
/* mitad en node(nd) */
nd2 = getnode();

```

```

if (pos > ndiv2) {
    /* newkey pertenece a node(nd2) */
    copy(nd, ndiv2 + 1, n - 2, nd2);
    insnode(nd2, pos - ndiv2 - 1, newkey, newrec, newnode);
    numtrees(nd) = ndiv2 + 1;
    midkey = k(nd, ndiv2);
    midrec = r(nd, ndiv2);
    return;
} /* fin de if */
if (pos == ndiv2) {
    /* newkey es la llave de en medio */
    copy(nd, ndiv2, n - 2, nd2);
    numtrees(nd) = ndiv2 + 1;
    son(nd2, 0) = newnode;
    midkey = newkey;
    midrec = newrec;
    return;
} /* fin de if */
if (pos < ndiv2) {
    /* newkey está en node(nd) */
    copy(nd, ndiv2, n - 2, nd2);
    numtrees(nd) = ndiv2;
    insnode(nd, pos, newkey, newrec, newnode);
    midkey = k(nd, ndiv2 - 1);
    midrec = r(nd, ndiv2 - 1);
    return;
} /* fin de if */

```

La rutina *insnode(nd, pos, newkey, newrec, newnode)* inserta un nuevo registro *newrec* con llave *newkey* dentro de la posición *pos* de un nodo no-lleno, *node(nd)*. *newnode* apunta a un subárbol que debe insertarse a la derecha del nuevo registro. Las claves restantes y subárboles en las posiciones *pos* o mayores se recorren una posición. El valor de *numtrees(nd)* se incrementa en 1. A continuación presentamos un algoritmo para *insnode*:

INSNODE

```

for (i = numtrees(nd) - 1; i >= pos + 1; i--) {
    son(nd, i + 1) = son(nd, i);
    k(nd, i) = k(nd, i - 1);
    r(nd, i) = r(nd, i - 1);
}
/* fin de for */
son(nd, pos + 1) = newnode;
k(nd, pos) = newkey;
r(nd, pos) = newrec;
numtrees(nd) += 1;

```

Cálculo de father e index

Antes de examinar la eficiencia del procedimiento de inserción, tenemos que aclarar un punto pendiente: el relacionado con las funciones *index* y *father*. Puede

haberse notado que, aunque esas funciones se utilizan en el procedimiento *insert*, y hemos sugerido que ellas podrían implantarse de manera más directa agregando los campos *index* y *father* a cada nodo, esos campos no se actualizan por medio del algoritmo de inserción. Examinemos cómo puede alcanzarse esta actualización y por qué elegimos omitir esa operación. Después examinamos métodos alternativos de implantación de las dos funciones que no requieren de la actualización.

Los campos *father* e *index* tendrían que ser actualizados cada vez que fuesen llamadas *copy* o *insnode*. En el caso de *copy*, tienen que ser modificados ambos campos en cada hijo cuyo apuntador se copia. En el caso de *insnode*, tiene que ser modificado el campo *index* de cada hijo cuyo apuntador se mueve, tanto como ambos campos en el hijo insertado. (Además, los campos tienen que actualizarse en las dos mitades de un nodo raíz dividido en la rutina *insert*). Sin embargo, esto impactaría la eficiencia del algoritmo de inserción de una manera inaceptable en especial cuando tratemos con nodos en la memoria externa. En todo el proceso de búsqueda e inserción en un árbol-B (excluyendo la actualización de los campos *father* e *index*), se acceden a lo sumo dos nodos en cada nivel del árbol. En la mayoría de los casos, cuando no ocurre una división en un nivel, se accesa sólo un nodo en el mismo. Las operaciones *copy* e *insnode*, no requieren en realidad el acceso a los nodos hijo movidos, aunque mueven nodos de un subárbol a otro, ya que lo hacen moviendo apuntadores dentro de uno o dos nodos padre. El requerimiento de una actualización de los campos *father* e *index* en aquellos hijos requeriría del acceso y modificación de todos los propios nodos hijo. Pero la lectura y escritura de un nodo de y en la memoria externa son las operaciones más costosas en todo el proceso de manejo de un árbol-B. Cuando se considera que, en un sistema práctico de almacenamiento de información, un nodo puede tener varios cientos de hijos, se hace claro que guardar los campos *index* y *father* daría por resultado que se reduzca en un ciento la eficiencia del sistema.

Entonces, ¿cómo podemos obtener los datos *father* e *index* requeridos por el proceso de inserción, sin guardar campos separados? Primero recuérdese que la función *nodedsearch(p, key)* da como resultado la posición de la llave menor en *node(p)* que sea mayor o igual que *key*, de modo que *index(nd)* es igual a *nodedsearch(father(nd), key)*. En consecuencia una vez que está disponible *father*, puede obtenerse *index* sin un campo aparte.

Para entender cómo podemos obtener *father*, veamos un problema relacionado. Ninguna inserción en un árbol-B puede ocurrir sin búsqueda previa para localizar la hoja donde tiene que insertarse la nueva llave. Esta búsqueda procede a partir de la raíz y accesa un nodo en cada nivel hasta alcanzar la hoja apropiada. Es decir, procede a través de un solo camino, de la raíz a una hoja. La inserción retrocede después a lo largo del camino, dividiendo todos los nodos llenos en el camino de la hoja a la raíz, hasta alcanzar un nodo no-lleno en el que puede insertarse una llave sin una división. Una vez que se ejecuta esta inserción, el proceso termina.

El proceso de inserción accesa los mismos nodos que el de búsqueda. Como habíamos visto el acceso de un nodo de la memoria externa es bastante costoso, tendría sentido que el proceso de búsqueda guardara los nodos del camino que recorre junto con sus direcciones externas en la memoria interna, donde el proceso de inserción los puede accesar sin una segunda operación de lectura costosa. Pero

una vez que todos los nodos de un camino estén almacenados en la memoria interna, el padre de un nodo puede ser localizado mediante un simple examen del nodo previo en el camino. Así, no hay necesidad de guardar y actualizar un campo *father*.

Presentaremos, por lo tanto, versiones modificadas de *find* e *insert* para localizar e insertar una llave en un árbol-B. Sea *pathnode(i)* una copia de *i*-ésimo nodo en el camino de la raíz a una hoja, *location(i)* su posición (o bien un apuntador si el árbol está en la memoria interna o bien una dirección de memoria externa si está en la memoria externa) e *index(i)* la posición del nodo entre los hijos de su padre (obsérvese que *index* puede ser determinado durante el proceso de búsqueda y retenido para su uso durante el proceso de inserción). Nos referimos a *son(i, j)*, *k(i, j)*, y *r(i, j)* como el *j*-ésimo campo hijo, llave y registro en *pathnode(i)* respectivamente. De manera similar, *numtrees(i)* es el campo *numtrees* en *pathnode(i)*.

El siguiente algoritmo para *find* utiliza la operación *access(i, loc)* para copiar un nodo de la posición *loc* (ya sea de la memoria interna o externa) dentro de *pathnode(i)* y el propio *loc* dentro de *location(i)*. Si el árbol está almacenado en la memoria interna esta operación se realiza por medio de

```
pathnode(i) = node(loc);  
location(i) = loc;
```

Si el árbol está almacenado en la memoria externa, la operación se efectúa con

```
directread(loc, pathnode(i));  
location(i) = loc;
```

donde *directread* lee un bloque de memoria de una dirección externa particular (*loc*) a un buffer de la memoria interna (*pathnode(i)*). También suponemos que *nodedsearch(i, key)* busca *pathnode(i)* en lugar de *node(i)*. Presentamos el algoritmo *find*:

```
q = null;  
p = tree;  
j = -1;  
i = -1;  
while (p != null) {  
    index(++j) = i;  
    access(j, p);  
    i = nodedsearch(j, key);  
    q = p;  
    if (i < numtrees(j) - 1 && key == k(j, i))  
        break;  
    p = son(j, i);  
} /* fin de while */  
position = i;  
return(j); /* la llave está en pathnode(j) o ahí pertenece */
```

El proceso de inserción está modificado en varios lugares. Primero, *insnode* y *copy* accesan *pathnode(nd)* en lugar de *node(nd)*. Es decir, *nd* es ahora el índice de

un arreglo en lugar de un apuntador, de manera que todas las referencias a *k*, *son*, *p* y *numtrees* significan referencias a campos dentro de un elemento de *pathnode*. Los algoritmos para *insnode* y *copy* no tienen que cambiarse.

Segundo, *split* debe ser modificada para producir las dos mitades de un nodo dividido. Esto supone una rutina *replace(i)*, que remplaza el nodo en *location(i)* con los contenidos de *pathnode(i)*. Esta rutina es la inversa de *access*. Si el árbol está guardado en la memoria interna, puede ser implantado mediante

```
node(location(i)) = pathnode(i);
```

y si está almacenado en la memoria externa mediante:

```
directwrite(location(i), pathnode(i));
```

donde *directwrite* escribe un buffer en memoria (*pathnode(i)*) hacia un bloque de memoria externa en una dirección externa particular (*location(i)*). *split* usa también una función *makenode(i)* que obtiene un nuevo bloque de memoria en la ubicación *x*, coloca *pathnode(i)* en ese bloque y da como resultado *x*. Lo que sigue es una versión revisada de *split*:

```
if (pos > ndiv2) {
    copy(nd, ndiv2 + 1, n - 2, nd + 1);
    insnode(nd + 1, pos - ndiv2 - 1, newkey, newrec, newnode);
    numtrees(nd) = ndiv2 + 1;
    midkey = k(nd, ndiv2);
    midrec = r(nd, ndiv2);
    return;
} /* fin de if */
if (pos == ndiv2) {
    copy(nd, ndiv2, n - 2, nd + 1);
    numtrees(nd) = ndiv2;
    son(nd + 1, 0) = newnode;
    midkey = newkey;
    midrec = newrec;
    return;
} /* fin de if */
if (pos < ndiv2) {
    copy(nd, ndiv2, n - 2, nd + 1);
    numtrees(nd) = ndiv2;
    insnode(nd, pos, newkey, newrec, newnode);
    midkey = newkey;
    midrec = newrec;
} /* fin de if */
replace(nd);
nd2 = makenode(nd + 1);
```

Obsérvese que *nd* es ahora una posición en *pathnode* en lugar de un apuntador a un nodo y que *pathnode(nd + 1)* se usa para construir la segunda mitad del nodo

dividido en lugar de *node(nd2)*. Esto se puede hacer dado que el nodo en el nivel *nd + 1* (si lo hay) del camino ya ha sido actualizado como se llama a *split* para *nd*, de manera que *pathnode(nd + 1)* puede ser usado otra vez. *nd2* se mantiene como la ubicación actual del nuevo nodo (asignado por *makenode*). (Deberíamos observar que puede ser deseable reservar un camino a la llave recién insertada en *pathnode* si, por ejemplo, queremos ejecutar un recorrido secuencial o inserciones secuenciales comenzando en ese punto. En ese caso, el algoritmo tiene que ajustarse de manera adecuada para colocar la mitad izquierda o derecha apropiada del nodo dividido en la posición indicada de *pathnode*. Tampoco podemos usar *pathnode(i + 1)* para construir la mitad derecha pero, en su lugar, debemos usar un nodo adicional de la memoria interna. Dejamos los detalles al lector.)

También se modifica la rutina *insert* al usar *nd - 1* en lugar de *father(nd)*. También llama a *replace* y *makenode*. Cuando se tiene que dividir la raíz, *maketree* construye un nuevo nodo raíz en la memoria interna. Este nodo se coloca en *pathnode(i)* (que ya no se necesita, dado que el antiguo nodo raíz ha sido actualizado por *split*) y extrae la escritura usando *makenode*. A continuación presentamos el algoritmo revisado para *insert*:

```
nd = s;
pos = position;
newnode = null;
newrec = rec;
newkey = key;
while (nd != 0 && numtrees(nd) == n) {
    split(nd, pos, newkey, newrec, newnode, nd2, midkey,
          midrec);
    newnode = nd2;
    pos = index(nd);
    nd--;
    newkey = midkey;
    newrec = midrec;
}
/* fin de while */
if (numtrees(nd) < n) {
    insnode(nd, pos, newkey, newrec, newnode);
    replace(nd);
    return;
} /* fin de if */
split(nd, pos, newkey, newrec, newnode, nd2, midkey, midrec);
pathnode(0) = maketree(midkey, midrec);
son(0, 0) = nd;
son(0, 1) = nd2;
tree = makenode(0);
```

Eliminación en árboles de búsqueda multivías

El método más simple para eliminar un registro de un árbol de búsqueda multivías es conservar la llave en el árbol pero marcada de alguna manera que indi-

que que representa un registro eliminado. Esto podría realizarse haciendo el apuntador al registro correspondiente igual a la llave *null* o asignando un campo indicador extra a cada llave para indicar si ha sido borrada o no. Por supuesto, el espacio ocupado por el propio registro se puede usar otra vez. De esta manera, la llave queda en el árbol como guía hacia los subárboles pero no representa un registro del archivo.

La desventaja de este enfoque es que el espacio ocupado por la llave en sí se gasta, conduciendo hacia nodos innecesarios cuando ya se han eliminado un gran número de registros. Bits "eliminados" extra, requieren todavía de más espacio.

Por supuesto, si un registro con una llave eliminada se inserta después, el espacio para la llave puede reciclarse. En un nodo no-hoja, sólo la misma llave podría reutilizar el espacio, dado que es difícil determinar de manera dinámica que la llave recién insertada está entre el antecesor y el sucesor de la llave eliminada. Sin embargo, en un nodo hoja (o, en ciertas situaciones, en una semihoya) el espacio de la llave eliminada se puede reusar por una llave vecina, dado que es muy fácil determinar la proximidad. Como una gran parte de las llaves están en hojas o semihoyas, si las inserciones y eliminaciones ocurren con la misma frecuencia (o si hay más inserciones que eliminaciones) y están distribuidas de manera uniforme (es decir, las eliminaciones no están agrupadas para reducir de manera significativa el número total de llaves en el árbol de manera temporal) se puede perder espacio a cambio de la ventaja que significa la facilidad de eliminación. Hay también un pequeño gasto de tiempo en búsquedas posteriores, dado que algunas llaves requerirán que más nodos sean examinados si es que la llave eliminada no ha sido nunca insertada en el primer lugar.

Si no queremos pagar este costo en cuanto a tiempo de búsqueda y espacio de la eliminación simplificada, hay técnicas más costosas de eliminación que lo evitan. En un árbol de búsqueda no restringido, de multivías, se puede emplear una técnica similar a la de eliminación en un árbol de búsqueda binaria:

1. Si la llave a ser eliminada tiene un subárbol izquierdo o derecho vacío, eliminar sólo la llave y compactar el nodo. Si era la única llave en dicho nodo, liberarlo.
2. Si la llave a ser eliminada tiene subárboles izquierdo y derecho no-vacíos, encontrar la llave que le sucede (que debe tener un subárbol izquierdo vacío); dejar que dicha llave tome su lugar y compactar el nodo que contenía al sucesor. Si el sucesor era la única llave del nodo, liberarlo.

Dejamos el desarrollo del algoritmo y del programa detallado al lector.

Sin embargo, este procedimiento puede dar como resultado un árbol que no satisface los requerimientos para ser de "arriba abajo" un árbol-B, aun cuando el árbol original lo fuese. En un árbol de "arriba abajo", si la llave que está siendo eliminada es de una semihoya que no es una hoja y la llave tiene subárboles izquierdo y derecho vacíos, la semihoya quedará con menos de $n - 1$ llaves, aun cuando no sea una hoja. Esto viola el requerimiento de un árbol de "arriba abajo". En ese caso, es necesario elegir un subárbol no-vacio aleatorio del nodo y mover la llave menor o la mayor de ese subárbol a la semihoya de la cual fue eliminada la llave. Este proceso tiene que repetirse hasta que la semihoya de la cual se toma una llave sea una hoja.

Esta hoja se puede entonces compactar o liberar. En el peor caso, esto requeriría reescribir un nodo en cada nivel del árbol.

En un árbol-B estricto, tenemos que preservar el requerimiento de que cada nodo contiene por lo menos $n/2$ llaves. Como ya observamos, si se está eliminando una llave de un nodo no-hoja, su sucesor (que tiene que estar en una hoja) se mueve a la posición eliminada y el proceso de eliminación prosigue como si el sucesor fuese eliminado del nodo hoja. Cuando una llave (ya sea la que debe ser eliminada o su sucesor) se elimina de un nodo hoja y el número de llaves en el nodo disminuye por debajo de $n/2$, hay que hacer algo para remediar esta situación. Esta situación se llama *subdesborde*. Cuando ocurre, la solución más simple es examinar el hermano mayor o menor de la hoja. Si el hermano contiene más de $n/2$ llaves, se puede adicionar la llave *ks* en el nodo padre que separa a los dos hermanos al nodo con subdesborde y la última o primera llave del hermano (la última si el hermano es mayor y la primera si es menor) puede agregarse al padre en el lugar de *ks*. La figura 7.3.9a ilustra este proceso en un árbol-B de orden-5. (Deberíamos observar que una vez que el hermano no está siendo accesado, podemos distribuir las llaves de manera uniforme entre los dos hermanos en lugar de recorrer sólo una llave. Por ejemplo, si el nodo con subdesborde, *n1* contiene 106 y 112, la llave que separa en el padre *f* es 120 y el hermano *n2* contiene 123, 128, 134, 139, 142 y 146 en un árbol-B de orden-7, podemos redistribuirlas de manera que *n1* contenga 106, 112, 120 y 123, 128 se mueve a *f* como separador y 134, 139, 142 y 146 se quedan en *n2*).

Si ambos hermanos contienen exactamente $n/2$ llaves, ninguna llave tiene que recorrerse. En ese caso el nodo con subdesborde y uno de sus hermanos se *concatenan* o *consolidan* dentro de un solo nodo que también contiene la llave separadora de su padre. Esto se ilustra en la figura 7.3.9b, donde combinamos el nodo con subdesborde con su hermano menor.

Por supuesto, es posible que el padre sólo contenga $n/2$ llaves, de manera que él tampoco tenga una llave extra qué ofrecer. En ese caso, la puede tomar de su padre y hermano como en la figura 7.3.10a. En el caso peor, cuando los hermanos del padre tampoco tienen llaves qué ofrecer, el padre y su hermano pueden también consolidarse y la llave puede ser tomada del abuelo. Esto se ilustra en la figura 7.3.10b. Potencialmente, si todos los ancestros no-raíz de un nodo y sus hermanos contienen exactamente $n/2$ llaves, se tomará una llave de la raíz, ya que las consolidaciones ocurren en cada nivel desde las hojas hasta el nivel justo debajo de la raíz. Si la raíz tenía más de una clave, el proceso termina con esto, dado que la raíz de un árbol-B no necesita tener más de una llave. Si, por otro lado, la raíz contenía una sola llave, esa llave se usa en la consolidación de los dos nodos debajo de la raíz, se libera la raíz, el nodo consolidado se convierte en la nueva raíz del árbol y se reduce la profundidad del árbol-B. Dejamos al lector el desarrollo de un algoritmo real para la eliminación en un árbol-B según esta descripción.

Se debería observar sin embargo, que es tonto formar un nodo consolidado con $n - 1$ llaves si una inserción posterior dividirá de inmediato el nodo en dos. En un árbol-B de orden grande, puede tener sentido dejar un nodo con subdesborde con menos de $n/2$ llaves (aun cuando esto viole los requerimientos formales de un árbol-B) de manera que puedan realizarse inserciones futuras sin división. De manera típica,

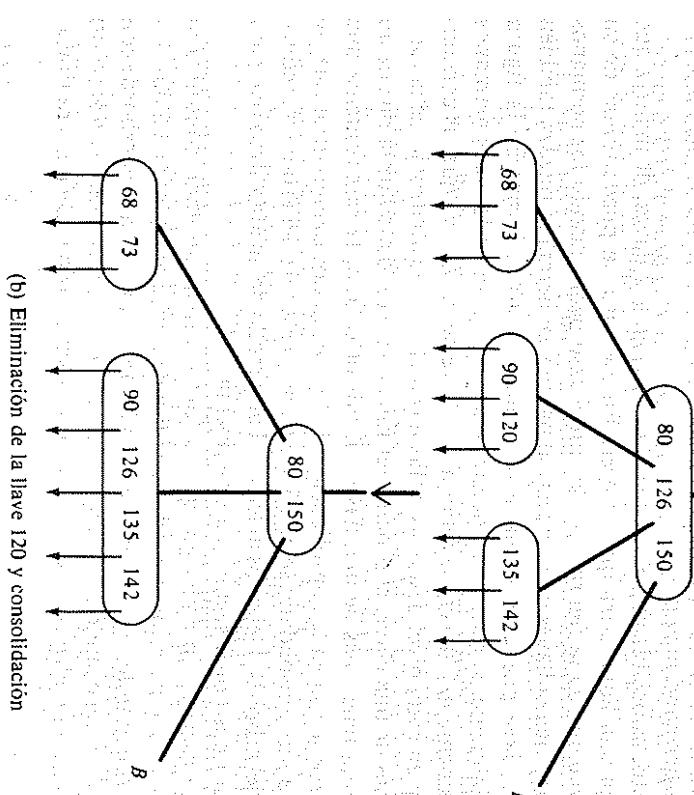
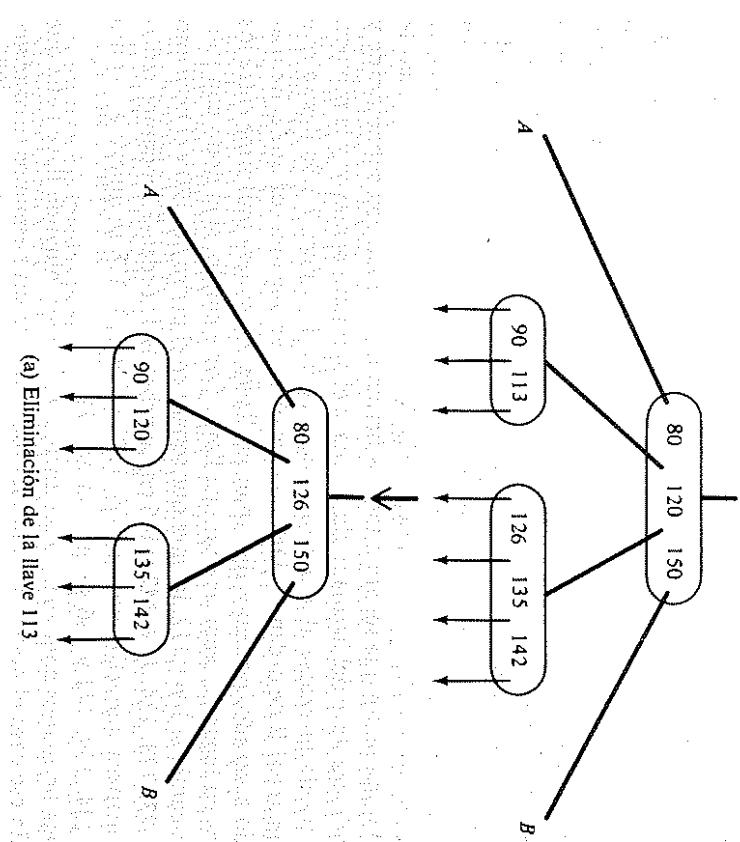
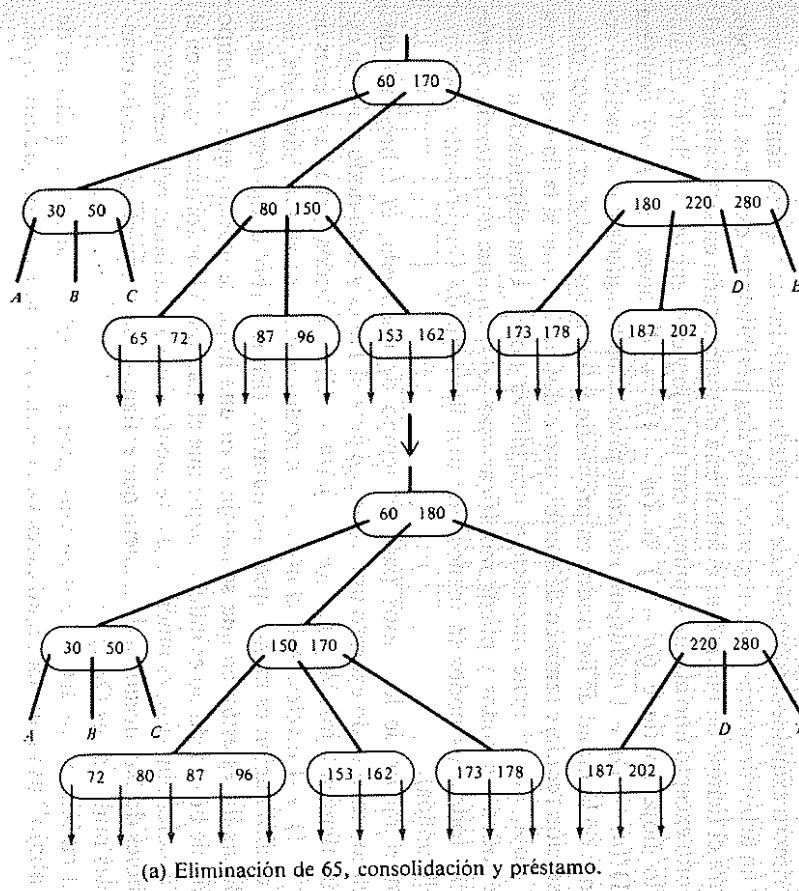


Figura 7.3.9



(a) Eliminación de la llave 113



(a) Eliminación de 65, consolidación y préstamo.

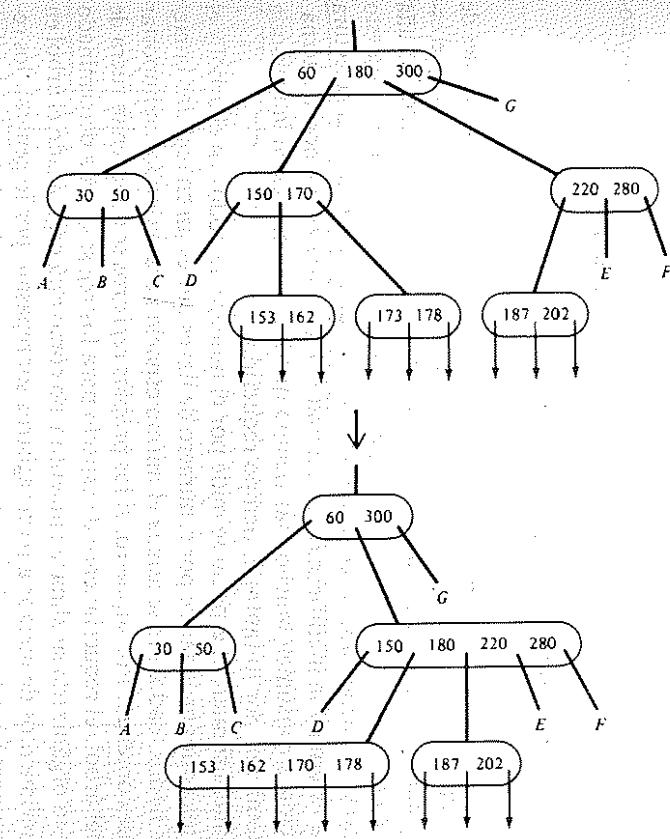


Figura 7.3.10

un número mínimo (m_{\min} menor que $n/2$) de llaves se define de manera que la consolidación ocurra sólo si restan menos que m_{\min} llaves en un nodo hoja.

Eficiencia de árboles de búsqueda multivías

Las consideraciones primarias en la evaluación de la eficiencia de árboles de búsqueda multivías, como para todas las estructuras de datos, son el tiempo y el espacio. El tiempo se mide por el número de nodos accesados o modificados en una operación, antes que por el número de comparaciones de llaves. La razón para esto es que, como ya se mencionó, el acceso a un nodo involucra por lo general la lectura en memoria externa y la modificación de un nodo la escritura en memoria externa. Esas operaciones consumen mucho más tiempo que las operaciones en la memoria interna y dominan, por lo tanto, el tiempo requerido.

De manera similar, el espacio se mide por el número de nodos en el árbol y el tamaño de los nodos en lugar de por el número de llaves que están en realidad contenidas en los nodos, dado que se asigna el mismo espacio para un nodo sin importar las llaves que contenga. Por supuesto, si los propios registros se almacenan fuera de los nodos del árbol, el requerimiento de espacio para los registros está determinado por cómo está organizado su almacenamiento y no cómo está organizado el árbol en sí. Los requerimientos de memoria para registros sobrepasan, por lo general, los requerimientos para el árbol de llaves, de manera que el espacio real para el árbol puede no ser significativo.

Primero examinemos árboles de búsqueda multivías de “arriba abajo”. Suponiendo un árbol de orden- m y n registros, hay dos posibilidades extremas. En el peor caso para el tiempo de búsqueda, el árbol está desbalanceado en su totalidad. Cualquier nodo excepto uno es una semihoya con un hijo y contiene $m - 1$ llaves. El único nodo hoja contiene $((n - 1) \% (m - 1)) + 1$ llaves. El árbol contiene $((n - 1)/(m - 1)) + 1$ nodos, uno en cada nivel. Una búsqueda o una inserción acceden más de la mitad de los nodos en promedio y todos los nodos en el peor caso. Una inserción requiere también de escribir uno o dos nodos (uno si la llave se inserta en una hoja, dos si tiene que crearse una nueva hoja.) Una eliminación accesa siempre todos los nodos y puede modificar uno sólo, aunque en potencia, puede modificar todos los nodos (a menos que una llave pueda simplemente marcarse como eliminada).

En el mejor de los casos para el tiempo de búsqueda, el árbol está casi balanceado, cada nodo excepto uno contiene $m - 1$ llaves y cada nodo no-hoja excepto uno tiene m hijos. Hay aún $((n - 1)/(m - 1)) + 1$ nodos, pero hay menos de $\log_m(n - 1) + 1$ niveles. Así, el número de nodos accesados en una búsqueda, inserción o eliminación es menor que este número. (En un árbol como ese, más de la mitad de las llaves están en una semihoya o en una hoja, de manera que el tiempo promedio de búsqueda no es mucho mejor que el máximo.) Por fortuna, como en el caso de árboles binarios, es mucho más frecuente el caso de árboles balanceados que el de árboles desbalanceados, de manera que el tiempo de búsqueda promedio usando árboles multivías es $O(\log n)$.

Sin embargo, un árbol de multivías general e incluso un árbol multivías de “arriba abajo” usan una cantidad excesiva de memoria. Para ver por qué esto es así,

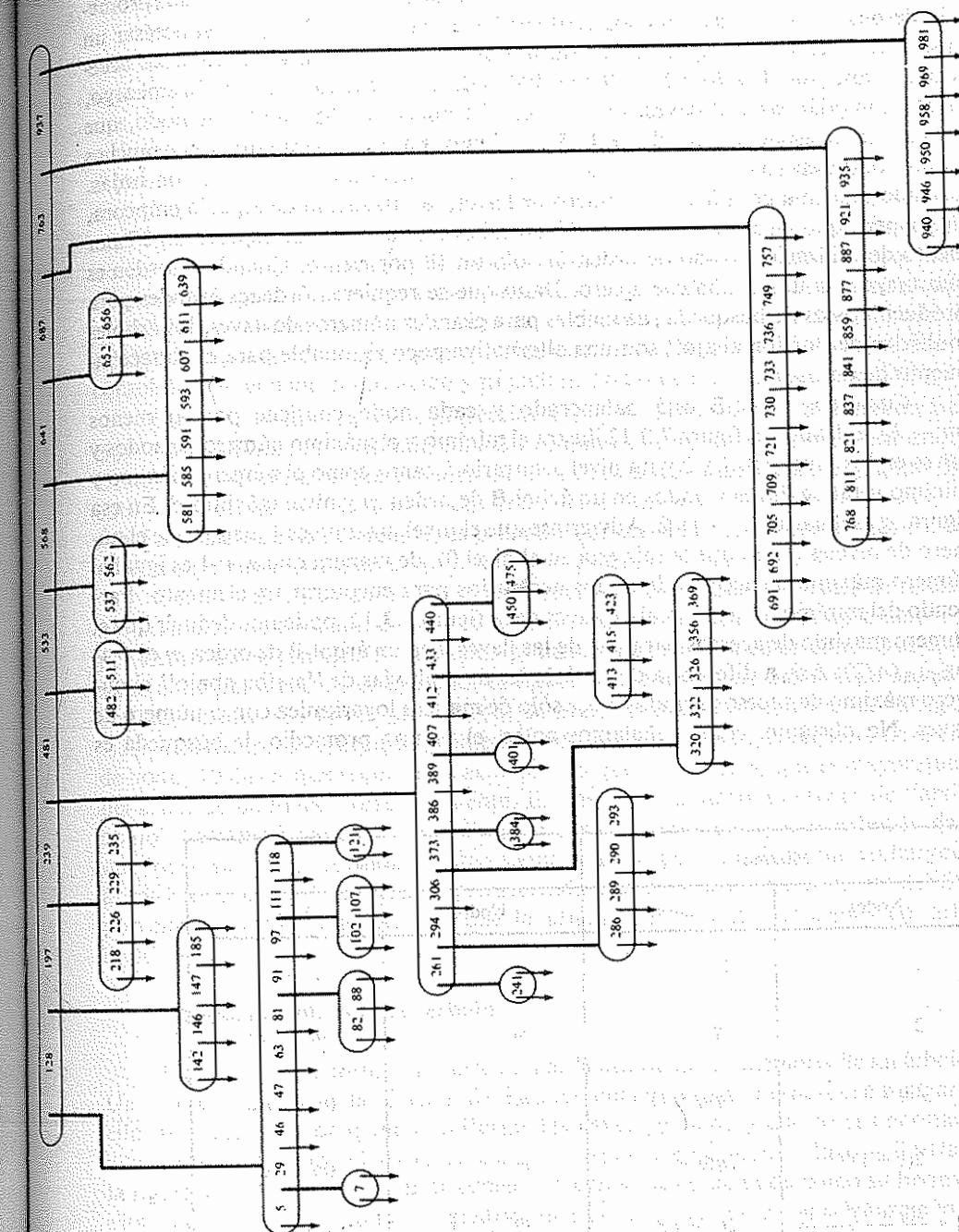


Figura 7.3.11

ver la figura 7.3.11, que ilustra un árbol de “arriba abajo” de búsqueda multivías típico de orden-11 con 100 llaves. El árbol es bastante balanceado y el costo promedio de búsqueda es más o menos 2.10 [10 llaves en el nivel 0 requieren accesar un nodo, 61 en el nivel 1 requieren accesar dos nodos, y 29 en el nivel 2 requieren accesar 3 nodos: $(10 * 1 + 61 * 2 + 29 * 3)/100 = 2.19$], que es razonable. Sin embargo, para acomodar las 100 llaves, el árbol usa 23 nodos o 4.35 llaves por nodo, que representa un espacio utilizado de 43.5 por ciento. La razón para esto es que muchas hojas contienen sólo una o dos llaves y la gran mayoría de los nodos son hojas. Cuando aumenta el orden y el número de llaves, la utilización de espacio empeora, de manera que un árbol de orden-11 con miles de llaves puede esperar un 27 por ciento de utilización y uno de orden-21 sólo un 17 por ciento. Cuando el orden es aún mayor, la utilización cae a cero. Dado que se requieren órdenes grandes para producir costos de búsqueda razonables para grandes números de llaves, los árboles multivías de “arriba abajo” son una alternativa poco razonable para el almacenamiento de datos.

Cualquier árbol-B está balanceado y cada nodo contiene por lo menos $(m - 1)/2$ llaves. La figura 7.3.12 ilustra el mínimo y el máximo número de nodos y llaves en los niveles 0, 1 y 2 y un nivel arbitrario i , tanto como el número máximo y mínimo total de llaves y nodos en un árbol-B de orden m y nivel máximo d . En esa figura, q es igual a $(m - 1)/2$. Adviértase que el nivel máximo es 1 menos que el número de niveles (dado que la raíz está en el nivel 0), de manera que $d + 1$ es igual al número máximo de accesos de nodos necesarios para encontrar un elemento. Partiendo del mínimo número total de llaves de la figura 7.3.12, podemos deducir que el número máximo de accesos para una de las llaves n en un árbol-B de orden m es $1 + \log_{q+1}(n/2)$. Así, a diferencia de los árboles de multivías de “arriba abajo” el número máximo de accesos a nodos crece sólo de manera logarítmica con el número de llaves. No obstante, como señalamos antes, el tiempo promedio de búsqueda es

Nivel	Mínimo		Máximo	
	Nodos	Llaves	Nodos	Llaves
0	1	1	1	$m - 1$
1	2	$2q$	m	$(m - 1)m$
2	$2(q + 1)$	$2q(q + 1)$	m^2	$(m - 1)m^2$
I	$2(q + 1)^{i-1}$	$2q(q + 1)^{i-1}$	m^i	$(m - 1)m^i$
Total	$1 + \frac{2(q + 1)^d - 1}{q}$	$2(q + 1)^d$	$\frac{m^{d+1} - 1}{m - 1}$	$m^{d+1} - 1$

Figura 7.3.12

vo entre árboles multivías de “arriba abajo” árboles-B, dado que los árboles de “arriba abajo” son por lo general bastante balanceados.

La inserción en un árbol-B requiere la lectura de un nodo por nivel y la escritura de un nodo como mínimo por nivel más dos nodos para toda división que ocurra. Si ocurren s divisiones, se escriben $2s + 1$ nodos (dos mitades para cada división más el padre del último nodo dividido). La eliminación requiere de la lectura de un nodo por nivel para encontrar una hoja, escribiendo un nodo si la llave eliminada está en una hoja y la eliminación no causa subdesborde y dos nodos si la llave *eliminada* está en un nodo no-hoja y la eliminación de la llave que remplaza supresión no causa que la hoja tenga un subdesborde. Si ocurre subdesborde, se requieren una lectura adicional (del hermano de cada nodo con subdesborde) por desborde, una escritura adicional para cada consolidación excepto la última y tres escrituras adicionales para el subdesborde final si no es necesaria una consolidación (el nodo de subdesborde, su hermano y su padre) o dos escrituras adicionales si es necesaria una consolidación (el nodo consolidado y su padre). Todas esas operaciones son $O(\log n)$.

Como en el caso de un heap (sección 6.3) y un árbol binario balanceado (sección 7.2), la inserción y eliminación del mínimo o máximo elementos son ambas $O(\log n)$ en un árbol-B; por lo tanto, la estructura puede usarse para implantar una cola de prioridad (ascendente o descendente) de manera eficiente. De hecho, un árbol 3-2 (un árbol-B de orden-3) es probable que sea el método práctico más eficiente para implantar una cola de prioridad en la memoria interna.

Como cada nodo en un árbol-B (excepto la raíz) tiene que estar, de manera aproximada, medio lleno, el peor caso de utilización de memoria se aproxima a 50 por ciento. En la práctica la utilización promedio de memoria en un árbol-B se aproxima al 69 por ciento. Por ejemplo, la figura 7.3.13 ilustra un árbol-B de orden-11 con las mismas 100 llaves que el árbol de acceso múltiple o multivías de la figura 7.3.11. El tiempo de búsqueda promedio es de 2.88 (una llave que requiere un acceso de nodo, 10 llaves que requieren 2 accesos y 89 que requieren 3) que es mayor que el del árbol de multivías correspondiente. En realidad, un árbol multivías de “arriba abajo” bastante balanceado tendrá menor costo de búsqueda que un árbol-B, dado que todos sus nodos superiores están siempre llenos por completo. Sin embargo, el árbol-B contiene sólo 15 nodos, produciendo una utilización de memoria del 66.7 por ciento, mucho más elevada que la utilización de 43.5 por ciento del árbol multivías.

Perfeccionamiento del árbol-B

Existen muchas formas de perfeccionar la utilización de memoria de un árbol-B. Un método es retrasar la división de un nodo cuando ocurre desborde. En lugar de ello, se distribuyen de manera uniforme las llaves del nodo y uno de sus hermanos adyacentes, así como la llave en el padre que separa los dos nodos. Esto se ilustra en la figura 7.3.14 en un árbol-B de orden-7. Cuando tanto un nodo como su hermano están llenos, los dos nodos se dividen en tres. Esto garantiza una utilización de memoria mínima de casi 67 por ciento, aunque la utilización de memoria es, en realidad, mayor en la práctica. Tal árbol se llama árbol-B*. En efecto, esta técnica se puede extender aún más redistribuyendo las llaves entre todos los hermanos y el

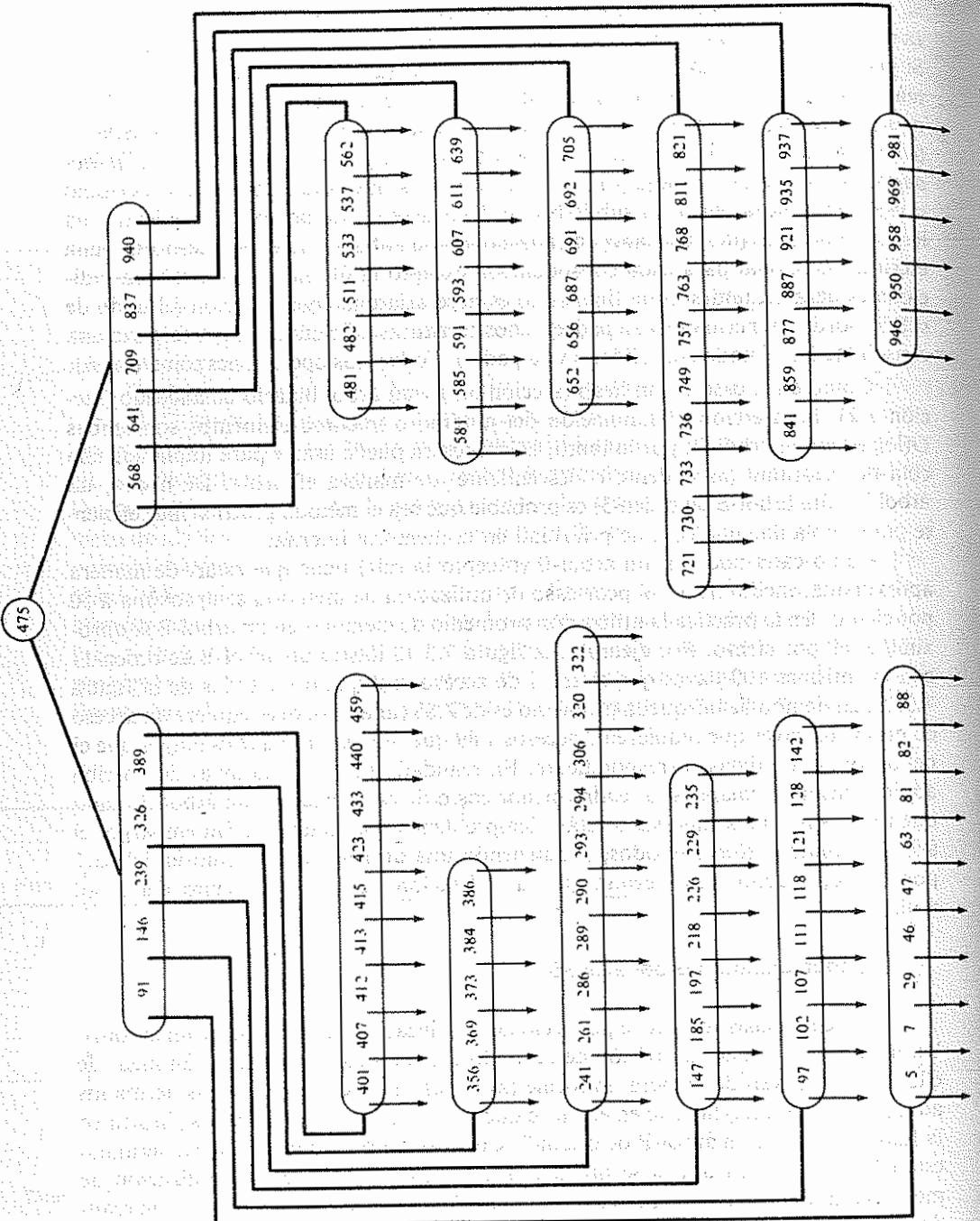


Figura 7.3.13

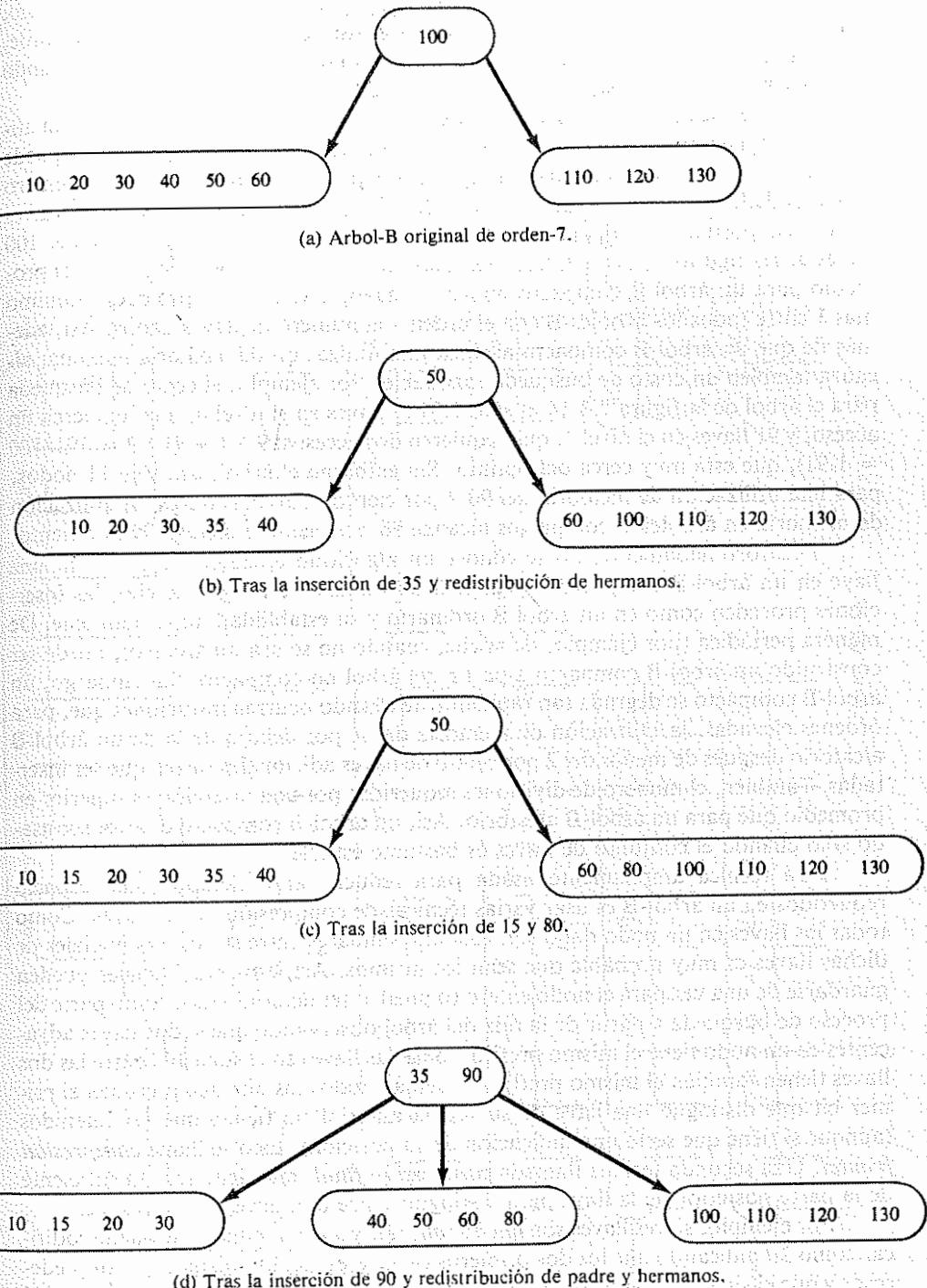


Figura 7.3.14 Arbol-B de orden-7 con redistribución de nodos múltiples

padre de un nodo lleno. Desafortunadamente, este método exige su propio precio, dado que requiere de accesos adicionales muy costosos en inserciones con desborde, mientras que la utilización de espacio adicional marginal alcanzada, considerando cada hermano extra, se vuelve cada vez más pequeña.

Otra técnica es usar un *árbol-B compacto*. Un árbol-B de ese tipo tiene una utilización de memoria máxima para un orden y un número de llaves dados. Se puede mostrar que esta utilización máxima de memoria para un árbol-B de orden y número de llaves dados se logra cuando los nodos del fondo del árbol contienen tantas llaves como sea posible. La figura 7.3.15 ilustra un árbol-B compacto para las 100 llaves de las figuras 7.3.11 y 7.3.13. Se puede mostrar que el costo de búsqueda promedio para un árbol-B compacto nunca es mayor que el costo promedio mínimo más 1 entre todos los árboles-B con el orden y el número de llaves dados. Así, además de que un árbol-B compacto alcanza una utilización de memoria máxima, alcanza también un costo de búsqueda razonable. Por ejemplo, el costo de búsqueda para el árbol de la figura 7.3.14 es sólo 1.91 (9 llaves en el nivel 0, que requieren un acceso, y 91 llaves en el nivel 1, que requieren dos accesos: $9 * 1 + 91 * 2 = 191/100 = 1.91$), que está muy cerca del óptimo. Sin embargo el árbol, usa sólo 11 nodos, para una utilización de memoria del 90.9 por ciento. Con más llaves, la utilización de memoria en árboles-B compactos alcanza 98 por ciento e incluso 99 por ciento.

Desafortunadamente, no se conoce un algoritmo eficiente para insertar una llave en un árbol-B compacto y mantener su tamaño. En lugar de ello, las inserciones proceden como en un árbol-B ordinario y su estabilidad no se mantiene. De manera periódica (por ejemplo, de noche, cuando no se usa un archivo), puede ser construido un árbol-B compacto a partir del árbol no-compacto. Sin embargo, un árbol-B compacto se degrada tan rápidamente cuando ocurren inserciones que, para órdenes elevadas, la utilización de memoria decae por debajo de la de un árbol-B aleatorio después de menos del 2 por ciento de llaves adicionales tienen que ser insertadas. También, el número de divisiones requeridas por una inserción es superior en promedio que para un árbol-B aleatorio. Así, un árbol-B compacto debería ser usado sólo cuando el conjunto de llaves es bastante estable.

Una técnica ampliamente usada para reducir tanto tiempo como espacio requeridos en un árbol-B es usar varias técnicas de compresión de las llaves. Como todas las llaves en un nodo dado son bastante similares entre sí, los bits iniciales de dichas llaves es muy probable que sean los mismos. Así, esos bits iniciales pueden guardarse de una vez para el nodo entero (o pueden ser determinados como parte del proceso de búsqueda a partir de la raíz del árbol observando que si dos llaves adyacentes en un nodo tienen el mismo prefijo, todas las llaves en el subárbol entre las dos llaves tienen también el mismo prefijo). Además, todos los bits que preceden al primer bit que distingue una llave de su vecina anterior no tienen que ser retenidos (aunque sí tiene que serlo una indicación de su posición). Esto se llama *compresión frontal*. Una segunda técnica llamada *compresión final*, mantiene sólo lo suficiente de la parte posterior de la llave para distinguir entre una llave y su sucesora.

Por ejemplo, si tres llaves son *anchor andrew* y *antoin*, *andrew* se puede codificar como *2d* indicando que los dos primeros caracteres son los mismos que su predecesor y que el siguiente carácter, *d*, distingue la llave de su predecesora y sucesora. Si

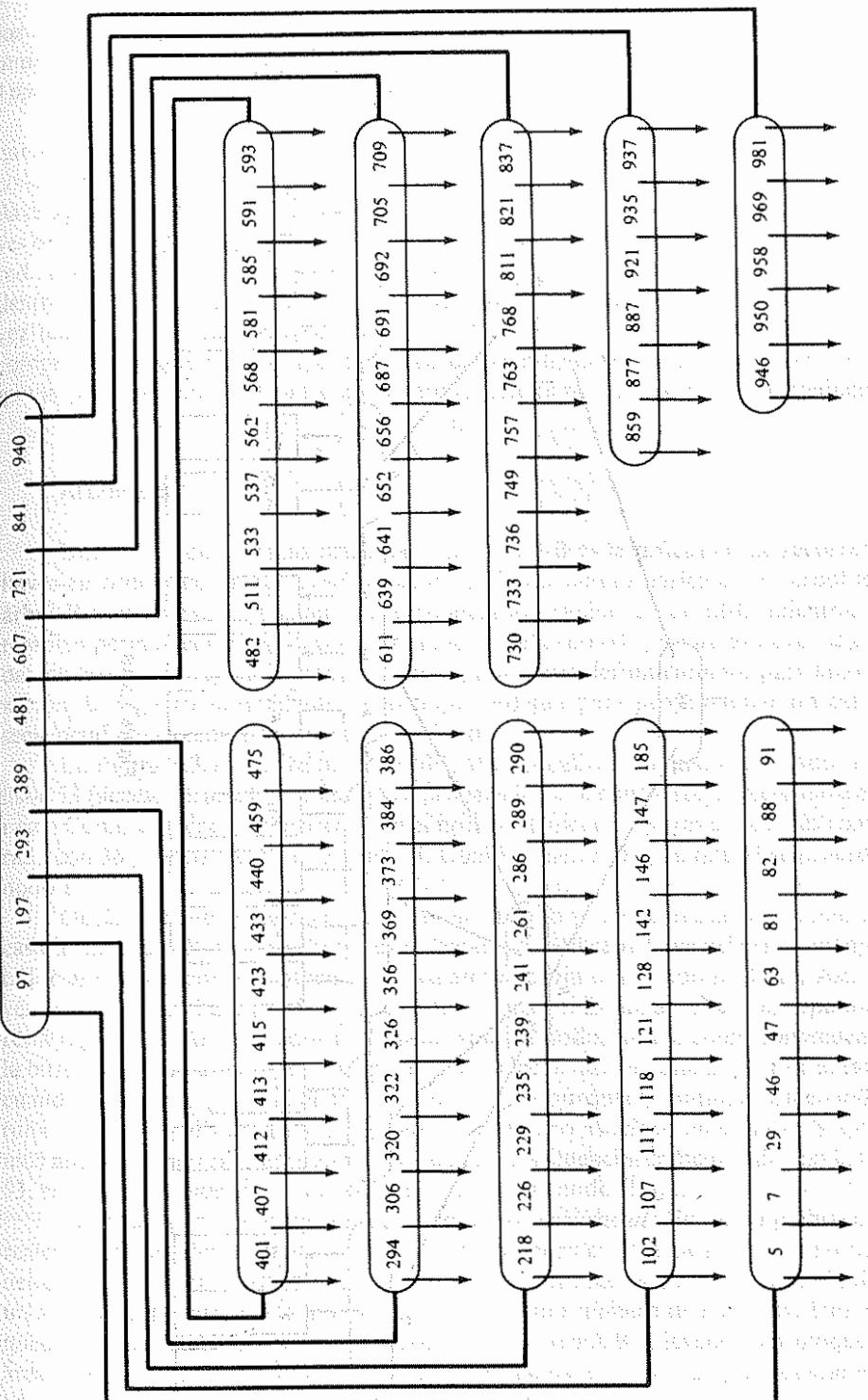


Figura 7.3.15

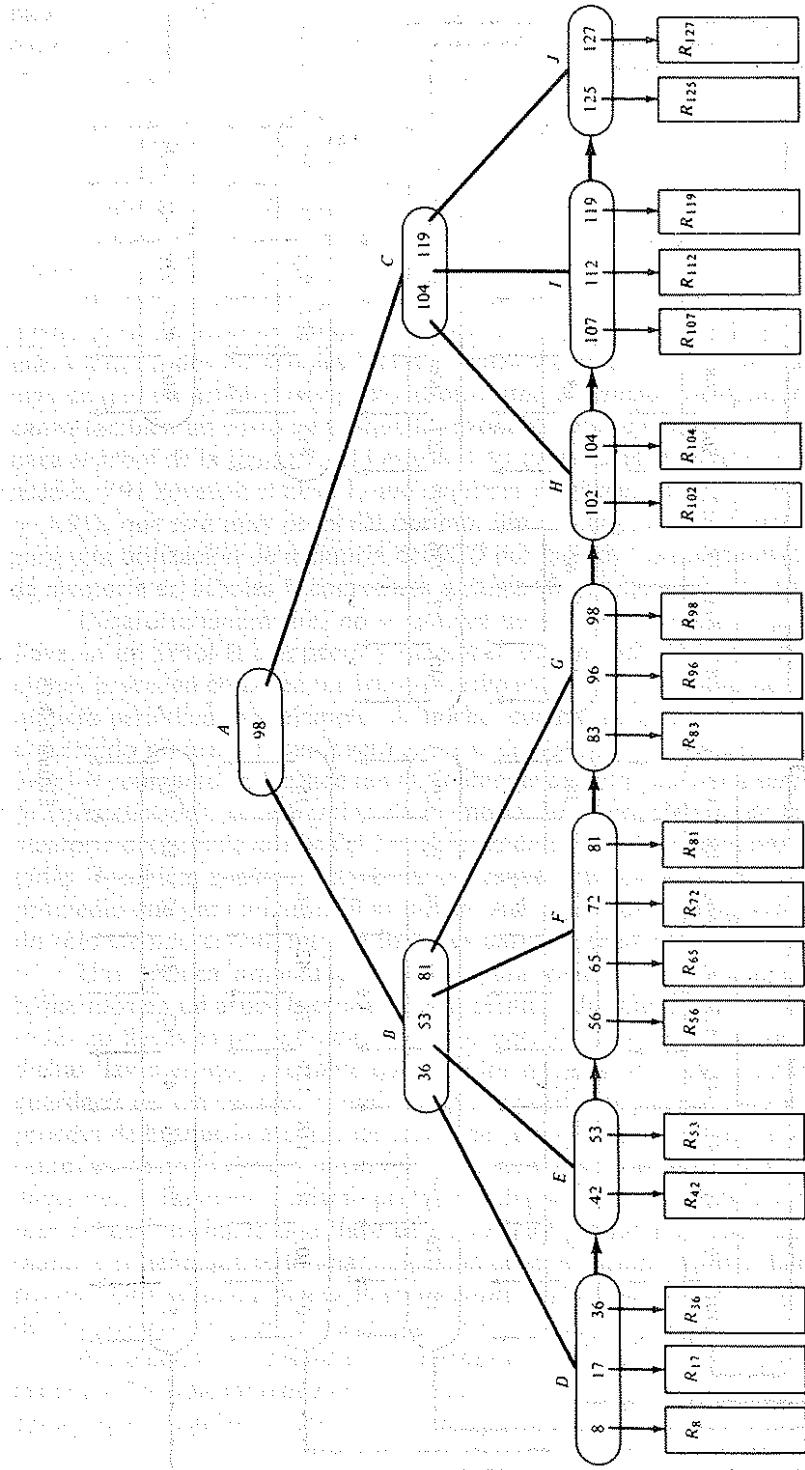


Figura 7.3.16

los sucesores de *andrew* dentro del nodo fuesen *andule*, *antoin*, *append* y *apples*, *andrew* sería codificado como *2d*, *andule* como *3u*, *antoin* como *2* y *append* como *1pe*.

Si se usa compresión final, es necesario accesar el registro mismo para determinar si una llave está presente en el archivo, dado que la llave completa no puede reconstruirse a partir de la codificación. También bajo ambos métodos, el código de la llave que se retiene es de longitud variable, de manera que el máximo número de llaves en un nodo ya no está fijo. Otra desventaja de la codificación de llaves con longitud variable es que la búsqueda binaria ya no puede usarse para localizar una llave dentro de un nodo. Además, el código de la llave para algunas llaves existentes tal vez tenga que ser cambiado cuando se inserta una nueva llave. La ventaja de la compresión es que permite que sean guardadas más llaves en un nodo, de manera que la profundidad del árbol y el número de nodos requeridos puede ser reducido.

Arboles-B⁺

Una de las desventajas principales del árbol-B es la dificultad de recorrer las llaves en orden secuencial. Una variación de la estructura básica de un árbol-B, el árbol-B⁺, retiene la propiedad de acceso aleatorio rápido del árbol-B, mientras que también permite el acceso secuencial rápido. En el árbol-B⁺, todas las llaves se guardan en hojas y se copian en nodos no-hoja para poder definir caminos para la localización de registros individuales. Las hojas se ligan para proporcionar un camino secuencial para recorrer las llaves en el árbol.

La figura 7.3.16 ilustra un árbol-B⁺. Para localizar el registro asociado con la llave 53 (acceso aleatorio), se compara primero la llave con 98 (la primera llave en la raíz). Como es menor, se procede con el nodo B. Cincuenta y tres es después comparada con 36 y luego con 53 en el nodo B. Como es menor o igual que 53 se procede al nodo E.

Obsérvese que la búsqueda no culmina cuando se encuentra la llave como en el caso de un árbol-B. En un árbol-B está contenido, con cada llave del árbol, un apuntador al registro correspondiente, ya sea en una hoja o un nodo no-hoja. Así, una vez encontrada la llave puede ser accedido el registro. En un árbol-B⁺ los apuntadores a registros están asociados a las llaves sólo en nodos hojas; como consecuencia, la búsqueda no está completa hasta que se localice la llave en una hoja. Por lo tanto, cuando se obtiene igualdad en un nodo no-hoja la búsqueda continúa. En el nodo E (una hoja) se localiza la llave 53, y de ella el registro asociado en la llave. Si queremos ahora recorrer las llaves en el árbol en orden secuencial comenzando con la llave 53, sólo necesitamos seguir los apuntadores en los nodos hoja.

La lista ligada de hojas se llama **conjunto secuencial**. En las implantaciones reales, los nodos del conjunto secuencial no contienen con frecuencia todas las llaves del archivo. En lugar de ello, cada nodo del conjunto secuencial sirve como índice a un área de datos grandes donde se guardan un gran número de registros. Una búsqueda involucra el recorrido de un camino en el árbol-B⁺, leyendo un bloque del área de datos asociada al nodo hoja que es accedida al final y después buscar en el bloque el registro requerido de manera secuencial.

El árbol-B⁺ puede ser considerado como una extensión natural del archivo secuencial indexado de la sección 7.1. Cada nivel del árbol es un índice al nivel que le sucede y el nivel menor, el conjunto secuencial, es un índice al propio archivo.

La inserción en un árbol-B⁺ procede de manera muy similar a la inserción en un árbol-B excepto que, cuando se divide un nodo, la llave del medio permanece en la mitad izquierda y es promovida al padre. Cuando se elimina una llave de una hoja, ésta puede retenerse en los nodos no-hojas, dado que es aún un separador válido entre las llaves en los nodos de abajo.

El árbol-B⁺ retiene la eficiencia en la búsqueda e inserción del árbol-B pero incrementa la eficiencia del hallazgo del registro siguiente en el árbol de $O(\log n)$ (en un árbol-B, donde encontrar el sucesor involucra subir o bajar en el árbol) a $O(1)$ (en un árbol-B⁺, donde se accesa una hoja adicional a lo sumo). Una ventaja adicional del árbol-B⁺ es que no se necesita guardar apuntadores a registros en los nodos no-hojas, lo que incrementa el orden potencial del árbol.

Arboles de búsqueda digitales

Otro método de usar árboles para apresurar una búsqueda es formar un árbol general basado en los símbolos de los que están compuestas las llaves. Por ejemplo, si las llaves son enteros, cada posición de dígito determina uno de diez hijos posibles de un nodo dado. Un bosque que representa uno de esos conjuntos de llaves se ilustra en la figura 7.3.17. Si las llaves constan de caracteres alfabéticos, cada letra del alfabeto determina una rama en el árbol. Adviértase que todo nodo hoja contiene el símbolo especial *eok*, que representa el final de una llave. Una hoja tal debe contener también un apuntador al registro que se está almacenando.

Si se representa un bosque por medio de un árbol binario, como en la sección 5.5, cada nodo del árbol binario contiene tres campos: *symbol*, que contiene un símbolo de la llave; *son*, que es un apuntador al hijo mayor del nodo en el árbol original; y *brother*, que es un apuntador al hermano siguiente más joven en el árbol original. El primer árbol en el bosque está apuntado por un apuntador externo *tree*, y las raíces de los otros árboles en el bosque están ligadas en una lista lineal mediante el campo *brother*. El campo *son* de una hoja en el bosque original apunta a un registro; la concatenación de todos los símbolos en el bosque original en el camino de nodos de la raíz a la hoja es la llave del registro. Hacemos dos estipulaciones adicionales que aceleran el proceso de búsqueda e inserción para un árbol de ese tipo; cada lista de hermanos está dispuesta en orden ascendente en el árbol binario según el campo *symbol*, y se considera que el símbolo *eok* es mayor que cualquier otro.

Usando esta representación de árbol binario, podemos presentar un algoritmo para buscar e insertar en un *árbol digital* no vacío de ese tipo. Como es usual, *key* es la llave para la cual se realiza la búsqueda, y *rec* es el registro que queremos insertar si no se encuentra la llave *key*. *key(i)* es el *i*-ésimo símbolo de la llave. Si la llave tiene *n* símbolos, suponemos que *key(n)* es igual a *eok*. El algoritmo usa la operación *get-node* para asignar un nuevo nodo al árbol cuando sea necesario. Suponemos que *recptr* es un apuntador al registro *rec* que será insertado. El algoritmo da como resultado un apuntador al registro que se está buscando y usa una función auxiliar *insert*, cuyo algoritmo también está dado.

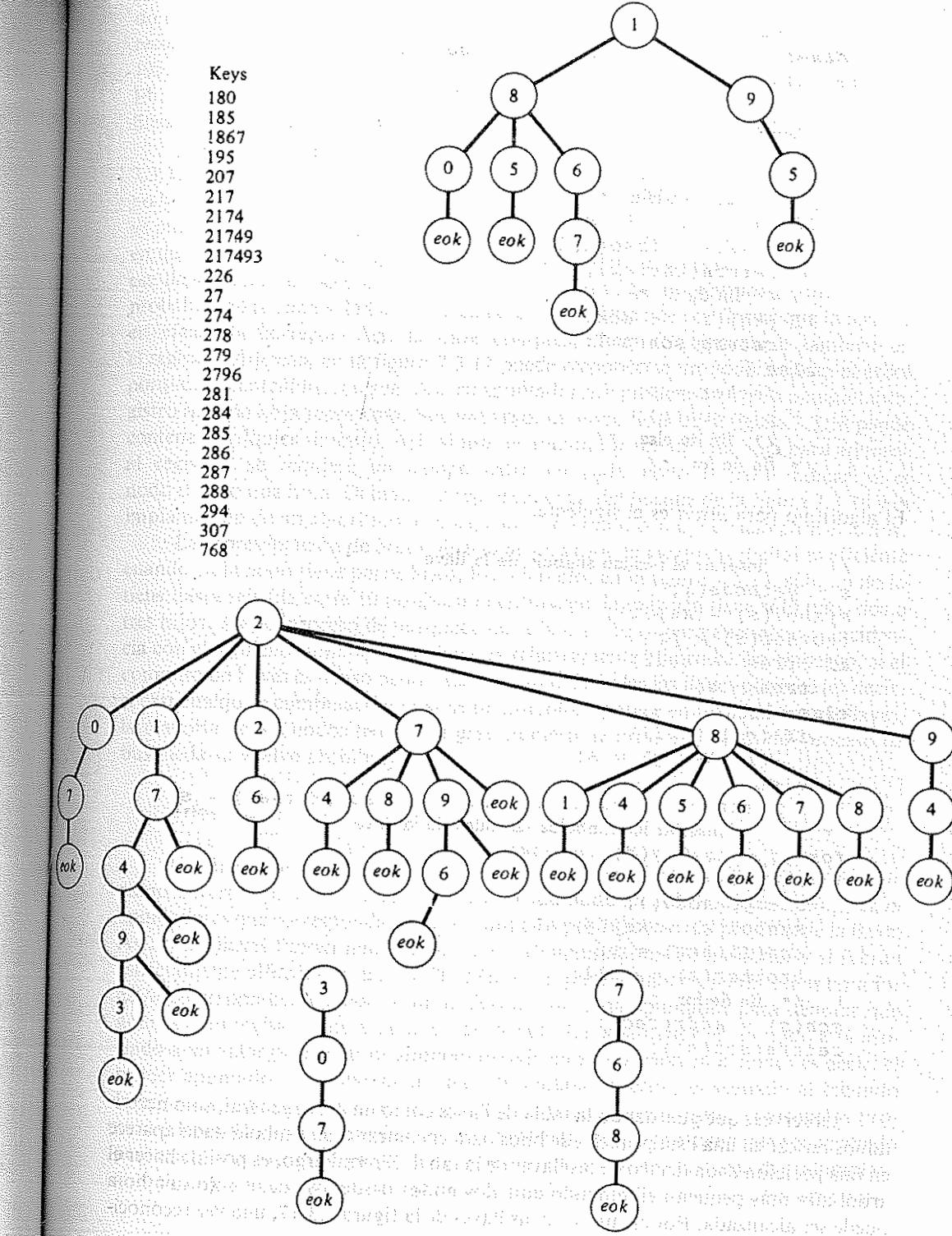


Figura 7.3.17 Un bosque que representa una tabla de llaves.

```

p = tree;
father = null; /* father es el padre de p */
for (i = 0;; i++) {
    q = null; /* q apunta al otro hermano de p */
    while (p != null && symbol(p) < key(i)) {
        q = p;
        p = brother(p);
    } /* fin de while */
    if (p == null || symbol(p) > key(i)) {
        insval = insert(i, p);
        return(insval);
    } /* fin de if */
    if (key(i) == eok)
        return(son(p));
    else {
        father = p;
        p = son(p);
    } /* fin de else */
} /* fin de for */

```

El algoritmo para *insert* es el siguiente:

```

/* insertar el i-ésimo símbolo de la llave */
s = getnode();
symbol(s) = key(i);
brother(s) = p;
if (tree == null)
    tree = s;
else
    if (q != null)
        brother(q) = s;
    else
        (father == null) ? tree = s : son(father) = s;
/* insertar los símbolos restantes de la llave */
for (j = i; key(j) != eok; j++) {
    father = s;
    s = getnode();
    symbol(s) = key(j + 1);
    son(father) = s;
    brother(s) = null;
} /* fin de for */
son(s) = addr(rec);
return(son(s));

```

Obsérvese que guardando la tabla de llaves como un árbol general, sólo necesitamos buscar en una lista pequeña de hijos para encontrar si un símbolo dado aparece en una posición dada dentro de las llaves de la tabla. Sin embargo, es posible hacer el árbol aún más pequeño eliminando aquellos nodos desde los cuales sólo una hoja puede ser alcanzada. Por ejemplo, en las llaves de la figura 7.3.17, una vez reconoci-

do el símbolo '7', la única llave que puede coincidir es 768. De manera similar, al reconocer los dos símbolos '1' y '9' la única llave donde puede haber coincidencia es 195. Así, el bosque de la figura 7.3.17 se puede abreviar como el de la figura 7.3.18. En esa figura un cuadrado indica una llave y un círculo un nodo del árbol. Se usa línea punteada para indicar un apuntador de un nodo del árbol a una llave.

Hay algunas diferencias significativas entre los árboles de las figuras 7.3.17 y 7.3.18. En la 7.3.17, un camino de la raíz a una hoja representa una llave entera; así no hay necesidad de repetir la propia llave. En la figura 7.3.18, sin embargo, una llave puede ser reconocida con sólo algunos de sus símbolos iniciales. En aquellos casos en que se busca una llave que se sabe está en la tabla, el registro que corresponde a esa llave puede ser accesado al encontrar una hoja. Si, no obstante, como es más probable, no se sabe si la llave está en la tabla, se tiene que confirmar que la llave es en efecto la correcta. Así, la llave completa tiene que guardarse también en el registro. Además, en la figura 7.3.17 puede reconocerse un nodo hoja en el árbol porque sus contenidos son *eok*. Así, su apuntador *son* puede usarse para apuntar al registro que esa hoja representa. Sin embargo, un nodo hoja de la figura 7.3.18 puede contener cualquier símbolo. Así, al usar el apuntador *son* de una hoja para apuntar al registro, se requiere un campo extra en cada nodo o para indicar si el nodo es o no una hoja. Dejamos la representación del bosque de la figura 7.3.18 y la implantación de un algoritmo de búsqueda e inserción como ejercicio para el lector.

La representación de árbol binario de un árbol de búsqueda digital es eficiente cuando cada nodo tiene pocos hijos. Por ejemplo, en la figura 7.3.18, sólo un nodo tiene hasta seis (de entre 10 posibles) mientras que la mayoría tiene sólo uno, dos o tres hijos. Así, el proceso de búsqueda en la lista de hijos para encontrar coincidencia con el siguiente símbolo de la llave es relativamente eficiente. Sin embargo, si el conjunto de llaves es denso dentro del conjunto de todas las llaves posibles (es decir, si casi cualquier combinación posible de símbolos aparece en realidad en una llave), la mayoría de los nodos tendrá un gran número de hijos y el costo del proceso de búsqueda se vuelve prohibitivo.

Tries

Un árbol de búsqueda digital no tiene que ser implantado necesariamente como un árbol binario. En lugar de ello, cada nodo en el árbol puede contener *m* apuntadores que correspondan a los *m* símbolos posibles en cada posición de la llave. Así, si las llaves fueran numéricas, habría 10 apuntadores en un nodo, y si fueran estrictamente alfabeticas, habría 26. (También podría haber además un apuntador extra correspondiente a *eok*, o un indicador con cada apuntador para denotar que apunta a un registro y no a un nodo del árbol). Un apuntador en un nodo está asociado a un valor particular de símbolo basado en su posición en el nodo; es decir, el primer apuntador corresponde al valor de símbolo inferior, el segundo al segundo inferior y así de manera sucesiva. En consecuencia es innecesario guardar los propios valores de símbolos en el árbol. El número de nodos que tiene que ser accesado para encontrar una llave particular es $\log mn$. Un árbol de búsqueda digital implantado de esta manera se llama un *trie* (de la palabra en inglés *retrieval*, recuperación).

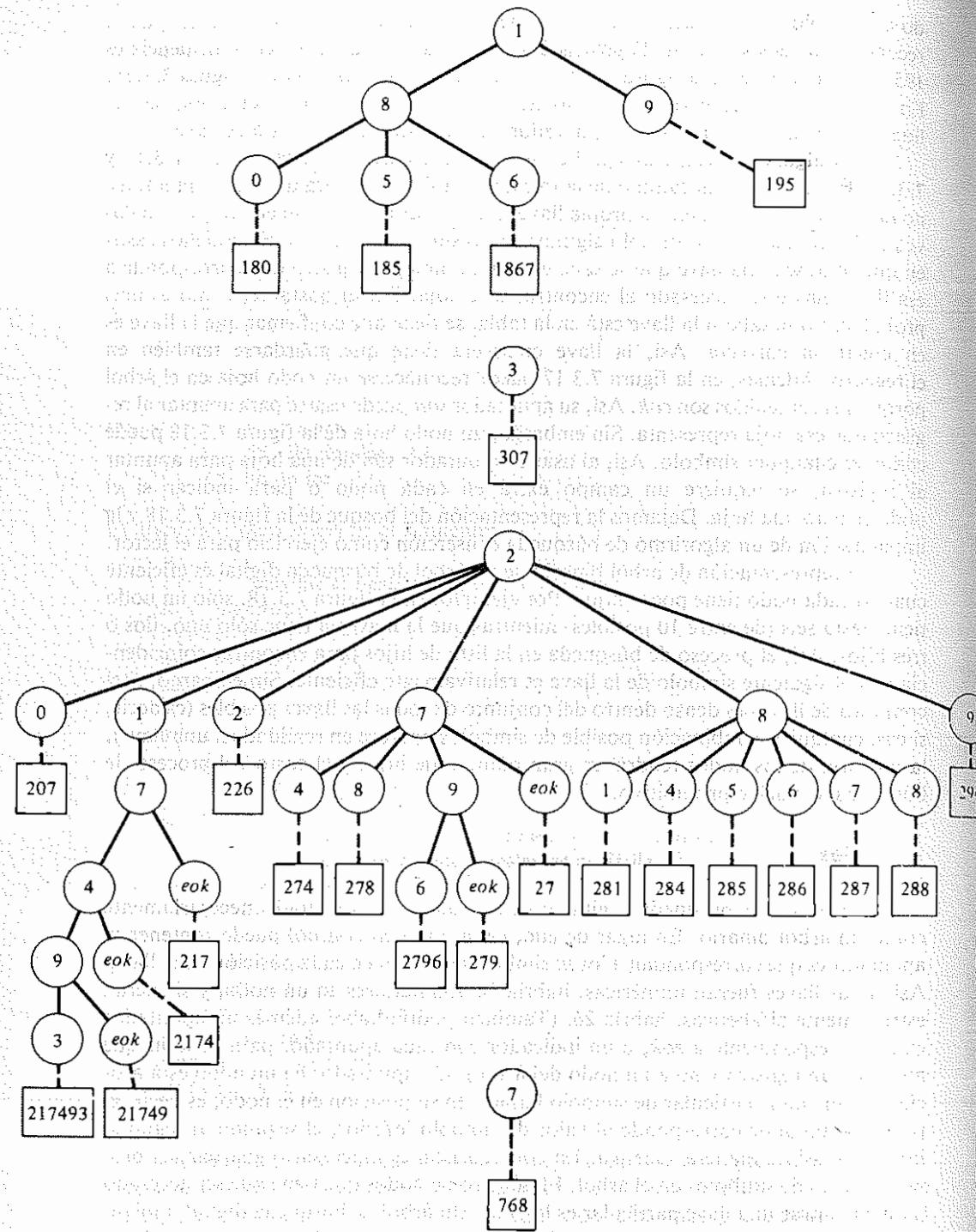


Figura 7.3.18 Un bosque condensado que representa una tabla de llaves.

Un trie es útil cuando el conjunto de llaves es denso, de manera que la mayoría de los apuntadores en cada nodo se usan. Cuando el conjunto de llaves es escaso, un trie gasta una gran cantidad de espacio con nodos muy grandes que están vacíos en su mayor parte. Si se conoce de antemano el conjunto de llaves en un trie y éste no cambia, hay un sinnúmero de técnicas para minimizar los requerimientos de espacio. Una técnica es establecer un orden diferente en el cual se usan los símbolos de una llave para la búsqueda (de manera que, por ejemplo, el tercer símbolo de la llave argumento pueda ser usado para accesar el apuntador apropiado en la raíz del trie, el primer símbolo en los nodos del nivel 1, y así de manera sucesiva.) Otra técnica es permitir que los nodos del trie se traslapen unos a otros, de manera que apuntadores ocupados de un nodo recubran apuntadores vacíos de otro.

EJERCICIOS

- 7.3.1. Muestre cómo puede usarse un árbol-B y un árbol-B⁺ para implantar una cola de prioridad (ver secciones 4.1 y 6.3). Mostrar que una secuencia de n inserciones y operaciones de eliminación mínimas puede ser ejecutada en $O(n \log n)$ pasos. Escribir rutinas en C para insertar y eliminar en una cola de prioridad implantada por medio de un árbol 2-3.
- 7.3.2. Elija cualquier párrafo grande de un libro. Insertar cada palabra del párrafo, en orden, dentro de un árbol de búsqueda multivías de “arriba abajo” inicialmente vacío de orden-5, omitiendo cualquier duplicado. Hacer lo mismo para un árbol-B de orden-5, un árbol-B⁺ de orden-5 y un árbol de búsqueda digital.
- 7.3.3. Escriba rutinas en lenguaje C para implantar las operaciones de inserción de un sucesor en un árbol-B si el árbol-B se guarda en:
 - memoria interna.
 - memoria externa de acceso directo.
- 7.3.4. Escriba un algoritmo y una rutina en lenguaje C para eliminar un registro de un árbol de búsqueda multivías de “arriba abajo” de orden n .
- 7.3.5. Escriba un algoritmo y una rutina en lenguaje C para eliminar un registro de un árbol-B de orden n .
- 7.3.6. Escriba un algoritmo para crear un árbol-B compacto a partir de una entrada ordenada. Usar el algoritmo para escribir una rutina en lenguaje C para producir un árbol-B compacto a partir de un árbol-B ordinario.
- 7.3.7. Escriba un algoritmo y una rutina en lenguaje C para realizar una búsqueda en un árbol-B.
- 7.3.8. Escriba una rutina en lenguaje C y un algoritmo para:
 - insertar en un árbol-B⁺
 - insertar en un árbol-B*
 - eliminar de un árbol-B⁺
 - eliminar de un árbol-B*
- 7.3.9. ¿Cuántos árboles 2-3 diferentes que contengan los enteros del 1 al 10 se pueden construir? ¿Cuántas permutaciones de esos enteros resultan en cada árbol si se insertan en un árbol al inicio vacío en orden de permutación?
- 7.3.10. Desarrolle algoritmos para buscar e insertar en un árbol-B que usen compresión frontal y final.

7.3.11. Escriba un algoritmo de búsqueda e inserción y una rutina en lenguaje C que lo realice para el bosque de búsqueda digital de la figura 7.3.18.

7.3.12. Muestre cómo implantar un trie en la memoria externa. Escriba una rutina de búsqueda e inserción en lenguaje C para un trie.

7.4. DISPERSION

En las dos secciones precedentes supusimos que el registro estaba guardado en una tabla y que era necesario pasar a través de cierto número de llaves antes de encontrar la deseada. La organización del archivo (secuencial, secuencial indexada, árbol binario, etc.) y el orden en el que se insertan las llaves afectan el número de llaves a inspeccionar antes de obtener la deseada. Es obvio que las técnicas de búsqueda eficientes son aquellas que hacen mínimo el número de comparaciones. De manera óptima, desearíamos tener una organización de la tabla y una técnica de búsqueda en la cual no hubiesen comparaciones innecesarias. Veamos si esto es viable.

Si cada llave debe recuperarse en un solo acceso, la localización del registro dentro de la tabla puede depender sólo de la llave; no puede depender de la localización de las otras llaves, como en un árbol. La manera más eficiente de organizar una tabla de ese tipo es como un arreglo (es decir, cada registro se guarda en un desplazamiento específico de la dirección base de la tabla). Si los registros son enteros, las propias llaves pueden servir como índices al arreglo.

Consideremos un ejemplo de un sistema de ese tipo. Suponer que una compañía de manufactura tiene un archivo de inventarios que consta de 100 piezas, cada una con un número de pieza único de dos dígitos. Entonces la manera obvia de almacenar este archivo es declarando un arreglo

```
parttype part[100];
```

donde *part[i]* representa el registro cuyo número de pieza es *i*. En esta situación, los números de pieza son llaves que se usan como índices del arreglo. Aun cuando la compañía tenga existencias de menos de 100 piezas, se puede usar la misma estructura para guardar el archivo inventario. Aunque muchas localizaciones en *part* pueden corresponder a llaves no existentes, este gasto está compensado por la ventaja del acceso directo a cada una de las piezas existentes.

Desafortunadamente, sin embargo, tal sistema no es siempre práctico. Por ejemplo, suponer que la compañía tiene un archivo inventario de más de 100 artículos y que la llave de cada registro es un número de pieza de siete dígitos. Entonces se requeriría de un arreglo de 10 millones de elementos para usar la indexación directa con la llave entera de siete dígitos. Esto desperdicia una cantidad de espacio inaceptable porque es muy improbable que una compañía surta más de unos pocos miles de piezas.

Lo que es necesario es algún método para convertir una llave en un entero dentro de un rango limitado. De manera ideal, dos llaves no deberían ser convertidas en el mismo entero. Sin embargo, tal método ideal por lo regular no existe. Intenta-

remos el desarrollo de métodos que estén próximos al ideal, y determinar qué hacer cuando no se alcanza el ideal.

Consideremos el ejemplo de una compañía con un archivo inventario en el cual cada registro tiene como llave un número de pieza de siete dígitos. Supóngase que la compañía tiene menos de 1000 piezas y que hay un solo registro para cada pieza. Entonces un arreglo de 1000 piezas es suficiente para contener el archivo completo. El arreglo está indexado por enteros del 0 al 999 inclusive. Los últimos tres dígitos del número de pieza se usan como índice para el registro de la pieza en el arreglo. Esto está ilustrado en la figura 7.4.1. Obsérvese que dos llaves que estén relativamente cerca la una de la otra en orden numérico, como 4618396 y 4618996 pueden estar más alejadas entre sí en la tabla que dos llaves muy separadas tales como 0000991 y 9846995. Sólo los últimos tres dígitos de la llave se usan para determinar la posición de un registro.

Una función que transforme una llave dentro del índice de una tabla se llama una *función de dispersión*. Si *h* es una función de dispersión y *key* es una llave, *h(key)* se llama la *dispersión de llave* y es el índice en el cual un registro con la llave

Posición	Llave	Registro
0	4967000	
1		
2	8421002	
3		
395		
396	4618396	
397	4957397	
398		
399	1286399	
400		
401		
990	0000990	
991	0000991	
992	1200992	
993	0047993	
994		
995	9846995	
996	4618996	
997	4967997	
998		
999	0001999	

Figura 7.4.1