

ESTRUCTURAS DE DATOS Y ALGORITMOS

ALBERTO V. RODRÍGUEZ
JUAN E. MELCHORREÑA
ANTONI E. SÁNCHEZ



Estructura de datos y algoritmos

Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
con la colaboración de Guillermo Levine Gutiérrez;
versión en español de Américo Vargas y Jorge Lozano
PUBLICACIÓN México, DF : Addison-Wesley
Iberoamericana: Sistemas Técnicos de Edición, 1988
ISBN 968-6048-19-7

1

Diseño y análisis de algoritmos

Escribir un programa de computador para resolver un problema comprende varios pasos que van desde la formulación y especificación del problema, el diseño de la solución, su implantación, prueba y documentación, hasta la evaluación de la solución. En este capítulo se presenta el punto de vista de los autores sobre estos pasos. En los capítulos siguientes se analizan los algoritmos y estructuras de datos con los que se construye la mayor parte de los programas de computador.

1.1 De los problemas a los programas

La mitad del trabajo es saber qué problema se va a resolver. Al abordar los problemas, por lo general, éstos no tienen una especificación simple y precisa de ellos. De hecho, problemas como crear una receta digna de un gastrónomo o preservar la paz mundial pueden ser imposibles de formular de forma que admitan una solución por computador; aunque se crea que el problema puede resolverse en un computador, es usual que la distancia entre varios de sus parámetros sea considerable. A menudo sólo mediante experimentación es posible encontrar valores razonables para estos parámetros.

Si es posible expresar ciertos aspectos de un problema con un modelo formal, por lo general resulta beneficioso hacerlo, pues una vez que el problema se formaliza, pueden buscarse soluciones en función de un modelo preciso y determinar si ya existe un programa que resuelva tal problema; aun cuando no sea tal el caso, será posible averiguar lo que se sabe acerca del modelo y usar sus propiedades como ayuda para elaborar una buena solución.

Se puede recurrir casi a cualquier rama de las matemáticas y de las ciencias para obtener un modelo de cierto tipo de problemas. En el caso de problemas de naturaleza esencialmente numérica, esto puede lograrse a través de conceptos matemáticos tan familiares como las ecuaciones lineales simultáneas (por ejemplo, para determinar las corrientes en un circuito eléctrico o encontrar los puntos de tensión en estructuras de vigas conectadas) o ecuaciones diferenciales (por ejemplo, para predecir el crecimiento de una población o la velocidad de una reacción química). Tratándose de problemas de procesamiento de símbolos y textos, se pueden construir modelos con cadenas de caracteres y gramáticas formales. Entre los problemas de esta categoría están la compilación (traducción a lenguaje de máquina de programas escritos en algún lenguaje de programación) y las tareas de recuperación de infor-

mación, como el reconocimiento de ciertas palabras en el catálogo de títulos de una biblioteca.

Algoritmos

Cuando ya se cuenta con un modelo matemático adecuado del problema, puede buscarse una solución en función de ese modelo. El objetivo inicial consiste en hallar una solución en forma de *algoritmo*, que es una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito. Una proposición de asignación entera como $x := y + z$ es un ejemplo de instrucción que puede ejecutarse con una cantidad finita de esfuerzo. Las instrucciones de un algoritmo pueden ejecutarse cualquier número de veces, siempre que ellas mismas indiquen la repetición. No obstante, se exige que un algoritmo termine después de ejecutar un número finito de instrucciones, sin importar cuáles fueron los valores de entrada. Así, un programa es un algoritmo mientras no entre en un ciclo infinito con ninguna entrada.

Es necesario aclarar un aspecto de esta definición de algoritmo. Se ha dicho que cada instrucción de un algoritmo debe tener un «significado preciso» y debe ser ejecutable con una «cantidad finita de esfuerzo»; pero lo que está claro para una persona, puede no estarlo para otra, y a menudo es difícil demostrar de manera rigurosa que una instrucción puede realizarse en un tiempo finito. A veces, también es difícil demostrar que una secuencia de instrucciones termina con alguna entrada, aunque esté muy claro el significado de cada instrucción. Sin embargo, al argumentar en favor y en contra, por lo general se llega a un acuerdo respecto a si una secuencia de instrucciones constituye o no un algoritmo. Quien afirme tener un algoritmo debe asumir la responsabilidad de demostrarlo. En la sección 1.5 se verá cómo estimar el tiempo de ejecución de algunas construcciones usuales con lenguajes de programación de las que se puede demostrar que requieren de un tiempo finito de ejecución.

Además de usar programas en Pascal como algoritmos, se empleará un *seudolenguaje* que es una combinación de las construcciones de un lenguaje formal de programación con proposiciones informales expresadas en español. Se utilizará Pascal como lenguaje de programación, pero para realizar los algoritmos que se estudiarán, éste puede sustituirse casi por cualquier otro lenguaje conocido. El ejemplo que se da a continuación ilustra muchos de los pasos que componen la escritura de un programa de computador, de acuerdo con el esquema de los autores.

Ejemplo 1.1. Un modelo matemático puede ayudar a diseñar un semáforo para una intersección complicada de calles. Para construir la secuencia de señales, se creará un programa que acepte como entrada el conjunto de giros permisibles en una intersección (un «giro» también será continuar en línea recta por una calle) y que divida este conjunto en el menor número posible de grupos, de manera que todos los giros de un grupo sean permisibles en forma simultánea sin colisiones. Después, se asociará una señal del semáforo a cada grupo de giros. Buscando una división que tenga el menor número posible de grupos, se obtendrá un semáforo con el menor número posible de fases.

Se tomará como ejemplo la intersección de la figura 1.1, donde las calles *C* y *E*

son de una dirección, mientras las restantes son de dos. En esta intersección es posible realizar 13 giros. Algunos pares de giros, como AB (de A hacia B) y EC , pueden ocurrir en forma simultánea, mientras que otros, como AD y EB , ocasionan el cruce entre líneas de tráfico y, por tanto, no pueden realizarse al mismo tiempo. Así, las luces del semáforo deben permitir los giros en un orden tal que AD y EB nunca ocurran al mismo tiempo, en tanto que AB y EC puedan ser simultáneos.

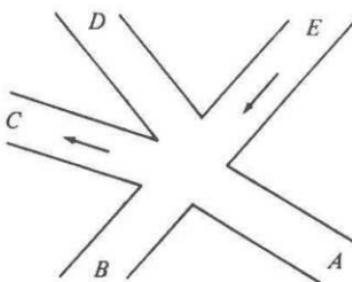


Fig. 1.1. Una intersección.

Puede obtenerse un modelo de este problema con la ayuda de una estructura matemática conocida como *grafo*. Un grafo se compone de un conjunto de puntos, llamados *vértices*, y de líneas que unen los puntos, llamadas *aristas*. Es posible construir un modelo del problema de la intersección de tráfico mediante un grafo cuyos vértices representen los giros posibles y cuyos arcos unan los pares de vértices que representan giros que no se permiten al mismo tiempo. En la figura 1.2 se muestra el grafo para la intersección de la figura 1.1. La figura 1.3 es otra representación del mismo grafo en forma de tabla, donde un 1 en la intersección de la fila i y la columna j indica que hay una arista entre los vértices i y j .

El grafo puede ayudar a solucionar el problema de diseñar el semáforo. La *coloración* de un grafo es la asignación de un color a cada vértice de éste, de tal forma que no haya dos vértices del mismo color unidos por un arco. Así pues, resulta evidente que el problema equivale a buscar una coloración del grafo de giros incompatibles que utilice el menor número posible de colores.

El problema de la coloración de grafos se ha estudiado durante varias décadas, la teoría de algoritmos ofrece mucha información. Lamentablemente, la coloración de un grafo arbitrario con el menor número posible de colores es un problema que se clasifica entre los llamados «NP-completos», para los cuales las únicas soluciones conocidas consisten, en esencia, en «intentar todas las posibilidades». En el problema de la coloración, esto significa intentar todas las asignaciones de colores a los vértices, primero con un solo color, luego con dos, con tres, y así sucesivamente, hasta encontrar una coloración válida. Con un poco de cuidado es posible acelerar este procedimiento, pero, en general, se cree que ningún algoritmo que resuelva este problema puede ser sustancialmente más eficiente que el enfoque, más obvio, recién descrito.

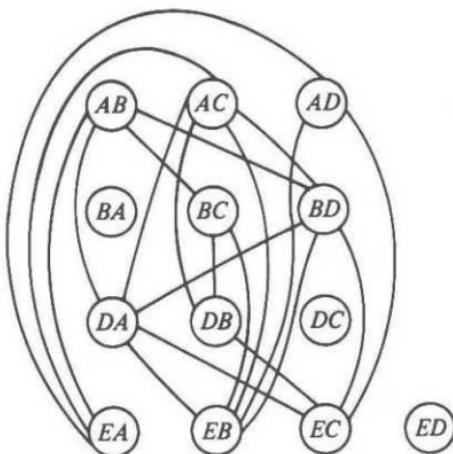


Fig. 1.2. Grafo que representa los giros incompatibles.

	<i>AB</i>	<i>AC</i>	<i>AD</i>	<i>BA</i>	<i>BC</i>	<i>BD</i>	<i>DA</i>	<i>DB</i>	<i>DC</i>	<i>EA</i>	<i>EB</i>	<i>EC</i>	<i>ED</i>
<i>AB</i>					1	1	1			1			
<i>AC</i>						1	1			1			
<i>AD</i>								1		1			
<i>BA</i>									1				
<i>BC</i>	1										1		
<i>BD</i>	1	1									1	1	
<i>DA</i>	1	1					1				1	1	
<i>DB</i>		1			1								1
<i>DC</i>													1
<i>EA</i>	1	1	1										
<i>EB</i>		1	1		1	1	1						
<i>EC</i>			1			1	1	1	1				
<i>ED</i>													

Fig. 1.3. Tabla de giros incompatibles.

Ahora, existe la posibilidad de que encontrar una solución óptima al problema en cuestión tenga un costo computacional muy elevado. Ante esto, puede elegirse entre tres enfoques. Si el grafo es pequeño se puede buscar una solución óptima a través de una búsqueda exhaustiva, intentando todas las posibilidades. Para grafos grandes, en cambio, este enfoque resulta prohibitivo por su costo, sin importar qué gra-

do de eficiencia se intente dar al programa. El segundo enfoque consiste en buscar información adicional sobre el problema. Podría suceder que el grafo tuviera propiedades especiales que hicieran innecesario probar todas las posibilidades para hallar una solución óptima. El tercer enfoque consiste en cambiar un poco el problema y buscar una solución buena aunque no necesariamente óptima; podría ser suficiente con encontrar una solución que determinara un número de colores cercano al mínimo en grafos pequeños, y que trabajara con rapidez, pues la mayor parte de las intersecciones no son tan complejas como las de la figura 1.1. Un algoritmo que produce con rapidez soluciones buenas, pero no necesariamente óptimas, se denomina *heurístico*.

El siguiente algoritmo «ávido» es un heurístico razonable para la coloración de grafos. Para empezar, se intenta colorear tantos vértices como sea posible con el primer color; después, todos los que sea posible con el segundo color, y así sucesivamente. Para dar a los vértices un nuevo color, se procede como sigue:

1. Se selecciona un vértice no coloreado y se le asigna el nuevo color.
2. Se examina la lista de vértices no coloreados. Para cada uno de ellos se determina si existe alguna arista que lo une con un vértice que ya tenga asignado el nuevo color; si tal arista no existe, se asigna el nuevo color al vértice.

Este enfoque recibe el nombre «ávido» porque le asigna color a un vértice en cuanto le es posible, sin considerar las potenciales desventajas inherentes a tal acción. Hay situaciones en las que podrían colorearse más vértices con un mismo color si fuera menos «ávido» y pasara por alto algún vértice que legalmente se podría colorear. Por ejemplo, considérese el grafo de la figura 1.4, donde, después de colorear con rojo el vértice 1, se puede asignar el mismo color a los vértices 3 y 4, siempre que no se coloree antes el vértice 2. Si se consideran los vértices en orden numérico, con el algoritmo exhaustivo se asignaría el color rojo a los vértices 1 y 2.

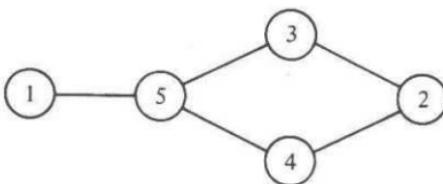


Fig. 1.4. Un grafo.

Como ejemplo del enfoque ávido aplicado a la figura 1.2, supóngase que se empieza por asignar a AB el color azul. Se pueden colorear AC , AD y BA de azul, pues ninguno de estos tres vértices tiene una arista común con AB . No es posible asignar el color azul al vértice BC , porque existe una arista entre AB y BC . Del mismo modo, no se pueden colorear de azul BD , DA ni DB , porque están unidos por una arista a uno o más vértices de los ya coloreados. Sin embargo, sí se puede asignar el azul a DC . Así, EA , EB y EC no se pueden colorear con azul, pero ED sí.

Ahora se empieza con el segundo color, por ejemplo, coloreando BC de rojo. BD puede colorearse de rojo, pero DA no, a causa de la arista entre BD y DA . De igual manera, a DB no se le puede asignar el color rojo y DC ya es azul, pero EA sí se puede colorear de rojo. Los demás vértices sin color tienen ahora una arista con un vértice rojo, de modo que ningún otro vértice puede ser rojo.

Los vértices que aún no tienen color son DA , DB , EB y EC . Si se colorea DA de verde, DB también podrá ser verde, pero EB y EC no. Estos deben colorearse con un cuarto color, por ejemplo, amarillo. La asignación de colores se resume en la figura 1.5. Al aplicar el enfoque ávido se determina que los giros «adicionales» de cierto color son compatibles con aquellos que ya tenían asignado ese mismo color y entre sí. Cuando el semáforo permita giros de un color, también permitirá los giros adicionales sin colisiones.

color	giros	giros adicionales
azul	AB, AC, AD, BA, DC, ED	—
rojo	BC, BD, EA	BA, DC, ED
verde	DA, DB	AD, BA, DC, ED
amarillo	EB, EC	BA, DC, EA, ED

Fig. 1.5. Coloración del grafo de la figura 1.2.

El enfoque ávido no siempre usa el menor número posible de colores. Ahora, es posible utilizar de nuevo la teoría de algoritmos para evaluar la bondad de la solución obtenida. En teoría de grafos, un *grafo completo-k* es un conjunto de k vértices donde cada par de ellos está unido por una arista. Puesto que en un grafo completo no hay dos vértices a los que se pueda asignar el mismo color, es obvio que se necesitan k colores para colorear un grafo completo- k .

En el grafo de la figura 1.2, el conjunto de cuatro vértices AC , DA , BD y EB es un grafo completo-4. Para este grafo no existe, pues, una coloración posible con tres colores o menos y, por tanto, la solución de la figura 1.5 es óptima en el sentido de que usa el menor número posible de colores. Para el problema original, ningún semáforo puede tener menos de cuatro fases para la intersección de la figura 1.1.

Así pues, considérese el control de un semáforo basado en la figura 1.5, donde cada fase del control corresponde a un color. En cada fase, se permiten los giros indicados en la fila de la tabla que corresponde a ese color, y se prohíben los restantes. Este modelo utiliza el menor número posible de fases. □

Seudolenguaje y refinamiento por pasos

Una vez que se tiene un modelo matemático apropiado para un problema, puede formularse un algoritmo basado en ese modelo. Las versiones iniciales del algoritmo a menudo están mezcladas en proposiciones generales que deberán refinarse después en instrucciones más pequeñas y definidas. Por ejemplo, se ha descrito el algoritmo ávido de coloración de grafos con expresiones del tipo «elegir algún vértice no coloreado». Es de esperar que esas instrucciones sean lo bastante claras para com-

prender el razonamiento. Sin embargo, para convertir en programa un algoritmo tan informal, es necesario pasar por varias etapas de formalización (*refinamiento por pasos*) hasta llegar a un programa cuyos pasos tengan un significado formalmente definido en el manual de algún lenguaje de programación.

Ejemplo 1.2. Considérese el algoritmo ávido de coloración de grafos con el fin de convertirlo en un programa en Pascal. A continuación, se supondrá que existe un grafo G con algunos vértices que pueden colorearse. El siguiente programa *ávido* determina un conjunto de vértices llamado *nue_color* a los cuales se puede asignar un nuevo color. Se llama repetidas veces al programa hasta colorear todos los vértices. Sin entrar en detalles, se puede especificar *ávido* en seudolenguaje como se muestra en la figura 1.6.

```

procedure ávido ( var  $G$ : GRAFO; var nue_color: CONJUNTO );
  { ávido asigna a nue_color un conjunto de vértices de  $G$  a los que se
    puede dar el mismo color }
  begin
    (1)   nue_color := Ø; †
    (2)   for cada vértice  $v$  no coloreado de  $G$  do
    (3)     if  $v$  no es adyacente a ningún vértice de nue_color then begin
    (4)       marca  $v$  como coloreado;
    (5)       agrega  $v$  a nue_color
    end
  end; { ávido }

```

Fig. 1.6. Primer refinamiento del algoritmo ávido.

En la figura 1.6 se observan algunas características importantes de este seudolenguaje. La primera de ellas es que se utilizan letras minúsculas negritas para las palabras clave de Pascal que tienen el mismo significado que en Pascal estándar. Las palabras en mayúscula, como GRAFO y CONJUNTO ††, son nombres de «tipos de datos abstractos». Estos se definirán mediante declaraciones tipo Pascal, y las operaciones asociadas con esos tipos de datos abstractos se definirán mediante procedimientos de Pascal al crear el programa final. En las dos secciones siguientes se analizarán con más detalle los tipos de datos abstractos.

Las construcciones de flujo de control de Pascal como if, for y while se pueden utilizar en las proposiciones en seudolenguaje, pero las expresiones condicionales, como la de la línea (3), pueden ser proposiciones informales, en vez de expresiones condicionales de Pascal. Obsérvese que la asignación de la línea (1) utiliza una expresión informal a la derecha, y que el ciclo for de la línea (2) itera sobre un conjunto.

Para su ejecución, el programa en seudolenguaje de la figura 1.6 se debe refinrar hasta convertirlo en un programa convencional en Pascal. Este proceso no se reali-

† El símbolo Ø representa el conjunto vacío.

†† Se debe distinguir entre el tipo de datos abstracto CONJUNTO y el tipo incorporado set de Pascal.

zará en forma completa para el programa de este ejemplo, pero sí se dará un ejemplo de refinamiento, transformando la proposición *if* de la línea (3) de la figura 1.6 en un código más convencional.

Para probar si un vértice *v* es adyacente a otro vértice de *nue_color*, se puede considerar cada miembro *w* de *nue_color* y examinar el grafo *G* para determinar la existencia de una arista entre *v* y *w*. Una manera organizada de realizar esta prueba consiste en utilizar una variable booleana *encontrado* que indique si se ha encontrado una arista. De este modo, las líneas (3) a (5) de la figura 1.6 pueden reemplazarse por el código de la figura 1.7.

```

procedure ávido ( var G: GRAFO; var nue_color: CONJUNTO );
begin
(1)      nue_color := Ø ;
(2)      for cada vértice v no coloreado de G do begin
(3.1)          encontrado := false;
(3.2)          for cada vértice w de nue_color do
(3.3)              if hay una arista entre v y w en G then
(3.4)                  encontrado := true;
(3.5)          if encontrado = false then begin
(4)              { v no es adyacente a ningún vértice de nue_color }
(5)              marca v como coloreado;
(5)              agrega v a nue_color
            end
        end
    end;
end;

```

Fig. 1.7. Refinamiento parcial de la figura 1.6.

Ahora, el algoritmo se ha reducido a una colección de operaciones sobre dos conjuntos de vértices. El ciclo externo, compuesto por las líneas (2) a (5), itera sobre el conjunto de vértices no coloreados de *G*. El ciclo interno, líneas (3.2) a (3.4); itera sobre los vértices que se encuentran en el conjunto *nue_color*. La línea (5) agrega vértices recién coloreados a *nue_color*.

En un lenguaje de programación como Pascal hay muchas formas de representar conjuntos. En los capítulos 4 y 5 se estudiarán varias de esas representaciones. En este ejemplo, los conjuntos de vértices se representarán simplemente por otro tipo de datos abstracto, LISTA, que se puede aplicar mediante una lista de enteros terminada con un valor especial *nulo* (para lo cual se podría aplicar el valor 0). Estos enteros podrían almacenarse, por ejemplo, en un arreglo, aunque hay muchas otras formas de representar una LISTA, como se verá en el capítulo 2.

Se puede ahora reemplazar la proposición *for* de la línea (3.2) por un ciclo en el que *w* se inicializa como primer miembro de *nue_color* y cambia al siguiente miembro en cada repetición del ciclo. También se puede realizar el mismo refinamiento con el ciclo *for* de la línea (2) de la figura 1.6. El procedimiento *ávido* revisado

se muestra en la figura 1.8. Después de esto, aún hay refinamientos por hacer, pero se hará una pausa para repasar lo estudiado. □

```

procedure ávido ( var G: GRAFO; var nue_color: LISTA );
{ ávido asigna a nue_color los vértices a los que se
  puede dar el mismo color }
var
  encontrado: boolean;
  v, w: integer;
begin
  nue_color := Ø;
  v := primer vértice no coloreado de G;
  while v <> nulo do begin
    encontrado := false;
    w := primer vértice de nue_color;
    while w <> nulo do begin
      if hay una arista entre v y w en G then
        encontrado := true;
      w := siguiente vértice de nue_color;
    end;
    if encontrado = false do begin
      marca v como coloreado;
      agrega v a nue_color;
    end;
    v := siguiente vértice no coloreado de G
  end
end; { ávido }

```

Fig. 1.8. Procedimiento *ávido* refinado.

Resumen

En la figura 1.9 se representa el proceso de programación tal como se tratará en este libro. La primera etapa es la modelación, mediante un modelo matemático apropiado, como un grafo. En esta etapa, la solución del problema es un algoritmo expresado de manera muy informal.

En la siguiente etapa, el algoritmo se escribe en seudolenguaje, es decir, en una mezcla de construcciones de Pascal y proposiciones menos formales en español. Esta etapa se alcanza sustituyendo el español informal por secuencias de proposiciones cada vez más detalladas, en un proceso denominado *refinamiento por pasos*. En algún punto de este proceso, el programa en seudolenguaje estará suficientemente detallado para que las operaciones que se deban realizar con los distintos tipos de da-

tos estén bien determinadas. Entonces, se crean los tipos de datos abstractos para cada tipo de datos (con excepción de los datos de tipo elemental como los enteros, los reales o las cadenas de caracteres) dando un nombre de procedimiento a cada operación y sustituyendo los usos de las operaciones por invocaciones a los procedimientos correspondientes.

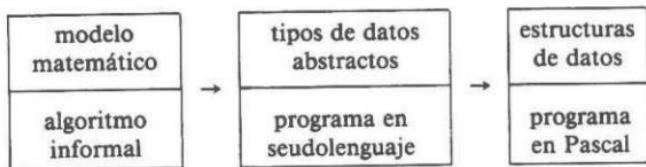


Fig. 1.9. Proceso de solución de problemas.

En la tercera etapa, se elige una aplicación para cada tipo de datos abstracto y se escribe el procedimiento que corresponde a cada operación con ese tipo. También se reemplaza por código Pascal toda proposición informal que quede en el algoritmo en seudolenguaje. El resultado será un programa ejecutable. Despues de depurarlo, será un programa operativo y se espera que por haber aplicado el desarrollo por pasos bosquejado en la figura 1.9, se requiera poca depuración.

1.2 Tipos de datos abstractos

Con seguridad, la mayor parte de los conceptos introducidos en la sección anterior le es familiar al lector desde sus primeros cursos de programación. Quizá la única noción nueva sea la de tipo de datos abstracto; por eso, antes de continuar, se analizará el papel de estos tipos durante el proceso general de diseño de programas. Para comenzar, es útil comparar este concepto con el más familiar, de procedimiento.

Los procedimientos, herramientas esenciales de programación, generalizan el concepto de operador. En vez de limitarse a los operadores incorporados de un lenguaje de programación (suma, resta, etcétera), con el uso de procedimientos, un programador es libre de definir sus propios operadores y aplicarlos a operandos que no tienen por qué ser de tipo fundamental. Un ejemplo de esta aplicación de los procedimientos es una rutina de multiplicación de matrices.

Otra ventaja de los procedimientos es que pueden utilizarse para *encapsular* partes de un algoritmo, localizando en una sección de un programa todas las proposiciones que tienen importancia en relación con cierto aspecto de éste. Un ejemplo de encapsulación es el uso de un procedimiento para leer todas las entradas y verificar su validez. La ventaja de realizar encapsulaciones es que se sabe hacia dónde ir para hacer cambios en el aspecto encapsulado del problema. Por ejemplo, si se decide verificar que todas las entradas sean no negativas, sólo se necesita cambiar unas cuantas líneas de código, y se sabe con exactitud dónde están esas líneas.

Definición de tipo de datos abstracto

Se puede pensar en un *tipo de datos abstracto* (TDA) como en un modelo matemático con una serie de operaciones definidas en ese modelo. Un ejemplo sencillo de TDA son los conjuntos de números enteros con las operaciones de unión, intersección y diferencia. Las operaciones de un TDA pueden tener como operandos no sólo los casos del TDA que se define, sino también otros tipos de operandos, como enteros o casos de otro TDA, y el resultado de una operación puede no ser un caso de ese TDA. Sin embargo, se supone que al menos un operando, o el resultado, de alguna operación pertenece al TDA en cuestión.

Las dos propiedades de los procedimientos mencionadas anteriormente, generalización y encapsulación, son igualmente aplicables a los tipos de datos abstractos. Los TDA son generalizaciones de los tipos de datos primitivos (enteros, reales, etcétera), al igual que los procedimientos son generalizaciones de operaciones primitivas (suma, resta, etcétera). Un TDA encapsula cierto tipo de datos en el sentido de que es posible localizar la definición del tipo y todas las operaciones con ese tipo se pueden localizar en una sección del programa. Así, si se desea cambiar la forma de implantar un TDA, se sabe hacia dónde dirigirse, y revisando una pequeña sección del programa se puede tener la seguridad de que no hay detalles en otras partes que puedan ocasionar errores relacionados con ese tipo de datos. Además, fuera de la sección en la que están definidas las operaciones con el TDA, éste se puede utilizar como si fuese un tipo de datos primitivo, es decir, sin preocuparse por la aplicación. Un inconveniente es que ciertas operaciones pueden implicar más de un TDA, y para que todo marche bien en tales casos, debe hacerse referencia a estas operaciones en las secciones de los dos TDA.

Para ilustrar las ideas básicas considérese el procedimiento *ávido* de la sección anterior (Fig. 1.8), implantado mediante operaciones primitivas en el tipo de datos abstracto LISTA (de enteros). Las operaciones ejecutadas en la LISTA *nue_color* fueron las siguientes:

1. vaciar la lista;
2. obtener el primer miembro de la lista y devolver *nulo* si la lista estaba vacía;
3. obtener el siguiente miembro de la lista y devolver *nulo* si no hay miembro siguiente, y, por último,
4. insertar un entero en la lista.

Existen muchas estructuras de datos que se pueden utilizar para implantar eficientemente a una lista como éstas; este tema se estudiará con profundidad en el capítulo 2. Si en la figura 1.8 se reemplazan esas operaciones por las proposiciones

1. ANULA (*nue_color*);
2. $w := \text{PRIMERO} (\text{nue_color})$;
3. $w := \text{SIGUIENTE} (\text{nue_color})$;
4. INSERTA (*v*, *nue_color*);

entonces se aprecia un aspecto importante de los tipos de datos abstractos. Se puede aplicar un tipo en la forma que se desee sin que los programas, como el de la fi-

gura 1.8, que utilizan objetos de ese tipo sufran cambios; solamente cambian los procedimientos que aplican las operaciones en el tipo.

Volviendo al tipo de datos abstracto GRAFO, se observa la necesidad de las siguientes operaciones:

1. obtener el primer vértice no coloreado;
2. determinar si hay una arista entre dos vértices;
3. marcar un vértice como coloreado, y
4. obtener el siguiente vértice no coloreado.

Es evidente que se necesitan otras operaciones fuera del procedimiento *dvido*, como insertar vértices y aristas en un grafo y marcar todos sus vértices como no coloreados. Hay muchas estructuras de datos que se pueden usar para apoyar grafos con estas operaciones; este tema se estudia en los capítulos 6 y 7.

Es importante resaltar el hecho de que no hay límite para el número de operaciones que se pueden aplicar a diversos casos de un modelo matemático dado. Cada conjunto de operaciones define un TDA distinto. Algunos ejemplos de operaciones que podrían definirse con un tipo de datos abstracto CONJUNTO son los siguientes:

1. ANULA(A). Este procedimiento hace que el valor del conjunto A sea el conjunto vacío.
2. UNION(A, B, C). Este procedimiento toma dos argumentos, A y B , cuyos valores son conjuntos, y hace que el conjunto C tome el valor de la unión de A y B .
3. TAMAÑO(A). Esta función toma un argumento A , cuyo valor es un conjunto, y devuelve un objeto de tipo entero cuyo valor es el número de elementos de A .

La *implantación* de un TDA es la traducción en proposiciones de un lenguaje de programación, de la declaración que define una variable como perteneciente a ese tipo, además de un procedimiento en ese lenguaje por cada operación del TDA. Una implantación elige una *estructura de datos* para representar el TDA; cada estructura de datos se construye a partir de los tipos de datos fundamentales del lenguaje de programación base, utilizando los dispositivos de estructuración de datos disponibles. Los arreglos y las estructuras de registro son dos importantes dispositivos de estructuración de datos de Pascal. Por ejemplo, una implantación posible de la variable S del tipo CONJUNTO sería un arreglo que contuviera los miembros de S .

Una razón importante para afirmar que dos TDA son diferentes si tienen el mismo modelo matemático, pero distintas operaciones, es que lo apropiado de una realización depende en gran medida de las operaciones que se van a realizar. Gran parte de este libro se dedica al examen de algunos modelos matemáticos básicos, como los conjuntos y los grafos, y al desarrollo de las realizaciones preferibles para varios conjuntos de operaciones.

Lo ideal sería escribir programas en lenguajes cuyos tipos de datos y operaciones primitivos estuvieran muy cerca de los modelos y operaciones de los TDA que se utilizarán aquí. En muchos sentidos, Pascal no es el lenguaje más apropiado para implantar varios TDA comunes, pero ninguno de los lenguajes en que un TDA puede declararse más directamente es tan conocido. Consultense las notas bibliográficas del final del capítulo.

1.3 Tipos de datos, estructuras de datos y tipos de datos abstractos

Aunque los términos «tipo de datos» (o simplemente «tipo»), «estructura de datos» y «tipo de datos abstracto» parecen semejantes, su significado es diferente. En un lenguaje de programación, el *tipo de datos* de una variable es el conjunto de valores que ésta puede tomar. Por ejemplo, una variable de tipo booleano puede tomar los valores verdadero o falso, pero ningún otro. Los tipos de datos básicos varían de un lenguaje a otro; en Pascal son entero (*integer*), real (*real*), booleano (*boolean*) y carácter (*char*). Las reglas para construir tipos de datos compuestos a partir de los básicos también varían de un lenguaje a otro; se verá ahora la construcción de esos tipos en Pascal.

Un tipo de datos abstracto es un modelo matemático, junto con varias operaciones definidas sobre ese modelo. Tal como se indicó, en este libro se diseñarán los algoritmos en función de los TDA, pero para implantar un algoritmo en un lenguaje de programación determinado debe hallarse alguna manera de representar los TDA en función de los tipos de datos y los operadores manejados por ese lenguaje. Para representar el modelo matemático básico de un TDA, se emplean *estructuras de datos*, que son conjuntos de variables, quizás de tipos distintos, conectadas entre sí de diversas formas.

El componente básico de una estructura de datos es la *celda*. Se puede representar una celda como una caja capaz de almacenar un valor tomado de algún tipo de datos básico o compuesto. Las estructuras de datos se crean dando nombres a agregados de celdas y (opcionalmente) interpretando los valores de algunas celdas como representantes de conexiones entre celdas (por ejemplo, los apuntadores).

El mecanismo de agregación más sencillo en Pascal y en la mayor parte de los lenguajes de programación es el *arreglo* (unidimensional), que es una sucesión de celdas de un tipo dado al cual se llamará casi siempre «*tipo_celda*». Se puede imaginar un arreglo como una transformación del conjunto índice (como los enteros 1, 2, ..., *n*) al tipo *tipo_celda*. Se puede hacer referencia a una celda de un arreglo dando el nombre del arreglo en unión de un valor tomado del conjunto índice del arreglo. En Pascal, el conjunto índice puede ser un tipo enumerado definido por el programador, como (norte, sur, este, oeste), o un subrango como 1..10. Los valores de las celdas de un arreglo pueden ser de cualquier tipo. Así, la declaración

nombre: array [*tipo_índice*] of *tipo_celda*;

indica que *nombre* es una sucesión de celdas, una por cada valor del tipo *tipo_índice*; el contenido de las celdas puede ser cualquier miembro del tipo *tipo_celda*.

A propósito de esto, Pascal tiene una riqueza considerable de tipos índice. Muchos lenguajes sólo admiten subrangos (conjuntos finitos de enteros consecutivos) como tipos índice. Por ejemplo, en FORTRAN, para indizar un arreglo con letras, debe simularse el efecto utilizando índices enteros, como el índice 1 para representar la 'A', el 2 para la 'B', y así sucesivamente.

Otro mecanismo habitual para agrupar celdas en los lenguajes de programación es la *estructura de registro*. Un *registro* es una celda constituida por un conjunto de celdas llamadas *campos*, que pueden ser de tipos distintos. A menudo, los registros

se agrupan en arreglos; el tipo definido por la agregación de los campos de un registro se convierte en el tipo_celda del arreglo. Por ejemplo, la declaración en Pascal

```
var
  lista_reg: array [1..4] of record
    datos: real;
    siguiente: integer
  end
```

declara que *lista_reg* es un arreglo de cuatro elementos cuyas celdas son registros que tienen dos campos: *datos* y *siguiente*.

Un tercer método de agrupación que se encuentra en Pascal y en otros lenguajes es el *archivo* (*file*). Al igual que un arreglo unidimensional, un archivo es una sucesión de valores de un tipo particular. La diferencia está en que un archivo no tiene tipo índice; los elementos sólo son accesibles en el orden en que aparecen en el archivo. En contraste, tanto el arreglo como el registro son estructuras de «acceso aleatorio», lo cual significa que el tiempo necesario para acceder a un componente de un arreglo o de un registro es independiente del valor del índice del arreglo o del selector de campo. La ventaja de agrupar por archivos en lugar de hacerlo por arreglos es que el número de elementos de un archivo puede ser ilimitado y variable con el tiempo.

Apuntadores y cursores

Además de las características de agrupación-celdas de un lenguaje de programación, es posible representar relaciones entre celdas mediante apuntadores y cursores. Un *apuntador* es una celda cuyo valor indica o señala a otra. Cuando se representan gráficamente estructuras de datos, el hecho de que una celda *A* sea un apuntador a la celda *B* se indica con una flecha de *A* a *B*.

En Pascal, se puede crear una variable *ap* que apunte a celdas de un tipo determinado, como tipo_celda, mediante la declaración

```
var
  ap:  $\dagger$  tipo_celda
```

En Pascal se utiliza una flecha ascendente posfixa como operador de desreferenciación. Así, la expresión *ap* \dagger denota el valor (de tipo tipo_celda) de la celda a la que apunta *ap*.

Un *cursor* es una celda de valor entero que se utiliza como apuntador a un arreglo. Como métodos de conexión, cursores y apuntadores son en esencia lo mismo, con la diferencia de que un cursor se puede utilizar en lenguajes que, como FORTRAN, carecen de los tipos apuntadores explícitos que tiene Pascal. Al tratar una celda de tipo entero como un valor índice para un arreglo, se hace que esa celda apunte a una celda del arreglo. Desafortunadamente, esta técnica sólo sirve para apuntar a celdas contenidas en un arreglo; no hay manera razonable de interpretar un entero como un «apuntador» a una celda que no sea parte de un arreglo.

En este texto, se dibujará una flecha desde una celda cursor hacia la celda a la

que «apunta». En ocasiones, se mostrará también el entero de la celda cursor, para recordar que no se trata de un verdadero apuntador. El lector debe observar que el mecanismo apuntador de Pascal es tal que a las celdas de los arreglos sólo se puede «apuntar» con cursores, y no con verdaderos apuntadores. Ciertos lenguajes, como PL/I o C, permiten que los componentes de un arreglo sean apuntados tanto por cursores como por apuntadores, mientras que en otros, como FORTRAN o ALGOL, no existe el tipo apuntador y sólo es posible usar cursores.

Ejemplo 1.3. En la figura 1.10 se observa una estructura de datos con dos partes, constituida por una cadena de celdas que contienen cursores al arreglo *lista_reg* definido antes. El propósito del campo *siguiente* de *lista_reg* es apuntar a otro registro del arreglo. Por ejemplo, *lista_reg* [4]. *siguiente* es 1, por tanto, el registro 4 va seguido del registro 1. Si se supone que el registro 4 es el primero, el campo *siguiente* de *lista_reg* ordena los registros en la secuencia 4, 1, 3, 2. Obsérvese que el campo *siguiente* del registro 2 es 0, lo cual indica que no existe un registro que lo siga. Una convención útil que se adoptará en este libro consiste en usar 0 como un «apuntador NIL» cuando se trabaje con cursores. Esto sólo da buenos resultados si también se adopta la convención de que los arreglos a los que «apuntan» cursores han de iniciarse a partir de 1, no de 0.

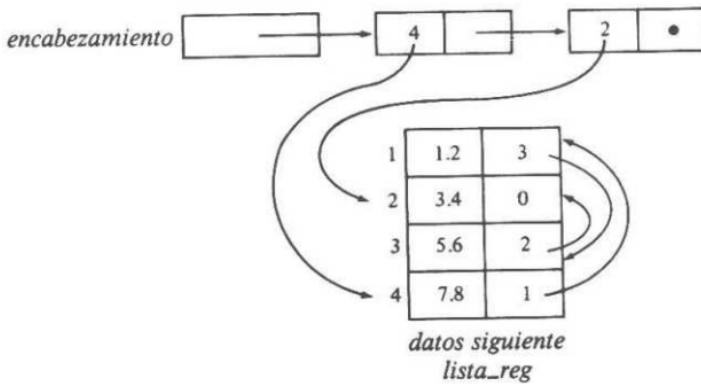


Fig. 1.10. Ejemplo de una estructura de datos.

Las celdas de la cadena de registros de la figura 1.10 son del tipo

```
type
  tipo_reg = record
    cursor: integer;
    ap: † tipo_reg
  end
```

A la cadena apunta una variable llamada *encabezamiento*, que es de tipo \uparrow tipo_reg; *encabezamiento* apunta a un registro anónimo de tipo tipo_reg \dagger . Ese registro tiene valor 4 en el campo *cursor*; este 4 se considera como un índice del arreglo *lista_reg*, y en el campo *ap* tiene un verdadero apuntador a otro registro anónimo. Este último tiene un índice en su campo *cursor*, que indica la posición 2 en *lista_reg*, y un apuntador *nil* en su campo *ap*. \square

1.4 Tiempo de ejecución de un programa

Cuando se resuelve un problema, con frecuencia hay necesidad de elegir entre varios algoritmos. ¿Cómo se debe elegir? Hay dos objetivos que suelen contradecirse:

1. Que el algoritmo sea fácil de entender, codificar y depurar.
2. Que el algoritmo use eficientemente los recursos del computador y, en especial, que se ejecute con la mayor rapidez posible.

Cuando se escribe un programa que se va a usar una o pocas veces, el primer objetivo es el más importante. En tal caso, es muy probable que el costo del tiempo de programación exceda en mucho al costo de ejecución del programa, de modo que el costo a optimizar es el de escritura del programa. En cambio, cuando se presenta un problema cuya solución se va a utilizar muchas veces, el costo de ejecución del programa puede superar en mucho al de escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. Entonces, es más ventajoso, desde el punto de vista económico, realizar un algoritmo complejo siempre que el tiempo de ejecución del programa resultante sea significativamente menor que el de un programa más evidente. Y aun en situaciones como esa, quizás sea conveniente implantar primero un algoritmo simple, con el objeto de determinar el beneficio real que se obtendría escribiendo un programa más complicado. En la construcción de un sistema complejo, a menudo es deseable implantar un prototipo sencillo en el cual se puedan efectuar simulaciones y mediciones antes de dedicarse al diseño definitivo. De esto se concluye que un programador no sólo debe estar al tanto de las formas de lograr que un programa se ejecute con rapidez, sino que también debe saber cuándo aplicar esas técnicas y cuándo ignorarlas.

Medición del tiempo de ejecución de un programa

El tiempo de ejecución de un programa depende de factores como:

1. los datos de entrada al programa,
2. la calidad del código generado por el compilador utilizado para crear el programa objeto,

\dagger El registro no tiene nombre conocido porque se creó con una llamada *new* (*encabezamiento*) cuyo efecto fue que *encabezamiento* apuntara a ese registro recién creado. No obstante, en el interior de la máquina existe una dirección de memoria que puede emplearse para localizar la celda.

3. la naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa, y
4. la complejidad de tiempo del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución de un programa debe definirse como una función de la entrada. Con frecuencia, el tiempo de ejecución no depende de la entrada exacta, sino sólo de su «tamaño». Un buen ejemplo de esto es el proceso conocido como *clasificación (sorting)*, que se analizará en el capítulo 8. En un problema de clasificación, se da como entrada una lista de elementos para ordenar a fin de producir como salida otra lista con los mismos elementos, pero clasificados de menor a mayor o viceversa. Por ejemplo, dada la lista 2, 1, 3, 1, 5, 8 como entrada, se desea producir la lista 1, 1, 2, 3, 5, 8 como salida. Se dice entonces que los elementos de la segunda lista están en *orden de menor a mayor*. La medida natural del tamaño de la entrada a un programa de clasificación es el número de elementos a ordenar o, en otras palabras, la longitud de la lista de entrada. En general, la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.

Se acostumbra, pues, a denominar $T(n)$ al tiempo de ejecución de un programa con una entrada de tamaño n . Por ejemplo, algunos programas pueden tener un tiempo de ejecución $T(n) = cn^2$, donde c es una constante. Las unidades de $T(n)$ se dejan sin especificar, pero se puede considerar a $T(n)$ como el número de instrucciones ejecutadas en un computador idealizado.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ella. En este caso se define $T(n)$ como el tiempo de ejecución del *peor caso*, es decir, el máximo valor del tiempo de ejecución para entradas de tamaño n . También suele considerarse $T_{\text{prom}}(n)$, el valor medio del tiempo de ejecución de todas las entradas de tamaño n . Aunque $T_{\text{prom}}(n)$ parece una medida más razonable, a menudo es engañoso suponer que todas las entradas son igualmente probables. En la práctica, casi siempre es más difícil determinar el tiempo de ejecución promedio que el del peor caso, pues el análisis se hace intratable en matemáticas, y la noción de entrada «promedio» puede carecer de un significado claro. Así pues, se utilizará el tiempo de ejecución del peor caso como medida principal de la complejidad de tiempo, aunque se mencionará la complejidad del caso promedio cuando pueda hacerse en forma significativa.

Considérense ahora las observaciones 2 y 3 anteriores: a saber, que el tiempo de ejecución depende del compilador y de la máquina utilizados. Este hecho implica que no es posible expresar $T(n)$ en unidades estándares de tiempo, como segundos. Antes bien, sólo se pueden hacer observaciones como «el tiempo de ejecución de tal algoritmo es proporcional a n^2 », sin especificar la constante de proporcionalidad, pues depende en gran medida del compilador, la máquina y otros factores.

Notación asintótica («o mayúscula» y «omega mayúscula»)

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como *notación asintótica* («o mayúscula»). Por ejemplo, de-

cir que el tiempo de ejecución $T(n)$ de un programa es $O(n^2)$, que se lee « α mayúscula de n al cuadrado» o tan sólo « α de n al cuadrado», significa que existen constantes enteras positivas c y n_0 tales que para n mayor o igual que n_0 , se tiene que $T(n) \leq cn^2$.

Ejemplo 1.4. Supóngase que $T(0) = 1$, $T(1) = 4$, y en general $T(n) = (n+1)^2$. Entonces se observa que $T(n)$ es $O(n^2)$ cuando $n_0 = 1$ y $c = 4$; es decir, para $n \geq 1$, se tiene que $(n+1)^2 \leq 4n^2$, que es fácil de demostrar. Obsérvese que no se puede hacer $n_0 = 0$, pues $T(0) = 1$ no es menor que $c0^2 = 0$ para ninguna constante c . \square

A continuación, se supondrá que todas las funciones del tiempo de ejecución están definidas en los enteros no negativos, y que sus valores son siempre no negativos, pero no necesariamente enteros. Se dice que $T(n)$ es $O(f(n))$ si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ cuando $n \geq n_0$. Cuando el tiempo de ejecución de un programa es $O(f(n))$, se dice que tiene *velocidad de crecimiento* $f(n)$.

Ejemplo 1.5. La función $T(n) = 3n^3 + 2n^2$ es $O(n^3)$. Para comprobar esto, sean $n_0 = 0$ y $c = 5$. Entonces, el lector puede mostrar que para $n \geq 0$, $3n^3 + 2n^2 \leq 5n^3$. También se podría decir que $T(n)$ es $O(n^4)$, pero sería una aseveración más débil que decir que $T(n)$ es $O(n^3)$.

A manera de segundo ejemplo, se demostrará que la función 3^n no es $O(2^n)$. Para esto, supóngase que existen constantes n_0 y c tales que para todo $n \geq n_0$, se tiene que $3^n \leq c2^n$. Entonces, $c \geq (3/2)^n$ para cualquier valor de $n \geq n_0$. Pero $(3/2)^n$ se hace arbitrariamente grande conforme n crece y, por tanto, ninguna constante c puede ser mayor que $(3/2)^n$ para toda n . \square

Cuando se dice que $T(n)$ es $O(f(n))$, se sabe que $f(n)$ es una cota superior para la velocidad de crecimiento de $T(n)$. Para especificar una cota inferior para la velocidad de crecimiento de $T(n)$, se usa la notación $T(n)$ es $\Omega(g(n))$, que se lee « $T(n)$ es omega mayúscula de $g(n)$ » o simplemente « $T(n)$ es omega de $g(n)$ », lo cual significa que existe una constante c tal que $T(n) \geq cg(n)$ para un número infinito de valores de n .[†]

Ejemplo 1.6. Para verificar que la función $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$, sea $c = 1$. Entonces, $T(n) \geq cn^3$ para $n = 0, 1, \dots$

En otro ejemplo, sea $T(n) = n$ para $n \geq 1$ impar, y sea $T(n) = n^2/100$ para $n \geq 0$ par. Para verificar que $T(n)$ es $\Omega(n^2)$, sea $c = 1/100$ y considérese el conjunto infinito de valores de n : $n = 0, 2, 4, 6, \dots$ \square

La «tiránica» de la velocidad de crecimiento

Se supondrá que es posible evaluar programas comparando sus funciones de tiempo de ejecución sin considerar las constantes de proporcionalidad. Según este supuesto, un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con tiempo de ejecu-

[†] Obsérvese la asimetría existente entre las notaciones « α mayúscula» y «omega mayúscula». La razón de que esta asimetría sea con frecuencia útil, es que muchas veces un algoritmo es rápido con muchas entradas pero no con todas. Por ejemplo, para probar si la entrada es de longitud prima, existen algoritmos que se ejecutan muy rápido si la longitud es par, de modo que para el tiempo de ejecución no es posible obtener una buena cota inferior que sea válida para toda $n \geq n_0$.

ción $O(n^3)$ por ejemplo. Sin embargo, además de los factores constantes debidos al compilador y a la máquina, existe un factor constante debido a la naturaleza del programa mismo. Es posible, por ejemplo, que con una combinación determinada de compilador y máquina, el primer programa tarde $100n^2$ milisegundos, mientras el segundo tarda $5n^3$ milisegundos. En este caso, ¿no es preferible el segundo programa al primero?

La respuesta a esto depende del tamaño de las entradas que se espera que procesen los programas. Para entradas de tamaño $n < 20$, el programa con tiempo de ejecución $5n^3$ será más rápido que el de tiempo de ejecución $100n^2$. Así pues, si el programa se va a ejecutar principalmente con entradas pequeñas, será preferible el programa cuyo tiempo de ejecución es $O(n^3)$. No obstante, conforme n crece, la razón de los tiempos de ejecución, que es $5n^3/100n^2 = n/20$, se hace arbitrariamente grande. Así, a medida que crece el tamaño de la entrada, el programa $O(n^3)$ requiere un tiempo significativamente mayor que el programa $O(n^2)$. Pero si hay algunas entradas grandes en los problemas para cuya solución se están diseñando estos dos programas, será mejor optar por el programa cuyo tiempo de ejecución tiene la menor velocidad de crecimiento.

Otro motivo para al menos considerar programas cuyas velocidades de crecimiento sean lo más pequeñas posibles, es que básicamente es la velocidad del crecimiento quien determina el tamaño de problema que se puede resolver en un computador. Para decirlo de otro modo, conforme los computadores se hacen más veloces, también aumentan los deseos del usuario de resolver con ellos problemas más grandes; sin embargo, a menos que un programa tenga una velocidad de crecimiento baja, como $O(n)$ u $O(n\log n)$, un incremento modesto en la rapidez de un computador no influye gran cosa en el tamaño del problema más grande que es posible resolver en una cantidad fija de tiempo.

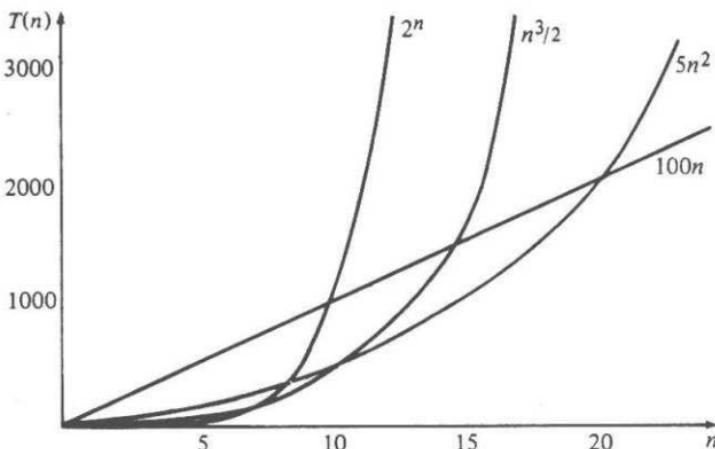


Fig. 1.11. Tiempos de ejecución de cuatro programas.

Ejemplo 1.7. En la figura 1.11 se pueden ver los tiempos de ejecución de cuatro programas de distintas complejidades de tiempo, medidas en segundos, para una combinación determinada de compilador y máquina. Supóngase que se dispone de 1000 segundos, o alrededor de 17 minutos, para resolver un problema determinado. ¿Qué tamaño de problema se puede resolver? Como se puede ver en la segunda columna de la figura 1.12, los cuatro algoritmos pueden resolver problemas de un tamaño similar en 10^3 segundos.

Supóngase ahora que se adquiere una máquina que funciona diez veces más rápido sin costo adicional. Entonces, con el mismo costo, es posible dedicar 10^4 segundos a la solución de problemas que antes requerían 10^3 segundos. El tamaño máximo de problema que es posible resolver ahora con cada uno de los cuatro programas se muestra en la tercera columna de la figura 1.12, y la razón entre los valores de las columnas segunda y tercera se muestra en la cuarta. Se observa que un aumento del 1000% en la velocidad del computador origina apenas un incremento del 30% en el tamaño de problema que se puede resolver con el programa $O(2^n)$. Los aumentos adicionales de un factor de diez en la rapidez del computador a partir de este punto originan aumentos porcentuales aún menores en el tamaño de los problemas. De hecho, el programa $O(2^n)$ sólo puede resolver problemas pequeños, independientemente de la rapidez del computador.

Tiempo de ejecución $T(n)$	Tamaño máximo de problema para 10^3 seg	Tamaño máximo de problema para 10^4 seg	Incremento en el tamaño máximo de problema
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Fig. 1.12. Efecto de multiplicar por diez la velocidad del computador.

En la tercera columna de la figura 1.12 se puede apreciar una superioridad evidente del programa $O(n)$; éste permite un aumento del 1000% en el tamaño de problema para un incremento del 1000% en la rapidez del computador. Se observa que los programas $O(n^3)$ y $O(n^2)$ permiten aumentos de 230% y 320%, respectivamente, en el tamaño de problema, para un incremento del 1000% en la rapidez del computador. Estas razones se mantendrán vigentes para incrementos adicionales en la rapidez del computador. □

Mientras exista la necesidad de resolver problemas cada vez más grandes, se producirá una situación casi paradójica. A medida que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así, la importancia del descubrimiento y el empleo de algoritmos eficientes (aquellos cuyas velocidades de crecimiento sean pequeñas) irá en aumento, en lugar de disminuir.

Aspectos importantes

Es necesario subrayar de nuevo que la velocidad de crecimiento del tiempo de ejecución del peor caso no es el único criterio, ni necesariamente el más importante, para evaluar un algoritmo o un programa. A continuación se presentan algunas condiciones en las cuales el tiempo de ejecución de un programa se puede ignorar en favor de otros factores.

1. Si un programa se va a utilizar sólo algunas veces, el costo de su escritura y depuración es el dominante, de manera que el tiempo de ejecución raramente influirá en el costo total. En ese caso debe elegirse el algoritmo que sea más fácil de aplicar correctamente.
2. Si un programa se va a ejecutar sólo con entradas «pequeñas», la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante de la fórmula del tiempo de ejecución. Determinar qué es una entrada «pequeña», depende de los tiempos de ejecución exactos de los algoritmos implicados. Hay algoritmos (como el de multiplicación de enteros de Schonhage y Strassen [1971]) que son asintóticamente los más eficientes para sus problemas, pero nunca se han llevado a la práctica ni siquiera con los problemas más grandes, debido a que la constante de proporcionalidad es demasiado grande comparada con la de otros algoritmos menos «eficientes».
3. Un algoritmo eficiente pero complicado puede no ser apropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor. Se espera que al difundir el conocimiento de las principales técnicas de diseño de algoritmos eficientes, se podrán utilizar libremente algoritmos más complejos, pero debe considerarse la posibilidad de que un programa resulte inútil debido a que nadie entiende sus sutiles y eficientes algoritmos.
4. Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lento, lo cual puede anular la eficiencia.
5. En los algoritmos numéricos, la precisión y la estabilidad son tan importantes como la eficiencia.

1.5 Cálculo del tiempo de ejecución de un programa

Calcular el tiempo de ejecución de un programa arbitrario, aunque sólo sea una aproximación a un factor constante, puede ser un problema matemático complejo. Sin embargo, en la práctica esto suele ser más sencillo; basta con aplicar unos cuantos principios básicos. Antes de presentar estos principios, es importante aprender a sumar y a multiplicar en notación asintótica.

Supóngase que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos fragmentos de programa P_1 y P_2 , y que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$. Entonces $T_1(n) + T_2(n)$, el tiempo de ejecución de P_1 seguido de P_2 , es $O(\max(f(n), g(n)))$. Para saber por qué, obsérvese que para algunas constantes, c_1 , c_2 , n_1 y n_2 , si $n \geq n_1$, entonces

$T_1(n) \leq c_1 f(n)$, y si $n \geq n_0$, entonces $T_2(n) \leq c_2 g(n)$. Sea $n_0 = \max(n_1, n_2)$. Si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. De aquí se concluye que si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n), g(n))$; por tanto, $T_1(n) + T_2(n)$ es $O(\max(f(n), g(n)))$.

Ejemplo 1.8. La regla de la suma anterior puede usarse para calcular el tiempo de ejecución de una secuencia de pasos de programa, donde cada paso puede ser un fragmento de programa arbitrario con ciclos y ramificaciones. Supóngase que se tienen tres pasos cuyos tiempos de ejecución son, respectivamente, $O(n^2)$, $O(n^3)$ y $O(n \log n)$. Entonces, el tiempo de ejecución de los dos primeros pasos ejecutados en secuencia es $O(\max(n^2, n^3))$ que es $O(n^3)$. El tiempo de ejecución de los tres juntos es $O(\max(n^3, n \log n))$, que es $O(n^3)$. □

En general, el tiempo de ejecución de una secuencia fija de pasos, dentro de un factor constante, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución. En raras ocasiones dos pasos pueden tener tiempos de ejecución *incomensurables* (ninguno es mayor que el otro, ni son iguales). Por ejemplo, puede haber pasos con tiempos de ejecución $O(f(n))$ y $O(g(n))$, donde

$$f(n) = \begin{cases} n^4 & \text{si } n \text{ es par} \\ n^2 & \text{si } n \text{ es impar} \end{cases} \quad g(n) = \begin{cases} n^2 & \text{si } n \text{ es par} \\ n^3 & \text{si } n \text{ es impar} \end{cases}$$

En tales casos, la regla de la suma debe aplicarse directamente; en el ejemplo, el tiempo de ejecución es $O(\max(f(n), g(n)))$, esto es, n^4 si n es par y n^3 si n es impar.

Otra observación útil sobre la regla de la suma es que si $g(n) \leq f(n)$ para toda n mayor que una constante n_0 , entonces $O(f(n) + g(n))$ es lo mismo que $O(f(n))$. Por ejemplo, $O(n^2 + n)$ es lo mismo que $O(n^2)$.

La regla del producto es la siguiente: si $T_1(n)$ y $T_2(n)$ son $O(f(n))$ y $O(g(n))$, respectivamente, entonces $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Se aconseja probar este hecho con las mismas ideas que se utilizaron para probar la regla de la suma. Según la regla del producto, $O(cf(n))$ significa lo mismo que $O(f(n))$ si c es una constante positiva cualquiera. Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Antes de pasar a las reglas generales de análisis de los tiempos de ejecución de los programas, se presentará un ejemplo sencillo para proporcionar una visión general del proceso.

Ejemplo 1.9. Considérese el programa de clasificación *burbuja* de la figura 1.13, que ordena un arreglo de enteros de menor a mayor. El efecto total de cada recorrido del ciclo interno de las proposiciones (3) a (6), es hacer que el menor de los elementos suba hasta el principio del arreglo.

El número n de elementos que se van a clasificar es la medida apropiada del tamaño de la entrada. La primera observación que se hace es que cada proposición de asignación toma cierta cantidad constante de tiempo, independiente del tamaño de la entrada. Esto significa que las proposiciones (4), (5) y (6) toman tiempo $O(1)$ cada una. Obsérvese que $O(1)$ es la notación «o mayúscula» de una «cantidad constante». Por la regla de la suma, el tiempo de ejecución combinado de este grupo de proposiciones es $O(\max(1, 1, 1)) = O(1)$.

Ahora deben tenerse en cuenta las proposiciones condicionales y las de control de ciclos. Las proposiciones **if** y **for** están anidadas unas dentro de otras, de modo

```

procedure burbuja ( var A: array [1..n] of integer );
{ burbuja clasifica el arreglo A de menor a mayor }
var
  i, j, temp: integer;
begin
  for i := 1 to n-1 do
    for j := n downto i + 1 do
      if A[j-1] > A[j] then begin
        { intercambia A[j-1] y A[j] }
        temp := A[j-1];
        A[j-1] := A[j];
        A[j] := temp
      end
    end; { burbuja }
  
```

Fig. 1.13. Clasificación burbuja.

que se debe ir de dentro hacia fuera para obtener el tiempo de ejecución del grupo condicional y de cada ciclo. En cuanto a la proposición **if**, la prueba de la condición requiere tiempo $O(1)$. No se sabe si el cuerpo de la proposición **if** (líneas (4) a (6)) se ejecutará, pero dado que se busca el tiempo de ejecución del peor caso, se supone lo peor, esto es, que se ejecute. Por tanto, el grupo **if** de las proposiciones (3) a (6) requiere tiempo $O(1)$.

Así, siguiendo hacia fuera, se llega al ciclo **for** de las líneas (2) a (6). La regla general para un ciclo es que el tiempo de ejecución total resulta de sumar, en cada iteración, los tiempos empleados en ejecutar el cuerpo del ciclo en esa iteración. Debe acumularse al menos $O(1)$ por cada iteración para justificar el incremento del índice, con el fin de verificar si se alcanzó el límite y para saltar de vuelta al principio del ciclo. Para el ciclo de las líneas (2) a (6), el cuerpo tarda tiempo $O(1)$ en cada iteración. El número de iteraciones es $n-i$, de modo que, por la regla del producto, el total de tiempo invertido en el ciclo de las líneas (2) a (6) es $O((n-i) \times 1)$, o sea, $O(n-i)$.

Se sigue ahora con el ciclo externo, el cual contiene todas las proposiciones ejecutables del programa. La proposición (1) se ejecuta $n-1$ veces, de manera que el tiempo total de ejecución del programa tiene como cota superior una constante multiplicada por

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = n^2/2 - n/2$$

que es $O(n^2)$. Por tanto, para ejecutar el programa de la figura 1.13 se necesita un tiempo proporcional al cuadrado del número de elementos que se van a clasificar.

En el capítulo 8 se presentarán programas de clasificación cuyo tiempo de ejecución es $O(n \log n)$, que es considerablemente menor, dado que $\log n$ † es mucho menor que n para valores grandes de n . □

Antes de proseguir con algunas reglas generales de análisis, recuérdese que la determinación de una cota superior precisa para el tiempo de ejecución de un programa, unas veces es sencilla, pero otras puede ser un desafío intelectual profundo. No existe un conjunto completo de reglas para analizar programas; en este libro sólo se proporcionarán ciertas sugerencias e ilustrarán algunos de sus aspectos más sutiles mediante ejemplos.

Se enumerarán ahora algunas reglas generales para el análisis de programas. En general, el tiempo de ejecución de una proposición o de un grupo de ellas puede tener como parámetros el tamaño de la entrada, una o más variables, o ambas cosas. El único parámetro permisible para el tiempo de ejecución del programa completo es n , el tamaño de la entrada.

1. El tiempo de ejecución de cada proposición de asignación (lectura y escritura), por lo común puede tomarse como $O(1)$. Hay unas cuantas excepciones, como en PL/I, donde una asignación puede implicar matrices arbitrariamente grandes, y en cualquier lenguaje donde se permitan llamadas a funciones en las proposiciones de asignación.
2. El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Esto es, el tiempo de ejecución de una secuencia es, dentro de un factor constante, el máximo tiempo de ejecución de una proposición de la secuencia.
3. El tiempo de ejecución de una proposición condicional *if* es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición. El tiempo para evaluar la condición, por lo general, es $O(1)$. El tiempo para una construcción *if-then-else* es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la condición es falsa.
4. El tiempo para ejecutar un ciclo es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser $O(1)$). A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo, pero, por seguridad, debe considerarse cada iteración por separado. Por lo común, se conoce con certeza el número de iteraciones, pero en ocasiones no es posible determinarlo con precisión. Incluso puede ocurrir que un programa no sea un algoritmo y que no exista un límite al número de iteraciones de ciertos ciclos.

† A menos que se especifique lo contrario, todos los logaritmos son de base 2. Obsérvese que $O(\log n)$ no depende de la base del logaritmo, puesto que $\log_b n = c \log_a n$, donde $c = \log_a b$.

Llamadas a procedimientos

Si se tiene un programa con procedimientos que no son recursivos, es posible calcular el tiempo de ejecución de los distintos procedimientos, uno a la vez, partiendo de aquellos que no llaman a otros. (Recuérdese que la invocación a una función debe considerarse una «llamada».) Debe haber al menos un procedimiento con esa característica, a menos que como mínimo un procedimiento sea recursivo. Después, puede evaluarse el tiempo de ejecución de los procedimientos que sólo llaman a procedimientos que no hacen llamadas, usando los tiempos de ejecución de los procedimientos llamados evaluados antes. Se continúa el proceso evaluando el tiempo de ejecución de cada procedimiento después de haber evaluado los tiempos correspondientes a los procedimientos que llama.

Si hay procedimientos recursivos, no es posible ordenar las evaluaciones de modo que cada una utilice sólo evaluaciones ya realizadas. Lo que se debe hacer ahora es asociar a cada procedimiento recursivo una función de tiempo desconocida $T(n)$, donde n mide el tamaño de los argumentos del procedimiento. Luego se puede obtener una *recurrencia* para $T(n)$, es decir, una ecuación para $T(n)$ en función de $T(k)$ para varios valores de k .

Se conocen ya técnicas para resolver varias clases de recurrencias; algunas de ellas se presentarán en el capítulo 9. Aquí se mostrará cómo analizar un programa recursivo sencillo.

Ejemplo 1.10. La figura 1.14 muestra un programa recursivo para calcular $n!$, que es el producto de todos los enteros de 1 a n inclusive.

Una medida de tamaño apropiado para esta función es el valor de n . Sea $T(n)$ el tiempo de ejecución para $\text{fact}(n)$. El tiempo de ejecución para las líneas (1) y (2) es $O(1)$, y para la línea (3) es $O(1) + T(n-1)$. Por tanto, para ciertas constantes c y d ,

$$T(n) = \begin{cases} c + T(n-1) & \text{si } n > 1 \\ d & \text{si } n \leq 1 \end{cases} \quad (1.1)$$

```
function fact ( n: integer ) : integer;
  {fact(n) calcula n!}
  begin
    (1)      if n <= 1 then
              fact := 1
            else
              fact := n * fact(n-1)
    end; {fact}
```

Fig. 1.14. Programa recursivo para calcular factoriales.

Suponiendo que $n > 2$, se puede desarrollar $T(n-1)$ en (1.1) para obtener

$$T(n) = 2c + T(n-2) \text{ si } n > 2$$

Esto es, $T(n-1) = c + T(n-2)$, como se puede ver al sustituir n por $n-1$ en (1.1). Así pues, es posible reemplazar $T(n-1)$ con $c + T(n-2)$ en la ecuación $T(n) = c + T(n-1)$. Despues, se puede usar (1.1) para desarrollar $T(n-2)$, con lo que se obtiene

$$T(n) = 3c + T(n-3) \quad \text{si } n > 3$$

y así sucesivamente. En general,

$$T(n) = ic + T(n-i) \quad \text{si } n > i$$

Por último, cuando $i = n-1$, se obtiene

$$T(n) = c(n-1) + T(1) = c(n-1) + d \quad (1.2)$$

Por (1.2) se concluye que $T(n)$ es $O(n)$. Es importante observar que en este análisis se ha supuesto que la multiplicación de dos enteros es una operación de tiempo $O(1)$. En la práctica, no obstante, no se puede emplear el programa de la figura 1.14 para calcular $n!$ cuando los valores de n son grandes, pues el tamaño de los enteros que se calculen excederá del tamaño de palabra de la máquina en cuestión. \square

El método general para resolver una ecuación de recurrencia, tal como se tipifica en el ejemplo 1.10, consiste en reemplazar en forma repetida términos $T(k)$ del lado derecho de la ecuación por el lado derecho completo, donde k se reemplaza por n , hasta obtener una fórmula en la que T no aparezca en el lado derecho como en (1.2). A menudo es necesario calcular la suma de una sucesión o, si no es posible encontrar la suma exacta, obtener una cota superior cercana para la suma a fin de hallar una cota superior para $T(n)$.

Programas con proposiciones GOTO

Hasta ahora, al analizar el tiempo de ejecución de un programa, se supuso de manera tácita que el flujo de control dentro de un procedimiento estaba determinado por construcciones de ciclos y de ramificación. Esto sirvió como base en la determinación del tiempo de ejecución de grupos de proposiciones cada vez más grandes, al suponer que sólo se necesitaba la regla de la suma para agrupar secuencias de proposiciones. Sin embargo, las proposiciones **goto** (proposiciones de transferencia incondicional de control) hacen más complejo el agrupamiento lógico de las proposiciones de un programa. Por esta razón, las proposiciones **goto** deberían evitarse, pero Pascal carece de proposiciones para salir de un ciclo o terminarlo en forma anormal (como *break* y *continue*), por lo que con frecuencia se utiliza **goto** para estos fines.

Para manejar proposiciones **goto** que realicen saltos de un ciclo a código que con seguridad está después del ciclo, se sugiere el siguiente enfoque. (Por lo general,

ésta es la única clase de *goto* que está justificada.) Puesto que es probable que se ejecute el *goto* en forma condicional dentro del ciclo, se puede suponer que nunca se efectúa. Debido a que el *goto* transfiere el control a una proposición que se ejecutará después de terminado el ciclo, esta suposición resulta conservadora; no es posible subestimar el tiempo de ejecución del peor caso si se presume que el ciclo se ejecuta por completo. Sin embargo, es raro el programa en el que ignorar el *goto* es tan conservador que puede llevar a sobreestimar la velocidad de crecimiento del tiempo de ejecución del programa para el peor caso. Obsérvese que si se presentara un *goto* que transfiriera el control a código ejecutado con anterioridad, no sería posible ignorarlo sin correr riesgos, pues ese *goto* podría crear un ciclo que consumiera la mayor parte del tiempo de ejecución.

No por esto debe pensarse que el uso de proposiciones *goto* hacia atrás en sí mismo hace que los tiempos de ejecución sean imposibles de analizar. El enfoque para analizar tiempos de ejecución, que se describió en esta sección, funcionará bien en la medida en que los ciclos de un programa tengan una estructura razonable, es decir, que cualquier par de ciclos sea siempre disjunto o anidado. (No obstante, es responsabilidad del analizador descubrir la estructura de los ciclos.) Así pues, no debe dudarse en aplicar estos métodos de análisis de programas a lenguajes como FORTRAN, donde los *goto* son esenciales, pero los programas tienden a tener una estructura de ciclos razonable.

Análisis de un seudoprograma

Si se conoce la velocidad de crecimiento del tiempo necesario para ejecutar proposiciones informales en español, es posible analizar seudoprogramas como si fueran programas reales. Sin embargo, muchas veces no se conoce el tiempo que demandarán las partes no completamente terminadas de un seudoprograma. Por ejemplo, si se tiene un seudoprograma donde las únicas partes no terminadas son operaciones con TDA, puede elegirse una de las varias implantaciones posibles para un TDA y el tiempo de ejecución total puede depender en gran medida de la realización elegida. Por supuesto, una de las razones en favor de la escritura de programas con TDA es que permite considerar los pros y los contras entre los tiempos de ejecución de varias operaciones que resultan de efectuar distintas realizaciones.

Para analizar seudoprogramas que contengan proposiciones en algún lenguaje de programación y llamadas a procedimientos aún no implantados, tales como operaciones con TDA, se calcula el tiempo de ejecución como una función de los tiempos de ejecución no especificados de esos procedimientos. El tiempo de ejecución de un procedimiento obtiene sus parámetros mediante el «tamaño» de su argumento (o argumentos). Así como sucede con el «tamaño de la entrada», la medida apropiada para el tamaño de un argumento es decisión de quien hace el análisis. Si el procedimiento es una operación con un TDA, el modelo matemático subyacente a menudo indica la noción lógica de tamaño. Por ejemplo, si el TDA está basado en conjuntos, el número de elementos de los conjuntos es con frecuencia la noción correcta de tamaño. En los capítulos siguientes se verán muchos ejemplos de análisis del tiempo de ejecución de seudoprogramas.

1.6 Buenas prácticas de programación

Hay un buen número de ideas que se deberían tener presente al diseñar un algoritmo e implantarlo como programa. Estas ideas a menudo parecen perogrulladas, porque sólo se pueden apreciar en toda su extensión a través de su empleo acertado en situaciones prácticas, más que en el desarrollo de una teoría. A pesar de esto, son lo bastante importantes para repetirlas aquí. El lector debe observar con detenimiento la aplicación de estas ideas en los programas diseñados en este libro, así como buscar oportunidades para ponerlas en práctica en sus propios programas.

1. *Planificar el diseño de un programa.* En la sección 1.1 se mostró cómo diseñar un programa partiendo de un esbozo informal del algoritmo, con el diseño de un seudoprograma y luego con el refinamiento gradual del seudocódigo hasta convertirlo en ejecutable. Esta estrategia de esbozar, y después detallar, tiende a la producción de un programa final más organizado que facilita su depuración y mantenimiento.
2. *Encapsular.* Empléense procedimientos y diversos TDA para que el código asociado a cada operación importante y a cada tipo de datos quede colocado en un solo lugar del listado del programa. Después, si es necesario hacer cambios, la sección de código pertinente se localizará con facilidad.
3. *Usar o modificar un programa existente.* Una de las principales causas de ineficiencia en el proceso de programación es que muchas veces un proyecto se aborda como si fuera el primer programa que se hubiera escrito. Se debería buscar primero un programa ya elaborado que realice la tarea completa o parte de ella. A la inversa, cuando se escribe un programa, se debería pensar en ponerlo a disposición de otras personas para posibles usos no previstos.
4. *Confeccionar herramientas.* En lenguaje de computación, una *herramienta* es un programa que tiene una variedad de usos. Cuando se escriba un programa, debe pensarse en la posibilidad de escribirlo de un modo más general sin mayor esfuerzo adicional. Por ejemplo, supóngase que se pide escribir un programa para la programación de exámenes finales; en lugar de eso, escribase una herramienta que tome un grafo arbitrario y coloréense sus vértices con el mínimo de colores posible, de modo que no haya dos vértices del mismo color unidos por una arista. En el contexto de la programación de exámenes, los vértices son los grupos de estudiantes, los colores son los períodos de exámenes, y una arista entre dos grupos significa que éstos tienen algún estudiante en común. El programa de coloración, junto con rutinas que traduzcan las listas de grupos en grafos y los colores en días y horas específicos, es el programador de exámenes. Sin embargo, el programa de coloración se puede usar para resolver problemas que no tengan relación alguna con la programación de exámenes, como el problema de los somáforos de la sección 1.1.
5. *Programar a nivel de mandatos.* En ocasiones, es imposible encontrar en una biblioteca el programa que se necesita para realizar cierto trabajo, ni adaptar una herramienta para tal efecto. Un sistema operativo bien diseñado permite conectar entre sí una red de programas disponibles sin que sea necesario escribir un programa nuevo, sino sólo una lista de mandatos (*commands*) del sistema ope-

rativo. Para que los mandatos sean pertinentes, es necesario que cada uno se comporte como un *filtro*, o sea como un programa con un archivo de entrada y otro de salida. Obsérvese que es posible componer un número arbitrario de filtros, y si el sistema operativo está diseñado de forma inteligente, una simple lista de mandatos (dispuestos como han de ejecutarse) será suficiente como programa.

Ejemplo 1.11. Como ejemplo de lo anterior, considérese el programa *spell*, tal como fue escrito por S. C. Johnson a partir de mandatos de UNIX †. Este programa toma como entrada un archivo a_1 que contiene un texto en inglés, y produce como salida todas aquellas palabras en a_1 que no se encuentren en un pequeño diccionario ‡‡. *spell* tiende a listar los nombres propios como faltas de ortografía, al igual que las palabras que no figuren en el diccionario, pero dado que la salida típica del programa es pequeña, puede examinarse visualmente, y la inteligencia de cualquier ser humano puede determinar si una palabra de la salida de *spell* tiene faltas de ortografía o no. (La ortografía de la versión original en inglés de este libro se verificó con *spell*.)

El primer filtro utilizado por *spell* es un mandato llamado *translate*; cuando este mandato recibe parámetros apropiados, puede reemplazar las letras mayúsculas por minúsculas y los espacios por saltos de línea, sin alterar los caracteres restantes. La salida de *translate* es un archivo a_2 que contiene las palabras de a_1 , sin mayúsculas y una en cada línea. A continuación, aparece el mandato *sort* que clasifica las líneas de su archivo de entrada en orden lexicográfico (alfabético). La salida de *sort* es un archivo a_3 que tiene todas las palabras de a_2 en orden alfabetico, con repeticiones. Entonces, un mandato *unique* elimina las líneas duplicadas de su archivo de entrada y produce un archivo de salida a_4 que contiene las palabras del archivo original, sin mayúsculas ni duplicaciones, en orden alfabetico. Por último, se aplica a a_4 un mandato *diff*, con un parámetro que indica un segundo archivo a_5 que contiene una lista alfabetizada de las palabras del diccionario, una en cada línea. El resultado tiene todas las palabras de a_4 (y por tanto de a_1) que no están en a_5 , esto es, todas las palabras de la entrada original que no se encuentran en el diccionario. El programa *spell* completo no es más que la siguiente secuencia de mandatos.

```
spell: translate [A-Z] → [a-z], espacio → nueva_línea
      sort
      unique
      diff diccionario
```

La programación a nivel de mandatos requiere disciplina por parte de una comunidad de programadores; éstos deben escribir programas como filtros, y escribir herramientas, en lugar de programas de propósito especial, siempre que les sea posible. La recompensa de esto, en función de la razón global entre trabajo y resultados, es sustancial. □

† UNIX es marca registrada de los Laboratorios Bell.

‡‡ Podría usarse un diccionario no abreviado, pero muchos de los errores ortográficos pueden ser palabras en desuso.

1.7 Súper Pascal

La mayoría de los programas de este libro están escritos en Pascal. No obstante, para aumentar su legibilidad, en ocasiones se usan tres construcciones que no forman parte de Pascal estándar, pero que se pueden traducir mecánicamente a Pascal puro. Una de tales construcciones es la etiqueta no numérica. Las pocas veces que se requieran etiquetas, serán no numéricas, porque facilitan la comprensión de los programas. Por ejemplo, «*goto salida*» siempre es más claro que «*goto 561*». Para convertir a Pascal puro un programa que contenga etiquetas no numéricas, se debe reemplazar cada una de ellas por una etiqueta numérica distinta y después declarar las etiquetas numéricas al principio del programa. Este proceso puede llevarse a cabo mecánicamente.

La segunda construcción no estándar es la proposición de retorno, que se usa porque permite escribir programas más fáciles de entender, sin emplear proposiciones *goto* para interrumpir el flujo de control. La proposición de retorno que aquí se emplea tiene la forma

```
return (expresión)
```

donde (*expresión*) es opcional. Es fácil convertir a Pascal estándar un procedimiento que contenga proposiciones **return**. En primer lugar, se declara una etiqueta nueva, como 999, y con ella se etiqueta la última proposición **end** del procedimiento. Si en una función de nombre *y*, por ejemplo, aparece la proposición **return(x)**, ésta se reemplaza por el bloque

```
begin
  y := x;
  goto 999
end
```

En un procedimiento, la proposición **return**, que puede carecer de argumento, simplemente se reemplaza por **goto 999**.

Ejemplo 1.12. La figura 1.15 muestra el programa para calcular factoriales, escrito con proposiciones **return**. En la figura 1.16 aparece el programa en Pascal resultante de la aplicación sistemática de la transformación indicada a la figura 1.15. □

```
function fact ( n: integer ) : integer;
begin
  if n <= 1 then
    return (1)
  else
    return (n * fact(n-1))
end; {fact}
```

Fig. 1.15. Programa de cálculo de factoriales con proposiciones de retorno.

```
function fact ( n: integer ) : integer;
label
  999;
begin
  if n <= 1 then
    begin
      fact := 1;
      goto 999
    end
  else
    begin
      fact := n * fact(n-1);
      goto 999
    end
  999:
end: {fact }
```

Fig. 1.16. Programa resultante en Pascal.

La tercera extensión a Pascal que se utiliza consiste en usar expresiones como nombres de tipos a lo largo de un programa de manera uniforme. Por ejemplo, una expresión como \uparrow tipo_celda, aunque puede emplearse en cualquier otra parte, no se permite como tipo de un parámetro de un procedimiento o como tipo del valor devuelto por una función. Técnicamente, Pascal requiere que se invente un nombre para la expresión de tipo, como ap_a_celda. En este libro se permitirá este uso de las expresiones de tipo, y se sugiere inventar un nombre para el tipo y que en forma mecánica se reemplacen las expresiones por los nombres. Así, se escribirán proposiciones como

function y (*A*: array [1..10] of integer): \uparrow tipo_celda

en lugar de

function y (*A*: arreglo_de_diez_enteros): ap_a_celda

Para concluir, debe hacerse una observación acerca de las convenciones tipográficas que se utilizan en los programas. Las palabras reservadas de Pascal se escriben en negritas; los tipos, en redondas, y los nombres de variables, en cursivas. Se hará distinción entre letras mayúsculas y minúsculas.

Ejercicios

- 1.1 Una liga de fútbol tiene seis equipos que se representarán con las letras *A*, *B*, *C*, *D*, *E* y *F*. El equipo *A* ya jugó contra los equipos *B* y *C*; el equipo *B*,

además, se enfrentó al *D* y al *F*. El *E* ha jugado contra el *C* y el *F*. Cada equipo tiene un encuentro por semana. Elabórese un calendario tal que cada equipo haya jugado contra todos los demás en el menor número posible de semanas. *Sugerencia:* Diséñese un grafo cuyos vértices representen pares de equipos que aún no se hayan enfrentado. ¿Cuáles deberían ser las aristas para que en una coloración lícita del grafo, cada color pudiera representar los juegos realizados en una semana?

- *1.2 Considérese un brazo robot fijo por un extremo. El brazo tiene dos articulaciones; en cada una de ellas, es posible rotarlo 90 grados hacia arriba o hacia abajo en un plano vertical. ¿Cómo podría modelarse matemáticamente la variedad de movimientos posibles del extremo del brazo? Describase un algoritmo para mover el extremo del brazo robot de una posición permisible a otra.
- *1.3 Supóngase que se desean multiplicar cuatro matrices de números reales: $M_1 \times M_2 \times M_3 \times M_4$, donde M_1 es de 10 por 20, M_2 de 20 por 50, M_3 de 50 por 1 y M_4 de 1 por 100. Supóngase que la multiplicación de una matriz de $p \times q$ por otra de $q \times r$ requiere pqr operaciones escalares, como sucede con el algoritmo usual de multiplicación de matrices. Encuéntrese el orden óptimo de multiplicación de matrices que minimice el número total de operaciones escalares. ¿Cómo se encontraría ese ordenamiento óptimo si se tuviera un número arbitrario de matrices?
- **1.4 Supóngase que se desean dividir las raíces cuadradas de los enteros del 1 al 100 en dos pilas de 50 números cada una, de modo que la suma de los números de la primera pila sea lo más cercana posible a la suma de los números de la segunda. Si sólo se dispusiera de dos minutos de tiempo de computador para resolver el problema, ¿qué cálculo sería aconsejable realizar?
- 1.5 Describase un algoritmo ávido para jugar al ajedrez. ¿Cabe esperar que tenga un buen rendimiento?
- 1.6 En la sección 1.2, se consideró el TDA CONJUNTO con las operaciones ANULA, UNION y TAMAÑO. Supóngase por conveniencia que todos los conjuntos son subconjuntos de $\{0, 1, \dots, 31\}$, y que el TDA CONJUNTO se interpreta como el tipo de datos de Pascal set of 0..31. Escribanse procedimientos en PASCAL para esas operaciones utilizando esta realización de CONJUNTO.
- 1.7 El *máximo común divisor* de dos enteros p y q es el mayor entero d que divide exactamente p y q . Se desea desarrollar un programa para calcular el máximo común divisor de dos enteros p y q con el siguiente algoritmo. Sea r el residuo de dividir p entre q . Si r es igual a cero, entonces q es el máximo común divisor. En caso contrario, hágase que p tome el valor de q , y q el de r , y repítase el proceso.
 - a) Muéstrese que este proceso encuentra el máximo común divisor correcto.

- b) Refíñese el algoritmo en la forma de un programa en seudolenguaje.
 c) Conviértase el programa en seudolenguaje en un programa en Pascal.
- 1.8** Se desea desarrollar un programa para formato de textos que coloque las palabras en las líneas de modo que queden justificadas a la izquierda y a la derecha. El programa tiene dos áreas de almacenamiento transitorio (*buffers*), uno de palabras y otro de líneas; ambas están vacías al inicio. Una palabra se lee al área transitoria, y si hay espacio suficiente en el área de líneas, la palabra se transfiere a ésta. En caso contrario, se insertan espacios adicionales entre las palabras del área transitoria de líneas para completar una línea; al imprimir esta línea, el área se vacía.
- a) Refíñese este algoritmo en la forma de un programa en seudolenguaje.
 b) Conviértase el programa en seudolenguaje en un programa en Pascal.
- 1.9** Considérese un conjunto de n ciudades y una tabla de distancias entre pares de ellas. Escribase un programa en seudolenguaje que encuentre un camino corto que pase por cada ciudad sólo una vez y vuelva a la ciudad en que empezó. No se conoce un método que no realice una búsqueda exhaustiva para obtener el recorrido más corto. Así pues, inténtese encontrar un algoritmo eficiente para este problema utilizando algún procedimiento heurístico razonable.
- 1.10** Considérense las siguientes funciones de n :

$$f_1(n) = n^2$$

$$f_2(n) = n^2 + 1000n$$

$$f_3(n) = \begin{cases} n & \text{si } n \text{ es impar} \\ n^3 & \text{si } n \text{ es par} \end{cases}$$

$$f_4(n) = \begin{cases} n & \text{si } n \leq 100 \\ n^3 & \text{si } n > 100 \end{cases}$$

Para cada posible par de valores de i y j , indíquese si $f_i(n)$ es $O(f_j(n))$ o no, y si $f_i(n)$ es $\Omega(f_j(n))$ o no.

- 1.11** Considérense las siguientes funciones de n :

$$g_1(n) = \begin{cases} n^2 & \text{para } n \text{ par} \geq 0 \\ n^3 & \text{para } n \text{ impar} \geq 1 \end{cases}$$

$$g_2(n) = \begin{cases} n & \text{para } 0 \leq n \leq 100 \\ n^3 & \text{para } n > 100 \end{cases}$$

$$g_3(n) = n^{2.5}$$

Para cada par de valores de i y j , indíquese si $g_i(n)$ es $O(g_j(n))$ o no, y si $g_i(n)$ es $\Omega(g_j(n))$ o no.

- 1.12 Mediante la notación asintótica, obténganse los tiempos de ejecución del peor caso supuesto para cada uno de los procedimientos siguientes como una función de n .

```

a) procedure prod_mat ( n: integer );
  var
    i, j, k: integer;
  begin
    for i := 1 to n do
      for j := 1 to n do begin
        C[i, j] := 0;
        for k := 1 to n do
          C[i, j] := C[i, j] + A[i, k] * B[k, j]
      end
    end
  end

b) procedure misterio ( n: integer );
  var
    i, j, k: integer;
  begin
    for i := 1 to n - 1 do
      for j := i + 1 to n do
        for k := 1 to j do
          { alguna proposición que requiera tiempo  $O(1)$  }
    end
  end

c) procedure muy_impar ( n: integer );
  var
    i, j, x, y: integer;
  begin
    for i := 1 to n do
      if odd(i) then begin
        for j := i to n do
          x := x + 1;
        for j := 1 to i do
          y := y + 1
      end
    end
  end

*d) function recursiva ( n: integer ) : integer;
  begin
    if n <= 1 then
      return (1)
    else
      return ( recursiva(n - 1) + recursiva(n - 1) )
  end

```

- 1.13** Muéstrese que las siguientes afirmaciones son verdaderas.
- 17 es $O(1)$.
 - $n(n-1)/2$ es $O(n^2)$.
 - $\max(n^3, 10n^2)$ es $O(n^3)$.
 - $\sum_{i=1}^n i^k$ es $O(n^{k+1})$ y $\Omega(n^{k+1})$ para k entero.
 - Si $p(x)$ es cualquier polinomio de grado k donde el coeficiente del término de mayor grado es positivo, entonces $p(n)$ es $O(n^k)$ y $\Omega(n^k)$.
- *1.14** Supóngase que $T_1(n)$ es $\Omega(f(n))$ y $T_2(n)$ es $\Omega(g(n))$. ¿Son verdaderas las siguientes afirmaciones?
- $T_1(n) + T_2(n)$ es $\Omega(\max(f(n), g(n)))$.
 - $T_1(n)T_2(n)$ es $\Omega(f(n)g(n))$.
- *1.15** Algunos autores definen la notación omega mayúscula de la siguiente manera: $f(n)$ es $\Omega(g(n))$ si existen n_0 y $c > 0$ tales que para todo $n \geq n_0$ se tiene que $f(n) \geq cg(n)$.
- ¿Es verdadero para esta definición que $f(n)$ es $\Omega(g(n))$ si, y sólo si, $g(n)$ es $O(f(n))$?
 - ¿Es a) verdadero para la definición de omega mayúscula que se dio en la sección 1.4?
 - ¿Se cumplen con esta definición de omega mayúscula las afirmaciones a) y b) del ejercicio 1.14?
- 1.16** Ordénense las siguientes funciones de acuerdo con su velocidad de crecimiento: a) n , b) \sqrt{n} , c) $\log n$, d) $\log\log n$, e) $\log^2 n$, f) $n/\log n$, g) $\sqrt{n}\log^2 n$, h) $(1/3)^n$, i) $(3/2)^n$, j) 17.
- 1.17** Supóngase que el parámetro n del procedimiento siguiente es una potencia positiva de 2, esto es, $n = 2, 4, 8, 16, \dots$ Proporciónese una fórmula que exprese el valor de la variable *cuenta* en función del valor de n cuando termina el procedimiento.

```

procedure misterio (n: integer);
var
  x, cuenta: integer;
begin
  cuenta := 0;
  x := 2;
  while x < n do begin
    x := 2 * x;
    cuenta := cuenta + 1
  end;
  writeln (cuenta)
end

```

- 1.18 La siguiente es una función *máx* (*i*, *n*) que devuelve el mayor elemento en las posiciones *i* a *i+n-1* de una matriz entera *A*. Por conveniencia, puede suponerse que *n* es una potencia de 2.

```

function máx (i, n: integer) : integer;
var
    m1, m2: integer;
begin
    if n = 1 then
        return (A[i])
    else begin
        m1 := máx(i, n div 2);
        m2 := máx(i+n div 2, n div 2);
        if m1 < m2 then
            return (m2)
        else
            return (m1)
    end
end

```

- a) Sea $T(n)$ el tiempo del peor caso para *máx* con segundo argumento igual a *n*. Esto es, *n* es el número de elementos de entre los cuales se encuentra el mayor. Escribase una ecuación que exprese $T(n)$ en función de $T(j)$ para uno o más valores de *j* menores que *n* y una constante o constantes que representen los tiempos tomados por las proposiciones individuales del programa *máx*.
- b) Obténgase una cota asintótica superior (notación «o mayúscula») lo más cercana posible para $T(n)$. La solución debe ser igual a una cota asintótica inferior (notación «omega mayúscula»), y ser lo más sencilla posible.

Notas bibliográficas

El origen del concepto de tipo de datos abstracto se remonta al tipo *class* en el lenguaje SIMULA 67 (Birtwistle *et al.* [1973]). Desde entonces, se han desarrollado varios lenguajes que manejan tipos de datos abstractos; entre ellos están Alphard (Shaw, Wulf y London [1977]), C con clases (Stroustrup [1982]), CLU (Liskov *et al.* [1977]), MESA (Geschke, Morris y Satterthwaite [1977]) y Russell (Demers y Donahue [1979]). El concepto de TDA se analiza con más profundidad en trabajos tales como Gotlieb y Gotlieb [1978] y Wulf *et al.* [1981].

Knuth [1968] fue el primer trabajo importante en el que se preconizó el estudio sistemático del tiempo de ejecución de programas. Aho, Hopcroft y Ullman [1974] relaciona la complejidad espacial y temporal de los algoritmos con varios modelos computacionales, como las máquinas de Turing y las de acceso aleatorio. Para más referencias sobre el tema del análisis de algoritmos y programas, véanse también las notas bibliográficas del capítulo 9.

Como material adicional sobre programación estructurada véanse Hoare, Dahl y Dijkstra [1972], Wirth [1973], Kernighan y Plauger [1974], y Yourdon y Constantine [1975]. Los problemas organizacionales y psicológicos que se presentan en el desarrollo de proyectos grandes de software se tratan en Brooks [1974] y Weinberg [1971]. En Kernighan y Plauger [1981] se muestra cómo construir herramientas útiles para un ambiente de programación.

2

Tipos de datos abstractos fundamentales

En este capítulo se estudiarán algunos tipos de datos abstractos fundamentales. Se considerarán las listas, que son secuencias de elementos, y dos tipos especiales de listas: las pilas, donde los elementos se insertan y eliminan sólo en un extremo, y las colas, donde los elementos se insertan por un extremo y se eliminan por el otro. Se estudiará después, en forma breve, la correspondencia o memoria asociativa (*mapping*), un TDA que se comporta como una función. Para cada uno de estos TDA, se considerarán varias realizaciones y se analizarán sus méritos relativos.

2.1 El tipo de datos abstracto «lista»

Las listas constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera; los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. Las listas también pueden concatenarse entre sí o dividirse en sublistas; se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación. Las técnicas de administración de memoria del tipo de las que se analizan en el capítulo 12 hacen uso extensivo de técnicas de procesamiento de listas. En esta sección se presentarán algunas operaciones básicas con listas, y en el resto del capítulo se presentarán estructuras de datos para representar listas que manejan con eficiencia varios subconjuntos de estas operaciones.

Matemáticamente, una *lista* es una secuencia de cero o más elementos de un tipo determinado (que por lo general se denominará *tipo_elemento*). A menudo se representa una lista como una sucesión de elementos separados por comas

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

donde $n \geq 0$ y cada a_i es del tipo *tipo_elemento*. Al número n de elementos se le llama *longitud* de la lista. Al suponer que $n \geq 1$, se dice que a_1 es el *primer elemento* y a_n el *último elemento*. Si $n = 0$, se tiene una *lista vacía*, es decir, que no tiene elementos.

Una propiedad importante de una lista es que sus elementos pueden estar ordenados en forma lineal de acuerdo con sus posiciones en la misma. Se dice que a_i *precede* a a_{i+1} para $i = 1, 2, \dots, n-1$, y que a_i *sigue* a a_{i-1} para $i = 2, 3, \dots, n$. Se dice que el elemento a_i está en la *posición i*. Es conveniente postular también la existencia de

una posición que sucede a la del último elemento de la lista. La función $\text{FIN}(L)$ devolverá la posición que sigue a la posición n en una lista L de n elementos. Obsérvese que la posición $\text{FIN}(L)$, con respecto al principio de la lista, está a una distancia que varía conforme la lista crece o se reduce, mientras que las demás posiciones guardan una distancia fija con respecto al principio de la lista.

Para formar un tipo de datos abstracto a partir de la noción matemática de lista, se debe definir un conjunto de operaciones con objetos de tipo LISTA †. Como sucede con muchos de los TDA que se analizan en este libro, ningún conjunto de operaciones es adecuado para todas las aplicaciones. Aquí se dará un conjunto representativo de operaciones. En la siguiente sección se mostrarán varias estructuras de datos para representar listas y se escribirán procedimientos para las operaciones características con listas en función de esas estructuras de datos.

Para ilustrar algunas operaciones comunes con listas, considérese una aplicación típica: se tiene una lista de direcciones y se desean eliminar las entradas dobles. En forma conceptual, este problema es bastante fácil de resolver: para cada elemento de la lista, eliminense todos los elementos sucesores equivalentes. Para presentar este algoritmo, sin embargo, es necesario definir operaciones que permitan encontrar el primer elemento de una lista, recorrer los demás elementos sucesivos, y recuperar y eliminar elementos.

Se presentará ahora un conjunto representativo de operaciones con listas. Ahí, L es una lista de objetos de tipo tipo_elemento, x es un objeto de ese tipo y p es de tipo posición. Obsérvese que «posición» es otro tipo de datos cuya implantación cambiará con aquella que se haya elegido para las listas. Aunque de manera informal se piensa en las posiciones como enteros, en la práctica pueden tener otra representación.

1. **INSERTA**(x, p, L). Esta función inserta x en la posición p de la lista L , pasando los elementos de la posición p y siguientes a la posición inmediata posterior. Esto quiere decir que si L es a_1, a_2, \dots, a_n , se convierte en $a_1, a_2, \dots, a_{p-1}, x, a_p, \dots, a_n$. Si p es $\text{FIN}(L)$, entonces L se convierte en a_1, a_2, \dots, a_n, x . Si la lista L no tiene posición p , el resultado es indefinido.
2. **LOCALIZA**(x, L). Esta función devuelve la posición de x en la lista L . Si x figura más de una vez en L , la posición de la primera aparición de x es la que se devuelve. Si x no figura en la lista, entonces se devuelve $\text{FIN}(L)$.
3. **RECUPERA**(p, L). Esta función devuelve el elemento que está en la posición p de la lista L . El resultado no está definido si $p = \text{FIN}(L)$ o si L no tiene posición p . Obsérvese que si se utiliza RECUPERA, los elementos deben ser de un tipo que pueda ser devuelto por una función. No obstante, en la práctica siempre es posible modificar RECUPERA para devolver un apuntador a un objeto de tipo tipo_elemento.
4. **SUPRIME**(p, L). Esta función elimina el elemento en la posición p de la lista L . Si L es a_1, a_2, \dots, a_n , L se convierte en $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$. El resultado no está definido si L no tiene posición p o si $p = \text{FIN}(L)$.

† En sentido estricto, el tipo es «LISTA de tipo_elemento». Sin embargo, las realizaciones de listas que se proponen no dependen del tipo_elemento; precisamente esta independencia es la que justifica la relevancia que se da al concepto de lista. Se usará «LISTA» en lugar de «LISTA de tipo_elemento», y se tratará de manera similar a los demás TDA que dependan de los tipos de elementos.

5. SIGUIENTE(p, L) y ANTERIOR (p, L) devuelven las posiciones siguiente y anterior, respectivamente, a p en la lista L . Si p es la última posición de L , SIGUIENTE(p, L)=FIN(L). SIGUIENTE no está definida si p es FIN(L). ANTERIOR no está definida si p es 1. Ambas funciones están indefinidas cuando L no tiene posición p .
6. ANULA(L). Esta función ocasiona que L se convierta en la lista vacía y devuelve la posición FIN(L).
7. PRIMERO(L). Esta función devuelve la primera posición de la lista L . Si L está vacía, la posición que se devuelve es FIN(L).
8. IMPRIME_LISTA(L). Imprime los elementos de L en su orden de aparición en la lista.

Ejemplo 2.1. Con estos operadores se escribirá un procedimiento PURGA que toma como argumento una lista y elimina sus elementos con doble entrada. Los elementos son de tipo tipo_elemento, y una lista de ellos es de tipo LISTA; esta convención se seguirá en todo el capítulo. Hay una función *mismo(x, y)*, donde x e y son de tipo tipo_elemento, que es verdadera si x e y son «el mismo» elemento, y es falsa en caso contrario. El significado de «ser el mismo» es intencionalmente impreciso. Si tipo_elemento es real, por ejemplo, podría desearse que *mismo(x, y)* fuera verdadera si, y sólo si, $x = y$. Sin embargo, si tipo_elemento es un registro que contiene el número de cuenta, el nombre y la dirección de un suscriptor como en

```
type
  tipo_elemento = record
    num_cta: integer;
    nombre: packed array [1..20] of char;
    dirección: packed array [1..50] of char
  end
```

por consiguiente podría desearse que *mismo(x, y)* fuera verdadera siempre que $x.\text{num_cta} = y.\text{num_cta}$ †.

La figura 2.1 muestra el código de PURGA. Las variables p y q se usan para representar dos posiciones en la lista. Conforme el programa avanza, se eliminan de la lista todas las entradas dobles de cualquier elemento a la izquierda de la posición p . En una iteración del ciclo (2)-(8), q se usa para revisar la lista a partir de la posición p , con el objeto de eliminar cualquier entrada doble del elemento en la posición p . Después, p avanza a la siguiente posición y el proceso se repite.

En la siguiente sección, se proporcionarán las declaraciones apropiadas para LISTA y posición, y una aplicación de las operaciones, de manera que PURGA se convierta en un programa ejecutable. Como ya se dijo, el programa es independiente de la forma en que se representen las listas, por lo que hay libertad para experimentar con varias realizaciones de listas.

† En este caso, si se eliminan los registros que son «el mismo», podría desearse verificar que los nombres y direcciones fueran iguales; si los números de cuenta fueran iguales, pero los otros datos no, es posible que se haya asignado inadvertidamente el mismo número de cuenta a dos personas. No obstante, es más probable que el mismo suscriptor aparezca más de una vez en la lista, con diferentes números y con los demás datos ligeramente distintos. En tales casos, resulta difícil eliminar las entradas dobles.

```

procedure PURGA ( var L: LISTA );
{PURGA elimina los elementos duplicados de la lista L}
var
  p, q: posición; { p será la posición "actual" en L, y q
    avanzará para encontrar elementos iguales }
begin
  (1)   p := PRIMERO(L);
  (2)   while p <> FIN(L) do begin
  (3)     q := SIGUIENTE(p, L);
  (4)     while q <> FIN(L) do
  (5)       if mismo(RECUPERA(p, L), RECUPERA(q, L)) then
  (6)         SUPRIME(q, L)
  (7)       else
  (8)         q := SIGUIENTE(q, L);
  end
end; { PURGA }

```

Fig. 2.1. Programa para eliminar duplicados.

Un aspecto que es importante observar está relacionado con el cuerpo del ciclo interno, las líneas (4)-(7) de la figura 2.1. Cuando se elimina el elemento de la posición q de la línea (6), los elementos que estaban en las posiciones $q+1$, $q+2$, ..., etcétera, avanzan una posición en la lista. En particular, si q fuera la última posición de L , el valor de q sería entonces $\text{FIN}(L)$. Si después de esto se ejecutara la línea (7), $\text{SIGUIENTE}(\text{FIN}(L), L)$ produciría un resultado indefinido. Por tanto, es esencial que sólo alguna de las líneas, (6) o (7), pero no las dos, se ejecute entre las pruebas para saber si $q = \text{FIN}(L)$ en la línea (4). \square

2.2 Realización de listas

En esta sección se describirán algunas estructuras de datos que pueden utilizarse para representar listas. Se considerarán realizaciones de listas basadas en arreglos, apunadores y cursores. Cada una de ellas permite realizar ciertas operaciones con listas de manera más eficiente que otras.

Realización de listas mediante arreglos

En la realización de una lista mediante arreglos, los elementos de ésta se almacenan en celdas contiguas de un arreglo. Esta representación permite recorrer con facilidad una lista y agregarle elementos nuevos al final. Pero insertar un elemento en la mitad de la lista obliga a desplazarse una posición dentro del arreglo a todos los elementos que siguen al nuevo elemento para concederle espacio. De la misma for-

ma, la eliminación de un elemento, excepto el último, requiere desplazamientos de elementos para llenar de nuevo el vacío formado.

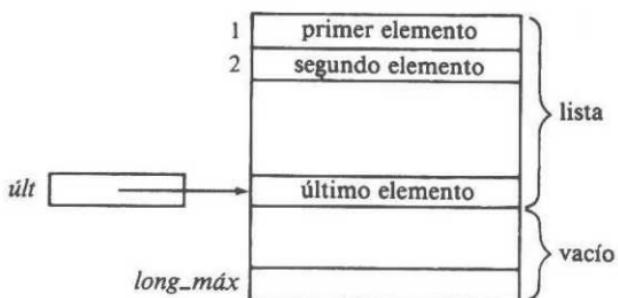


Fig. 2.2. Realización de una lista mediante un arreglo.

En la realización con arreglos se define el tipo LISTA como un registro con dos campos; el primero es un arreglo de elementos que tiene la longitud adecuada para contener la lista de mayor tamaño que se pueda presentar. El segundo campo es un entero *últ* que indica la posición del último elemento de la lista en el arreglo. El *i*-ésimo elemento de la lista está en la *i*-ésima celda del arreglo, para $1 \leq i \leq \text{últ}$, como se muestra en la figura 2.2. Las posiciones en la lista se representan mediante enteros; la *i*-ésima posición, mediante el entero *i*. La función FIN(*L*) sólo tiene que devolver *últ* + 1. Las declaraciones importantes son:

```

const
  long_máx = 100 { alguna constante apropiada };
type
  LISTA = record
    elementos: array[1..long_máx] of tipo_elemento;
    últ: integer
  end;
  posición = integer;
function FIN (var L: LISTA): posición;
begin
  return (L.últ + 1)
end; {FIN}

```

La figura 2.3 muestra cómo se podrían implantar las operaciones INSERTA, SUPRIME y LOCALIZA con esta realización basada en arreglos. INSERTA pasa los elementos de las localidades $p, p+1, \dots, \text{últ}$ a las localidades $p+1, p+2, \dots, \text{últ}+1$ y después inserta el nuevo elemento en la localidad p . Si no hay espacio en el arreglo para insertar un nuevo elemento, se invoca a la rutina *error*, que imprime su argumento.

[†] Aunque L no se modifica, pasa por referencia, pues por lo común es una estructura grande y no se desea perder tiempo copiándola.

```

procedure INSERTA ( x: tipo_elemento; p: posición; var L: LISTA );
{ INSERTA coloca x en la posición p de la lista L }
var
  q: posición;
begin
  if L.últ > = long_máx then
    error ('la lista está llena')
  else if (p > L.últ + 1) or (p < 1) then
    error ('la posición no existe')
  else begin
    for q := L.últ downto p do
      { desplaza los elementos en p, p + 1,... una posición hacia abajo }
      L.elementos [q + 1] := L.elementos[q];
    L.últ := L.últ + 1;
    L.elementos[p] := x
  end
end; { INSERTA }

procedure SUPRIME ( p: posición; var L: LISTA );
{ SUPRIME elimina el elemento en la posición p de la lista L }
var
  q: posición;
begin
  if (p > L.últ) or (p < 1) then
    error ('la posición no existe')
  else begin
    L.últ := L.últ - 1;
    for q := p to L.últ do
      { desplaza los elementos en p + 1, p + 2... una posición hacia
        arriba }
      L.elementos[q] := L.elementos[q + 1]
  end
end; { SUPRIME }

function LOCALIZA ( x: tipo_elemento; var L: LISTA ) : posición;
{ LOCALIZA devuelve la posición de x en la lista L }
var
  q: posición;
begin
  for q := 1 to L.últ do
    if L.elementos[q] = x then
      return (q);
  return (L.últ + 1) { si no se encuentra }
end; { LOCALIZA }

```

Fig. 2.3. Realización basada en arreglos de algunas operaciones con listas.

to y da por terminada la ejecución del programa. SUPRIME elimina el elemento de la posición p pasando los elementos de las posiciones $p + 1, p + 2, \dots, \text{últ}$, a las posiciones $p, p + 1, \dots, \text{últ} - 1$. LOCALIZA revisa el arreglo secuencialmente, en busca de un elemento determinado; si no lo encuentra, LOCALIZA devuelve $\text{últ} + 1$.

Debería estar claro cómo codificar las restantes operaciones con listas con esta forma de implantación. Por ejemplo, PRIMERO siempre devuelve uno; SIGUIENTE devuelve uno más que su argumento, y ANTERIOR, uno menos; cada uno verifica primero si el resultado está dentro del intervalo permisible; ANULA(L) coloca $L.\text{últ}$ en 0. Si el procedimiento PURGA de la figura 2.1 va precedido de

1. las definiciones de tipo_elemento y de la función *mismo*.
2. las definiciones de LISTA, posición y FIN(L) anteriores,
3. la definición de SUPRIME de la figura 2.3, y
4. definiciones adecuadas para los procedimientos triviales PRIMERO, SIGUIENTE y RECUPERA,

entonces se obtiene un procedimiento PURGA ejecutable.

Al principio puede parecer tedioso escribir procedimientos que rijan todos los accesos a las estructuras subyacentes de un programa. Sin embargo, si se logra establecer la disciplina de escribir programas en función de operaciones de manipulación de tipos de datos abstractos en lugar de usar ciertos detalles de implantación particulares, es posible modificar los programas más fácilmente con sólo aplicar de nuevo las operaciones, en lugar de buscar en todos los programas aquellos lugares donde se hacen accesos a las estructuras de datos subyacentes. La flexibilidad que se obtiene con esto, puede ser especialmente importante en proyectos grandes, y no se debe poner en tela de juicio este concepto por los ejemplos, necesariamente pequeños, que se encuentran en este libro.

Realización de listas mediante apuntadores

La segunda forma de realización de listas, celdas enlazadas sencillas, utiliza apuntadores para enlazar elementos consecutivos. Esta implantación permite eludir el empleo de memoria contigua para almacenar una lista y, por tanto, también elude los desplazamientos de elementos para hacer inserciones o llenar vacíos creados por la eliminación de elementos. No obstante, por esto hay que pagar el precio de un espacio adicional para los apuntadores.

En esta representación, una lista está formada por celdas; cada celda contiene un elemento de la lista y un apuntador a la siguiente celda. Si la lista es a_1, a_2, \dots, a_n , la celda que contiene a_i tiene un apuntador a la celda que contiene a a_{i+1} , para $i = 1, 2, \dots, n - 1$. La celda que contiene a_n posee un apuntador **nil**. Existe también una celda de *encabezamiento* que apunta a la celda que contiene a_1 ; esta celda de encabezamiento no tiene ningún elemento[†]. En el caso de una lista vacía, el apuntador del encabe-

[†] El empleo de una celda completa para el encabezamiento simplifica la implantación de operaciones con listas en Pascal. Es posible usar apuntadores como encabezamientos si se pretende aplicar operaciones de modo que las inserciones y las eliminaciones al principio de la lista se manejen de manera especial. Véase el análisis de este método en esta misma sección, en el apartado que explica la realización de listas mediante cursores.

zamiento es **nil** y no se tienen más celdas. La figura 2.4 muestra una lista enlazada de esta manera.

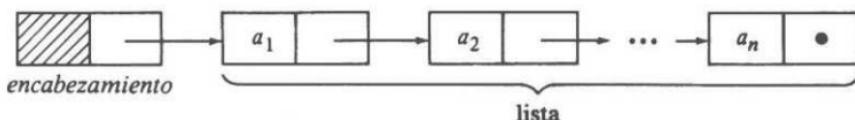


Fig. 2.4. Lista enlazada.

Para las listas enlazadas sencillas es conveniente usar una definición de posición ligeramente distinta de la que se empleó para las listas logradas mediante arreglos. Aquí, la posición i será un apuntador a la celda que contiene el apuntador a a_i para $i = 2, 3, \dots, n$. La posición 1 es un apuntador al encabezamiento, y la posición $\text{FIN}(L)$ es un apuntador a la última celda de L .

El tipo de una lista, en realidad, es el mismo que el de una posición; es un apuntador a una celda particular: el encabezamiento. La siguiente es una definición formal de las partes esenciales de una estructura de datos de lista enlazada.

```
type
    tipo_celda = record
        elemento: tipo_elemento;
        sig: ↑ tipo_celda
    end;
    LISTA = ↑ tipo_celda;
    posición = ↑ tipo_celda;
```

En la figura 2.5 se muestra la función $\text{FIN}(L)$; ésta funciona moviendo el apuntador q por la lista, hasta que alcanza el final, el cual se detecta por el hecho de que q apunta a una celda con un apuntador **nil**. Obsérvese que esta realización de FIN es ineficiente, pues requiere revisar la lista entera cada vez que se desea calcular $\text{FIN}(L)$. Si es necesario hacer esto con frecuencia, como en el programa PURGA de la figura 2.1, puede optarse por cualquiera de las opciones siguientes:

```
function FIN ( L: LISTA ) : posición;
    { FIN devuelve un apuntador a la última celda de  $L$  }
    var
        q: posición;
    begin
        (1)      q := L;
        (2)      while q↑.sig <> nil do
        (3)          q := q↑.sig;
        (4)      return (q)
    end; { FIN }
```

Fig. 2.5. La función FIN.

1. usar una representación de listas que incluya un apuntador de la última celda, o
2. sustituir el uso de $\text{FIN}(L)$ donde sea posible. Por ejemplo, la condición $p <> \text{FIN}(L)$ de la línea (2) de la figura 2.1 podría reemplazarse por $p \uparrow. \text{sig} <> \text{nil}$, al costo de que ese programa dependa de una realización particular de listas.

La figura 2.6 contiene rutinas para las operaciones INSERTA, SUPRIME, LOCALIZA y ANULA usando esta realización de listas con apuntadores. Las operaciones restantes se pueden obtener como rutinas de un solo paso, con la excepción de ANTERIOR, que requiere revisar la lista desde el principio. Estas rutinas se dejan como ejercicio para el lector. Obsérvese que muchos de los mandatos no necesitan el parámetro L , la lista, por lo que se omite.

En la figura 2.7 se muestra la mecánica del manejo de apuntadores en el procedimiento INSERTA de la figura 2.6. La figura 2.7 (a) muestra la situación antes de la ejecución de INSERTA. Se desea insertar un nuevo elemento ante la celda que contiene b , por lo que p es un apuntador a la celda de la lista que contiene el apuntador a b . En la línea (1), temp apunta a la celda que contiene b . En la línea (2), se crea una nueva celda de la lista y se hace que el campo sig de la celda que contiene a apunte hacia esta misma celda. En la línea (3) se almacena x en el campo elemento de la celda recién creada, y en la línea (4), el campo sig toma el valor de temp y, de esta manera, apunta a la celda que contiene b . La figura 2.7 (b) muestra el resultado de la ejecución de INSERTA. Los apuntadores nuevos se representan con líneas punteadas y se marca el paso en el cual fueron creados.

El procedimiento SUPRIME es más sencillo. La figura 2.8 muestra las manipulaciones de apuntadores que realiza el procedimiento SUPRIME de la figura 2.6. Los apuntadores antiguos se representan por medio de líneas de trazo continuo, y los nuevos por medio de líneas punteadas.

Obsérvese que una posición en una implantación de listas enlazadas se comporta de distinta forma que una posición en una realización de arreglos. Supóngase que se tiene una lista con tres elementos a, b, c y una variable p de tipo posición que tiene actualmente el valor 3; es decir, apunta a la celda que contiene b y, por tanto, representa la posición de c . Si se ejecuta un mandato para insertar x en la posición 2, la lista se convierte en a, x, b, c , con lo cual el elemento b pasaría a ocupar la posición 3. Si se utilizara la realización de arreglos descrita anteriormente, b y c se moverían hacia el principio del arreglo, de modo que b ocuparía en realidad la tercera posición.

```

procedure INSERTA ( x: tipo_elemento; p: posición );
var
    temp: posición;
begin
(1)    temp := p↑.sig;
(2)    new(p↑.sig);
(3)    p↑.sig↑.elemento := x;
(4)    p↑.sig↑.sig := temp
end; { INSERTA }

```

```

procedure SUPRIME ( p: posición );
begin
   $p \uparrow .sig := p \uparrow .sig \uparrow .sig$ 
end; { SUPRIME }

function LOCALIZA ( x: tipo_elemento; L: LISTA ) : posición;
var
  p: posición;
begin
  p := L;
  while  $p \uparrow .sig <> \text{nil}$  do
    if  $p \uparrow .sig \uparrow .elemento = x$  then
      return (p)
    else
      p :=  $p \uparrow .sig$ ;
  return (p) { si no es encontrado }
end; { LOCALIZA }

function ANULA ( var L: LISTA ) : posición;
begin
  new (L);
  L  $\uparrow .next := \text{nil}$ ;
  return(L)
end; { ANULA }

```

Fig. 2.6. Algunas operaciones con la realización de listas enlazadas.

Pero, si se usara la realización de listas enlazadas, el valor de p , que es un apuntador a la celda que contiene b , no cambiaría como consecuencia de la inserción, de modo que después de ésta el valor de p sería la «posición 4», no la 3. Esta variable de posición debe actualizarse si se va a usar después como posición de $b \uparrow$.

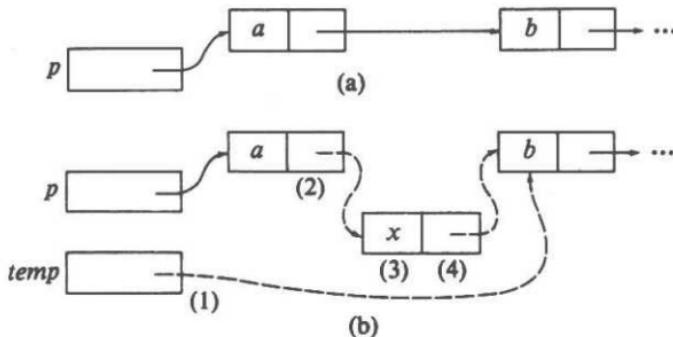


Fig. 2.7. Diagrama de INSERTA.

† Por supuesto, existen muchas situaciones en las que se podría desear que p represente la posición de c .

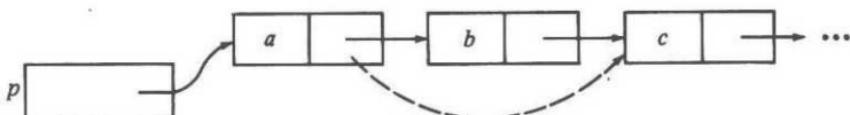


Fig. 2.8. Diagrama de SUPRIME.

Comparación de los métodos

Cabe preguntarse si en determinadas circunstancias es mejor usar una realización de listas basada en apuntadores o una basada en arreglos. A menudo, la respuesta depende de cuáles son las operaciones que se desea realizar, o las que se realizan más frecuentemente. Otras veces, la decisión depende de lo larga que puede llegar a ser la lista. Los aspectos más importantes a tener en cuenta son los siguientes.

1. La realización con arreglos requiere especificar el tamaño máximo de una lista en el momento de la compilación. Si no es posible acotar la probable longitud de la lista, quizás deba optarse por una realización con apuntadores.
2. Ciertas operaciones son más lentas en una realización que en otra. Por ejemplo, INSERTA y SUPRIME tienen un número constante de pasos para una lista enlazada, pero cuando se utiliza la realización con arreglos, requieren un tiempo proporcional al número de elementos que siguen. A la inversa, la ejecución de ANTERIOR y FIN requiere un tiempo constante con la realización con arreglos, pero se usa un tiempo proporcional a la longitud de la lista si se usan apuntadores.
3. Si un programa efectúa inserciones o supresiones que afecten al elemento que ocupa la posición denotada por alguna variable de posición, y el valor de esa variable se va a utilizar posteriormente, entonces la representación con apuntadores no se puede usar de la forma descrita aquí. Como principio general, los apuntadores se deben utilizar con gran cuidado y moderación.
4. La realización con arreglos puede malgastar espacio, puesto que usa la cantidad máxima de espacio, independientemente del número de elementos que en realidad tiene la lista en un momento dado. La realización con apuntadores utiliza sólo el espacio necesario para los elementos que actualmente tiene la lista, pero requiere espacio para el apuntador en cada celda. Así, cualquiera de los dos métodos podría usar más espacio que el otro, dependiendo de las circunstancias.

Realización de listas basadas en cursores

Algunos lenguajes, como FORTRAN y ALGOL, no tienen apuntadores. Si se trabaja con un lenguaje tal, se pueden simular los apuntadores mediante cursores; esto es, con enteros que indican posiciones en arreglos. Se crea un arreglo de registros para

almacenar todas las listas de elementos cuyo tipo es tipo_elemento; cada registro contiene un elemento y un entero que se usa como cursor. Es decir, se define

```

var
  ESPACIO: array [1..long_máx] of record
    elemento: tipo_elemento;
    sig: integer
  end

```

Si L es una lista de elementos, se declara una variable entera, por ejemplo, *Lencab*, como encabezamiento para L . Se puede tratar *Lencab* como un cursor a la celda de encabezamiento en *ESPACIO* con un campo *elemento* vacío. Las operaciones con listas se pueden implantar, entonces, como en la realización basada en apuntadores recién descrita.

Aquí se describirá una realización alternativa que evita el uso de celdas de encabezamiento, al tomar como casos especiales las inserciones y supresiones en la posición 1. Esta técnica se puede usar también con listas enlazadas basadas en apuntadores, para evitar el empleo de celdas de encabezamiento. Para una lista L , el valor de *ESPACIO* [*Lencab*].*elemento* es el valor del primer elemento de L . El valor de *ESPACIO* [*Lencab*].*sig* es el índice de la celda que contiene el segundo elemento, y así sucesivamente. Un valor de 0 ya sea en *Lencab* o en el campo *sig*, representa un «apuntador nil», esto es, no hay un elemento siguiente.

Una lista tendrá tipo entero, ya que el encabezamiento es una variable entera que representa la lista como un todo. Las posiciones serán también de tipo entero. Se adopta aquí la convención de que la posición i de la lista L es el índice de la celda de *ESPACIO* que contiene el elemento $i - 1$ de L , ya que el campo *sig* de esa celda contendrá el cursor al elemento i . Como caso especial, la posición 1 de cualquier lista se representa por 0. Dado que el nombre de la lista es siempre un parámetro de las operaciones que usan posiciones, es posible distinguir entre las primeras posiciones de distintas listas. La posición *FIN(L)* es el índice del último elemento de L .

La figura 2.9 muestra dos listas, $L = a, b, c$ y $M = d, e$, que comparten el arreglo *ESPACIO* con *long_máx* = 10. Obsérvese que las celdas del arreglo no contenidas

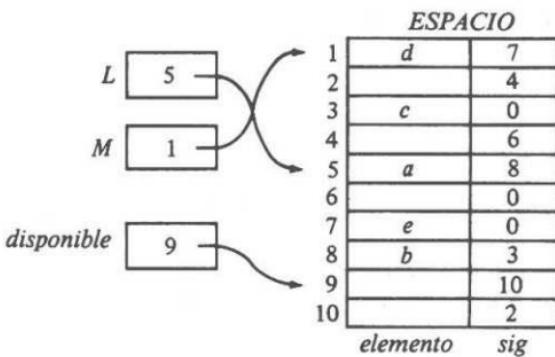


Fig. 2.9. Realización de listas enlazadas mediante cursores.

en L ni en M están enlazadas en otra lista llamada *disponible*. Tal lista es necesaria para obtener una celda vacía cuando se deseé hacer una inserción, y disponer de lugar donde poner las celdas suprimidas para su uso posterior.

Para insertar un elemento x en la lista L , se toma la primera celda de la lista *disponible* y se coloca en la posición correcta en la lista L . El elemento x se pone entonces en el campo *elemento* de esta celda. Para suprimir un elemento x de la lista L , se quita de L la celda que contiene x y se devuelve al principio de la lista *disponible*. Estas dos acciones pueden verse como casos especiales del acto de tomar una celda C apuntada por un cursor p y hacer que otro cursor q apunte a C , a la vez que se hace que p quede apuntando hacia donde C apuntaba y C quede apuntando hacia donde q apuntaba. De esta forma, C se inserta entre q y aquello hacia lo que apuntaba q . Por ejemplo, si se suprime b de la lista L en la figura 2.9, C es la fila 8 de *ESPACIO*, p es *ESPACIO[5].sig* y q es *disponible*. Los cursores antes (en líneas de trazo continuo) y después (en líneas punteadas) de esta acción se muestran en la figura 2.10, y el código está incluido en la función *mueve* de la figura 2.11, que realiza el movimiento si C existe y devuelve falso si C no existe.

La figura 2.12 muestra los procedimientos *INSERTA* y *SUPRIME*, y un procedimiento *val_inicial* que enlaza las celdas del arreglo *ESPACIO* para formar una lista de espacio disponible. Estos procedimientos no incluyen verificaciones para detectar errores; es aconsejable insertarlas como ejercicio para el lector. Se dejan también como ejercicios otras operaciones similares a las de las listas enlazadas basadas en apuntadores.

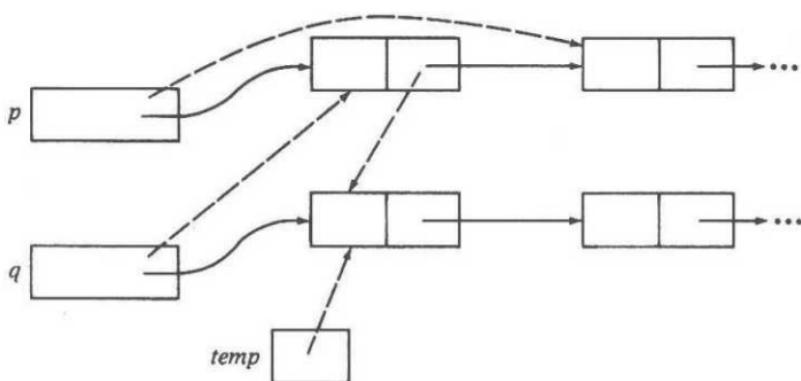


Fig. 2.10. Cambio de una celda C de una lista a otra.

```

function mueve (var  $p, q$ : integer) : boolean;
{mueve coloca la celda apuntada por  $p$  delante de la celda apuntada por  $q$ }
var
    temp: integer;
begin

```

```

if p = 0 then begin { celda inexistente }
    writeln('la celda no existe');
    return (false)
end
else begin
    temp := q;
    q := p;
    p := ESPACIO[q].sig;
    ESPACIO[q].sig := temp;
    return (true)
end
end; { mueve }

```

Fig. 2.11. Código para cambiar una celda.

Listas doblemente enlazadas

En algunas aplicaciones puede ser deseable recorrer eficientemente una lista, tanto hacia adelante como hacia atrás. O, dado un elemento, podría desearse determinar con rapidez el siguiente y el anterior. En tales situaciones, quizás se quisiera poner en cada celda de una lista un apuntador a la siguiente celda y otro a la anterior, como se sugiere en la lista doblemente enlazada de la figura 2.13. En el capítulo 12 se mencionan algunas situaciones específicas en que las listas doblemente enlazadas son esenciales por razones de eficiencia.

```

procedure INSERTA ( x: tipo_elemento; p: posición; var L: LISTA );
begin
    if p = 0 then begin
        { inserta en la primera posición }
        if mueve ( disponible, L ) then
            ESPACIO [L]. elemento := x
    end
    else { inserta en una posición que no es la primera }
        if mueve ( disponible, ESPACIO [p].sig ) then
            { celda que ocupará x ya apuntada por ESPACIO [p].sig }
            ESPACIO [ESPACIO [p].sig]. elemento := x
end; { INSERTA }

procedure SUPRIME ( p: posición; var L: LISTA );
begin
    if p = 0 then
        mueve (L, disponible)
    else
        mueve (ESPACIO [p].sig, disponible)
end; { SUPRIME }

```

```

procedure val_inicial;
{ val_inicial enlaza ESPACIO en una lista de celdas disponibles }
var
  i: integer;
begin
  for i := long_máx-1 downto 1 do
    ESPACIO [i].sig := i + 1;
  disponible := 1;
  ESPACIO [long_máx].sig := 0 { marca el final de la lista de celdas disponibles}
end; { val_inicial }

```

Fig. 2.12. Algunos procedimientos para listas enlazadas basadas en cursores.



Fig. 2.13. Una lista doblemente enlazada.

Otra ventaja importante de las listas doblemente enlazadas es que permiten usar un apuntador a la celda que contiene el i -ésimo elemento de una lista para representar la posición i , en vez de usar el apuntador a la celda anterior, que es menos natural. El único precio que se paga por estas características es la presencia de un apuntador adicional en cada celda, y los procedimientos algo más lentos para algunas de las operaciones básicas con listas. Si se usan apuntadores (en vez de cursores) se puede declarar que las celdas contienen un elemento y dos apuntadores, mediante

```

type
  tipo_celda = record
    elemento: tipo_elemento;
    sig, ant: ↑ tipo_celda
  end;
  posición = ↑ tipo_celda;

```

En la figura 2.14 se da un procedimiento para suprimir un elemento en la posición p en una lista doblemente enlazada. La figura 2.15 muestra los cambios causados en los apuntadores por este procedimiento; los apuntadores anteriores se representan con líneas de trazo continuo, y los nuevos con líneas punteadas, en el supuesto de que la celda suprimida no es la primera ni la última †. Primero se localiza la celda precedente, usando el campo ant . Se hace que el campo sig de esta celda apunte a

† A tal efecto, es práctica común hacer que el encabezamiento de una lista doblemente enlazada sea una celda que efectivamente «cierra el círculo». Esto es, que el campo ant del encabezado apunte a la última celda y que su campo sig apunte a la primera. De esta manera no se necesita hacer verificaciones de campos nil en la figura 2.14.

la celda que sigue a la que ocupa la posición p . Después se hace que el campo ant de esta celda siguiente apunte a la celda que precede a la que ocupa la posición p . La celda apuntada por p queda sin usar y el sistema de tiempo de ejecución de Pascal debe de usarla de nuevo automáticamente.

```

procedure SUPRIME (var p: posición );
begin
  if  $p \uparrow .ant <> \text{nil}$  then
    { la celda a suprimir no es la primera }
     $p \uparrow .ant \uparrow .sig := p \uparrow .sig;$ 
  if  $p \uparrow .sig <> \text{nil}$  then
    { la celda a suprimir no es la última}
     $p \uparrow .sig \uparrow .ant := p \uparrow .ant$ 
end; { SUPRIME }
```

Fig. 2.14. Supresión en una lista doblemente enlazada.

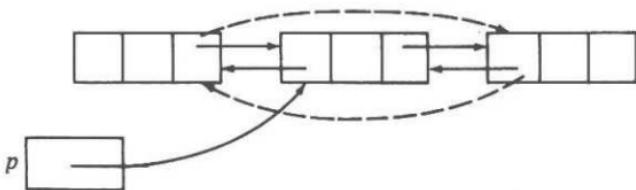


Fig. 2.15. Cambios de los apuntadores para realizar una supresión.

2.3 Pilas

Una *pila* es un tipo especial de lista en la que todas las inserciones y supresiones tienen lugar en un extremo denominado *tope*. A las pilas se les llama también «listas LIFO» (*last in first out*) o listas «último en entrar, primero en salir». El modelo intuitivo de una pila es precisamente una pila de fichas de póquer puesta sobre una mesa, o de libros sobre el piso, o de platos en una estantería, situaciones todas en las que sólo es conveniente quitar o agregar un objeto del extremo superior de la pila, al que se denominará en lo sucesivo «*tope*». Un tipo de datos abstracto de la familia PILA incluye a menudo las cinco operaciones siguientes:

1. ANULA(P) convierte la pila P en una pila vacía. Esta operación es exactamente la misma que para las listas generales.
2. TOPE(P) devuelve el valor del elemento de la parte superior de la pila P . Si se identifica la parte superior de una pila con la posición 1, como suele hacerse, entonces TOPE(P) puede escribirse en función de operaciones con listas como RECUPERA(PRIMERO(P), P).
3. SACA(P), en inglés *POP*, suprime el elemento superior de la pila, es decir, equivale a SUPRIME(PRIMERO(P), P). Algunas veces resulta conveniente implantar

SACA como una función que devuelve el elemento que acaba de suprimir, aunque aquí no se hará eso.

4. METE(x, P), en inglés *PUSH*, inserta el elemento x en la parte superior de la pila P . El anterior tope se convierte en el siguiente elemento, y así sucesivamente. En función de operaciones primitivas con listas, esta operación es INSERTA($x, \text{PRIMERO}(P), P$).
5. VACIA(P) devuelve verdadero si la pila P está vacía, y falso en caso contrario.

Ejemplo 2.2. Los editores de textos siempre permiten usar un carácter (por ejemplo, *back-space*) como *carácter de borrado* que cancela el carácter anterior no cancelado. Por ejemplo, si '#' es el carácter de borrado, la cadena *abc#d##e* es en realidad la cadena *ae*. El primer '#' cancela la *c*, el segundo la *d* y el tercero la *b*.

Los editores de textos también tienen un *carácter de eliminación de línea*, que cancela todos los caracteres anteriores de la línea actual. A efectos de este ejemplo, se usará '@' como carácter de eliminación de línea.

Un editor puede procesar una línea de texto usando una pila. El editor lee un carácter a la vez. Si el carácter leído no es de borrado ni de eliminación de línea, el editor lo mete en la pila. Si el carácter es de borrado, el editor saca un carácter de la pila, y si es de eliminación de línea, vacía la pila. En la figura 2.16 se muestra un programa que ejecuta estas acciones.

```

procedure EDITA;
var
  P: PILA;
  c: char;
begin
  ANULA (P);
  while not eoln do begin
    read(c);
    if c = '#' then
      SACA (P)
    else if c = '@' then
      ANULA (P)
    else { c es un carácter normal }
      METE (c, P)
  end;
  imprime P en orden inverso
end; { EDITA }

```

Fig. 2.16. Programa que logra el efecto de los caracteres de borrado y cancelación de línea.

En este programa, el tipo PILA debe declararse como una lista de caracteres. El proceso de escribir la pila en orden inverso en la última línea del programa tiene un objetivo: sacar de la pila un carácter a la vez da como resultado la inversión de la

línea. Algunas realizaciones de pilas, como la basada en arreglos, que se analizará a continuación, permiten escribir un procedimiento sencillo para imprimir los caracteres de la pila a partir de la base. Sin embargo, en general, para invertir una pila, cada elemento debe sacarse y meterse en otra pila; luego, los elementos pueden sacarse de la segunda pila e imprimirse en el orden en que se sacan. □

Realización de pilas basada en arreglos

Todas las realizaciones de listas descritas sirven para pilas, puesto que una pila con sus operaciones es un caso especial de lista con sus operaciones. La representación de una pila como lista enlazada es sencilla, pues METE y SACA operan sólo con la celda de encabezamiento y con la primera celda de la lista. De hecho, los encabezamientos pueden ser apuntadores o cursosres, más que celdas completas, puesto que para las pilas no existe la noción de «posición» para las listas y, por tanto, no es necesario representar la posición 1 en forma análoga a las demás posiciones.

Sin embargo, la realización de listas basada en arreglos que se dio en la sección 2.3 no es particularmente adecuada para las pilas, dado que cada METE o SACA tiene que mover la lista entera hacia arriba o hacia abajo, lo cual lleva un tiempo proporcional al número de elementos de la pila. Un mejor criterio para usar un arreglo es el que tiene en cuenta el hecho de que las inserciones y las supresiones ocurren sólo en la parte superior. Se puede anclar la base de la pila a la base del arreglo (el extremo de índice más alto) y dejar que la pila crezca hacia la parte superior del arreglo (el extremo de índice más bajo). Un cursor llamado *tope* indicará la posición actual del primer elemento de la pila. Esta idea se ilustra en la figura 2.17.

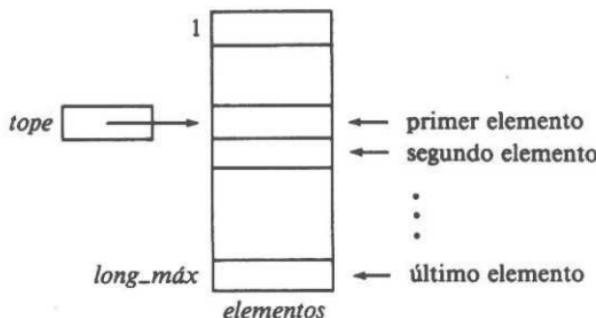


Fig. 2.17. Realización de una pila mediante un arreglo.

Para esta realización de pilas basada en arreglos, el tipo de datos abstracto PILA se define como

```

type
  PILA = record
    tope: integer;
    elementos: array[1..long_max] of tipo_elemento
  end;

```

Un ejemplo de la pila consiste en la sucesión *elementos[tope]*, *elementos[tope + 1]*, ..., *elementos[long_máx]*. Obsérvese que si *tope = long_máx + 1*, la pila está vacía.

Las cinco operaciones típicas con pilas están realizadas en la figura 2.18. Obsérvese que para que TOPE devuelva un objeto de tipo tipo_elemento, ese tipo debe ser lícito como resultado de una función. De lo contrario, TOPE debe ser un procedimiento que modifique su segundo argumento asignándole el valor TOPE(*P*) o una función que devuelva un apuntador a tipo_elemento.

```

procedure ANULA ( var P: PILA );
begin
  P.tope := long_máx + 1
end; { ANULA }

function VACIA ( P: PILA ): boolean;
begin
  if P.tope > long_máx then
    return (true)
  else
    return (false)
end; { VACIA }

function TOPE ( var P: PILA ): tipo_elemento;
begin
  if VACIA (P) then
    error ('la pila está vacía')
  else
    return (P.elementos[P.tope])
end; { TOPE }

procedure SACA ( var P: PILA );
begin
  if VACIA(P) then
    error ('la pila está vacía')
  else
    P.tope := P.tope + 1
end; { SACA }

procedure METE ( x: tipo_elemento; P: PILA );
begin
  if P.tope := 1 then
    error ('la pila está llena')
  else begin
    P.tope := P.tope - 1;
    P.elementos[P.tope] := x
  end
end; { METE }

```

Fig. 2.18. Operaciones con pilas.

2.4 Colas .

Una *cola* es otro tipo especial de lista en el cual los elementos se insertan en un extremo (el *posterior*) y se suprimen en el otro (el *anterior* o *frente*). Las colas se conocen también como listas «FIFO» (*first-in first-out*) o listas «primero en entrar, primero en salir». Las operaciones para una cola son análogas a las de las pilas; las diferencias sustanciales consisten en que las inserciones se hacen al final de la lista, y no al principio, y en que la terminología tradicional para colas y listas no es la misma. Se usarán las siguientes operaciones con colas.

1. ANULA(*C*) convierte la cola *C* en una lista vacía.
2. FRENT(*C*) es una función que devuelve el valor del primer elemento de la cola *C*. FRENT(*C*) se puede escribir en función de operaciones con listas, como RECUPERA(PRIMERO(*C*), *C*).
3. PONE_EN_COLA(*x*, *C*) inserta el elemento *x* al final de la cola *C*. En función de operaciones con listas, PONE_EN_COLA(*x*, *C*) es INSERTA(*x*, FIN(*C*), *C*).
4. QUITA_DE_COLA(*C*) suprime el primer elemento de *C*; es decir, QUITA_DE_COLA(*C*) es SUPRIME(PRIMERO(*C*), *C*).
5. VACIA(*C*) devuelve verdadero si, y sólo si, *C* es una cola vacía.

Realización de colas basada en apuntadores

Igual que en el caso de las pilas, cualquier realización de listas es lícita para las colas. No obstante, para aumentar la eficacia de PONE_EN_COLA es posible aprovechar el hecho de que las inserciones se efectúan sólo en el extremo posterior. En lugar de recorrer la lista de principio a fin cada vez que se desea hacer una inserción, se puede mantener un apuntador (o un cursor) al último elemento. Como en las listas de cualquier clase, también se mantiene un apuntador al frente de la lista; en las colas, ese apuntador es útil para ejecutar mandatos del tipo FRENT o QUITA_DE_COLA. En Pascal, se utilizará una celda ficticia como encabezamiento y se tendrá el apuntador frontal dirigido a ella. Esta convención permite manejar convenientemente una cola vacía.

Aquí se desarrollará una implantación de colas basada en apuntadores de Pascal. Se puede desarrollar una realización análoga basada en cursores, pero en el caso de las colas se dispone de una representación orientada a arreglos, mejor que la que se puede obtener por imitación directa de apuntadores mediante cursores. Al final de esta sección se analizará otra implantación, llamada de «arreglos circulares». Para la realización basada en apuntadores, se definirán las celdas como antes:

```

type
  tipo_celda = record
    tipo_elemento: tipo_elemento;
    sig: ^ tipo_celda
  end;

```

Por tanto, es posible definir una cola como una estructura que consiste en apuntadores al extremo anterior de la lista y al extremo posterior. La primera celda de una cola es una celda de encabezamiento cuyo campo elemento se ignora. Como se mencionó, esta convención permite representar de manera simple una cola vacía. Se define:

```

type
  COLA = record
    ant, post: ↑ tipo_celda
  end;

```

La figura 2.19 muestra programas para las cinco operaciones con colas. En ANULA, la primera proposición *new(C.ant)* crea una variable de tipo tipo_celda y asigna su dirección a *C.ant*. La segunda proposición coloca nil en el campo *sig* de esa celda. La tercera proposición hace que el encabezamiento quede como primera y última celda de la cola.

El procedimiento QUITA_DE_COLA(*C*) suprime el primer elemento de *C* desconectando el encabezado antiguo de la cola. El primer elemento de la lista se convierte en la nueva celda ficticia de encabezamiento.

La figura 2.20 muestra los resultados originados por la sucesión de mandatos ANULA(*C*), PONE_EN_COLA(*x, C*), PONE_EN_COLA(*y, C*), QUITA_DE_COLA(*C*). Obsérvese que después de la supresión, el elemento *x* ya no se considera parte de la cola, por estar en el campo elemento de la celda de encabezamiento.

Realización de colas con arreglos circulares

La representación de listas por medio de arreglos analizada en la sección 2.2 puede usarse para las colas, pero no es muy eficiente. Es cierto que con un apuntador al último elemento es posible ejecutar PONE_EN_COLA en un número fijo de pasos, pero QUITA_DE_COLA, que suprime el primer elemento, requiere que la cola complete ascienda una posición en el arreglo. Así pues, QUITA_DE_COLA lleva un tiempo $\Omega(n)$ si la cola tiene longitud *n*.

Para evitar este gasto, se debe adoptar un punto de vista diferente. Imagínese un arreglo como un círculo en el que la primera posición sigue a la última, en la forma sugerida en la figura 2.21. La cola se encuentra en alguna parte de ese círculo, ocupando posiciones consecutivas †, con el extremo posterior en algún lugar a la izquierda del extremo anterior. Para insertar un elemento en la cola, se mueve el apuntador *C.post* una posición en el sentido de las manecillas del reloj, y se escribe el elemento en esa posición. Para suprimir, simplemente se mueve *C.ant* una posición en el sentido de las manecillas del reloj. De esta manera, la cola se mueve en ese mismo sentido conforme se insertan y suprimen elementos. Obsérvese que utilizando

† Obsérvese que la palabra «consecutivas» debe interpretarse en un sentido circular. Esto es, una cola de longitud cuatro puede ocupar, por ejemplo, las dos últimas y las dos primeras posiciones del arreglo.

este modelo los procedimientos PONE_EN_COLA y QUITA_DE_COLA se pueden escribir de tal manera que su ejecución se realice en un número constante de pasos.

Hay una sutileza que surge en la representación de la figura 2.21 y en cualquier variante menor de esta estrategia (es decir, si *C.post* apunta a una posición adelantada con respecto al último elemento, en el sentido de las manecillas del reloj, y no a ese mismo elemento). El problema reside en que no hay manera de distinguir entre una cola vacía y una que ocupa el círculo completo, a menos que se mantenga un bit que sea verdadero si, y sólo si, la cola está vacía. Si no se está dispuesto a mantener ese bit, se debe evitar que la cola llegue a llenar todo el arreglo.

Para comprender la razón de esto, supóngase que la cola de la figura 2.21 tiene *long_máx* elementos. Entonces *C.post* debería apuntar una posición adelante de *C.ant* en el sentido contrario al de las manecillas del reloj. ¿Y qué ocurriría si la cola estuviera vacía? Para ver cómo se representa una cola vacía, considérese primero una que tenga un solo elemento. Entonces *C.ant* y *C.post* apuntan a la misma posición. Si en estas condiciones se suprime ese elemento, *C.ant* avanza una posición en el sentido de las manecillas del reloj, formando una cola vacía. Así, en una cola vacía, *C.post* está a una posición de *C.ant* en sentido contrario al de las manecillas del reloj; es decir, está exactamente en la misma posición relativa que ocuparía si la cola tuviera *long_máx* elementos. Resulta evidente, pues, que aun cuando el arreglo tenga *long_máx* lugares, no se puede permitir que la cola crezca más que *long_máx*-1, a menos que se introduzca un mecanismo para distinguir las colas vacías.

```

procedure ANULA ( var C: COLA );
begin
    new(C.ant); { crea la celda de encabezamiento }
    C.ant^.sig := nil;
    C.post := C.ant { el encabezamiento es la primera y la última celdas }
end; { ANULA }

function VACIA ( C: COLA ): boolean;
begin
    if C.ant = C. post then
        return (true)
    else
        return (false)
end; { VACIA }

function FRENTA ( C: COLA ): tipo_elemento;
begin
    if VACIA(C) then
        error('la cola está vacía')
    else
        return ( C.ant^.sig^.elemento )
end; { FRENTA }

```

```

procedure PONE_EN_COLA ( x: tipo_elemento; var C: COLA );
begin
  new ( C.post↑.sig ); { agrega una nueva celda en el extremo
                        posterior de la cola }
  C.post := C.post↑.sig;
  C.post↑.elemento := x;
  C.post↑.sig := nil
end; { PONE_EN_COLA }

procedure QUITA_DE_COLA ( var C: COLA );
begin
  if VACIA(C) then
    error ('la cola está vacía')
  else
    C.ant := C.ant↑.sig
end; { QUITA_DE_COLA }

```

Fig. 2.19. Implementación de mandatos para colas.

ANULA (C)

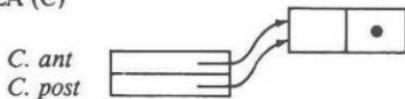
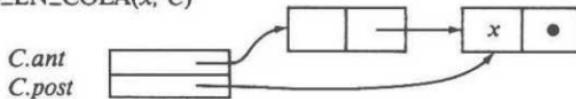
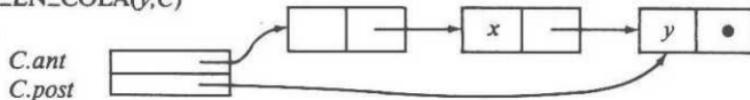
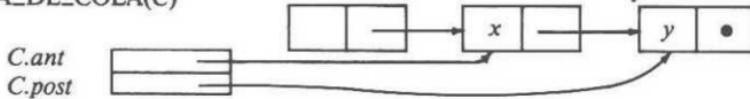
PONE_EN_COLA(*x*, *C*)PONE_EN_COLA(*y*, *C*)QUITA_DE_COLA(*C*)

Fig. 2.20. Sucesión de operaciones con colas.

Se escribirán ahora los cinco mandatos para colas usando esta representación. Formalmente, las colas se definen:

```
type
  COLA = record
    elementos: array[1..long_máx] of tipo_elemento;
    ant, post: integer
  end;
```

Los mandatos aparecen en la figura 2.22. La función *suma_uno(i)* suma uno a la posición *i*, en el sentido circular.

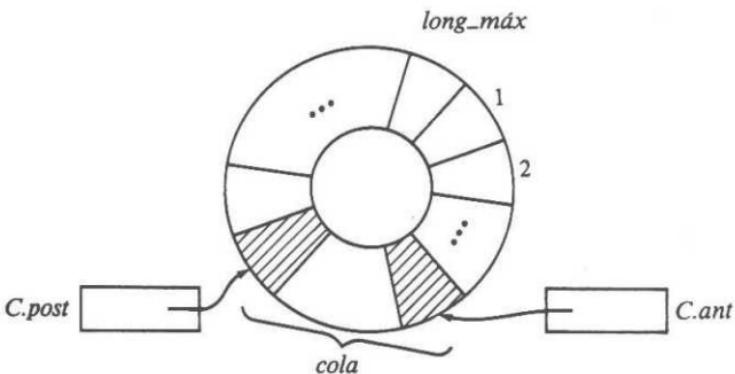


Fig. 2.21. Realización circular para colas.

2.5 Correspondencias

Una *correspondencia* o *memoria asociativa* es una función de elementos de un tipo llamado *tipo_dominio*, a elementos de otro tipo (quizás el mismo) llamado *tipo_contradicomnio*. Se expresa el hecho de que la correspondencia *M* asocia el elemento *r* de tipo *tipo_contradicomnio* con el elemento *d* de tipo *tipo_dominio* por *M(d) = r*.

Ciertas correspondencias, como *cuadrado(i) = i²*, se pueden obtener con facilidad como una función de Pascal por medio de una expresión aritmética u otro medio simple de calcular *M(d)* a partir de *d*. Sin embargo, en el caso de muchas correspondencias, la única manera clara de describir *M(d)* consiste en almacenar para cada *d* el valor de *M(d)*. Por ejemplo, para aplicar una función de nómina que asocie a cada empleado un salario semanal, parece obligatorio almacenar el salario actual de cada empleado. En lo que resta de esta sección se describirá un método para implantar funciones como la función «nómina».

Considérese qué operaciones es deseable realizar en una correspondencia *M*. Dado un elemento *d* de un tipo dominio, se podría desear obtener *M(d)* o saber si *M(d)* está definida (es decir, si *d* pertenece actualmente al dominio de *M*). O quizás

```

function suma_uno ( i : integer ): integer;
begin
    . . .
    return ((i mod long_máx) + 1)
end; { suma_uno }

procedure ANULA ( var C: COLA );
begin
    C. ant := 1;
    C. post := long_máx
end; { ANULA }

function VACIA ( var C: COLA ) : boolean;
begin
    if suma_uno(C. post) = C.ant then
        return (true)
    else
        return (false)
end; { VACIA }

function FRENTA ( var C: COLA ) : tipo_elemento;
begin
    if VACIA (C) then
        error ('la cola está vacía')
    else
        return (C.elementos[C.ant])
end; { FRENTA }

procedure PONE_EN_COLA ( x: tipo_elemento; var C: COLA );
begin
    if suma_uno(suma_uno(C.post)) = C.ant then
        error ('la cola está llena')
    else begin
        C.post := suma_uno(C.post);
        C.elementos[C.post] := x
    end
end; { PONE_EN_COLA }

procedure QUITA_DE_COLA ( var C: COLA );
begin
    if VACIA (C) then
        error ('la cola está vacía')
    else
        C.ant := suma_uno(C.ant)
end; { QUITA_DE_COLA }

```

Fig. 2.22. Realización circular de colas.

se desee introducir nuevos elementos en el dominio actual de M y establecer sus valores asociados de contradominio. Como solución distinta, se podría desechar cambiar el valor de $M(d)$. También se necesita una manera de asignar valor inicial a una correspondencia como la *correspondencia nula* o vacía, cuyo dominio está vacío. Estas operaciones se resumen en los tres mandatos siguientes:

1. ANULA (M). Hace que M sea la correspondencia nula.
2. ASIGNA (M, d, r). Define $M(d)$ como r , tanto si $M(d)$ está definido previamente como si no.
3. CALCULA (M, d, r). Devuelve verdadero y da a r el valor $M(d)$ si este último está definido; en caso contrario, devuelve falso.

Realización de correspondencias mediante arreglos

Muchas veces, el tipo dominio de una correspondencia será un tipo elemental que pueda usarse como tipo índice de un arreglo. En Pascal los tipos índice incluyen todos los subcontradominios finitos de los enteros, como $1..100$ ó $17..23$, el tipo *char* y los subcontradominios de *char*, como ' A '..' Z ', y los tipos enumerados como (norte, sur, este, oeste). Por ejemplo, un programa para descifrar textos podría guardar una correspondencia *cif*, con ' A '..' Z ' como sus tipo_dominio y tipo_contradominio, de modo que *cif*(*letra_texto*) sea la letra que en un momento dado se suponga que representa a la letra *letra_texto*.

Tales correspondencias se pueden realizar sencillamente por medio de arreglos, siempre que haya un valor del tipo_contradominio que signifique «indefinido». Por ejemplo, la correspondencia *cif* anterior podría definirse con *char* como su tipo_contradominio, en lugar de ' A '..' Z ', y '?' podría usarse para denotar «indefinido».

Supóngase que los tipos dominio y contradominio son tipo_dominio y tipo_contradominio, respectivamente, y que tipo_dominio es un tipo de Pascal básico. Entonces, se puede definir el tipo CORRESPONDENCIA (estrictamente hablando, correspondencia de tipo_dominio a tipo_contradominio) por la declaración:

```
type
  CORRESPONDENCIA = array[tipo_dominio] of tipo_contradominio;
```

En el supuesto de que «indefinido» sea una constante de tipo_contradominio y que primer_valor y último_valor sean el primero y el último valores de tipo_dominio †, es posible implantar los tres mandatos en correspondencias como en la figura 2.23.

Realización de correspondencias mediante listas

Existen muchas aplicaciones posibles de las correspondencias con dominios finitos. Por ejemplo, las tablas de dispersión (*hash*) son una excelente alternativa en mu-

† Por ejemplo, primer_valor = 'A' y último_valor = 'Z', si el tipo_dominio es ' A '..' Z '.

chas situaciones, pero su análisis se deja para el capítulo 4. Cualquier correspondencia con un dominio finito se puede representar mediante la lista de pares $(d_1, r_1), (d_2, r_2), \dots, (d_k, r_k)$, donde d_1, d_2, \dots, d_k son todos los miembros actuales del dominio, y r_i es el valor que la correspondencia asocia a d_i , para $i = 1, 2, \dots, k$. Se puede entonces usar cualquier implantación de listas que se elija para representar esa lista de pares.

Para precisar más, el tipo de datos abstracto CORRESPONDENCIA se puede aplicar por medio de listas de tipo_elemento si se define

```
type
  tipo_elemento = record
    dominio: tipo_dominio;
    contradominio: tipo_contradominio
  end;
```

y luego se define CORRESPONDENCIA como se haría con el tipo LISTA (de tipo_elemento) en la realización de listas elegida. En la figura 2.24 se definen los tres mandatos de correspondencias en función de mandatos sobre el tipo LISTA.

```
procedure ANULA ( var M: CORRESPONDENCIA );
  var
    i: tipo_dominio;
  begin
    for i := primer_valor to último_valor do
      M[i] := indefinido
    end; { ANULA }

procedure ASIGNA ( var M: CORRESPONDENCIA;
  d: tipo_dominio; r: tipo_contradominio);
  begin
    M[d] := r
  end; { ASIGNA }

function CALCULA ( var M: CORRESPONDENCIA;
  d: tipo_dominio; var r: tipo_contradominio): boolean;
  begin
    if M[d] = indefinido then
      return (false)
    else begin
      r := M[d];
      return (true)
    end
  end; { CALCULA }
```

Fig. 2.23. Realización de correspondencias mediante arreglos.

2.6 Pilas y procedimientos recursivos

Una aplicación importante de las pilas se da en la aplicación de procedimientos recursivos en los lenguajes de programación. La *organización a tiempo de ejecución* de uno de tales lenguajes es el conjunto de estructuras de datos usadas para representar los valores de las variables de un programa durante su ejecución. Todo lenguaje que, como Pascal, permita procedimientos recursivos, utiliza una pila de *registros de activación* para registrar los valores de todas las variables que pertenecen a cada procedimiento activo de un programa. Cuando se llama a un procedimiento P , se coloca en la pila un nuevo registro de activación para P , con independencia de si ya existe en ella otro registro de activación para ese mismo procedimiento. Cuando P vuelve, su registro de activación debe estar en el tope de la pila, puesto que P no puede volver si no lo han hecho previamente todos los procedimientos a los que P ha llamado. Así, se puede sacar de la pila el registro de activación correspondiente a la llamada actual de P y hacer que el control regrese al punto en el que P fue llamado (este punto, conocido como *dirección de retorno*, se colocó en el registro de activación de P al llamar a este procedimiento).

```

procedure ANULA ( var M: CORRESPONDENCIA );
{ igual que para listas }

procedure ASIGNA ( var M: CORRESPONDENCIA;
d: tipo_dominio; r: tipo_contradominio);
var
  x: tipo_elemento; { el par (d, r) }
  p: posición; { usada para ir de la primera a la última posición de
    la lista M }
begin
  x.dominio := d;
  x.contradominio := r;
  p := PRIMERO(M);
  while p <> FIN(M) do
    if RECUPERA(p, M).dominio = d then
      SUPRIME(p, M) { elimina el elemento con dominio d }
    else
      p := SIGUIENTE(p, M);
    INSERTA (x, PRIMERO(M), M) { coloca (d, r,) al inicio de la lista }
  end; { ASIGNA }

function CALCULA ( var M: CORRESPONDENCIA;
d: tipo_dominio; var r: tipo_contradominio): boolean;
var
  p: posición;
begin
  p: PRIMERO (M);
  while p <> FIN(M) do begin
    if RECUPERA(p, M).dominio = d then begin

```

```

r := RECUPERA(p, M). contradominio;
return (true)
end;
p := SIGUIENTE(p, M)
end;
return (false) { si d no está en el dominio }
end; { CALCULA }

```

Fig. 2.24. Realización de correspondencias en función de listas.

La recursión simplifica la estructura de muchos programas. Sin embargo, en algunos lenguajes las llamadas a procedimientos tienen un costo mucho mayor que el de las proposiciones de asignación, de modo que la ejecución de un programa puede resultar más rápida, en un factor constante considerable, si se eliminan las llamadas recursivas. Al decir esto no se propugna la eliminación habitual de la recursión o de otras llamadas a procedimientos; es frecuente que la sencillez estructural justifique el tiempo de ejecución. Sin embargo, podría ser deseable eliminar la recursión en las porciones de los programas que se ejecutan con mayor frecuencia, y el propósito del análisis que sigue es ilustrar cómo se pueden convertir los procedimientos recursivos en procedimientos no recursivos mediante la introducción de una pila definida por el programador.

Ejemplo 2.3. Considérense dos soluciones, una recursiva y otra no recursiva, a una versión simplificada del clásico *problema de la mochila*, en el cual se da un *objetivo* o y una colección de *pesos* p_1, p_2, \dots, p_n (enteros positivos). Se pide determinar si existe una selección de pesos que totalice exactamente o . Por ejemplo, si $o = 10$ y los pesos son 7, 5, 4, 4 y 1, se pueden elegir el segundo, el tercero y el quinto, ya que $5 + 4 + 1 = 10$.

La justificación del nombre «problema de la mochila» es que se desea llevar en la espalda no más de o kilogramos, y se tiene para elegir un conjunto de objetos de pesos dados. Se supone, además, que la utilidad de los objetos es proporcional a su peso †, y por ello se desea cargar la mochila con un peso lo más cercano posible al objetivo.

En la figura 2.25 se puede ver una función *mochila* que opera en una matriz

pesos: array[1..n] of integer.

La llamada a *mochila(s, i)* determina si existe una colección de los elementos entre *peso[i]* y *peso[n]* que sume exactamente s , e imprime sus pesos de ser tal el caso. Lo primero que *mochila* hace es determinar si puede responder de inmediato. Específicamente, si $s = 0$, entonces el conjunto vacío de pesos es una solución. Si $s < 0$, no puede haber solución, y si $s > 0$ e $i > n$, entonces ya no hay más pesos que considerar y, por tanto, no se puede encontrar una suma de pesos igual a s .

Si no se tiene ninguno de estos casos, entonces sencillamente se hace la llamada a *mochila(s - p_i, i + 1)*, para ver si existe una solución que incluya a p_i . Si esa solu-

† En el «verdadero» problema de la mochila, se dan valores de utilidad y pesos, y se pide maximizar la utilidad de los objetos transportados, con una restricción del peso.

ción existe, todo el problema está resuelto y la solución incluye a p_i , de modo que éste se imprime. Si no hay solución, se efectúa la llamada a $mochila(s, i + 1)$, para ver si existe una solución que no use p_i . \square

Eliminación de la recursión de cola

A menudo, es posible eliminar en forma mecánica la última llamada que un procedimiento se hace a sí mismo. Si un procedimiento $P(x)$ tiene como último paso una llamada a $P(y)$, entonces es posible reemplazar la llamada a $P(y)$ por una asignación $x = y$, seguida de un salto al principio del código de P . Aquí, y puede ser una expresión, pero x debe ser un parámetro pasado por valor, de modo que su valor se almacena en una localidad de memoria privada de esta llamada a P .^f Por supuesto, P podría tener más de un parámetro, en cuyo caso se tratarían exactamente igual que x e y .

```

function mochila ( objetivo: integer; candidato: integer ): boolean;
begin
(1)    if objetivo = 0 then
(2)        return ( true )
(3)    else if (objetivo < 0) or (candidato > n) then
(4)        return (false)
(5)    else { considera soluciones con y sin candidato }
(6)        if mochila (objetivo - pesos[candidato], candidato + 1) then
            begin
(6)            writeln (pesos [candidato] );
(7)            return (true)
            end
(8)        else { la única solución posible es sin candidato }
            return (mochila (objetivo, candidato + 1))
end; { mochila }
```

Fig. 2.25. Solución recursiva al problema de la mochila.

Este cambio funciona, porque volver a ejecutar P con el nuevo valor de x equivale exactamente a llamar a $P(y)$ y luego volver de esa llamada. Obsérvese que el hecho de que algunas de las variables locales de P tengan valores en la segunda llamada no tiene consecuencias. P no podría usar ninguno de esos valores, pues si se hubiera llamado a $P(y)$ como se intentaba en un principio, esos valores no habrían estado definidos.

En la figura 2.25 se ilustra otra variante de la recursión de cola; ahí, el último paso de la función $mochila$ sólo devuelve el resultado de llamarse a sí misma con otros parámetros. En una situación tal, suponiendo que los parámetros se pasan por

^f Alternativamente, x podría pasarse por referencia si y es x .

valor (o por referencia, si el mismo parámetro es el que se pasa a la llamada), se puede reemplazar la llamada por asignaciones a los parámetros y un salto al principio de la función. En el caso de la figura 2.25, se puede reemplazar la línea (8) por

```
candidato := candidato + 1;
goto principio
```

donde *principio* es una etiqueta que se va a asignar a la proposición (1). Obsérvese que no se necesita ningún cambio a *objetivo*, puesto que su valor pasa intacto como primer parámetro. De hecho, se puede advertir que, al no haber cambiado *objetivo*, las pruebas de las proposiciones (1) y (3) que lo incluyen están destinadas a fallar y, por tanto, es posible omitir las líneas (1) y (2), realizar sólo la prueba *candidato > n* en la línea (3) y luego proseguir directamente a la línea (5).

Eliminación completa de la recursión

El procedimiento de eliminación de la recursión de cola suprime la recursión por completo sólo cuando la llamada recursiva se encuentra al final del procedimiento y tiene la forma adecuada. Existe un enfoque más general que convierte cualquier procedimiento (o función) recursivo en no recursivo, pero que introduce una pila definida por el programador. En general, una celda de esa pila contendrá:

1. los valores actuales de los parámetros del procedimiento;
2. los valores actuales de todas las variables locales del procedimiento, y
3. una indicación de la dirección de retorno, esto es, del lugar a donde deberá devolver el control cuando la invocación actual del procedimiento termine.

En el caso de la función *mochila*, se puede hacer algo más sencillo. Primero, se observa que cada vez que se hace una llamada (que implica meter un registro a la pila), *candidato* se incrementa en 1. Así pues, se puede dejar *candidato* como una variable global, incrementando su valor en 1 cada vez que se mete un registro en la pila y disminuyéndolo en 1 cada vez que se saca un registro.

Una segunda simplificación posible consiste en mantener dentro de la pila una «dirección de retorno» modificada. En términos estrictos, la dirección de retorno para esta función es un lugar de otro procedimiento que llama a *mochila*, la llamada de la línea (5) o la llamada de la línea (8). Estas tres posibilidades se representan por una variable «de estado» que tiene uno de los tres valores siguientes:

1. *ninguno*, que indica que la llamada procede de fuera de la función *mochila*.
2. *incluido*, que indica la llamada de la línea (5), la cual incluye *pesos[candidato]* dentro de la solución, o
3. *excluido*, que indica la llamada de la línea (8), la cual excluye a *pesos[candidato]*.

Si se almacena esta variable de estado como la dirección de retorno, se puede manejar *objetivo* como una variable global. Cuando el estado cambia de *ninguno* a *incluido*, se sustrae *pesos[candidato]* de *objetivo*, y se suma de nuevo cuando el estado

cambia de *incluido* a *excluido*. Como ayuda para representar el efecto del retorno de *mochila* cuando indican si se ha hallado la solución, se usa una variable global *bandera-exito*. Una vez que *bandera-exito* toma el valor verdadero, lo conserva y hace que se saquen de la pila los registros, imprimiendo aquellos pesos que tienen asociado un estado *incluido*. Con estas modificaciones, se puede declarar la pila como una pila de estados, mediante.

type

estados = (*ninguno*, *incluido*, *excluido*);

PILA = { declaración adecuada para una pila de estados }

La figura 2.26 muestra el procedimiento resultante no recursivo *mochila*, que opera en un arreglo de *pesos* como antes. Aunque este procedimiento puede ser más rápido que la función *mochila* original, es claramente más largo y más difícil de entender. Por ello sólo se debe eliminar la recursión cuando la velocidad sea muy importante.

```

procedure mochila (objetivo: integer);
var
  candidato: integer;
  bandera-exito: boolean;
  P: PILA;
begin
  candidato := 1;
  bandera-exito := false;
  ANULA(P);
  METE(ninguno, P); { asigna valor inicial a la pila considerando pesos[1] }
  repeat
    if bandera-exito then begin
      { saca de la pila e imprime los pesos incluidos en la solución }
      if TOPE(P) = incluido then
        writeln (pesos[candidato]);
      candidato := candidato-1;
      SACA(P)
    end
    else if objetivo = 0 then begin { se encontró la solución }
      bandera-exito := true;
      candidato := candidato -1;
      SACA(P)
    end
    else if (((objetivo < 0) and (TOPE(P) = ninguno))
      or (candidato > n)) then begin
      { no hay solución posible con las elecciones hechas }
      candidato := candidato -1;
      SACA(P)
    end
  end

```

```

end
else { aún no hay decisión, se considera el estado del candidato actual }
if TOPE( $P$ ) = ninguno then begin { se intenta primero incluyendo al
candidato }
    objetivo := objetivo-pesos[candidato];
    candidato := candidato +1;
    SACA( $P$ ); METE(incluido,  $P$ ); METE (ninguno,  $P$ )
end
else if TOPE( $P$ ) = excluido then begin { se intenta excluyendo al
candidato }
    objetivo := objetivo + pesos[candidato];
    candidato := candidato +1;
    SACA( $P$ ); METE(excluido,  $P$ ); METE(ninguno,  $P$ )
end
else begin { TOPE( $P$ ) = excluido; la elección actual no da resultado }
    SACA( $P$ );
    candidato := candidato -1
end
until VACIA ( $P$ )
end; { mochila }

```

Fig. 2.26. Procedimiento no recursivo para el problema de la mochila.

Ejercicios

- 2.1 Escribase un programa que imprima los elementos de una lista. En los ejercicios siguientes úsense operaciones con listas para realizar los programas.
- 2.2 Escribanse programas para insertar, suprimir y localizar un elemento en una lista clasificada, usando realizaciones de
 - a) arreglos
 - b) apuntadores y
 - c) cursores.

¿Cuál es el tiempo de ejecución de cada uno de estos programas?
- 2.3 Escribase un programa para intercalar
 - a) dos listas clasificadas,
 - b) n listas clasificadas.
- 2.4 Escribase un programa para concatenar una lista de listas.
- 2.5 Supóngase que se desea manipular polinomios de la forma $p(x) = c_1x^{e_1} + c_2x^{e_2} + \dots + c_nx^{e_n}$, donde $e_1 > e_2 > \dots > e_n \geq 0$. Un polinomio de ese tipo se puede representar mediante una lista enlazada en la que cada celda

tiene tres campos: uno para el coeficiente c_i , otro para el exponente e_i , y otro para el apuntador a la siguiente celda. Escribáse un programa para diferenciar polinomios representados de esta manera.

- 2.6 Escribanse programas para sumar y multiplicar polinomios similares a los del ejercicio 2.5. ¿Cuál es el tiempo de ejecución de los programas en función del número de términos?
- *2.7 Supóngase que se declaran celdas mediante

```
type
  tipo_celda = record
    bit: 0..1;
    sig: ^ tipo_celda
  end;
```

Un número binario $b_1 b_2 \dots b_n$, donde cada b_i es 0 ó 1, tiene el valor numérico $\sum_{i=1}^n b_i 2^{n-i}$. Este número se puede representar por la lista b_1, b_2, \dots, b_n .

Esta lista, a su vez, puede representarse como una lista enlazada de celdas de tipo tipo_celda. Escribase un procedimiento *incrementa(número_bin)* que sume uno al número binario apuntado por *número_bin*. Sugerencia: Hágase *incrementa* recursivo.

- 2.8 Escribase un procedimiento para intercambiar los elementos de las posiciones p y *SIGUIENTE(p)* de una lista enlazada sencilla.
- *2.9 El siguiente procedimiento se hizo con el propósito de suprimir todas las apariciones de un elemento x de una lista L . Explíquese por qué no siempre funciona y sugírerase una manera de arreglarlo de modo que realice la tarea para la que fue propuesto.

```
procedure suprime (x:tipo_elemento; var L: LISTA);
  var
    p: posición;
  begin
    p:= PRIMERO (L);
    while p <> FIN (L) do begin
      if RECUPERA (p, L) = x then
        SUPRIME (p, L);
      p:= SIGUIENTE (p, L)
    end
  end; { suprime }
```

- 2.10 Se desea almacenar una lista en un arreglo A cuyas celdas contienen dos campos: *datos*, que contiene un elemento, y *posición*, que da la posición (entera) del elemento. Un entero *último* indica que la lista ocupa las posiciones $A[1]$ a $A[\text{último}]$. El tipo LISTA se puede definir como:

```

type
  LISTA = record
    ultimo: integer;
    elementos: array[1..long_máx] of record
      datos: tipo_elemento;
      posición: integer
    end
  end;

```

Escríbese un procedimiento SUPRIME(p, L) que elimine el elemento de la posición p . Inclúyanse todas las verificaciones necesarias para detectar errores.

- 2.11 Supóngase que L es una LISTA y que p, q y r son posiciones. En función de la longitud n de la lista L , determínese cuántas veces se ejecutan en el siguiente programa las funciones PRIMERO, FIN y SIGUIENTE.

```

 $p := \text{PRIMERO}(L);$ 
while  $p <> \text{FIN}(L)$  do begin
   $q := p;$ 
  while  $q <> \text{FIN}(L)$  do begin
     $q := \text{SIGUIENTE}(q, L);$ 
     $r := \text{PRIMERO}(L);$ 
    while  $r <> q$  do
       $r := \text{SIGUIENTE}(r, L)$ 
    end;
     $p := \text{SIGUIENTE}(p, L)$ 
  end;

```

- 2.12 Reescríbese el código de las operaciones con listas, suponiendo una representación de listas enlazadas, pero sin celdas de encabezado. Supóngase que se usan apuntadores verdaderos y que la posición 1 se representa por nil.
- 2.13 Agréguese al procedimiento de la figura 2.12 todas las verificaciones necesarias para detectar errores.
- 2.14 Otra representación de listas por medio de arreglos consiste en insertar, como en la sección 2.2, pero suprimir simplemente reemplazando el elemento en cuestión por un valor especial «suprimido», que se supone ya no aparece en las listas. Reescríbanse las operaciones con listas para aplicar esta estrategia. ¿Cuáles son las ventajas y las desventajas de este enfoque en comparación con la representación original de listas mediante arreglos?

- 2.15** Supóngase que se desea utilizar un bit extra en los registros de colas, que indique si una cola está vacía o no. Modifíquense las declaraciones y las operaciones para una cola circular, de manera que se tenga en cuenta esta característica. ¿Cabe esperar que el cambio valga la pena?
- 2.16** Una *cola doble* o de doble extremo es una lista en la cual se pueden insertar o suprimir elementos en cualquiera de los extremos. Desarróllense realizaciones para colas dobles basadas en arreglos, apuntadores y cursosres.
- 2.17** Definase un TDA que maneje las operaciones PONE_EN_COLA, QUIITA_DE_COLA y ESTA_EN_COLA. ESTA_EN_COLA(x) es una función que devuelve verdadero o falso dependiendo de si x está o no en la cola.
- 2.18** ¿Cómo podría obtenerse una cola cuyos elementos son cadenas de longitud arbitraria? ¿Cuánto tiempo lleva poner en cola una cadena?
- 2.19** Otra posible implantación de colas mediante listas enlazadas consiste en no usar una celda de encabezamiento y hacer que *ant* apunte directamente a la primera celda. Si la cola está vacía, se hace que *ant - post = nil*. Obténgase las operaciones con colas para esta representación. ¿Cómo se compara esta aplicación con la que se dio en la sección 2.4, en función de la velocidad, la utilización de espacio y la concisión del código?
- 2.20** Una variante de la cola circular registra la posición del elemento frontal y la longitud de la cola.
- ¿Es necesario en esta realización limitar la longitud de la cola a $long_máx - 1$?
 - Escríbanse las cinco operaciones con colas para esta realización.
 - Compárese esta realización con la de colas circulares de la sección 2.4.
- 2.21** Es posible guardar dos pilas en un solo arreglo, si una de ellas crece desde la posición 1 del arreglo y la otra lo hace desde la última posición. Escríbase un procedimiento METE(x, P) que inserte el elemento x en la pila P , donde P es una u otra de las dos pilas. Inclúyanse en el procedimiento todas las verificaciones de error necesarias.
- 2.22** Se pueden almacenar k pilas en un solo arreglo si se usa la estructura de datos sugerida en la figura 2.27 para el caso $k = 3$. Para meter y sacar de cada pila se opera como se sugiere en la sección 2.3, en relación con la figura 2.17. Sin embargo, si una inserción en la pila i hace que TOPE(i) igualle a BASE($i - 1$), antes de efectuarla hay que desplazar todas las pilas, de modo que quede una separación del tamaño apropiado entre cada par de pilas adyacentes. Por ejemplo, es posible hacer que las separaciones entre pilas sean todas iguales, o que la separación que queda encima de la pila i sea proporcional al tamaño actual de la pila i (en el supuesto de que las pilas más grandes tienen mayor probabilidad de crecer antes, y se desea posergar la siguiente reorganización el mayor tiempo posible).
- Si se dispone de un procedimiento *reorganiza*, al que se llama si las pilas colisionan, escríbase el código de las cinco operaciones con pilas.

- b) Si existe un procedimiento *haz_nuevos_topes* que calcula *nuevo_topo[i]*, la posición «apropiada» para el tope de la pila *i*, con $1 \leq i \leq k$, escribase el procedimiento *reorganiza*. Sugerencia: Obsérvese que la pila *i* puede moverse arriba o abajo y que hay que mover la pila *i* antes que la *j* si la nueva posición de *j* se traslapa con la posición antigua de *i*. Considerérense las pilas 1, 2, 3 ..., *k* en orden, pero manténgase una pila de «objetivos» consistentes en desplazar una pila. Si al considerar la pila *i* ésta se puede mover con seguridad, hágase, y reconsiderérese luego la pila cuyo número esté en el tope de la pila de objetivos. Si no hay seguridad, métase *i* en la pila de objetivos.
- c) ¿Qué realización es apropiada para la pila de objetivos de b)? ¿Es realmente necesario mantenerla como una lista de enteros, o sería suficiente con una representación más sucinta?
- d) Obténgase *haz_nuevos_topes* de forma que el espacio que está sobre todas las pilas sea proporcional a los tamaños actuales de éstas.
- e) ¿Qué modificaciones habría que hacer a la figura 2.27 para que esa realización pueda trabajar con colas? ¿Y con listas generales?
- 2.23 Modifíquense las realizaciones de SACA y PONE_EN_COLA de las secciones 2.3 y 2.4, de modo que devuelven el elemento suprimido de la pila o cola. ¿Qué modificaciones habría que hacer si el tipo de los elementos no es alguno que pueda ser devuelto por una función?
- 2.24 Usese una pila para eliminar la recursión de los siguientes procedimientos.
- a) **function** *comb* (*n, m* : integer): integer;
 {calcula $\binom{n}{m}$ suponiendo que $0 \leq m \leq n$ y $n \geq 1$ }
begin
 if (*n* = 1) or (*m* = 0) or (*m* = *n*) then
 return (1)
 else
 return (*comb* (*n* - 1, *m*) + *comb* (*n* - 1, *m* - 1))
end; {*comb*}

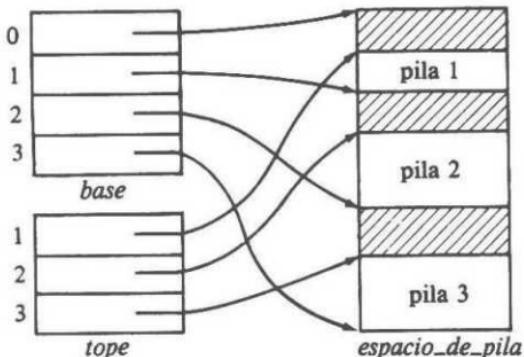


Fig. 2.27. Varias pilas en un arreglo.

```

b) procedure invierte (var L: LISTA);
   { invierte la lista L }
  var
    x: tipo_elemento;
  begin
    if not VACIA(L) then begin
      x := RECUPERA(PRIMERO(L), L);
      SUPRIME(PRIMERO(L), L);
      invierte(L);
      INSERTA(x, FIN(L), L)
    end
  end; { invierte }

```

- *2.25 ¿Es posible eliminar la recursión de cola de los programas del ejercicio 2.24?
De ser así, hágase.

Notas bibliográficas

Knuth [1968] contiene información adicional sobre la realización de listas, pilas y colas. Ciertos lenguajes de programación como LISP y SNOBOL, manejan listas y cadenas de manera apropiada. Véanse Sammet [1969], Nicholls [1975], Pratt [1975] o Wexelblat [1981] sobre una historia y una descripción de muchos de estos lenguajes.

3

Arboles

Un árbol impone una estructura jerárquica sobre una colección de objetos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles. Entre otras cosas, los árboles son útiles para analizar circuitos eléctricos y para representar la estructura de fórmulas matemáticas. También se presentan naturalmente en diversas áreas de computación. Por ejemplo, se usan para organizar información en sistemas de bases de datos y para representar la estructura sintáctica de un programa fuente en los compiladores. En el capítulo 5 se describe el uso de árboles en la representación de datos. En este libro se usan muchas variantes de árboles. En este capítulo se presentan las definiciones básicas y algunas de las operaciones más comunes con árboles. Despues se describen algunas de las estructuras de datos más frecuentemente usadas para representar árboles que permiten manejar esas operaciones con eficiencia.

3.1 Terminología fundamental

Un árbol es una colección de elementos llamados *nodos*, uno de los cuales se distingue como *raíz*, junto con una relación (de «paternidad») que impone una estructura jerárquica sobre los nodos. Un nodo, como un elemento de una lista, puede ser del tipo que se desee. A menudo se representa un nodo por medio de una letra, una cadena de caracteres o un círculo con un número en su interior. Formalmente, un *árbol* se puede definir de manera recursiva como sigue:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es también la raíz de dicho árbol.
2. Supóngase que n es un nodo y que A_1, A_2, \dots, A_k son árboles con raíces n_1, n_2, \dots, n_k , respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n_1, n_2, \dots, n_k . En dicho árbol, n es la raíz y A_1, A_2, \dots, A_k son los *subárboles* de la raíz. Los nodos n_1, n_2, \dots, n_k reciben el nombre de *hijos* del nodo n .

A veces, conviene incluir entre los árboles el *árbol nulo*, un «árbol» sin nodos que se representa mediante Λ .

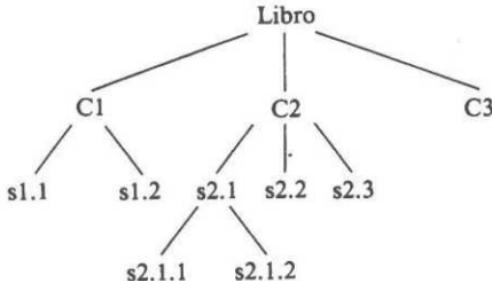
Ejemplo 3.1. Considérese el índice general de un libro, que se representa en la figura 3.1(a). Tal índice es un árbol. Se puede redibujar en la forma mostrada en la

figura 3.1(b). La relación padre-hijo entre dos nodos se representa por una línea que los une. Los árboles normalmente se dibujan de arriba hacia abajo, como en la figura 3.1(b), con el padre encima de los hijos.

La raíz, el nodo llamado «Libro», tiene tres subárboles que corresponden a los capítulos C1, C2 y C3. Esta relación se representa por medio de las líneas descendentes que unen a Libro con C1, C2 y C3. Libro es el padre de C1, C2 y C3, y estos tres nodos son los hijos de Libro.

Libro
C1
s1.1
s1.2
C2
s2.1
s2.1.1
s2.1.2
s2.2
s2.3
C3

(a)



(b)

Fig. 3.1. Un índice general y su representación como árbol.

El tercer subárbol de Libro, que tiene raíz C3, es un árbol de un solo nodo, en tanto que los otros dos subárboles tienen una estructura no trivial. Por ejemplo, el subárbol con raíz C2 tiene tres subárboles que corresponden a las secciones s2.1, s2.2 y s2.3; estas dos últimas son árboles de un solo nodo, mientras que la primera tiene dos subárboles que corresponden a las subsecciones s2.1.1 y s2.1.2. □

El ejemplo 3.1 es típico de cierta clase de datos cuya mejor representación se logra mediante un árbol. En el ejemplo, la relación de paternidad representa inclusión: un nodo padre está constituido por sus hijos, como Libro está constituido por C1, C2 y C3. En esta obra se encontrarán otras relaciones que se pueden representar por la relación de paternidad en un árbol.

Si n_1, n_2, \dots, n_k , es una sucesión de nodos de un árbol tal que n_i es el padre de n_{i+1} para $1 \leq i < k$, entonces la secuencia se denomina *camino* del nodo n_1 al nodo n_k . La *longitud* de un camino es el número de nodos del camino menos 1. Por tanto, hay un camino de longitud cero de cualquier nodo a sí mismo. Por ejemplo, en la figura 3.1 hay un camino de longitud 2, a saber, (C2, s2.1, s2.1.2), de C2 a s2.1.2.

Si existe un camino de un nodo a a otro b , entonces a es un *antecesor* de b , y b es un *descendiente* de a . Por ejemplo, en la figura 3.1, los antecesores de s2.1 son él mismo, C2 y Libro. Obsérvese que cada nodo es a la vez un antecesor y un descendiente de sí mismo.

Un antecesor o un descendiente de un nodo que no sea él mismo recibe el nombre de *antecesor propio* o *descendiente propio*, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes pro-

pios se denomina *hoja*. Un subárbol de un árbol es un nodo junto con todos sus descendientes.

La *altura* de un nodo en un árbol es la longitud del camino más largo de ese nodo a una hoja. En la figura 3.1 el nodo C1 tiene altura 1, C2 altura 2 y el nodo C3 altura 0. La *altura del árbol* es la altura de la raíz. La *profundidad* de un nodo es la longitud del camino único desde la raíz a ese nodo.

Orden de los nodos

A menudo los hijos de un nodo se ordenan de izquierda a derecha. Así, los dos árboles de la figura 3.2 son diferentes porque los dos hijos del nodo *a* aparecen en distintos órdenes en los dos árboles. Si se desea ignorar explícitamente el orden de los hijos, se habla entonces de un árbol *no ordenado*.

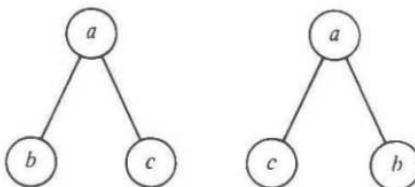


Fig. 3.2. Dos árboles (ordenados) distintos.

El orden de izquierda a derecha de los *hermanos* (hijos del mismo nodo) se puede extender para comparar dos nodos cualesquiera entre los cuales no exista la relación antecesor-descendiente. La regla que se aplica es que si *a* y *b* son hermanos y *a* está a la izquierda de *b*, entonces todos los descendientes de *a* están a la izquierda de todos los descendientes de *b*.

Ejemplo 3.2. Considérese el árbol de la figura 3.3. El nodo 8 está a la derecha del nodo 2, a la izquierda de los nodos 9, 6, 10, 4 y 7, y no está a la izquierda ni a la derecha de sus antecesores 1, 3 y 5.

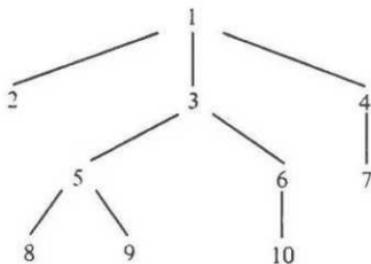


Fig. 3.3. Un árbol.

Dado un nodo n , una regla sencilla para determinar qué nodos están a su izquierda y cuáles a su derecha, consiste en dibujar el camino de la raíz a n . Todos los nodos que salen a la izquierda de este camino, y todos sus descendientes, están a la izquierda de n . Los nodos, y sus descendientes, que salen a la derecha, están a la derecha de n . \square

Orden previo, orden posterior y orden simétrico

Existen varias maneras útiles de ordenar sistemáticamente todos los nodos de un árbol. Los tres ordenamientos más importantes se llaman orden previo (*preorder*), orden simétrico (*inorder*) y orden posterior (*postorder*); tales ordenamientos se definen recursivamente como sigue:

- Si un árbol A es nulo, entonces la lista vacía es el listado de los nodos de A en los órdenes previo, simétrico y posterior.
- Si A contiene un solo nodo, entonces ese nodo constituye el listado de los nodos de A en los órdenes previo, simétrico y posterior.

Si ninguno de los anteriores es el caso, sea A un árbol con raíz n y subárboles A_1, A_2, \dots, A_k , como se representa en la figura 3.4.

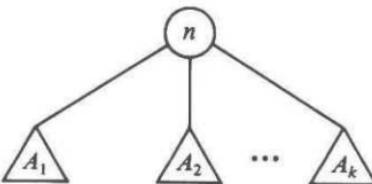


Fig. 3.4. Árbol A .

1. El *listado en orden previo* (o *recorrido en orden previo*) de los nodos de A está formado por la raíz de A , seguida de los nodos de A_1 en orden previo, luego por los nodos de A_2 en orden previo y así sucesivamente hasta los nodos de A_k en orden previo.
2. El *listado en orden simétrico* de los nodos de A está constituido por los nodos de A_1 en orden simétrico, seguidos de n y luego por los nodos de A_2, \dots, A_k , con cada grupo de nodos en orden simétrico.
3. El *listado en orden posterior* de los nodos de A tiene los nodos de A_1 en orden posterior, luego los nodos de A_2 en orden posterior y así sucesivamente hasta los nodos de A_k en orden posterior y por último la raíz n .

La figura 3.5(a) muestra el esbozo de un procedimiento para listar los nodos de un árbol en orden previo. Para convertirlo en un procedimiento de recorrido en orden posterior, basta invertir el orden de los pasos (1) y (2). La figura 3.5(b) es un esbozo de un procedimiento de recorrido en orden simétrico. En los tres casos, el or-

denamiento deseado de un árbol se produce llamando al procedimiento apropiado en la raíz del árbol.

Ejemplo 3.3. Listense en orden previo los nodos del árbol de la figura 3.3. Primero se lista 1, y luego se llama recursivamente a ORD_PRE en el primer subárbol de 1, o sea el subárbol con raíz 2. Este subárbol tiene un solo nodo, de manera que simplemente se lista. Luego se sigue con el segundo subárbol de 1, el que tiene raíz 3. Se lista 3 y luego se vuelve a llamar a ORD_PRE en el primer subárbol de 3. Esta llamada origina que se listen 5, 8 y 9, en ese orden. Continuando de esta forma, se obtiene el recorrido completo en orden previo de la figura 3.3: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.

```
procedure ORD_PRE ( n: nodo );
begin
```

- (1) lista n ;
- (2) for cada hijo c de n , si los hay, en orden desde la izquierda do
 ORD_PRE(c)
 end; { ORD_PRE }

(a) procedimiento ORD_PRE.

```
procedure ORD_SIM ( n: nodo );
```

```
begin
```

```
if  $n$  es una hoja then
```

```
    lista  $n$ 
```

```
else begin
```

```
    ORD_SIM(hijo de  $n$  de más a la izquierda);
```

```
    lista  $n$ ;
```

```
    for cada hijo  $c$  de  $n$ , excepto el de más a la izquierda, en orden  
        desde la izquierda do
```

```
        ORD_SIM( $c$ )
```

```
    end
```

```
end; { ORD_SIM }
```

(b) procedimiento ORD_SIM.

Fig. 3.5. Procedimientos recursivos de ordenamiento.

De manera similar, simulando la figura 3.5(a) con el orden de los pasos invertido, se puede descubrir que el recorrido en orden posterior de la figura 3.3 es 2, 8, 9, 5, 10, 6, 3, 7, 4, 1. Simulando la figura 3.5(b), se encuentra que el listado en orden simétrico de la figura 3.3 es 2, 1, 8, 5, 9, 3, 10, 6, 7, 4. \square

Un truco útil para producir los tres ordenamientos de nodos es el siguiente. Imáginese que se camina por la periferia del árbol, partiendo de la raíz, y que se avanza en sentido contrario al de las manecillas del reloj, manteniéndose tan cerca del árbol

como sea posible; el camino previsto para la figura 3.3 está representado en la figura 3.6.

Para el orden previo se lista un nodo la primera vez que se pasa por él. En el caso del orden posterior, se lista un nodo la última vez que se pasa por él, conforme se sube hacia su padre. Tratándose del orden simétrico, se lista una hoja la primera vez que se pasa por ella, y un nodo interior, la segunda vez que se pasa por él. A manera de ejemplo, en la figura 3.6 se pasa por el nodo 1 al empezar, y otra vez al pasar por la «bahía» entre los nodos 2 y 3. Obsérvese que el orden de las hojas en los tres ordenamientos corresponde al mismo ordenamiento de izquierda a derecha de las hojas. Sólo el orden de los nodos interiores y su relación con las hojas varía entre los tres ordenamientos.

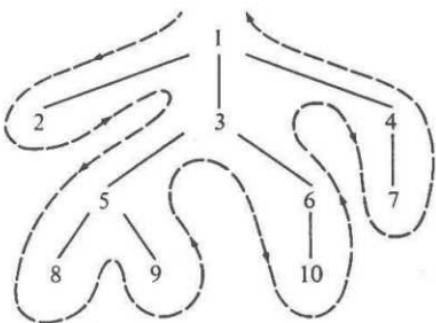


Fig. 3.6. Recorrido de un árbol.

Arboles etiquetados y árboles de expresiones

A menudo es útil asociar una *etiqueta*, o valor, a cada nodo de un árbol, siguiendo la misma idea con que se asoció un valor a un elemento de una lista en el capítulo anterior. Esto es, la etiqueta de un nodo no será el nombre del nodo, sino un valor «almacenado» en él. En algunas aplicaciones, incluso se cambiará la etiqueta de un nodo sin modificar su nombre. Una analogía útil a este respecto es: árbol es a lista como etiqueta es a elemento y nodo es a posición.

Ejemplo 3.4. La figura 3.7 muestra un árbol etiquetado que representa la expresión aritmética $(a + b) * (a + c)$, donde n_1, n_2, \dots, n_7 son los nombres de los nodos, cuyas etiquetas se muestran, por convención, en las proximidades de los nodos correspondientes. Las reglas para representar una expresión mediante un árbol etiquetado son éstas:

1. Cada hoja está etiquetada con un operando y sólo consta de ese operando. Por ejemplo, el nodo n_4 representa la expresión a .
2. Cada nodo interior n está etiquetado con un operador. Supóngase que n está etiquetado con el operador binario θ , como $+ o *$, y que el hijo izquierdo de n re-

presenta la expresión E_1 y el hijo derecho la expresión E_2 . Entonces, n representa la expresión $(E_1) \theta (E_2)$. Los paréntesis se pueden quitar si no son necesarios.

Por ejemplo, el nodo n_2 tiene al operador $+$ como etiqueta, y sus hijos izquierdo y derecho representan las expresiones a y b , respectivamente. Por tanto, n_2 representa $(a) + (b)$, o, más simple, $a + b$. El nodo n_1 representa $(a + b) * (a + c)$, puesto que $*$ es la etiqueta de n_1 , y $a + b$ y $a + c$ son las correspondientes expresiones representadas por n_2 y n_3 . \square

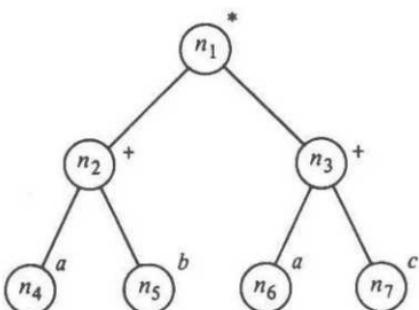


Fig. 3.7. Árbol de expresión con etiquetas.

A menudo, cuando se recorre un árbol en orden previo, simétrico o posterior, se prefiere listar las etiquetas de los nodos, en vez de sus nombres. En el caso de un árbol que representa una expresión, el listado en orden previo de las etiquetas da lo que se conoce como forma *prefija* de la expresión, en la cual un operador precede a sus operandos izquierdo y derecho. Para precisar, la expresión prefija correspondiente a un solo operando a es a mismo. La expresión prefija correspondiente a $(E_1) \theta (E_2)$, donde θ es un operador binario, es $\theta P_1 P_2$, donde P_1 y P_2 son las expresiones prefijas correspondientes a E_1 y E_2 . Obsérvese que en la expresión prefija no se necesitan paréntesis, dado que es posible revisar la expresión prefija $\theta P_1 P_2$ e identificar únicamente a P_1 como el prefijo más corto (y único) de $P_1 P_2$, que es además una expresión prefija válida.

Por ejemplo, el listado en orden previo de las etiquetas de la figura 3.7 es $*+ab+ac$. La expresión prefija correspondiente a n_2 , que es $+ab$, es el prefijo válido más corto de $+ab+ac$.

De manera similar, el listado en orden posterior de las etiquetas de un árbol de expresión da lo que se conoce como representación *postfija* (o *polaca*). La expresión $(E_1) \theta (E_2)$ se representa con la expresión postfija $P_1 P_2 \theta$, donde P_1 y P_2 son las representaciones postfijas de E_1 y E_2 , respectivamente. De nuevo, los paréntesis son innecesarios, porque se puede identificar a P_2 buscando el sufijo más corto de $P_1 P_2$ que sea una expresión postfija válida. Por ejemplo, la expresión postfija correspondiente a la figura 3.7 es $ab+ac+*$. Si se escribe una expresión en la forma $P_1 P_2 *$, entonces P_1 es $ac+$, el sufijo más corto de $ab+ac+$ que es una expresión postfija válida.

El recorrido en orden simétrico de un árbol de expresión da la expresión infija misma, pero sin paréntesis. Por ejemplo, el listado en orden simétrico de las etiquetas de la figura 3.7 es $a+b * a+c$. Se invita a elaborar un algoritmo que recorra un árbol de expresión, y que produzca una expresión infija con todos los pares de paréntesis necesarios.

Determinación de los antecesores

Los recorridos de un árbol en los órdenes previo y posterior permiten determinar los antecesores de un nodo. Supóngase que $ord_post(n)$ es la posición del nodo n en el listado en orden posterior de los nodos de un árbol, y que $desc(n)$ es el número de descendientes propios del nodo n . Por ejemplo, en el árbol de la figura 3.7 se verá cómo las posiciones en orden posterior de los nodos n_2 , n_4 y n_5 son 3, 1 y 2, respectivamente.

Las posiciones en orden posterior de los nodos tienen la útil propiedad de que los nodos de un subárbol con raíz n ocupan posiciones consecutivas de $ord_post(n)-desc(n)$ a $ord_post(n)$. Para determinar si un vértice x es descendiente de un vértice y , basta verificar que se cumple

$$ord_post(y)-desc(y) \leq ord_post(x) \leq ord_post(y)$$

Una propiedad similar se cumple para el recorrido en orden previo.

3.2 EL TDA ARBOL

En el capítulo 2, se vieron las listas, pilas, colas y correspondencias como tipos de datos abstractos (TDA). En este capítulo, los árboles se considerarán como TDA y también como estructuras de datos. Uno de los usos más importantes de los árboles se presenta en el diseño de implantaciones de varios TDA que se estudian en este libro. Por ejemplo, en la sección 5.1 se verá cómo un «árbol binario de búsqueda» puede usarse para obtener tipos de datos abstractos basados en el modelo matemático de un conjunto junto con operaciones como INSERTA, SUPRIME y MIEMBRO (esta última para probar si un elemento es o no elemento de un conjunto). Los dos capítulos siguientes presentan otras realizaciones de varios TDA mediante árboles.

En esta sección se describirán varias operaciones útiles con árboles y se verá cómo diseñar algoritmos para árboles en función de esas operaciones. Igual que con las listas, hay una gran variedad de operaciones que se pueden efectuar con árboles. Aquí se considerarán las siguientes:

1. **PADRE(n , A)**. Esta función devuelve el padre del nodo n en el árbol A . Si n es la raíz, que no tiene padre, se devuelve Λ . En este contexto, Λ es un «nodo nulo», que se usa como señal de que se ha salido del árbol.
2. **HIJO_MAS_IZQ(n , A)** devuelve el hijo más a la izquierda del nodo n en el árbol A , y devuelve Λ si n es una hoja y, por tanto, no tiene hijos.

3. HERMANO_DER(n, A) devuelve el hermano a la derecha del nodo n en el árbol A , el cual se define como el nodo m que tiene el mismo padre p que n , de forma que m está inmediatamente a la derecha de n en el ordenamiento de los hijos de p . Por ejemplo, para el árbol de la figura 3.7, HIJO_MAS_IZQ(n_2) = n_4 , HERMANO_DER(n_4) = n_5 y HERMANO_DER(n_5) = Λ .
4. ETIQUETA(n, A) devuelve la etiqueta del nodo n en el árbol A . Sin embargo, no se requiere que haya etiquetas definidas para cada árbol.
5. CREA*i*(v, A_1, A_2, \dots, A_i) es un miembro de una familia infinita de funciones, una para cada valor de $i = 0, 1, 2, \dots$. CREA*i* crea un nuevo nodo r con etiqueta v y le asigna i hijos que son las raíces de los árboles A_1, A_2, \dots, A_i , en ese orden desde la izquierda. Se devuelve el árbol con raíz r . Obsérvese que si $i = 0$, entonces r es a la vez una hoja y la raíz.
6. RAIZ(A) devuelve el nodo raíz del árbol A , o Λ si A es el árbol nulo.
7. ANULA(A) convierte A en el árbol nulo.

Ejemplo 3.5. Escribábase un procedimiento recursivo y otro que no lo sea, para listar las etiquetas de los nodos de un árbol en orden previo. Se supone que los tipos de datos nodo y ARBOL ya están definidos, y que el tipo de datos ARBOL es para árboles con etiquetas del tipo tipo_etiqueta. La figura 3.8 muestra un procedimiento recursivo que, dado un nodo n , lista las etiquetas del subárbol con raíz n en orden previo. Para obtener un listado en orden previo correspondiente al árbol A se hace la llamada ORD_PRE(RAIZ(A)).

```

procedure ORD_PRE (  $n$ : nodo );
  { lista las etiquetas de los descendientes de  $n$  en orden previo }
  var
    c: nodo;
  begin
    imprime (ETIQUETA( $n, A$ ));
     $c :=$  HIJO_MAS_IZQ( $n, A$ );
    while  $c <> \Lambda$  do begin
      ORD_PRE( $c$ );
       $c :=$  HERMANO_DER( $c, A$ )
    end
  end;
end; { ORD_PRE }

```

Fig. 3.8. Procedimiento recursivo de listado en orden previo.

También se desarrollará un procedimiento no recursivo para imprimir las etiquetas en orden previo. Para ubicarse en el árbol se usará una pila P , cuyo tipo PILA es, en realidad, «pila de nodos». La idea básica del algoritmo es que cuando se esté en un nodo n , la pila contendrá el camino de la raíz a n , con la raíz en la base de la pila y el nodo n en la parte superior †.

† Recuérdese el análisis sobre recursividad de la sección 2.6, donde se mostró que la implantación de un procedimiento recursivo requiere una pila de registros de activación. Si analiza la figura 3.8, se observa

Una manera de efectuar un recorrido de un árbol en orden previo no recursivo se da en el programa ORD_PRE_NO_REC de la figura 3.9. Este programa tiene dos modos de operación. En el primero, desciende por el camino más a la izquierda aún no explorado del árbol, imprimiendo y apilando los nodos a lo largo del camino, hasta que alcanza una hoja.

Luego, el programa entra en el segundo modo de operación, en el cual regresa por el camino apilado, sacando de la pila los nodos del camino hasta que encuentra un nodo que tiene un hermano derecho. El programa vuelve entonces al primer modo de operación, iniciando el descenso a partir de ese hermano derecho aún no explorado.

El programa empieza en el primer modo de operación en la raíz, y termina cuando la pila queda vacía. En la figura 3.9 se muestra el programa completo.

3.3 Realizaciones de árboles

En esta sección se presentarán varias realizaciones básicas de árboles y se analizarán sus funciones para manejar las operaciones con árboles presentadas en la sección 3.2.

Representación de árboles mediante arreglos

Sea A un árbol cuyos nodos tienen nombres $1, 2, \dots, n$. Tal vez la representación más sencilla de A que es capaz de manejar la operación PADRE sea un arreglo lineal L en la cual cada entrada $L[i]$ constituya un apuntador o un cursor al padre del nodo i . La raíz de A se puede distinguir dándole como padre un apuntador nulo o que apunte a sí misma. En Pascal no se pueden usar apuntadores a elementos de arreglos, por tanto, habrá que emplear un esquema de cursos donde $L[i] = j$ si el nodo j es el padre del nodo i , y $L[i] = 0$ si el nodo i es la raíz.

Esta representación se basa en la propiedad de que en los árboles cada nodo tiene un parente único, y permite hallar el parente de un nodo en un tiempo constante. Un camino ascendente en un árbol, es decir de un nodo a su parente, de éste a su parente y así sucesivamente, se puede recorrer en un tiempo proporcional al número de nodos del camino. También es posible manejar la operación ETIQUETA agregando otro arreglo E , tal que $E[i]$ sea la etiqueta del nodo i , o haciendo que los elementos del arreglo L sean registros que contengan un entero (cursor) y una etiqueta.

Ejemplo 3.6. El árbol de la figura 3.10(a) tiene la representación por apuntadores al parente que se muestra en el arreglo A de la figura 3.10(b). □

que cuando se llama a ORD_PRE(n), las llamadas a procedimientos activas, y por tanto la pila de registros de activación, corresponden a las llamadas a ORD_PRE para todos los antecesores de n . Así, el procedimiento no recursivo de orden previo, como el ejemplo de la sección 2.6, modela con exactitud la forma de implantar el procedimiento recursivo.

La representación por apuntadores al padre no facilita las operaciones que requieren información de los hijos. Dado un nodo n , resulta caro determinar sus hijos o su altura. Además, esta representación no especifica el orden de los hijos de un nodo. Por tanto, las operaciones como HIJO_MAS_IZQ y HERMANO_DER no están bien definidas. Se puede imponer un orden artificial, por ejemplo, numerando los hijos de cada nodo después de numerar al padre, y numerando los hijos en orden

```

procedure ORD_PRE_NO_REC ( A: ARBOL );
  { recorrido en orden previo no recursivo del árbol A }
  var
    m: nodo; { temporal }
    P: PILA; { pila de nodos que contiene el camino de la raíz al padre TO-
              PE(P) del nodo "actual" m }
  begin
    { asigna valor inicial }
    ANULA(P);
    m := RAIZ(A);
    while true do
      if m <> Λ then begin
        print(ETIQUETA(m, A));
        METE(m, P);
        { explora el hijo más a la izquierda de m }
        m := HIJO_MAS_IZQ(m, A)
      end
      else begin
        { la exploración del camino contenido en la pila está completa }
        if VACIA(P) then
          return;
        { explora el hermano derecho del nodo al tope de la pila }
        m := HERMANO_DER(TOPE(P), A);
        SACAR(P)
      end
    end; { ORD_PRE_NO_REC }
  
```

Fig. 3.9. Procedimiento no recursivo de listado en orden previo.

ascendente, de izquierda a derecha. Con esa suposición se ha escrito la función HERMANO_DER de la figura 3.11, para los tipos nodo y ARBOL que se definen como sigue:

```

type
  nodo = integer;
  ARBOL = array [1..nodos_máx] of nodo;
  
```

Para esta representación se supone que el nodo nulo Λ se representa por 0.

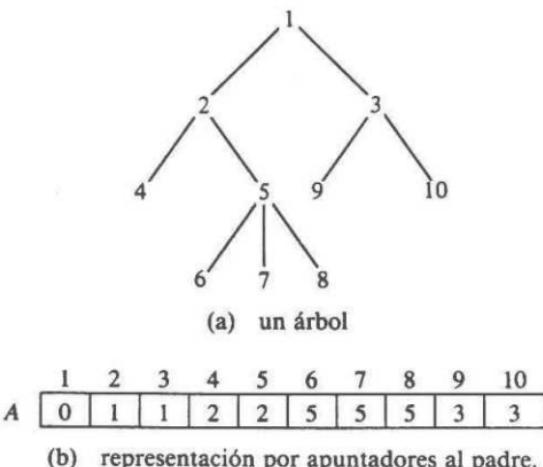


Fig. 3.10. Un árbol y su representación mediante apuntadores al padre.

```

function HERMANO_DER ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el hermano derecho del nodo n del árbol A }
  var
    i, padre: nodo;
  begin
    padre := A[n];
    for i := n + 1 to nodos_máx do
      { busca un nodo después de n que tenga el mismo padre }
      if A[i] = padre then
        return (i);
      return (0) { devuelve el nodo nulo si no se encontró hermano
                  derecho }
  end; { HERMANO_DER }
```

Fig. 3.11. Operación HERMANO_DER usando la representación por arreglos.

Representación de árboles mediante listas de hijos

Una manera importante y útil de representar árboles consiste en formar una lista de los hijos de cada nodo. Las listas se pueden representar con cualquiera de los métodos sugeridos en el capítulo 2, pero como el número de hijos que cada nodo puede tener es variable, la representación por medio de listas enlazadas es a menudo más apropiada.

La figura 3.12 sugiere cómo se podría representar el árbol de la figura 3.10(a). Hay un arreglo de celdas de encabezamiento indizadas por nodos, los cuales se supone

que están numerados del 1 al 10. Cada encabezamiento apunta a una lista enlazada de «elementos» que son nodos del árbol. Los elementos de la lista encabezada por *encabezamiento[i]* son los hijos del nodo *i*; por ejemplo, 9 y 10 son los hijos de 3.

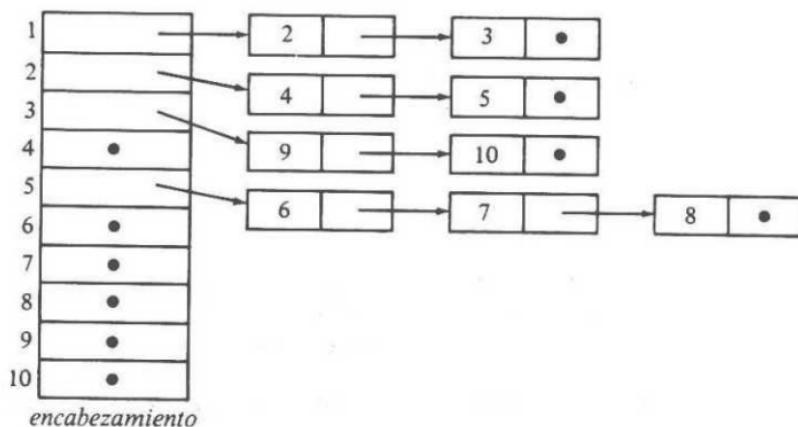


Fig. 3.12. Representación de un árbol por listas enlazadas.

Primero se desarrollarán las estructuras de datos necesarias en función de un tipo de datos abstracto LISTA (de nodos), y luego se dará una realización particular para las listas y se verá cómo las abstracciones compaginan entre sí. Despues se mostrarán algunas simplificaciones posibles. Se empieza con las siguientes declaraciones de tipo:

```
type
  nodo = integer;
  LISTA = { definición apropiada para listas de nodos };
  posición = { definición apropiada para posiciones en listas };
  ARBOL = record
    encabezamiento: array [1..nodos_máx] of LISTA;
    etiquetas: array [1..nodos_máx] of tipo_etiqueta;
    raíz: nodo
  end;
```

Se supone que la raíz de cada árbol está almacenada explícitamente en el campo *raíz*, y que se usa 0 para representar el nodo nulo.

La figura 3.13 muestra el código para la operación HIJO_MAS_IZQ. Como ejercicio, sería útil escribir el código para las restantes operaciones.

```
function HIJO_MAS_IZQ ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el hijo más a la izquierda del nodo n del árbol A }
  var
    L: LISTA; { abreviatura para la lista de hijos de n }
```

```

begin
  L := A.encabezamiento[n];
  if VACIA(L) then { n es una hoja }
    return (0)
  else
    return (RECUPERA(PRIMERO(L), L))
end; { HIJO_MAS_IZQ }

```

Fig. 3.13. Función para hallar el hijo más a la izquierda.

Ahora se elegirá una implantación particular para las listas, en la cual tanto LISTA como posición sean enteros, empleados como cursores a un arreglo *espacio_celdas* de registros:

```

var
  espacio_celdas: array [1..nodos_máx] of record
    nodo: integer;
    sig: integer
  end;

```

Para simplificar, no se insistirá en que las listas de hijos tienen celdas de encabezamiento. Más bien, se hará que *A.encabezamiento[n]* apunte directamente a la primera celda de la lista, como se sugiere en la figura 3.12. La figura 3.14(a) muestra la función HIJO_MAS_IZQ de la figura 3.13, escrita otra vez para esta realización en particular. La figura 3.14(b) muestra el operador PADRE, que es más difícil de escribir usando esta representación de listas, puesto que se requiere una búsqueda en todas ellas para determinar en cuál aparece un nodo dado.

```

function HIJO_MAS_IZQ ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el hijo más a la izquierda del nodo n del árbol A }
  var
    L: integer; { un cursor al principio de la lista de hijos de n }
  begin
    L := A.encabezado[n];
    if L = 0 then { n es una hoja }
      return (0)
    else
      return (espacio_celdas[L].nodo)
  end; { HIJO_MAS_IZQ }

```

(a) La función HIJO_MAS_IZQ.

```

function PADRE ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el padre del nodo n del árbol A }

```

```

var
  p: nodo; { recorre los posibles padres de n }
  i: posición; { recorre la lista de hijos de p }
begin
  for p := 1 to nodos_máx do begin
    i := A.encabezado[p];
    while i <> 0 do { verifica si n está entre los hijos de p }
      if espacio_celdas[i].nodo = n then
        return (p)
      else
        i := espacio_celdas[i].sig
  end;
  return (0) { devuelve el nodo nulo si no se encontró padre }
end; { PADRE }

```

(b) La función PADRE.

Fig. 3.14. Dos funciones que emplean representación por listas enlazadas de árboles.

Representación «hijo más a la izquierda — hermano derecho»

La estructura de datos descrita antes adolece, entre otras cosas, de no ser adecuada para crear árboles grandes a partir de árboles más chicos por medio de los operadores CREA*i*. La razón es que, a pesar de que todos los árboles comparten *espacio_celdas* para las listas ligadas de hijos, cada uno tiene su arreglo propio de encabezamientos correspondiente a sus nodos. Por ejemplo, si se deseara obtener CREA2(*v*, *A*₁, *A*₂), se tendría que copiar *A*₁ y *A*₂ en un tercer árbol y agregar un nuevo nodo con etiqueta *v* y dos hijos: las raíces de *A*₁ y *A*₂.

Si se desea construir árboles a partir de otros menores, es mejor que las representaciones de los nodos de todos los árboles compartan un área. La extensión lógica de la figura 3.12 para lograr esto consiste en reemplazar el arreglo de encabezamientos por un arreglo *espacio_nodos* que contenga registros con dos campos *etiqueta* y *encabezamiento*. Este arreglo contendrá los encabezamientos de la totalidad de nodos de todos los árboles. Así pues, se declara:

```

var
  espacio_nodos: array [1..nodos_máx] of record
    etiqueta: tipo_etiqueta;
    encabezamiento: integer { cursor a espacio_celdas }
  end;

```

Entonces, puesto que ya no hay nodos con nombres 1, 2, ..., *n*, sino que los nodos están representados por índices arbitrarios de *espacio_nodos*, no es posible que el campo *nodo* de *espacio_celdas* represente el «número» de un nodo; en cambio, *nodo* es ahora un cursor a *espacio_nodos*, que indica la posición de ese nodo. El tipo ARBOL es sencillamente un cursor a *espacio_nodos* que indica la posición de la raíz.

Ejemplo 3.7. La figura 3.15(a) muestra un árbol, y la figura 3.15(b), la estructura de datos correspondiente, en la cual los nodos con etiquetas *A*, *B*, *C*, y *D* se han colocado de manera arbitraria en las posiciones 10, 5, 11 y 2 de *espacio_nodos*. También se ha hecho una elección arbitraria de las celdas de *espacio_celdas* utilizadas para las listas de hijos. □

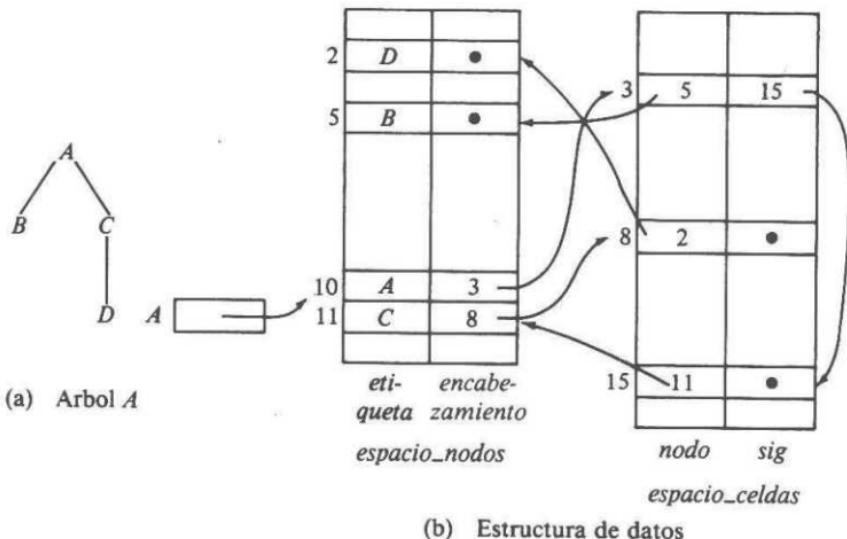


Fig. 3.15. Otra estructura de listas enlazadas para árboles.

La estructura de la figura 3.15(b) es adecuada para combinar árboles por medio de operaciones CREAr. Sin embargo, esta estructura se puede simplificar de modo significativo. Primero, se observa que las cadenas de apuntadores *sig* de *espacio_celdas* son en realidad apuntadores a hermanos derechos.

Usando estos apuntadores, se pueden obtener los hijos más a la izquierda como sigue. Supóngase que *espacio_celdas[i].nodo = n*. (Recuérdese que el «nombre» de un nodo, en oposición a su etiqueta, es en realidad su índice en *espacio_nodos*, que es el valor de *espacio_celdas[i].nodo*.) Entonces, *espacio_nodos[n].encabezamiento* indica la celda ocupada por el hijo más a la izquierda de *n* en *espacio_celdas*, en el sentido de que el campo *nodo* de esa celda es el nombre de tal nodo en *espacio_nodos*.

Se pueden simplificar las cosas si se identifica un nodo no por su índice en *espacio_nodos*, sino por el índice de la celda de *espacio_celdas* que lo representa como hijo. Entonces los apuntadores *sig* de *espacio_celdas* apuntarán en realidad a hermanos derechos, y la información contenida en el arreglo *espacio_nodos* se podrá guardar introduciendo un campo *hijo_más_izq* en *espacio_celdas*. El tipo de datos ARBOL se convertirá en un entero empleado como cursor a *espacio_celdas* para indicar la raíz del árbol. Se declara *espacio_celdas* para tener la siguiente estructura:

```

var
  espacio_celdas: array [1..nodos_máx] of record
    etiqueta: tipo_etiqueta;
    hijo_más_izq: integer;
    hermano_der: integer
  end;

```

Ejemplo 3.8. El árbol de la figura 3.15(a) se representa con la nueva estructura de datos en la figura 3.16. Para los nodos se han utilizado los mismos índices arbitrarios de la figura 3.15(b). □

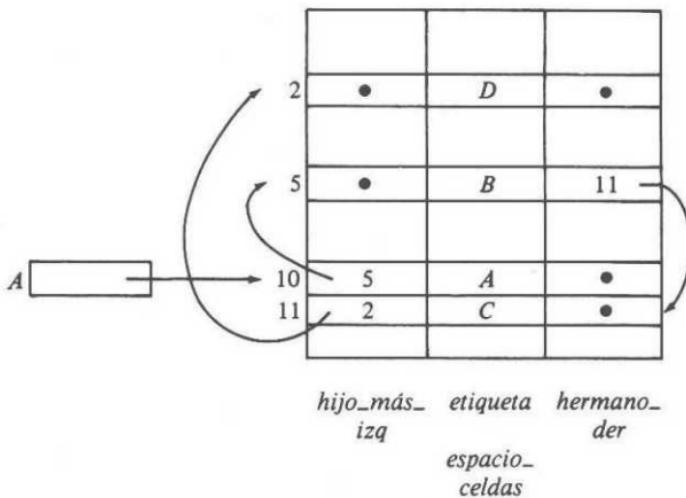


Fig. 3.16. Representación "hijo más a la izquierda-hermano derecho" de un árbol.

En la representación hijo más a la izquierda-hermano derecho, todas las operaciones son fáciles de realizar, excepto PADRE. Esta requiere de una búsqueda en todo *espacio_celdas*. Si es necesario realizar la operación PADRE eficientemente, se puede agregar a *espacio_celdas* un cuarto campo que indique al padre de un nodo directamente.

Como ejemplo de una operación con árboles escrita para utilizar una estructura hijo más a la izquierda-hermano derecho, como la de la figura 3.16, se da la función CREA2 en la figura 3.17. Se supone que las celdas no utilizadas están enlazadas en una lista de espacio disponible, encabezada por *disp*, y que este enlace utiliza los campos hermano derecho. La figura 3.18 muestra los apuntadores antiguos (con líneas de trazo continuo) y los nuevos (con líneas punteadas).

```

function CREA2 ( v: tipo_etiqueta; A1, A2: integer ) : integer;
  { devuelve un nuevo árbol con raíz v, que tiene A1 y A2 como subárboles }
  var
    temp: integer; { guarda el índice de la primera celda disponible
      para la raíz del nuevo árbol }
  begin
    temp := disp;
    disp := espacio_celdas[disp].hermano_der;
    espacio_celdas[temp].hijo_más_izq := A1;
    espacio_celdas[temp].etiqueta := v;
    espacio_celdas[temp].hermano_der := 0;
    espacio_celdas[A1].hermano_der := A2;
    espacio_celdas[A2].hermano_der := 0; { no es necesario; ese
      campo debería valer 0, pues la celda era una raíz }
    return (temp)
  end; { CREA2 }

```

Fig. 3.17. La función CREA2.

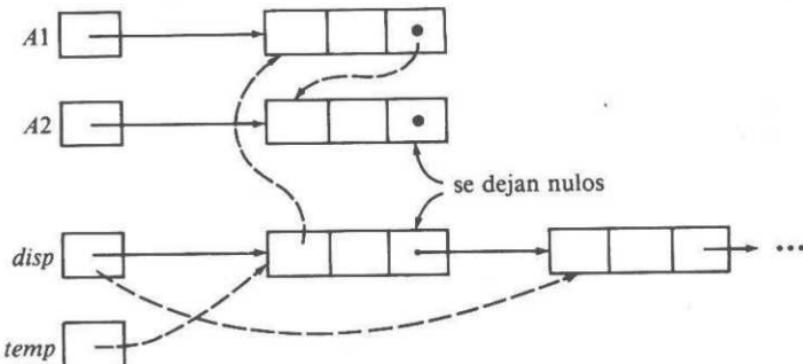


Fig. 3.18. Cambios de apuntadores producidos por CREA2.

Como alternativa, se puede usar menos espacio, pero más tiempo, si se coloca en el campo hermano derecho del hijo más a la derecha un apuntador al padre, en lugar del apuntador nulo que, de lo contrario, estaría ahí. Para evitar confusiones, se necesitaría en cada celda un bit que indicase si el campo hermano derecho tiene un apuntador al hermano derecho o al padre.

Así, dado un nodo, se encontraría su padre siguiendo apuntadores hermano derecho hasta encontrar uno que fuera apuntador al padre. Como todos los hermanos tienen el mismo padre, se encontraría el camino al padre del nodo del que se partió. El tiempo necesario para encontrar el padre de un nodo en esta representación depende del número de hermanos que tenga el nodo.

3.4 Arboles binarios

El árbol que se definió en la sección 3.1 recibe a veces el nombre de árbol *ordenado orientado*, porque los hijos de cada nodo están ordenados de izquierda a derecha, y porque existe un camino orientado (un camino en una dirección particular) de cada nodo a sus descendientes. Otro concepto útil y muy distinto de «árbol» es el de *árbol binario*, que define un árbol vacío o un árbol en el que cada nodo no tiene hijos, tiene un *hijo izquierdo*, un *hijo derecho* o un hijo izquierdo y un hijo derecho. El hecho de que cada hijo se designe en este caso como hijo izquierdo o como hijo derecho hace que un árbol binario sea distinto de un árbol ordenado orientado, tal como se definió en la sección 3.1.

Ejemplo 3.9. Si se adopta la convención de que los hijos izquierdos se dibujan hacia la izquierda de su padre y los hijos derechos hacia la derecha, las figuras 3.19(a) y (b) representan dos árboles binarios distintos, aunque ambos «se parecen» al árbol ordinario (ordenado orientado) de la figura 3.20. Sin embargo, se hace hincapié en que los de la figura 3.19 no son iguales entre sí ni son en ningún sentido iguales al de la figura 3.20, por la sencilla razón de que los árboles binarios no son directamente comparables con los árboles ordinarios. Por ejemplo, en la figura 3.19(a), 2 es el hijo izquierdo de 1, que no tiene hijo derecho; en cambio, en la figura 3.19(b), 1 no tiene hijo izquierdo, pero tiene a 2 como hijo derecho. En ambos árboles binarios, 3 es el hijo izquierdo de 2 y su hijo derecho es 4. □

Los listados en órdenes previo y posterior de un árbol binario son similares a los de un árbol ordinario dados en la página 79. El listado en orden simétrico de los nodos de un árbol binario con raíz n , subárbol izquierdo A_1 y subárbol derecho A_2 , es el listado en orden simétrico de A_1 seguido de n y del listado en orden simétrico de A_2 . Por ejemplo, 35241 es el listado en orden simétrico de los nodos del árbol de la figura 3.19(a).

Representación de árboles binarios

Una forma conveniente de estructurar datos para representar un árbol binario consiste en dar a sus nodos los nombres 1, 2, ..., n , y utilizar un arreglo de registros declarado como

```

var
    espacio_celdas: array [1..nodos_máx] of record
        hijo_izq: integer;
        hijo_der: integer
    end;

```

La idea es que $espacio_celdas[i].hijo_izq$ sea el hijo izquierdo del nodo i , y que suceda lo mismo con $hijo_der$. Un valor 0 en cualquiera de esos campos indicará la ausencia de un hijo.

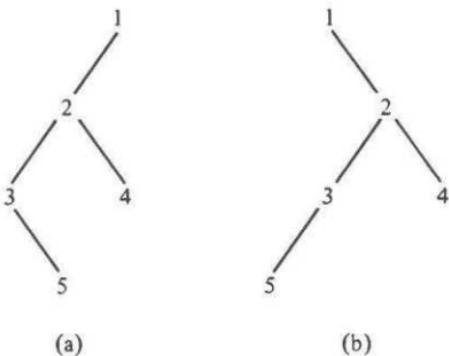


Fig. 3.19. Dos árboles binarios.

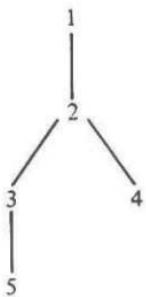


Fig. 3.20. Un árbol «ordinario».

Ejemplo 3.10. El árbol binario de la figura 3.19(a) puede representarse como se muestra en la figura 3.21. □

Un ejemplo: códigos de Huffman

Para exemplificar la forma en que los árboles binarios pueden usarse como estructuras de datos, se verá un problema particular que consiste en la construcción de «códigos de Huffman». Supóngase que se tienen mensajes compuestos por secuencias de caracteres. En cada mensaje, los caracteres son independientes entre sí, y aparecen con una probabilidad conocida en cualquier posición dada del mensaje; las probabilidades son las mismas para todas las posiciones. Como ejemplo, supóngase que se tiene un mensaje constituido por los caracteres *a*, *b*, *c*, *d*, *e*, los cuales aparecen con probabilidades 0.12, 0.4, 0.15, 0.08 y 0.25, respectivamente.

Se desea codificar cada carácter por medio de una sucesión de ceros y unos, de manera que ningún código de un carácter sea prefijo del código de otro carácter. Esta propiedad de prefijos permitiría decodificar una cadena de ceros y unos, eliminado repetidas veces de la cadena los prefijos que sean códigos de caracteres.

	<i>hijo_izq</i>	<i>hijo_der</i>
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Fig. 3.21. Representación de un árbol binario.

Ejemplo 3.11. La figura 3.22 muestra dos códigos posibles para el alfabeto de cinco símbolos que se está utilizando. Es evidente que el código 1 tiene la propiedad de prefijos, puesto que ninguna sucesión de tres bits puede ser prefijo de otra sucesión con igual número de bits. El algoritmo de decodificación para el código 1 es sencillo. Basta tomar tres bits de cada vez y traducir cada grupo de éstos en un carácter. Por supuesto, las sucesiones 101, 110 y 111 no pueden presentarse si la cadena de bits codifica realmente caracteres de acuerdo con el código 1. Por ejemplo, si se recibe el mensaje codificado 001010011, se puede saber que el mensaje original era *bcd*.

Símbolo	Probabilidad	Código 1	Código 2
<i>a</i>	0.12	000	000
<i>b</i>	0.40	001	11
<i>c</i>	0.15	010	01
<i>d</i>	0.08	011	001
<i>e</i>	0.25	100	10

Fig. 3.22. Dos códigos binarios.

También es fácil verificar que el código 2 tiene la propiedad de prefijos. Se puede decodificar una cadena de bits tomando repetidas veces los prefijos que sean códigos de caracteres y quitándolos, tal como se hizo con el código 1. La única diferencia consiste en que no se puede «rebanar» la sucesión completa de bits de una sola vez, porque tomar dos o tres bits para un carácter depende de los bits. Por ejemplo, si una cadena comienza con 1101001, se puede asegurar de nuevo que los caracteres codificados fueron *bcd*. Los dos primeros bits, 11, deben proceder de *b*, por tanto, es posible quitarlos y preocuparse sólo de 01001. Se deduce, entonces, que los bits 01 proceden de *c*, y así sucesivamente. □

El problema que se enfrenta es: dado un conjunto de caracteres y sus probabilidades, encontrar un código con la propiedad de prefijos, tal que la longitud media del código de un carácter sea mínima. La razón por la que se desea minimizar la longitud media de un código es que ello permite comprimir la longitud de un mensaje tipo. Cuanto más corta sea la longitud media de un carácter, tanto más corta será la longitud de un mensaje codificado. Por ejemplo, el código 1 tiene una longitud

media de código de 3. Esta longitud se obtiene multiplicando las longitudes de los códigos de todos los símbolos por sus respectivas probabilidades de aparición. El código 2 tiene una longitud media de 2.2, puesto que los símbolos a y d , que en conjunto aparecen el 20% de las veces, tienen códigos de longitud 3, y los otros símbolos, códigos de longitud 2.

¿Se puede obtener un código mejor que el 2? Una respuesta completa a esta pregunta consistiría en exhibir un código con la propiedad de prefijos que tuviera una longitud media de 2.15. Ese sería el mejor código posible para las probabilidades dadas de aparición de los símbolos. Una técnica que permite hallar códigos de prefijos óptimos recibe el nombre de *algoritmo de Huffman*. Funciona seleccionando dos caracteres, como a y b , que tengan las probabilidades más pequeñas, y reemplazándolos con un único carácter (imaginario), por ejemplo x , cuya probabilidad de aparición sea la suma de las probabilidades de a y b . Despues se encuentra un código de prefijos óptimo para este conjunto más pequeño de caracteres, usando recursivamente el mismo procedimiento. Los códigos para el conjunto original de caracteres se obtienen usando el código para x seguido de un cero, como código para a , y seguido de un uno, como código para b .

Se pueden considerar los códigos de prefijos como caminos en árboles binarios. Piénsese que seguir el camino de un nodo a su hijo izquierdo es como agregarle un cero a un código, y que tomar el camino hacia el hijo derecho es como agregarle un uno. Si se etiquetan las hojas de un árbol binario con los caracteres representados, se puede simbolizar cualquier código de prefijos por medio de un árbol binario. La propiedad de prefijos garantiza que ningún carácter puede tener un código que corresponda a un nodo interior y, reciprocamente, etiquetar las hojas de cualquier árbol binario con caracteres da un código con la propiedad de prefijos de esos caracteres.

Ejemplo 3.12. Los árboles binarios correspondientes a los códigos 1 y 2 de la figura 3.22 se muestran en la figura 3.23(a) y (b), respectivamente. □

Se obtendrá el algoritmo de Huffman mediante un *bosque* o conjunto de árboles, cada uno de los cuales tendrá sus hojas etiquetadas con los caracteres cuyos códigos se desea seleccionar, y sus raíces etiquetadas con la suma de las probabilidades de todas las etiquetas de las hojas. Estas sumas se denominan *pesos* de los árboles. Inicialmente, cada carácter estará en un árbol de un solo nodo, y cuando el algoritmo termine se tendrá un solo árbol con todos los caracteres como hojas. En este árbol, el camino de la raíz a cualquier hoja representará el código para la etiqueta de esa hoja, de acuerdo con el esquema izquierda = 0, derecha = 1 de la figura 3.23.

El paso fundamental del algoritmo consiste en seleccionar los dos árboles del bosque que tengan los pesos más bajos (decidiendo arbitrariamente en caso de empate), y en combinar esos dos árboles en uno solo, cuyo peso sea la suma de los pesos de ambos. Para combinar los árboles se crea un nuevo nodo que queda como raíz que tiene las raíces de los dos árboles dados como sus hijos izquierdo y derecho (sin importar cuál sea uno y otro). Este proceso continúa hasta que quede un solo árbol representativo del código que, para las probabilidades dadas, tenga la menor longitud media de código posible.

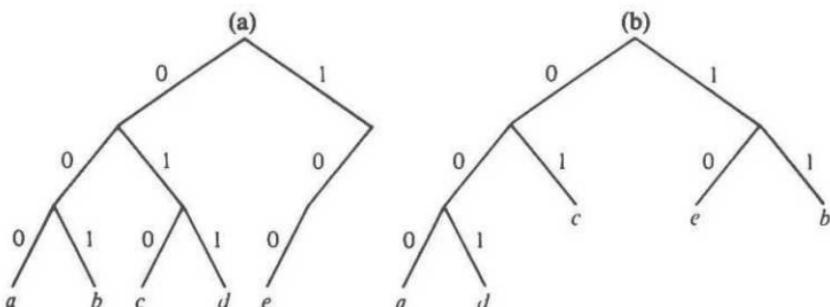


Fig. 3.23. Arboles binarios que representan códigos con la propiedad de prefijos.

Ejemplo 3.13. La secuencia de pasos seguidos para los caracteres y probabilidades del ejemplo que se está desarrollando se muestra en la figura 3.24. En la parte (e), se puede ver que las palabras de código para *a*, *b*, *c*, *d* y *e* son 1111, 0, 110, 1110 y 10. En este ejemplo, hay un solo árbol no trivial, pero, en general, puede haber varios. Por ejemplo, si las probabilidades de *b* y *e* fueran 0.33 y 0.32, entonces, después del paso (c) de la figura habría que combinar *b* y *e*, en vez de añadir *e* al árbol más grande, como se hizo en el paso (d). □

Se describen ahora las estructuras de datos necesarias. Primero, se usa un arreglo ARBOL de registros del tipo

```

record
    hijo_izq: integer;
    hijo_der: integer;
    padre: integer
end

```

para representar los árboles binarios. Los apunadores al padre facilitan el hallazgo de caminos de las hojas a la raíz, que permiten descubrir los códigos de los caracteres. En segundo término, se usa un arreglo ALFABETO de registros del tipo

```

record
    símbolo: char;
    probabilidad: real;
    hoja: integer { cursor a un árbol }
end

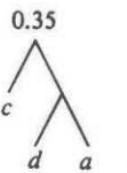
```

0.12	0.40	0.15	0.08	0.25
•	•	•	•	•
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>

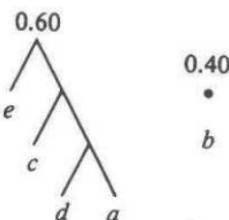
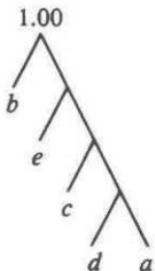
(a) Inicial

0.20	0.40	0.15	0.25
•	•	•	•
<i>d</i>	<i>a</i>	<i>b</i>	<i>c</i>

(b) Combina *a* y *d*

(c) Combina a, d con c

0.40 0.25
• •
b e

(d) Combina a, c, d con e 

(e) Arbol final

Fig. 3.24. Pasos para la construcción de un árbol de Huffman.

para asociar a cada símbolo del alfabeto a codificar la hoja que le corresponda. Este arreglo también registra la probabilidad de cada carácter. En tercer lugar, se necesita un arreglo *BOSQUE* de registros que representen los árboles mismos. El tipo de esos registros es

record

peso: real;

raíz: integer { cursor a un árbol }

end

Los valores iniciales de todos los arreglos correspondientes a los datos de la figura 3.24(a) se muestran en la figura 3.25. En la figura 3.26 se muestra un esbozo del programa para construir el árbol de Huffman.

1	0.12	1
2	0.40	2
3	0.15	3
4	0.08	4
5	0.25	5

peso raíz

1	a	0.12	1
2	b	0.40	2
3	c	0.15	3
4	d	0.08	4
5	e	0.25	5

símbolo- proba- hoja
lo bilitad

ALFABETO

1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

hijo_ hijo_ padre
izq der

ARBOL

BOSQUE**Fig. 3.25.** Contenido inicial de los arreglos.

```

(1) while haya más de un árbol en el bosque do
begin
(2)     i := índice del árbol de menor peso en BOSQUE;
(3)     j := índice del segundo árbol de menor peso en BOSQUE;
(4)     crea un nuevo nodo con hijo izquierdo BOSQUE[i].raíz
           e hijo derecho BOSQUE[j].raíz;
(5)     reemplaza el árbol i de BOSQUE por un árbol cuya
           raíz es el nuevo nodo y cuyo peso es
           BOSQUE[i].peso + BOSQUE[j].peso;
(6)     elimina el árbol j de BOSQUE
end;

```

Fig. 3.26. Esbozo de la construcción de árboles de Huffman.

Para aplicar la línea (4) de la figura 3.26, la cual incrementa el número de celdas empleadas del arreglo *ARBOL*, y las líneas (5) y (6), que disminuyen el número de celdas utilizadas de *BOSQUE*, se introducen los cursores *últ_arbol* y *últ_nodo*, que apuntan hacia *BOSQUE* y *ARBOL*, respectivamente. Se supone que las celdas 1 a *últ_arbol* de *BOSQUE* y 1 a *últ_nodo* de *ARBOL* están ocupadas †. Se supone también que los arreglos de la figura 3.25 tienen declaradas sus longitudes, pero en lo que sigue se omitirán las comparaciones entre esos límites y los valores de los cursores.

```

procedure más_ligeros ( var menor, segundo: integer );
{ asigna menor y segundo a los índices en BOSQUE de los dos árboles
  de menor peso; se supone que últ_arbol ≥ 2. }
var
  i: integer;
begin { asigna valor inicial a menor y a segundo teniendo en cuenta
        los dos primeros árboles }
  if BOSQUE[1].peso <= BOSQUE[2].peso then
    begin menor := 1; segundo := 2 end
  else
    begin menor := 2; segundo := 1 end;
{ Ahora recorre i desde 3 a últ_arbol. En cada iteración, menor es
  el árbol de menor peso entre los i primeros de BOSQUE; y
  segundo, el siguiente en peso entre los mismos }
  for i := 3 to últ_arbol do
    if BOSQUE[i].peso < BOSQUE[menor].peso then
      begin segundo := menor; menor := i end
    else if BOSQUE[i].peso < BOSQUE[segundo].peso then
      segundo := i
  end; { más_ligeros }

```

† En la fase de lectura, aquí omitida, también se requiere un cursor para el arreglo *ALFABETO*, mientras se llena con los símbolos y sus probabilidades.

```

function crea ( árbol_izq, árbol_der; integer ) : integer;
  { devuelve un nuevo nodo cuyos hijos son BOSQUE[árbol_izq].raíz
    y BOSQUE[árbol_der].raíz }
begin
  últ_nodo := últ_nodo + 1;
  { la celda para el nuevo nodo es ARBOL[últ_nodo] }
  ARBOL[últ_nodo].hijo_izq := BOSQUE[árbol_izq].raíz;
  ARBOL[últ_nodo].hijo_der := BOSQUE[árbol_der].raíz;
  { ahora asigna apuntadores padre al nuevo nodo y a sus hijos }
  ARBOL[últ_nodo].padre := 0;
  ARBOL[BOSQUE[árbol_izq].raíz].padre := últ_nodo;
  ARBOL[BOSQUE[árbol_der].raíz].padre := últ_nodo;
  return (últ_nodo)
end; { crea }

```

Fig. 3.27. Dos procedimientos.

La figura 3.27 muestra dos procedimientos útiles; el primero es una realización de las líneas (2) y (3) de la figura 3.26, en las que se seleccionan los índices de los dos árboles de menor peso. El segundo es la orden *crea(n_1, n_2)*, que crea un nuevo nodo y hace que n_1 y n_2 se conviertan en sus hijos izquierdo y derecho.

Ahora es posible describir con más detalles los pasos de la figura 3.26. En la figura 3.28 se puede ver un procedimiento *Huffman*, que no tiene parámetros sino que trabaja con las estructuras globales de la figura 3.25.

```

procedure Huffman;
var
  i, j: integer; { los dos árboles de menor peso en BOSQUE }
  nueva_raíz: integer;
begin
  while últ_arbol > 1 do begin
    más_ligeros(i, j);
    nueva_raíz := crea(i, j);
    { Ahora reemplaza el árbol i por el árbol cuya raíz es nueva_raíz }
    BOSQUE[i].peso := BOSQUE[i].peso + BOSQUE[j].peso;
    BOSQUE[i].raíz := nueva_raíz;
    { luego reemplaza el árbol j, que ya no se necesita,
      por últ_arbol, y acorta BOSQUE en uno }
    BOSQUE[j] := BOSQUE[últ_arbol];
    últ_arbol := últ_arbol - 1
  end
end; { Huffman }

```

Fig. 3.28. Algoritmo de Huffman.

La figura 3.29 muestra la estructura de datos de la figura 3.25 después de que *últ_árbol* ha quedado reducido a 3; es decir, cuando el bosque tiene el aspecto de la figura 3.24(c).

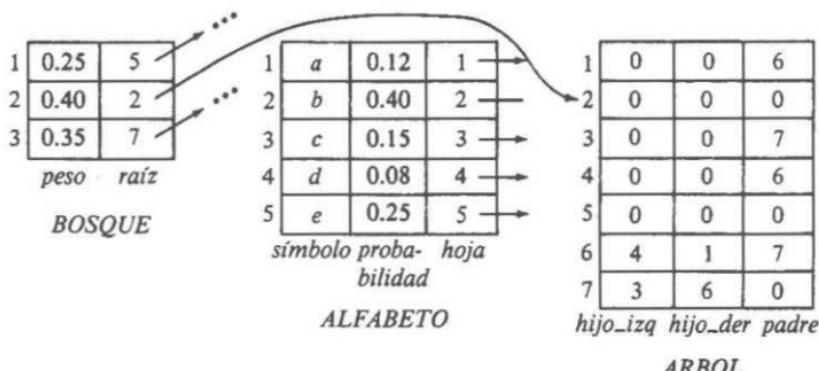


Fig. 3.29. Estructura de datos de un árbol después de dos iteraciones.

Después de terminada la ejecución del algoritmo, el código de cada símbolo se puede determinar como sigue. Encuéntrese el símbolo en el campo *símbolo* del arreglo *ALFABETO*. Sigase el campo *hoja* del mismo registro para obtener un registro del arreglo *ARBOL*; este registro corresponde a la hoja asociada al símbolo en cuestión. Sígase repetidamente el apuntador *padre* desde el registro «actual», por ejemplo, el del nodo *n*, al registro en el arreglo *ARBOL* de su padre *p*. Hágase esto sin olvidar el nodo *n*, de manera que sea posible examinar los apuntadores *hijo_izq* e *hijo_der* de *p* y determinar cuál de ellos apunta a *n*. En el primer caso, imprímase 0; en el segundo, 1. La secuencia de bits impresa de esta forma será el código del símbolo con el orden de los bits invertido. Si se desea imprimir los bits en el orden correcto, se pueden meter en una pila conforme se sube por el árbol, tras lo cual se sacan en forma reiterada bits de la pila, imprimiéndolos al mismo tiempo.

Realización de árboles binarios basada en apuntadores

En vez de utilizar cursores para apuntar a los hijos izquierdos y derechos (y a los padres, si se desea), es posible usar apuntadores genuinos de Pascal. Por ejemplo, se puede declarar

```

type
  nodo = record
    hijo_izq: ^nodo;
    hijo_der: ^nodo;
    padre: ^nodo
  end

```

Por otro lado, si se usara este tipo de nodos para un árbol binario, la función *crea* de la figura 3.27 se escribiría como se muestra en la figura 3.30.

```
function crea (árbol_izq, árbol_der: ↑ nodo) : ↑ nodo;
var
    raíz: ↑ nodo;
begin
    new(raíz);
    raíz↑.hijo_izq := árbol_izq;
    raíz↑.hijo_der := árbol_der;
    raíz↑.padre := 0;
    árbol_izq↑.padre := raíz;
    árbol_der↑.padre := raíz;
    return (raíz)
end; { crea }
```

Fig. 3.30. Realización de árboles binarios basada en apuntadores.

Ejercicios

3.1 Respóndase a las siguientes preguntas acerca del árbol de la figura 3.31.

- ¿Qué nodos son hojas?
- ¿Qué nodo es la raíz?
- ¿Cuál es el parente del nodo C?
- ¿Qué nodos son los hijos de C?
- ¿Qué nodos son los antecesores de C?
- ¿Qué nodos son los descendientes de E?
- ¿Cuáles son los hermanos derechos de D y E?
- ¿Qué nodos están a la izquierda y a la derecha de G?
- ¿Cuál es la profundidad del nodo C?
- ¿Cuál es la altura del nodo C?

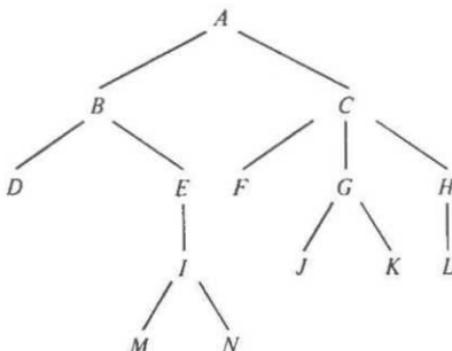


Fig. 3.31. Un árbol.

- 3.2 ¿Cuántos caminos distintos de longitud 3 hay en el árbol representado en la figura 3.31?
- 3.3 Escríbanse programas para calcular la altura de un árbol, usando cada una de las tres representaciones de árboles de la sección 3.3.
- 3.4 Lístense los nodos de la figura 3.31 en
- orden previo,
 - orden posterior, y
 - orden simétrico.
- 3.5 Muéstrese que si m y n son dos nodos distintos del mismo árbol, exactamente una de las siguientes afirmaciones es verdadera:
- m está a la izquierda de n
 - m está a la derecha de n
 - m es un antecesor propio de n
 - m es un descendiente propio de n .
- 3.6 Márquese la intersección de la fila i con la columna j si las situaciones representadas por esa fila y esa columna pueden ocurrir simultáneamente.

	$ord_pre(n)$ $< ord_pre(m)$	$ord_sim(n)$ $< ord_sim(m)$	$ord_post(n)$ $< ord_post(m)$
n está a la izquierda de m			
n está a la derecha de m			
n es un antecesor propio de m			
n es un descendiente propio de m			

Por ejemplo, póngase una marca en la intersección de la fila 3 y la columna 2 si se cree que n puede ser a la vez un antecesor propio de m y preceder a m en orden simétrico.

- 3.7 Supóngase que se tienen los arreglos $ORD_PRE[n]$, $ORD_SIM[n]$ y $ORD_POST[n]$ que dan las posiciones en los órdenes previo, simétrico y posterior, respectivamente, de cada nodo n de un árbol. Describáse un algoritmo que diga si un nodo i es un antecesor o no de un nodo j , para cualquier par de nodos i, j . Explíquese cómo trabaja el algoritmo.
- *3.8 Es posible probar si un nodo m es un descendiente propio o no de un nodo n , determinando si m precede a n en orden X, pero sucede a n en orden Y, donde X e Y se eligen {pre, post, sim}. Determínense todos los pares X, Y para los cuales se cumple esta afirmación.

- 3.9 Escribase un programa para recorrer un árbol binario en
- orden previo,
 - orden posterior,
 - orden simétrico.
- 3.10 El listado en *orden de nivel* de los nodos de un árbol lista primero la raíz, luego todos los nodos de profundidad 1, después todos los de profundidad 2, y así sucesivamente. Los nodos que estén a la misma profundidad se listan en orden de izquierda a derecha. Escribase un programa para listar los nodos de un árbol en orden de nivel.
- 3.11 Conviértase la expresión $((a + b) * c * (d + e) + f) * (g + h)$ en
- expresión prefija
 - expresión postfixia.
- 3.12 Dibújense representaciones de árbol para las expresiones
- $* a + b * c + d e$
 - $* a + * b + c d e$.
- 3.13 Sea A un árbol en el que cada nodo que no es hoja tiene dos hijos. Escribase un programa para convertir
- un listado en orden previo de A en un listado en orden posterior;
 - un listado en orden posterior de A en un listado en orden previo;
 - un listado en orden previo de A en un listado en orden simétrico.
- 3.14 Escribase un programa para evaluar expresiones aritméticas en
- orden previo
 - orden posterior.
- 3.15 Como modelo matemático, es posible definir un árbol binario como un TDA con estructura de árbol binario y con operaciones como HIJO-IZQ(n), HIJO-DER(n), PADRE(n) y NULO(n). Las tres primeras operaciones devuelven el hijo izquierdo, el hijo derecho y el padre del nodo n (Λ , si no existe), y la última devuelve verdadero si, y sólo si, n es Λ . Implántense estos procedimientos con la representación de árboles binarios de la figura 3.21.
- 3.16 Obténganse las siete operaciones con árboles de la sección 3.2, usando las siguientes realizaciones de árboles:
- apuntadores al padre,
 - listas de hijos,
 - apuntadores al hijo más a la izquierda, hermano derecho.
- 3.17 El *grado* de un nodo es el número de hijos que tiene. Muéstrese que en cualquier árbol binario el número de hojas es uno más que el número de nodos de grado dos.

- 3.18** Muéstrese que el máximo número de nodos de un árbol binario de altura h es $2^{h+1} - 1$. Un árbol binario de altura h con ese máximo número de nodos se denomina árbol binario *lleno*.
- *3.19** Supóngase que los árboles están realizados por medio de apuntadores al hijo más a la izquierda, al hermano derecho y al padre. Elabórense algoritmos no recursivos de recorrido en los órdenes previo, posterior y simétrico, que no utilicen «estados» ni una pila, como se hace en la figura 3.9.
- 3.20** Supóngase que los caracteres a, b, c, d, e , y f tienen probabilidades 0.07, 0.09, 0.12, 0.22, 0.23 y 0.27, respectivamente. Encuéntrese un código de Huffman óptimo y dibújese el árbol de Huffman correspondiente. ¿Cuál es la longitud media del código?
- *3.21** Supóngase que A es un árbol de Huffman y que la hoja del símbolo a tiene una profundidad mayor que la del símbolo b . Demuéstrese que la probabilidad del símbolo b no es menor que la del símbolo a .
- *3.22** Demuéstrese que el algoritmo de Huffman funciona, es decir, que produce un código óptimo para las probabilidades dadas. *Sugerencia.* Utilícese el ejercicio 3.21.

Notas bibliográficas

Berge [1958] y Harary [1969] analizan las propiedades matemáticas de los árboles. Knuth [1973] y Nievergelt [1974] contienen información adicional sobre los árboles de búsqueda binarios. Muchos de los trabajos sobre grafos y sus aplicaciones, a las que se hace referencia en el capítulo 6, también contienen material sobre árboles.

El algoritmo dado en la sección 3.4 para encontrar un árbol con mínima longitud ponderada de camino es de Huffman [1952]. Parker [1980] contiene algunas exploraciones más recientes en ese algoritmo.

4

Operaciones básicas con conjuntos

Los conjuntos constituyen la estructura básica que fundamenta las matemáticas. En el diseño de algoritmos, los conjuntos se utilizan como la base de una gran cantidad de tipos de datos abstractos y se han desarrollado muchas técnicas para la implantación de tipos de datos abstractos basadas en conjuntos. En este capítulo se hace una revisión de las operaciones básicas con conjuntos y se presentan algunas de las formas más sencillas de realización de conjuntos; se estudian los «diccionarios» y las «colecciones de prioridad», dos tipos de datos abstractos basados en el modelo de conjunto. Las realizaciones de estos tipos de datos abstractos se cubren en este capítulo y el siguiente.

4.1 Introducción a los conjuntos

Un conjunto es una colección de *miembros* (o *elementos*); cada miembro de un conjunto puede ser un conjunto, o un elemento primitivo que recibe el nombre de *átomo*. Todos los miembros de un conjunto son distintos, lo cual significa que ningún conjunto puede contener dos copias de un mismo elemento.

Por lo general, cuando los átomos se usan como herramientas para el diseño de algoritmos y estructuras de datos, son enteros, caracteres o cadenas, y todos los elementos de cualquier conjunto suelen ser del mismo tipo. A menudo se hará la suposición de que los átomos están ordenados linealmente por una relación que se designa mediante el símbolo «<», y se lee «menor que» o «precede a». Un *orden lineal* < sobre un conjunto *S* satisface dos propiedades:

1. Para cualesquiera *a* y *b* en *S*, sólo una de las afirmaciones *a* < *b*, *a* = *b* o *b* < *a* es verdadera.
2. Para todo *a*, *b* y *c* en *S*, si *a* < *b* y *b* < *c*, entonces *a* < *c* (transitividad).

Los enteros, reales, caracteres y cadenas tienen un orden lineal natural que en Pascal se designa con el símbolo <. Se puede definir un orden lineal sobre los objetos que consisten en conjuntos de objetos ordenados. Como ejercicio, determiníese una forma de desarrollar un orden de este tipo. Por ejemplo, una pregunta que se debe contestar al desarrollar un orden lineal para un conjunto de enteros es si el conjunto que contiene los enteros 1 y 4 debe considerarse menor o mayor que el conjunto que contiene los enteros 2 y 3.

Notación de conjuntos

Un conjunto de átomos por lo general se representa encerrando sus miembros entre llaves; así, $\{1, 4\}$ denota el conjunto cuyos únicos miembros son 1 y 4. Debe recordarse que un conjunto no es una lista, a pesar de que se representen los conjuntos como si fueran listas; en el caso de los conjuntos, el orden en que se listan los elementos no es importante, y se pudo haber escrito $\{4, 1\}$ en lugar de $\{1, 4\}$. Se debe observar también que cada elemento aparece en un conjunto sólo una vez, de modo que $\{1, 4, 1\}$ no es un conjunto †.

Algunas veces se representan los conjuntos mediante una expresión de la forma

$$\{x \mid \text{proposición sobre } x\}$$

donde la proposición sobre x es un predicado que dice con exactitud qué se necesita para que un objeto arbitrario x pertenezca al conjunto. Por ejemplo, $\{x \mid x \text{ es un entero positivo y } x \leq 1000\}$ es otra manera de representar el conjunto $\{1, 2, \dots, 1000\}$, y $\{x \mid \text{para algún entero } y, x = y^2\}$ designa el conjunto de los cuadrados perfectos. Obsérvese que el conjunto de los cuadrados perfectos es infinito y no se puede representar listando sus miembros.

La relación fundamental en teoría de conjuntos es la de pertenencia, que se indica por el símbolo \in . Esto es, $x \in A$ significa que x pertenece a A o que el elemento x es un miembro del conjunto A , y puede ser un átomo u otro conjunto, pero A no puede ser un átomo. Se utiliza $x \notin A$ para señalar que « x no es un miembro de A ». Existe un conjunto especial que no tiene miembros, y se simboliza con \emptyset , que se denomina *conjunto nulo* o *conjunto vacío*. Obsérvese que \emptyset es un conjunto y no un átomo, a pesar de no tener miembros. La diferencia radica en que $x \in \emptyset$ es falso para toda x , mientras que si y es un átomo, entonces $x \in y$ no tiene sentido, porque más que una afirmación falsa, es un error de sintaxis.

Se dice que un conjunto A está *incluido* (*o contenido*) en un conjunto B , y se escribe $A \subseteq B$ o $B \supseteq A$, si todo miembro de A también es miembro de B . También se dice en este caso que A es un *subconjunto* de B o que B es un *supraconjunto* de A . Por ejemplo, $\{1, 2\} \subseteq \{1, 2, 3\}$, pero $\{1, 2, 3\}$ no es un subconjunto de $\{1, 2\}$ porque 3 pertenece al primero de estos conjuntos, pero no al segundo. Todo conjunto está incluido en sí mismo, y el conjunto vacío está incluido en todo conjunto. Dos conjuntos son iguales si cada uno de ellos está incluido en el otro, esto es, si sus miembros son los mismos. Un conjunto A es un *subconjunto propio* o un *supraconjunto propio* de otro B , si $A \neq B$ y $A \subset B$ o $A \supset B$, respectivamente.

Las operaciones elementales con conjuntos son unión, intersección y diferencia. Si A y B son conjuntos, entonces $A \cup B$, la *unión* de A y B , es el conjunto de los elementos que son miembros de A , de B o de ambos. La *intersección* de A y B , que se escribe $A \cap B$, es el conjunto de los elementos que pertenecen tanto a A como a B , y la *diferencia*, $A - B$, es el conjunto de los elementos de A que no pertenecen a B .

† Algunas veces el término *conjunto múltiple* se aplica a un «conjunto con repeticiones», es decir, el conjunto múltiple $\{1, 4, 1\}$ tiene (dos veces) 1 y (una vez) 4 como miembros. Un conjunto múltiple tampoco es una lista, por lo que el anterior se pudo haber escrito 4, 1, 1 ó 1, 1, 4.

Por ejemplo, si $A = \{a, b, c\}$ y $B = \{b, d\}$, entonces $A \cup B = \{a, b, c, d\}$, $A \cap B = \{b\}$ y $A - B = \{a, c\}$.

Tipos de datos abstractos basados en conjuntos

Se considerarán ahora los TDA que incorporan diversas operaciones con conjuntos. Algunas colecciones de estas operaciones reciben nombres especiales y tienen realizaciones muy eficientes. Algunas de las operaciones con conjuntos más comunes son las siguientes.

- 1.-3. Los procedimientos UNION(A, B, C), INTERSECCION(A, B, C) y DIFERENCIA(A, B, C) toman los argumentos A y B cuyos valores son conjuntos y asignan el resultado, $A \cup B$, $A \cap B$ o $A - B$, respectivamente, a la variable C de conjuntos.
4. Algunas veces se emplea una operación llamada *combinación* o *unión de conjuntos disjuntos*, que no difiere de la unión, pero supone que sus operandos son disjuntos (no tienen miembros en común). El procedimiento COMBINA(A, B, C) asigna a la variable C de conjuntos el valor $A \cup B$, pero no está definido cuando $A \cap B \neq \emptyset$, es decir, si los conjuntos A y B no son disjuntos.
5. La función MIEMBRO(x, A) toma el conjunto A y el objeto x , cuyo tipo es el de los elementos de A , y devuelve un valor booleano: verdadero si $x \in A$ y falso si $x \notin A$.
6. El procedimiento ANULA(A), hace que el conjunto nulo sea el valor de la variable conjunto A .
7. El procedimiento INSERTA(x, A), donde A es una variable cuyos valores son conjuntos y x es un elemento del tipo de los miembros de A , hace de x un miembro de A . Esto es, el nuevo valor de A es $A \cup \{x\}$. Obsérvese que si x ya es miembro de A , entonces INSERTA(x, A) no modifica el conjunto A .
8. SUPRIME(x, A) elimina x de A , es decir, A se reemplaza por $A - \{x\}$. Si x no está originalmente en A , SUPRIME(x, A) deja sin cambios el conjunto A .
9. ASIGNA(A, B) hace que el valor de la variable A de conjuntos sea igual al valor de la variable B , también de conjuntos.
10. La función MIN(A) devuelve el elemento menor del conjunto A . Esta operación sólo es aplicable cuando los miembros del conjunto parámetro están ordenados linealmente. Por ejemplo, $\text{MIN}(\{2, 3, 1\}) = 1$ y $\text{MIN}(\{'a', 'b', 'c'\}) = 'a'$. También se utiliza la función MAX, cuyo significado es obvio.
11. IGUAL(A, B) es una función cuyo valor es verdadero si, y sólo si, los conjuntos A y B contienen los mismos elementos.
12. La función ENCUENTRA(x) opera en un ambiente donde hay una colección de conjuntos disjuntos. ENCUENTRA(x) devuelve el nombre del (único) conjunto del cual x es miembro.

4.2 Un TDA con UNION, INTERSECCION y DIFERENCIA

Para empezar, se definirá un TDA para el modelo matemático «conjunto» con las tres operaciones básicas de la teoría de conjuntos, unión, intersección y diferencia. Primero se dará un ejemplo en el que un TDA es útil y después se analizarán varias realizaciones sencillas del mismo.

Ejemplo 4.1. Se desea escribir un programa que efectúe una forma sencilla de «análisis de flujo de datos» sobre diagramas de flujo que representen procedimientos. El programa usará variables de un tipo de datos abstracto CONJUNTO, cuyas operaciones son UNION, INTERSECCION, DIFERENCIA, IGUAL, ASIGNA Y ANULA, según se definieron en la sección anterior.

En la figura 4.1 se observa un diagrama de flujo cuyas casillas tienen los nombres B_1, \dots, B_8 y contienen *definiciones de datos* (proposiciones de lectura y asignación) numeradas del 1 al 9. Este diagrama de flujo es una aplicación del algoritmo de Euclides para calcular el máximo común divisor de sus entradas p y q , pero los detalles del algoritmo no tienen importancia para el ejemplo.

En general, un *análisis de flujo de datos* se refiere a la parte de un compilador que examina la representación de un programa fuente en forma de un diagrama de flujo, como la figura 4.1, y reúne información acerca de lo que puede ser cierto conforme el control llega a cada casilla del diagrama. A menudo, las casillas reciben el nombre de *bloques* o *bloques básicos* y representan grupos de proposiciones a través de los cuales el flujo de control procede en forma secuencial. La información que se colecta en el análisis de flujo de datos sirve para mejorar el código generado por el compilador. Por ejemplo, si el análisis permitió determinar que cada vez que el control llegó al bloque B , la variable x tuvo el valor 27, entonces se podría sustituir x por 27 en el bloque B , a menos que se hubiera asignado un nuevo valor a x dentro de ese bloque. Si el acceso a las constantes fuera más rápido que el acceso a las variables, este cambio haría que el código generado por el compilador se ejecutara a mayor velocidad.

En el ejemplo se desea determinar el lugar donde una variable pudo haber recibido, por última vez, un valor nuevo. Esto es, se desea calcular para cada bloque B_i el conjunto $DEF_ENT[i]$ de definiciones de datos d , de modo que exista un camino de B_1 a B_i en el cual aparezca d , pero que no esté seguida de otra definición de la misma variable que define d . El conjunto $DEF_ENT[i]$ recibe el nombre de *definiciones de alcance* de B_i .

Para ver cómo podría ser útil esa información, considérese la figura 4.1. El primer bloque B_1 es un bloque «ficticio» con tres definiciones de datos que hacen que las variables t , p y q tomen valores «indefinidos». Si, por ejemplo, se descubre que $DEF_ENT[7]$ incluye la definición 3, la cual da a q un valor indefinido, entonces el programa podría tener un error, puesto que aparentemente podría imprimir q sin antes haberle asignado un valor válido. Por fortuna, se descubrirá que es imposible alcanzar el bloque B_7 sin haber hecho una asignación a q , es decir, 3 no pertenece a $DEF_ENT[7]$.

Varias reglas ayudan a calcular los $DEF_ENT[i]$. Primero, se hace un cálculo pre-

vio, para cada bloque i , de los conjuntos $GEN[i]$ y $ELIM[i]$. $GEN[i]$ es el conjunto de las definiciones de datos del bloque i , con la excepción de que si B_i contiene dos o más definiciones de una variable x , entonces sólo la última de ellas pertenece a $GEN[i]$. Así, $GEN[i]$ es el conjunto de las definiciones de B_i que son «generadas» por B_i ; alcanzan el final de B_i sin que sus variables sean redefinidas.

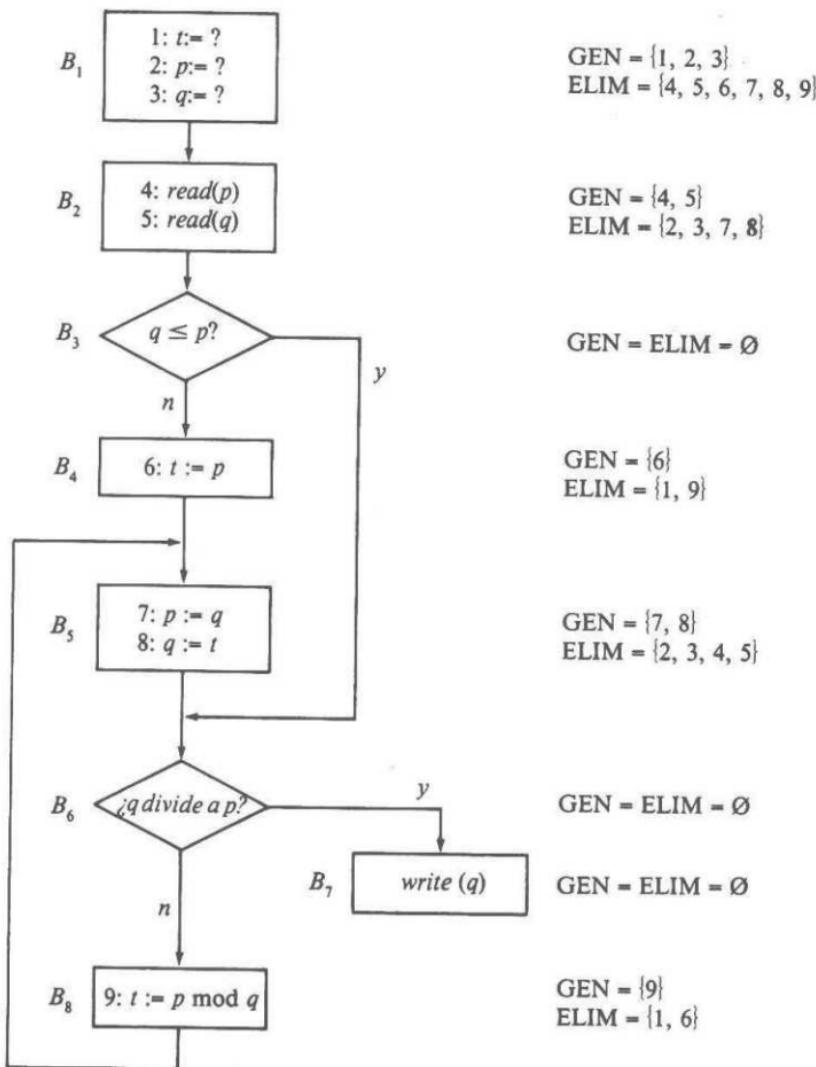


Fig. 4.1. Diagrama de flujo para el algoritmo de Euclides.

El conjunto $ELIM[i]$ es el conjunto de las definiciones d que no están en B_i y tales que B_i contiene una definición de la misma variable que d . Por ejemplo, en la figura 4.1, $GEN[4] = \{6\}$, puesto que la definición 6 (de la variable t) está en B_4 y no hay ninguna definición posterior de t en B_4 . $ELIM[4] = \{1, 9\}$, puesto que 1 y 9 son las definiciones de la variable t que no se encuentran en B_4 .

Además de los $DEF_ENT[i]$, se calcula el conjunto $DEF_SAL[i]$ para cada uno de los bloques B_i . De la misma manera que $DEF_ENT[i]$ es el conjunto de definiciones que alcanzan el principio de B_i , $DEF_SAL[i]$ es el conjunto de definiciones que alcanzan el final de B_i . Existe una fórmula muy sencilla que relaciona DEF_ENT y DEF_SAL :

$$DEF_SAL[i] = (DEF_ENT[i] - ELIM[i]) \cup GEN[i] \quad (4.1)$$

Esto es, la definición d alcanza el final de B_i si, y sólo si, d alcanza el principio de B_i y no es eliminada por B_i o es generada en B_i . La segunda regla que relaciona DEF_ENT y DEF_SAL es que $DEF_ENT[i]$ es la unión, sobre todos los predecesores p de B_i , de $DEF_SAL[p]$, es decir:

$$DEF_ENT[i] = \bigcup_{B_p \text{ precede a } B_i} DEF_SAL[p] \quad (4.2)$$

La regla 4.2 establece que una definición de datos entra en B_i si, y sólo si, alcanza el final de uno de los predecesores de B_i . Como caso especial, si B_i no tiene predecesores, como ocurre con B_1 en la figura 4.1, entonces $DEF_ENT[i] = \emptyset$.

Como ya se han presentado varios conceptos nuevos en este ejemplo, no se profundizará al escribir un algoritmo general para calcular las definiciones de entrada de un diagrama de flujo arbitrario. En vez de ello, se escribirá una parte de un programa que supone que $GEN[i]$ y $ELIM[i]$ se conocen para $i = 1, \dots, 8$, que calcula $DEF_ENT[i]$ y $DEF_SAL[i]$ para $i = 1, \dots, 8$, en el caso particular del diagrama de flujo de la figura 4.1. Este fragmento de programa supone la existencia de un TDA CONJUNTO con las operaciones UNION, INTERSECCION, DIFERENCIA, IGUAL, ASIGNA y ANULA. Más adelante se darán varias implantaciones alternativas de este TDA.

El procedimiento *propaga(GEN, ELIM, DEF_ENT, DEF_SAL)* aplica la regla 4.1. al cálculo de DEF_SAL para un bloque, dado DEF_ENT . En el caso de un programa sin ciclos, el cálculo de DEF_SAL sería directo. La presencia de un ciclo en este caso exige un procedimiento iterativo. La aproximación a $DEF_ENT[i]$ da comienzo con $DEF_ENT[i] = \emptyset$ y $DEF_SAL[i] = GEN[i]$ para toda i , y después aplica repetidas veces (4.1) y (4.2) hasta que ya no ocurran más cambios en los DEF_ENT y DEF_SAL . Como puede mostrarse que cada nuevo valor asignado a los $DEF_ENT[i]$ y $DEF_SAL[i]$ es un supraconjunto (no necesariamente propio) de su valor anterior, y dado que en cualquier programa hay solamente un número finito de definiciones de datos, el proceso debe converger finalmente a una solución de (4.1) y (4.2).

En la figura 4.3 se muestran los valores sucesivos de $DEF_ENT[i]$ después de cada iteración del ciclo `repeat` de la figura 4.2. Obsérvese que ninguna de las asignaciones ficticias 1, 2 y 3 alcanza un bloque donde se emplea su variable, por tanto, en el programa de la figura 4.1 no hay usos de variables indefinidas. Obsérvese también que al diferir la aplicación de (4.2) para B_i hasta antes de aplicar (4.1) para B_i , haría que en general el proceso de la figura 4.2 convergiera en menos iteraciones. \square

4.3 Realización de conjuntos mediante vectores de bits

Obtener la mejor realización para un TDA CONJUNTO depende de las operaciones que se vayan a efectuar y del tamaño de los conjuntos. Cuando todos los conjuntos del dominio en cuestión son subconjuntos de un pequeño «conjunto universal» cuyos elementos son los enteros 1, 2, .., N , para algún N fijo, se puede usar una realización basada en un *vector de bits* (arreglo booleano). Un conjunto se representa mediante un vector de bits en el que el i -ésimo bit es verdadero si i es un elemento del conjunto. La principal ventaja de esta representación radica en que las operaciones MIEMBRO, INSERTA y SUPRIME se pueden realizar en un tiempo constante mediante una referencia directa al bit apropiado. UNION, INTERSECCION y DIFERENCIA se pueden realizar en un tiempo proporcional al tamaño del conjunto universal.

```

var
  GEN, ELIM, DEF_ENT, DEF_SAL: array[1..8] of CONJUNTO;
  { se supone que GEN y ELIM se calculan por fuera }
  i: integer;
  hay_cambios: boolean;

procedure propaga ( G,K,I: CONJUNTO; var O: CONJUNTO );
  { aplica (4.1) y asigna verdadero a hay_cambios si se detecta algún cambio
    en DEF_SAL }
var
  TEMP: CONJUNTO;
begin
  DIFERENCIA(I, K, TEMP);
  UNION(TEMP, G, TEMP);
  if not IGUAL(TEMP, O) do begin
    ASIGNA(O, TEMP);
    hay_cambios := true
  end
end; { propaga }

begin
  for i := 1 to 8 do
    ASIGNA(DEF_SAL[i], GEN[i]);

```

```

repeat
    hay_cambios := false;
    { las ocho proposiciones siguientes aplican (4.2) al diagrama de la figura
        4.1 solamente }
    ANULA(DEF_ENT[1]);
    ASIGNA(DEF_ENT[2], DEF_SAL[1]);
    ASIGNA(DEF_ENT[3], DEF_SAL[2]);
    ASIGNA(DEF_ENT[4], DEF_SAL[3]);
    UNION(DEF_SAL[4], DEF_SAL[8], DEF_ENT[5]);
    UNION(DEF_SAL[3], DEF_SAL[5], DEF_ENT[6]);
    ASIGNA(DEF_ENT[7], DEF_SAL[6]);
    ASIGNA(DEF_ENT[8], DEF_SAL[6]);
    for i := 1 to 8 do
        propaga(GEN[i], ELIM[i], DEF_ENT[i], DEF_SAL[i]);
    until
        not hay_cambios
end.

```

Fig. 4.2. Programa para calcular definiciones de alcance.

<i>i</i>	paso 1	paso 2	paso 3	paso 4
1	\emptyset	\emptyset	\emptyset	\emptyset
2	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}
3	{4,5}	{1,4,5}	{1,4,5}	{1,4,5}
4	\emptyset	{4,5}	{1,4,5}	{1,4,5}
5	{6,9}	{6,9}	{4,5,6,7,8,9}	{4,5,6,7,8,9}
6	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
7	\emptyset	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}
8	\emptyset	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}

Fig. 4.3. *DEF_ENT [i]* después de cada iteración.

Si el conjunto universal es suficientemente pequeño para que un vector de bits quepa en una palabra de computador, entonces UNION, INTERSECCION y DIFERENCIA se pueden realizar mediante una sola operación lógica del lenguaje de la máquina en cuestión. Algunos conjuntos pequeños se pueden representar directamente en Pascal por medio de la construcción set. El tamaño máximo de tales conjuntos depende del compilador particular que se esté usando y, por desgracia, a menudo es demasiado pequeño para problemas típicos con conjuntos. Sin embargo, al escribir los propios programas no es necesario restringirse a un límite para el tamaño de los conjuntos, siempre que sea posible tratar cualquier conjunto como un subconjunto de algún conjunto universal $\{1, \dots, N\}$. Se pretende que si *A* es un conjunto representado por un arreglo booleano, entonces *A*[*i*] será verdadero si, y sólo si, el elemento

i es miembro de *A*. Así, se puede definir un TDA CONJUNTO mediante la declaración en Pascal

```
const
  N = {cualquier valor adecuado};
type
  CONJUNTO = packed array[1..N] of boolean;
```

Se puede implantar entonces el procedimiento UNION tal como se muestra en la figura 4.4. Para implantar INTERSECCION y DIFERENCIA basta reemplazar or en la figura 4.4, por and y and not, respectivamente. Es posible poner en práctica las otras operaciones mencionadas en la sección 4.1 (exceptuando COMBINA y ENCUENTRA, que tienen poco sentido en este contexto) a manera de ejercicios sencillos.

Se puede utilizar la implantación de conjuntos mediante vectores de bits, cuando el conjunto universal es un conjunto finito diferente de un conjunto de enteros consecutivos. De ordinario, se necesitaría una manera de «traducir» entre los miembros del conjunto universal y los enteros 1, ..., *N*. Así, en el ejemplo 4.1, se supuso que las definiciones de datos tenían asignados números del 1 al 9. En general, las traducciones en ambas direcciones se pueden efectuar mediante el TDA CORRESPONDENCIA descrito en el capítulo 2. Sin embargo, la traducción en sentido inverso de enteros a elementos del conjunto universal puede lograrse mejor con un arreglo *A* en el cual *A*[*i*] sea el elemento correspondiente al entero *i*.

```
procedure UNION ( A, B: CONJUNTO; var C: CONJUNTO );
  var
    i: integer;
  begin
    for i := 1 to N do
      C[i] := A[i] or B[i]
  end
```

Fig. 4.4. Realización de UNION.

4.4 Realización de conjuntos mediante listas enlazadas

Debe ser evidente que también los conjuntos se pueden representar por medio de listas enlazadas, cuyos elementos son los miembros del conjunto. A diferencia de la representación mediante vectores de bits, la representación mediante listas utiliza un espacio proporcional al tamaño del conjunto representado, y no al del conjunto universal. Más aún, la representación mediante listas es más general, puesto que puede manejar conjuntos que no necesitan ser subconjuntos de algún conjunto universal finito.

Cuando hay operaciones como INTERSECCION con conjuntos representados por listas enlazadas, se presentan varias opciones. Si el conjunto universal tiene un

orden lineal, puede representarse un conjunto por medio de una lista clasificada. Esto es, se supone que todos los miembros de un conjunto son comparables por medio de una relación « $<$ » y los miembros de un conjunto aparecen en una lista en el orden e_1, e_2, \dots, e_n , donde $e_1 < e_2 < \dots < e_n$. La ventaja de una lista clasificada radica en que no es necesario revisarla toda para determinar si un elemento se encuentra en la lista.

Un elemento está en la intersección de las listas L_1 y L_2 si, y sólo si, aparece en ambas listas. Para determinar la intersección con listas no clasificadas, es necesario parear cada elemento de L_1 con cada elemento de L_2 en un proceso de $O(n^2)$ pasos con listas de longitud n . La razón por la cual la clasificación de listas facilita la intersección y algunas otras operaciones, consiste en que si se desea parear un elemento e de una lista L_1 con los elementos de otra lista L_2 , sólo es necesario observar los elementos de L_2 hasta que se encuentre e o un elemento mayor que e ; en el primer caso, se habrá encontrado el par, mientras que en el segundo, se sabe que no existe. Es más, si d es el elemento de L_1 que precede inmediatamente a e , y ya se ha encontrado en L_2 el primer elemento, por ejemplo f , tal que $d \leq f$, para buscar en L_2 una aparición de e , se puede empezar con f . La conclusión de este razonamiento es que se pueden encontrar pares para todos los elementos de L_1 , si existen, buscando en L_1 y L_2 una sola vez, siempre que se hagan avanzar los marcadores de posición de las dos listas en el orden apropiado, es decir, avanzando siempre el que apunta al elemento más pequeño. La rutina para realizar INTERSECCION se muestra en la figura 4.5. Ahí, los conjuntos se representan mediante listas enlazadas de «celdas» cuyo tipo se define como

```
type
  tipo_celda = record
    elemento: tipo_elemento;
    sig: ^ tipo_celda
  end
```

En la figura 4.5, se supone que tipo_elemento es un tipo, por ejemplo un entero, que se puede comparar mediante $<$. En caso contrario, es necesario escribir una función que, dados dos elementos, determine cuál de ellos precede al otro.

Las listas enlazadas de la figura 4.5 están encabezadas por celdas vacías que sirven como puntos de entrada a las listas. Como ejercicio, escríbese este programa en una forma abstracta más general, usando primitivas de listas. Sin embargo, el programa de la figura 4.5 puede ser más eficiente que el programa más abstracto. Por ejemplo, en la figura 4.5 se usan apuntadores a celdas particulares, en vez de variables «de posición» que apunten a celdas anteriores. Se puede hacer esto porque sólo se agrega al final de la lista C , y las listas A y B sólo se revisan sin hacer en ellas inserciones ni supresiones.

Las operaciones UNION y DIFERENCIA se pueden escribir de modo que se asemejen en forma sorprendente al procedimiento INTERSECCION de la figura 4.5. Para UNION, se deben agregar todos los elementos de la lista A o de la B , a la lista C en el orden apropiado de clasificación, de modo que cuando los elementos no sean iguales (líneas 12-14) se agregue el menor a la lista C , como se hace cuando los

```

procedure INTERSECCION ( encab_a, encab_b: ↑ tipo_celda;
    var apc: ↑ tipo_celda );
    { calcula la intersección de las listas clasificadas A y B con celdas de en-
    cabezamiento encab_a y encab_b, dejando el resultado como una
    lista clasificada a cuyo encabezamiento apunta apc }
var
    actual_a, actual_b, actual_c: ↑ tipo_celda;
    { las celdas actuales de las listas A y B y la última celda agregada a la
    lista C }
begin
    (1) new(apc); { crea el encabezamiento de la lista C }
    (2) actual_a := encab_a↑.sig;
    (3) actual_b := encab_b↑.sig;
    (4) actual_c := apc;
    (5) while (actual_a <> nil) and (actual_b <> nil) do begin
        { compara los elementos actuales de las listas A y B }
    (6) if actual_a↑.elemento = actual_b↑.elemento then begin
            { agrega a la intersección }
            (7) new(actual_c↑.sig);
            (8) actual_c := actual_c↑.sig;
            (9) actual_c↑.elemento := actual_a↑.elemento;
            (10) actual_a := actual_a↑.sig;
            (11) actual_b := actual_b↑.sig
        end
        else { elementos distintos }
            (12) if actual_a↑.elemento < actual_b↑.elemento then
                (13) actual_a := actual_a↑.sig
            else
                (14) actual_b := actual_b↑.sig
        end;
        (15) actual_c↑.sig := nil
    end; { INTERSECCION }

```

Fig. 4.5. Procedimiento de intersección que utiliza listas clasificadas.

elementos son iguales. También se deben agregar a la lista C todos los elementos de la lista que no se haya agotado cuando la prueba de la línea (5) falle. Para DIFERENCIA, no se agrega un elemento a la lista C cuando se encuentren elementos iguales; sólo se agrega el elemento actual de la lista A a la lista C cuando sea menor que el elemento actual de la lista B, porque entonces se sabe que el primero no puede encontrarse en la lista B. Además, se añaden a C los elementos de A cuando, y si la prueba de la línea (5) falla porque B se agotó.

El operador ASIGNA(*A, B*) copia la lista *A* en la lista *B*. Obsérvese que este operador no se puede implantar haciendo simplemente que la celda de encabezamiento de *A* apunte al mismo lugar que la celda de encabezamiento de *B*, porque en tal caso cualquier cambio posterior en *B* causaría cambios no esperados en *A*. El operador MIN

es sencillo; sólo devuelve el primer elemento de la lista. SUPRIME y ENCUENTRA se pueden realizar encontrando el elemento objetivo como se expuso para las listas generales y en el caso de SUPRIME, eliminando su celda.

Por último, la inserción no es difícil de implantar, pero debe hacerse de modo que se inserte el nuevo elemento en la posición apropiada. La figura 4.6 muestra un procedimiento INSERTA que toma como parámetros un elemento y un apuntador a la celda de encabezamiento de una lista, e inserta el elemento en la lista. La figura 4.7 muestra las celdas y apuntadores cruciales antes (líneas de trazo continuo) y después (líneas punteadas) de la inserción.

```

procedure INSERTA ( x: tipo_elemento; a: ↑ tipo_celda );
{ inserta x en la lista a cuyo encabezamiento apunta p }
var
    actual, cel_nue: ↑ tipo_celda;
begin
    actual := a;
    while actual↑.sig <> nil do begin
        if actual↑.sig↑.elemento = x then
            return; { si x ya está en la lista, regresa }
        if actual↑.sig↑.elemento > x then
            goto agrega; { salida del ciclo }
        actual := actual↑.sig
    end;
agrega: { actual es ahora la celda después de la cual se debe insertar x }
    new(cel_nue);
    cel_nue↑.elemento := x;
    cel_nue↑.sig := actual↑.sig;
    actual↑.sig := cel_nue
end; { INSERTA }
```

Fig. 4.6. Procedimiento de inserción.

4.5 El diccionario

Cuando se usa un conjunto en el diseño de un algoritmo, quizás no sean necesarias algunas operaciones poderosas como unión e intersección. A menudo, lo único que se necesita es mantener un conjunto de objetos «actuales», con inserciones y supresiones periódicas en el conjunto. Con cierta frecuencia, también puede ser necesario saber si un elemento particular está en el conjunto. Un TDA CONJUNTO con las operaciones INSERTA, SUPRIME y MIEMBRO recibe el nombre de *diccionario*. Se incluirá también ANULA como una operación de diccionario para asignar un valor inicial a cualquier estructura utilizada en la aplicación. Se considerará un ejemplo de realización del diccionario para después analizar algunas realizaciones apropiadas para representar diccionarios.

Ejemplo 4.2. La Sociedad para la Prevención de Injusticias con el Atún (SPIA) mantiene una base de datos que registra los votos más recientes de los legisladores en asuntos de importancia para los amantes del atún. Desde el punto de vista conceptual, esta base de datos consta de dos conjuntos de nombres de legisladores llamados *chicos_buenos* y *chicos_malos*. La sociedad olvida con facilidad los errores del pasado y, de igual modo, tiende a olvidar a quienes fueron amigos. Por ejemplo, si se efectuara una votación para decidir acerca de la prohibición de la pesca de atún en el lago Erie, todos los legisladores que votaran a favor se incluirían en *chicos_buenos* y se excluirían de *chicos_malos*, y lo contrario sucedería con los que votaran en contra. Los legisladores que se abstuvieran, permanecerían en el conjunto donde se encontraran, si estuvieran en alguno.

Cuando está en operación, el sistema de bases de datos acepta tres mandatos, cada uno representado por un solo carácter, seguido de una cadena de diez caracteres que denota el nombre de un legislador; cada mandato está en una línea aparte. Los mandatos son:

1. F (sigue un legislador con voto favorable)
2. D (sigue un legislador con voto desfavorable)
3. ? (determina el estado del legislador que sigue).

También se permite el carácter 'E' en la línea de entrada para señalar el fin del proceso. La figura 4.8 muestra un esbozo del programa, escrito en función del TDA DICCIONARIO aún no definido, que en este caso se pretende que sea un conjunto de cadenas de longitud 10. □

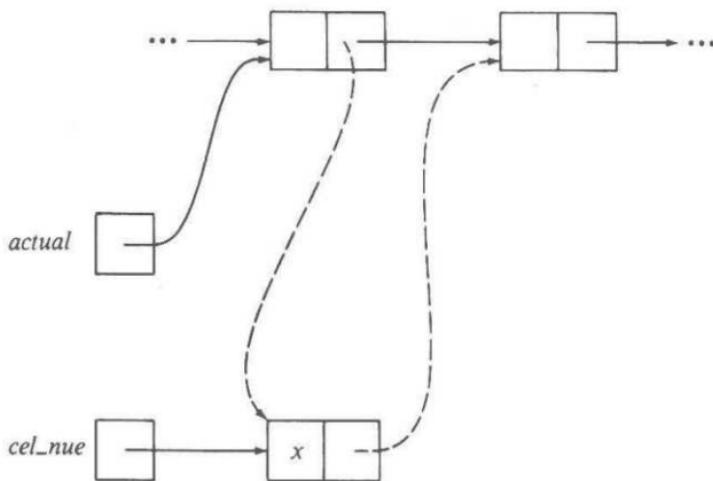


Fig. 4.7. Representación de la inserción.

4.6 Realizaciones sencillas de diccionarios

Un diccionario puede implantarse mediante una lista enlazada, clasificada o no. Otra posible implantación de un diccionario es mediante un vector de bits, siempre que los elementos del conjunto en cuestión estén restringidos a los enteros 1, 2, ..., N para algún N , o a un conjunto que se pueda hacer corresponder con ese conjunto de enteros.

Una posible tercera implantación de diccionarios consiste en usar un arreglo de longitud fija con un apuntador a la última entrada del arreglo en uso. Esta realización sólo es factible si se puede suponer que los conjuntos nunca serán más grandes que la longitud del arreglo. Tiene la ventaja de la sencillez sobre la representación con listas enlazadas, mientras sus desventajas son que 1) los conjuntos no pueden crecer arbitrariamente, 2) la supresión es más lenta, y 3) el espacio no se utiliza de forma eficiente si los conjuntos son de tamaño variable.

A causa de la última razón, no se estudió la realización mediante arreglos en relación con conjuntos cuyas uniones e intersecciones se realizan con frecuencia. No obstante, dado que los arreglos, al igual que las listas, se pueden clasificar, podría considerarse la realización con arreglos que ahora se describe para los diccionarios como una posible realización para conjuntos en general. La figura 4.9 muestra las declaraciones y procedimientos necesarios para complementar la figura 4.8, haciendo de ella un programa ejecutable.

```

program atún ( input, output );
  { base de datos de legisladores }
  type
    tipo_nombre = array[1..10] of char;
  var
    mandato: char;
    legislador: tipo_nombre;
    chicos_buenos, chicos_malos: DICCCIONARIO;
  procedure favorable ( amigo: tipo_nombre );
  begin
    INSERTA(amigo, chicos_buenos);
    SUPRIME(amigo, chicos_malos)
  end; { favorable }

  procedure desfavorable ( enemigo: tipo_nombre );
  begin
    INSERTA(enemigo, chicos_malos);
    SUPRIME(enemigo, chicos_buenos)
  end; { desfavorable }

  procedure consulta ( sujeto: tipo_nombre );
  begin
    if MIEMBRO(sujeto, chicos_buenos) then
      writeln(sujeto, 'es un amigo')
    else
      writeln(sujeto, 'no es un amigo')
  end; { consulta }
end.

```

```

else if MIEMBRO(sujeto, chicos_malos) then
    writeln(sujeto, 'es un enemigo')
else
    writeln('no se tiene información sobre', sujeto)
end; { consulta }

begin { programa principal }
    ANULA(chicos_buenos);
    ANULA(chicos_malos);
    read(mandato);
    while mandato <> 'E' do begin
        readln(legislador);
        if mandato = 'F' then
            favorable(legislador)
        else if mandato = 'D' then
            desfavorable(legislador)
        else if mandato = "?" then
            consulta(legislador)
        else
            error('mandato desconocido');
        read(mandato)
    end
end. { atún }

```

Fig. 4.8. Esbozo del programa de base de datos SPIA.

```

const
    tamaño_máx = { algún número adecuado };
type
    DICCCIONARIO = record
        últ: integer;
        datos: array[1..tamaño_máx] of tipo_nombre
    end;
procedure ANULA ( var A: DICCCIONARIO );
begin
    A.últ := 0
end; { ANULA }

function MIEMBRO ( x: tipo_nombre; var A: DICCCIONARIO ) : boolean;
var
    i: integer;
begin
    for i := 1 to A.últ do
        if A.datos[i] = x then return (true);
    return (false) { si no se encuentra x }
end; { MIEMBRO }

```

```

procedure INSERTA ( x: tipo_nombre; var A: DICCIONARIO );
begin
  if not MIEMBRO(x, A) then
    if A.últ < tamaño_máx then begin
      A.últ := A.últ + 1;
      A.datos[A.últ] := x
    end
    else error('la base de datos está llena')
  end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCIONARIO );
var
  i: integer;
begin
  if A.últ > 0 then begin
    i := 1;
    while (A.datos[i] <> x) and (i < A.últ) do
      i := i + 1;
    { cuando se llega aquí, se ha encontrado x o el elemento
      actual es el último del conjunto A, o ambas cosas }
    if A.datos[i] = x then begin
      A.datos[i] := A.datos[A.últ];
      { pasa el último elemento al lugar ocupado por x; ob-
        sérvase que si i = A.últ, este paso no hace nada,
        pero el siguiente elimina x }
      A.últ := A.últ - 1
    end
  end
end;
end; { SUPRIME }

```

Fig. 4.9. Declaraciones de tipos y procedimientos para un diccionario basado en arreglos.

4.7 La estructura de datos tabla de dispersión

En promedio, la realización de diccionarios mediante arreglos requiere $O(N)$ pasos para la ejecución de una sola instrucción INSERTA, SUPRIME o MIEMBRO en un diccionario de N elementos; si se usa una realización mediante listas, se obtiene una velocidad similar. En la realización mediante vectores de bits, cualquiera de estas tres operaciones utiliza un tiempo constante, pero existe la limitación de trabajar con conjuntos de enteros en un intervalo pequeño para la realización que sea factible.

Existe otra técnica importante y muy útil para implantar diccionarios, y se denomiña «dispersión» (*hashing*). La dispersión utiliza tiempo constante por operación, en promedio, y no existe la exigencia de que los conjuntos sean subconjuntos de algún conjunto universal finito. En el peor caso, este método requiere, para

cada operación, un tiempo proporcional al tamaño del conjunto, como sucede con las realizaciones mediante arreglos y listas. Sin embargo, con un diseño cuidadoso es posible hacer que sea arbitrariamente pequeña la probabilidad de que la dispersión demande más de un tiempo constante para cada operación.

Se considerarán dos formas de dispersión algo diferentes. La primera, llamada dispersión *abierta* o *externa*, permite que el conjunto se almacene en un espacio potencialmente ilimitado, por lo que no impone un límite al tamaño del conjunto. La segunda, llamada dispersión *cerrada* o *interna*, usa un espacio fijo para el almacenamiento, por lo que limita el tamaño de los conjuntos †.

Dispersión abierta

En la figura 4.10 se puede ver la estructura de datos básica para la dispersión abierta. La idea fundamental es que el conjunto (posiblemente infinito) de miembros potenciales se divide en un número finito de clases. Si se desea tener B clases, numeradas de 0 a $B-1$, se usa una función de dispersión h tal que para cada objeto x del tipo de datos de los miembros del conjunto que se va a representar, $h(x)$ sea uno de los enteros de 0 a $B-1$. Lógicamente, el valor de $h(x)$ es la clase a la cual x pertenece. A menudo se da a x el nombre de *clave* y a $h(x)$ el de *valor de dispersión* de x . A las «clases» se les da el nombre de *cubetas* y se dice que x pertenece a la cubeta $h(x)$.

En un arreglo llamado *tabla de cubetas*, indizado por los *números de cubeta* 0, 1, ..., $B-1$, se tienen los encabezamientos de B listas. Los elementos de la i -ésima lista son los miembros del conjunto que se está representando y que pertenecen a la clase i , esto es, aquellos elementos x del conjunto tales que $h(x) = i$.

Se espera que las cubetas tengan casi el mismo tamaño, de modo que la lista para

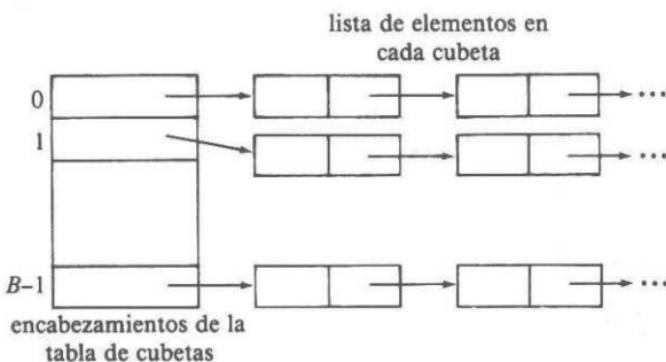


Fig. 4.10. Organización de datos en la dispersión abierta.

† Es posible hacer cambios en la estructura de datos para acelerar la dispersión abierta y permitir que la cerrada maneje conjuntos más grandes. Estas técnicas se describirán después de estudiar los métodos básicos.

cada cubeta sea corta. Entonces, si hay N elementos en el conjunto, en promedio, cada cubeta tendrá N/B miembros. Si se puede estimar N y elegir B aproximadamente igual de grande, entonces una cubeta tendrá en promedio sólo uno o dos miembros, y las operaciones con el diccionario tendrán, en promedio, un número pequeño y constante de pasos, independiente del valor que tenga N (o, en forma equivalente, B).

No siempre está clara la posibilidad de elegir h de modo que un conjunto típico tenga sus miembros distribuidos de forma relativamente uniforme entre las cubetas. Más adelante se estudiará otra vez la forma de elegir h de modo que, en verdad, «disperse» su argumento, es decir, para que $h(x)$ sea un valor «aleatorio» que no dependa de x de ninguna forma trivial.

```
function h ( x: tipo_nombre ) : 0..B-1;
var
    i, suma: integer;
begin
    suma := 0;
    for i := 1 to 10 do
        suma := suma + ord(x[i]);
    h := suma mod B
end; { h }
```

Fig. 4.11. Una función de dispersión sencilla h .

Con el fin de especificar, se presenta ahora una función de dispersión para cadenas de caracteres que es muy buena, aunque no perfecta. La idea es considerar a los caracteres como enteros, utilizando el código de caracteres de la máquina para definir la correspondencia. Pascal proporciona la función incorporada ord , donde $ord(c)$ es el código entero del carácter c . Así, si x es una clave y el tipo de las claves es $\text{array}[1..10] \text{ of char}$ (lo que se llamó `tipo_nombre` en el Ejemplo 4.2), se podría declarar la función de dispersión h como en la figura 4.11. En esa función, se suman los enteros para cada carácter y se divide el resultado entre B , tomando el residuo, que es un entero entre 0 y $B-1$.

En la figura 4.12 se observan las declaraciones de las estructuras de datos para una tabla de dispersión abierta y los procedimientos que realizan las operaciones para un diccionario. El tipo de los miembros del diccionario se supone que es `tipo_nombre` (un arreglo de caracteres), de modo que estas declaraciones se pueden usar en forma directa en el ejemplo 4.2. Debe observarse que en la figura 4.12 se ha hecho que los encabezamientos de las listas de cubetas sean apuntadores a celdas, en lugar de celdas completas. Esto se hizo así, porque la tabla de cubetas, donde se encuentran los encabezamientos, podría ocupar tanto espacio como las propias listas si fuera un arreglo de celdas, en lugar de uno de apuntadores. Obsérvese, sin embargo, que se paga un precio por este ahorro de espacio. Ahora, el código del procedimiento `SUPRIME` debe hacer distinción entre la primera celda y las restantes.

```

const
     $B = \{ \text{constante adecuada} \}$ 
type
    tipo_celda = record
        elemento: tipo_nombre;
        sig:  $\uparrow$  tipo_celda
    end;
    DICCCIONARIO = array[0..B-1] of  $\uparrow$  tipo_celda;

procedure ANULA ( var A: DICCCIONARIO );
var
    i: integer;
begin
    for i := 0 to B-1 do
        A[i] := nil
end; { ANULA }

function MIEMBRO ( x: tipo_nombre; var A: DICCCIONARIO ) : boolean;
var
    actual:  $\uparrow$  tipo_celda;
begin
    actual := A[h(x)];
    { actual es inicialmente el encabezado de la cubeta para x }
    while actual <> nil do
        if actual^.elemento = x then
            return (true)
        else
            actual := actual^.sig;
    return (false) { si no se encuentra x }
end; { MIEMBRO }

procedure INSERTA ( x: tipo_nombre; var A: DICCCIONARIO );
var
    cubeta: integer;
    encab_ant:  $\uparrow$  tipo_celda;
begin
    if not MIEMBRO(x, A) then begin
        cubeta := h(x);
        encab_ant := A[cubeta];
        new(A[cubeta]);
        A[cubeta]^.elemento := x;
        A[cubeta]^.sig := encab_ant
    end
end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCCIONARIO );
var

```

```

actual: ↑ tipo_celda;
cubeta: integer;
begin
  cubeta: = h(x);
  if A[cubeta] <> nil then begin
    if A[cubeta]↑.elemento = x then { x está en la primera celda }
      A[cubeta] := A[cubeta]↑.sig { suprime x de la lista }
    else begin { x, si está, no ocupa la primera celda de la cubeta }
      actual := A[cubeta]; { actual apunta a la celda anterior }
      while actual↑.sig <> nil do
        if actual↑.sig↑.elemento = x then begin
          actual↑.sig := actual↑.sig↑.sig
          { suprime x de la lista }
        return { salida del ciclo }
      end
      else { aún no se encuentra x }
        actual := actual↑.sig
    end
  end
end; { SUPRIME }

```

Fig. 4.12. Realización de un diccionario mediante una tabla de dispersión abierta.

Dispersión cerrada

Una tabla de dispersión cerrada guarda los miembros del diccionario en la tabla de cubetas, en vez de usar esa tabla para almacenar encabezamientos de listas. En consecuencia, parece que sólo es posible colocar un elemento en una cubeta; sin embargo, la dispersión cerrada tiene asociada una *estrategia de redispersión*. Si se intenta colocar x en la cubeta $h(x)$ y esta ya tiene un elemento —situación denominada *colisión*—, la estrategia de redispersión elige una sucesión de localidades alternas $h_1(x)$, $h_2(x)$, ... dentro de la tabla de cubetas, en la cual es posible colocar x . Se prueba en todas esas localidades, en orden, hasta encontrar una vacía. Si ninguna está vacía, la tabla está llena y no es posible insertar x .

Ejemplo 4.3. Supóngase que $B = 8$ y que las claves a, b, c y d tienen valores de dispersión $h(a) = 3, h(b) = 0, h(c) = 4, h(d) = 3$. Se usará la estrategia de redispersión más sencilla, denominada *dispersión lineal*, en la que $h_i(x) = (h(x) + i) \bmod B$. Así, por ejemplo, si se deseara insertar a y se encontrara que la cubeta 3 ya está llena, se probaría con las cubetas 4, 5, 6, 7, 0, 1 y 2, en ese orden.

En principio, se supone que la tabla está vacía, esto es, que cada cubeta guarda un valor especial *vacio*, que no es igual a ningún valor que podría intentarse insertar \dagger . Si se insertan a, b, c y d , en ese orden, en una tabla inicialmente vacía, resulta

\dagger Si el tipo de los miembros del diccionario no sugiere un valor adecuado para *vacio*, se puede hacer que cada cubeta tenga un campo adicional de un bit que informe si está vacía o no.

que a va a la cubeta 3, b a la 0 y c a la 4. Al insertar d , primero se hace la prueba con $h(d) = 3$ para encontrar que está llena. Luego se intenta con $h_1(d) = 4$ y ocurre lo mismo. Por último, se prueba con $h_2(d) = 5$, se encuentra un espacio vacío y d se coloca allí. Las posiciones resultantes están en la figura 4.13. \square

La prueba de pertenencia de un elemento x al conjunto requiere examinar $h(x)$, $h_1(x)$, $h_2(x)$, ..., hasta encontrar x o una cubeta vacía. Para ver por qué es posible detenerse al alcanzar una cubeta vacía, supóngase primero que las supresiones no están permitidas. Si $h_3(x)$ es la primera cubeta vacía encontrada en la serie, no es posible que x esté en las cubetas $h_4(x)$, $h_5(x)$ o más adelante en la sucesión, porque x no pudo haber sido colocada allí a menos que $h_3(x)$ hubiera estado llena en el momento de insertarla.

0	<i>b</i>
1	
2	
3	<i>a</i>
4	<i>c</i>
5	<i>d</i>
6	
7	

Fig. 4.13. Tabla de dispersión parcialmente llena.

Sin embargo, obsérvese que si se permiten las supresiones, nunca puede existir la seguridad, si se alcanza una cubeta vacía sin encontrar x , de que x no se encuentra en alguna otra parte, y la cubeta ahora vacía estaba ocupada cuando se insertó x . Cuando deban hacerse supresiones, el enfoque más efectivo para resolver este problema es colocar una constante llamada *suprimido* en una cubeta que contenga un elemento que se desee suprimir. Es importante que haya una diferencia entre *suprimido* y *vacío*, la constante que se encuentra en todas las cubetas que nunca se han llenado. De esta manera, es posible permitir supresiones sin tener que buscar en la tabla completa durante la prueba MIEMBRO. En el momento de la inserción, es posible tratar *suprimido* como un espacio disponible, de modo que con suerte el espacio de un elemento suprimido puede volver a utilizarse. Sin embargo, como el espacio de un elemento suprimido no es reclamable de inmediato, cosa que sí ocurre en la dispersión abierta, puede preferirse el esquema abierto al cerrado.

Ejemplo 4.4. Supóngase que se desea probar si e está en el conjunto representado en la figura 4.13. Si $h(e) = 4$, se prueba con las cubetas 4, 5 y luego 6. La cubeta 6 está vacía y como e no se ha encontrado, la conclusión es que no está en el conjunto.

Si se suprime c , es necesario colocar la constante *suprimido* en la cubeta 4. Así,

al buscar d y comenzar en $h(d) = 3$, se pueden examinar 4 y 5 para encontrar d , y no detenerse en 4 como se hubiera hecho de haber puesto *vacio* en esa cubeta. \square

En la figura 4.14 se observan las declaraciones de tipos y las operaciones para el TDA DICCIONARIO con miembros de conjunto del tipo tipo_nombre y la tabla de dispersión cerrada como estructura fundamental. Se utiliza una función de dispersión arbitraria h , de la cual la figura 4.11 es una posibilidad, y se usa la estrategia de dispersión lineal para redispersar en caso de colisiones. Por conveniencia, se identifica *vacio* con una cadena de diez espacios y *suprimido* con una de diez asteriscos, con la suposición de que ninguna de esas cadenas puede ser una clave real. (En la base de datos SPIA esas cadenas tienen poca probabilidad de ser nombres de legisladores.) El procedimiento INSERTA(x, A) primero usa *localiza* para determinar si x ya está en A , y si no, utiliza una rutina especial *localizal* para encontrar una localidad en la cual se pueda insertar x . Obsérvese que *localizal* busca localidades marcadas tanto con *vacio* como con *suprimido*.

```

const
    vacio = '          '; { 10 espacios }
    suprimido = '*****'; { 10 asteriscos }
type
    DICCIONARIO = array[0..B-1] of tipo_nombre;
procedure ANULA ( var A: DICCIONARIO );
var
    i: integer;
begin
    for i := 0 to B-1 do
        A[i] := vacio
end; { ANULA }

function localiza ( x: tipo_nombre ) : integer;
{ localiza examina el DICCIONARIO desde la cubeta para h(x) hasta que
  se encuentre x, o una cubeta vacía, o se haya revisado toda la tabla
  y determinado que no contiene a x; localiza devuelve el índice
  de la cubeta en la que se detiene por cualquiera de esas razones }

var
    inicial, i: integer;
    { inicial guarda h(x); i cuenta el número de cubetas examinadas
      hasta el momento cuando se busca x }
begin
    inicial := h(x);
    i := 0;
    while (i < B) and (A[(inicial + i) mod B] <> x) and
        (A[(inicial + i) mod B] <> vacio) do
        i := i + 1;
    return ((inicial + i) mod B)
end; { localiza }

```

```

function localiza ( x: tipo_nombre ): integer;
  { como localiza, pero también se detiene en una entrada con suprimido
    y devuelve ese valor }

function MIEMBRO ( x: tipo_nombre; var A: DICCIONARIO ) : boolean;
begin
  if A[localiza(x)] = x then
    return (true)
  else
    return (false)
end; { MIEMBRO }

procedure INSERTA ( x: tipo_nombre; var A: DICCIONARIO );
var
  cubeta: integer;
begin
  if A[localiza(x)] = x then
    return; { x ya está en A }
  cubeta := localiza(x);
  if (A[cubeta] = vacío) or (A[cubeta] = suprimido) then
    A[cubeta] := x
  else
    error ('INSERTA falló: la tabla está llena')
end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCIONARIO );
var
  cubeta: integer;
begin
  cubeta := localiza(x);
  if A[cubeta] = x then
    A[cubeta] := suprimido
end; { SUPRIME }

```

Fig. 4.14. Realización de un diccionario mediante una tabla de dispersión cerrada.

4.8 Estimación de la eficiencia de las funciones de dispersión

Como ya se mencionó, la dispersión es una manera eficiente de representar diccionarios y otros tipos de datos abstractos basados en conjuntos. En esta sección se examinará el tiempo promedio por operación de diccionario en una tabla de dispersión abierta. Si hay B cubetas y N elementos almacenados en la tabla de dispersión, las cubetas tienen, en promedio, N/B miembros y se puede esperar que una operación normal INSERTA, SUPRIME o MIEMBRO lleve un tiempo $O(1 + N/B)$. La constante 1 representa el tiempo necesario para hallar la cubeta, y N/B , el tiempo para buscar en ella. Si puede elegirse B casi igual a N , este tiempo se convierte

en una constante para cada operación. Por tanto, el tiempo promedio para insertar, suprimir y probar la pertenencia de un elemento al conjunto, suponiendo que un elemento cualquiera tiene la misma probabilidad de ser dispersado en cualquier cubeta, es una constante que no depende de N .

Supóngase que se tiene un programa escrito en algún lenguaje como Pascal y se desea insertar todos los identificadores que aparecen en el programa en una tabla de dispersión. Cada vez que se encuentra la declaración de un nuevo identificador, éste se inserta en la tabla de dispersión después de verificar que ya no se encuentra ahí. Durante esta fase, es razonable suponer que un identificador tiene la misma probabilidad de ser dispersado en cualquier cubeta. Por tanto, se puede construir una tabla de dispersión de N elementos en tiempo $O(N(1 + N/B))$. Al elegir B igual a N , este tiempo es $O(N)$.

En la siguiente fase, los identificadores se hallan en el cuerpo del programa; es necesario localizarlos en la tabla de dispersión para recuperar información asociada con ellos. Pero, ¿cuál es el tiempo estimado para localizar un identificador? Si el elemento que se busca tiene la misma probabilidad que cualquiera de los elementos de la tabla, entonces el tiempo estimado para buscarlo es simplemente el tiempo promedio que demanda la inserción de un elemento. Para comprender esto, obsérvese que el tiempo que demanda buscar una vez cada elemento en la tabla es el empleado en insertarlo, suponiendo que los elementos siempre se agregan al final de la lista de la cubeta apropiada. Entonces, el tiempo estimado para una búsqueda es también $O(1 + N/B)$.

El análisis anterior supone que una función de dispersión distribuye los elementos de manera uniforme en las cubetas. ¿Existen funciones de esta clase? Una función como la de la figura 4.11 (que convierte caracteres en enteros, suma y toma el residuo módulo B) puede considerarse una función típica de dispersión. El siguiente ejemplo examina su funcionamiento.

Ejemplo 4.5. Supóngase que se emplea la función de la figura 4.11 para dispersar 100 claves que consisten en las cadenas de caracteres A0, A1, ..., A99, en una tabla de 100 cubetas. Si se establece que $ord(0)$, $ord(1)$, ..., $ord(9)$ forman una progresión aritmética, como ocurre con los códigos de caracteres más comunes ASCII y EBCDIC, es fácil verificar que las claves se dispersan hasta en 29 de las 100 cubetas † y que la cubeta más grande contiene A18, A27, A36, ..., A90, es decir, 9 de los 100 elementos. Si se calcula el número promedio de pasos para una inserción, partiendo del hecho de que la inserción del i -ésimo elemento en una cubeta requiere $i+1$ pasos, se obtienen 395 pasos para las 100 claves. En comparación, el estimado $N(1+N/B)$ sugiere 200 pasos. □

La función de dispersión sencilla de la figura 4.11 puede tratar ciertos conjuntos de entradas, como las cadenas consecutivas del ejemplo 4.5, de manera no aleatoria. Hay funciones de dispersión «más aleatorias»; como ejemplo, puede usarse la idea de elevar al cuadrado y tomar los dígitos medios. Así, si se tiene como

† Obsérvese que A2 y A20 no necesariamente se dispersan en la misma cubeta, pero, por ejemplo, A23 y A41 deben hacerlo.

clave un número n de 5 dígitos y se eleva al cuadrado, se obtiene un número de 9 ó 10 dígitos. Los «dígitos medios», como los que están entre las posiciones cuarta y séptima a la derecha, tienen valores que dependen de casi todos los dígitos de n ; por ejemplo, el cuarto dígito depende de todos ellos excepto del que se encuentre más a la izquierda de n , y el quinto depende de todos los dígitos de n . De modo que si $B = 100$, se podrían tomar los dígitos sexto y quinto para formar el número de cubeta.

Esta idea se puede generalizar para situaciones en las que B no sea una potencia de 10. Supóngase que las claves son enteros en el intervalo 0, 1, ..., K . Si se escoge un entero C tal que BC^2 sea casi igual a K^2 , la función

$$h(n) = \lfloor n^2/C \rfloor \bmod B$$

efectivamente extrae un dígito de base B del centro de n^2 .

Ejemplo 4.6. Si $k = 1000$ y $B = 8$, se puede elegir $C = 354$. Entonces,

$$h(456) = \lfloor \frac{207936}{354} \rfloor \bmod 8 = 587 \bmod 8 = 3. \quad \square$$

Para usar la estrategia de «elevar al cuadrado y tomar el medio» cuando las claves son cadenas de caracteres, primero se agrupan los caracteres en la cadena de derecha a izquierda, en bloques de un número fijo de caracteres, como 4, rellenando a la izquierda con espacios si es necesario. Se trata cada bloque como un solo entero formado al concatenar los códigos binarios de los caracteres. Por ejemplo, ASCII maneja un código de caracteres de 7 bits, de modo que los caracteres pueden considerarse como «dígitos» de base 2^7 ó 128. De este modo, se puede considerar la cadena de caracteres $abcd$ como el entero $(128)^3 a + (128)^2 b + (128)c + d$. Después de convertir todos los bloques a enteros, se suma † y se procede como se sugirió con anterioridad para los enteros.

Análisis de dispersión cerrada

En un esquema de dispersión cerrada, la velocidad de inserción y de otras operaciones no sólo depende de lo aleatoriamente que la función de dispersión distribuye los elementos en las cubetas, sino también de lo bien que la estrategia de redispersión evita colisiones adicionales cuando una cubeta ya esté llena. Por ejemplo, la estrategia lineal para resolver colisiones no es la mejor posible. Aunque el análisis de este hecho no se efectúa en este libro, es posible observar lo siguiente. Tan pronto como se llenen unas cuantas cubetas consecutivas, cualquier clave que se dispersa a una de ellas será enviada al final del grupo por la estrategia de redispersión, con lo cual el tamaño de ese grupo de cubetas consecutivas se incrementará. De este modo, es probable encontrar sucesiones más largas de cubetas consecutivas

† Si las cadenas pueden ser muy largas, esta suma tendría que hacerse para alguna constante c . Por ejemplo, c valdría uno más que el mayor entero que se pudiera obtener de un solo bloque.

llenas que si los elementos llenaran las cubetas al azar. Más aún, las sucesiones de bloques llenos causan largas secuencias de intentos antes de que un elemento encuentre una cubeta vacía, de modo que tener grupos de cubetas llenas extraordinariamente grandes hace más lentas la inserción y otras operaciones.

Podría desearse saber cuántos intentos (o *sondeos*) se necesitan en promedio para insertar un elemento cuando N de las B cubetas están llenas, al suponer que todas las posibles combinaciones de N de las B cubetas tienen la misma probabilidad de estar llenas. Por lo general se presume, aunque no se demuestre, que ninguna estrategia de dispersión cerrada puede tener un rendimiento temporal medio mejor que ésa para operaciones con diccionarios. Se derivará entonces una fórmula para el costo de inserción cuando las localidades alternas empleadas por la estrategia de redispersión se elijan al azar. Por último, se considerarán algunas estrategias de redispersión, que se aproximan a ese comportamiento aleatorio.

La probabilidad de una colisión en el sondeo inicial es N/B . Suponiendo una colisión, la primera redispersión intentará con una de $B - 1$ cubetas, de las cuales $N - 1$ estarán llenas, de manera que la probabilidad de al menos dos colisiones es $\frac{N(N-1)}{B(B-1)}$. De modo similar, la probabilidad de al menos i colisiones es

$$\frac{N(N-1) \cdots (N-i+1)}{B(B-1) \cdots (B-i+1)}. \quad (4.3)$$

Si B y N son grandes, esta probabilidad se approxima a $(N/B)^i$. El número promedio de sondeos es uno (para inserción exitosa) más la suma con todos los $i \geq 1$ de la probabilidad de al menos i colisiones, esto es, alrededor de $1 + \sum_{i=1}^{\infty} (N/B)^i$, o $\frac{B}{B-N}$. Se puede demostrar que el valor exacto de la sumatoria, cuando la fórmula (4.3) se usa para $(N/B)^i$, es $\frac{B+1}{B+1-N}$, de manera que la aproximación obtenida es buena, excepto cuando N se acerca bastante a B .

Obsérvese que $\frac{B+1}{B+1-N}$ crece en una forma muy lenta conforme N comienza a crecer de 0 a $B-1$, que es el mayor valor de N para el cual es posible otra inserción. Por ejemplo, si N es la mitad de B , se necesitan alrededor de dos sondeos para la siguiente inserción. El costo de inserción promedio por cubeta para llenar M de las B cubetas es $\frac{1}{M} \sum_{N=0}^{M-1} \frac{B+1}{B+1-N}$, que se approxima a $\frac{1}{M} \int_0^{M-1} \frac{B}{B-x} dx$, o $\frac{B}{M} \log_e \left(\frac{B}{B-M+1} \right)$. Llenar por completo la tabla ($M = B$) requiere un promedio de $\log_e B$ sondeos por cubeta o $B \log_e B$ sondeos en total. Sin embargo, llenar la tabla al 90% de su capacidad ($M = 0.9 B$) sólo requiere $B((10/9)\log_{10} 10)$ o unos 2.56 B sondeos. El costo medio de la prueba de pertenencia para un elemento no existente es idén-

tico al de insertar el siguiente elemento, pero el costo de la prueba de pertenencia para un elemento que está en el conjunto, es el costo promedio de todas las inserciones realizadas hasta el momento, el cual es menor en gran medida si la tabla está bastante llena. Las supresiones tienen el mismo costo medio que las pruebas de pertenencia, pero, a diferencia de la dispersión abierta, las supresiones de una tabla de dispersión cerrada no ayudan a acelerar las inserciones o pruebas de pertenencia posteriores. Se debe subrayar que si no se permite que las tablas de dispersión cerradas se llenen en más de una fracción fija, menor que uno, de su capacidad total, el costo medio de las operaciones es una constante; esa constante crece conforme lo hace la fracción de la capacidad que se permite usar. En la figura 4.15 se puede ver una gráfica del costo de las inserciones, supresiones y pruebas de pertenencia como una función del porcentaje de la tabla que está lleno cuando se realiza la operación.

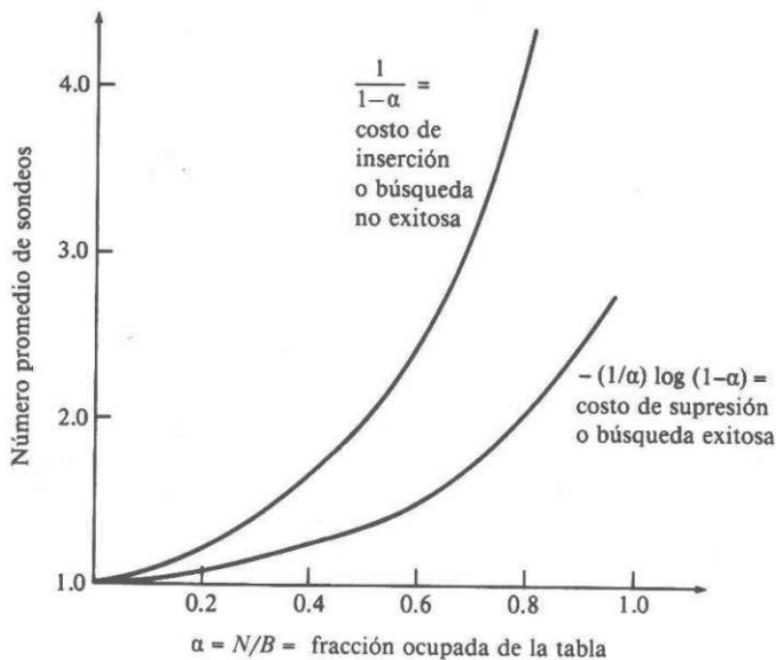


Fig. 4.15. Costo medio de operación.

Estrategias «aleatorias» de resolución de colisiones

Se ha observado que la estrategia de redispersión lineal tiende a agrupar las cubetas llenas en bloques grandes consecutivos. Tal vez se pueda lograr un comportamiento más «aleatorio» si se hacen los sondeos a intervalos constantes mayores que

uno. Esto es, sea $h_i(x) = (h(x) + ci) \bmod B$ para algún $c > 1$. Por ejemplo, si $B = 8$, $c = 3$ y $h(x) = 4$, se sondearían las cubetas 4, 7, 2, 5, 0, 3, 6 y 1, en ese orden. Por supuesto, si c y B tienen un factor común mayor que uno, esta estrategia no permite siquiera buscar en todas las cubetas; como ejemplo, pruébese con $B = 8$ y $c = 2$. Pero todavía más significativo es el hecho de que, aun si c y B son primos relativos (no tienen factores comunes), se tiene el mismo problema de «amontonamiento» que con la dispersión lineal, aunque aquí tienden a aparecer las secuencias de cubetas llenas, separadas por una distancia c . Este fenómeno hace más lentas las operaciones, como en la dispersión lineal, puesto que un intento de inserción en una cubeta llena originará un desplazamiento en una cadena de cubetas llenas separadas por la distancia c , y la longitud de esa cadena se incrementará en uno.

En realidad, cualquier estrategia de redispersión en la que el blanco de un sondeo sólo dependa del blanco del sondeo anterior (en lugar de depender del número de sondeos no exitosos ya realizados, la cubeta original $h(n)$, o el valor x de la clave misma) exhibirá la propiedad de amontonamiento de la dispersión lineal. Quizá la estrategia más simple en la que no se presenta este problema sea hacer que $h_i(x) = (h(x) + d_i) \bmod B$, donde d_1, d_2, \dots, d_{B-1} , es una permutación «aleatoria» de los enteros 1, 2, ..., $B - 1$. Desde luego, en todas las inserciones, supresiones y pruebas de pertenencia se usa la misma sucesión d_1, \dots, d_{B-1} ; la disposición «aleatoria» de los enteros se decide una sola vez, al diseñar el algoritmo de redispersión.

La generación de números «aleatorios» es un asunto complejo, pero por fortuna muchos métodos comunes producen una sucesión de ellos que van de 1 a cierto límite. Estos generadores de números aleatorios, cuando se reajustan sus valores iniciales para cada operación con la tabla de dispersión, sirven para generar la sucesión deseada d_1, \dots, d_{B-1} .

Un enfoque efectivo utiliza «sucesiones de registros de desplazamiento». Sea B una potencia de 2, y k una constante entre 1 y $B - 1$. Empiezan con algún número d_1 en el intervalo de 1 a $B - 1$, generan los números siguientes de la sucesión tomando el valor anterior, lo doblan y, si el resultado es mayor que B , le sustraen B y toman la suma módulo 2 bit a bit del resultado y la constante seleccionada k . La suma módulo 2 bit a bit de x e y , que se escribe $x \oplus y$, se calcula escribiendo x e y en binario, con ceros a la izquierda si hace falta, para que ambos sean de la misma longitud, y formando los números cuya representación binaria tenga un 1 en las posiciones en que x o y , pero no ambos, también lo tengan.

Ejemplo 4.7. $25 \oplus 13$ se calcula tomando

$$\begin{array}{rcl} 25 & = & 11001 \\ 13 & = & 01101 \\ \hline 25 \oplus 13 & = & 10100 \end{array}$$

Obsérvese que esta «suma» se puede considerar como una suma binaria ordinaria en la que se ignora el residuo. □

No todo valor de k produce una permutación de 1, 2, ..., $B - 1$; algunas veces, un número se repite antes de generarlos todos. Sin embargo, para una B dada, existe

una probabilidad pequeña, pero finita, de que algún valor particular de k funcione, y sólo hace falta encontrar una k para cada B .

Ejemplo 4.8. Sea $B = 8$. Si se toma $k = 3$, será fácil generar todos los enteros del 1 al 7. Por ejemplo, si se comienza con $d_1 = 5$, se calcula luego d_2 , doblando primero d_1 para obtener 10. Como $10 > 8$, se resta 8 para obtener 2 y luego se calcula $d_2 = 2 \oplus 3 = 1$. Obsérvese que $x \oplus 3$ se puede calcular al tomar el complemento de los dos últimos bits de x .

Es bueno observar las representaciones binarias en 3 bits de d_1, d_2, \dots, d_7 , que se muestran en la figura 4.16, junto con su método de cálculo. Obsérvese que la multiplicación por 2, en binario corresponde a un desplazamiento hacia la izquierda. De ahí se obtiene una indicación acerca del origen del término «sucesión de registros de desplazamiento».

Se deberá verificar que también se genera una permutación de 1, 2, ..., 7 si se elige $k = 5$, pero que otros valores de k fracasan. \square

	$d_1 = 101 = 5$
desplazamiento	1010
supresión del primer 1	010
$\oplus 3$	$d_2 = 001 = 1$
desplazamiento	$d_3 = 010 = 2$
desplazamiento	$d_4 = 100 = 4$
desplazamiento	1000
supresión del primer 1	000
$\oplus 3$	$d_5 = 011 = 3$
desplazamiento	$d_6 = 110 = 6$
desplazamiento	1100
supresión del primer 1	100
$\oplus 3$	$d_7 = 111 = 7$

Fig. 4.16. Cálculo de una sucesión de registros de desplazamiento.

Reestructuración de tablas de dispersión

Si se usa una tabla de dispersión abierta, el tiempo medio de las operaciones crece con N/B , cantidad que crece con rapidez conforme el número de elementos excede el número de cubetas. De manera similar, para una tabla de dispersión cerrada, como se vio en la figura 4.15, la eficiencia disminuye a medida que N se acerca a B , y no es posible que N exceda a B .

Para mantener el tiempo constante por operación, que en teoría es posible con las tablas de dispersión, se sugiere que si N crece demasiado, por ejemplo $N \geq 0.9B$ para una tabla cerrada o $N \geq 2B$ para una abierta, sencillamente se produzca una nueva tabla de dispersión con el doble de cubetas. La inserción de los elementos actuales del conjunto en la nueva tabla llevará, en promedio, menos tiempo que el requerido para insertarlas en la tabla más pequeña, y ese costo se compensa cuando se realizan operaciones de diccionario posteriores.

4.9 Realización del TDA CORRESPONDENCIA

Recuérdese el análisis del TDA CORRESPONDENCIA del capítulo 2, en el cual se definió una correspondencia como una función de los elementos de un dominio a los elementos de un contradominio. Las operaciones de este TDA son:

1. ANULA(A) inicializa la correspondencia A , dejando cada elemento del dominio sin valor asignado en el contradominio.
2. ASIGNA(A, d, r) define $A(d)$ como r .
3. CALCULA(A, d, r) devuelve verdadero y hace que r tome el valor $A(d)$ si $A(d)$ está definido; en caso contrario, devuelve falso.

Las tablas de dispersión son una forma efectiva de obtener correspondencias. Las operaciones ASIGNA y CALCULA se realizan de la misma manera que las operaciones INSERTA y MIEMBRO para un diccionario. Considerese primero una tabla de dispersión abierta. Se supone que la función de dispersión $h(d)$ asocia elementos del dominio con números de cubeta. Mientras que para el diccionario las cubetas consistían en listas enlazadas de elementos, para la correspondencia se necesita una lista de elementos del dominio pareados con sus correspondientes valores del contradominio. Para esto, la definición de celda de la figura 4.12 se reemplaza por

```
type
  tipo_celda = record
    elemento_dominio: tipo_dominio;
    contradominio: tipo_contradominio;
    sig: ^tipo_celda
  end
```

donde tipo_dominio y tipo_contradominio son cualesquiera tipos que tengan los elementos del dominio y contradominio de la correspondencia. La declaración de CORRESPONDENCIA queda como

```
type
  CORRESPONDENCIA = array[0..B - 1] of ^tipo_celda
```

Este arreglo es el arreglo de cubetas para una tabla de dispersión. El procedimiento ASIGNA está escrito en la figura 4.17. Los códigos de ANULA y CALCULA se dejan como ejercicio.

De manera parecida, es posible usar una tabla de dispersión cerrada como una correspondencia. Se definen las celdas como campos de elementos del dominio y de elementos del contradominio, y se declara una CORRESPONDENCIA como un arreglo de celdas. Como ocurre con las tablas de dispersión abiertas, se hace que la función de dispersión se aplique a los elementos del dominio, no a los del contradominio. Se deja como ejercicio la realización de las operaciones de correspondencia usando una tabla de dispersión cerrada.

4.10 Colas de prioridad

La cola de prioridad es un TDA basado en el modelo de conjunto con las operaciones INSERTA y SUPRIME-MIN, así como con ANULA para asignar valores iniciales a la estructura de datos. Con el fin de definir la nueva operación SUPRIME-MIN, primero se supone que los elementos del conjunto tienen una función «prioridad» definida sobre ellos; para cada elemento a , $p(a)$, la *prioridad* de a es un número real o, en términos más generales, un miembro de algún conjunto linealmente ordenado. La operación INSERTA tiene el significado habitual, mientras que SUPRIME-MIN es una función que devuelve un elemento de más baja prioridad y, como efecto colateral, lo suprime del conjunto. Así, como su nombre indica, SUPRIME-MIN es una combinación de las operaciones SUPRIME y MIN estudiadas antes en este capítulo.

```

procedure ASIGNA ( var A: CORRESPONDENCIA; a: tipo_dominio;
    r: tipo_contradominio );
var
    cubeta: integer;
    actual: ↑ tipo_celda;
begin
    cubeta := h(d);
    actual := A[cubeta];
    while actual < > nil do
        if actual ↑. elemento_dominio = d then begin
            actual ↑. contradominio := r; { reemplaza el valor anterior
                de d }
            return
        end
        else
            actual := actual ↑. sig;
    { en este punto, d no se encontró en la lista }
    actual := A[cubeta]; { usa actual para almacenar la primera celda }
    new(A[cubeta]);
    A[cubeta] ↑. elemento_dominio := d;
    A[cubeta] ↑. contradominio := r;
    A[cubeta] ↑. sig := actual
end; { ASIGNA }
```

Fig. 4.17. El procedimiento ASIGNA para una tabla de dispersión abierta.

Ejemplo 4.9. El término «cola de prioridad» proviene de la siguiente forma de usar este TDA. La palabra «cola» sugiere la espera de cierto servicio por ciertas personas u objetos, y la palabra «prioridad» sugiere que ese servicio no se proporciona por medio de la disciplina «primero en llegar, primero en ser atendido», que es la base del TDA COLA, sino que cada persona tiene una prioridad basada en la urgencia de su necesidad. Un ejemplo es la sala de espera de un hospital, donde los pacientes

con problemas potencialmente mortales serán atendidos antes que los demás, sin importar el tiempo de espera.

Como ejemplo más corriente del uso de las colas de prioridad, un sistema de cómputo de tiempo compartido necesita mantener un conjunto de procesos que esperan servicio. En general, los diseñadores de estos sistemas desean hacer que los procesos cortos parezcan instantáneos (en la práctica, una respuesta en uno o dos segundos parece instantánea), de modo que a estos procesos se les da prioridad sobre los que ya tienen un consumo sustancial de tiempo. Un proceso que requiera varios segundos de tiempo de cómputo no puede ser instantáneo, por lo que es razonable la estrategia de diferir estos procesos hasta que todos los que tienen posibilidades de parecer instantáneos hayan sido atendidos. Sin embargo, si no se tiene cuidado, los procesos que ya hayan tomado un tiempo mucho mayor que el promedio, quizás no vuelvan a obtener una porción de tiempo y queden esperando para siempre.

Una posible manera de favorecer los procesos cortos sin bloquear los largos, consiste en dar a un proceso P la prioridad $100t_{\text{cons}}(P) - t_{\text{inic}}(P)$. El parámetro t_{cons} es el tiempo consumido hasta el momento por el proceso, y t_{inic} es el momento en que se inició el proceso, medido a partir de cierto «instante cero». Obsérvese que, en general, las prioridades serán enteros negativos grandes, a menos que se elija medir t_{inic} a partir de un instante en el futuro. Obsérvese también que 100 en la fórmula anterior es un «número mágico»; se selecciona de forma que sea algo mayor que el mayor número de procesos que puedan estar activos a la vez. Así, si siempre se elige el proceso con el número de prioridad más bajo y no hay demasiados procesos cortos, a la larga, un proceso que no termine rápido recibirá el 1% del tiempo de procesador. Si esto es mucho o muy poco, otra constante puede reemplazar el 100 de la fórmula de la prioridad.

Los procesos se representan mediante registros con un identificador de proceso y un número de prioridad. Esto es, se define

```
type
  tipo_proceso = record
    id: integer;
    prioridad: integer
  end;
```

La prioridad de un proceso es el valor del campo de prioridad, que se ha definido como un entero. La función de prioridad se puede definir como sigue:

```
function p (a: tipo_proceso): integer;
begin
  return (a.prioridad)
end;
```

Para seleccionar el proceso que reciba una porción de tiempo, el sistema mantiene una cola de prioridad ESPERA con elementos de tipo tipo_proceso, y emplea dos procedimientos, *inicial* y *selecciona*, para manejar la cola de prioridad mediante las operaciones *INSERTA* y *SUPRIME_MIN*. Siempre que un proceso comienza,

se llama al procedimiento *inicial*, el cual coloca un registro para ese proceso en ESPERA. El procedimiento *selecciona* es llamado cuando el sistema dispone de una porción de tiempo assignable a algún proceso. El registro del proceso que resulte seleccionado se elimina de ESPERA, pero *selecciona* lo retiene para reingresarlo en la cola con una nueva prioridad, que es la anterior incrementada en 100 veces la cantidad de tiempo utilizada.

Se emplea la función *tiempo_actual*, que devuelve el tiempo actual en cualesquier unidades que el sistema use, como microsegundos, y el procedimiento *ejecuta* (*P*) que hace que el proceso con identificador *P* se ejecute durante una porción de tiempo. La figura 4.18 muestra los procedimientos *inicial* y *selecciona*. □

```

procedure inicial ( P: integer );
{ inicial coloca un proceso con identificador P en la cola }
var
    proceso: tipo_proceso;
begin
    proceso.id := P,
    proceso.prioridad := -tiempo_actual;
    INSERTA (proceso, ESPERA)
end; { inicial }

procedure selecciona;
{ selecciona asigna una porción de tiempo al proceso de mayor prioridad }
var
    tiempo_inicio, tiempo_fin: integer;
    proceso: tipo_proceso;
begin
    proceso := ↑ SUPRIME_MIN(ESPERA)
    { SUPRIME_MIN devuelve un apuntador al elemento eliminado }
    tiempo_inicio := tiempo_actual;
    ejecuta(proceso.id);
    tiempo_fin := tiempo_actual;
    proceso.prioridad := proceso.prioridad + 100* (tiempo_fin -
        tiempo_inicio);
    { ajusta la prioridad para incorporar la cantidad de tiempo usado }
    INSERTA (proceso, ESPERA)
    { devuelve el proceso seleccionado a la cola con la nueva prioridad }
end; { selecciona }

```

Fig. 4.18. Asignación de tiempo a procesos.

4.11 Realizaciones de colas de prioridad

Con excepción de la tabla de dispersión, todas las realizaciones estudiadas hasta el momento para los conjuntos son también apropiadas para las colas de prioridad. La razón de que la tabla de dispersión no sea adecuada, es que no proporciona

un medio conveniente de encontrar el menor elemento, por lo que la dispersión sólo agrega complicaciones y no mejora el rendimiento respecto a una lista enlazada, por ejemplo.

En caso de manejarse una lista enlazada, se tienen las opciones de clasificarla o dejarla sin clasificar. Si la lista se clasifica, es fácil encontrar un mínimo; basta tomar su primer elemento. En cambio, la inserción requiere revisar en promedio la mitad de la lista para mantenerla clasificada. Por otro lado, es posible dejar la lista sin clasificar, lo que facilita la inserción y dificulta la selección del mínimo.

Ejemplo 4.10. Se obtendrá SUPRIME_MIN para una lista no clasificada de elementos del tipo tipo_proceso, que se definió en el ejemplo 4.9. El encabezado de la lista es una celda vacía; las aplicaciones de INSERTA y ANULA son directas, y la aplicación por medio de listas clasificadas se deja como ejercicio. La figura 4.19 proporciona la declaración de las celdas, para el tipo COLA_PRIORIDAD, y para el procedimiento SUPRIME_MIN. □

Realización de colas de prioridad mediante árboles parcialmente ordenados

Tanto si se desea emplear listas clasificadas como no clasificadas para representar colas de prioridad, se debe gastar un tiempo proporcional a n para realizar INSERTA o SUPRIME_MIN en conjuntos de tamaño n . Existe otra realización en la que ambas operaciones requieren $O(\log n)$ pasos, una mejora sustancial para valores grandes de n (como $n \geq 100$). La idea básica consiste en organizar los elementos de la cola de prioridad en un árbol binario que esté lo más balanceado posible; hay un ejemplo en la figura 4.20. En el nivel más bajo, en el cual pueden faltar algunas hojas, se exige que las hojas que falten estén a la derecha de las que se encuentran en el nivel más bajo.

Más importante aún, el árbol es *parcialmente ordenado*, es decir, la prioridad del nodo v no es mayor que la de los hijos de v , donde la prioridad de un nodo es el número de prioridad del elemento almacenado en ese nodo. Obsérvese en la figura 4.20 que los nodos con número de prioridad pequeño no necesitan estar a niveles más altos que los de número de prioridad más grande. Por ejemplo, el nivel tres tiene 6 y 8, que son números de prioridad menores que el 9, que aparece en el nivel dos. En cambio, el padre de 6 y 8 tiene prioridad 5, la cual es, y debe ser, al menos tan pequeña como las prioridades de sus hijos.

Para ejecutar SUPRIME_MIN, se devuelve el elemento de menor prioridad, que, como se puede ver con facilidad, debe estar en la raíz. Sin embargo, si lo único que se hace es eliminar la raíz, lo que queda ya no es un árbol. Para que se mantenga la propiedad de árbol parcialmente ordenado, lo más balanceado y con hojas tan a la izquierda posible, se toma la hoja de más a la derecha del nivel más bajo y se coloca de modo provisional en la raíz. La figura 4.21(a) muestra este cambio a partir de la figura 4.20. Despues, este elemento se coloca en el árbol lo más abajo posible, intercambiándolo con el hijo que tenga la prioridad más baja, hasta que el elemento

esté en una hoja o en una posición cuya prioridad no sea mayor que la de cualquiera de sus hijos.

En la figura 4.21(a) se debe intercambiar la raíz con su hijo de menor prioridad, el cual tiene prioridad 5. El resultado de este intercambio se muestra en la figura 4.21(b). El elemento que se está desplazando hacia abajo es todavía mayor que sus hijos, los cuales tienen prioridades 6 y 8. Al intercambiarlo con el menor de los dos, 6, se logra el árbol de la figura 4.21(c). Este árbol tiene ya la propiedad de ser parcialmente ordenado.

En esta filtración, si un nodo v tiene un elemento con prioridad a y sus hijos tienen prioridades b y c , y si al menos b o c son menores que a , el intercambio de a con el menor de b y c logra que v tenga un elemento cuya prioridad sea menor que la de cualquiera de sus hijos. Para demostrar esto, supóngase que $b \leq c$. Después del intercambio, el nodo v adquiere la prioridad b , y sus hijos, las prioridades a y c . Se supuso que $b \leq c$ y se dijo que a es mayor que b o c . Por tanto, es seguro que se cumplirá que $b \leq a$. Así, el proceso de inserción esbozado con anterioridad filtrará un elemento hacia abajo en el árbol hasta que no haya más violaciones de la propiedad de orden parcial.

```

type
  tipo_celda = record
    elemento: tipo_proceso;
    siguiente: ↑ tipo_celda
  end;

  COLA_CON_PRIORIDAD = ↑ tipo_celda;
  { la celda apuntada es un encabezado de lista }

function SUPRIME_MIN ( var A: COLA_CON_PRIORIDAD ) : ↑ tipo_celda;
  var
    actual: ↑ tipo_celda; { celda anterior a la "examinada" }
    menor_prioridad: integer; { prioridad más baja encontrada }
    preganador: ↑ tipo_celda; { celda anterior a la del elemento con menor
      prioridad }

  begin
    if A ↑.siguiente = nil then
      error ('no puede encontrar el menor de una lista vacía')
    else begin
      menor_prioridad := p(A ↑.siguiente^.elemento);
      { p devuelve la prioridad del primer elemento. Obsérvese que
        A apunta a una celda de encabezado que no contiene un
        elemento }

      preganador := A;
      actual := A ↑.siguiente;
      while actual↑.siguiente <> nil do begin
        { compara las prioridades del ganador actual con las del elemento
          siguiente }
    
```

```

if  $p(actual^{\dagger}.siguiente^{\dagger}.elemento) < menor\_prioridad$  then begin
    pganador := actual;
    menor_prioridad :=  $p(actual^{\dagger}.siguiente^{\dagger}.elemento)$ 
end;
actual :=  $actual^{\dagger}.siguiente$ 
end;
SUPRIME_MIN := pganador^{\dagger}.siguiente;
{ devuelve el apuntador al ganador }
pganador^{\dagger}.siguiente := pganador^{\dagger}.siguiente^{\dagger}.siguiente
{ elimina el ganador de la lista }
end
end; { SUPRIME_MIN }

```

Fig. 4.19. Realización de una cola de prioridad mediante listas enlazadas.

Obsérvese también que SUPRIME_MIN consume un tiempo $O(\log n)$, aplicado a un conjunto de n elementos. Esto ocurre porque ningún camino del árbol tiene más de $1 + \log n$ nodos y el proceso de forzar el descenso de un elemento en el árbol consume un tiempo constante por nodo. Obsérvese que para cualquier constante c , la cantidad $c(1 + \log n)$ es, a lo sumo, igual a $2c \log n$ para $n \geq 2$. Por consiguiente, $c(1 + \log n)$ es $O(\log n)$.

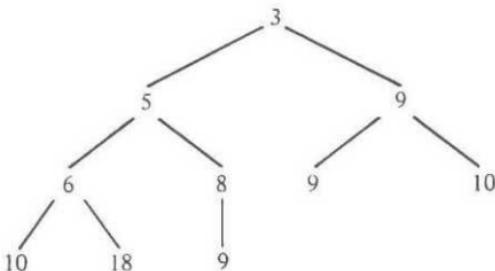


Fig. 4.20. Un árbol parcialmente ordenado.

Ahora se considerará cómo debería trabajar INSERTA. En primer lugar, se coloca el nuevo elemento lo más a la izquierda posible en el nivel más bajo, iniciando un nuevo nivel si el más bajo actual se encuentra lleno en su totalidad. La figura 4.22(a) muestra el resultado de colocar un elemento con prioridad 4 en el árbol de la figura 4.20. Si el nuevo elemento tiene una prioridad más baja que la de su padre, se intercambia con él. El nuevo elemento queda entonces en una posición de menor prioridad que cualquiera de sus hijos, pero puede ser también que tenga menor prioridad que su padre, en cuyo caso se debe intercambiar con él y repetir el proceso hasta que el nuevo elemento llegue hasta la raíz o alcance una posición en la que tenga mayor prioridad que su padre. La figura 4.22(b) y (c) muestra el ascenso de 4 en este proceso.

Se podría demostrar que los pasos anteriores dan como resultado un árbol parcialmente ordenado. No se intentará una demostración rigurosa de este hecho, pero sí se observará que un elemento con prioridad a puede llegar a ser el padre de un elemento con prioridad b de tres maneras. (En el razonamiento siguiente se identificará un elemento con su prioridad.)

1. a es el nuevo elemento y asciende en el árbol reemplazando al padre anterior de b . Sea c la prioridad del padre anterior de b . Entonces, $a < c$, de otro modo, el intercambio no habría tenido lugar. Pero $c \leq b$, puesto que el árbol original estaba parcialmente ordenado. Por tanto, $a < b$. Tómese como ejemplo la figura 4.22(c), donde 4 se convierte en el padre de 6 después de reemplazar a un padre de mayor prioridad, 5.
2. a fue desplazado hacia abajo en el árbol debido a un intercambio con el nuevo elemento. En este caso, a tiene que haber sido un antecesor de b en el árbol parcialmente ordenado original. Por tanto, $a \leq b$. Por ejemplo, en la figura 4.22(c), 5 se convierte en el padre de los elementos con prioridad 8 y 9. Originalmente, 5 era el padre del primero y el «abuelo» del segundo.
3. b sería el nuevo elemento y asciende hasta ser un hijo de a . Si ocurriera que $a > b$, entonces a y b se intercambiarían en el siguiente paso, con lo que se eliminaría la violación de la propiedad parcialmente ordenada.

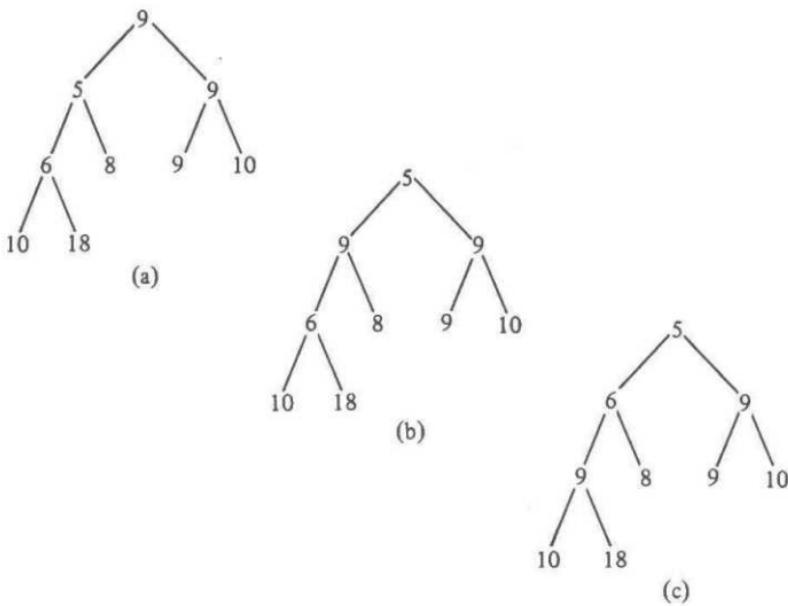


Fig. 4.21. Descenso de un elemento en un árbol.

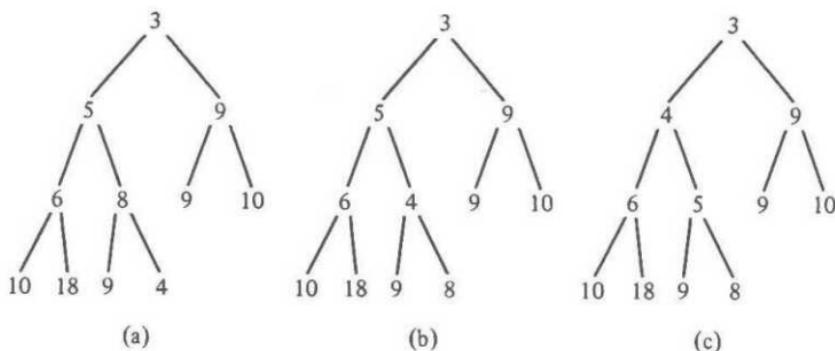


Fig. 4.22. Inserción de un elemento.

El tiempo necesario para efectuar una inserción es proporcional a la distancia que el elemento asciende en el árbol. Como en el caso de SUPRIME-MIN, se observa que esta distancia tal vez no sea mayor que $1 + \log n$, de modo que tanto INSERTA como SUPRIME MIN dan $O(\log n)$ pasos.

Realización por arreglos de los árboles parcialmente ordenados

El hecho de que los árboles que se han estado considerando sean binarios, lo más balanceados posibles, y tengan sus hojas del nivel más bajo lo más a la izquierda posibles, permitirá utilizar para ellos una representación poco usual, conocida como *montículo (heap)*. De haber n nodos, se usan las n primeras posiciones de un arreglo A . $A[1]$ contiene la raíz. El hijo izquierdo del nodo $A[i]$, si existe, está en $A[2i]$, y el hijo derecho, si existe, en $A[2i + 1]$. Una manera equivalente de expresar esto es decir que el padre de $A[i]$ es $A[i \text{ div } 2]$, para $i > 1$. Otra observación, también equivalente, es que los nodos del árbol llenan $A[1], A[2], \dots, A[n]$ nivel a nivel, desde arriba, y de izquierda a derecha dentro de cada nivel. Por ejemplo, la figura 4.20 corresponde a un arreglo que contiene los elementos 3, 5, 9, 6, 8, 9, 10, 10, 18, 9.

Se puede declarar una cola de prioridad con elementos de cierto tipo, como tipo_proceso del ejemplo 4.9, como un arreglo de tipos tipo_proceso y un entero *ult* que indica cuál es el último elemento de un arreglo que en un momento dado está en uso. Si se supone que *tam_máx* es el tamaño deseado para los arreglos de colas de prioridad, se puede declarar:

```

type
    COLA_DE_PRIORIDAD = record
        contenido: array[1..tam_máx] of tipo_proceso;
        últ: integer
    end;

```

La realización de las operaciones con colas de prioridad se puede ver en la figura 4.23.

```

procedure ANULA ( var A: COLA_DE_PRIORIDAD );
begin
  A.últ := 0
end; { ANULA }

procedure INSERTA ( x: tipo_proceso; var A: COLA_DE_PRIORIDAD );
var
  i: integer;
  temp: tipo_proceso;
begin
  if A.últ >= long_máx then
    error('la cola de prioridad está llena')
  else begin
    A.últ := A.últ + 1;
    A.contenido[A.últ] := x;
    i := A.últ; { i es el índice de la posición actual de x }
    while (i > 1) and (p(A.contenido[i]) < p(A.contenido[i div 2])) do
      begin { sube x en el árbol y lo intercambia con su padre
              de mayor prioridad. Recuérdese que p calcula
              la prioridad de un elemento de tipo_proceso }
      temp := A.contenido[i];
      A.contenido[i] := A.contenido[i div 2];
      A.contenido[i div 2] := temp;
      i := i div 2
    end
  end
end;
end; { INSERTA }

function SUPRIME_MIN ( var A: COLA_DE_PRIORIDAD ) : ↑ tipo_proceso;
var
  i, j: integer;
  temp: tipo_proceso;
  mínimo: ↑ tipo_proceso;
begin
  if A.últ = 0 then
    error('la cola de prioridad está vacía')
  else begin
    new(mínimo);
    mínimo↑ := A.contenido[1];
    { debe devolverse un apuntador a una copia de la raíz de A }
    A.contenido[1] := A.contenido[A.últ];
    A.últ := A.últ - 1;
    { mueve el último elemento hacia el principio }
    i := 1; { i es la posición actual del que antes era
              el último elemento }
  end
end;

```

```

while i < = A.últ div 2 do begin
    { lleva hacia abajo el anterior último elemento
      en el árbol }
    if (p(A.contenido[2*i]) < p(A.contenido[2*i + 1]))
        or (2*i = A.últ) then
            j := 2*i
        else
            j := 2*i + 1;
    { j será el hijo de i que tiene la menor prioridad o bien,
      si 2*i = A.últ, j será el único hijo de i }
    if p(A.contenido[i]) > p(A.contenido[j]) then begin
        { intercambia el anterior último elemento con el hijo
          de menor prioridad }
        temp := A.contenido[i];
        A.contenido[i] := A.contenido[j];
        A.contenido[j] := temp;
        i := j
    end
    else
        return (mínimo) { no puede meterlo más }
end;
return (mínimo) { se metió hasta llegar a una hoja }
end
end; { SUPRIME_MIN }

```

Fig. 4.23. Realización de colas de prioridad mediante arreglos.

4.12 Algunas estructuras complejas de conjuntos

En esta sección se considerarán dos usos más complejos de los conjuntos para representar datos. El primer problema será el de representar asociaciones de muchos a muchos, como las que pueden presentarse en un sistema de bases de datos. Un segundo caso de estudio mostrará cómo un par de estructuras de datos que representan el mismo objeto (una correspondencia en el ejemplo que se analizará), pueden dar una representación más eficiente que cualquiera de ellas por separado.

Asociaciones de muchos a muchos y la estructura de listas múltiples

Un ejemplo de asociación de muchos a muchos entre estudiantes y cursos está representada en la figura 4.24. Esta asociación se llama «de muchos a muchos» porque puede haber muchos estudiantes que tomen un curso y cada estudiante puede tomar muchos cursos.

	CS101	CS202	CS303
Alan	X	X	X
Alejandro			
Alicia			X
Amanda	X		
Andrés		X	X
Angélica	X		X

*Inscripción***Fig. 4.24.** Ejemplo de relación entre estudiantes y cursos.

De vez en cuando, el responsable de los cursos puede desear insertar o eliminar estudiantes de los cursos, determinar cuáles estudiantes están tomando un curso dado, o saber qué cursos está tomando un estudiante dado. La estructura de datos más simple con la que es posible responder estas preguntas es obvia; basta usar el arreglo bidimensional sugerido por la figura 4.24, donde el valor 1 (o verdadero) reemplaza las X y el valor 0 (o falso) reemplaza los espacios.

Por ejemplo, para insertar un estudiante en un curso se necesita una correspondencia, *CE*, tal vez aplicada mediante una tabla de dispersión, que traduzca nombres de estudiante a índices del arreglo y otra, *CC*, que traduzca nombres de curso a índices del arreglo. Entonces, para insertar al estudiante *e* en el curso *c*, simplemente se asigna

$$\text{Inscripción}[\text{CE}(e), \text{CC}(c)] := 1.$$

Las supresiones se realizan haciendo que este elemento tome el valor 0. Para hallar los cursos que toma un estudiante de nombre *e*, se recorre la fila *CE(e)* y, de manera similar, se recorre la columna *CC(c)* para hallar los estudiantes que están en el curso *c*.

¿Por qué sería deseable buscar una estructura de datos más apropiada? Considérese una universidad grande con, tal vez, 1000 cursos y 20 000 estudiantes, que toman en promedio tres cursos cada uno. El arreglo sugerido por la figura 4.24 tendría 20 000 000 de elementos, de los cuales 60 000, o el 0.3 %, tendrían el valor 1 †. Un arreglo así, que recibe el nombre de *dispersa*, para indicar que casi todos sus elementos valen cero, se puede representar en mucho menos espacio con sólo listar las entradas que no son nulas. Más aún, se puede gastar mucho tiempo revisando una columna de 20 000 entradas en busca de, en promedio, 60 que no son nulos y las revisiones de filas pueden llevar también mucho tiempo.

Una forma de mejorar esto, consiste en plantear el problema como si se tratara del mantenimiento de una colección de conjuntos. Dos de esos conjuntos son *E* y *C*, los conjuntos de todos los estudiantes y todos los cursos. Cada elemento de *E* es en realidad un registro de un tipo como

† Si ésta fuera una base de datos real, el arreglo se guardaría en almacenamiento secundario. Sin embargo, esta estructura de datos malgastaría mucho espacio.

```

type
  tipo_estudiante = record
    ident: integer;
    nombre: array[1..30] of char;
  end

```

y habrá que inventar un tipo de registro parecido para los cursos. Para obtener la estructura pensada, se necesita un tercer conjunto, I , cuyos elementos representen las inscripciones. Cada uno de los elementos de I representará una de las celdas del arreglo de la figura 4.24 que contenga una X. Los elementos de I serían registros de un tipo fijo. Hasta este punto no se sabe qué campos van en esos registros †, pero pronto se sabrá de ellos. Por el momento, basta postular que hay un registro de inscripción por cada entrada marcada con X en la matriz y que los registros de inscripción son distinguibles unos de otros de alguna manera.

También se requieren conjuntos que representen respuestas a las preguntas cruciales: dado un estudiante o un curso, ¿cuáles son los cursos o estudiantes, según el caso, asociados? Sería interesante tener, para cada estudiante e , un conjunto C_e de todos los cursos que e estuviera tomando y, por otro lado, un conjunto E_c de todos los estudiantes que siguen el curso c . Tales conjuntos serían difíciles de obtener, porque no habría límite para el número de conjuntos en los que podría estar cualquier elemento, lo cual daría lugar a complicados registros de estudiantes y cursos. En cambio, si se podría hacer que E_c y C_e fueran conjuntos de apuntadores, en vez de registros, pero existe un método que permite un ahorro significativo de espacio y ofrece respuestas igual de rápidas a las preguntas sobre estudiantes y cursos.

Sea cada conjunto C_e el conjunto de registros de inscripción correspondientes al estudiante e y algún curso c . Es decir, si se considera una inscripción como un par (e, c) , entonces

$$C_e = \{(e, c) \mid e \text{ está tomando el curso } c\}.$$

De la misma manera, se puede definir

$$E_c = \{(e, c) \mid e \text{ está tomando el curso } c\}.$$

Obsérvese que la única diferencia en el significado de las propiedades de estos dos conjuntos es que, en el primer caso, e es constante, y en el segundo lo es c . Por ejemplo, con base en la figura 4.24, $C_{\text{Alejandro}} = \{(\text{Alejandro}, \text{CS101}), (\text{Alejandro}, \text{CS202})\}$ y $E_{\text{CS101}} = \{(\text{Alejandro}, \text{CS101}), (\text{Amanda}, \text{CS101}), (\text{Angélica}, \text{CS101})\}$.

Estructuras de listas múltiples

En general, una estructura de listas múltiples es cualquier colección de celdas, donde algunas contienen más de un apuntador y pueden, por tanto, pertenecer a más de una lista a la vez. Para cada tipo de celda de una estructura de listas múltiples, es importante distinguir entre los campos apuntadores, de modo que se pueda se-

† En la práctica, sería útil colocar algunos campos, como calificaciones y otros, en los registros de inscripción, pero el planteamiento original del problema no los requiere.

guir una lista en particular sin que haya confusión con respecto a cuál de los diferentes apuntadores de una celda en particular se debe seguir.

Para el caso en cuestión, es posible colocar un campo apuntador en cada registro de estudiante y curso que apunte al primer registro de inscripción en C_e o E_c , respectivamente. Cada registro de inscripción necesita dos campos apuntadores: uno que se llamará c_sig , para apuntar a la siguiente inscripción en la lista que representa al conjunto C_e , al cual pertenece el registro, y otro, e_sig , para apuntar al siguiente elemento del conjunto E_c al que pertenece.

Resulta que un registro de inscripción no indica en forma explícita el estudiante ni el curso que representa. Esta información está implícita en las listas en las cuales aparece el registro de inscripción. Llámese *propietarios* del registro de inscripción a los registros de estudiante y de curso que encabezan estas listas. Entonces, para poder decir qué cursos toma el estudiante e , es preciso examinar los registros de inscripción de C_e y hallar para cada uno su registro de curso propietario. Esto podría hacerse colocando un apuntador en cada registro de inscripción para el registro de curso propietario, y podría necesitarse también un apuntador para el registro de estudiante propietario.

Si bien se pueden usar esos apuntadores con el objeto de responder a las preguntas en el menor tiempo posible, existe la alternativa de lograr un ahorro considerable de espacio †, al costo de hacer más lentos algunos cálculos, si se eliminan los apuntadores y se coloca al final de cada lista E_c un apuntador al registro de curso propietario, y al final de cada lista C_e , un apuntador al registro de estudiante propietario. En esta forma, cada registro de estudiante y de curso será parte de un anillo que incluya todos los registros de inscripción de los cuales es propietario. Estos anillos están representados en la figura 4.25, para los datos de la figura 4.24. Obsérvese que los registros de inscripción tienen como primer apuntador c_sig , y como segundo, e_sig .

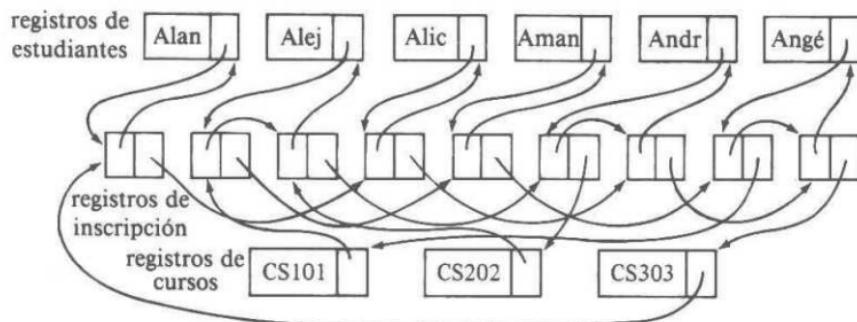


Fig. 4.25. Representación de la figura 4.24 con listas múltiples.

† Obsérvese que es probable que haya muchos más registros de inscripción que registros de estudiantes o cursos, de modo que reducir de tamaño los registros de inscripción permite disminuir la necesidad total de espacio casi en la misma proporción.

Ejemplo 4.11. Para responder a una pregunta como «¿qué estudiantes toman el curso CS101?», se busca el registro de curso correspondiente a CS101. Cómo encontrar este registro depende de la forma en que se mantiene el conjunto de cursos. Por ejemplo, podría haber una tabla de dispersión que tuviera todos esos registros, y para obtener el registro deseado se aplicaría cierta función de dispersión a «CS101».

Luego se sigue el apuntador del registro de CS101 al primer registro de inscripción del anillo de CS101. Este es, en la figura, el segundo registro de inscripción desde la izquierda. Después, hay que buscar al estudiante propietario de este registro de inscripción, lo cual se logra siguiendo los apuntadores *c-sig* (el primero en los registros de inscripción), hasta llegar a un registro de estudiantes \dagger . En este caso, después del tercer registro de inscripción se llega al registro de estudiante para Alejandro; ahora se sabe que Alejandro está tomando el curso CS101.

A continuación, se debe encontrar el siguiente estudiante CS101, y se hace siguiendo el apuntador *e-sig* (el segundo apuntador) del segundo registro de inscripción, el cual conduce al quinto registro de inscripción. El apuntador *c-sig* de ese registro conduce directamente a su propietario, Amanda, de modo que ella está en el curso CS101. Por último, se sigue el apuntador *e-sig* del quinto registro de inscripción hasta el octavo. El anillo de apuntadores *c-sig* de ese registro conduce al noveno registro de inscripción, y de ahí, al registro de estudiante de Angélica, de modo que ella está inscrita en CS101. El apuntador *e-sig* del octavo registro de inscripción conduce de vuelta a CS101, por tanto no hay más estudiantes inscritos en CS101. \square

La operación del ejemplo 4.11 se puede expresar en términos abstractos como sigue:

```
for cada registro de inscripción en el conjunto para CS101 do begin
    e := el estudiante propietario del registro de inscripción;
    imprime (e)
end
```

Esta asignación a *e* se puede expresar como

```
f := e;
repeat
    f := f. c-sig
until
    f es un apuntador a un registro de estudiante;
    e := campo nombre-estudiante del registro apuntado por f;
```

donde *e* es un apuntador al primer registro de inscripción en el conjunto para CS101.

Para aplicar una estructura como la de la figura 4.25 en Pascal, se necesita un solo tipo de registro, con variantes para cubrir los casos de registros de estudiantes,

\dagger Se debe tener alguna forma de identificar los tipos de registro y se dará momentáneamente una manera de hacerlo.

cursos e inscripciones. En Pascal esto tiene que ser así, ya que los campos *c-sig* y *e-sig* tienen la capacidad de apuntar a tipos de registro distintos. Sin embargo, esta estructura resuelve uno de los problemas que quedan, pues ahora es fácil decir a qué tipo de registro se llega conforme se recorre un anillo. La figura 4.26 muestra una posible declaración de los registros y un procedimiento que imprime los nombres de los estudiantes inscritos en un curso en particular.

Estructuras de datos duales para mayor eficiencia

Con frecuencia, un problema aparentemente simple de representación de un conjunto o correspondencia, conlleva un difícil problema de elección de estructuras de datos. La elección de una estructura de datos para el conjunto simplifica ciertas operaciones, pero hace que otras lleven demasiado tiempo, y, al parecer, no existe una estructura de datos que haga más sencillas todas las operaciones. En tales casos, la solución suele ser el uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia.

Supóngase que se desea mantener una «escala de tenistas» en la que cada jugador esté situado en un solo «peldaño». Los jugadores nuevos se agregan en la base de la escala, es decir, en el peldaño con la numeración más alta. Un jugador puede retar a otro que esté en el peldaño inmediato superior, y si le gana, cambia de peldaño con él. Se puede representar esta situación mediante un tipo de datos abstracto cuyo modelo fundamental sea una correspondencia de nombres (cadenas de caracteres) con peldanos (los enteros 1, 2, ...).

Las tres operaciones a realizar son:

1. AGREGA(*nombre*) agrega a la persona nombrada al peldaño de numeración más alta.
2. RETA(*nombre*) es una función que devuelve el nombre de la persona del peldaño *i* - 1 si el jugador nombrado está en el peldaño *i*, *i* > 1.
3. CAMBIA(*i*) intercambia los nombres de los jugadores que estén en los peldaños *i* e *i* - 1, *i* > 1.

type

```

tipo_e = array[1..20] of char;
tipo_c = array[1..5] of char;
clase_registro = (estudiante, curso, inscripción);
tipo_registro = record
    case clase : clase_registro of
        estudiante : (nombre_estudiante: tipo_e;
                      primer_curso: ^tipo_registro);
        curso: (nombre_curso: tipo_c;
                  primer_estudiante: ^tipo_registro);
        inscripción: (c_sig, e_sig: ^tipo_registro)
end;
```

```

procedure imprime_estudiantes ( nombre_c: tipo_c );
var
  c, e, f: tipo_registro;
begin
  c := apuntador al registro de curso con c^.nombre_curso = nom-
  bre_c;
  { depende de la aplicación del conjunto curso }
  e := c^.primer_estudiante;
  { e recorre el anillo de inscripciones apuntadas por c }
  while e^.clase = inscripción do begin
    f := e;
    repeat
      f := f^.c.sig
    until
      f^.clase = estudiante;
    { ahora f apunta al estudiante a quien pertenece la inscripción e^. }
    writeln(f^.nombre_estudiante);
    e := e^.e.sig
  end
end

```

Fig. 4.26. Realización de una búsqueda en una lista múltiple.

Obsérvese que se ha elegido pasar a CAMBIA sólo el número de peldaño más alto, mientras que las otras dos operaciones tienen un nombre como argumento.

Como alternativa, se podría considerar el uso de un arreglo *ESCALA*, en la que *ESCALA[i]* sea el nombre de la persona del peldaño *i*. Si además se lleva la cuenta del número de jugadores, la adición de un jugador al primer peldaño desocupado se puede hacer en un pequeño número constante de pasos.

CAMBIA también es fácil, puesto que sólo hay que intercambiar dos elementos del arreglo, pero RETA(*nombre*) requiere examinar todo el arreglo en busca del nombre, lo cual lleva un tiempo $O(n)$, si *n* es el número de jugadores en la escala.

Por otra parte, se podría considerar el uso de una tabla de dispersión para representar la correspondencia de nombres a peldaños. En el supuesto de que es posible mantener el número de cubetas proporcional al número de jugadores, AGRE-GA llevaría en promedio un tiempo $O(1)$. A un reto le llevaría un tiempo promedio $O(1)$ buscar el nombre dado, un tiempo $O(n)$ encontrar el nombre que ocupa el peldaño con el siguiente número más bajo, dado que para eso se necesitaría buscar en toda la tabla de dispersión. El intercambio de jugadores requeriría un tiempo $O(n)$ para hallar los jugadores en los peldaños *i* e *i* - 1.

Supóngase, sin embargo, que se combinan las dos estructuras. Las celdas de la tabla de dispersión contendrán pares compuestos de un nombre y un peldaño, mientras que el arreglo tendrá en *ESCALA[i]* un apuntador a la celda correspondiente al jugador que ocupe el peldaño *i*, como se sugiere en la figura 4.27.

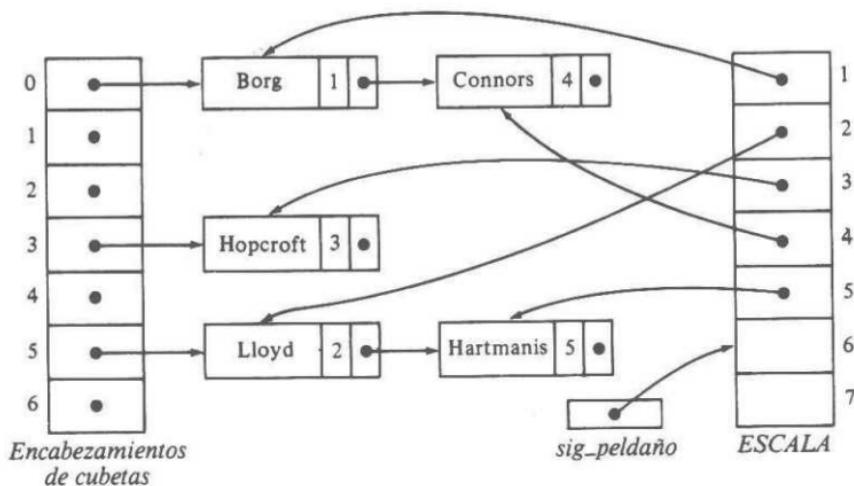


Fig. 4.27. Estructura combinada para alto rendimiento.

Ahora se puede agregar un nombre mediante una inserción en la tabla de dispersión en un tiempo $O(1)$ en promedio, colocando también un apuntador a la celda recién creada dentro del arreglo *ESCALA* en la posición marcada por el cursor *sig-peldaño* de la figura 4.27. Para los retos, se busca el nombre en la tabla de dispersión en un tiempo $O(1)$ en promedio, obteniéndose el peldaño i correspondiente al jugador dado y luego se sigue el apuntador en *ESCALA*[$i - 1$] a la celda del jugador que se va a retar. La consulta de *ESCALA*[$i - 1$] lleva un tiempo constante en el peor caso y la búsqueda en la tabla de dispersión demanda un tiempo $O(1)$ en promedio, de modo que RETA es $O(1)$ en el caso promedio.

A CAMBIA(i) le lleva un tiempo $O(1)$ hallar las celdas de los jugadores en los peldaños i e $i - 1$, intercambiar los números de peldaño de esas celdas, e intercambiar en *ESCALA* los apuntadores a las dos celdas. Así, CAMBIA requiere un tiempo constante incluso en el peor caso.

Ejercicios

4.1 Si $A = \{1, 2, 3\}$ y $B = \{3, 4, 5\}$, ¿cuáles son los resultados de

- a) UNION(A, B, C)
- b) INTERSECCION(A, B, C)
- c) DIFERENCIA(A, B, C)
- d) MIEMBRO(1, A)
- e) INSERTA(1, A)
- f) SUPRIME(1, A)
- g) MIN(A)?

- *4.2 Escribase un procedimiento en función de las operaciones básicas con conjuntos que imprima todos los elementos de un conjunto (finito). Puede suponerse que se dispone de un procedimiento para imprimir un objeto del tipo de los elementos. El conjunto a imprimir no debe quedar destruido. ¿Qué estructuras de datos serían las más apropiadas para implantar conjuntos en este caso?
- 4.3 La realización de conjuntos mediante vectores de bits se puede usar siempre que el «conjunto universal» se pueda traducir a los enteros de 1 a N . Describase cómo se haría esa traducción si el conjunto universal fuera
- los enteros 0, 1, ..., 99
 - los enteros de n a m para cualquier $n \leq m$
 - los enteros $n, n+2, n+4, \dots, n+2k$, para cualesquiera n y k
 - los caracteres 'a', 'b', ..., 'z'
 - arreglos de dos caracteres, cada uno de ellos elegidos entre 'a' y 'z'.
- 4.4 Escribanse procedimientos ANULA, UNION, INTERSECCION, MIEMBRO, MIN, INSERTA y SUPRIME para conjuntos representados mediante listas enlazadas, por medio de las operaciones abstractas del TDA lista clasificada. Obsérvese que la figura 4.5 es un procedimiento para INTERSECCION que maneja una realización específica del TDA lista.
- 4.5 Repítase el ejercicio 4.4 para las realizaciones de conjuntos siguientes:
- tabla de dispersión abierta (úsense operaciones abstractas con listas dentro de las cubetas).
 - tabla de dispersión cerrada con resolución lineal de colisiones.
 - lista no clasificada (empléense operaciones abstractas con listas).
 - un arreglo de longitud fija con un apuntador a la última posición usada.
- 4.6 Para cada una de las operaciones y aplicaciones de los ejercicios 4.4 y 4.5, proporcionese el orden de magnitud del tiempo de ejecución con conjuntos de tamaño n .
- 4.7 Supóngase que se están dispersando enteros en una tabla de dispersión de siete cubetas sirviéndose de la función de dispersión $h(i) = i \bmod 7$.
- Muéstrese la tabla de dispersión abierta si se insertan los cubos perfectos 1, 8, 27, 125, 216, 343.
 - Repítase el apartado a) usando una tabla de dispersión cerrada con resolución lineal de colisiones.
- 4.8 Supóngase que se está usando una tabla de dispersión cerrada con cinco cubetas y la función de dispersión $h(i) = i \bmod 5$. Muéstrese la tabla de dispersión cerrada con resolución lineal de colisiones que resulta de insertar la sucesión 23, 48, 35, 4, 10, en una tabla inicialmente vacía.
- 4.9 Obténganse las operaciones del TDA correspondencia, con tablas de dispersión abiertas y cerradas.

- 4.10** Para mejorar la velocidad de las operaciones podría desearse reemplazar una tabla de dispersión abierta con B_1 cubetas con más de B_1 elementos por otra tabla de dispersión con B_2 cubetas. Escribase un procedimiento para construir la nueva tabla a partir de la anterior, usando las operaciones del TDA lista para procesar cada cubeta.
- 4.11** En la sección 4.8 se habló de las funciones de dispersión «aleatorias» para las cuales $h_i(X)$, la cubeta en la que se va a probar después de i colisiones, es $(h(x) + d_i) \bmod B$ para cierta sucesión d_1, d_2, \dots, d_{B-1} . También se sugirió que una manera de calcular una sucesión semejante apropiada era elegir una constante k , y un $d_1 > 0$ arbitrario, y hacer

$$d_i = \begin{cases} 2d_{i-1} & \text{si } 2d_{i-1} < B \\ (2d_{i-1} - B) \oplus k & \text{si } 2d_{i-1} \geq B \end{cases}$$

donde $i > 1$, B es una potencia de 2, y \oplus representa la suma módulo 2 bit a bit. Si $B = 16$, encuéntrense los valores de k para los cuales la sucesión d_1, d_2, \dots, d_{15} incluye todos los enteros entre 1 y 15.

- 4.12** a) Muéstrese el árbol parcialmente ordenado que resulta cuando los enteros 5, 6, 4, 9, 3, 1, 7 se insertan en un árbol vacío.
 b) ¿Cuál es el resultado de tres operaciones SUPRIME-MIN sucesivas en el árbol de a)?
- 4.13** Supóngase que se representa el conjunto de cursos mediante
 a) una lista enlazada
 b) una tabla de dispersión
 c) un árbol binario de búsqueda.

Modifíquense las declaraciones de la figura 4.26 para cada una de estas estructuras.

- 4.14** Modifíquese la estructura de datos de la figura 4.26 de modo que cada registro de inscripción tenga un apuntador directo al estudiante y curso propietarios. Escribase otra vez el procedimiento *imprime_estudiantes* de la figura 4.26, aprovechando esta estructura.
- 4.15** Supóngase que hay 20 000 estudiantes, 1000 cursos y cada estudiante en un promedio de tres cursos; compárese la estructura de datos de la figura 4.26 con la modificación sugerida en el ejercicio 4.14 en lo referente a
 a) la cantidad de espacio requerida
 b) el tiempo promedio de ejecución de *imprime_estudiantes*
 c) el tiempo promedio de ejecución del procedimiento análogo que imprima los cursos que toma un estudiante dado.
- 4.16** Considérese la estructura de datos de la figura 4.26, dado un registro de curso c y un registro de estudiante e y escribanse procedimientos para insertar y suprimir el hecho de que e toma c .

- 4.17 Si existe, ¿cuál es la diferencia entre la estructura de datos del ejercicio 4.14 y la estructura en la que los conjuntos C_e y E_e se representan mediante listas de apuntadores a los registros de cursos y estudiantes, respectivamente?
- 4.18 Los empleados de cierta compañía se representan en la base de datos de la compañía por su nombre (que se supone único), número de empleado y número de seguridad social. Sugírase una estructura de datos que permita, dada una representación de un empleado, encontrar las otras dos representaciones del mismo individuo. ¿Qué rápida, en promedio, puede lograrse que sea cada una de esas operaciones?

Notas bibliográficas

Knuth [1973] es una buena fuente de información adicional sobre dispersión. La dispersión se desarrolló en la segunda mitad de la década de 1950, y Peterson [1957] es de los primeros artículos fundamentales sobre el tema. Morris [1968] y Maurer y Lewis [1975] dan buena información sobre la dispersión.

La lista múltiple es la estructura de datos central de los sistemas de bases de datos basados en redes propuestos en DBTG [1971]. Ullman [1982] proporciona información adicional sobre las aplicaciones, en bases de datos, de las estructuras de ese tipo.

La aplicación mediante montículos de los árboles parcialmente ordenados se basa en una idea de Williams [1964]. Las colas de prioridad se estudian más a fondo en Knuth [1973].

Reingold [1972] analiza la complejidad computacional de las operaciones básicas con conjuntos. Las técnicas de análisis de flujo de datos basadas en conjuntos se tratan con detalle en Cocke y Allen [1976] y en Aho y Ullman [1977].