Estructuras de datos

Tercera edición

Osvaldo Cairó
Silvia Guardati

Mc Graw Hill

ESTRUCTURAS DE DATOS

ESTRUCTURAS DE DATOS

Tercera edición

DR. OSVALDO CAIRÓ

Profesor-Investigador del
Instituto Tecnológico Autónomo de México (ITAM)
Miembro del Sistema Nacional
de Investigadores (SNI) Nivel 1

M.C. SILVIA GUARDATI

Profesor Numerario del
Instituto Tecnológico Autónomo de México (ITAM)

MÉXICO • AUCKLAND • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA
LISBOA • LONDRES • MADRID • MILÁN • MONTREAL • NUEVA DELHI • NUEVA YORK
SAN FRANCISCO • SAN JUAN • SAN LUIS • SANTIAGO
SÃO PAULO • SIDNEY • SINGAPUR • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Director editorial: Ricardo A. del Bosque Alayón
Editor sponsor: Pablo E. Roig Vázquez
Editora de desarrollo: Diana Karen Montaño González
Supervisor de producción: Zeferino García García

ESTRUCTURAS DE DATOS
Tercera edición

McGraw-Hill Interamericana

Prolongación Paseo de la Reforma 1015, Torre A,
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN 970-10-5908-5
(ISBN 970-10-3534-8 segunda edición)
(ISBN 970-10-0258-X primera edición)

1234567890 9875432106

Impreso en México *Printed in Mexico*

A nuestro hijo Facundo

CONTENIDO

PRESENTACIÓN

OBJETIVO

Este libro tiene como objetivo presentar las **estructuras de datos**, así como los algoritmos necesarios para tratarlas. El lenguaje utilizado es algorítmico, escrito en **seudo código**, independiente de cualquier lenguaje comercial de programación. Esta característica es muy importante, ya que permite al lector comprender las estructuras de datos y los algoritmos asociados a ellas sin relacionarlos con lenguajes de programación particulares. Se considera que una vez que el lector domine estos conceptos, los podrá implementar fácilmente en cualquier lenguaje.

Si bien cada uno de los temas son desarrollados desde niveles básicos a niveles complejos, se supone que el lector ya conoce ciertos conceptos, por ejemplo el de datos simples —enteros, reales, booleanos, carácter—; el de instrucción —declarativa, asignación, entrada/salida—, y el de operadores —aritméticos, relacionales y lógicos—. Asimismo se utiliza, pero no se explica, el concepto de variables y constantes. En los algoritmos se escriben los nombres de variables con mayúsculas —SUMA, N, etc.—, lo mismo para las constantes booleanas —VERDADERO y FALSO—.

Cabe aclarar que en este libro no se abordan los tipos abstractos de datos de manera explícita. Sin embargo, se tratan algunos de ellos sin presentarlos como tales; por ejemplo, las pilas y colas en el capítulo tres.

Cada capítulo cuenta con un número importante de ejercicios. Con éstos se sigue el mismo criterio aplicado en el desarrollo de los distintos temas, es decir, se proponen ejercicios en los que se aumenta gradualmente el nivel de complejidad.

LENGUAJE UTILIZADO

El lenguaje utilizado para describir los algoritmos es estructurado. Las estructuras algorítmicas selectivas y repetitivas se enumeran y las instrucciones que forman parte de ellas se escriben dejando sangrías para proporcionar mayor claridad. Además, con el objeto de ayudar al entendimiento de los mismos, se escriben comentarios encerrados entre {}. A continuación se presentan las estructuras algorítmicas empleadas en los algoritmos:

Selectiva simple

pi *Si* (condición) *entonces*
acción
p(i + 1) {Fin del condicional del paso pi}

Donde *condición* es cualquier expresión relacional y/o lógica, y *acción* es cualquier operación o conjunto de operaciones —lectura, escritura, asignación u otras—.

Esta estructura permite seleccionar una alternativa dependiendo de que la condición sea verdadera. Es decir, al ser evaluada la condición, si ésta resulta con un valor igual a **VERDADERO**, entonces se ejecutará la acción indicada. En caso contrario se sigue con el flujo establecido.

Selectiva doble

pi *Si* (condición)
entonces
acción$_1$
sino
acción$_2$
p(i + 1) {Fin del condicional del paso pi}

Donde *condición* es una expresión relacional y/o lógica, y *acción*$_1$ y *acción*$_2$ son cualquier operación o conjunto de operaciones —lectura, escritura, asignación u otras—.

Esta estructura permite seleccionar una de dos alternativas, según la condición sea verdadera o falsa. Si la condición es verdadera se ejecutará la acción$_1$, en caso contrario se ejecutará la acción$_2$.

Selectiva múltiple

pi *Si* (variable)
= valor$_1$: acción$_1$
= valor$_2$: acción$_2$
.
.
.
= valor$_n$: acción$_n$
p(i + 1) {Fin del condicional del paso pi}

Donde *valor*$_j$, $1 \leq j \leq n$, son los posibles valores que puede tomar la variable; *acción*$_j$, $1 \leq j \leq n$, es cualquier operación o conjunto de operaciones —lectura, escritura, asignación u otras—.

Este tipo de estructura se utiliza para una selección sobre múltiples alternativas. Según el valor de la variable se ejecutará la acción correspondiente. De esta manera, si variable es igual a valor$_1$ se ejecutará la acción$_1$, si variable es igual a valor$_2$ se ejecutará la acción$_2$, y así en los demás casos.

Repetitiva condicionada

```
        pi  Mientras (condición) Repetir
                acción
    p(i + 1)  {Fin del ciclo del paso pi}
```

Donde *condición* es cualquier expresión relacional y/o lógica, y *acción* es cualquier operación o conjunto de operaciones —lectura, escritura, asignación u otras—.

Esta estructura permite repetir una o más operaciones mientras la condición sea verdadera.

Repetitiva predefinida

```
        pi  Repetir con variable desde VI hasta VF
                acción
    p(i + 1)  {Fin del ciclo del paso pi}
```

Donde *variable* es cualquier variable, VI es un valor inicial que se le asigna a *variable*, VF es el valor final que va a tomar *variable* y *acción* es cualquier operación o conjunto de operaciones —lectura, escritura, asignación u otras—. Se asume que el valor de la variable se incrementa de uno en uno.

Esta estructura permite repetir una o más operaciones un número fijo de veces. El número de repeticiones queda determinado por la diferencia entre VF y VI más uno.

ORGANIZACIÓN

El libro está organizado en nueve capítulos, cada uno de ellos cuenta con numerosos ejemplos y ejercicios que ilustran y ayudan a entender los conceptos vertidos en ellos. Se utilizan tablas con seguimientos de los algoritmos para presentar cómo funcionan y de qué manera afectan a las estructuras de datos involucradas.

Algunos lectores quizá sepan que esta obra tiene dos ediciones anteriores, publicada por primera vez por la misma casa editorial en 1993, con múltiples reimpresiones. Trece años es un tiempo extenso en computación, un área donde los cambios se presentan velozmente. Esta edición ofrece una cuidadosa revisión de los temas tratados, algoritmos mejorados y ejercicios adicionales, en fin, muchos cambios para alcanzar el objetivo propuesto de esta nueva edición. Además, en los capítulos 1, 3, 5, 6 y 7 se incluyó una breve introducción a la programación orientada a objetos, presentando a las estructuras de datos —objetos de estudio en dichos capítulos— con este enfoque.

El lenguaje utilizado en los programas es **pseudocódigo**, es decir, independiente de cualquier otro lenguaje de programación comercial. Esta característica permite al estudiante concentrarse en las estructuras de datos y en los algoritmos asociados a ellas sin tener que atender los detalles de implementación. Una vez que domine los conceptos, los podrá llevar a la práctica con la ayuda de cualquier lenguaje de programación comercial. La generalidad con la que se explican los conceptos y posibles aplicaciones de los mismos facilitan, incluso, la implementación en lenguajes estructurados o en lenguajes orientados a objetos.

Capítulo 1: Estructuras fundamentales

En este capítulo se presentan las estructuras fundamentales de datos. Se estudian los arreglos unidimensionales, bidimensionales y multidimensionales. Además, se explican los registros. Por último, se incluye una breve introducción a la programación orientada a objetos con el fin de que sirva como base para entender las principales estructuras de datos desde este enfoque. También se describe la clase arreglo.

Capítulo 2: Arreglos multidimensionales representados en arreglos unidimensionales

La mayoría de los lenguajes de programación de alto nivel proporcionan medios eficaces para almacenar y recuperar elementos de arreglos bidimensionales y multidimensionales. Por ello, el usuario no se preocupa por los detalles del almacenamiento y el tratamiento físico del dato, sino por el tratamiento lógico del mismo. Esto representa una ventaja. Sin embargo, si las estructuras son muy grandes y no todos los campos están llenos, se presenta entonces una desventaja: gran desperdicio de espacio. Puede ocurrir también que el usuario necesite representar dichas estructuras de forma lineal. Por esta razón, en este capítulo se estudiará la representación lineal de arreglos bidimensionales y multidimensionales. Se analizarán, además, las matrices poco densas, las triangulares y tridiagonales, las simétricas y antisimétricas.

Capítulo 3: Pilas y colas

Este capítulo se dedicará a las pilas y colas, las cuales son estructuras de datos lineales, estáticas o dinámicas —dependiendo de si éstas se implementan con arreglos o listas—. Tales estructuras de datos tienen la particularidad de que la inserción y eliminación de los elementos se hace solamente por alguno de los extremos según su estructura. También se presentan estas estructuras con un enfoque orientado a objetos.

Capítulo 4: Recursión

La recursión permite definir un objeto en términos de sí mismo. Aparece en numerosas actividades de la vida diaria; por ejemplo, en la fotografía de una fotografía. Casos típicos de estructuras de datos definidas de manera recursiva son las listas y los árboles, que se estudiarán en los dos siguientes capítulos. La recursividad es una propiedad esencial en el desarrollo de software; por esta razón, se analizan aquí la descripción de la recursividad, así como el uso de algoritmos recursivos clásicos y complejos.

Capítulo 5: Listas

Las listas son estructuras lineales y dinámicas de datos. La principal ventaja del dinamismo lo representa el hecho de que se adquieren posiciones de memoria a medida que se necesitan y se liberan cuando ya no se requieren. Es decir, se llegan a expandir o contraer, dependiendo de la aplicación. El dinamismo de estas estructuras soluciona el problema de decidir cuánto espacio se necesita *a priori*, por ejemplo, en una estructura de datos estática como el arreglo. En este capítulo estudiaremos las listas lineales, circulares y doblemente ligadas. También se presentan estas estructuras con un enfoque orientado a objetos.

Capítulo 6: Árboles

Los árboles representan las estructuras de datos no-lineales y las dinámicas más relevantes en computación. No lineales, puesto que a cada elemento del árbol pueden seguirle varios elementos. Dinámicas, dado que la estructura del árbol suele cambiar durante la ejecución del programa. Los árboles balanceados son la estructura de datos más importante para trabajar en la memoria interna de la computadora. Por otra parte, los árboles-B^+ constituyen la estructura de datos más útil para trabajar con almacenamiento secundario. También se presenta esta estructura con un enfoque orientado a objetos.

Capítulo 7: Gráficas

Este capítulo se dedica a las estructuras de datos que permiten representar diferentes tipos de relaciones entre los objetos: *las gráficas*. Estudiaremos las gráficas dirigidas y no dirigidas, los conceptos más importantes y los algoritmos más destacados para trabajar con ellas, tales como Dijkstra, Floyd, Warshall, Prim y Kruskal. Además, se incluye una introducción a la solución de problemas —tema muy relacionado con las gráficas— y se estudian los algoritmos Breadth-First y Depth-First. También se presenta esta estructura con un enfoque orientado a objetos.

Capítulo 8: Métodos de ordenación

Ordenar significa colocar o reorganizar un conjunto de datos u objetos en una secuencia específica. Los procesos, tanto de ordenación como de búsqueda, son frecuentes en nuestra vida. En este capítulo estudiaremos los métodos de ordenación interna y externa más importantes de la actualidad. Se presenta, además, el análisis de eficiencia de cada uno de los métodos.

Capítulo 9: Métodos de búsqueda

Este capítulo se dedicó a una de las operaciones más importantes en el procesamiento de la información: *la búsqueda.* Tal operación permite recuperar datos almacenados. La

búsqueda puede ser interna, cuando todos los elementos se encuentran en la memoria principal, o externa, cuando están en la memoria secundaria. Se estudian los métodos de búsqueda más importantes que existen. Se presenta también el análisis de eficiencia de cada uno de estos métodos.

AGRADECIMIENTOS

Esta obra es fruto de la colaboración de amigos, estudiantes y colegas que, de alguna u otra forma, participaron para que este proyecto sea una realidad. Especialmente queremos agradecer al doctor Arturo Fernández Pérez, rector del ITAM, y a los funcionarios de la División Académica de Ingeniería del ITAM, quienes nos apoyaron para la realización de este libro.

OSVALDO CAIRÓ
SILVIA GUARDATI

Capítulo 1

ESTRUCTURAS FUNDAMENTALES DE DATOS

1.1 INTRODUCCIÓN

La importancia de las computadoras radica fundamentalmente en su capacidad para procesar información. Esta característica les permite realizar actividades que antes sólo las realizaban los humanos.

Con el propósito de que la información sea procesada, se requiere que ésta se almacene en la memoria de la computadora. De acuerdo con la forma en que los datos se organizan, se clasifican en:

- Tipos de datos simples.
- Tipos de datos estructurados.

La principal característica de los tipos de datos simples consiste en que ocupan sólo una casilla de memoria (fig. 1.1*a*); por tanto, una variable simple hace referencia a un único valor a la vez. En este grupo de datos se encuentran: números enteros y reales, caracteres, booleanos, enumerados y subrangos. Cabe señalar que los dos últimos no existen en algunos lenguajes de programación.

Por otra parte, los tipos de datos estructurados se caracterizan por el hecho de que con un nombre —identificador de variable estructurada— se hace referencia a un grupo de casillas de memoria (fig. 1.1*b*). Es decir, un tipo de dato estructurado tiene varios componentes. Cada uno de éstos puede ser un tipo de dato simple o estructurado. Sin embargo, los componentes básicos, los del nivel más bajo, de cualquier tipo de datos estructurado son siempre tipos de datos simples.

El estudio de las estructuras de datos constituye una de las principales actividades para llegar al desarrollo de grandes sistemas de software. En este capítulo se tratarán las estructuras de datos básicos que son útiles para la mayoría de los lenguajes de programación. Éstas son: arreglos y registros.

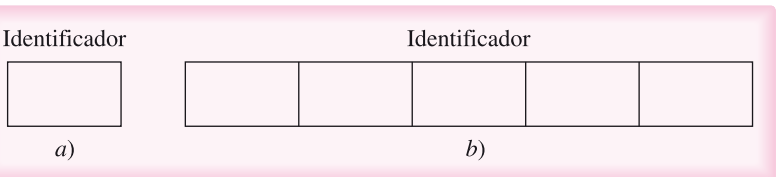


FIGURA 1.1
Tipos de datos simples y estructurados.
a) Dato simple.
b) Dato estructurado.

1.2 ARREGLOS

Con frecuencia se presentan en la práctica problemas cuya solución no resulta fácil —a veces es imposible— si se utilizan tipos de datos simples.

Con el propósito de ilustrar esta dificultad, a continuación se presentarán un problema y dos de sus posibles soluciones mediante tipos simples de datos. El objetivo de este ejemplo es demostrar lo complejo que resulta un algoritmo de solución para ciertos problemas, si no se utilizan tipos de datos estructurados. Finalmente, y luego de presentar los arreglos, se ofrecerá una solución al problema mencionado en primer término usando arreglos.

Ejemplo 1.1

Consideremos que en una universidad se conocen las calificaciones de un grupo de 50 alumnos. Se necesita saber cuántos de éstos tienen calificación superior al promedio del grupo.

¿Cómo resolver este problema?

Primera solución
Algoritmo 1.1 Doble_lectura

Doble_lectura

{Este algoritmo resuelve el problema planteado en el ejemplo 1.1, realizando dos veces la lectura de los datos}

{*I* y CONT son variables de tipo entero. *AC*, PROM y *C* son variables de tipo real}

1. Hacer $AC \leftarrow 0$ e $I \leftarrow 1$
2. Mientras ($I \leq 50$) *Repetir*
    Escribir "Ingrese la calificación", *I*
    Leer *C*
    Hacer $AC \leftarrow AC + C$ e $I \leftarrow I + 1$
3. {Fin del ciclo del paso 2}
4. Hacer PROM $\leftarrow AC/50$

{Como se necesita indicar cuántos alumnos obtuvieron calificación superior al promedio, se releerán las 50 calificaciones para comparar cada una de ellas con el promedio calculado en el paso 4}

```
   Hacer CONT ← 0 e I ← 1
5. Mientras (I ≤ 50) Repetir
        Escribir "Ingrese la calificación", I
        Leer C
   5.1 Si C > PROM entonces
           Hacer CONT ← CONT + 1
   5.2 {Fin del condicional del paso 5.1}
           Hacer I ← I + 1
6. {Fin del ciclo del paso 5}
7. Escribir CONT
```

Segunda solución

Algoritmo 1.2 Muchas_variables

Muchas_variables

{Este algoritmo resuelve el problema planteado en el ejemplo 1.1, pero ahora mediante muchas variables}

{CONT es una variable de tipo entero. PROM, AC y C_i son variables de tipo real}

```
1.  Leer C1, C2, C3, ..., C50
    {Las calificaciones corresponden a los 50 alumnos}

2.  Hacer AC ← C1 + C2 + C3 + ... + C50,
    PROM ← AC/50 y CONT ← 0

3.  Si C1 > PROM entonces
        Hacer CONT ← CONT + 1

4.  {Fin del condicional del paso 3}
5.  Si C2 > PROM entonces
        Hacer CONT ← CONT + 1

6.  {Fin del condicional del paso 5}
    ...
100. Si C50 > PROM entonces
         Hacer CONT ← CONT + 1

101. {Fin del condicional del paso 100}
102. Escribir CONT
```

Estas dos soluciones son muy representativas de los inconvenientes a los que uno se puede enfrentar, al plantear una solución algorítmica a un problema al usar sólo tipos de datos simples.

En la solución planteada en el algoritmo 1.1 el usuario debe ingresar dos veces el conjunto de datos. Esto último tiene varias desventajas: es totalmente molesto —considere que el número de datos puede ser mayor a 50—, ineficiente —la operación de lectura, ya sea de manera interactiva con el usuario o desde un archivo, se debe repetir, lo que ocasiona pérdida de tiempo— y causa de errores —en los casos donde la entrada de datos se haga de forma manual—.

Por otra parte, en la solución planteada en el algoritmo 1.2 se manejan 50 variables en memoria. Esta solución presenta el inconveniente de que el manejo de las variables se puede tornar incontrolable, sobre todo si su número crece en forma considerable. Además, algunos pasos especificados en el algoritmo, que posteriormente serán instrucciones de algún lenguaje de programación, se repiten, ya que no se pueden generalizar. Esta característica no sólo provoca más trabajo, sino también posibles errores. Es sabido que ejecutar una tarea en forma repetida, en este caso escribir un mismo paso varias veces, resta interés en la acción que se está llevando a cabo, y puede propiciar más errores.

Se observa, entonces, que ninguna de las dos soluciones resulta práctica ni eficiente. Es necesario un tipo de dato que permita manejar mucha información, generalizando sus operaciones. Los tipos de datos estructurados que ayudan a resolver problemas como éste son los arreglos.

Un **arreglo unidimensional** se define como una colección finita, homogénea y ordenada de elementos.

- **Finita:** todo arreglo tiene un límite; es decir, se debe determinar cuál será el número máximo de elementos que formarán parte del arreglo.
- **Homogénea:** todos los elementos de un arreglo son del mismo tipo. Es decir, todos enteros, todos booleanos, etcétera, pero nunca una combinación de distintos tipos.
- **Ordenada:** se puede determinar cuáles son el primero, el segundo, el tercero, ... y el enésimo elementos.

Un arreglo unidimensional se puede representar gráficamente como se muestra en la figura 1.2.

Si un arreglo tiene la característica de que puede almacenar a *N* elementos del mismo tipo, entonces deberá permitir la recuperación de cada uno de ellos. Como consecuencia, se distinguen dos partes fundamentales en los arreglos:

- Los componentes.
- Los índices.

Los primeros hacen referencia a los elementos que forman el arreglo; es decir, a los valores que se almacenan en cada una de sus casillas (fig. 1.3). Considerando el

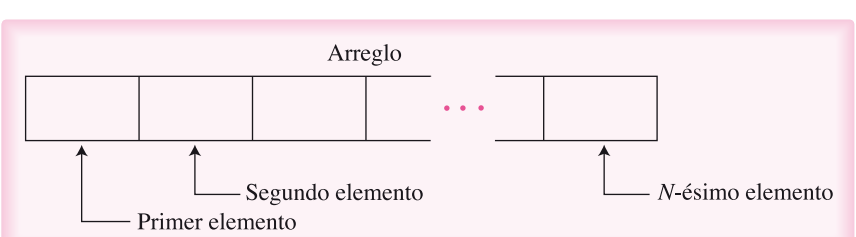


FIGURA 1.2
Representación de arreglos.

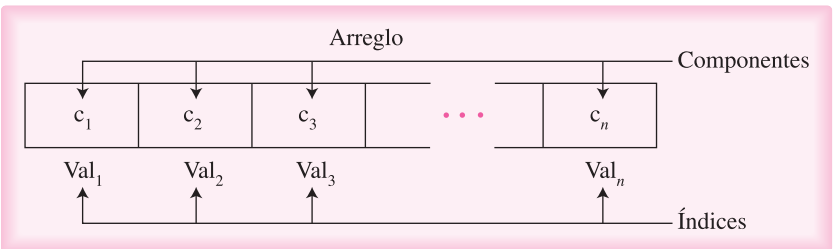


FIGURA 1.3
Índices y componentes de un arreglo.

ejemplo anterior, cada una de las 50 calificaciones será un componente de un arreglo "calificaciones". En este contexto, los índices especifican cuántos elementos tendrá el arreglo y además de qué modo podrán recuperarse esos componentes. Los índices también permiten hacer referencia a los componentes del arreglo en forma individual; es decir, distinguirán entre sus elementos. Por tanto, para hacer referencia a un elemento de un arreglo se debe utilizar:

- El nombre del arreglo.
- El índice del elemento.

En la figura 1.3 se representa un arreglo unidimensional y se indican tanto sus componentes como sus índices.

1.2.1 Declaración de arreglos unidimensionales

No es el propósito de este libro seguir la sintaxis de algún lenguaje de programación en particular; un arreglo unidimensional se define de la siguiente manera:

ident_arreglo = ARREGLO [líminf .. límsup] DE tipo

Con los valores **líminf** y **límsup** se declara el tipo de los índices, así como el número de elementos que tendrá el arreglo. El número total de componentes (NTC) que tendrá el arreglo unidimensional se calcula con

$$NTC = límsup - líminf + 1 \qquad \textbf{Fórmula 1.1}$$

Con **tipo** se declara el tipo de datos para todos los componentes del arreglo unidimensional. El tipo de los componentes no tiene que ser el mismo que el de los índices. En general, los lenguajes de programación establecen restricciones al respecto.

Observaciones:

a) El tipo del índice puede ser cualquier tipo ordinal: carácter, entero, enumerado. En la mayoría de los lenguajes usados actualmente se permite sólo números enteros.

b) El tipo de los componentes puede ser cualquier tipo de datos —entero, real, cadena de caracteres, registro, arreglo, etcétera—.
c) Se utilizan los corchetes "[]" para indicar el índice de un arreglo. Entre [] se debe escribir un valor ordinal; puede ser una variable, una constante o una expresión tan compleja como se quiera, pero que dé como resultado un valor ordinal.

Enseguida se verán algunos ejemplos de arreglos unidimensionales:

Ejemplo 1.2

Sea *V* un arreglo unidimensional de 50 elementos enteros con índices enteros. Su representación se indica en la figura 1.4.

V = ARREGLO[1..50] DE enteros

- NTC = (50 – 1 + 1) = 50.
- Cada componente del arreglo unidimensional V será un número entero, al cual se tendrá acceso por medio de un índice que será un valor comprendido entre 1 y 50.

Por ejemplo:

V[1] hace referencia al elemento de la posición 1.
V[2] hace referencia al elemento de la posición 2.
...
V[50] hace referencia al elemento de la posición 50.

Los índices de tipo entero no necesariamente deben tener un límite inferior igual a cero o a uno. Podrían usarse valores negativos [−10..10] o valores mayores a uno [100..200].

Ejemplo 1.3

Sea *A* un arreglo de 26 elementos booleanos con índices de tipo carácter. Su representación se muestra en la figura 1.5.

A = ARREGLO ['*a*'.. '*z*'] DE booleanos

- NTC = (ord('*z*') - ord('*a*') + 1) = 122 − 97 + 1 = 26.
- Cada componente del arreglo unidimensional *A* será uno de los dos posibles valores lógicos (VERDADERO o FALSO) al cual se tendrá acceso por medio de un índice, que será un valor comprendido entre los caracteres '*a*' y '*z*'.

Por ejemplo:

A['*a*'] hace referencia al elemento de la posición '*a*' (1era.)
A['*b*'] hace referencia al elemento de la posición '*b*' (2da.)

FIGURA 1.4

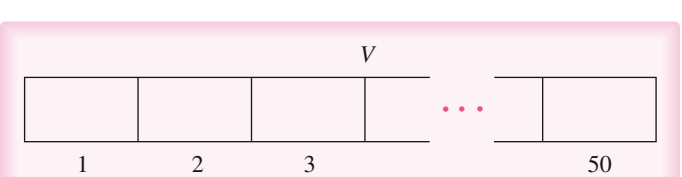
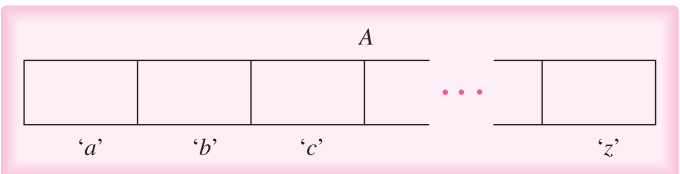


FIGURA 1.5



...
A['z'] hace referencia al elemento de la posición 'z' (26)

Ejemplo 1.4

Sea CICLO un arreglo de 12 elementos reales con índices de tipo escalar o enumerados. Su representación se muestra en la figura 1.6.

meses = (ene, feb, mar, abr, may, jun, jul, ago, sept, oct, nov, dic)

CICLO = ARREGLO [meses] DE reales

- NTC = (ord(dic) − ord(ene) + l) = 11 – 0 + 1 = 12.
- Cada componente del arreglo unidimensional CICLO será un número real, al cual se tendrá acceso por medio de un índice, que será un valor comprendido entre ene y dic.

Por ejemplo:

CICLO[ene] hace referencia al elemento de la posición ene (1era.)
CICLO[feb] hace referencia al elemento de la posición feb (2da.)
...
CICLO[dic] hace referencia al elemento de la posición dic (12ava.)

1.2.2 Operaciones con arreglos unidimensionales

Como ya se mencionó, los arreglos se utilizan para almacenar datos. Por tanto, resulta necesario leer, escribir, asignar o simplemente modificar datos en un arreglo. Asimismo, al considerar que es una estructura, a una colección de elementos se deben incorporar nuevos elementos, así como eliminar algunos de los ya almacenados. Las operaciones válidas en arreglos son las siguientes:

- Lectura/Escritura.
- Asignación.

FIGURA 1.6

- Actualización: Inserción.
  Eliminación.
  Modificación.
- Ordenación.
- Búsqueda.

Como los arreglos son tipos de datos estructurados, muchas de estas operaciones no se pueden llevar a cabo de manera global; es decir, tratando al arreglo como un todo, sino que se debe trabajar sobre cada componente.

A continuación se analizará cada una de estas operaciones. Cabe destacar que las dos últimas, ordenación y búsqueda, serán tema de estudio en próximos capítulos. Para ilustrarlas se utilizarán los ejemplos presentados anteriormente.

Lectura

El proceso de lectura de un arreglo consiste en leer y asignar un valor a cada uno de sus componentes. Suponga que se desea leer todos los elementos del arreglo unidimensional *V* en forma consecutiva. Se podría hacer de la siguiente manera:

Leer *V*[1],
Leer *V*[2],
...
Leer *V*[50]

Pero es importante que el lector observe que de esta forma no resulta práctico. Por tanto, se usará un ciclo para leer todos los elementos del arreglo unidimensional.

Repetir con *I* desde 1 hasta 50
  Leer *V*[*I*]

Al variar el valor de *I*, cada elemento leído se asigna al correspondiente componente del arreglo según la posición indicada por *I*.

Para *I* = 1, se lee *V*[1]
  I = 2, se lee *V*[2]
  ...
  I = *N*, se lee *V*[*N*]

Al finalizar el ciclo de lectura se tendrá asignado un valor a cada uno de los componentes del arreglo unidimensional *V*. El arreglo se muestra en la figura 1.7.

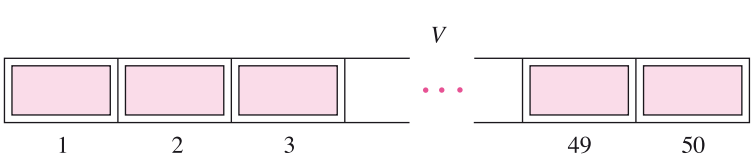


FIGURA 1.7
Lectura de arreglos.

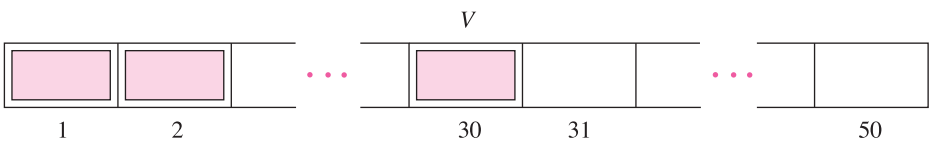


FIGURA 1.8
Lectura de arreglos.

Puede suceder que no se necesiten leer todos los componentes del arreglo, sino solamente alguno de ellos. Supongamos que se deben leer los elementos con índices comprendidos entre el 1 y el 30. A continuación se muestra el ciclo que se necesita para realizar esta operación:

Repetir con *I* desde 1 hasta 30
  Leer *V*[*I*]

El arreglo se muestra en la figura 1.8.

Escritura

El caso de la operación de escritura es similar al de lectura. Se debe escribir el valor de cada uno de los componentes. Supongamos que se desea escribir los primeros *N* componentes del arreglo unidimensional *V* en forma consecutiva. Los pasos a seguir son:

Repetir con *I* desde 1 hasta *N*
  Escribir *V*[*I*]

Al variar el valor de *I* se escribe el elemento del arreglo unidimensional *V*, correspondiente a la posición indicada por *I*.

Para *I* = 1, se escribe el valor de *V*[1]
  I = 2, se escribe el valor de *V*[2]
  ...
  I = *N*, se escribe el valor de *V*[*N*]

Asignación

En general, no es posible asignar directamente un valor a todo el arreglo, sino que se debe asignar el valor deseado a cada componente. Enseguida se analizan algunos ejemplos de asignación.

Observe que en los dos primeros casos se asigna un valor a una determinada casilla del arreglo, en el primero a la señalada por el índice ene, y en el segundo a la indicada por el índice mar.

CICLO[ene] ← 123.89
CICLO[mar] ← CICLO[ene]/2

En el tercer caso se asigna el 0 a todas las casillas del arreglo, con lo que éste queda como se muestra en la figura 1.9.

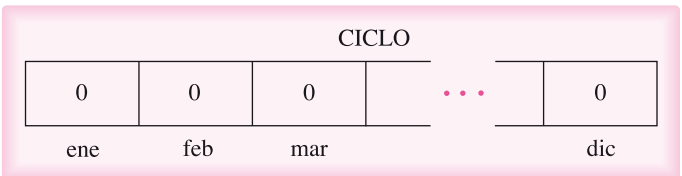


FIGURA 1.9
Asignación de arreglos.

Repetir con MES desde ene hasta dic
  Hacer CICLO[MES] ← 0

Cabe destacar que en algunos lenguajes de programación es posible asignar una variable tipo arreglo a otra del mismo tipo.

$V_1 \leftarrow V$

La expresión anterior es equivalente a realizar lo siguiente:

Repetir con *I* desde 1 hasta 50
  Hacer $V_1[I] \leftarrow V[I]$

Actualización

La actualización es una operación que se realiza en forma frecuente en los arreglos. La cantidad de actualizaciones está relacionada con el tipo de problema que se intente resolver. A diferencia de las otras operaciones estudiadas, la actualización lleva implícita otros tipos de operaciones, como inserción y eliminación de elementos.

Con el propósito de realizar una actualización de manera eficiente, es importante conocer si el arreglo está o no ordenado; es decir, si sus componentes respetan algún orden, ya sea creciente o decreciente. Cabe destacar que las operaciones de inserción, eliminación y modificación serán tratadas en forma separada para arreglos ordenados y desordenados.

Finalmente, es importante señalar que la operación de búsqueda se utiliza como auxiliar en las operaciones de inserción, eliminación y modificación. Esta es la principal razón por la cual a continuación se presenta el algoritmo de búsqueda secuencial en arreglos desordenados. En el capítulo correspondiente a métodos de búsqueda se tratará con mayor detalle este tema.

Algoritmo 1.3 Busca_secuencial_desordenado

Busca_secuencial_desordenado

{El algoritmo busca en forma secuencial un elemento en un arreglo unidimensional que se encuentra desordenado. *V* es un arreglo de 100 elementos, *N* el número actual de elementos y *X* el valor a buscar}
{*I* es una variable auxiliar de tipo entero}

1. Hacer *I* ← 1

2. Mientras ($I \leq N$) y ($X \neq V[I]$) *Repetir*

Hacer $I \leftarrow I + 1$

3. {Fin del ciclo del paso 2}
4. *Si* $I > N$ {No se encontró el valor buscado}

entonces
Escribir "El valor X no está en el arreglo"
si no Escribir "El valor X está en la posición I"

5. {Fin del condicional del paso 4}

Este método de búsqueda es sencillo, aunque no muy eficiente. Consiste en recorrer el arreglo, comparando cada elemento del mismo con el valor a buscar. El proceso se repite hasta que el valor se encuentre —éxito— o hasta que se haya superado el tamaño del arreglo —fracaso—.

a) Arreglos desordenados Considere un arreglo unidimensional V de 100 elementos, como el que se presenta en la figura 1.10. Observe que los primeros N componentes tienen asignado un valor.

a.1) Inserción: Para insertar un elemento Y en un arreglo unidimensional V desordenado, se debe verificar que exista espacio. Si se cumple esta condición, entonces se asignará en la posición $N + 1$ el nuevo elemento y se incrementará en N el total de elementos del arreglo.

A continuación se presenta el algoritmo de inserción en arreglos unidimensionales desordenados.

Algoritmo 1.4 Inserta_desordenado

Inserta_desordenado (V, N, Y)

{El algoritmo inserta un elemento en un arreglo unidimensional desordenado. V es un arreglo de máximo 100 elementos. N es el número actual de elementos. Y representa el valor a insertar}

1. *Si* $N < 100$

entonces
Hacer $N \leftarrow N + 1$ y $V[N] \leftarrow Y$
si no {No hay espacio en el arreglo}
Escribir "El valor Y no se puede insertar. No hay espacio"

2. {Fin del condicional del paso 1}

Luego de la inserción el arreglo unidimensional V queda como se muestra en la figura 1.10*a*.

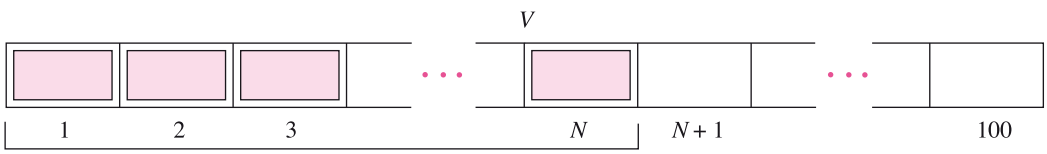


FIGURA 1.10
Actualización de arreglos desordenados.

a.2) Eliminación: Para eliminar un elemento X de un arreglo unidimensional V desordenado, se debe verificar que X se encuentre en el arreglo. Si se cumple esta condición, entonces se procederá a recorrer todos los elementos que están a su derecha una posición a la izquierda, disminuyendo en uno el número de componentes del arreglo.

A continuación se presenta el algoritmo de eliminación en arreglos desordenados. Cabe destacar que la operación de búsqueda presentada en el algoritmo 1.3 se usa para determinar si el elemento X se encuentra en el arreglo. Para el caso de que la respuesta sea positiva, se obtiene también la posición en que se encuentra. Con el propósito de ofrecer mayor claridad en la solución de este problema, se incluye dentro del algoritmo de eliminación el algoritmo de búsqueda secuencial en arreglos desordenados.

Algoritmo 1.5 Elimina_desordenado

Elimina_desordenado (*V*, *N*, *X*)

{El algoritmo elimina un elemento en un arreglo unidimensional desordenado. V es un arreglo de 100 elementos. N es el número actual de elementos. X es el valor a eliminar}
{I y K son variables de tipo entero}

1. Hacer $I \leftarrow 1$
2. Mientras $(I \leq N)$ y $(X \neq V[I])$ *Repetir*

Hacer $I \leftarrow I + 1$

3. {Fin del ciclo del paso 2}
4. *Si* $(I > N)$ {No se encontró el valor buscado}

entonces
Escribir "El valor X no se encuentra en el arreglo"
si no
1.1 *Repetir* con K desde I hasta $(N - 1)$
Hacer $V[K] \leftarrow V[K + 1]$
1.2 {Fin del ciclo del paso 4.1}
Hacer $N \leftarrow N - 1$

5. {Fin del condicional del paso 4}

Luego de la eliminación, el arreglo unidimensional V queda como se muestra en la figura 1.10*b*.

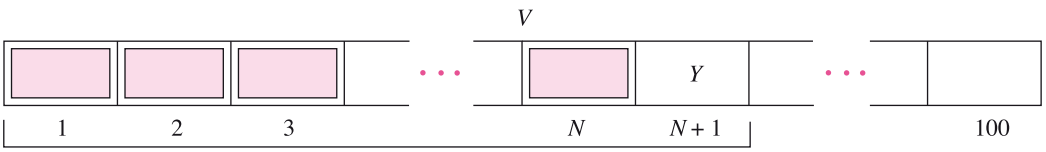


FIGURA 1.10*a*
Inserción en arreglos desordenados.

a.3) Modificación: Para modificar un elemento X de un arreglo unidimensional V desordenado se debe verificar que X se encuentre en el arreglo. Si se cumple esta condición, entonces se procederá a su actualización.

A continuación se presenta el algoritmo de modificación en arreglos desordenados, en el cual se incluye la búsqueda secuencial.

Algoritmo 1.6 Modifica_desordenado

Modifica_desordenado (*V*, *N*, *X*, *Y*)

{El algoritmo modifica un elemento de un arreglo unidimensional desordenado. V es un arreglo de máximo 100 elementos. N es el número actual de elementos. X es el elemento a modificar por el elemento Y}
{I es una variable de tipo entero}

1. Hacer $I \leftarrow 1$
2. Mientras $(I \leq N)$ y $(X \neq V[I])$ *Repetir*

Hacer $I \leftarrow I + 1$

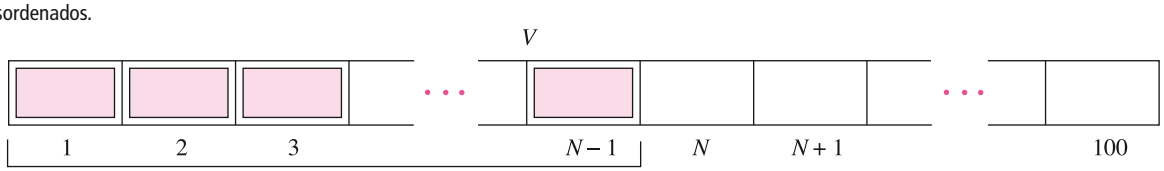
3. {Fin del ciclo del paso 2}
4. *Si* $(I > N)$ {No se encontró el valor buscado}

entonces
Escribir "El valor X no se encuentra en el arreglo"
si no
Hacer $V[I] \leftarrow Y$

5. {Fin del condicional del paso 4}

Luego de la modificación, el arreglo unidimensional V queda como se muestra en la figura 1.10*c*.

FIGURA 1.10*b*
Eliminación en arreglos desordenados.



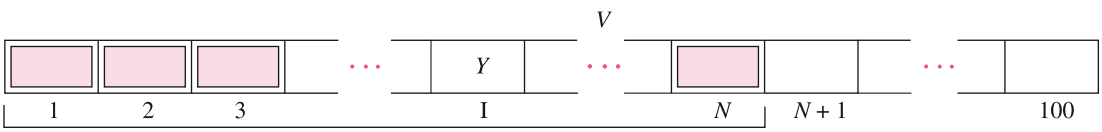


FIGURA 1.10*c*
Modificación en arreglos desordenados.

***b)* Arreglos ordenados** Considere el arreglo unidimensional ordenado V de 100 elementos de la figura 1.11. Los primeros N componentes del mismo tienen asignado un valor. En este caso se trabajará con un arreglo ordenado de manera creciente, es decir:

$$V[1] \leq V[2] \leq V[3] \leq \ldots \leq V[N]$$

Cuando se trabaja con arreglos ordenados se debe evitar alterar el orden al insertar nuevos elementos o al modificar los existentes.

*b.*1) Inserción: Para insertar un elemento X en un arreglo unidimensional V ordenado, primero se debe verificar que exista espacio. Luego se encontrará la posición en la que debería estar el nuevo valor para no alterar el orden del arreglo. Cuando se detecte la posición, se procederá a recorrer todos los elementos desde ahí hasta la N-ésima posición, un lugar a la derecha. Finalmente se asignará el valor de X en la posición encontrada. Cabe destacar que el desplazamiento no se lleva a cabo cuando el valor a insertar es mayor que el último elemento del arreglo.

Generalmente, cuando se quiere hacer una inserción se debe verificar que el elemento no se encuentre en el arreglo. En la mayoría de los casos prácticos no interesa tener información duplicada; por tanto, si el valor que se desea insertar ya estuviera en el arreglo, la operación no se llevará a cabo.

Antes de presentar el algoritmo de inserción, se definirá una función de búsqueda auxiliar, para arreglos ordenados, que se utilizará tanto en el proceso de inserción como en el de eliminación. Esta función es una variante de la presentada en el algoritmo 1.3, y da como resultado la posición en la que encontró al elemento X o el negativo de la posición en la que debería estar. Para mayor información sobre algoritmos de búsqueda, consulte el capítulo 9.

Algoritmo 1.7 Busca_secuencial_ordenado

Busca_secuencial_ordenado (V, N, X, POS)

{El algoritmo busca un elemento X en un arreglo unidimensional V de N elementos que se encuentra ordenado crecientemente. POS indica la posición de X en V o la posición en la que estaría X}
{I es una variable de tipo entero}

1. Hacer $I \leftarrow 1$
2. Mientras ($I \leq N$) y ($V[I] < X$) *Repetir*

Hacer $I \leftarrow I + 1$

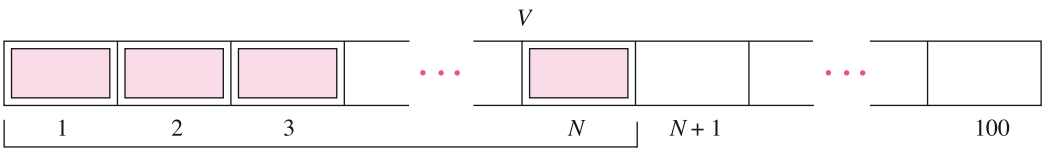


FIGURA 1.11
Actualización de arreglos ordenados.

```
3. {Fin del ciclo del paso 2}
4. Si ((I > N) o (V[I] > X))
        entonces
            Hacer POS ← −I
        si no
            Hacer POS ← I
5. {Fin del condicional del paso 4}
```

A continuación se presenta el algoritmo de inserción en un arreglo unidimensional que se encuentra ordenado en forma creciente.

Algoritmo 1.8 Inserta_ordenado

Inserta_ordenado (*V*, *N*, *Y*)

```
{Este algoritmo inserta un elemento Y en un arreglo unidimensional que se encuentra ordenado
de forma creciente. La capacidad máxima del arreglo es de 100 elementos. N indica el número
actual de elementos de V}
{POS e I son variables de tipo entero}

1. Si (N < 100)
        entonces
            Llamar al algoritmo Busca_secuencial_ordenado con V, N, Y y POS
   1.1 Si POS > 0 {El elemento fue encontrado en el arreglo}
            entonces
                Escribir "El elemento ya existe"
            si no
            Hacer N ← N + 1 y POS ← POS * (−1)
       1.1.1 Repetir con I desde N hasta POS + 1
            Hacer V[I] ← V[I − 1]
       1.1.2 {Fin del ciclo del paso 1.1.1}
            Hacer V[POS] ← Y
   1.2 {Fin del condicional del paso 1.1}
       si no
                Escribir "No hay espacio en el arreglo"
2. {Fin del condicional del paso 1}
```

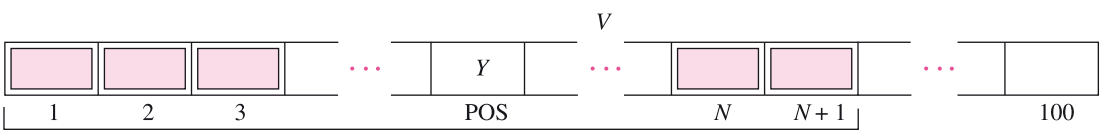


FIGURA 1.11*a*
Inserción en arreglos ordenados.

Luego de la inserción, el arreglo queda como se muestra en la figura 1.11*a*.

b.2) Eliminación: Para eliminar un elemento *X* de un arreglo unidimensional ordenado *V* se debe buscar la posición del elemento a eliminar. Si el resultado de la función es un valor positivo, significa que el elemento se encuentra en el arreglo y, por tanto, se puede eliminar; en caso contrario, no se puede realizar la operación de eliminación.

A continuación se presenta el algoritmo de eliminación en arreglos ordenados.

Algoritmo 1.9 Elimina_ordenado

Elimina_ordenado (*V*, *N*, *X*)

{El algoritmo elimina un elemento *X* de un arreglo unidimensional *V* de *N* elementos que se encuentra ordenado en forma creciente}

{POS e *I* son variables de tipo entero}

```
1. Si (N > 0)
        entonces
        Llamar al algoritmo Busca_secuencial_ordenado con V, N, X y POS
   1.1  Si (POS < 0) {No se puede eliminar porque X no existe}
                entonces
                    Escribir "El elemento no existe"
                si no
                Hacer N ← N – 1
        1.1.1  Repetir con I desde POS hasta N
                        Hacer V[I] ← V[I + 1]
        1.1.2  {Fin del ciclo del paso 1.1.1}
   1.2  {Fin del condicional del paso 1.1}
        si no
                        Escribir "El arreglo está vacío"
2. {Fin del condicional del paso 1}
```

Luego de la eliminación, el arreglo queda como se muestra en la figura 1.11*b*.

b.3) Modificación: Esta operación consiste en reemplazar un componente del arreglo con otro valor. Para ello, primero se buscará el elemento en el arreglo. Si se encuentra, antes de realizar el cambio se debe verificar que el orden del arreglo no se altere. Si esto llegara a suceder, entonces es necesario realizar dos opera-

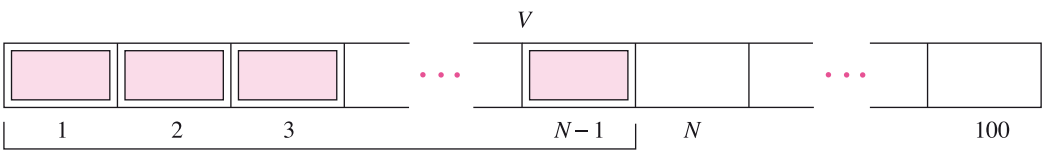


FIGURA 1.11*b*
Eliminación en arreglos ordenados.

ciones; primero se debe eliminar el elemento que se quiere modificar y luego insertar en la posición correspondiente el nuevo valor. Como consecuencia de que las operaciones que se necesitan para realizar una modificación ya han sido presentadas, se deja como tarea la construcción del algoritmo de modificación en arreglos ordenados.

Hasta el momento se ha analizado cómo declarar arreglos y cómo usarlos. Ahora se puede dar solución al problema del ejemplo 1.1 mediante este tipo de estructura de datos.

Algoritmo 1.10 Con_arreglos

Con_arreglos (CAL)

{Este algoritmo resuelve el problema del ejemplo 1.1 al aplicar arreglos unidimensionales. CAL es un arreglo de 50 elementos de números reales}
{*AC*, *I* y CONT son variables de tipo entero. PROM es una variable de tipo real}

1. Hacer *AC* ← 0
2. *Repetir* con *I* desde 1 hasta 50
  Leer CAL[*I*]
  Hacer *AC* ← *AC* + CAL[*I*] e I ← *I* + 1
3. {Fin del ciclo del paso 2}
4. Hacer PROM ← *AC*/50 y CONT ← 0
5. *Repetir* con *I* desde 1 hasta 50
  5.1 *Si* (CAL[*I*] > PROM) *entonces*
    Hacer CONT ← CONT + 1
  5.2 {Fin del condicional del paso 5.1}
6. {Fin del ciclo del paso 5}
7. Escribir CONT

Ésta es una solución más eficiente que las que se presentaron en los algoritmos 1.1 y 1.2. Se realiza una lectura de los datos y además se define una variable para almacenar las 50 calificaciones.

Al utilizar un arreglo puede disponerse de los datos tantas veces como sea necesario sin que se deba volver a leerlos, ya que éstos permanecen en memoria. Además se facilita el procesamiento de los datos, al generalizar ciertas operaciones.

Los arreglos presentados hasta el momento se denominan **arreglos unidimensionales** o **lineales**, debido a que cualquier elemento se referencia solamente con un índice.

Sin embargo, es importante destacar que para la mayoría de los lenguajes de programación se pueden definir arreglos multidimensionales; es decir, arreglos con múltiples índices. El número de dimensiones —índices— depende tanto del problema que se quiera resolver como del lenguaje utilizado.

Se analizarán primero los arreglos bidimensionales, que representan un caso especial de los multidimensionales, por ser los más ampliamente utilizados.

1.3 ARREGLOS BIDIMENSIONALES

Para que el lector entienda mejor la estructura de los arreglos bidimensionales, se presenta el siguiente ejemplo.

Ejemplo 1.5

La tabla 1.1 contiene los costos de producción de cada departamento de una fábrica, correspondientes a los 12 meses del año anterior.

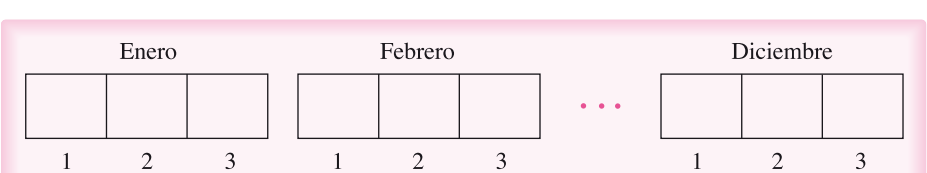
La tabla se interpreta de la siguiente manera: dado un mes, se conocen los costos de producción de cada uno de los departamentos de la fábrica; y dado un departamento, se conocen los costos de producción mensuales. Si se quisiera almacenar esta información utilizando los arreglos unidimensionales, se tendrían dos alternativas:

1. Definir 12 arreglos de tres elementos cada uno. En este caso, cada arreglo almacenará la información relativa a un mes.

TABLA 1.1
Costos mensuales por departamentos

Meses/Deptos.	Dulces	Conservas	Bebidas
Enero	100	300	120
Febrero	400	200	200
Marzo	350	250	210
Abril	280	300	200
Mayo	300	320	300
Junio	250	300	350
Julio	200	280	300
Agosto	180	300	400
Septiembre	500	400	450
Octubre	350	420	220
Noviembre	400	450	360
Diciembre	600	550	531

FIGURA 1.12
Almacenamiento de la información por mes.



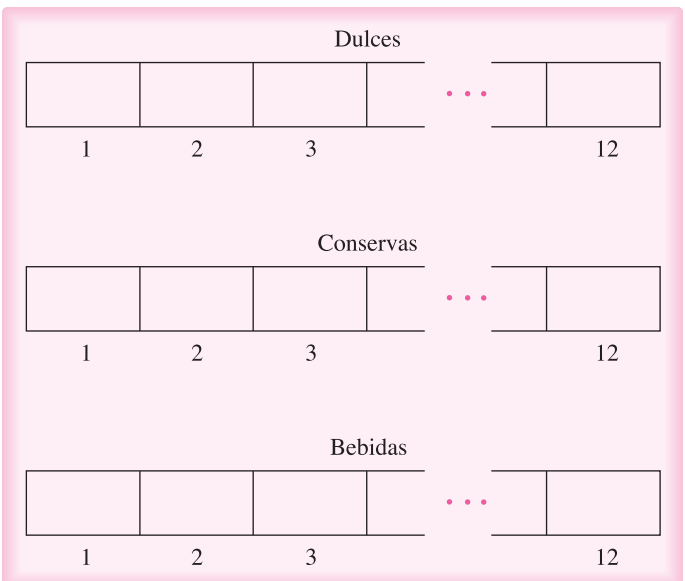


FIGURA 1.13
Almacenamiento de la información por departamento.

2. Definir tres arreglos de 12 elementos cada uno. De esta forma, cada arreglo almacenará la información relativa a un departamento a lo largo del año.

Sin embargo, no resulta muy práctico adoptar alguna de las dos alternativas. Se necesita una estructura que permita manejar los datos considerando los meses —renglones de la tabla—, y los departamentos —columnas de la tabla—; es decir, una estructura que trate a la información como un todo. La estructura que tiene esta característica se denomina **arreglo bidimensional**.

Un arreglo bidimensional es una colección **homogénea**, **finita** y **ordenada** de datos, en la que se hace referencia a cada componente del arreglo por medio de dos índices. El primero se utiliza para indicar el renglón, y el segundo para señalar la columna. Un arreglo bidimensional también se puede definir como un arreglo de arreglos. En la figura 1.14 se presenta un arreglo de tipo bidimensional.

El arreglo $A(M \times N)$ tiene M renglones y N columnas. Un elemento $A[I, J]$ se localiza en el renglón I, y en la columna J. Internamente en memoria se reservan $M \times N$ posiciones consecutivas para almacenar todos los elementos del arreglo.

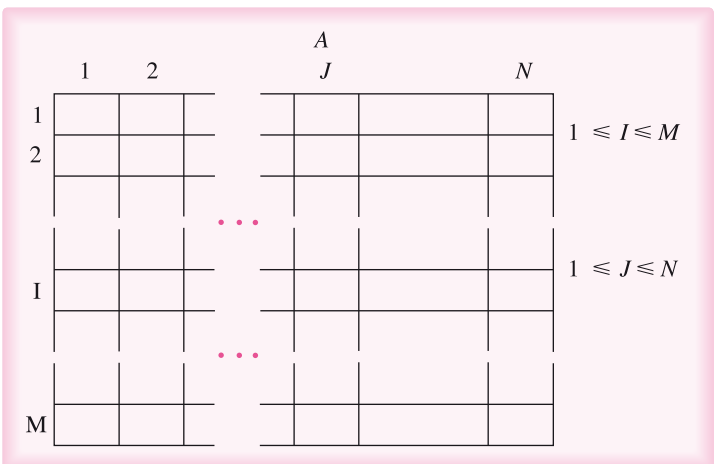
1.3.1 Declaración de arreglos bidimensionales

Los arreglos bidimensionales se declaran cuando se especifican el número de renglones y el número de columnas, junto con el tipo de dato de los componentes.

id_arreglo = ARREGLO [líminfr..límsupr,líminfc..límsupc] DE tipo

Con **líminfr** y **límsupr** se declara el tipo de dato del índice de los renglones y cuántos renglones tendrá el arreglo. Asimismo, con **líminfc** y **límsupc** se declara el tipo

FIGURA 1.14
Representación de un arreglo bidimensional.



de dato del índice de las columnas y cuántas columnas tendrá el arreglo. Con **tipo** se declara el tipo de datos de todos los componentes del arreglo.

El número total de componentes (NTC) de un arreglo bidimensional está determinado por la expresión:

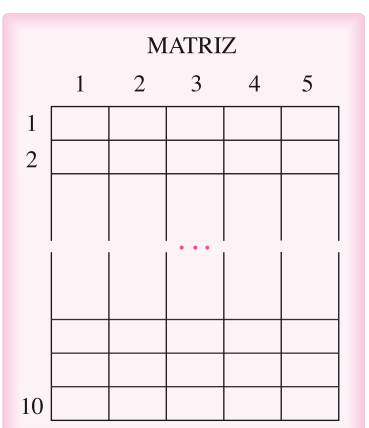
NTC = (límsupr − líminfr + 1) * (límsupc − líminfc + 1) **Fórmula 1.2**

Al igual que en el caso de los arreglos unidimensionales, los índices pueden ser cualquier tipo de dato ordinal (escalar, entero, carácter), mientras que los componentes pueden ser de cualquier tipo (reales, enteros, cadenas de caracteres, etc.). A continuación se analizan algunos ejemplos de arreglos bidimensionales.

Ejemplo 1.6

Sea MATRIZ un arreglo bidimensional de números reales con índices enteros. Su representación se muestra en la figura 1.15.

FIGURA 1.15



MATRIZ = ARREGLO[1..10,1..5] DE reales

- NTC = (10 −1 + 1)*(5 − 1 + 1) = 10 * 5 = 50
- Cada componente de MATRIZ será un número real. Para hacer referencia a cada uno de ellos se usarán dos índices y el nombre de la variable tipo arreglo: MATRIZ[i,j]

Donde: $1 \leq i \leq 10$
$1 \leq j \leq 5$

Ejemplo 1.7

Sea COSTOS un arreglo bidimensional de números reales con índices de tipo escalar. Su representación se muestra en la figura 1.16.

meses = (ene, feb, mar, abr, may, jun, jul, ago, set, oct, nov, dic)
departamentos = (dulces, conservas, bebidas)

COSTOS = ARREGLO[meses, departamentos] DE reales

- NTC = (ord(dic) − ord(ene) + 1) * (ord(bebidas) − ord(dulces) + 1)
  = (11 − 0 + 1) * (2 − 0 + 1) = 12 * 3 = 36

- Cada componente de COSTOS será un real. Para hacer referencia a cada uno de ellos usaremos dos índices y el nombre de la variable tipo arreglo COSTOS[i, j]

Donde: ene ≤ i ≤ dic
dulces ≤ j ≤ bebidas

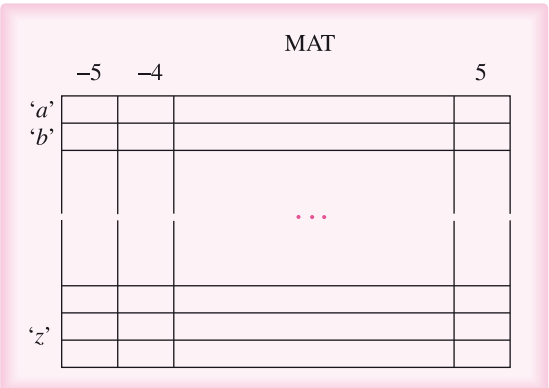
Ejemplo 1.8

Sea MAT un arreglo bidimensional de cadenas de caracteres con índices para los renglones de tipo carácter y para las columnas de tipo entero. Su representación se muestra en la figura 1.17.

MAT = ARREGLO[‘a’..‘z’, − 5..5] DE cadena-de-caracteres

FIGURA 1.16

FIGURA 1.17



- NTC = (ord('z') − ord('a') + 1) * (5 − (−5) + 1)
  = (122 − 97 + 1) * (5 + 5 + 1) = 26 * 11 = 286

- Cada componente de MAT será un valor de tipo cadena de caracteres. Para hacer referencia a cada uno de ellos, se usarán dos índices y el nombre de la variable tipo arreglo: MAT[*i*,*j*]

  Donde: '*a*' ≤ *i* ≤ 'z'
  −5 ≤ *j* ≤ 5

Ejemplo 1.9

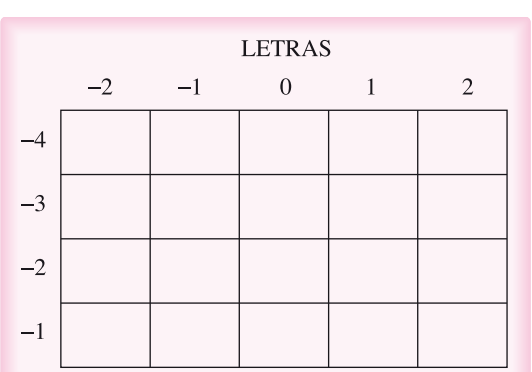
Sea LETRAS un arreglo bidimensional de caracteres con índices enteros. Su representación se muestra en la figura 1.18.

LETRAS = ARREGLO [−4..−1, −2..2] DE caracteres

- NTC = (−1 – (−4) + 1) * (2 – (−2) + 1) = 4 * 5 = 20
- Cada componente de LETRAS será un valor tipo carácter. Para hacer referencia a cada uno de ellos, se usarán dos índices y el nombre de la variable tipo arreglo: LETRAS[*i*, *j*]

  Donde: −4 ≤ *i* ≤ −1
  −2 ≤ *j* ≤ 2

FIGURA 1.18



1.3.2. Operaciones con arreglos bidimensionales

Las operaciones que se pueden realizar con arreglos bidimensionales son:

- Lectura/Escritura
- Asignación
- Actualización: Inserción
  Eliminación
  Modificación
- Ordenación
- Búsqueda

Los arreglos bidimensionales se consideran una generalización de los unidimensionales, por lo que se presentará una revisión rápida de algunas de las operaciones mencionadas. Para ilustrarlas se utilizarán los ejemplos anteriores.

Lectura

Cuando se presentó la operación de lectura en arreglos unidimensionales, se mencionó que con la ayuda de un ciclo se iban leyendo y asignando valores a cada uno de los componentes. Lo mismo sucede con los arreglos bidimensionales. Sin embargo, como sus elementos deben indicarse por medio de dos índices, normalmente se usan dos ciclos para lograr la lectura de elementos consecutivos.

Supongamos, por ejemplo, que se desea leer todos los elementos del arreglo bidimensional MATRIZ. Los pasos a seguir son:

```
Repetir con I desde 1 hasta 10
    Repetir con J desde 1 hasta 5
        Leer MATRIZ[I, J]
```

Al variar los índices de *I* y *J*, cada elemento de MATRIZ que se lee se asigna al lugar que le corresponde en el arreglo, según la posición de los índices *I* y *J*.

Para *I* = 1 y *J* = 1, se lee el elemento del renglón 1 y columna 1.
I = 1 y *J* = 2, se lee el elemento del renglón 1 y columna 2.
...
I = 10 y *J* = 5, se lee el elemento del renglón 10 y columna 5.

Escritura

La escritura de un arreglo bidimensional también se lleva a cabo elemento tras elemento. Supongamos que se quiera escribir todos los componentes del arreglo MATRIZ. Los pasos a seguir son:

```
Repetir con I desde 1 hasta 10
    Repetir con J desde 1 hasta 5
        Escribir MATRIZ[I,J]
```

Al variar los valores de *I* y *J* se escribe el elemento de MATRIZ correspondiente a la posición indicada justamente por los índices *I* y *J*.

Para $I = 1$ y $J = 1$, se escribe el elemento del renglón 1 y columna 1.
$I = 1$ y $J = 2$, se escribe el elemento del renglón 1 y columna 2.
...
$I = 10$ y $J = 5$, se escribe el elemento del renglón 10 y columna 5.

Asignación

La asignación de valores a un arreglo bidimensional se realiza de diferentes formas. La forma depende del número de componentes involucrados. Observemos a continuación dos alternativas diferentes.

1. Se asignan valores a todos los elementos del arreglo: en este caso se necesitarán dos ciclos para recorrer todo el arreglo.

```
Repetir con I desde 1 hasta 10
    Repetir con J desde 1 hasta 5
        MATRIZ[I,J] ← 0
```

Al variar los valores de I y J se asigna el 0 al elemento de MATRIZ correspondiente a la posición indicada por los índices I y J.

Para $I = 1$ y $J = 1$, se asigna el valor 0 al elemento del renglón 1 y columna 1.
$I = 1$ y $J = 2$, se asigna el valor 0 al elemento del renglón 1 y columna 2.
...
$I = 10$ y $J = 5$, se asigna el valor 0 al elemento del renglón 10 y columna 5.

En la figura 1.19 se presenta cómo queda el arreglo bidimensional cuando se asigna el valor 0 a cada una de las casillas.

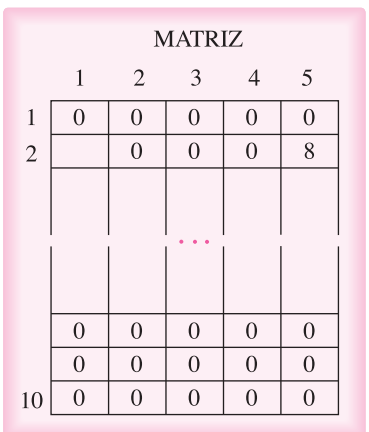
2. Se asigna un valor a un elemento en particular del arreglo: en este caso la asignación es directa y se debe indicar el renglón y la columna del componente involucrado. Por ejemplo, para asignar el valor 8 al elemento del renglón 2 y columna 5 se procede de la siguiente manera:

FIGURA 1.19
Asignación de arreglos.

MATRIZ

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
			...		
	0	0	0	0	0
	0	0	0	0	0
10	0	0	0	0	0

FIGURA 1.20
Asignación de arreglos.



MATRIZ[2,5] ← 8

El arreglo se muestra en la figura 1.20.

Es importante aclarar que las operaciones de lectura, escritura y asignación a todos los elementos de un arreglo bidimensional se pueden hacer tanto por renglones como por columnas.

1.4 ARREGLOS DE MÁS DE DOS DIMENSIONES

Un arreglo multidimensional —*N* dimensiones— se define como una colección finita, homogénea y ordenada de $K_1 \times K_2 \times ... \times K_N$ elementos. Para hacer referencia a cada componente de un arreglo de *N* dimensiones, se usarán *N* índices, uno para cada dimensión.

El arreglo *A* de *N* dimensiones se declara de la siguiente manera:

$$A = \text{ARREGLO}[LI_1..LS_1, LI_2..LS_2,...,LI_N..LS_N] \text{ DE tipo}$$

El total de componentes de *A* será:

$$NTC = (LS_1 - LI_1 + 1) * (LS_2 - LI_2 + 1) * ... * (LS_N - LI_N + 1) \qquad \textbf{Fórmula 1.3}$$

Por ejemplo, el arreglo tridimensional A[1...3, 1...2, 1...3] tendrá:

$$(3 - 1 + 1) * (2 - 1 + 1) * (3 - 1 + 1) = 3 * 2 * 3 = 18 \text{ elementos}$$

Gráficamente el arreglo *A* se puede representar como se muestra en las figuras 1.21 y 1.22:

A continuación se presenta un ejemplo de un arreglo tridimensional.

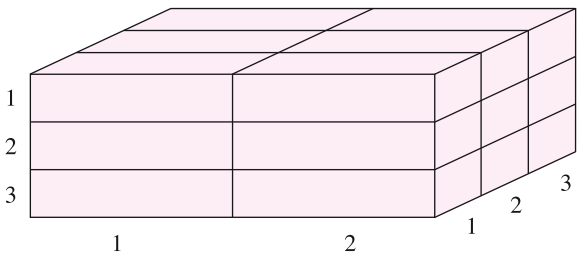


FIGURA 1.21
Representación de arreglos de más de dos dimensiones.

Ejemplo 1.10

Una empresa lleva un registro del total producido mensualmente por cada departamento. La empresa consta de cinco departamentos y la información se ha registrado a lo largo de los últimos cuatro años. Para almacenar los datos de la producción de la empresa, se requiere entonces de un arreglo de tres dimensiones (5 × 12 × 4 = 240 elementos), como el de la figura 1.23.

$$A = \text{ARREGLO [1..5, 1..12, 1..4] DE reales}$$

Supongamos que la empresa necesita obtener la siguiente información:

a) El total mensual de cada departamento durante el segundo año. Para obtener la información solicitada se deben realizar los siguientes pasos:

```
Repetir con I desde 1 hasta 5
    Repetir con J desde 1 hasta 12
        Escribir A[I,J,2]
```

Observe que para este caso se asigna la constante 2 al tercer índice —el de los años— y se hace variar a los otros dos índices. De esta manera se escribirán las producciones mensuales.

b) El total de la producción durante el primer año. Para obtener la información solicitada se deben realizar los siguientes pasos:

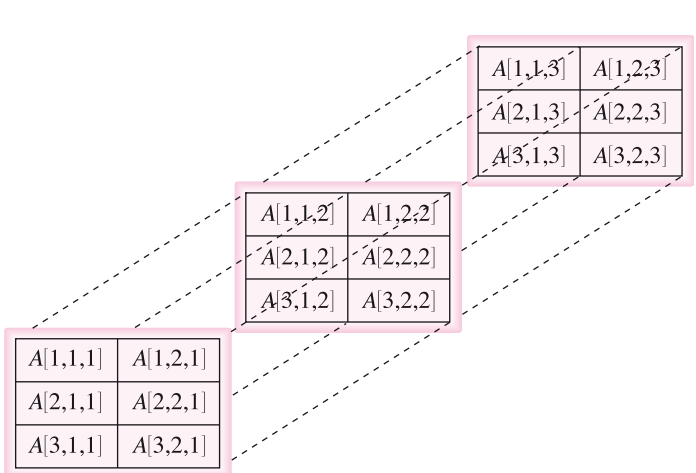
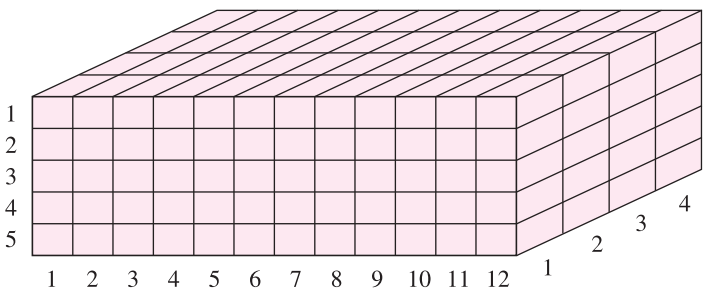


FIGURA 1.22
Representación de arreglos de más de dos dimensiones.

FIGURA 1.23




```
Hacer SUMA ← 0
Repetir con I desde 1 hasta 5
    Repetir con J desde 1 hasta 12
        Hacer SUMA ← SUMA + A[I,J,1]
Escribir SUMA
```

Observe que este caso es similar al anterior. La diferencia radica en que las cantidades mensuales no se escribirán, sino que se acumularán obteniendo el total anual.

c) El total de la producción del departamento 3 a lo largo del último año. Para obtener la información solicitada será necesario ejecutar los siguientes pasos:

```
Hacer SUMA ← 0
Repetir con J desde 1 hasta 12
    Hacer SUMA ← SUMA + A[3,J,4]
Escribir SUMA
```

Note que en este caso se tienen dos índices constantes, el de departamentos y el de años, y se hace variar solamente el índice de meses. Concluido el ciclo se escribirá el total producido por el departamento 3 durante el cuarto año.

1.5 LA CLASE *ARREGLO*

Para entender la clase arreglo, se requiere primero conocer algunos conceptos básicos relacionados con el paradigma de la programación orientada a objetos (POO).

Una **clase** define a un **objeto** por medio de la descripción de sus datos, conocidos como **atributos** y de su comportamiento, representado por métodos. Se dice que los atributos y los métodos son **miembros** de la clase.

Una clase puede representar a los alumnos de una escuela. En este caso los datos son los atributos que caracterizan a un alumno, por ejemplo, nombre, fecha de nacimiento, dirección, teléfono, etcétera, mientras que el comportamiento hace referencia a las operaciones que pueden realizarse sobre esos datos, por ejemplo, cambiar dirección o teléfono del alumno.

La programación orientada a objetos tiene cuatro propiedades:

1. Abstracción.
2. Encapsulamiento u ocultamiento de la información.

3. Herencia.
4. Polimorfismo.

La **abstracción** permite concentrarse en los datos y operaciones que definen a un conjunto de objetos, ignorando los elementos que no son relevantes. La segunda propiedad, **encapsulamiento**, implica que tanto los atributos como los métodos forman un todo —la clase— y pueden ocultarse de los clientes de la clase, al controlar de esta manera el acceso que se tenga a sus integrantes. Por su parte, la **herencia** representa la propiedad que permite compartir atributos y métodos entre clases. Por último, el **polimorfismo** ofrece la facilidad de que ciertos métodos puedan adoptar distintas formas.

La **clase *Arreglo*** tendrá atributos y métodos. Los atributos constituirán la colección de elementos y el tamaño. Los métodos serán todas las operaciones analizadas en las secciones previas: lectura, inserción, eliminación, etcétera. Gráficamente la clase *Arreglo* puede verse como se muestra en la figura 1.24.

Un **objeto** es una instancia de una clase. Es decir, esta última representa a un conjunto de objetos, a un concepto general, por ejemplo, los alumnos de una escuela o los arreglos, mientras que los primeros son ocurrencias de la clase. Considerando la clase *Arreglo*, un ejemplo de objeto será el arreglo de calificaciones de un grupo de alumnos.

En los lenguajes de programación orientada a objetos más conocidos se usa la notación de puntos para tener acceso a los miembros no privados de un objeto.

<objeto>.<miembro>

Dentro de un método de una clase, la referencia a cualquiera de sus otros miembros no requiere el uso de esta notación. Asumiendo que la variable CALIF es un objeto de la clase *Arreglo*, se pueden tener las siguientes instrucciones:

```
CALIF.Tamaño = CALIF.Tamaño −2
CALIF.Datos[6] = 10
```

Para el caso de que las instrucciones fueran parte de un método, se puede omitir el nombre del objeto y el punto, y usar directamente el atributo.

```
Datos[6] = 10
```

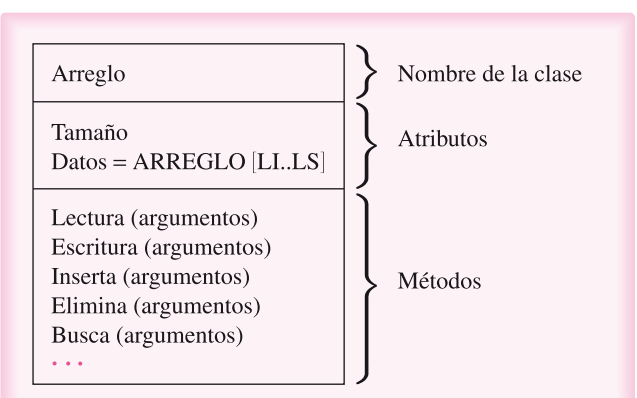


FIGURA 1.24
Clase Arreglo.

1.6 REGISTROS

De acuerdo con lo estudiado en las secciones previas, los arreglos son estructuras de datos muy útiles para almacenar una colección de datos, todos del mismo tipo. Sin embargo, en la práctica, a veces se necesitan estructuras que permitan almacenar datos de distintos tipos que sean manipulados como un único dato. Para ilustrar este problema se incluye el siguiente ejemplo.

Ejemplo 1.11

Una compañía tiene por cada empleado la siguiente información:

- Nombre (cadena de caracteres)
- Dirección (cadena de caracteres)
- Edad (entero)
- Sexo (carácter)
- Antigüedad (entero)

Si se quisiera almacenar estos datos no sería posible usar un arreglo, ya que sus componentes deben ser todos del mismo tipo. La estructura que puede guardar esta información de manera efectiva se conoce como **registro** o **estructura**.

Un registro se define como una colección finita y heterogénea de elementos. También representa un tipo de dato estructurado, en el que cada uno de sus componentes se denomina **campo**. Los campos de un registro pueden ser todos de diferentes tipos de datos. Por tanto, también podrán ser registros o arreglos. Cada campo se identifica con un nombre único, el identificador de campo. Otra diferencia importante con los arreglos es que no es necesario establecer un orden entre los campos.

1.6.1 Declaración de registros

Como no es la intención de los autores seguir la sintaxis de algún lenguaje de programación en particular, un registro se declara de la siguiente forma:

ident_registro = REGISTRO
    id_campo_1: $tipo_1$
    id_campo_2: $tipo_2$
    ...
    id_campo_n: $tipo_n$
{Fin de la declaración del registro 1}

Donde: ident_registro es el nombre del dato tipo registro
    id_campo_i es el nombre del campo i
    $id\_campo_i \neq id\_campo_j \; \forall_{i,j} = {}_{1,..,n} e_{i \neq j}$
    $tipo_i$ es el tipo del campo i

Los que siguen son ejemplos de declaraciones de registros, con su correspondiente representación gráfica.

Ejemplo 1.12

Sea FECHA un registro formado por tres campos numéricos. Su representación se muestra en la figura 1.25.

```
FECHA = REGISTRO
    día:  1..31
    mes: 1..12
    año:  0..2100
{Fin de la declaración del registro FECHA}
```

Ejemplo 1.13

Sea DOMICILIO un registro formado por cuatro campos, uno de ellos es numérico y los tres restantes del tipo cadena de caracteres. Su representación se muestra en la figura 1.26.

```
DOMICILIO = REGISTRO
    calle: cadena_de_caracteres
    número: entero
    ciudad: cadena_de_caracteres
    país: cadena_de_caracteres
{Fin de la declaración del registro DOMICILIO}
```

Ejemplo 1.14

Sea CLIENTE un registro formado por cuatro campos, dos del tipo cadena de caracteres, uno del tipo real y el otro del tipo booleano. Su representación se muestra en la figura 1.27.

```
CLIENTE = REGISTRO
    nombre: cadena_de_caracteres
    teléfono: cadena_de_caracteres
    saldo: real
    moroso: booleano
{Fin de la declaración del registro CLIENTE}
```

1.6.2 Acceso a los campos de un registro

Como un registro es un tipo de dato estructurado, no se puede tener acceso a él directamente como un único dato, sino que se debe especificar el elemento —campo— del

FIGURA 1.25

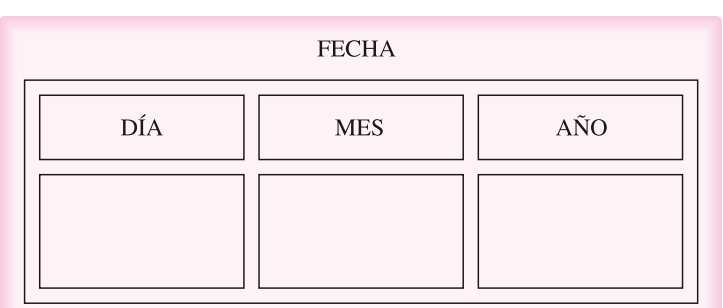
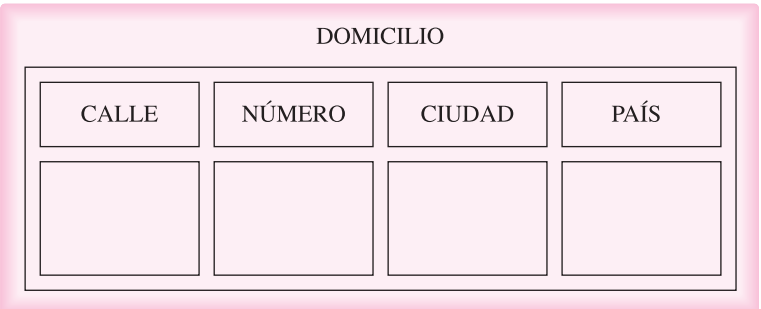


FIGURA 1.26



registro que nos interesa. Para ello, en la mayoría de los lenguajes se sigue la siguiente sintaxis:

variable_registro.id_campo

Donde: variable_registro es una variable de tipo registro
id_campo es el identificador del campo deseado

Es decir, se usarán dos identificadores para hacer referencia a un elemento: el nombre de la variable tipo registro y el nombre del campo, separados entre sí por un punto.

De acuerdo con los ejemplos de registros 1.12, 1.13 y 1.14, se presentan a continuación diferentes casos que ilustran el acceso a los campos de un registro.

a) Para leer los tres campos de una variable *F* de tipo FECHA:
Leer *F.día,F.mes,F.año*

b) Para escribir los cuatro campos de una variable *D* de tipo DOMICILIO:
Escribir *D.calle,D.número,D.ciudad,D.país*

c) Para asignar valores a algunos de los campos de una variable *C* de tipo CLIENTE:

C.saldo ← C.saldo + cant
C.moroso ← VERDADERO
C.nombre ← "Juan Pérez"

FIGURA 1.27

En general, como se mencionó anteriormente, el orden en el que se manejan los campos no es importante. Es decir, se podrían haber leído los campos de la variable *F* de la siguiente manera:

Leer *F.año,F.día,F.mes*

Sólo se debe tener en cuenta que los datos proporcionados por el usuario o asignados en un algoritmo se correspondan en tipo con los campos.

1.6.3 Diferencias entre registros y arreglos

Las dos diferencias sustanciales existentes entre registros y arreglos son:

1. Un arreglo puede almacenar *N* elementos del mismo tipo —estructura de datos homogénea—, mientras que un registro puede almacenar *N* elementos de diferentes tipos de datos —estructura de datos heterogénea—.
2. A los componentes de un arreglo se tiene acceso por medio de índices que indican la posición del elemento correspondiente en el arreglo, mientras que a los componentes de un registro, los campos, se tiene acceso por medio de su nombre, que es único.

1.6.4 Combinaciones entre arreglos y registros

Los registros tienen varios campos. Cada uno de ellos puede ser de cualquier tipo de datos, simples o estructurados. Sin embargo, los componentes del nivel más bajo de un tipo estructurado siempre deben ser tipos simples de datos.

De acuerdo con esta condición, se infiere que un campo de un registro puede ser otro registro o bien un arreglo. Por otra parte, los componentes de un arreglo también pueden ser registros. Estos casos enunciados, además, se pueden presentar en forma anidada.

Arreglos de registros

En este caso, cada elemento del arreglo es un registro. Todos los componentes del arreglo tienen que ser del mismo tipo de registro, ya que es una estructura de datos homogénea. A continuación presentamos un ejemplo.

Ejemplo 1.15

Una empresa registra para cada uno de sus clientes los siguientes datos:

- Nombre (cadena de caracteres)
- Teléfono (cadena de caracteres)
- Saldo (real)
- Moroso (booleano)

Si la empresa tiene *N* clientes necesitará un arreglo de *N* elementos, en el cual cada uno de sus componentes es un registro como el descrito en el ejemplo 1.14. La figura 1.28 muestra la estructura de datos correcta para resolver este problema:

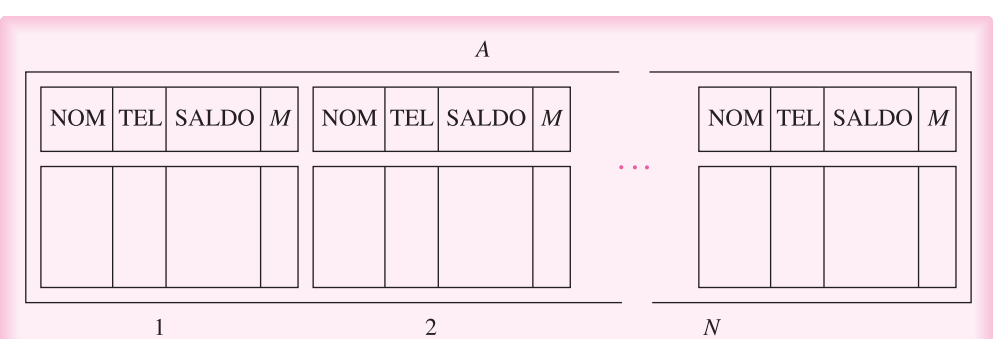
A = ARREGLO [1..100] DE CLIENTE

Cada elemento de *A* será un dato tipo CLIENTE. Por tanto, si se quiere, por ejemplo, leer el arreglo *A*, debe leerse por cada componente cada uno de los campos que forman al registro.

```
Repetir con I desde 1 hasta N
    Leer A[I].nombre
    Leer A[I].teléfono
    Leer A[I].saldo
    Leer A[I].moroso
```

Con *A*[*I*] se hace referencia al elemento *I* del arreglo *A*, que es un registro; con .id_campo se especifica cuál de los campos del registro se leerá. De forma similar se procede para escritura, asignación, etcétera.

FIGURA 1.28
Arreglo de registros.



Registros anidados

En los registros anidados, al menos un campo del registro es del tipo registro. Observemos a continuación el siguiente ejemplo.

Ejemplo 1.16

Una empresa registra para cada uno de sus acreedores los siguientes datos:

- Nombre (cadena de caracteres)
- Dirección:
  - Calle (cadena de caracteres)
  - Número (entero)
  - Ciudad (cadena de caracteres)
  - País (cadena de caracteres)
- Saldo (real)

Para definir el tipo de dato del campo *dirección*, es necesario declarar previamente un registro formado por los cuatro componentes: calle, número, ciudad y país que se especifican. Se usará el registro del ejemplo 1.13, presentado anteriormente, para resolver este caso.

```
ACREEDOR = REGISTRO
    nombre: cadena_de_caracteres
    dirección: DOMICILIO
    saldo: real
{Fin de la declaración del registro ACREEDOR}
```

La figura 1.29 muestra la estructura de datos requerida.

En este caso, el registro tiene un campo —dirección— que es del tipo de datos DOMICILIO, el cual es un registro de cuatro campos. Para tener acceso a los campos que, a su vez, son registros, en la mayoría de los lenguajes se sigue la siguiente sintaxis:

variable_registro.id_campo$_1$.id_campo$_n$

Donde: variable_registro es una variable de tipo registro
id_campo$_1$ es el identificador de un campo del registro: el campo es de tipo registro
id_campo$_n$ representa el identificador de un campo

FIGURA 1.29
Registros anidados.

Para tener acceso a los campos de la variable *AC* de tipo ACREEDOR, la secuencia a seguir es la siguiente:

AC.nombre
AC.dirección.calle
AC.dirección.número
AC.dirección.ciudad
AC.dirección.país
AC.saldo

Registros con arreglos

Los registros con arreglos tienen, por lo menos, un campo que es de tipo arreglo. Analice cuidadosamente el siguiente ejemplo.

Ejemplo 1.17

Una empresa registra para cada uno de sus clientes los siguientes datos:

- Nombre (cadena de caracteres)
- Teléfono (cadena de caracteres)
- Saldo mensual del último año (arreglo de reales)
- Moroso (booleano)

La definición del registro correspondiente es:

```
CLIENTE = REGISTRO
      nombre: cadena_de_caracteres
      teléfono: cadena_de_caracteres
      saldos: ARREGLO [1..12] DE reales
      moroso: booleano
{Fin de la declaración del registro CLIENTE}
```

La figura 1.30 muestra la estructura requerida.

Para este caso el registro tiene un campo, saldos, que es un arreglo unidimensional de 12 elementos reales. Con el propósito de hacer referencia a ese campo, se procede de la siguiente manera:

FIGURA 1.30
Registros con arreglos.

variable_registro.id_campo[índice]

Para tener acceso a los campos de la variable *CLI* de tipo CLIENTE se debe seguir la secuencia:

CLI.nombre
CLI.teléfono
Repetir con I desde 1 hasta 12
    CLI.saldos[*I*]
CLI.moroso

Las tres posibles combinaciones analizadas aquí: arreglos de registros, registros anidados y registros con arreglos, pueden presentarse de manera simultánea y en diferentes niveles en una misma estructura de datos. En estos casos, se recomienda que la estructura resultante sea comprensible y que no se complique demasiado el acceso a los datos individuales.

1.6.5 Arreglos paralelos

Por **arreglos paralelos** se entiende dos o más arreglos cuyos elementos se corresponden. Es decir, los componentes que ocupan una misma posición en diferentes arreglos tienen una estrecha relación semántica. Para ilustrar esta idea, a continuación se presentará un caso práctico y su solución, mediante arreglos paralelos.

Supongamos que se conoce el nombre del alumno y la calificación obtenida por éste en un examen que fue aplicado a un grupo de 30 alumnos. Si se quisiera usar estos datos para generar información, por ejemplo, promedio del grupo, calificación más alta, nombre de los alumnos con calificación inferior al promedio, etc., se tendrían dos alternativas principales en el diseño de la solución.

Uso de arreglos paralelos

Si se utilizan arreglos paralelos para resolver este problema, se requiere de dos arreglos unidimensionales; en uno se almacenará el nombre de los alumnos, y en otro la calificación obtenida por éste en el examen. Es decir, a cada elemento del arreglo NOMBRES le corresponderá entonces uno del arreglo CALIFICACIÓN. Así, si se quiere hacer referencia a la calificación de NOMBRES[*I*], se utilizará CALIFICACIÓN[*I*]. Observe la figura 1.31.

López	obtuvo una calificación de 9.5
Martínez	obtuvo una calificación de 5.8
Torres	obtuvo una calificación de 7.4
...	...
Viasa	obtuvo una calificación de 10.0

A continuación se incluye un algoritmo que calcula el promedio del grupo e imprime el nombre de los alumnos que tengan calificación menor al promedio.

FIGURA 1.31
Arreglos paralelos.

Algoritmo 1.11 Arreglos_paralelos

Arreglos_paralelos

{Este algoritmo calcula el promedio del grupo e imprime el nombre de los alumnos con calificación menor al promedio}
{NOMBRE y CALIFICACIÓN son variables de tipo arreglo. *I* es una variable de tipo entero. PROM y *AC* son variables de tipo real}

1. Hacer *AC* ← 0
2. *Repetir* con *I* desde 1 hasta 30
Leer NOMBRE[*I*] y CALIFICACIÓN[*I*]
Hacer *AC* ← *AC* + CALIFICACIÓN[*I*]
3. {Fin del ciclo del paso 2}
{Se calcula el promedio del grupo}
4. Hacer PROM ← *AC*/30
5. Escribir "El promedio del grupo es": PROM

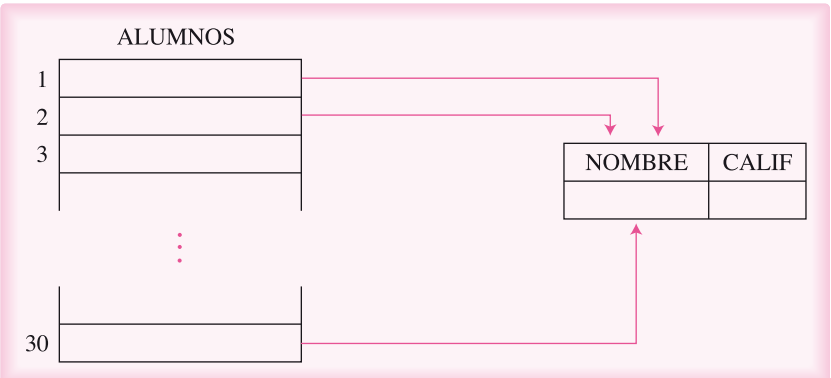
{Búsqueda e impresión de los nombres de los alumnos con calificación inferior al promedio}

6. *Repetir* con *I* desde 1 hasta 30
6.1 *Si* (CALIFICACIÓN[*I*] < PROM) *entonces*
Escribir NOMBRE[*I*]
6.2 {Fin del condicional del paso 6.1}
7. {Fin del ciclo del paso 6}

Uso de arreglos de registros

Otra solución al problema sería utilizar un arreglo de registros. En este caso, cada componente del arreglo ALUMNO es un registro que tiene dos campos: NOMBRE y CALIF. Observe la figura 1.32.

FIGURA 1.32



Así: ALUMNOS[*I*].NOMBRE hará referencia al nombre del alumno *I*
ALUMNOS[*I*].CALIF hará referencia a la calificación obtenida por el alumno *I*

El siguiente algoritmo presenta la solución al problema anterior mediante un arreglo de registros.

Algoritmo 1.12 Arreglo_de_registros

Arreglo_de_registros

{Este algoritmo calcula el promedio del grupo e imprime el nombre de los alumnos con calificación menor al promedio}
{ALUMNOS es un arreglo de registros. *I* es una variable de tipo entero. *AC* y PROM son variables de tipo real}

1. Hacer *AC* ← 0
2. *Repetir* con *I* desde 1 hasta 30
Leer ALUMNOS[*I*].NOMBRE y ALUMNOS[*I*].CALIF
Hacer *AC* ← *AC* + ALUMNOS[*I*].CALIF
3. {Fin del ciclo del paso 2}
4. Hacer PROM ← *AC*/30
5. Escribir "El promedio del grupo es": PROM

{Búsqueda e impresión de los alumnos con calificación inferior al promedio}

6. *Repetir* con *I* desde 1 hasta 30
6.1 *Si* (ALUMNOS[*I*].CALIF < PROM) *entonces*
Escribir ALUMNOS[*I*].NOMBRE
6.2 {Fin del condicional del paso 6.1}
7. {Fin del ciclo del paso 6}

1.7 REGISTROS Y CLASES

Los registros son las estructuras de datos que más se parecen al concepto de clase presentado. En la sección anterior se dijo que un registro almacena las principales características de un conjunto de objetos. Cada una de esas características constituye un campo del registro. Al establecer la relación con las clases, los campos representan los atributos. Por tanto, sólo se agregan los métodos —operaciones que pueden aplicarse sobre los campos— para completar la definición de una clase.

La **clase *Registro*** como tal no se declara, porque lo que se requiere es una clase por cada registro. Es decir, si se desea representar a los clientes de una empresa, según el ejemplo visto en la sección anterior, desde el punto de vista de la programación orientada a objetos, se deberá definir una clase que contendrá tanto los atributos —lo que en registros se llaman campos— como todas las operaciones válidas para un cliente, por ejemplo, actualizar el saldo, cambiar el número telefónico, etcétera. Gráficamente la clase ***Cliente*** puede verse como se muestra en la figura 1.33.

Un objeto de la clase ***Cliente*** es una instancia de la misma. Es decir, está representando a un cliente con un nombre, un número telefónico y un saldo específico.

La **notación de puntos** utilizada en los registros —<variable_registro>.<campo>— es similar a la usada en los lenguajes orientados a objetos para tener acceso a los miembros no privados de un objeto —<objeto>.<miembro>—. Al asumir que la variable CLI es un objeto de la clase *Cliente* previamente definida, se puede tener acceso a los miembros no privados de dicho objeto por medio de las instrucciones:

a) CLI.ActualizarSaldo (NuevoSaldo)
En este ejemplo se está invocando al método que actualiza el saldo del cliente. El método tiene un argumento que indica el nuevo valor que se asignará al atributo Saldo.

b) CLI.CambiaTeléfono (NuevoTel)
En este ejemplo se está invocando al método que actualiza el número telefónico del cliente. El método tiene un argumento que indica el nuevo valor que se asignará al atributo *Teléfono*.

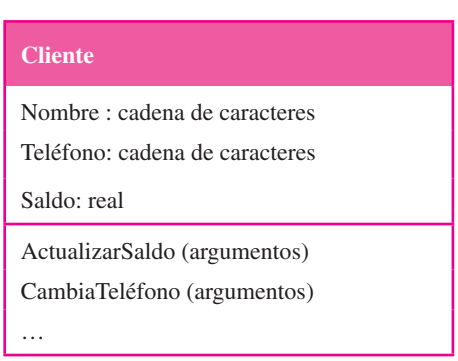


FIGURA 1.33
Clase *Cliente*.

EJERCICIOS

Arreglos de una dimensión y arreglos paralelos

1. En un arreglo unidimensional se ha almacenado el número total de toneladas de cereales cosechadas durante cada mes del año anterior. Escriba un programa que obtenga e imprima la siguiente información:

a) El promedio anual de toneladas cosechadas.
b) ¿Cuántos meses tuvieron cosecha superior al promedio anual?
c) ¿Cuántos meses tuvieron cosecha inferior al promedio anual?

2. En un arreglo unidimensional se almacenan las calificaciones finales de *N* alumnos de un curso universitario. Escriba un programa que calcule e imprima:

a) El promedio general del grupo.
b) Número de alumnos aprobados y reprobados.
c) Porcentaje de alumnos aprobados y reprobados.
d) Número de alumnos cuya calificación fue mayor o igual a 8.

3. Dada una cadena de caracteres como dato, se desea saber el número de veces que aparecen las letras '*a*', '*b*',...,'*z*' y '*A*', '*B*',...,'*Z*' en dicha cadena. Escriba un programa que resuelva el problema.

a) Si usó arreglos, ¿cuántos necesitó? ¿Por qué?
b) ¿Existe otra forma de resolverlo?

4. Dado un arreglo unidimensional de números enteros, ordenados crecientemente, escriba un programa que elimine todos los elementos repetidos. Considere que de haber valores repetidos, éstos se encontrarán en posiciones consecutivas del arreglo.

5. Una compañía almacena la información relacionada con sus proveedores en los siguientes arreglos:

PROVEEDORES

p_1	p_2	p_3	...	p_n

Cada p_1 es el nombre del proveedor *i*. Este arreglo está ordenado alfabéticamente.

CIUDAD

c_1	c_2	c_3	...	c_n

Cada c_1 representa el nombre de la ciudad en la que reside el proveedor *i*.

NÚMERO DE ARTÍCULOS

a_1	a_2	a_3	...	a_n

Cada a_i es el número de artículos diferentes que provee el proveedor *i*.

Escriba un programa que pueda llevar a cabo las siguientes transacciones:

a) Dado el nombre de un proveedor, informar el nombre de la ciudad en la que reside y el número de artículos que provee.
b) Actualizar el nombre de la ciudad, en caso de que un proveedor cambie de domicilio. Los datos serán el nombre del proveedor y el nombre de la ciudad a la cual se mudó.
c) Actualizar el número de artículos, manejados por un proveedor para el caso de que éste aumente o disminuya. Los datos serán el nombre del proveedor y la cantidad en la que aumenta (+) o disminuye (−) el total de artículos que provee.
d) La compañía incorpora a un nuevo proveedor. Actualizar los arreglos sin alterar el orden de PROVEEDORES. Los datos serán el nombre del proveedor, el nombre de la ciudad y el total de artículos que provee.
e) La compañía da de baja a un proveedor. Actualizar los arreglos. El dato será el nombre del proveedor.

6. Una inmobiliaria tiene información sobre departamentos en renta almacenada en dos arreglos:

EXTENSIÓN

e_1	e_2	e_3	...	e_n

El arreglo EXTENSIÓN almacena la superficie, en metros cuadrados, de cada uno de los *N* departamentos.

PRECIO

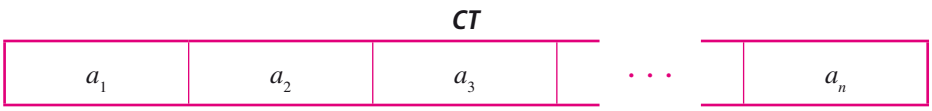
P_1	P_2	P_3	...	P_N

El arreglo PRECIO almacena los precios de alquiler de los *N* departamentos. Este arreglo está ordenado de manera creciente. Considere que no existen departamentos con igual superficie y distintos precios.

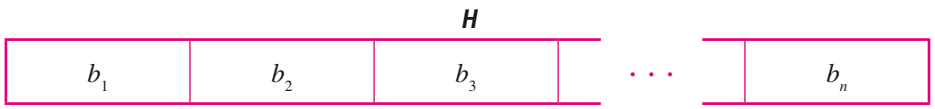
Escriba un programa que pueda llevar a cabo las siguientes operaciones:

a) Llega un cliente a la inmobiliaria y solicita rentar un departamento. Si existe alguno con superficie mayor o igual a la buscada y precio menor o igual al buscado, se dará de baja al departamento seleccionado.
b) Se vence un contrato y el cliente no desea renovarlo. Se deben actualizar los arreglos.

7. Se tiene la siguiente información:



En el arreglo *CT* se almacenan los nombres de *N* centros turísticos del país.



En el arreglo *H* se almacena el número de habitaciones de cada tipo, sencilla o doble, de cada centro turístico.

Por ejemplo:
H[1] guarda el número de habitaciones sencillas del centro 1.
H[2] guarda el número de habitaciones dobles del centro 1.
H[3] guarda el número de habitaciones sencillas del centro 2.
H[4] guarda el número de habitaciones dobles del centro 2.
etcétera.

En el arreglo *TR* se almacena el número total de restaurantes por centro turístico. Deberá desarrollar un programa que proporcione la siguiente información:

a) El nombre del centro turístico que cuenta con más restaurantes.
b) El nombre del centro turístico que cuenta con más habitaciones, teniendo en cuenta las sencillas y las dobles.
c) Dado el nombre de un centro turístico como dato, informar cuántas habitaciones tiene: sencillas, dobles y el total.
d) El nombre del centro turístico que más restaurantes tiene en relación con el número de habitaciones.

8. Se tienen tres arreglos: SUR, CENTRO y NORTE que almacenan los nombres de los países de Sur, Centro y Norteamérica, respectivamente. Los tres arreglos están ordenados alfabéticamente.

Escriba un programa que mezcle los tres arreglos anteriores, formando un cuarto arreglo, AMÉRICA, en el cual aparezcan los nombres de todos los países del continente ordenados alfabéticamente.

9. Se tienen dos arreglos: CINES y TEATROS. El primero almacena los nombres de todos los cines de la ciudad. Está ordenado alfabéticamente de manera ascendente:

$$CINES[1] \leq CINES[2] \leq ... \leq CINES[N]$$

El segundo arreglo guarda los nombres de todos los teatros de la ciudad. Está ordenado alfabéticamente de manera descendente:

$$TEATROS[1] \geq TEATROS[2] \geq ... \geq TEATROS[K]$$

Escriba un programa que mezcle estos arreglos formando un tercero, ENTRETENIMIENTOS, que quede ordenado alfabéticamente de manera ascendente.

10. Se tienen registradas las calificaciones obtenidas en un examen a 50 alumnos. Los datos son cal_1, cal_2, ..., cal_{50}, donde cal_i es un número entero comprendido entre los valores 0 y 10 ($0 \leq cal_i \leq 10$).

Escriba un programa que calcule e imprima la frecuencia de cada uno de los posibles valores.

La salida del programa se muestra a continuación:

Calificación	Frecuencia
0	1 ALUMNO
1	—
2	—
3	4 ALUMNOS
4	2 ALUMNOS
...	...
10	3 ALUMNOS

11. Escriba sus propios algoritmos para insertar, eliminar o modificar un elemento de un arreglo:

a) Si el arreglo está desordenado.
b) Si el arreglo está ordenado.

12. Dado un arreglo unidimensional de tipo entero que contiene calificaciones de exámenes de alumnos, construya un programa que calcule lo siguiente:

a) *Media aritmética*. Se calcula como la suma de los elementos entre el número de elementos.
b) *Varianza*. Se calcula como la suma de los cuadrados de las desviaciones de la media, entre el número de elementos.
c) *Desviación estándar*. Se calcula como la raíz cuadrada de la varianza.
d) *Moda*. Se calcula al obtener el número con mayor frecuencia.

13. Escriba un programa que almacene en un arreglo unidimensional los primeros 30 números perfectos. Un número se considera perfecto, si la suma de los divisores excepto el mismo es igual al propio número. El 6, por ejemplo, es un número perfecto.

Arreglos multidimensionales

14. Sean los arreglos bidimensionales $A(M \times N)$ y $B(M \times N)$
Donde: $1 \leq M \leq 10$,
$1 \leq N \leq 20$,
$a_{i,j}$ y $b_{i,j}$ son reales.
Escriba un programa que calcule $C(M \times N) = A(M \times N) + B(M \times N)$.

15. Sean los arreglos bidimensionales $A(M \times N)$ y $B(N \times P)$
Donde: $1 \leq M \leq 10$,
$1 \leq N \leq 10$,
$1 \leq P \leq 5$,
$a_{i,j}$ y $b_{i,j}$ son reales.
Escriba un programa que calcule $C(M \times P) = A(M \times N) * B(N \times P)$

16. Escriba un programa que llene de ceros una matriz $A(N \times N)$ excepto en la diagonal principal donde debe asignar 1. Si $N = 4$, la matriz debe quedar:

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

17. Escriba un programa que intercambie por renglón los elementos de un arreglo bidimensional. Los elementos del renglón 1 deben intercambiarse con los del renglón N, los del renglón 2 con los del $N - 1$, y así sucesivamente.
Por ejemplo, si A es:

	1	2	3	4
1	1	4	5	–5
2	0	87	1	0
3	2	4	10	3
4	9	5	7	5

El resultado de la operación debe ser:

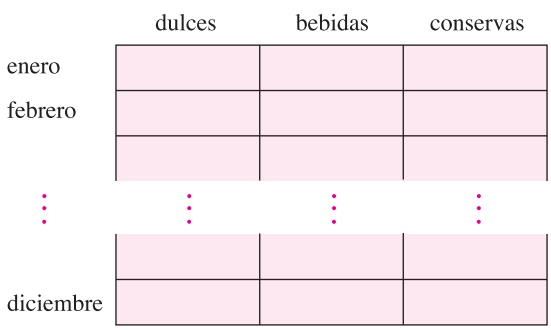
	1	2	3	4
1	9	5	7	5
2	2	4	10	3
3	0	87	1	0
4	1	4	5	–5

18. Dado como dato el arreglo bidimensional $A(M \times N)$, que almacena números reales.

Donde: $1 \leq M \leq 20$,
$1 \leq N \leq 20$,

Escriba un programa que encuentre e imprima el valor más grande almacenado en cada una de las columnas y en cada uno de los renglones del arreglo. Su programa debe imprimir, junto al valor encontrado, la columna o renglón en la cual se encontró.

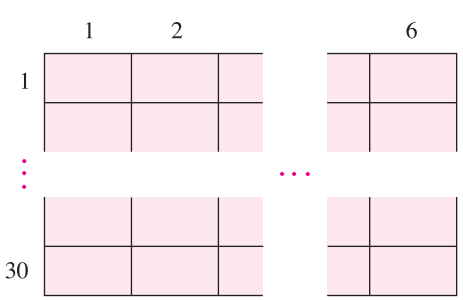
19. Se tienen los costos de producción de tres departamentos: dulces, bebidas y conservas, correspondientes a los 12 meses del año anterior.



Escriba un programa que pueda proporcionar la siguiente información:

a) ¿En qué mes se registró el mayor costo de producción de dulces?
b) Promedio anual de los costos de producción de bebidas.
c) ¿En qué mes se registró el mayor costo de producción en bebidas, y en qué mes el menor costo?
d) ¿Cuál fue el rubro que tuvo el menor costo de producción en diciembre?

20. Se tiene una tabla con las calificaciones obtenidas por 30 alumnos en seis exámenes diferentes:



Escriba un programa que calcule:

a) El promedio general de calificaciones de los 30 alumnos, considerando los seis exámenes.
b) El alumno que obtuvo la mayor calificación en el tercer examen.
c) El alumno, si lo hubiera, que obtuvo la mayor calificación en el primero y en el sexto exámenes.
d) Dado el número que identifica a un alumno, informar en qué examen logró la menor calificación.
e) ¿En cuál examen fue más alto el promedio de los 30 alumnos?

21. Escriba un programa que genere e imprima un cuadrado mágico de dimensión *N*. Observe que *N* es entero, positivo e impar. Un cuadrado mágico es una matriz cuadrada de orden *N*, que contiene a los números naturales del 1 al *N* * *N*, y donde la suma de cualquiera de los renglones, columnas o diagonales principales es siempre la misma. Puede utilizar los siguientes pasos para generar un cuadrado mágico:

a) El número 1 se coloca en la casilla central del primer renglón.
b) El siguiente número se coloca en la casilla correspondiente al renglón anterior y columna posterior.
c) El renglón anterior al primero es el último, y la columna posterior a la última es la primera.
d) Si el número es un sucesor de un múltiplo de *N*, no se aplica la regla 2, sino que se coloca en la casilla del renglón posterior y en la misma columna.

Si $N = 5$, el cuadrado generado debe quedar:

	1	2	3	4	5
1	17	24	1	8	15
2	23	5	7	14	16
3	4	6	13	20	22
4	10	12	19	21	3
5	11	18	25	2	9

22. Sean $A(M \times N)$ y $B(N)$ arreglos de dos y una dimensión, respectivamente. Escriba un programa que asigne valores a *B*, a partir de *A*, teniendo en cuenta los siguientes criterios:

a) $b_i = \sum_{j=1}^{n} a_{i,j}$ Si *i* es impar

b) $b_i = \sum_{j-1}^{n} a_{i,j} * a_{i-1,j} * a_{i-1,j}$ Si *i* es par

23. Sean $A(M \times N)$ y $B(N)$ dos arreglos de dos y una dimensión, respectivamente.

Escriba un programa que asigne valores a *A*, a partir de *B*, teniendo en cuenta los siguientes criterios:

a) $a_{ij} = bi$ Si $i \leq j$
b) $a_{ij} = 0$ Si $i > j$

Combinaciones entre arreglos y registros

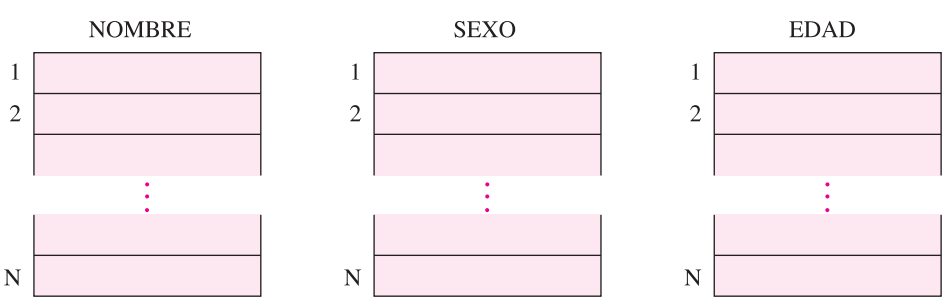
24. El departamento de personal de una escuela tiene registros del nombre, sexo y edad de cada uno de los profesores adscritos ahí.

NOMBRE	SEXO	EDAD

Escriba un programa que calcule e imprima los siguientes datos:

a) Edad promedio del grupo de profesores.
b) Nombre del profesor más joven del grupo.
c) Nombre del profesor de más edad.
d) Número de profesoras con edad mayor al promedio.
e) Número de profesores con edad menor al promedio.

25. Resuelva el problema anterior con tres arreglos paralelos. Compare sus soluciones.



26. En una escuela por cada alumno se tienen los siguientes datos:

- Nombre
- Matrícula
- Número de semestres cursados
- Calificación promedio por semestre

Escriba un programa que, dada la información de *N* alumnos, pueda realizar las siguientes operaciones:

a) Listar nombre y matrícula de estudiantes con promedios generales mayores o iguales a 8.
b) Actualizar los campos que correspondan cuando un estudiante ha concluido un semestre.
c) Listar nombre y matrícula de estudiantes que hayan obtenido 9 o más de calificación en todos los semestres cursados hasta el momento.

27. Una compañía distribuye *N* productos a distintos comercios de la ciudad. Para ello almacena en un arreglo toda la información relacionada con su mercancía:

- Clave
- Descripción
- Existencia
- Mínimo a mantener de existencia
- Precio unitario

Escriba un programa que efectúe las siguientes operaciones:

a) *Venta de un producto*: se deben actualizar los campos que correspondan y verificar que la nueva existencia no esté por debajo del mínimo. (Datos: clave, cantidad vendida.)
b) *Reabastecimiento de un producto*: se deben actualizar los campos que correspondan. (Datos: clave, cantidad comprada.)
c) *Actualizar el precio de un producto*. (Datos: clave, porcentaje de aumento.)
d) *Informar sobre un producto*: se deben proporcionar todos los datos relacionados con un producto. (Dato: clave.)

28. Al momento de su ingreso al hospital, a un paciente se le solicitan los siguientes datos:

- Nombre
- Edad
- Sexo
- Domicilio:
  - Calle
  - Número
  - Ciudad
- Teléfono
- Seguro (este campo tendrá el valor VERDADERO si el paciente tiene seguro médico y FALSO en otro caso)

Escriba un programa que pueda llevar a cabo las siguientes operaciones:

a) Listar los nombres de todos los pacientes hospitalizados.
b) Obtener el porcentaje de pacientes hospitalizados en las siguientes categorías (dadas por la edad):

Niños: hasta 13 años.
Jóvenes: mayores de 13 años y menores de 30.
Adultos: mayores de 30 años.

c) Obtener el porcentaje de hombres y de mujeres hospitalizados.
d) Dado el nombre de un paciente, listar todos los datos relacionados con dicho paciente.
e) Calcular el porcentaje de pacientes que poseen seguro médico.

29. Una inmobiliaria tiene información sobre departamentos en renta. De cada departamento se conoce:

- *Clave*: es un entero que identifica al inmueble.
- *Extensión*: superficie del departamento, en metros cuadrados.
- *Ubicación*: (excelente, buena, regular, mala).
- *Precio*: es un real.
- *Disponible*: VERDADERO si está disponible para la renta y FALSO si ya está rentado.

Diariamente acuden muchos clientes a la inmobiliaria solicitando información.

Escriba un programa capaz de realizar las siguientes operaciones sobre la información disponible:

a) Liste los datos de todos los departamentos disponibles que tengan un precio inferior o igual a cierto valor *P*.
b) Liste los datos de los departamentos disponibles que tengan una superficie mayor o igual a un cierto valor dado *E* y una ubicación excelente.
c) Liste el monto de la renta de todos los departamentos alquilados.
d) Llega un cliente solicitando rentar un departamento. Si existe alguno con una superficie mayor o igual a la deseada, con precio y ubicación que se ajustan a las necesidades del cliente, el departamento se rentará. Actualizar los datos que correspondan.
e) Se vence un contrato si no se renueva, actualizar los datos que correspondan.
f) Se ha decidido aumentar las rentas en un X%. Actualizar los precios de las rentas de los departamentos no alquilados.

Problemas interesantes

(Decida el lector qué estructura de datos debe utilizar para resolverlos.)

30. Escriba un programa que lea un número romano e imprima su equivalente en arábigo.

Recuerde que:

I = 1
V = 5

```
X =      10
L =      50
C =     100
D =     500
M =   1 000
```

31. Escriba un programa que calcule e imprima los números perfectos comprendidos entre dos números *A* y *B*. Un número es perfecto si la suma de sus divisores, excepto él mismo, es igual al propio número.

32. Escriba un subprograma que reciba como datos el nombre de un día de la semana y un número entero *N*, positivo o negativo, e imprima el día de la semana correspondiente a *N* días después —positivo— o *N* días antes —negativo— del día dado.

33. Lo mismo que en el problema 35, pero ahora con respecto a un mes.

34. Escriba un programa que calcule e imprima los números primos menores que cierto número dado *N*.

35. Escriba un programa que calcule e imprima los números primos gemelos menores que cierto número dado *N*. Dos números son primos gemelos si son números primos con una diferencia entre ellos de exactamente 2. Por ejemplo, 3 y 5 son primos gemelos.

Capítulo 2

ARREGLOS MULTIDIMENSIONALES REPRESENTADOS EN ARREGLOS UNIDIMENSIONALES

2.1 INTRODUCCIÓN

Actualmente la mayoría de los lenguajes de programación de alto nivel proporcionan al usuario medios eficaces para almacenar y recuperar elementos de arreglos bidimensionales, tridimensionales e incluso de más de tres dimensiones. El usuario del lenguaje no debe preocuparse por detalles específicos del almacenamiento ni por el manejo físico del dato. Su atención se debe concentrar solamente en el tratamiento lógico de este último; es decir, en encontrar una estructura de datos que permita resolver ciertos problemas de manera óptima.

Por otra parte, las computadoras no pueden almacenar directamente un arreglo multidimensional. Su representación en memoria debe ser lineal —a cada elemento le sigue un único elemento—, mediante un bloque de posiciones sucesivas.

En este capítulo se estudiarán algunas técnicas utilizadas para el almacenamiento lineal de arreglos multidimensionales.

2.2 ARREGLOS BIDIMENSIONALES

Los lenguajes de programación pueden representar un arreglo bidimensional A, de $m \times n$ elementos, mediante un bloque de $m \times n$ posiciones sucesivas. La distribución de los elementos se puede realizar de dos formas diferentes: renglón a renglón, llamada tam-

bién **ordenación por renglones**, que utilizan la mayoría de los lenguajes de programación, por ejemplo, BASIC, COBOL, PASCAL, C, etcétera, o bien columna a columna, llamada también **ordenación por columnas**, que utiliza FORTRAN.

Sea el arreglo bidimensional *A* de 2 × 3 elementos (fig. 2.1*a*). Su representación en un arreglo unidimensional ordenado por renglones se observa en la figura 2.1*b*, mientras que el que corresponde a un arreglo unidimensional ordenado por columnas se observa en la figura 2.1*c*.

Una vez almacenados los valores de manera lineal se requiere una fórmula que proporcione la posición en el arreglo unidimensional que le corresponde a cada elemento del arreglo bidimensional original.

Sean, entonces, *m* el número de renglones y *n* el número de columnas de un arreglo bidimensional. Por otra parte, *i* y *j* indican el renglón y columna, respectivamente, de la posición del elemento que se quiere ubicar. La fórmula para localizar un elemento determinado, en un arreglo unidimensional ordenado por renglones, es la siguiente:

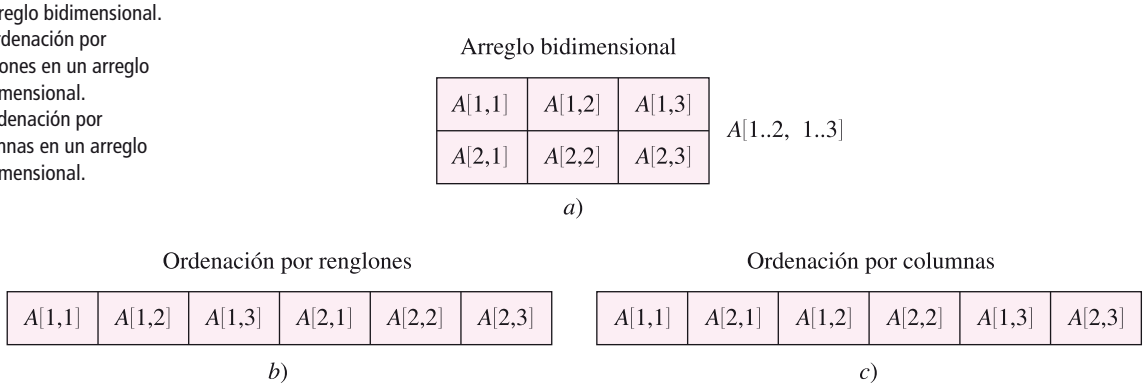
$$\text{LOC}(A[i, j]) = \text{POSINI} + n * (i - 1) + (j - 1) \qquad \textbf{Fórmula 2.1}$$

Donde **POSINI**, primer término de la fórmula 2.1, representa la posición del arreglo unidimensional a partir de la cual se encuentra almacenado el arreglo bidimensional. En general, para llegar a cualquier renglón *i* se deben contabilizar los elementos correspondientes a $(i - 1)$ renglones completos. Este resultado se obtiene mediante la operación $n * (i - 1)$, segundo término de la fórmula 2.1. Cuando se llega al renglón correspondiente se deben contabilizar los $(j - 1)$ elementos necesarios para llegar a la columna *j*, tercer término de la fórmula 2.1. La suma de los tres términos proporciona la localización del elemento *i*, *j* correspondiente, en un arreglo unidimensional ordenado por renglones.

Así, por ejemplo, si deseamos localizar el elemento *A*[2, 1] del arreglo de la figura 2.1*a* hacemos:

$$\text{LOC}(A[2, 1]) = 1 + 3 * (2 - 1) + (1 - 1) = 4$$

FIGURA 2.1
Representación lineal de arreglos bidimensionales. *a*) Arreglo bidimensional. *b*) Ordenación por renglones en un arreglo unidimensional. *c*) Ordenación por columnas en un arreglo unidimensional.



Ahora bien, si el arreglo se encuentra almacenado por columnas, la fórmula para localizar un elemento determinado es:

$$\text{LOC}(A[i, j]) = \text{POSINI} + m * (j - 1) + (i - 1) \qquad \textbf{Fórmula 2.2}$$

En este caso, **POSINI**, primer término de la fórmula 2.2, representa, como en el caso anterior, la posición del arreglo unidimensional a partir de la cual se encuentra almacenado el arreglo bidimensional. En general, para llegar a cualquier columna *j*, primero se deben contabilizar los elementos correspondientes a $(j - 1)$ columnas completas. Este resultado se obtiene con la operación $m * (j - 1)$ segundo término de la fórmula 2.2. Luego que se llega a la columna deseada, se deben considerar los $(i - 1)$ elementos necesarios para llegar al renglón *i,* tercer término de la fórmula 2.2. La suma de los tres términos define la localización del elemento *i, j* correspondiente, en un arreglo unidimensional ordenado por columnas.

Así, por ejemplo, si se desea localizar el elemento $A[1, 3]$ del arreglo presentado en la figura 2.1*a* se hace:

$$\text{LOC}(A[1, 3]) = 1 + 2 * (3 - 1) + (1 - 1) = 5$$

Ejemplo 2.1

Considere el arreglo bidimensional COSTOS de 12 renglones y tres columnas, correspondiente a los costos mensuales de producción de tres departamentos: dulces, conservas y bebidas, de una fábrica. Considere también que aquél se encuentra ordenado por renglones a partir de la posición 180, en un arreglo unidimensional llamado COS. Analice los siguientes casos:

a) Se necesita conocer el costo de producción del departamento de conservas, durante agosto.

Se procede de esta forma:

```
Hacer I ← 8, J ← 2
Escribir COS[180 + 3 * (I − 1) + (J − 1)]
    {El resultado del cálculo es 202}
```

b) Se necesita el costo de producción anual del departamento de bebidas.
Los pasos a seguir son:

```
Hacer SUM ← 0
Repetir con I desde 1 hasta 12
    Hacer SUM ← SUM + COS[180 + 3 * (I − 1) + (3 − 1)]
```

c) Se necesita el costo total de producción de los tres departamentos, durante septiembre.
Los pasos a seguir son:

```
Hacer SUM ← 0
Repetir con J desde 1 hasta 3
    Hacer SUM ← SUM + COS[180 – 3 * (9 − 1) + (J − 1)]
```

Ejemplo 2.2

Supongamos que se tiene almacenado el arreglo bidimensional $A[1..6, 1..4]$ en dos arreglos unidimensionales diferentes. El primero ordenado por renglones a partir de la posición 1, y el segundo ordenado por columnas a partir de la posición 20. Considere los siguientes casos:

a) Se necesita obtener la posición del elemento $A[5, 3]$ en el arreglo unidimensional ordenado por renglones.

Se procede de esta forma:

$$\text{LOC}(A[5,3]) = \underbrace{1}_{\text{POSINI}} + \underbrace{4}_{n} * \underbrace{(5-1)}_{(i-1)} + \underbrace{(3-1)}_{(j-1)} = 17$$

b) Se necesita obtener la posición del elemento $A[4, 3]$ en el arreglo unidimensional ordenado por columnas.

$$\text{LOC}(A[4,3]) = \underbrace{20}_{\text{POSINI}} + \underbrace{6}_{m} * \underbrace{(3-1)}_{(j-1)} + \underbrace{(4-1)}_{(i-1)} = 35$$

2.3 ARREGLOS DE MÁS DE DOS DIMENSIONES

Los lenguajes de programación de alto nivel almacenan un arreglo A de N dimensiones, siendo $N > 2$ y, por tanto, de $m_1 \times m_2 \times ... \times m_n$ elementos, mediante un bloque de $m_1 \times m_2 \times ... \times m_n$ posiciones sucesivas. Esta representación, al igual que en el caso de arreglos bidimensionales, se puede realizar de dos formas diferentes: renglón a renglón, llamada también **ordenación por renglones**, y columna a columna, llamada también **ordenación por columnas**.

Sea A un arreglo tridimensional de $2 \times 3 \times 2$ elementos (fig. 2.2*a*). Su representación en un arreglo unidimensional ordenado por renglones puede observarse en la figura 2.2*b*, mientras que la que corresponde a un arreglo unidimensional ordenado por columnas se observa en la figura 2.2*c*.

Una vez almacenados los valores de manera lineal, se requiere calcular la posición de cualesquiera de los elementos guardados en el arreglo unidimensional. Para ello se necesita primero obtener los índices (K_i), los tamaños de las dimensiones (T_i) y los índices efectivos (IE_i) correspondientes. Cabe aclarar que un índice efectivo (IE_i) se calcula como la diferencia del índice K_i correspondiente y el límite inferior de la dimensión i, donde i varía desde 1 hasta N, siendo N el número de dimensiones del arreglo multidimensional.

La forma de localizar un elemento determinado en un arreglo ordenado por renglones es:

$$\text{LOC}\left(A\left[\begin{matrix}n\\ k_i\\ i=1\end{matrix}\right]\right) = \text{POSINI} + \left(\left(\left(\left(IE_1 * T_2 + IE_2\right) * T_4 + \ldots + IE_{n-1}\right) * T_n + IE_n\right.\right. \qquad \textbf{Fórmula 2.3}$$

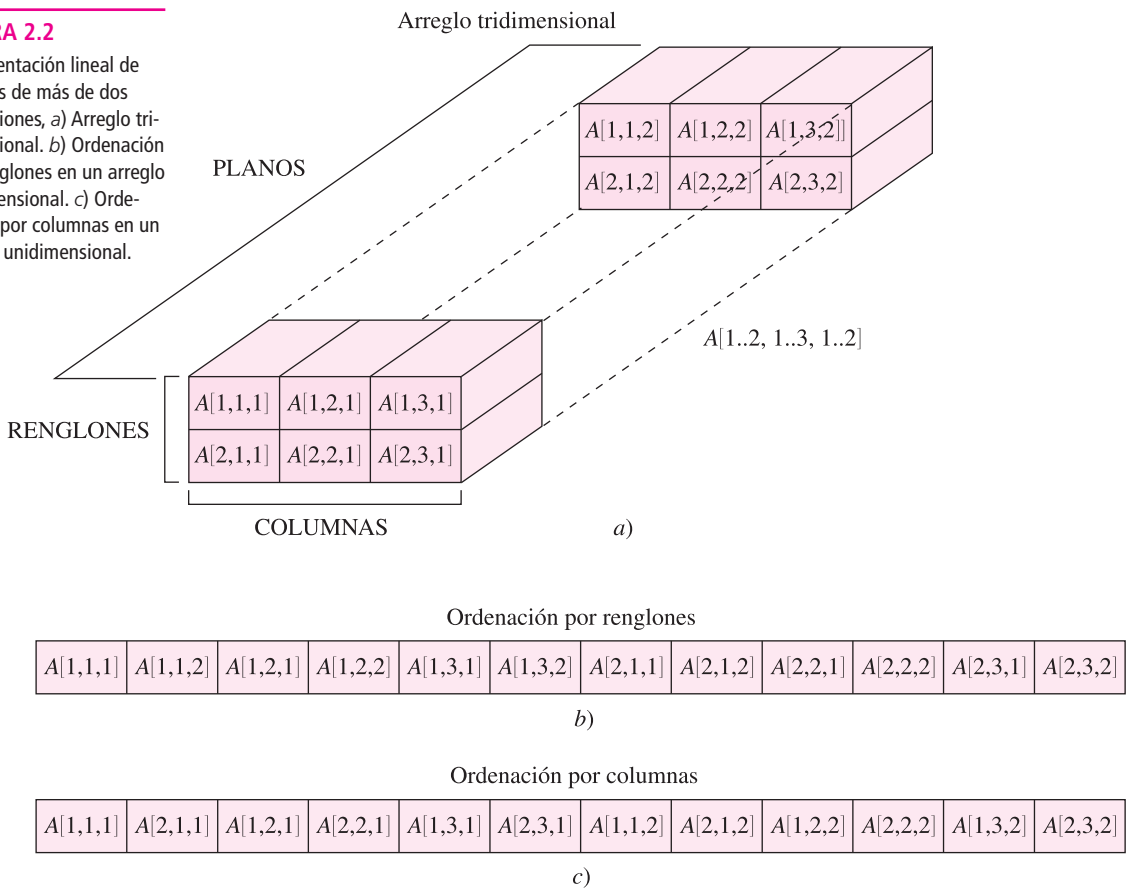


FIGURA 2.2
Representación lineal de arreglos de más de dos dimensiones, *a*) Arreglo tridimensional. *b*) Ordenación por renglones en un arreglo unidimensional. *c*) Ordenación por columnas en un arreglo unidimensional.

Supongamos que se desea localizar el elemento $A[2, 3, 1]$ del arreglo presentado en la figura 2.2*a*. En primer lugar, se realizan los siguientes cálculos para obtener los T_i y los IE_i:

$$T_1 = \text{límsup}_1 - \text{líminf}_1 + 1 = 2 - 1 + 1 = 2$$
$$T_2 = \text{límsup}_2 - \text{líminf}_2 + 1 = 3 - 1 + 1 = 3$$
$$T_3 = \text{límsup}_3 - \text{líminf}_3 + 1 = 2 - 1 + 1 = 2$$

$$IE_1 = K_1 - \text{líminf}_1 = 2 - 1 = 1$$
$$IE_2 = K_2 - \text{líminf}_2 = 3 - 1 = 2$$
$$IE_3 = K_3 - \text{líminf}_3 = 1 - 1 = 0$$

Luego se aplica la fórmula 2.3 para obtener la posición correspondiente del elemento en un arreglo unidimensional ordenado por renglones. Se procede así:

$$\text{LOC}(A[2, 3, 1]) = 1 + ((1 * 3 + 2) * 2 + 0) = 11$$

Para calcular la posición del elemento $A[1, 2, 2]$ se realizan las siguientes operaciones para obtener los T_i —se usan los generados en el caso anterior— y los IE_i:

$T_1 = 2 \quad IE_1 = K_1 - \text{líminf}_1 = 1 - 1 = 0$
$T_2 = 3 \quad IE_2 = K_2 - \text{líminf}_2 = 2 - 1 = 1$
$T_3 = 2 \quad IE_3 = K_3 - \text{líminf}_3 = 2 - 1 = 1$

y luego se aplica la fórmula 2.3 para obtener la posición requerida.

$$\text{LOC}(A[1, 2, 2]) = 1 + ((0 * 3 + 1) * 2 + 1) = 4$$

Ahora bien, si el arreglo se encuentra almacenado por columnas, la forma de localizar un elemento es:

$$\text{LOC}\left(A\left[\begin{matrix}n\\ k_i\\ i=1\end{matrix}\right]\right) = \text{POSINI} + \left(\left(\left(\left(IE_n * T_{n-1} + IE_{n-1}\right) * T_{n-2} + IE_3\right) * T_2 + \ldots + IE_2\right) * T_1 + IE_1\right) \quad \textbf{Fórmula 2.4}$$

Supongamos que se desea encontrar el elemento $A[2, 3, 1]$ del arreglo presentado en la figura 2.2*a*. Se tienen que calcular primero los T_i —se usan los generados anteriormente— y los IE_i correspondientes:

$T_1 = 2 \quad IE_1 = K_1 - \text{líminf}_1 = 2 - 1 = 1$
$T_2 = 3 \quad IE_2 = K_2 - \text{líminf}_2 = 3 - 1 = 2$
$T_3 = 2 \quad IE_3 = K_3 - \text{líminf}_3 = 1 - 1 = 0$

y posteriormente se aplica la fórmula 2.4 para obtener la posición del elemento $A[2, 3, 1]$, en un arreglo unidimensional ordenado por columnas.

$$\text{LOC}(A[2, 3, 1]) = 1 + ((0 * 3 + 2) * 2 + 1) = 6$$

Si se quiere calcular la posición del elemento $A[1, 2, 2]$, se realizan las siguientes operaciones para obtener los T_i —se usan los generados anteriormente— y los IE_i:

$T_1 = 2 \quad IE_1 = K_1 - \text{líminf}_1 = 1 - 1 = 0$
$T_2 = 3 \quad IE_2 = K_2 - \text{líminf}_2 = 2 - 1 = 1$
$T_3 = 2 \quad IE_3 = K_3 - \text{líminf}_3 = 2 - 1 = 1$

y luego se aplica la fórmula 2.4 para obtener la posición requerida.

$$\text{LOC}(A[1, 2, 2]) = 1 + ((1 * 3 + 1) * 2 + 0) = 9$$

Ejemplo 2.3

Considere el arreglo tridimensional COSTOS de 12 × 3 × 5, correspondiente a los costos de producción mensuales en tres departamentos: dulces, conservas y bebidas, de una fábrica, en los últimos cinco años, desde 2001 hasta 2005. Considere también que aquél se encuentra ordenado por renglones a partir de la primera posición en un arreglo unidimensional llamado COS. Analice los siguientes casos:

a) Se necesita la posición en el arreglo COS donde se encuentra el costo de producción del departamento de bebidas, durante agosto y durante el año de 2004.

Se obtienen los T_i y los IE_i correspondientes:

$T_1 = \text{límsup}_1 - \text{líminf}_1 + 1 = 12 - 1 + 1 = 12$
$T_2 = \text{límsup}_2 - \text{líminf}_2 + 1 = 3 - 1 + 1 = 3$
$T_3 = \text{límsup}_3 - \text{líminf}_3 + 1 = 5 - 1 + 1 = 5$

$IE_1 = K_1 - \text{líminf}_1 = 8 - 1 = 7$
$IE_2 = K_2 - \text{líminf}_2 = 3 - 1 = 2$
$IE_3 = K_3 - \text{líminf}_3 = 4 - 1 = 3$

y luego se procede de la siguiente forma:

$$\text{LOC(COSTOS[8, 3, 4])} = 1 + ((7 * 3 + 2) * 5 + 3) = 119$$

b) Se necesita el costo de producción del departamento de conservas, durante 2003.

Se obtienen los T_i —se usan los generados en el inciso *a*)— y los IE_i correspondientes:

$T_1 = 12$ $IE_1 = 1$ desde 0 hasta 11
$T_2 = 3$ $IE_2 = K_2 - \text{liminf}_2 = 2 - 1 = 1$
$T_3 = 5$ $IE_3 = K_3 - \text{liminf}_3 = 3 - 1 = 2$

y luego se realizan los siguientes pasos:

Hacer SUM ← 0
Repetir con *I* desde 0 hasta 11
    Hacer SUM ← SUM + COS[1 + ((*I* * 3 + 1) * 5 + 2)]

c) Se necesita el costo total de producción de los tres departamentos, durante septiembre de 2005.

Se obtienen los T_i —se usan los generados en el inciso *a*)— y los IE_i correspondientes:

$T_1 = 12$ $IE_1 = K_1 - \text{líminf}_1 = 9 - 1 = 8$
$T_2 = 3$ $IE_2 = I$ desde 0 hasta 2
$T_3 = 5$ $IE_3 = K_3 - \text{líminf}_3 = 5 - 1 = 4$

y luego se aplican los siguientes pasos:

Hacer SUM ← 0
Repetir con *I* desde 0 hasta 2
    Hacer SUM ← SUM + COS[1 + ((8 * 3 + *I*) * 5 + 4)]

d) Se necesita el costo total de producción de los tres departamentos durante todo 2004.

Se obtienen los T_i —se usan los generados en el inciso *a*)— *y* los IE_i correspondientes:

$T_1 = 12$ $IE_1 = I$ desde 0 hasta 11
$T_2 = 3$ $IE_2 = J$ desde 0 hasta 2
$T_3 = 5$ $IE_3 = K_3 - \text{líminf}_3 = 4 - 1 = 3$

y luego se aplican los siguientes pasos:

Hacer SUM ← 0
Repetir con *I* desde 0 hasta 11
  Repetir con *J* desde 0 hasta 2
    Hacer SUM ← SUM + COS[1 + ((*I* * 3 + *J*] * 5 + 3)]

Ejemplo 2.4

Supongamos que se tiene almacenado el arreglo de cuatro dimensiones *A*[1..11, 1..5, 1..3, 1..5] en dos arreglos unidimensionales diferentes. El primero ordenado por renglones a partir de la posición 1 y el segundo por columnas a partir de la posición 820. Considere los siguientes casos:

a) Se necesita la posición del elemento *A*[9, 2, 2, 4] en el arreglo unidimensional ordenado por renglones.

Primero se obtienen los T_i y los IE_i correspondientes:

$T_1 = \text{límsup}_1 - \text{líminf}_1 + 1 = 11 - 1 + 1 = 11$
$T_2 = \text{límsup}_2 - \text{líminf}_2 + 1 = 5 - 1 + 1 = 5$
$T_3 = \text{límsup}_3 - \text{líminf}_3 + 1 = 3 - 1 + 1 = 3$
$T_4 = \text{límsup}_4 - \text{líminf}_4 + 1 = 5 - 1 + 1 = 5$

$IE_1 = K_1 - \text{líminf}_1 = 9 - 1 = 8$
$IE_2 = K_2 - \text{líminf}_2 = 2 - 1 = 1$
$IE_3 = K_3 - \text{líminf}_3 = 2 - 1 = 1$
$IE_4 = K_4 - \text{líminf}_4 = 4 - 1 = 3$

y luego se procede de esta manera:

$$\text{LOC}(A[4, 0, 2, 8]) = 1 + (((8 * 5 + 1) * 3 + 1) * 5 + 3) = 624$$

b) Se necesita la posición del elemento *A*[10, 1, 2, 1] en el arreglo unidimensional ordenado por columnas.

Primero se obtienen los T_i —se usan los generados en el inciso *a*)— y los IE_i correspondientes:

$T_1 = 11$ $IE_1 = K_1 - \text{líminf}_1 = 10 - 1 = 9$
$T_2 = 5$ $IE_2 = K_2 - \text{líminf}_2 = 1 - 1 = 0$
$T_3 = 3$ $IE_3 = K_3 - \text{líminf}_3 = 2 - 1 = 1$
$T_4 = 5$ $IE_4 = K_4 - \text{líminf}_4 = 1 - 1 = 0$

y luego se procede así:

$$\text{LOC}(A[5, -1, 2, 5]) = 820 + (((0 * 3 + 1) * 5 + 0) * 11 + 9) = 884$$

2.4 MATRICES POCO DENSAS

Matriz representa un término matemático que se utiliza para indicar un conjunto de elementos organizados por medio de renglones y columnas. Es equivalente al término arreglo bidimensional utilizado en computación. Este término se emplea en esta sección, fundamentalmente porque a los arreglos bidimensionales poco densos se les conoce mucho más como matrices poco densas.

Poco denso índica proporción muy alta de ceros entre los elementos de la matriz. Observe la matriz *A* de 5 × 7 elementos de la figura 2.3.

Es fácil darse cuenta de que esta matriz tiene gran cantidad de ceros. Siendo precisos, 71% de sus elementos son ceros. Piense el lector qué ocurriría si en lugar de tener una matriz de 5 × 7 se tuviera una matriz de 500 × 800 y la mayoría de sus elementos fueran iguales a cero. Con el porcentaje anterior y para este caso en particular, se tendrían 284 000 elementos iguales a cero.

Una situación como ésta exige que se haga un uso más eficiente del espacio de memoria. Con ese propósito existen diversos métodos para almacenar sólo los valores diferentes de cero de una matriz poco densa. A continuación presentamos dos de los más usados.

Arreglo de registros

Se utiliza un arreglo unidimensional, donde cada elemento representa un registro formado por tres campos: uno para guardar el renglón donde se encontró el valor diferente de cero; otro para guardar la columna, y el tercero para guardar el valor del elemento distinto de cero de la matriz.

En la tabla 2.1 se muestra la forma de almacenar los elementos de la matriz poco densa que se observa en la figura 2.3.

TABLA 2.1

Renglón	Columna	Valor
1	1	1
1	5	6
2	3	8
2	6	5
3	3	7
4	1	4
4	2	3
4	7	9
5	3	2
5	7	8

FIGURA 2.3
Matriz de 5 × 7.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 5 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 4 & 3 & 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 2 & 0 & 0 & 0 & 8 \end{bmatrix} \quad A[1..5,\ 1..7]$$

Es pertinente aclarar que puede resultar muy conveniente almacenar el total de renglones y de columnas de la matriz original.

A continuación se presenta un algoritmo muy simple que almacena en un arreglo unidimensional los elementos distintos de cero de una matriz poco densa.

Algoritmo 2.1 Almacena_matriz_poco_densa

Almacena_matriz_poco_densa (MAT)

{El algoritmo almacena los elementos distintos de cero de una matriz poco densa en un arreglo unidimensional. MAT constituye un arreglo unidimensional de registros. Los campos del registro son RENGLÓN, COLUMNA y VALOR}

{*FI*, *CO*, *I*, *J* y *K* son variables de tipo entero. VAL es una variable de tipo real}

1. Leer el número de renglones y de columnas de la matriz (*FI* y *CO*)
2. Hacer MAT[0].RENGLÓN ← *FI*,
MAT[0].COLUMNA ← *CO* y *K* ← 1
3. *Repetir* con *I* desde 1 hasta *FI*
3.1 *Repetir* con *J* desde 1 hasta *CO*
Leer información (VAL)
3.1.1 (*Si* VAL ≠ 0) *entonces*
Hacer MAT[*K*].RENGLÓN ← *I*,
MAT[*K*].COLUMNA ← *J*,
MAT[*K*].VALOR ← VAL ← y *K* ← *K* + 1
3.1.2 {Fin del condicional del paso 3.1.1}
3.2 {Fin del ciclo del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Hacer MAT[0].VALOR ← *K* – 1

{En MAT[0].RENGLÓN y MAT[0].COLUMNA quedan almacenados el número de renglones y de columnas, respectivamente, de la matriz. En MAT[0].VALOR queda almacenado el total de elementos diferentes de cero de la matriz}

Arreglo de listas

Se sugiere que antes de estudiar este método usado para el almacenamiento de matrices poco densas consulte el capítulo 5.

La representación de la matriz poco densa se realiza por medio de un arreglo de listas. Es decir, en el elemento *i* de un arreglo unidimensional se tiene un registro que almacena en uno de sus campos el total de elementos diferentes de cero encontrados en el renglón *i* de la matriz, y en otro campo la dirección al primer nodo de una lista. Cada nodo de la lista almacenará: la columna de la matriz en la que se encuentra el valor diferente de cero, el propio valor y la dirección al siguiente nodo de la lista.

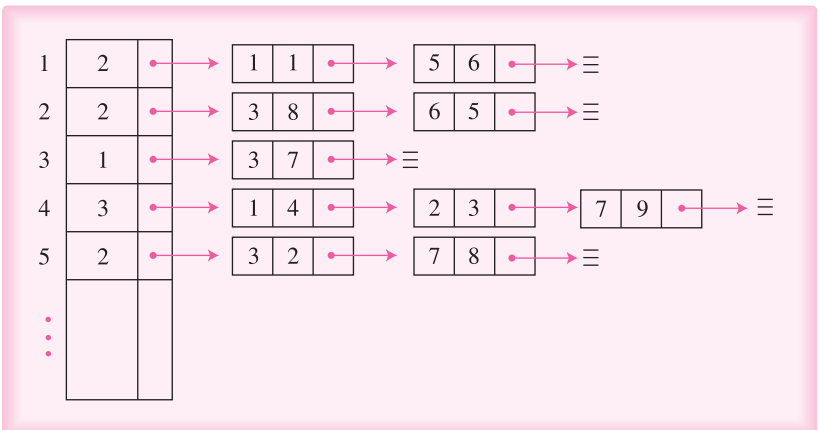


FIGURA 2.4
Arreglo de listas.

En la figura 2.4 se presenta un esquema de esta estructura de datos que se aplica a la matriz poco densa de la figura 2.3.

2.4.1 Matrices cuadradas poco densas

Las **matrices cuadradas** son aquellas que tienen igual número de renglones y de columnas. Si además estas matrices tienen una proporción muy alta de ceros, se denominan **poco densas**.

Por otra parte, las matrices cuadradas en las que los elementos que se encuentran arriba o debajo de la diagonal principal son iguales a cero, se llaman **matrices triangulares**. Éstas, según la ubicación de los ceros, se clasifican en **matriz triangular inferior** si los elementos iguales a cero se encuentran sobre la diagonal principal (fig. 2.5*a*), y en **matriz triangular superior** si los elementos iguales a cero se encuentran debajo de la diagonal principal (fig. 2.5*b*).

2.4.2 Matriz triangular inferior

Supongamos que se desea almacenar en un arreglo unidimensional B (fig. 2.6) la matriz triangular inferior de la figura 2.5*a*.

Es fácil observar que el arreglo B tendrá:

$1 + 2 + 3 + 4 + \ldots + n$

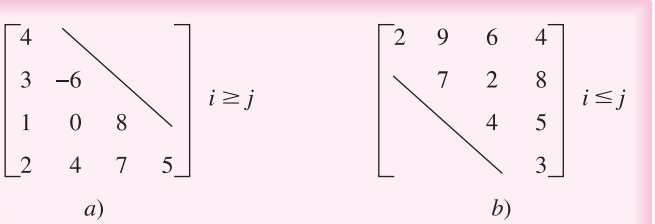


FIGURA 2.5
Matrices cuadradas poco densas. *a*) Matriz triangular inferior. *b*) Matriz triangular superior.

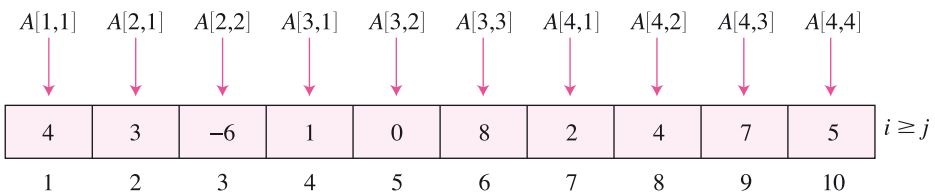


FIGURA 2.6
Almacenamiento de una matriz triangular inferior en un arreglo unidimensional.

elementos, que es igual a:

$$\frac{n*(n+1)}{2}$$

por el principio de inducción matemática. Cabe señalar que en lo que resta de este capítulo, así como en los siguientes, se hará uso de este principio.

Asumiendo que los elementos de la matriz triangular inferior fueron almacenados en un arreglo unidimensional, se requiere encontrar una fórmula que permita localizar a cada uno de ellos. En primer lugar, se debe considerar a **POSINI**, que indica la posición a partir de la cual se encuentra almacenado el arreglo —primer término de la fórmula—. En general, para llegar a cualquier renglón i se deben contabilizar los elementos correspondientes a $(i - 1)$ renglones. Se debe tener en cuenta entonces:

$$1 + 2 + \ldots + (i - 1)$$

elementos, lo que es igual a:

$$\frac{(i-1)*i}{2} \quad \text{(segundo término de la fórmula)}$$

Una vez en el renglón i deseado, se deben contabilizar los $(j - 1)$ elementos necesarios para llegar a la columna j —tercer término de la fórmula—. La suma de los tres términos determina la localización del elemento i, j de la matriz triangular inferior, en un arreglo unidimensional. La fórmula es:

$$\text{LOC}(A[i,j]) = \text{POSINI} + \frac{(i-1)*i}{2} + (j-1) \qquad \textbf{Fórmula 2.5}$$

Así, por ejemplo, si se desea localizar la posición del elemento $A[3, 2]$ del arreglo presentado en la figura 2.5*a* almacenado en un arreglo unidimensional, aplicando la fórmula 2.5 se tiene:

$$\text{LOC}(A[3,2]) = 1 + \frac{(3-1)*3}{2} + (2-1) = 5$$

Si, en cambio, se desea localizar la posición del elemento $A[4, 3]$ del arreglo presentado en la figura 2.5*a*, al aplicar la fórmula 2.5 se tiene:

$$\text{LOC}(A[4,3]) = 1 + \frac{(4-1)*4}{2} + (3-1) = 9$$

2.4.3 Matriz triangular superior

Para el caso de la matriz triangular superior de la figura 2.5*b*, si se almacena en un arreglo unidimensional *B* (fig. 2.7), éste tendrá:

$$n + ... + 4 + 3 + 2 + 1$$

elementos, que es igual a:

$$\frac{n*(n+1)}{2}$$

Se requiere, entonces, de una fórmula para localizar la posición de un elemento de una matriz triangular superior, en un arreglo unidimensional. A diferencia del caso anterior —matriz triangular inferior—, el proceso es más complicado.

En primer lugar, se debe considerar a **POSINI**, que representa la posición a partir de la cual se encuentra almacenado el arreglo —primer término de la fórmula—. En segundo lugar, se deben contabilizar los elementos necesarios para llegar a un renglón *i* cualquiera, esto es, los elementos correspondientes a los $(i - 1)$ renglones anteriores a *i*. Este cálculo se puede realizar en dos partes. Primero se contabilizan los elementos correspondientes a $(i - 1)$ renglones completos: $(n * (i - 1))$ y luego se restan a este resultado los que son ceros en los $(i - 1)$ renglones anteriores a *i*.

Si:

$i = 1 \rightarrow$ tenemos 0 ceros en los renglones anteriores.
$i = 2 \rightarrow$ tenemos 0 ceros en los renglones anteriores.
$i = 3 \rightarrow$ tenemos 1 cero en los renglones anteriores.
$i = 4 \rightarrow$ tenemos 1 + 2 ceros en los renglones anteriores.

En general, podemos afirmar que para $(i - 1)$ renglones se tienen:

$$0 + 0 + 1 + 2 + ... + (i - 2)$$

ceros, que es igual a:

$$\frac{(i-2)*(i-1)}{2}$$

La expresión obtenida en la primera parte menos la fórmula obtenida en la segunda, da como resultado el segundo término de la fórmula 2.6.

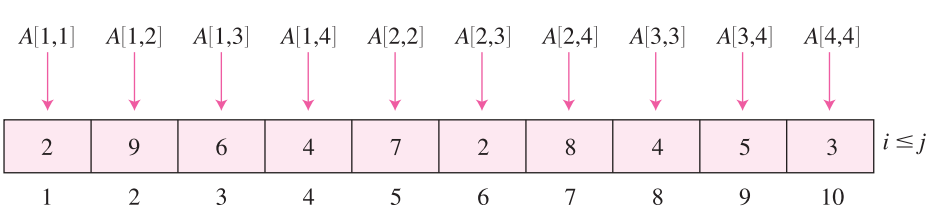


FIGURA 2.7
Almacenamiento de una matriz triangular superior en un arreglo unidimensional.

$$n*(i-1)-\frac{(i-2)*(i-1)}{2}$$

Por último, y una vez en el renglón i deseado, se deben contabilizar los $(j - i)$ elementos necesarios para llegar a la columna j —tercer término de la fórmula—. La suma de los tres términos indica la localización del elemento i, j de la matriz triangular superior, en un arreglo unidimensional. La fórmula es la siguiente:

$$\text{LOC}(A[i,j]) = \text{POSINI} + \left(n*(i-1)-\frac{(i-2)*(i-1)}{2}\right)+(j-i) \qquad \textbf{Fórmula 2.6}$$

Así, por ejemplo, si se desea localizar la posición del elemento $A[2, 3]$, del arreglo presentado en la figura 2.5*b*, en un arreglo unidimensional, se realiza lo siguiente:

$$\text{LOC}([2,3]) = 1+\left(4*(2-1)-\frac{(2-2)*(2-1)}{2}\right)+(3-2)=6$$

Si, en cambio, se desea localizar la posición del elemento $A[3, 4]$ del arreglo presentado en la figura 2.4*b*:

$$\text{LOC}(A[3,4]) = 1+\left(4*(3-1)-\frac{(3-2)*(3-1)}{2}\right)+(4-3)=9$$

Ejemplo 2.5

Supongamos que se tiene una matriz triangular inferior A con la siguiente dimensión $A[1..8, 1..8]$, almacenada en un arreglo unidimensional B a partir de la posición 10. Analice los siguientes casos:

a) Se necesita la posición donde se encuentra almacenado el elemento $A[5, 4]$. Se procede de esta forma:

$$\text{LOC}(A[5,4]) = 10+\frac{(5-1)*5}{2}+(4-1)=10+10+3=23$$

b) Se necesita la posición del elemento $A[6, 3]$. Se procede así:

$$\text{LOC}(A[6,3]) = 10+\frac{(6-1)*6}{2}+(3-1)=10+15+2=27$$

Ejemplo 2.6

Supongamos que se tiene una matriz triangular superior A con la siguiente dimensión $A[1..10, 1..10]$, almacenada en un arreglo unidimensional B a partir de la posición 50.

Analice los siguientes casos:

a) Se necesita la posición del elemento *A*[1, 9]. Se procede de esta forma:

$$\text{LOC}(A[1,9]) = 50 + \left(10*(1-1) - \frac{(1-2)*(1-1)}{2}\right) + (9-1) = 58$$

b) Se necesita la posición donde se encuentra almacenado el elemento *A*[7, 9]. Se procede así:

$$\text{LOC}(A[7,9]) = 50 + \left(10*(7-1) - \frac{(7-2)*(7-1)}{2}\right) + (9-7) = 97$$

2.4.4 Matriz tridiagonal

Se dice que una matriz es **tridiagonal** si los elementos distintos de cero se encuentran localizados en la diagonal principal y en las diagonales por encima y por debajo de ésta. Por tanto, el valor absoluto del índice *i* menos el índice *j* será menor o igual que 1. En la figura 2.8 el lector puede observar una matriz tridiagonal.

Supongamos que se desea almacenar en un arreglo unidimensional *B* (figura 2.9) la matriz tridiagonal presentada anteriormente.

Es fácil observar que el arreglo *A* tiene *n* elementos en la diagonal principal y (*n* – 1) elementos en las diagonales por encima y debajo de ésta. Por tanto, el número de elementos de una matriz tridiagonal se calcula como:

$$n + 2 * (n - 1)$$

Al hacer las operaciones queda:

$$3 * n - 2$$

Con el propósito de localizar la posición de un elemento de la matriz tridiagonal, almacenada en un arreglo unidimensional, se requiere de una fórmula. En primer lugar, se debe considerar a **POSINI**, que indica la posición inicial, a partir de la cual se encuentra almacenado el arreglo —primer término de la fórmula—. En segundo lugar, se deben contabilizar los elementos necesarios para llegar a un renglón *i* cualquiera; esto es, los elementos correspondientes a los (*i* – 1) renglones anteriores. Se calcula como la suma de los elementos de la primera fila (2), más tres elementos por (*i* – 2) renglones. Por tanto, el segundo término de la fórmula es:

$$2 + 3 * (i - 2)$$

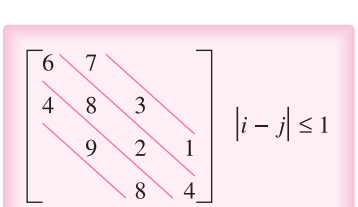


FIGURA 2.8
Matriz tridiagonal.

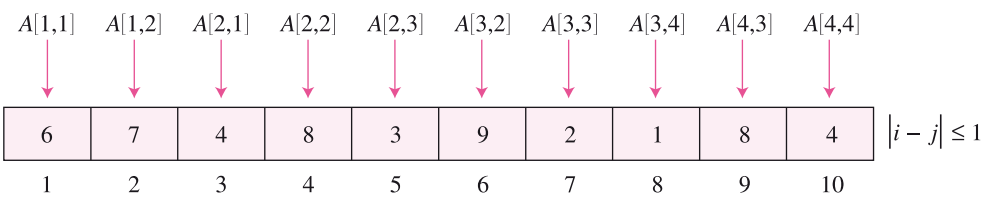


FIGURA 2.9
Almacenamiento de una matriz tridiagonal en un arreglo unidimensional.

Por último, y una vez en el renglón i deseado, se deben contabilizar los elementos correspondientes a las columnas. Con este fin se sigue el siguiente criterio.

$i > j \rightarrow$ no se tiene que contabilizar ningún elemento.
$i = j \rightarrow$ se debe contabilizar un elemento.
$i < j \rightarrow$ se tienen que contabilizar dos elementos.

Con lo cual se obtiene la siguiente expresión:

$(j - i + 1)$ tercer término de la fórmula

Observe que esta expresión contempla los tres casos enunciados. Si:

$i > j$, a lo sumo lo será en una unidad; por tanto, la expresión da cero elementos.
$i = j$, los mismos se anulan, quedando un elemento.
$i < j$, a lo sumo lo será en una unidad; por tanto, la expresión da dos elementos.

Nuevamente la suma de los tres términos da la localización del elemento i, j de la matriz tridiagonal en un arreglo unidimensional. Por tanto:

$$\text{LOC}(A[i, j]) = \text{POSINI} + (2 + 3 * (i - 2)) + (j - i + 1)$$

operando queda:

$$\text{LOC}(A[i, j]) = \text{POSINI} + 2i + j - 3 \qquad \textbf{Fórmula 2.7}$$

Así, por ejemplo, si se desea localizar la posición del elemento $A[3, 3]$, del arreglo presentado en la figura 2.8, en un arreglo unidimensional se hace:

$$\text{LOC}(A[3, 3]) = 1 + 2 * 3 + 3 - 3 = 7$$

Si, en cambio, se desea localizar la posición del elemento $A[4, 3]$ del arreglo presentado en la misma figura, se hace:

$$\text{LOC}(A[4, 3]) = 1 + 2 * 4 + 3 - 3 = 9$$

Se debe tener en cuenta, además, que si el renglón que se evalúa es igual a 1, se puede aplicar la siguiente fórmula:

$$\text{LOC}\left(A\left[\underset{i=1}{i,j}\right]\right) = 1 + (j-1) \qquad \textbf{Fórmula 2.8}$$

Es importante aclarar que la fórmula anterior (2.7), aunque un poco más larga, también funciona para este caso.

Ejemplo 2.7

Supongamos que se tiene una matriz tridiagonal A con la dimensión $A[1..8, 1..8]$, almacenada en un arreglo unidimensional B a partir de la posición 40. Analicemos los siguientes casos:

a) Se necesita la posición donde se encuentra el elemento $A[6, 7]$. Se procede así:

$$\text{LOC}([6, 7]) = 40 + 2 * 6 + 7 - 3 = 56$$

b) Si se necesita la posición donde se encuentra el elemento $A[3, 2]$, se procede de esta forma:

$$\text{LOC}([3, 2]) = 40 + 2 * 3 + 2 - 3 = 45$$

2.4.5 Matrices simétricas y antisimétricas

Una matriz A de $n \times n$ elementos es **simétrica** si $A[i, j]$ es igual a $A[j, i]$, y esto último se cumple para todo i y para todo j. En la figura 2.10 se presentan dos ejemplos de matrices simétricas.

Por otra parte, una matriz A de $n \times n$ elementos es **antisimétrica** si $A[i, j]$ es igual a $-A[j, i]$, y lo anterior se cumple para todo i y para todo j; considerando a $i \neq j$. En la figura 2.11 se observan dos ejemplos de matrices antisimétricas.

Supongamos ahora que se desea almacenar en un arreglo unidimensional B la matriz simétrica de la figura 2.10*a*. Esto último se puede hacer al almacenar únicamente los elementos de la matriz triangular inferior o superior. En la figura 2.12 se presenta un arreglo unidimensional B que almacena la matriz triangular inferior de la matriz simétrica mostrada en la figura 2.10*a*.

Para localizar cualquier elemento de la matriz simétrica se debe aplicar la fórmula 2.5, presentada anteriormente, para matriz triangular inferior. Cabe aclarar que si en un

FIGURA 2.10
Matrices simétricas. *a*) Matriz simétrica de 4 × 4. *b*) Matriz simétrica de 5 × 5.

$$\begin{bmatrix} 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 6 \\ 0 & 0 & 8 & 9 \\ 0 & 6 & 9 & 0 \end{bmatrix} \qquad a)$$

$$\begin{bmatrix} 8 & 5 & 0 & 0 & 3 \\ 5 & 6 & 3 & 0 & 9 \\ 0 & 3 & 0 & 7 & 1 \\ 0 & 0 & 7 & 8 & 0 \\ 3 & 9 & 1 & 0 & 0 \end{bmatrix} \qquad b)$$

FIGURA 2.11
Matrices antisimétricas. *a*) Matriz antisimétrica de 4 × 4. *b*) Matriz antisimétrica de 5 × 5.

$$\begin{bmatrix} 0 & 4 & 0 & 0 \\ -4 & 0 & 0 & 6 \\ 0 & 0 & 8 & 9 \\ 0 & -6 & -9 & 0 \end{bmatrix} \quad a)$$

$$\begin{bmatrix} 8 & -5 & 0 & 0 & 3 \\ 5 & 6 & -3 & 0 & 9 \\ 0 & 3 & 0 & 7 & 1 \\ 0 & 0 & -7 & 8 & 0 \\ -3 & -9 & -1 & 0 & 0 \end{bmatrix} \quad b)$$

determinado momento se necesitara localizar un elemento de la matriz simétrica tal que el índice *j* sea mayor que el índice *i*, se necesitarían invertir los mismos y aplicar posteriormente la misma fórmula. Por ejemplo, si se desea localizar el elemento *A*[1, 2], se tendrá que buscar el elemento *A*[2, 1].

Si ahora se desea almacenar en un arreglo unidimensional *B* los elementos de la matriz antisimétrica de la figura 2.11*a*, se procede de manera similar que en el caso anterior. Se almacenan en un arreglo unidimensional solamente los elementos de la matriz triangular inferior o superior. En la figura 2.13 se presenta un arreglo unidimensional *B* que almacena la matriz triangular superior de la matriz antisimétrica de la figura 2.10*a*.

Para localizar, en este caso, un elemento de la matriz antisimétrica, se debe aplicar la fórmula 2.6 para matriz triangular superior. Es importante señalar que si se tuviera que localizar un elemento de la matriz antisimétrica, tal que el índice *i* sea mayor que el índice *j*, se necesitaría invertir los mismos y aplicar la misma fórmula. Posteriormente, el contenido de la celda *i*, *j* se debe multiplicar por –1. Por ejemplo, si nos interesa localizar el elemento *A*[3, 1], se buscará la posición del elemento *A*[1, 3] y el contenido de dicha posición se multiplicará por –1.

FIGURA 2.12
Almacenamiento de una matriz simétrica en un arreglo unidimensional.

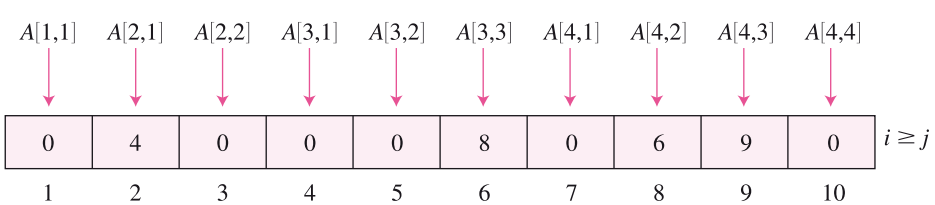
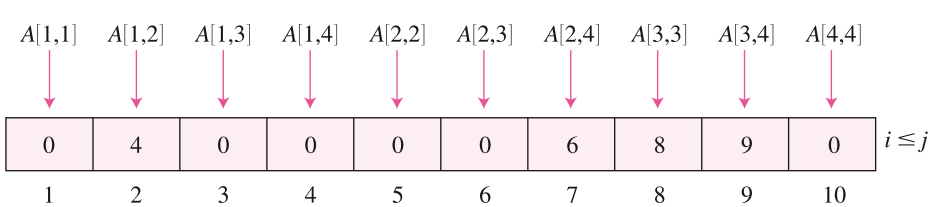


FIGURA 2.13
Almacenamiento de una matriz antisimétrica en un arreglo unidimensional.



EJERCICIOS

Arreglos multidimensionales

1. Considere que el arreglo bidimensional *A*[1..7, 1..7] se encuentra almacenado renglón por renglón en el arreglo unidimensional VEC, a partir de la posición 1. Considere, además, que el arreglo bidimensional *B*[1..9, 1..4] también se encuentra almacenado en el arreglo VEC, columna a columna, a partir de la posición 100. Calcule lo siguiente:

a) La posición del elemento *A*[2, 6] en el arreglo VEC.
b) La posición del elemento *A*[5, 7] en el arreglo VEC.
c) La posición del elemento *B*[8, 1] en el arreglo VEC.
d) La posición del elemento *B*[3, 3] en el arreglo VEC.

2. Considere los arreglos multidimensionales *AM* y *BM* declarados de la siguiente forma:

AM[1..5, 1..5, 1..15] y *BM*[1..8, 1..6, 1..8, 1..4]

AM está almacenado renglón por renglón en el arreglo unidimensional VEAM, a partir de la primera posición. *BM* está almacenado, columna a columna, en el arreglo unidimensional VEBM, a partir de la posición 65. Calcule lo siguiente:

a) La posición del elemento *AM*[5, 3, 10] en el arreglo VEAM.
b) La posición del elemento *AM*[1, 2, 5] en el arreglo VEAM.
c) La posición del elemento *BM*[3, 6, 4, 2] en el arreglo VEBM.
d) La posición del elemento *BM*[6, 5, 8, 1] en el arreglo VEBM.

3. Consideremos que los arreglos bidimensionales *A* y *B* de $m \times n$ elementos se encuentran almacenados en un arreglo unidimensional VEC de $2 \times m \times n$ elementos. *A* está almacenado renglón por renglón a partir de la primera posición. *B* está almacenado columna a columna a partir de la posición $(m * n) + 1$. Escriba un programa que realice lo siguiente:

a) Obtenga la suma de los arreglos bidimensionales almacenados en VEC y almacénela en el arreglo unidimensional SUM, ordenado por renglones, a partir de la primera posición.
b) Imprima el resultado de la suma, almacenado en SUM, en forma de matriz.

4. Sean los arreglos bidimensionales *A* y *B* de $m \times n$ y $n \times p$ elementos, respectivamente, ambos almacenados en un arreglo unidimensional VEC. *A* está almacenado renglón por renglón a partir de la posición 1 y *B* también se encuentra ordenado por renglones a partir de la posición $(m * n) + 1$. Escriba un procedimiento que realice lo siguiente:

a) Obtenga el producto de los arreglos bidimensionales *A* y *B* almacenados en VEC y guarde el resultado en el mismo arreglo, columna a columna, a partir de la posición $(m * n) + (n * p) + 1$.
b) Imprima el resultado del producto, almacenado en VEC, en forma de matriz.

5. Sea CAL un arreglo bidimensional de 30×6 correspondiente a las calificaciones de 30 alumnos en seis exámenes diferentes. CAL se encuentra almacenado renglón por renglón, a partir de la posición 185, en el arreglo unidimensional UNI. Escriba un procedimiento que obtenga lo siguiente:

a) El promedio de calificaciones de los 30 alumnos en los seis exámenes.
b) El alumno que obtuvo la mayor calificación en el tercer examen. Observe que puede haber más de un alumno con la más alta calificación.
c) El examen en el que el promedio de los 30 alumnos fue el más alto.

6. Considere el arreglo tridimensional ATRI de $m \times n \times p$ elementos almacenados, columna a columna, en el arreglo unidimensional AUNI a partir de la primera posición. Escriba un procedimiento que imprima lo siguiente:

a) El renglón y la columna donde se encuentran elementos nulos (usted debe trabajar con el arreglo AUNI).
b) El total de elementos nulos.
c) El porcentaje de elementos nulos, con respecto al número total de elementos del arreglo tridimensional.

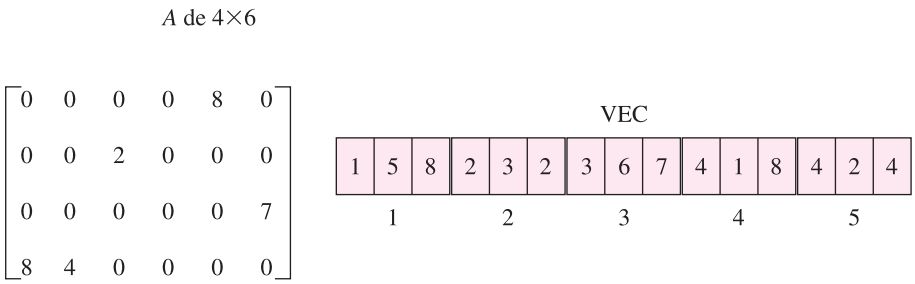
7. Los elementos de un arreglo bidimensional de 4×4 elementos se almacenaron, renglón por renglón, en un arreglo unidimensional *A*. Escriba un subprograma que:

a) Intercambie los elementos del renglón 1 con los del renglón *N* y los del renglón 2 con los del renglón $(N - 1)$, y así sucesivamente.
b) Imprima el arreglo resultante, en forma de matriz.

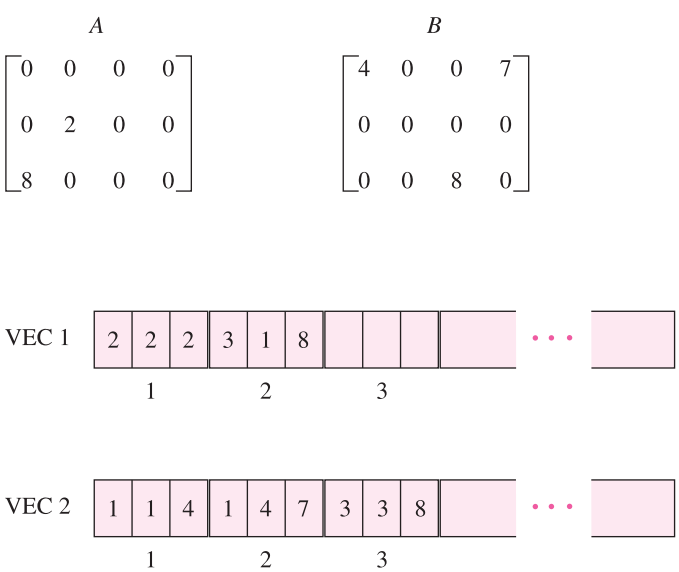
Matrices poco densas

8. Sea VEC un arreglo unidimensional que almacena los elementos distintos de cero de una matriz poco densa *A* de cuatro renglones y seis columnas. Cada elemento del arreglo VEC es un registro que contiene el renglón, la columna y el valor distinto de cero de la matriz. Escriba subprogramas que realicen lo siguiente:

a) Determine el valor del elemento *i*, *j* de la matriz. Tome en cuenta que la búsqueda a realizar en el arreglo VEC debe ser óptima.
b) Imprima la matriz poco densa *A*, a partir del arreglo VEC.



9. Supongamos que existen dos matrices poco densas *A* y *B* de 3 × 4 elementos, que tienen almacenados sus valores distintos de cero en los arreglos unidimensionales VEC1 y VEC2, respectivamente. Ejemplo:



Escriba un subprograma que obtenga la suma de dichas matrices poco densas, utilizando solamente los arreglos VEC1 y VEC2, y almacene el resultado —considere solamente los elementos distintos de cero— en el arreglo unidimensional VEC3.

10. Considere las matrices poco densas *A* y *B*, declaradas de la siguiente forma:

$$A[1..6, 1..3] \text{ y } B[1..3, 1..4]$$

A se encuentra almacenada en el arreglo unidimensional VEC1 y *B* en el arreglo unidimensional VEC2. Escriba un subprograma que obtenga el producto de *A* y *B* —utilizando solamente los arreglos VEC1 y VEC2— y almacene el resultado, columna a columna, en el arreglo unidimensional VEC3.

11. Se tienen los datos de la producción agrícola, por tipo de cultivo —en total 10—, de las 32 entidades del país. No todos los estados tienen todos los cultivos. En aquellos casos en los cuales un estado no cultiva ciertos productos, habrá ceros en las posiciones correspondientes. Escriba un programa que:

a) Almacene los datos diferentes de cero, en un arreglo unidimensional.
b) Encuentre el estado que obtuvo mayor producción agrícola, considerando todos los cultivos.
c) Encuentre el producto, si existiera, que se cultiva en todos los estados.
d) Encuentre el estado, si existiera, que cultiva todos los productos.
e) Encuentre el estado, si existiera, que cultiva sólo los cultivos de tipos 3 y 6.

12. Las matrices cuadradas poco densas *A* y *B* de orden 4 fueron almacenadas en el arreglo unidimensional UNI a partir de la primera y decimoprimera posición, en forma respectiva. De la matriz *A* solamente se almacenaron los elementos correspondientes a la matriz triangular inferior. De la matriz *B*, en cambio, se almacenaron sólo los elementos de la matriz triangular superior. Escriba un subprograma que sume dichas matrices y almacene el resultado, columna a columna, en el arreglo unidimensional SUMA a partir de la primera posición.

Matrices simétricas y antisimétricas

13. Las matrices simétricas *A* y *B* de dimensión 6 fueron almacenadas en un arreglo unidimensional VEC a partir de las posiciones 1 y 22, respectivamente. Sólo se almacenaron los elementos pertenecientes a la matriz triangular inferior. Escriba un subprograma que sume dichas matrices y almacene el resultado, renglón por renglón, en el arreglo SUMSIM a partir de la posición 32.

14. La matriz simétrica *A* de dimensión 5 y la matriz antisimétrica *B*, de la misma dimensión, fueron almacenadas en el arreglo unidimensional VEC a partir de las posiciones 1 y 16, respectivamente. De la matriz *A* solamente se almacenaron los elementos correspondientes a la matriz triangular inferior. De la matriz *B* sólo se almacenaron los elementos de la matriz triangular superior. Escriba un subprograma que obtenga la suma de dichas matrices y almacene el resultado, renglón por renglón, en el arreglo unidimensional SUMA a partir de la primera posición.

15. Una persona tiene que viajar de una ciudad a otra, vía terrestre, en la República mexicana y desea realizar el recorrido en el menor tiempo posible. Los datos referentes a los tiempos entre ciudades se encuentran dados de la siguiente forma:

$$\begin{bmatrix} 0 & & & \\ \text{TIEMPO}_{2,1} & 0 & & \\ \text{TIEMPO}_{3,1} & \text{TIEMPO}_{3,2} & 0 & \\ \text{TIEMPO}_{n,1} & \text{TIEMPO}_{n,2} & \text{TIEMPO}_{n,n-1} & 0 \end{bmatrix}$$

Puede suceder que entre dos ciudades no exista una carretera directa y, por tanto, el tiempo entre ambas sea representado como 0. Sin embargo, es posible llegar a una ciudad intermedia y desde ahí trasladarse hasta la ciudad destino. Por ejemplo, si de la ciudad 3 a la 2 no hay carretera directa, se podría ir primero a la ciudad 1 —si existe carretera entre la 3 y la 1— y luego de la ciudad 1 a la 2 —si entre ellas existen carreteras—. Escriba un programa que realice lo siguiente:

a) Lea los tiempos entre las distintas ciudades y las almacene en un arreglo unidimensional.
b) Lea la ciudad origen y la ciudad destino.
c) Determine el menor tiempo de traslado entre dichas ciudades.
d) Presente la ruta a seguir.

16. Se tiene información sobre costos de boletos aéreos entre N ciudades del país. El costo del boleto para ir de la ciudad i a la ciudad j es igual al costo del boleto para ir de la ciudad j a la i. Por tanto, se puede ahorrar espacio de memoria —recuerde lo visto sobre matrices simétricas— utilizando un arreglo unidimensional para almacenar todos los costos. Escriba un programa que:

a) Lea el número de ciudades.
b) Lea los costos y los almacene usando un arreglo unidimensional.
c) Dado el número de una ciudad origen y de una ciudad destino, imprima el costo del boleto correspondiente.
d) Dado el número de una ciudad, imprima los números de todas las ciudades a las que hay vuelo, partiendo de la ciudad específica.

Capítulo 3

PILAS Y COLAS

3.1 INTRODUCCIÓN

Cuando se presentaron los arreglos, en el capítulo 1, se mencionó que eran estructuras lineales. Es decir, cada componente tiene un único sucesor y un único predecesor con excepción del primero y del último, respectivamente. Por otra parte, al analizar las operaciones de inserción y eliminación, se observó que los elementos se podían insertar o eliminar en cualquier posición del arreglo. Cabe señalar, sin embargo, que existen problemas que por su naturaleza requieren que los elementos se agreguen o se quiten sólo por un extremo. Este capítulo se dedica al estudio de **pilas** y **colas**, que son estructuras de datos lineales con restricciones en cuanto a la posición en la cual se pueden llevar a cabo las operaciones de inserción y eliminación de componentes.

3.2 PILAS

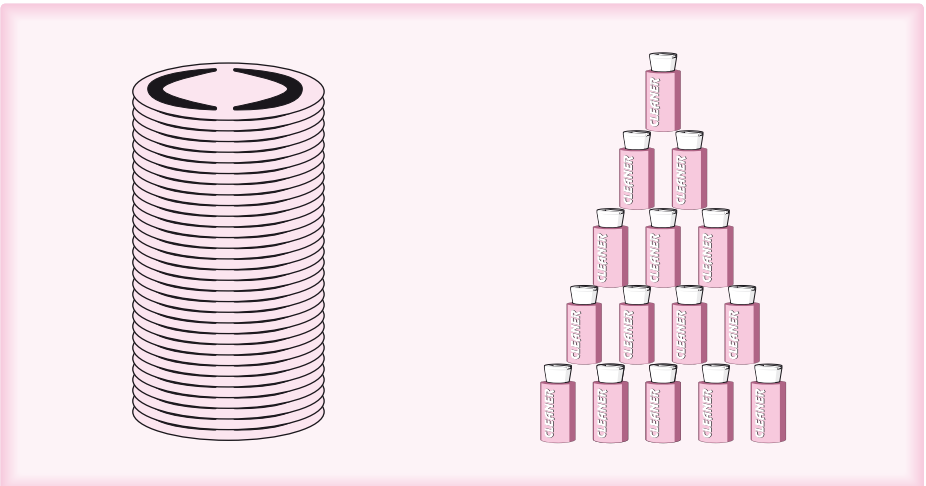
Una **pila** representa una estructura lineal de datos en la que se puede agregar o quitar elementos únicamente por uno de los dos extremos. En consecuencia, los elementos de una pila se eliminan en orden inverso al que se insertaron; es decir, el último elemento que se mete en la pila es el primero que se saca. Debido a esta característica, se le conoce como estructura **LIFO** (*Last-Input, First-Output:* el último en entrar es el primero en salir).

Existen numerosos casos prácticos en los que se utiliza el concepto de pila; por ejemplo, una pila de platos, una pila de latas en un supermercado, una pila de libros que se exhiben en una librería, etcétera. En la figura 3.1 se observa una pila de platos. Es de suponer que si el cocinero necesita un plato limpio, tomará el que está encima de todos, que es el último que se colocó en la pila.

Las **pilas** son estructuras de datos lineales, como los arreglos, ya que los componentes ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un único sucesor y un único predecesor, con excepción del último y del primero, respectivamente.

Una **pila** se define formalmente como una colección de datos a los cuales se puede acceder mediante un extremo, que se conoce generalmente como tope.

FIGURA 3.1
Ejemplos prácticos de pilas.



3.2.1 Representación de pilas

Las pilas no son estructuras fundamentales de datos; es decir, no están definidas como tales en los lenguajes de programación. Para su representación requieren el uso de otras estructuras de datos, como:

- Arreglos
- Listas

En este libro se utilizarán arreglos. En consecuencia, es importante definir el tamaño máximo de la pila, así como una variable auxiliar a la que se denomina TOPE. Esta variable se utiliza para indicar el último elemento que se insertó en la pila. En la figura 3.2 se presentan dos alternativas de representación de una pila, utilizando arreglos.

FIGURA 3.2
Representación de pilas.

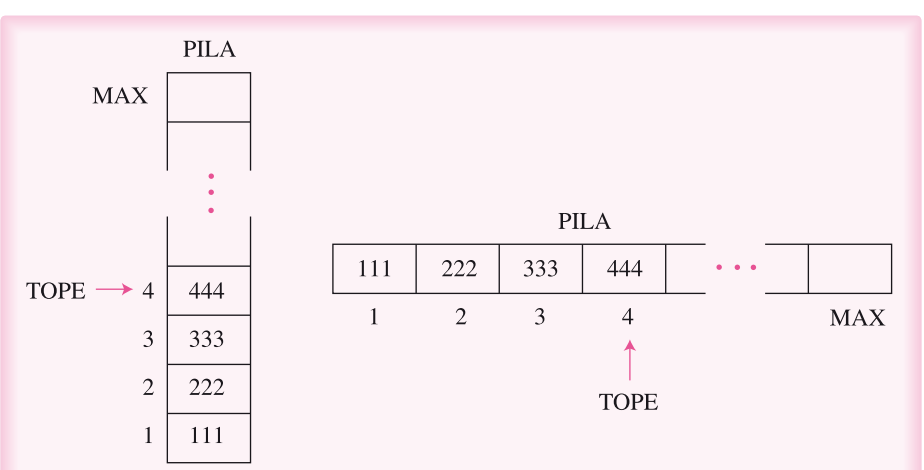
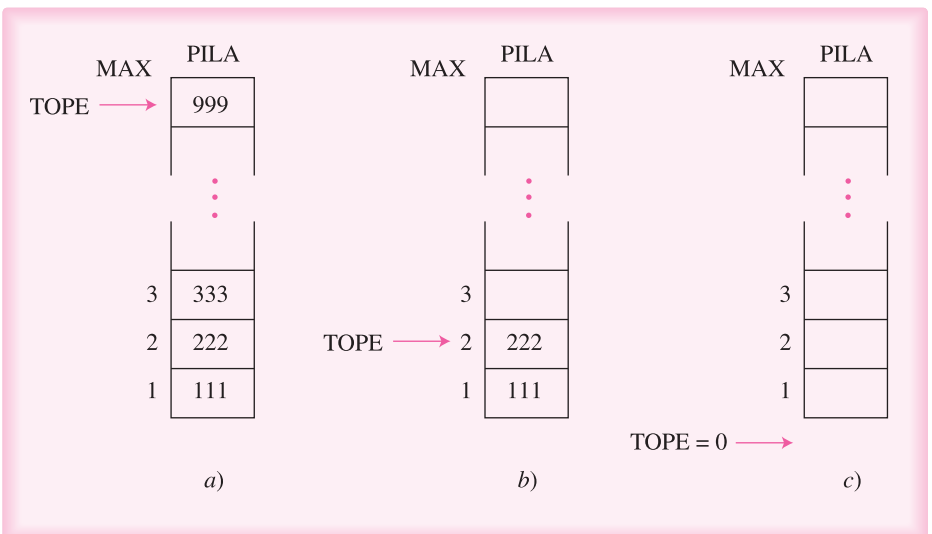


FIGURA 3.3
Representación de pilas. *a*) Pila llena. *b*) Pila con algunos elementos. *c*) Pila vacía.



En la figura 3.3 se presentan ejemplos de *a*) pila llena, *b*) pila con algunos elementos y *c*) pila vacía.

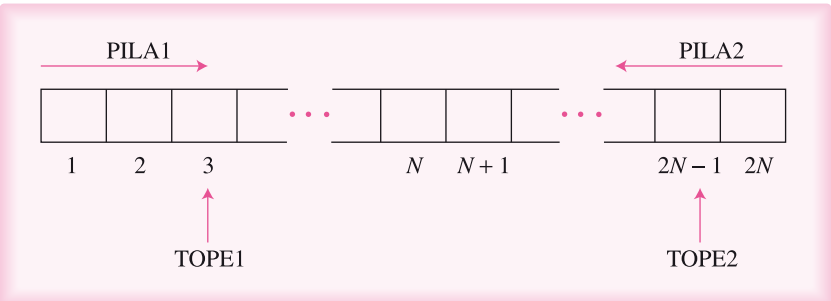
Al utilizar arreglos para implementar pilas se tiene la limitación de que se debe reservar espacio de memoria con anticipación, característica propia de los arreglos. Una vez dado un máximo de capacidad a la pila no es posible insertar un número de elementos mayor al máximo establecido. Si la pila estuviera llena y se intentara insertar un nuevo elemento, se producirá un error conocido como **desbordamiento** —*overflow*—. Por ejemplo, si en la pila que se presenta en la figura 3.3*a*, donde TOPE = MAX, se quisiera insertar un nuevo elemento, se producirá un error de este tipo. La pila está llena y el espacio de memoria reservado es fijo, no se puede expandir ni contraer.

Una posible solución a este tipo de inconvenientes consiste en definir pilas de gran tamaño, pero esto último resultaría ineficiente y costoso si sólo se utilizaran algunos elementos. No siempre es viable saber con exactitud cuál es el número de elementos a tratar; por tanto, siempre existe la posibilidad de cometer un error de desbordamiento —si se reserva menos espacio del que efectivamente se usará— o bien de hacer uso ineficiente de la memoria —si se reserva más espacio del que realmente se necesita—.

Existe otra alternativa de solución a este problema. Consiste en usar **espacios compartidos** de memoria para la implementación de pilas. Supongamos que se necesitan dos pilas, cada una de ellas con un tamaño máximo de *N* elementos. Se definirá entonces un solo arreglo unidimensional de 2 * *N* elementos, en lugar de dos arreglos de *N* elementos cada uno.

Como se ilustra en la figura 3.4, la PILA1 ocupará desde la posición 1 en adelante (2, 3, ...), mientras que la PILA2 ocupará desde la posición 2**N* hacia atrás (2**N* – 1, 2**N* – 2, ...). Si en algún punto del proceso la PILA1 necesitara más espacio del que realmente tiene —*N*— y en ese momento la PILA2 no tuviera ocupados sus *N* lugares, entonces sería posible agregar elementos a la PILA1 sin caer en un error de desbordamiento (figura 3.5*)*. Algo similar podría suceder para la PILA2, si ésta necesitara más de *N* espacios y la PILA1 tuviera lugares disponibles (figura 3.5*b)*.

FIGURA 3.4
Representación de pilas en espacios compartidos.



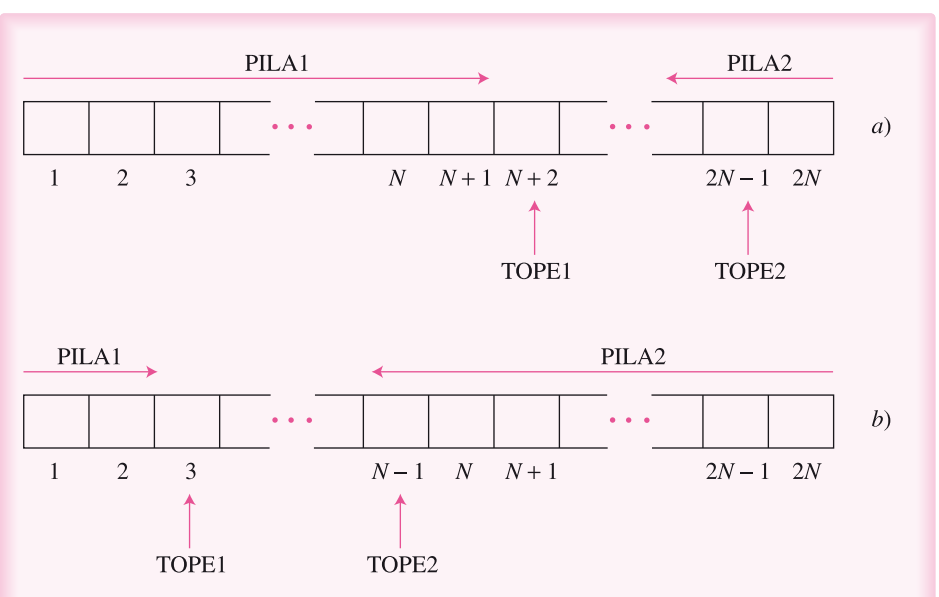
Otro error que se puede presentar al trabajar con pilas es tratar de eliminar un elemento de una pila vacía. Este tipo de error se conoce cómo **subdesbordamiento** —*underflow*—. Por ejemplo, si en la pila que se presenta en la figura 3.3*c*, donde TOPE < 1, se deseara eliminar un elemento, se presentaría un error de este tipo.

3.2.2 Operaciones con pilas

La definición de una estructura de datos queda completa al incluir las operaciones que se pueden realizar en ella. Para el caso de las pilas, las operaciones básicas que se pueden llevar a cabo son:

- Insertar un elemento —*Push*— en la pila
- Eliminar un elemento —*Pop*— de la pila

FIGURA 3.5
Representación de pilas en espacios compartidos. *a*) PILA1 tiene más de *N* elementos y PILA2 tiene menos de *N* elementos. *b*) PILA2 tiene más de *N* elementos y PILA1 tiene menos de *N* elementos.



Y las operaciones auxiliares:

- Pila_vacía
- Pila_llena

Considerando que se tiene una pila con capacidad para almacenar un número máximo de elementos —MAX—, y que el último de ellos se indica con TOPE, a continuación se presentan los algoritmos correspondientes a las operaciones mencionadas. Si la pila está vacía, entonces TOPE es igual a 0.

Algoritmo 3.1 Pila_vacía

Pila_vacía (PILA, TOPE, BAND)

{Este algoritmo verifica si una estructura tipo pila —PILA— está vacía, asignando a BAND el valor de verdad correspondiente. La pila se implementa en un arreglo unidimensional. TOPE es un parámetro de tipo entero. BAND es un parámetro de tipo booleano}

1. *Si* (TOPE = 0) {Verifica si no hay elementos almacenados en la pila}
    entonces
        Hacer BAND ← VERDADERO {La pila está vacía}
    si no
        Hacer BAND ← FALSO {La pila no está vacía}
2. {Fin del condicional del paso 1}

Algoritmo 3.2 Pila_llena

Pila_llena (PILA, TOPE, MAX, BAND)

{Este algoritmo verifica si una estructura tipo pila —PILA— está llena, asignando a BAND el valor de verdad correspondiente. La pila se implementa en un arreglo unidimensional de MAX elementos. TOPE es un parámetro de tipo entero. BAND es un parámetro de tipo booleano}

1. *Si* (TOPE = MAX)
    entonces
        Hacer BAND ← VERDADERO {La pila está llena}
    si no
        Hacer BAND ← FALSO {La pila no está llena}
2. {Fin del condicional del paso 1}

Algoritmo 3.3 Pone

Pone (PILA, TOPE, MAX, DATO)

{Este algoritmo agrega el elemento DATO en una estructura tipo pila —PILA—, si la misma no está llena. Actualiza el valor de TOPE. MAX representa el número máximo de elementos que puede almacenar PILA. TOPE es un parámetro de tipo entero}

1. Llamar a Pila_llena con PILA, TOPE, MAX y BAND
2. *Si* (BAND = VERDADERO)
   entonces
   Escribir "Desbordamiento – Pila llena"
   si no
   Hacer TOPE ← TOPE + 1 y PILA[TOPE] ← DATO
   {Actualiza TOPE e inserta el nuevo elemento en el TOPE de PILA}
3. {Fin del condicional del paso 2}

Algoritmo 3.4 Quita

Quita (PILA, TOPE, DATO)

{Este algoritmo saca un elemento —DATO— de una estructura tipo pila —PILA—, si ésta no se encuentra vacía. El elemento que se elimina es el que se encuentra en la posición indicada por TOPE}

1. Llamar a Pila_vacía con PILA, TOPE y BAND
2. *Si* (BAND = VERDADERO)
   entonces
   Escribir "Subdesbordamiento – Pila vacía"
   si no
   Hacer DATO ← PILA [TOPE] y TOPE ← TOPE – 1 {Actualiza TOPE}
3. {Fin del condicional del paso 2}

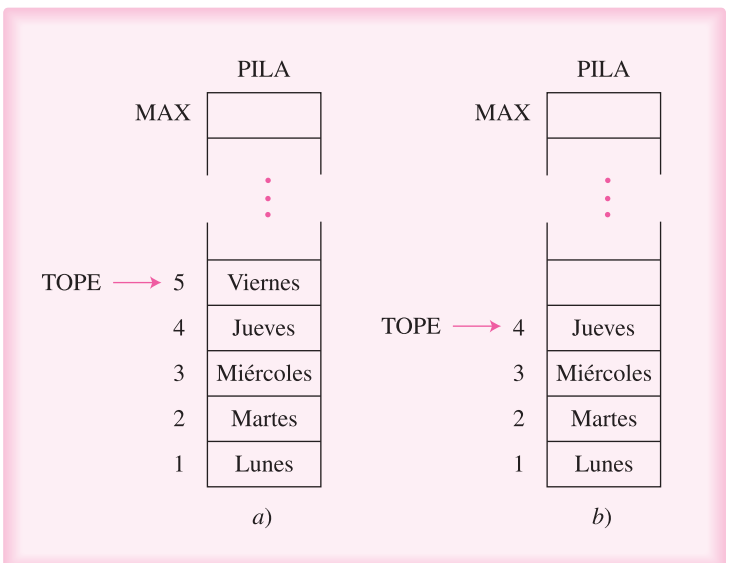
A continuación se presenta un ejemplo para ilustrar el funcionamiento de las operaciones de inserción y eliminación en pilas.

Ejemplo 3.1

Si se insertaran los elementos *lunes*, *martes*, *miércoles*, *jueves* y *viernes* en PILA, la estructura quedaría tal y como se muestra en la figura 3.6*a*. Ahora bien, si se eliminara el elemento *viernes*, el TOPE apuntaría ahora a jueves (fig. 3.6*b*).

Si en algún momento se quisiera eliminar al elemento *martes*, esto no sería posible ya que sólo se puede tener acceso al elemento que se encuentra en la cima de la pila.

FIGURA 3.6
Inserción y eliminación en pilas.

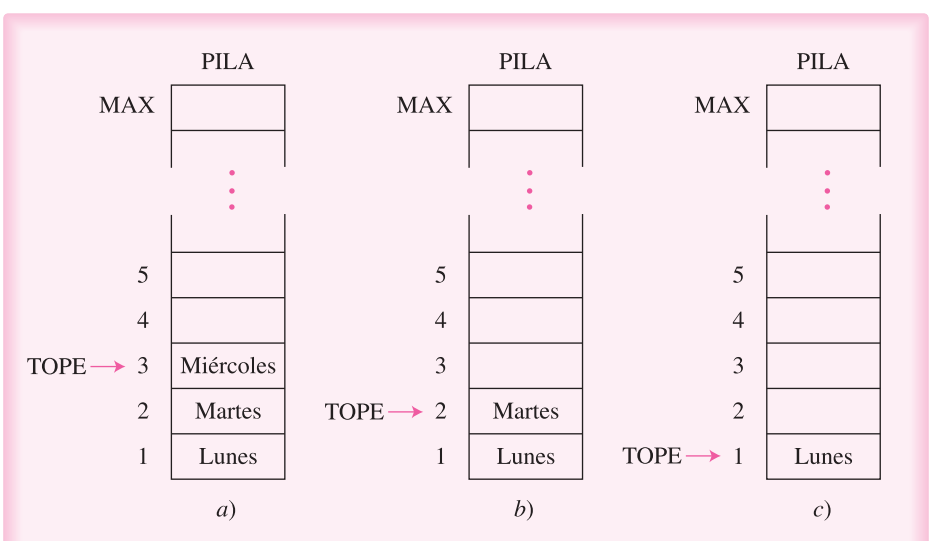


Una forma de resolver este problema es eliminar primeramente los elementos *jueves* y *miércoles*, de esta manera *martes* quedaría ubicado en la cima de PILA y ahora sería posible extraerlo (figuras 3.7*a*, 3.7*b* y 3.7*c*).

3.2.3 Aplicaciones de pilas

Las pilas son una estructura de datos muy usada en la solución de diversos tipos de problemas, en el área de la computación. Ahora se analizarán algunos de los casos más representativos de aplicación de las mismas:

FIGURA 3.7
Inserción y eliminación, *a*) Luego de sacar jueves. *b*) Luego de sacar miércoles. *c*) Luego de sacar martes.



- Llamadas a subprogramas
- Recursividad
- Tratamiento de expresiones aritméticas
- Ordenación

Llamadas a subprogramas

Cuando se tiene un programa que llama a un subprograma, también conocido como módulo o función, internamente se usan pilas para guardar el estado de las variables del programa, así como las instrucciones pendientes de ejecución en el momento que se hace la llamada. Cuando termina la ejecución del subprograma, los valores almacenados en la pila se recuperan para continuar con la ejecución del programa en el punto en el cual fue interrumpido. Además de las variables se recupera la dirección del programa en la que se hizo la llamada, porque a esa posición se regresa el control del proceso.

Supongamos, por ejemplo, que se tiene un programa principal (*PP*) que llama a los subprogramas UNO y DOS. A su vez, el subprograma DOS llama al TRES. Cada vez que la ejecución de uno de los subprogramas concluye, se regresa el control al nivel inmediato superior (fig. 3.8).

Cuando el programa *PP* llama a UNO, se guarda en una pila la posición en la que se hizo la llamada (fig. 3.9*a*). Al terminar UNO, el control se regresa a *PP* recuperando previamente la dirección de la pila (fig. 3.9*b*). Al llamar a DOS, nuevamente se guarda la dirección de *PP* en la pila (fig. 3.9*c*). Cuando DOS llama a TRES, se pone en la pila la dirección de DOS (fig. 3.9*d*). Después de procesar TRES, se recupera la posición de DOS para continuar con su ejecución (fig. 3.9*e*). Al terminar DOS se regresa el control a *PP*, obteniendo previamente la dirección guardada en la pila (fig. 3.9*f*).

Finalmente podemos concluir que las pilas son necesarias en este tipo de aplicaciones por lo siguiente:

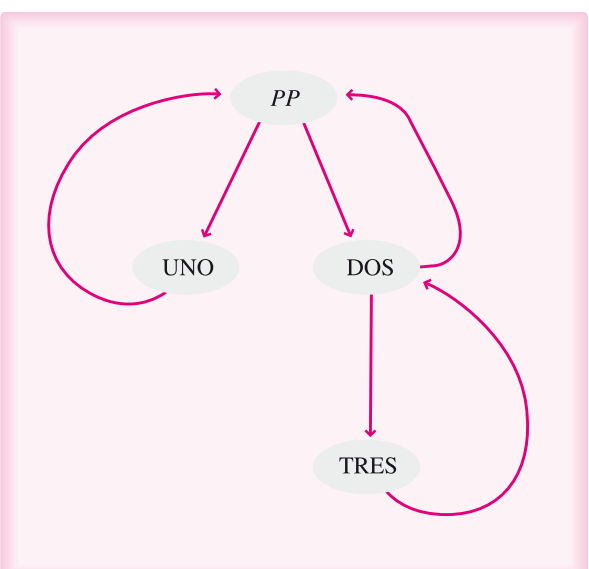


FIGURA 3.8
Llamada a subprogramas.

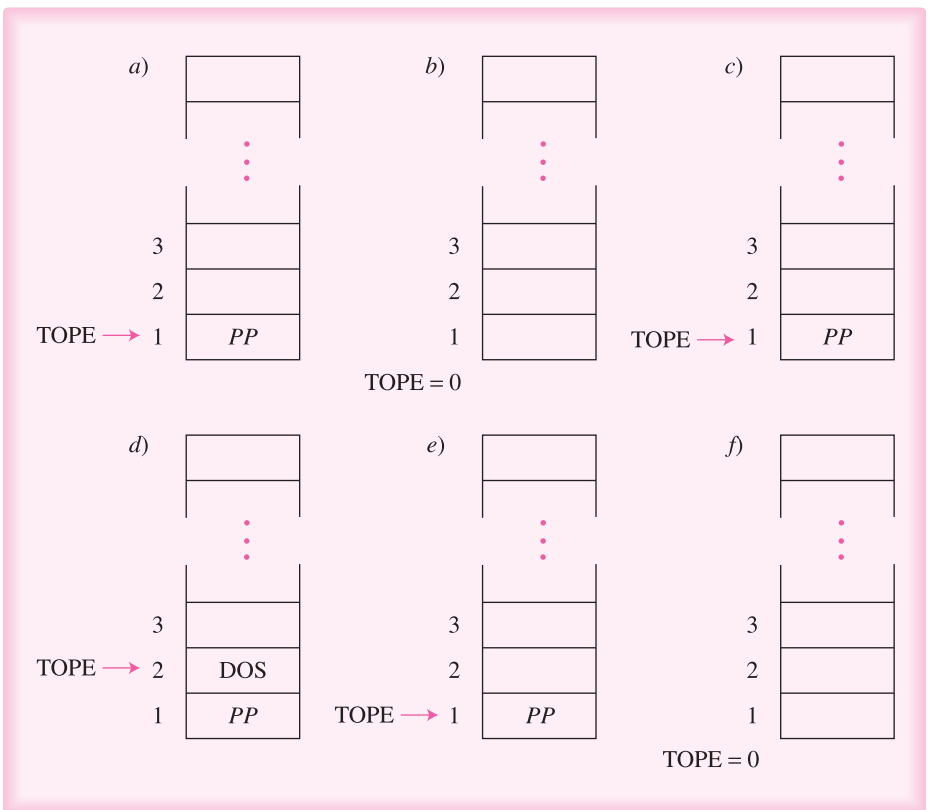


FIGURA 3.9
Aplicación de pilas: llamadas a subprogramas.

- Permiten guardar la dirección del programa, o subprograma, desde donde se hizo la llamada a otros subprogramas, para regresar posteriormente y seguir ejecutándolo a partir de la instrucción inmediata a la llamada.
- Permiten guardar el estado de las variables en el momento en que se hace la llamada, para seguir ocupándolas al regresar del subprograma.

Recursividad

El capítulo 4 está dedicado al estudio de la recursividad. Se dejará para entonces la aplicación de pilas en procesos recursivos.

Tratamiento de expresiones aritméticas

Un problema interesante en computación consiste en convertir expresiones en notación infija a su equivalente en notación prefija o posfija o prefija. Se presenta primero una breve introducción a estos conceptos.

- Dada la expresión ***A + B*** se dice que ésta se encuentra en notación **infija**, porque el operador (+) se encuentra **entre** los operandos (*A* y *B*).

- Dada la expresión *AB*+ se dice que ésta se encuentra en notación **postfija**, porque el operador (+) se encuentra **después** de los operandos (*A* y *B*).
- Dada la expresión +*AB* se dice que ésta se encuentra en notación **prefija**, porque el operador (+) está **precediendo** a los operandos (*A* y *B*).

La ventaja de usar expresiones en notación postfija o prefija radica en que no son necesarios los paréntesis para indicar orden de operación, ya que éste queda establecida por la ubicación de los operadores con respecto a los operandos.

Para convertir una expresión dada en notación infija a una en notación postfija o prefija se establecen primero ciertas condiciones:

- Solamente se manejarán los siguientes operadores —se presentan de mayor a menor según sea su prioridad de ejecución—:

Operador	Nombre de la operación
^	Potencia
* /	Multiplicación y división
+ –	Suma y resta

- Los operadores de más alta prioridad se ejecutan primero.
- Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procesarán de izquierda a derecha.
- Las subexpresiones que se encuentran entre paréntesis tendrán más prioridad que cualquier operador.

Se presentan a continuación, paso a paso, algunos ejemplos de conversión de expresiones infijas a notación posfija.

Ejemplo 3.2

En este ejemplo se exponen dos casos de traducción de notación infija a posfija. El primero de ellos es una expresión simple, mientras que el segundo presenta mayor grado de complejidad. En la tabla 3.1 se muestran los pasos necesarios para lograr la traducción de la primera expresión, y en la tabla 3.2 los correspondientes a la segunda expresión.

a) Expresión infija: $X + Z * W$

Expresión posfija: *XZW**+

El primer operador que se procesa durante la traducción de la expresión es la multiplicación, paso 1, debido a que es el de más alta prioridad. Se coloca el operador de

TABLA 3.1
Traducción de infija a posfija

Paso	Expresión
0	$X + Z * W$
1	$X + ZW*$,
2	$XZW* +$

tal manera que los operandos afectados por él lo precedan. Para el operador de suma se sigue el mismo criterio, los dos operandos lo preceden. En este caso el primer operando es *X* y el segundo es *ZW**.

b) Expresión infija: $(X + Z)^* \; W / T \; \wedge \; Y - V$

Expresión postfija: *XZ+W*TY^/V–*

TABLA 3.2
Traducción de expresión infija a posfija.

Paso	Expresión
0	$(X + Z)^* \; W/T \; \wedge \; Y - V$
1	$XZ + {}^*W/T \; \wedge \; Y - V$
2	$XZ + {}^*W/TY \; \wedge \; - V$
3	$XZ + W^*/TY \; \wedge \; - V$
4	$XZ + W \; ^* \; TY \; \wedge \; / \; - V$
5	$XZ + W \; ^* \; TY \; \wedge \; / \; V -$

En el paso 1 se convierte la subexpresión que se encuentra entre paréntesis por ser la de más alta prioridad. Luego se sigue con el operador de potencia, paso 2, y así con los demás, según su jerarquía. Como consecuencia de que la multiplicación y la división tienen igual prioridad, se procesa primero la multiplicación por encontrarse más a la izquierda en la expresión, paso 3. El operador de la resta es el último que se mueve, paso 5. A continuación se presenta el algoritmo que traduce una expresión infija a otra posfija.

Algoritmo 3.5 Conv_postfija

Conv_postfija (EI, EPOS)

{Este algoritmo traduce una expresión infija —*EI*— a postfija —EPOS—, haciendo uso de una pila —PILA—. MAX es el número máximo de elementos que puede almacenar la pila}

1. Hacer TOPE ← 0
2. Mientras (*EI* sea diferente de la cadena vacía) Repetir
Tomar el símbolo más a la izquierda de *EI*. Recortar luego la expresión
2.1 *Si* (el símbolo es paréntesis izquierdo)
entonces {Poner símbolo en PILA. Se asume que hay espacio en PILA}
Llamar a Pone con PILA, TOPE, MAX y símbolo
si no
2.1.1 *Si* (el símbolo es paréntesis derecho)
entonces
2.1.1.1 Mientras (PILA[TOPE] ≠ paréntesis izquierdo) Repetir
Llamar a Quita con PILA, TOPE y DATO
Hacer EPOS ← EPOS + DATO

```
        2.1.1.2 {Fin del ciclo del paso 2.1.1.1}
                Llamar a Quita con PILA, TOPE y DATO
                {Se quita el paréntesis izquierdo de PILA y no se agrega
                 a EPOS}
                    si no
        2.1.1.3 Si (el símbolo es un operando)
                        entonces
                            Agregar símbolo a EPOS
                        si no {Es un operador}
                        Llamar Pila_vacía con PILA, TOPE y BAND
      2.1.1.3A  Mientras (BAND = FALSO) y (la prioridad del
                            operador sea menor o  igual que la prioridad
                            del operador que está en la cima de PILA)
                            Repetir
                            Llamar a Quita con PILA, TOPE y DATO
                            Hacer EPOS ← EPOS + DATO
                            Llamar a Pila_vacía con PILA, TOPE y BAND
      2.1.1.3B  {Fin del ciclo del paso 2.1.1.3A}
                Llamar a Pone con PILA, TOPE, MAX y símbolo
        2.1.1.4 {Fin del condicional del paso 2.1.1.3}
    2.1.2 {Fin del condicional del paso 2.1.1}
  2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Llamar a Pila_vacía con PILA, TOPE y BAND
5. Mientras (BAND = FALSO) Repetir
     Llamar a Quita con PILA, TOPE y DATO
     Hacer EPOS ← EPOS + DATO
     Llamar a Pila_vacía con PILA, TOPE y BAND
6. {Fin del ciclo del paso 5}
7. Escribir EPOS
```

Cabe señalar que para este algoritmo se maneja la escala de prioridades presentada al inicio de esta sección.

Ejemplo 3.3

En este ejemplo se retoman los casos del ejemplo 3.2 para ilustrar el funcionamiento del algoritmo Conv_posfija.

a) Expresión infija: $X + Z * W$

Expresión posfija: $XZW*+$

En la tabla 3.3 se presentan los pasos necesarios para lograr la traducción deseada siguiendo el algoritmo 3.5.

En los pasos 1, 3 y 5 el símbolo analizado —un operando— se agrega directamente a EPOS. Al analizar el operador +, paso 2, se verifica si en PILA hay operadores con mayor o igual prioridad. En este caso, PILA está vacía; por tanto, se pone el símbolo en el tope de ella. Con el operador *, paso 4, sucede algo similar. En PILA no existen

TABLA 3.3
Traducción de expresión infija a posfija

Paso	*EI*	Símbolo analizado	Pila	EPOS
0	$X + Z * W$			
1	$+ Z * W$	X		X
2	$Z * W$	+	+	X
3	$* W$	Z	+	XZ
4	W	*	+ *	XZ
5		W	+ *	XZW
6			+	$XZW *$
7				$XZW * +$

operadores de mayor o igual prioridad —la suma tiene menor prioridad que la multiplicación—, por lo que se agrega el operador * a PILA. En los dos últimos pasos, 6 y 7, se extraen de PILA sus elementos, agregándolos a EPOS.

b) Expresión infija: $(X + Z)* W / T \wedge Y - V$

Expresión postfija: $XZ+W*TY\wedge/V-$

En la tabla 3.4 se presentan los pasos necesarios para lograr la traducción deseada, siguiendo el algoritmo 3.5.

Los pasos que se consideran más relevantes son: en el paso 5, al analizar el paréntesis derecho se extraen repetidamente todos los elementos de PILA (en este caso sólo el operador +), agregándolos a EPOS hasta encontrar un paréntesis izquierdo. El paréntesis izquierdo se quita de PILA pero no se incluye en EPOS —recuerde que las expresiones en notación polaca no necesitan de paréntesis para indicar prioridades—. Cuando se trata el operador de división, paso 8, se quita de PILA el operador * y se agrega a EPOS, ya que la multiplicación tiene igual prioridad que la división. Al analizar el operador de resta, paso 12, se extraen de PILA y se incorporan a EPOS todos los operadores de mayor o igual prioridad, en este caso son todos los que están en ella —la potencia y la división—, agregando finalmente el símbolo en PILA. Luego de agregar a EPOS el último operando, y habiendo revisado toda la expresión inicial, se vacía PILA y se incorporan los operadores (en este caso el operador –) a la expresión postfija.

A continuación se presenta el algoritmo para convertir expresiones infijas a expresiones escritas en notación prefija.

Ejemplo 3.4

En este ejemplo se exponen dos casos de traducción de notación infija a prefija. El primero de ellos es una expresión simple, mientras que el segundo presenta mayor grado de complejidad.

a) Expresión infija: $X + Z * W$

Expresión prefija: $+ X*ZW$

TABLA 3.4
Traducción de expresión infija a postfija

Paso	*EI*	Símbolo analizado	Pila	EPOS
0	$(X + Z)^* W / T \wedge Y - V$			
1	$X + Z)^* W / T \wedge Y - V$	(	(	
2	$+ Z)^* W / T \wedge Y - V$	X	(	X
3	$Z)^* W / T \wedge Y - V$	+	(+	X
4	$)^* W / T \wedge Y - V$	Z	(+	XZ
5	$^* W / T \wedge Y - V$	)	(	$XZ+$
		)		$XZ+$
6	$W / T \wedge Y - V$	*	*	$XZ+$
7	$/ T \wedge Y - V$	W	*	$XZ+W$
8	$T \wedge Y - V$	/	/	$XZ+W*$
		/	/	$XZ+W*$
9	$\wedge Y - V$	T	/	$XZ+W*T$
10	$Y - V$	$\wedge$	/ $\wedge$	$XZ+W*T$
11	$- V$	Y	/ $\wedge$	$XZ+W*TY$
		–	/	$XZ+W*TY\wedge$
12	V	–		$XZ+W*TY\wedge /$
		–	–	$XZ+W*TY\wedge /$
13		V	–	$XZ+W*TY\wedge /V$
14				$XZ+W*TY\wedge /V-$

En la tabla 3.5 se presentan los pasos necesarios para lograr la traducción deseada.

Como en el caso de la notación postfija, ejemplo 3.2, aquí también el operador de multiplicación se procesa primero. De la traducción de la expresión, paso 1, resulta el operador precediendo a los operandos. Lo mismo para el operador de suma, paso 2.

b) Expresión infija: $(X + Z)^* W / T \wedge Y - V$

Expresión prefija: $-/*+XZW\wedge TYV$

En la tabla 3.6 se presentan los pasos necesarios para lograr la traducción deseada.

Lo primero que se procesa en este caso es la subexpresión que se encuentra entre paréntesis, paso 1. El orden en que se procesan los operadores es el mismo que se siguió

TABLA 3.5
Traducción de expresión infija a prefija

Paso	Expresión
0	$X + Z * W$
1	$X + * ZW$
2	$+ X * ZW$

TABLA 3.6
Traducción de expresión infija a prefija

Paso	Expresión
0	$(X + Z) * W / T \wedge Y - V$
1	$+ XZ * W / T \wedge Y - V$
2	$+ XZ * W / \wedge TY - V$
3	$* + XZW / \wedge TY - V$
4	$/ * + XZW \wedge TY - V$
5	$- / * + XZW \wedge TYV$

para la conversión de infija a posfija. Por tanto, sería reiterativo volver a explicar paso a paso el contenido de la tabla 3.6. Sin embargo, es de destacar la posición que ocupan los operadores con respecto a los operandos: los primeros preceden a los segundos.

A continuación se incluye el algoritmo de conversión de notación infija a prefija. Este algoritmo se diferencia del anterior básicamente en el hecho de que los elementos de la expresión en notación infija se recorrerán de derecha a izquierda.

Algoritmo 3.6 Conv_prefija

Conv_prefija (EI, EPRE)

{Este algoritmo traduce una expresión en notación infija, EI a prefija, EPRE, haciendo uso de una pila —PILA—}

{TOPE es una variable de tipo entero y MAX representa el máximo número de elementos que puede almacenar la pila}

1. Hacer TOPE ← 0
2. Mientras (*EI* sea diferente de la cadena vacía) *Repetir*
Tomar el símbolo más a la derecha de *EI* recortando luego la expresión
2.1 *Si* (el símbolo es paréntesis derecho)
entonces {Poner símbolo en pila}
Llamar a Pone con PILA, TOPE, MAX y símbolo
si no
2.1.1 Si (símbolo es paréntesis izquierdo)
entonces
2.1.1.1 Mientras (PILA[TOPE] ≠ paréntesis derecho) *Repetir*
Llamar a Quita con PILA, TOPE y DATO
Hacer EPRE ← EPRE + DATO
2.1.1.2 {Fin del ciclo del paso 2.1.1.1}
{Sacamos el paréntesis derecho de PILA y no se agrega a EPRE}
Llamar a Quita con PILA, TOPE y DATO
si no
2.1.1.3 *Si* (símbolo es un operando)
entonces
Agregar símbolo a EPRE

```
                    si no {Es un operador}
                        Llamar a Pila_vacía con PILA, TOPE y BAND
                    2.1.1.3A Mientras (BAND = FALSO) y (la prioridad del operador
                             sea menor que la prioridad del operador que está en la cima de
                             PILA) Repetir
                                 Llamar a Quita con PILA, TOPE y DATO
                                 Hacer EPRE ← EPRE + DATO
                                 Llamar a Pila_vacía con PILA, TOPE y BAND
                    2.1.1.3B {Fin del ciclo del paso 2.1.1.3A}
                             Llamar a Pone con PILA, TOPE, MAX y símbolo
                2.1.1.4 {Fin del condicional del paso 2.1.1.3}
            2.1.2 {Fin del condicional del paso 2.1.1}
        2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
   Llamar a Pila_vacía con PILA, TOPE y BAND
4. Mientras (BAND = FALSO) Repetir
       Llamar a Quita con PILA, TOPE y DATO
       Hacer EPRE ← EPRE + DATO
       Llamar a Pila_vacía con PILA, TOPE y BAND
5. {Fin del ciclo del paso 4}
6. Escribir EPRE en forma invertida
```

Ejemplo 3.5

Se analizan nuevamente los casos del ejemplo 3.4 para ilustrar el funcionamiento del algoritmo 3.6.

a) Expresión infija: $X + Z * W$

Expresión prefija: $+X*ZW$

En la tabla 3.7 se presentan los pasos necesarios para lograr la traducción deseada siguiendo el algoritmo 3.6.

TABLA 3.7
Traducción de expresión infija a prefija

Paso	EI	Símbolo analizado	Pila	EPRE
0	$X + Z * W$			
1	$X + Z *$	W		W
2	$X + Z$	$*$	$*$	W
3	$X +$	Z	$*$	WZ
4	X	$+$		$WZ *$
		$+$	$+$	$WZ *$
5		X	$+$	$WZ * X$
6				$WZ * X +$
7		Invertir EPRE:		$+ X * ZW$

El operador de multiplicación se pone en PILA al ser examinado, paso 2. Al estar vacía PILA no hay otros operadores que se pudieran quitar —según su prioridad— antes de poner el operador *. En cambio, al analizar el operador de suma, paso 4, se compara su prioridad con la del operador del tope de PILA. En este caso, es menor; por tanto, se extrae el elemento del tope de PILA y se agrega a la expresión prefija, poniendo finalmente el operador + en PILA. Cuando la expresión de entrada queda vacía (es decir, que ya se han analizado todos sus símbolos), se extrae repetidamente cada elemento de PILA y se agrega a la expresión prefija hasta que PILA quede vacía.

b) Expresión infija: $(X + Z) * W / T \wedge Y - V$

Expresión prefija: $-/* + XZW\wedge TYV$

En la tabla 3.8 se presentan los pasos necesarios para lograr la traducción deseada siguiendo el algoritmo 3.6.

Como la expresión se recorre de derecha a izquierda, el primer operador que se procesa es la resta, paso 2. Pero éste es el operador de la expresión de más baja priori-

TABLA 3.8
Traducción de expresión infija a prefija

Paso	*EI*	Símbolo analizado	Pila	EPRE
0	$(X + Z) * W / T \wedge Y - V$			
1	$(X + Z) * W / T \wedge Y -$	V		V
2	$(X + Z) * W / T \wedge Y$	–	–	V
3	$(X + Z) * W / T \wedge$	Y	–	VY
4	$(X + Z) * W / T$	^	– ^	VY
5	$(X + Z) * W /$	T	– ^	VYT
6	$(X + Z) * W$	/ /	– – /	$VYT\wedge$ $VYT\wedge$
7	$(X + Z) *$	W	– /	$VYT\wedge W$
8	$(X + Z)$	*	– / *	$VYT\wedge W$
9	$(X + Z$	)	– / *)	$VYT\wedge W$
10	$(X +$	Z	– / *)	$VYT\wedge WZ$
11	$(X$	+	– / *) +	$VYT\wedge WZ$
12	(	X	– / *) +	$VYT\wedge WZX$
13		((	– / *) – / *	$VYT\wedge WZX+$ $VYT\wedge WZX+$
14			– /	$VYT\wedge WZX+*$
15			–	$VYT\wedge WZX+*/$
16				$VYT\wedge WZX+*/-$
17	Invertir EPRE			$-/*+XZW\wedge TYV$

dad; por tanto, permanecerá en PILA hasta el final del proceso de conversión, paso 15. Cuando se encuentra un paréntesis derecho, paso 9, se agrega directamente a PILA, mientras que cuando el símbolo analizado es un paréntesis izquierdo, paso 13, se extrae repetidamente cada elemento de PILA agregándolo a EPRE, hasta que se encuentra un paréntesis derecho. Éste se retira de PILA y no se agrega a EPRE. Cuando ya se analizaron todos los símbolos de la expresión se procede a quitar de PILA sus elementos, añadiéndolos a EPRE. Finalmente se invierte EPRE para obtener la expresión en notación prefija, paso 17. Para evitar el último paso del algoritmo, invertir la expresión, se podrían ir concatenando los símbolos en EPRE en orden inverso.

Ordenación

Otra aplicación de las pilas se puede ver en el método de ordenación rápida. Como el tema de ordenación es ampliamente tratado en el capítulo 8, se sugiere remitirse a él.

3.2.4 La clase *Pila*

La **clase *Pila*** tiene atributos y métodos. Los atributos son la colección de elementos y el TOPE. Los métodos, por otra parte, son todas aquellas operaciones analizadas en la sección anterior —Pila_vacía, Pila_llena, Pone y Quita—. En la figura 3.10 se puede observar gráficamente la clase Pila.

Se tiene acceso a los miembros de un objeto de la clase *Pila* por medio de la notación de puntos. Al asumir que la variable PIOBJ representa un objeto de la clase Pila, previamente creado, se puede hacer:

PIOBJ.Pila_llena para invocar el método que determina si la pila está llena o no. En este método no hay argumentos, ya que todos los valores requeridos son miembros de la clase.

PIOBJ.Pone(argumento) para insertar un nuevo elemento en la pila. En este método sólo hay un argumento que indica el valor a guardar en la pila; los demás valores requeridos son miembros de la clase.

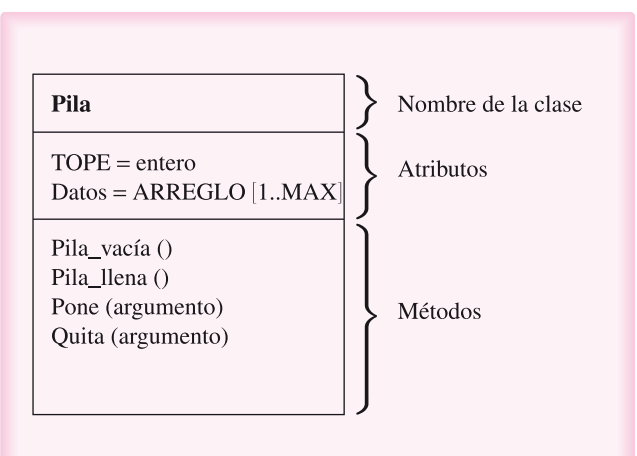


FIGURA 3.10
Clase *Pila*.

3.3 COLAS

Una **cola** constituye una estructura lineal de datos en la que los nuevos elementos se introducen por un extremo y los ya existentes se eliminan por el otro. Es importante señalar que los componentes de la cola se eliminan en el mismo orden en el cual se insertaron. Es decir, el primer elemento que se introduce en la estructura será el que se eliminará en primer orden. Debido a esta característica, las colas también reciben el nombre de estructuras **FIFO** (*First-In, First-Out*: el primero en entrar es el primero en salir).

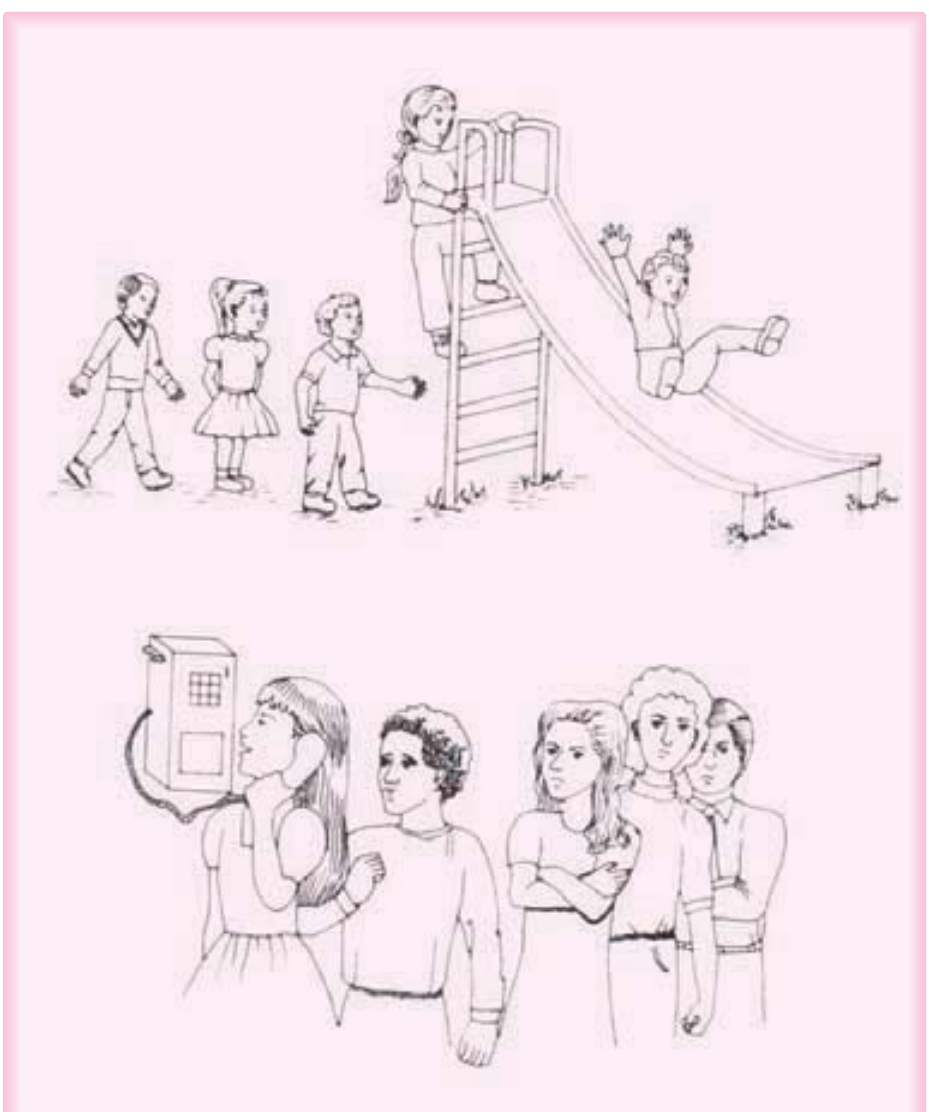


FIGURA 3.11
Ejemplos prácticos de colas.

Existen numerosos casos de la vida real en los cuales se usa este concepto. Por ejemplo, la cola de los bancos en las que los clientes esperan para ser atendidos —la primera persona de la cola será la primera en recibir el servicio—, la cola de los niños que esperan a veces pacientemente para subir a un juego mecánico, las colas de los vehículos esperando la luz verde del semáforo, las colas para entrar a un cine, teatro o estadio de fútbol, etcétera.

3.3.1 Representación de colas

Las colas, al igual que las pilas, no existen como estructuras de datos estándar en los lenguajes de programación. Este tipo de estructura de datos se puede representar mediante el uso de:

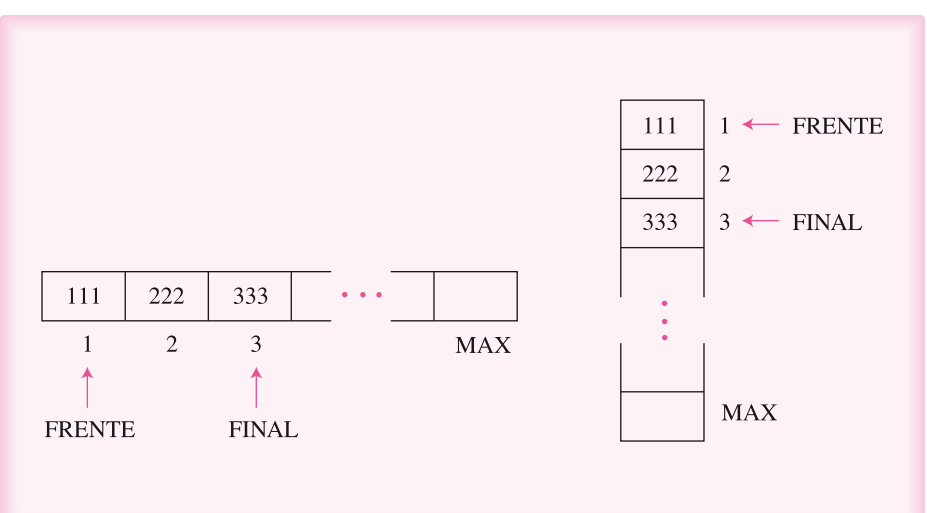
- Arreglos
- Listas

Al igual que en el caso de las pilas, en este libro se utilizarán arreglos para mostrar su funcionamiento. Sin embargo, la implementación mediante listas es incluso más sencilla. El lector puede implementar los algoritmos necesarios para colas, después de estudiar el capítulo que se dedica a la estructura lineal de datos.

Cuando se implementan con arreglos unidimensionales, es importante definir un tamaño máximo para la cola y dos variables auxiliares. Una de ellas para que almacene la posición del primer elemento de la cola —FRENTE— y otra para que guarde la posición del último elemento de la cola —FINAL—. En la figura 3.12 se muestra la representación de una cola en la cual se han insertado tres elementos: 111, 222 y 333, en ese orden.

El elemento 111 está en el FRENTE ya que fue el primero que se insertó; mientras que el elemento 333, que fue el último en entrar, está en el FINAL de la cola.

FIGURA 3.12
Representación de colas.



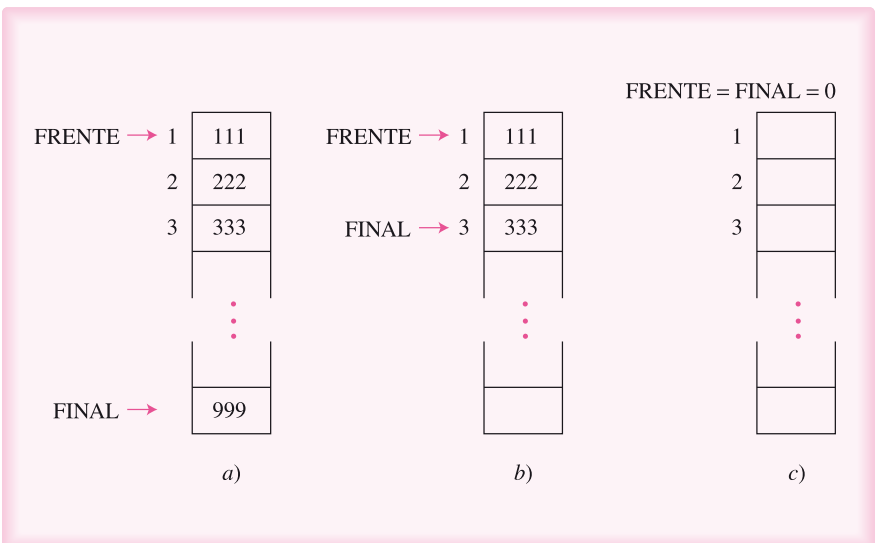


FIGURA 3.13
Representación de colas, *a*) Cola llena. *b*) Cola con algunos elementos. *c*) Cola vacía.

En la figura 3.13, por otra parte, se presentan ejemplos de: *a)* cola llena, *b)* cola con algunos elementos y *c)* cola vacía.

3.3.2 Operaciones con colas

La definición de la estructura de datos tipo cola queda completa al incluir las operaciones que se pueden realizar en ella. Las operaciones básicas que pueden efectuarse son:

- Insertar un elemento en la cola
- Eliminar un elemento de la cola

Las inserciones se llevarán a cabo por el FINAL de la cola, mientras que las eliminaciones se harán por el FRENTE —recuerde que el primero en entrar es el primero en salir—.

Considerando que una cola puede almacenar un máximo número de elementos y que además FRENTE indica la posición del primer elemento y FINAL la posición del último, se presentan a continuación los algoritmos correspondientes a las operaciones mencionadas.

Algoritmo 3.7 Inserta_cola

Inserta_cola (COLA, MAX, FRENTE, FINAL, DATO)

{Este algoritmo inserta el elemento DATO al final de una estructura tipo cola. FRENTE y FINAL son los punteros que indican, respectivamente, el inicio y fin de COLA. La primera vez FRENTE y FINAL tienen el valor 0, ya que la cola está vacía. MAX es el máximo número de elementos que puede almacenar la cola}

```
1. Si (FINAL < MAX) {Verifica que hay espacio libre}
       entonces
              Hacer FINAL ← FINAL + 1 {Actualiza FINAL} y  COLA[FINAL] ← DATO
   1.1 Si (FINAL = 1) entonces {Se insertó el primer elemento de COLA}
              Hacer FRENTE ← 1
   1.2 {Fin del condicional del paso 1.1}
   si no
         Escribir "Desbordamiento – Cola llena"
2. {Fin del condicional del paso 1}
```

Algoritmo 3.8 Elimina_cola

Elimina_cola (COLA, FRENTE, FINAL, DATO)

```
{Este algoritmo elimina el primer elemento de una estructura tipo cola y lo almacena en DATO.
FRENTE y FINAL son los punteros que indican, respectivamente, el inicio y fin de la cola}

1. Si (FRENTE ≠ 0) {Verifica que la cola no esté vacía}
       entonces
            Hacer DATO ← COLA [FRENTE]
   1.1 Si (FRENTE = FINAL) {Si hay un solo elemento}
            entonces
               Hacer FRENTE ← 0 y FINAL ← 0 {Indica COLA vacía}
            si no
               Hacer FRENTE ← FRENTE + 1
   1.2 {Fin del condicional del paso 1.1}
            si no
         Escribir "Subdesbordamiento – Cola vacía"
2. {Fin del condicional del paso 1}
```

Es posible definir algoritmos auxiliares para determinar si una cola está llena o vacía. A partir de estos algoritmos se podrían reescribir los algoritmos 3.7 y 3.8.

A continuación se presentan los algoritmos que permiten verificar el estado de una cola, quedando como tarea sugerida la reescritura de los dos algoritmos anteriores.

Algoritmo 3.9 Cola_vacía

Cola_vacía (COLA, FRENTE, BAND)

```
{Este algoritmo determina si una estructura de datos tipo cola está vacía, asignando a BAND
el valor de verdad correspondiente}

1. Si (FRENTE = 0)
```

```
        entonces
            Hacer BAND ← VERDADERO
        si no
            Hacer BAND ← FALSO
2.  {Fin del condicional del paso 1}
```

Algoritmo 3.10 Cola_llena

Cola_llena (COLA, FINAL, MAX, BAND)

{Este algoritmo determina si una estructura de datos tipo cola está llena, asignando a BAND el valor de verdad correspondiente. MAX es el número máximo de elementos que puede almacenar COLA}

```
1.  Si (FINAL = MAX)
        entonces
            Hacer BAND ← VERDADERO
        si no
            Hacer BAND ← FALSO
2.  {Fin del condicional del paso 1}
```

Aquí se incluye un ejemplo para ilustrar el funcionamiento de las operaciones de inserción y eliminación en colas.

Ejemplo 3.6

Retome el ejemplo 3.1 de la sección 3.1.2. Se insertan en COLA los elementos: *lunes*, *martes*, *miércoles*, *jueves* y *viernes* —en ese orden—, de modo que la estructura queda como se muestra en la figura 3.14. Para este ejemplo MAX = 7.

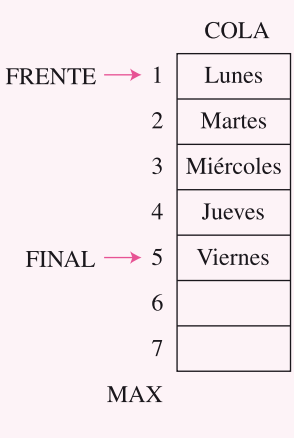
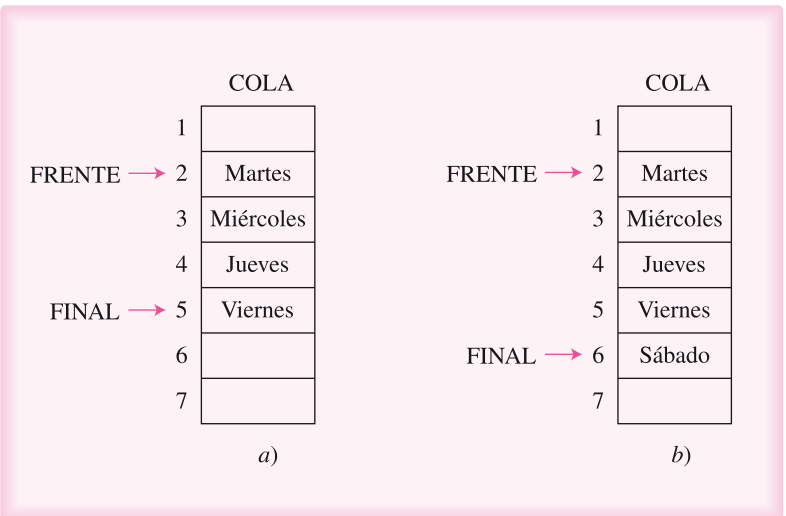


FIGURA 3.14
Inserción y eliminación en colas.

FIGURA 3.15
Inserción y eliminación en colas. *a*) Luego de eliminar lunes. *b*) Luego de insertar sábado.

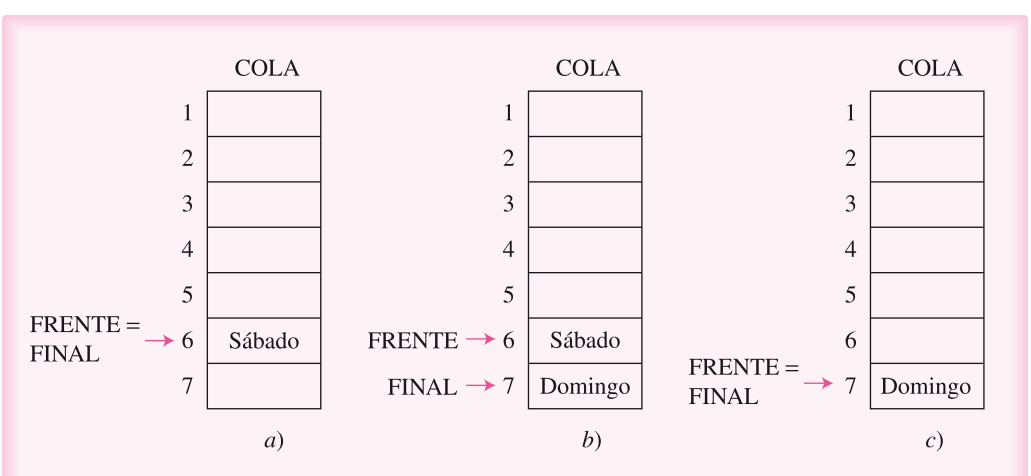


El elemento *lunes* es el primero que se puede eliminar por ser el primero que se insertó en la cola. Luego de la eliminación, FRENTE guarda la posición del siguiente elemento (fig. 3.15*a*). Si ahora se insertara *sábado*, éste ocuparía ahora la posición siguiente al elemento *viernes* (fig. 3.15*b*).

Analice lo que ocurre en la cola, si se llevan a cabo las siguientes operaciones:

Se eliminan *martes*, *miércoles*, *jueves* y *viernes* (fig. 3.16*a*).
Se inserta *domingo* (fig. 3.16*b*).
Se elimina *sábado* (fig. 3.16*c*).

FIGURA 3.16
Inserción y eliminación en colas. *a*) Luego de eliminar martes, miércoles, jueves y viernes. *b*) Luego de insertar domingo. *c*) Luego de eliminar sábado.



Después de insertar al elemento *domingo*, ya no se pueden insertar nuevos elementos a la cola porque FINAL es igual que MAX (FINAL = MAX = 7). Sin embargo, como lo refleja la figura 3.16*c*, existe espacio disponible desperdiciado.

Observe que luego de insertar *domingo* se llegó a una situación conflictiva porque a pesar de que hay espacio disponible, no se pueden insertar otros elementos. Este inconveniente se puede superar perfectamente si manejamos las colas en forma circular.

3.3.3 Colas circulares

Una **cola circular** constituye una estructura de datos lineal en la cual el siguiente elemento del último en realidad es el primero. De esta forma se utiliza de manera más eficiente la memoria de la computadora. En la figura 3.17 se muestra la representación gráfica de una cola circular.

En la figura 3.17 se ilustra cómo se actualizan los punteros FRENTE y FINAL en una cola circular, a medida que se insertan o eliminan elementos. En la figura 3.17*a* la cola tiene algunos elementos (FRENTE = 2 y FINAL = 8). En la figura 3.17*b* se han eli-

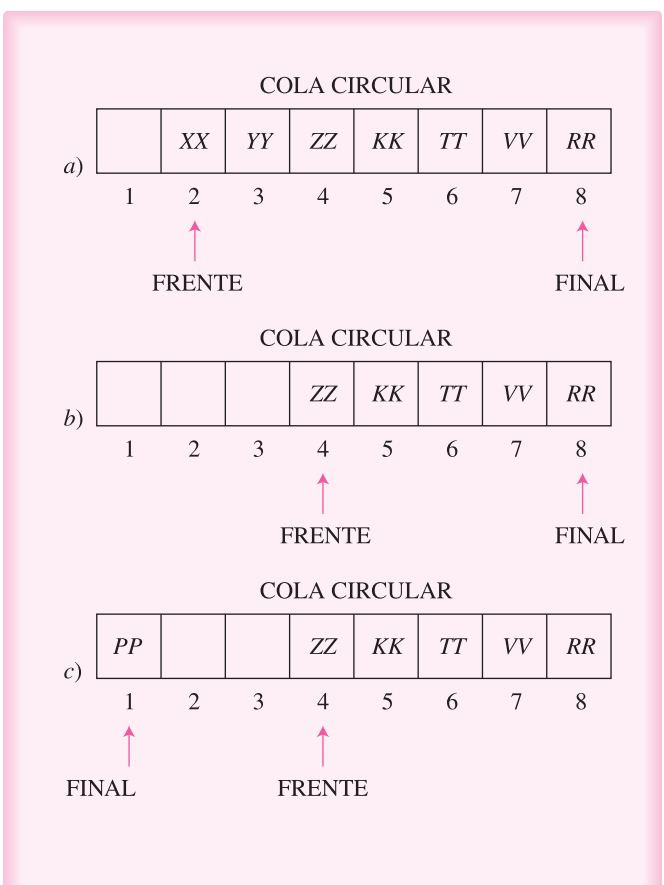


FIGURA 3.17
Representación de colas circulares. *a*) Frente < Final. *b*) Frente < Final. *c*) Frente > Final.

minado de la cola dos elementos —primero se quitó *XX* y luego *YY*—, quedando FRENTE = 4. Por último, en la figura 3.17*c* se ha insertado un nuevo elemento —*PP*— en la cola. Como FINAL = MAX se llevó el apuntador a la primera posición que estaba vacía (FINAL = 1). De esta manera se logra mejor aprovechamiento del espacio de memoria, ya que al eliminar un elemento la casilla correspondiente de la cola queda disponible para futuras inserciones.

A continuación se presentan los algoritmos de inserción y eliminación en colas circulares.

Algoritmo 3.11 Inserta_circular

Inserta_circular (COLACIR, MAX, FRENTE, FINAL, DATO)

{Este algoritmo inserta el elemento DATO al final de una estructura tipo cola circular —COLACIR—. FRENTE y FINAL son los punteros que indican, respectivamente, el inicio y el fin de la cola circular. MAX es el número máximo de elementos que puede almacenar COLACIR}

```
1. Si ((FINAL = MAX) y (FRENTE = 1)) o ((FINAL +1) = FRENTE)
        entonces
            Escribir "Desbordamiento – Cola llena"
        si no
   1.1  Si (FINAL = MAX)
            entonces
                Hacer FINAL ← 1
            si no
                Hacer FINAL ← FINAL + 1
   1.2  {Fin del condicional del paso 1.1}
        Hacer COLACIR[FINAL] ← DATO
   1.3  Si (FRENTE = 0) entonces
            Hacer FRENTE ← 1
   1.4  {Fin del condicional del paso 1.3}
2. {Fin del condicional del paso 1}
```

Algoritmo 3.12 Elimina_circular

Elimina_circular (COLACIR, MAX, FRENTE, FINAL, DATO)

{Este algoritmo elimina el primer elemento de una estructura tipo cola circular —COLACIR— y lo almacena en DATO. FRENTE y FINAL son los punteros que indican, respectivamente, el inicio y fin de la estructura. MAX es el tamaño de COLACIR}

```
1. Si (FRENTE = 0) {Verifica si la cola está vacía}
        entonces
            Escribir "Subdesbordamiento – Cola vacía"
```

```
        si no
        Hacer DATO ← COLACIR[FRENTE]
   1.1  Si FRENTE = FINAL {Si hay sólo un elemento}
              entonces
                   Hacer FRENTE ← 0 y FINAL ← 0
              si no
         1.1.1  Si (FRENTE = MAX)
                        entonces
                             Hacer FRENTE ← 1
                        si no
                             Hacer FRENTE ← FRENTE + 1
         1.1.2  {Fin del condicional del paso 1.1.1}
   1.2   {Fin del condicional del paso 1.1}
2.      {Fin del condicional del paso 1}
```

A continuación se presenta un ejemplo para ilustrar el funcionamiento de las operaciones de inserción y eliminación en colas circulares.

Ejemplo 3.7

En la figura 3.18*a* se presenta una estructura tipo cola circular de máximo 8 elementos (MAX = 8), en la cual ya se han almacenado algunos valores. En la figura 3.18*b* se muestra el estado de la cola luego de insertar el elemento *NN*.

Si se quisiera insertar otro elemento se presentaría un error de desbordamiento, ya que FINAL + 1 = FRENTE.

A continuación se eliminan los valores *XX*, *YY*, *ZZ*, *KK*, *TT* y *VV* en ese orden. La cola queda como se muestra en la figura 3.19.

FIGURA 3.18

Inserción y eliminación en colas circulares. *a*) Estado inicial de la cola circular. *b*) Luego de insertar *NN*.

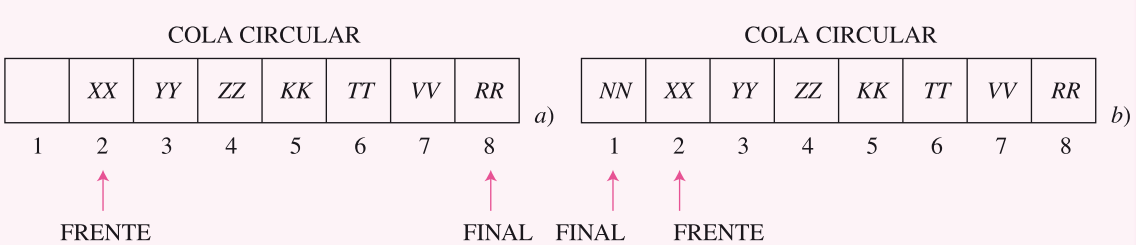
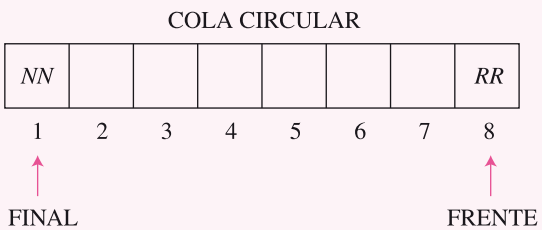
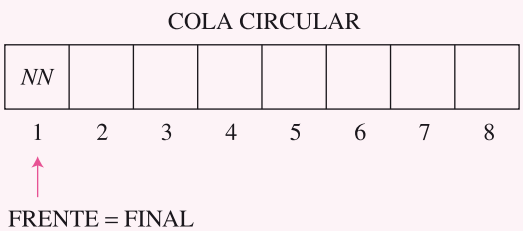


FIGURA 3.19
Inserción y eliminación en colas circulares.



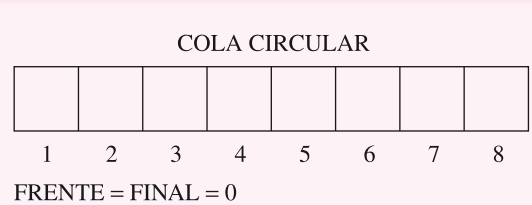
Ahora se elimina el siguiente elemento *RR*. Al ser FRENTE = MAX, se le da el valor de 1, figura 3.20.

FIGURA 3.20
Inserción y eliminación en colas circulares.



Al eliminar *NN*, como FRENTE = FINAL, es decir, sólo quedaba un elemento en la cola, actualizamos los dos punteros en cero. La cola queda vacía, figura 3.21.

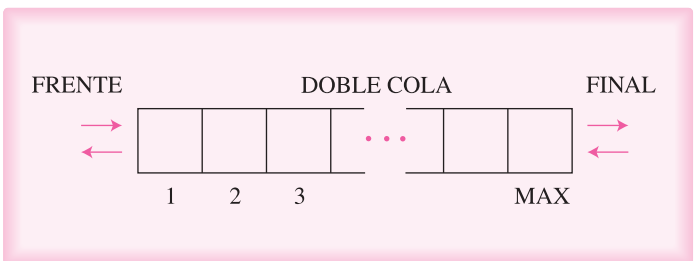
FIGURA 3.21
Inserción y eliminación en colas circulares.



3.3.4 Doble cola

Una **doble cola** o **bicola** es una generalización de una estructura de datos tipo cola. En una doble cola, los elementos se pueden insertar o eliminar por cualquiera de los dos extremos. Es decir, se pueden insertar y eliminar valores tanto por el FRENTE como por el FINAL de la cola. Una doble cola se representa como se muestra en la figura 3.22. Observe que las dos flechas en cada extremo indican que ambas operaciones son posibles.

FIGURA 3.22
Representación de doble cola.



Existen dos variantes de las dobles colas:

- Doble cola con entrada restringida
- Doble cola con salida restringida

La primera de ellas permite que las eliminaciones se realicen por cualesquiera de los dos extremos, mientras que las inserciones sólo por el FINAL de la cola (fig. 3.23).

La segunda variante permite que las inserciones se realicen por cualquiera de los dos extremos, mientras que las eliminaciones sólo por el FRENTE de la cola (fig. 3.24).

FIGURA 3.23
Doble cola con entrada restringida.

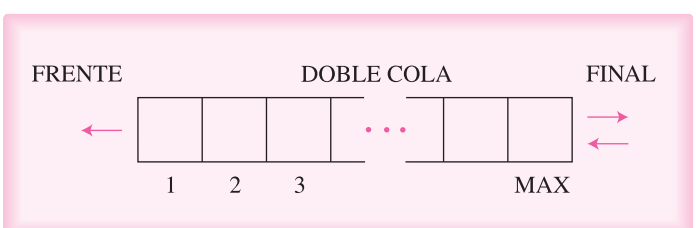
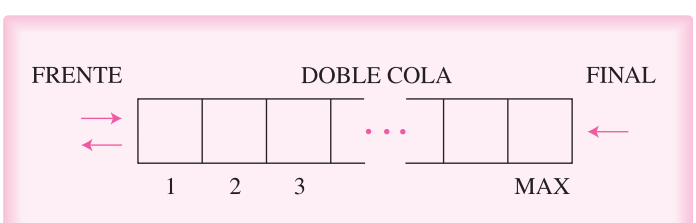


FIGURA 3.24
Doble cola con salida restringida.



3.3.5 Aplicaciones de colas

El concepto de cola está ligado a computación. Una aplicación común de las colas se presenta cuando se envía a imprimir algún documento o programa en las colas de impresión. Cuando hay una sola impresora para atender a varios usuarios, suele suceder que algunos de ellos soliciten los servicios de impresión al mismo tiempo o mientras el dispositivo está ocupado. En estos casos se forma una cola con los trabajos que esperan para ser impresos; éstos se procesarán en el orden en el cual fueron introducidos en la cola.

Otro caso de aplicaciones de colas en computación es el que se presenta en los sistemas de tiempo compartido. Varios usuarios comparten ciertos recursos, como CPU y memoria de la computadora. Los recursos se asignan a los procesos que están en cola de espera, suponiendo que todos tienen una misma prioridad, en el orden en el cual fueron introducidos en la cola.

3.3.6 La clase *Cola*

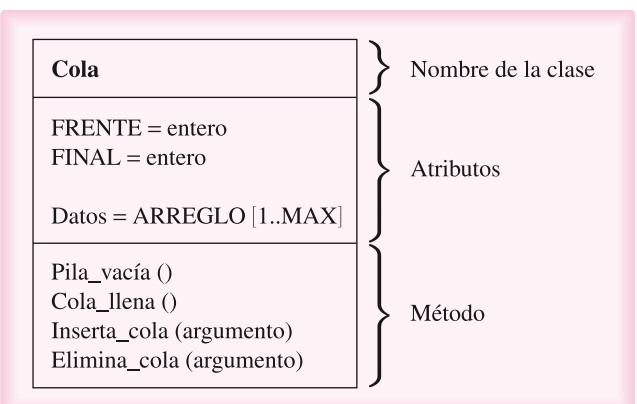
La **clase *Cola*** tiene atributos y métodos, como cualquier clase. Los atributos son la colección de elementos y los punteros FRENTE y FINAL. Los métodos, por otra parte, son todas las operaciones analizadas en las secciones anteriores: Cola_vacía, Cola_llena, Inserta_cola y Elimina_cola.

En la figura 3.25 se puede observar gráficamente a la clase *Cola.* Se tiene acceso a los miembros de un objeto de la clase *Cola* por medio de la notación de puntos. Cuando se asume que la variable COOBJ es un objeto de la clase Cola, previamente creado, se puede hacer:

COOBJ.Cola_llena para invocar el método que determina si la cola está o no llena. En este método no hay argumentos, ya que todos los valores requeridos son miembros de la clase.

COOBJ.Inserta_cola(argumento) para insertar un nuevo elemento en la cola. En este método hay un único argumento, que es el dato a insertar; todos los otros valores requeridos son miembros de la clase.

FIGURA 3.25
Clase *Cola.*



EJERCICIOS

Pilas

1. Traduzca las siguientes expresiones a notación posfija mediante el algoritmo 3.3.

a) $X * (Z + W) / (T - V)$
b) $Z - W * Y + X \wedge K$
c) $X \wedge (Z - T) * W$
d) $(X - Y) * (T - Z)$
e) $Z / (X + Y * T) \wedge W$
f) $W * (Z / (K - T))$

2. Traduzca las siguientes expresiones a notación prefija con el algoritmo 3.4.

a) $(X - T) \wedge Z$
b) $(Z * (K - W) + X) \wedge Y - T$
c) $X + T * W / K$
d) $Z * X - W \wedge K$
e) $(Z - X) \wedge Z + (Z - Y) \wedge K$
f) $(W / (X - Z * T) - Y) \wedge K$

3. Escriba un programa que lea una expresión en notación infija, y la traduzca a notación posfija.

4. Escriba un programa que lea una expresión en notación posfija, y la traduzca a notación prefija.

5. Escriba un programa que evalúe, con la ayuda de una pila, una expresión aritmética dada en notación prefija.

6. Escriba un programa que evalúe, con la ayuda de una pila, una expresión aritmética dada en notación posfija.

7. Escriba un subprograma que inserte un elemento en una pila. Considere todos los casos que se puedan presentar.

8. Escriba un subprograma que elimine un elemento en una pila. Considere todos los casos que se puedan presentar.

9. Dibuje los distintos estados de una estructura tipo pila si se llevan a cabo las siguientes operaciones. Muestre cómo van quedando la pila y el puntero al tope de la misma. Considere que la pila está inicialmente vacía (TOPE = 0) y tiene una capacidad máxima para 8 elementos

a) Insertar (PILA, *X*)
b) Insertar (PILA, *Y*)
c) Eliminar (PILA, *Z*)
d) Eliminar (PILA, *T*)
e) Eliminar (PILA, *U*)
f) Insertar (PILA, *V*)
g) Insertar (PILA, *W*)
h) Eliminar (PILA, *P*)
i) Insertar (PILA, *R*)

- ¿Con cuántos elementos quedó la pila?
- ¿Hubo algún caso de error (desbordamiento o subdesbordamiento)? Si ocurrió algún error, explíquelo.

10. Escriba un subprograma que elimine los elementos repetidos de una pila. Los elementos repetidos ocupan posiciones sucesivas.

11. Escriba un subprograma que invierta los elementos de una pila.

12. Defina la clase *Pila*, usando algún lenguaje orientado a objetos, con base en los algoritmos presentados para programar los métodos para insertar, eliminar y verificar el estado de la pila.

Colas

13. Sea *C* una cola circular de 6 elementos. Inicialmente la cola está vacía (FRENTE = FINAL = 0). Dibuje el estado de *C* luego de realizar cada una de las siguientes operaciones:
a) Insertar los elementos *AA*, *BB* y *CC*
b) Eliminar el elemento *AA*
c) Insertar los elementos *DD*, *EE* y *FF*
d) Insertar el elemento *GG*
e) Insertar el elemento *HH*
f) Eliminar los elementos *BB* y *CC*

- ¿Con cuántos elementos quedó *C*?
- ¿Hubo algún caso de error (desbordamiento o subdesbordamiento)? Si ocurrió algún error, explíquelo.

14. Escriba un subprograma que inserte un elemento en una cola circular. Considere todos los casos que se puedan presentar.

15. Escriba un subprograma que elimine un elemento de una cola circular. Considere todos los casos que se puedan presentar.

16. Utilice una estructura de cola para simular el movimiento de clientes en una cola de espera de un banco —se puede auxiliar con los subprogramas escritos en los ejercicios 14 y 15—.

17. Escriba un subprograma que invierta los elementos de una cola.

18. Defina un algoritmo para insertar un elemento en una doble cola.

19. Defina un algoritmo para eliminar un elemento de una doble cola.

20. Sea *C* una doble cola circular de 6 elementos. Inicialmente la doble cola está vacía (FRENTE = FINAL = 0). Dibuje el estado de la cola después de realizar cada una de las siguientes operaciones.

a) Insertar por el extremo derecho tres elementos: *A*, *B* y *C*.
b) Eliminar por el extremo izquierdo un elemento.
c) Insertar por el extremo izquierdo dos elementos: *D* y *E*.
d) Eliminar por la derecha un elemento.

21. Defina la clase *Cola*, mediante algún lenguaje de programación orientado a objetos, tomando como base para programar los métodos los algoritmos estudiados para insertar, eliminar y verificar el estado de la cola.

22. Se tiene una cola de impresión donde se almacenan las claves de los documentos que se deben imprimir. En la medida en que llega un nuevo trabajo, éste se encola. Cuando la impresora se libera se toma un trabajo de la cola y se imprime. Utilice la clase previamente definida para declarar el objeto ColaImpresión y empléelo en el desarrollo de la aplicación descrita.

Capítulo 4

RECURSIÓN

4.1 INTRODUCCIÓN

La **recursión** o **recursividad** es un concepto amplio, con muchas variantes, y difícil de precisar con pocas palabras. Aparece en numerosas actividades de la vida diaria; por ejemplo, en una fotografía donde se observa otra fotografía. Otro caso ilustrativo es el que se presenta en los programas de televisión, en los cuales un periodista transfiere el control de la noticia a otro periodista que se encuentra en otra ciudad, y éste, a su vez, hace lo mismo con un tercero. Cuando este último termina su participación regresa el control al segundo, y cuando éste también finaliza su intervención regresa el control al primero. En este capítulo nos limitaremos a tratar la recursión como herramienta de programación.

La recursión es un recurso muy poderoso que permite expresar soluciones simples y naturales a ciertos tipos de problemas. Es importante considerar que no todos los problemas son naturalmente recursivos; algunos sí lo son y otros no.

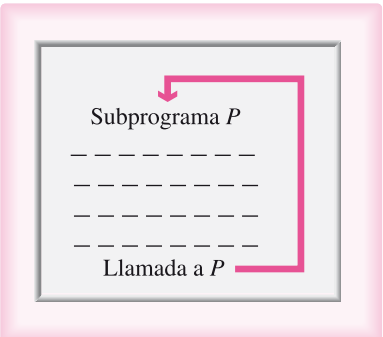
Un objeto recursivo es aquel que aparece en la definición de sí mismo, así como el que se llama a sí mismo. Los árboles, por ejemplo, que se estudiarán en el capítulo 6, representan las estructuras de datos, no lineales y dinámicas, más eficientes que existen actualmente en computación. La característica de los árboles es que son estructuras inherentemente recursivas. Es decir, en cualquier actividad de programación que se realice con árboles se utiliza la recursividad.

La recursión se puede presentar de dos maneras diferentes:

a) **Directa:** el programa o subprograma se llama directamente a sí mismo. Por ejemplo, observe que en la figura 4.1 *P* representa un programa y en alguna parte de él aparece una llamada a sí mismo.
b) **Indirecta:** el subprograma llama a otro subprograma, y éste, en algún momento, llama nuevamente al primero. Por ejemplo, en la figura 4.2 el subprograma *P* llama al subprograma *Q* y éste, a su vez, invoca al primero; es decir, el control regresa a *P*.

En toda definición recursiva de un problema siempre se deben establecer dos pasos diferentes y muy importantes; el **paso básico** y el **paso recursivo**. El primero, uno o varios, dependiendo del problema, se utiliza como condición de parada o fin de la recursividad. A éste llegamos cuando encontramos la solución del problema o cuando decidimos que ya no vamos a seguir, porque no están dadas las condiciones para hacerlo. El

FIGURA 4.1
Recursión directa.



paso segundo, por otra parte, propicia la recursividad. Se pueden presentar uno o varios, nuevamente dependiendo del problema a resolver.

Cuando se analiza la solución recursiva de un problema es importante determinar con precisión cuáles serán los pasos básico y recursivo. En cada vuelta del ciclo es importante que nos acerquemos cada vez más a la solución del problema, o sea, al paso básico. Si esto no ocurre, entonces podemos estar ante un ciclo extraño. Es decir, el problema estaría mal definido y, en tal caso, la máquina se quedaría ejecutando por tiempo indefinido el programa en que estuviera, y sólo terminaría al agotarse la memoria.

A continuación se presentan algunos ejemplos que nos pueden ayudar a comprender más rápidamente el concepto recursión.

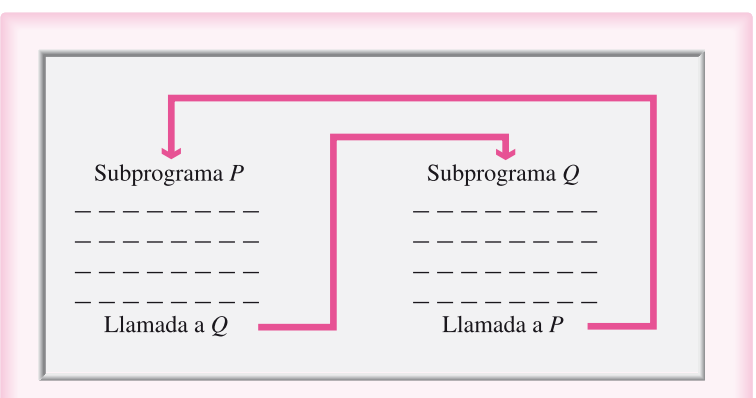
Ejemplo 4.1 Factorial de un número

El factorial de un número entero positivo n se define como el producto de los números comprendidos entre 1 y n. También como n por el factorial de $(n - 1)$, así aparece el concepto de recursión. La expresión $\boldsymbol{n!}$ simboliza el factorial de n. A continuación se ilustra este caso.

Por definición:

$0! = 1$ → **Paso básico**
$1! = 1$ → **Paso básico**

FIGURA 4.2
Recursión indirecta.



$2! = 2 * 1! = 2 * 1$
$3! = 3 * 2! = 3 * 2 * 1$
$4! = 4 * 3! = 4 * 3 * 2 * 1$
.
.
.
$n! = n * (n-1)! = n * (n-1) * \ldots * 4 * 3 * 2 * 1$ → **Paso recursivo**

Al calcular, por ejemplo, el factorial de 4, decimos que 4! es igual a 4 * 3!. Al factorial de 3, lo calculamos como 3* 2!. Al factorial de 2, como 2 * 1!, y así sucesivamente hasta llegar a un paso básico que detiene la recursividad. Llegamos a definir, entonces, el factorial de un número n en términos del factorial del número $(n-1)$.

Fórmula 4.1

$$n! \begin{cases} 1 & \text{Si } n = 0 \text{ o } n = 1 \quad \textbf{Paso básico} \\ n * (n-1)! & \text{Si } n > 1 \quad \textbf{Paso recursivo} \end{cases}$$

A continuación se presenta el algoritmo recursivo del cálculo del factorial.

Algoritmo 4.1 Factorial_rec

Factorial_rec (*N*)

{Este algoritmo calcula el factorial de un número *N* en forma recursiva, donde *N* es un valor numérico entero, positivo o nulo}

1. *Si* (*N* = 0)
entonces
Hacer Factorial_rec ← 1 {Paso básico}
si no
Hacer Factorial_rec ← *N* * Factorial_rec (*N* – 1)
{Llamada —paso— recursiva}
2. {Fin del condicional del paso 1}

Cuando se realiza una llamada recursiva, se utiliza, en forma implícita, una estructura tipo pila para almacenar las instrucciones pendientes de ejecutar, con todos los valores que tienen las variables o constantes en ese momento. Cuando se termina la ejecución se llega al estado básico, se toma la instrucción que se encuentra en el tope de la pila y se continúa operando. Esta acción se repite hasta que la pila quede vacía.

En la figura 4.3 se puede observar en forma gráfica el seguimiento del algoritmo para $N = 5$.

Observe que en la pila el número que se encuentra entre corchetes en la primera columna de la izquierda hace referencia a la llamada recursiva que se realiza. Este símbolo permite observar el orden en que se realizan las llamadas recursivas. Por ejemplo, [1] expresa que esa es la primera llamada recursiva, [2] representa la segunda llamada y así sucesivamente. Por otra parte, el mismo número entre corchetes a la izquierda relaciona la llamada recursiva con el valor que ingresa al algoritmo en cada llamada.

A continuación se presenta la variante iterativa del cálculo del factorial. Nunca hay que descartar las variantes iterativas de la solución de un problema, porque aun cuando éste se puede resolver naturalmente de forma recursiva, es importante tener siempre en cuenta que la recursión necesita de una pila para su funcionamiento y que ésta consume espacio de memoria. En algunos lenguajes de programación, el espacio dedicado a la pila es muy pequeño, por lo que si el problema que se intenta resolver requiere de una cantidad de espacio mayor —pila— que la que ofrece el lenguaje, el problema no se podrá resolver por falta de memoria. En tal caso, una forma para remediar este inconveniente es utilizando iteratividad en lugar de recursividad.

Algoritmo 4.2 Factorial_ite

Factorial_ite (*N*)

{Este algoritmo calcula el factorial de un número *N* en forma iterativa, donde *N* es un valor numérico entero, positivo o nulo}

{FACT es una variable de tipo entero}

1. Hacer FACT ← 1
2. Mientras ($N > 0$) *Repetir* {Ciclo para calcular *N*!}
   Hacer FACT ← *N* * FACT y *N* ← *N* – 1
3. {Fin del ciclo del paso 2}
4. Escribir FACT

FIGURA 4.3
Funcionamiento interno de la recursión-factorial.

	Factorial rec (*N*)			Pila
Valor inicial	→ 5		[5]	1 * Factorial_rec (0) ← 1
[1]	→ 4	2 = 2 * 1	[4]	2 * Factorial_rec (1) ← 1
[2]	→ 3	6 = 3 * 2	[3]	3 * Factorial_rec (2) ← 2
[3]	→ 2	24 = 4 * 6	[2]	4 * Factorial_rec (3) ← 6
[4]	→ 1	120 = 5 * 24	[1]	5 * Factorial_rec (4) ← 24
[5]	→ 0 → 1			

La siguiente tabla presenta los valores que van adquiriendo las variables, durante el cálculo de 4!.

TABLA 4.1
Cálculo del factorial en forma iterativa

N	FACT
4	1
3	4
2	12
1	24
0	**24**

Ejemplo 4.2

Sucesión de Fibonacci. Otro problema clásico de recursividad es el del cálculo de la sucesión de Leonardo de Pisa, conocido como Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21,..., etcétera.

Algunas propiedades de esta sucesión, según Wikipedia, la enciclopedia libre, son:

- El cociente entre un término y el inmediatamente anterior varía continuamente, pero se estabiliza en un número irracional conocido como *razón áurea* o número áureo, que es la solución positiva de la ecuación $x^2 - x - 1 = 0$, y se puede aproximar por 1618033989.
- Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, 17 = 13 + 3 + 1, 65 = 55 + 8 + 2.
- Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo *m*, para cualquier *m*.
- Si Fibonacci de *p*, *F*(*p*), es un número primo, *p* también lo es, con una única excepción: *F*(4) = 3; 3 es primo, pero 4 no lo es.
- La suma infinita de los términos de la sucesión $F(n)/10^n$ es exactamente 10/89.

El Fibonacci de un número se obtiene de la suma de los dos números Fibonacci anteriores. Por definición:

Fibonacci(0) = 0 → **Paso básico**
Fibonacci(1) = 1 → **Paso básico**
Fibonacci(2) = Fibonacci(1) + Fibonacci(0)
= 1 + 0 = 1
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
= 1 + 1 = 2
Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
= 2 + 1 = 3
...
Fibonacci(*n*) = Fibonacci(*n* – 1) + Fibonacci (*n* – 2) → **Paso recursivo**

La fórmula 4.2 presenta una definición recursiva de la serie de Fibonacci.

Fórmula 4.2

$$\text{Fibonacci}(n) = \begin{cases} n & \text{si} \quad (n = 0) \text{ o } (n = 1) \\ \text{Fibonacci}\,(n-1) + \text{Fibonacci}\,(n-2) & \text{si} \quad n > 1 \end{cases}$$

En este ejemplo el paso básico se presenta cuando $n = 1$ o $n = 0$. En el paso recursivo de la fórmula se utiliza el concepto de Fibonacci aplicado a $(n - 1)$ y $(n - 2)$. Por ser n un número entero positivo serán $(n - 1)$ y $(n - 2)$ valores más cercanos al estado básico.

El algoritmo 4.3 presenta una solución recursiva del cálculo de un número Fibonacci n.

Algoritmo 4.3 Fibonacci_rec

Fibonacci_rec (*N*)

{Este algoritmo calcula el número Fibonacci correspondiente a *N* en forma recursiva, donde *N* es un valor numérico entero, positivo o nulo}

1. *Si* ((*N* = 0) o (*N* = 1))
entonces
Hacer Fibonacci_rec ← *N* {Paso básico}
si no
Hacer Fibonacci_rec ← Fibonacci_rec(*N* – 1) + Fibonacci_rec(*N* – 2)
{Llamadas recursivas}
2. {Fin del condicional del paso 1}

En la figura 4.4 se puede observar el seguimiento del algoritmo para $N = 6$.

Observe que en la pila el número que se encuentra entre corchetes hace referencia a la llamada recursiva que se realiza. Por ejemplo, [1] expresa que ésa es la primera llamada recursiva, [2] la segunda llamada y así sucesivamente. Por otra parte, el mismo número entre corchetes a la izquierda, Fibonacci_rec(*N*), relaciona la llamada recursiva con el valor que ingresa al algoritmo en cada llamada.

Fibonacci representa el caso de un algoritmo que se puede resolver naturalmente de manera recursiva, pero que resulta ineficiente hacerlo de esta forma tanto en cuanto a tiempo como a espacio que se utiliza en la pila para almacenar las llamadas pendientes de ejecutar. Además, observe que en muchas ocasiones se tiene que resolver el mismo

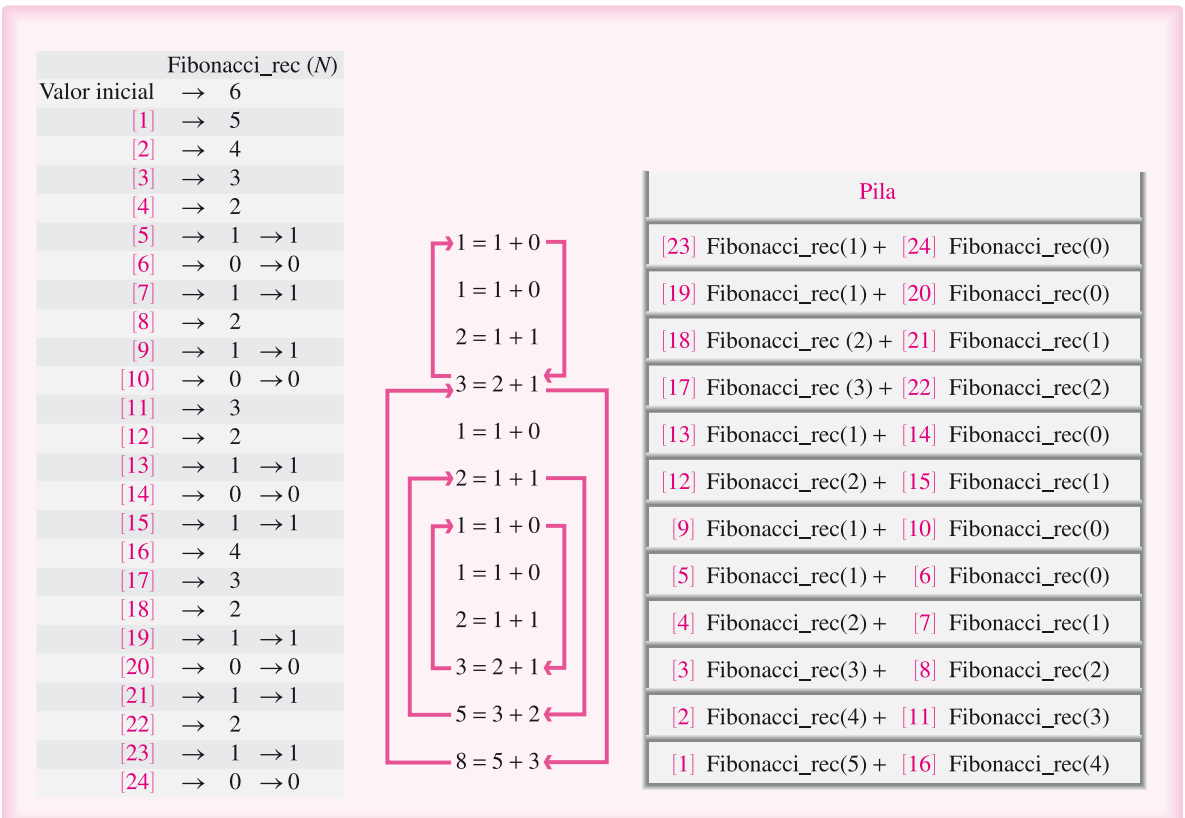


FIGURA 4.4
Funcionamiento interno de la recursión: números Fibonacci.

problema, aun cuando ya tengamos una solución para ese caso, y esto resulta completamente impráctico además de ineficiente. En el ejemplo que se muestra en la figura 4.4, Fibonacci_rec(2) se tuvo que resolver en cinco ocasiones, llamadas recursivas 4, 8, 12, 18 y 22.

A continuación se presenta una variante iterativa para resolver el problema de los números de Fibonacci.

Algoritmo 4.4 Fibonacci_ite

Fibonacci_ite (*N*)

{Este algoritmo calcula el número Fibonacci correspondiente a *N* en forma iterativa. *N* es un valor numérico entero, positivo o nulo}
{FIBO, FIBA, FIBB e *I* son variables de tipo entero}

```
1. Si ((N = 0) o (N = 1))
       entonces
            Hacer FIBO ← N
       si no
            Hacer FIBA ← 0, FIBB ← 1 e I ← 2
2. {Fin del condicional del paso 1}
3. Mientras (I ≤ N) Repetir
            Hacer FIBO ← FIBB + FIBA, FIBA ← FIBB, FIBB ← FIBO e I ← I + 1
4. {Fin del ciclo del paso 3}
5. Escribir FIBO
```

En la tabla 4.2 se presentan los valores que van adquiriendo las variables durante el cálculo del número Fibonacci $N = 6$.

TABLA 4.2
Cálculo de los números Fibonacci en forma iterativa

N	FIBO	FIBA	FIBB	*I*
6		0	1	2
	1	1	1	3
	2	1	2	4
	3	2	3	5
	5	3	5	6
	8	5	8	7

Ejemplo 4.3 Impresión de un arreglo

Dado como dato un arreglo unidimensional de tipo entero, escriba el contenido de las casillas del mismo de **izquierda a derecha**.

A continuación se presenta el algoritmo correspondiente.

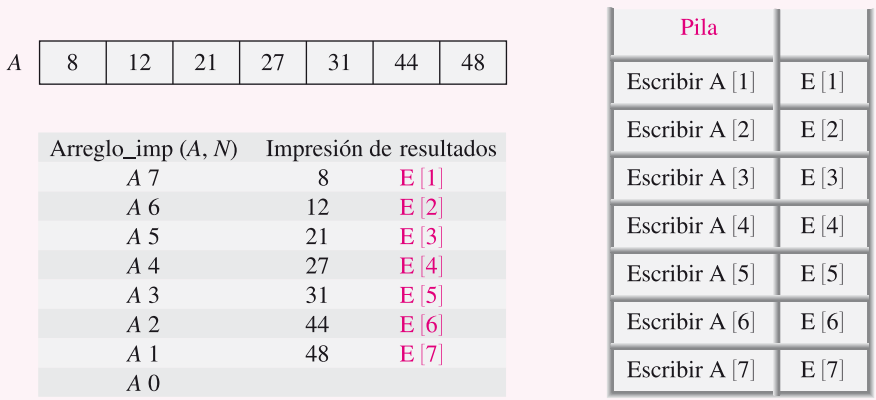
Algoritmo 4.5 Arreglo_imp

Arreglo_imp (*A*, *N*)

{El algoritmo escribe de izquierda a derecha los elementos de un arreglo unidimensional *A* de tipo entero. *N* representa el tamaño del arreglo}

```
1. Si (N ≠ 0) entonces
       Arreglo_imp (A, N – 1)
       Escribir A[N]
2. {Fin del condicional del paso 1}
```

FIGURA 4.5
Funcionamiento interno de la recursión: impresión de arreglos.



En la figura 4.5 se puede observar el seguimiento del algoritmo para un arreglo unidimensional de tipo entero de siete elementos.

Observe que *E*[*N*] en la pila se utiliza para mostrar el orden en que se realizan las llamadas recursivas. Por otra parte, en *Impresión de resultados*, el mismo símbolo se utiliza para relacionar la llamada recursiva con el valor que se imprime.

Ejemplo 4.4

Impresión de un arreglo

Dado como dato un arreglo unidimensional de tipo entero, escriba el contenido de las casillas del mismo de **derecha a izquierda**.

A continuación se presenta el algoritmo correspondiente.

Algoritmo 4.6 Arreglo_imp

Arreglo_imp (*A*, *N*)

{El algoritmo escribe de derecha a izquierda los elementos de un arreglo unidimensional *A* de tipo entero. *N* representa el tamaño del arreglo}

1. *Si* ($N \neq 0$) *entonces*
    Escribir *A*[*N*]
    Arreglo_imp (*A*, *N* – 1)
2. {Fin del condicional del paso 1}

Es importante señalar que, en este caso, a diferencia del algoritmo 4.5, no se utiliza la pila —internamente— para ir guardando instrucciones pendientes de ejecución. Luego de evaluarse la condición, si ésta resulta verdadera se escribe el valor de la posición *N*

del arreglo y se llama nuevamente a la función, ahora con $N - 1$. Una vez que se alcanza el estado básico, el algoritmo termina.

Ejemplo 4.5 Suma de un arreglo

Dado como dato un arreglo unidimensional de tipo entero, obtenga la suma del mismo.

A continuación se presenta el algoritmo correspondiente.

Algoritmo 4.7 Arreglo_sum

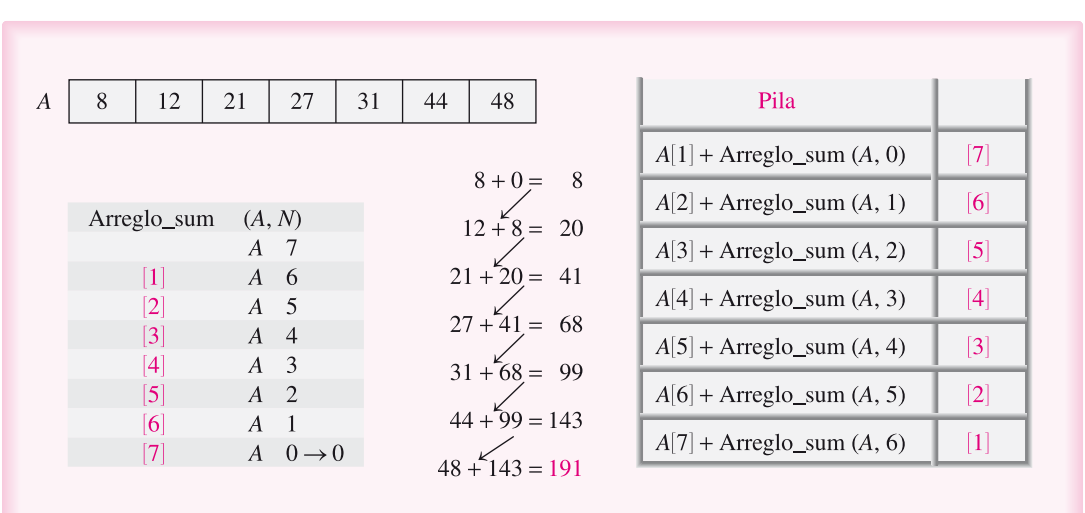
Arreglo_sum (*A*, *N*)

{El algoritmo obtiene la suma de los elementos de un arreglo unidimensional *A* de tipo entero. *N* representa el tamaño del arreglo}

1. *Si* ($N = 0$)
  entonces
    Hacer Arreglo_sum ← 0
  si no
    Hacer Arreglo_sum ← $A[N]$ + Arreglo_sum(A, $N - 1$)
2. {Fin del condicional del paso 1}

En la figura 4.6 se observa el seguimiento del algoritmo para un arreglo unidimensional de tipo entero de siete elementos.

FIGURA 4.6
Funcionamiento interno de la recursión: suma de arreglos.



Observe nuevamente que el número entre corchetes en la *Pila*, se utiliza para mostrar el orden con que se realizan las llamadas recursivas. El número entre corchetes que se encuentra en *Arreglo_sum(A, N)* relaciona la llamada recursiva con los valores que recibe la función.

Ejemplo 4.6

Euclides

El algoritmo de Euclides representa un método efectivo para encontrar el máximo común divisor (mcd) entre dos números enteros positivos. El algoritmo consiste en varias divisiones euclidianas sucesivas. En la primera división se toma como dividendo el mayor de los números y como divisor el otro ahorrando así un paso. Luego el divisor y el resto sirven, respectivamente, de dividendo y divisor de la siguiente división. El proceso se detiene cuando se obtiene un resto nulo.

La fórmula 4.3 presenta la definición recursiva del algoritmo de Euclides. Es importante mencionar que debe cumplirse que $M \geq N$. A continuación se presenta el algoritmo correspondiente:

Fórmula 4.3

$$\text{Euclides}(M, N) \begin{cases} M & \text{Si } N = 0 \\ \text{Euclides}(N, M \text{ MOD } N) & \text{En cualquier otro caso} \end{cases}$$

Algoritmo 4.8 Euclides

Euclides (*M*, *N*)

{Este algoritmo calcula el máximo común divisor de dos números enteros positivos. *M* y *N* son valores numéricos enteros positivos}

1. *Si* (*N* = 0)
    entonces
        Hacer Euclides ← *M*
    si no
        Hacer Euclides ← Euclides (*N*, *M* MOD *N*)
2. {Fin del condicional del paso 1}

En la figura 4.7 se observa el seguimiento del algoritmo para dos corridas diferentes. En la primera, *M* = 2 353 y *N* = 1 651. En la segunda, *M* = 25 680 y *N* = 11 892.

Es importante señalar que, en este caso, no se utiliza la pila internamente para ir guardando instrucciones pendientes de ejecución. Una vez que se alcanza el estado básico, el algoritmo termina dando el resultado final.

FIGURA 4.7
Funcionamiento interno de la recursión: algoritmo de Euclides.

Euclides	(M, N)	
2 353	1 651	
1 651	702	[1]
702	247	[2]
247	208	[3]
208	39	[4]
39	13	[5]
13	0	[6] → **13**

Euclides	(M, N)	
25 680	11 892	
11 892	1 896	[1]
1 896	516	[2]
516	348	[3]
348	168	[4]
168	12	[5]
12	0	[6] → **12**

Ejemplo 4.7 Ackermann

La función de Ackermann, utilizada en la teoría de la computación, es una función recursiva que toma dos números naturales como argumentos y devuelve un número natural. La fórmula 4.4 presenta la definición recursiva de la función de Ackermann:

Fórmula 4.4

$$A(m,n)=\begin{cases} n+1, & si\ m=0; \\ A(m-1,1), & si\ m>0\ \text{y}\ n=0; \\ A\big(m-1,A(m,n-1)\big), & si\ m>0\ \text{y}\ n>0 \end{cases}$$

Según Wikipedia, la función crece rápidamente. Para darse una idea de la magnitud de los valores que aparecen de la fila 4 en adelante cuando m es igual a 4, se puede destacar que, por ejemplo, $A(4, 2)$ es mayor que el número de partículas que forman el universo elevado a la potencia 200 y el resultado de $A(5, 2)$ no se puede escribir dado que no cabría en el universo físico. En general, por encima de la fila 4 ya no es posible escribir todos los dígitos del resultado de la función.

En la tabla 4.3 se presenta, con el objeto de que el lector pueda observar la complejidad de la función cuando los valores se incrementan, el resultado de la función de Ackermann y el nivel de profundidad que se alcanza cuando M y N toman ciertos valores.

TABLA 4.3
Ackermann y valores de *M* y *N*

M	N	Ackermann	Profundidad
1	0	2	2
2	0	3	5
3	0	5	15
2	1	5	14

(*continúa*)

TABLA 4.3
(*continúación*)

M	*N*	Ackermann	Profundidad
2	2	7	27
2	3	9	44
2	4	11	65
2	7	17	152
3	5	253	42 438
3	6	509	172 233
3	10	8 189	44 698 325

A continuación se presenta el algoritmo que resuelve el problema de Ackermann.

Algoritmo 4.9 Ackermann

Ackermann (*M*, *N*)

{Este algoritmo calcula la función de Ackermann. *M* y *N* son valores numéricos enteros, positivos o nulos}

1. *Si* (*M* = 0)
entonces
Hacer Ackermann ← (*N* + 1) {Estado básico}
si no
1.1 *Si* (*N* = 0)
entonces
Hacer Ackermann ← Ackermann (*M* – 1, 1)
si no
Hacer Ackermann ← Ackermann (*M* – 1, Ackermann (*M*, *N* – 1))
1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

En la figura 4.8 se puede observar el seguimiento del algoritmo de Ackermann para valores de *M* = 2 y *N* = 2.

Considere nuevamente que el número entre corchetes, tanto en la pila como en *Ackermann(N)*, se utiliza para observar el orden con que se realizan las llamadas recursivas.

Ejemplo 4.8

Algoritmo de partición

Este algoritmo, recientemente descubierto, permite conocer todas las formas en las cuales un número entero positivo puede ser descompuesto como la suma de varios suman-

Ackermann (*M*, *N*)			
Valor inicial	→	(2, 2)	
[1]	→	(2, 1)	
[2]	→	(2, 0)	
Viene de *N* = 0	→	(1, 1)	
[3]	→	(1, 0)	
Viene de *N* = 0	→	(0, 1)	→ 2
[4]	→	(0, 2)	→ 3
[5]	→	(1, 3)	
[6]	→	(1, 2)	
[7]	→	(1, 1)	
[8]	→	(1, 0)	
Viene de *N* = 0	→	(0, 1)	→ 2
[9]	→	(0, 2)	→ 3
[10]	→	(0, 3)	→ 4
[11]	→	(0, 4)	→ 5
[12]	→	(1, 5)	
[13]	→	(1, 4)	
[14]	→	(1, 3)	
[15]	→	(1, 2)	
[16]	→	(1, 1)	
[17]	→	(1, 0)	
Viene de *N* = 0	→	(0, 1)	→ 2
[18]	→	(0, 2)	→ 3
[19]	→	(0, 3)	→ 4
[20]	→	(0, 4)	→ 5
[21]	→	(0, 5)	→ 6
[22]	→	(0, 6)	→ 7

	Pila		
2	[18]	Ackermann(0, Ackermann(1, 0))	[17]
3	[19]	Ackermann(0, Ackermann(1, 1))	[16]
4	[20]	Ackermann(0, Ackermann(1, 2))	[15]
5	[21]	Ackermann(0, Ackermann(1, 3))	[14]
7 ← 6	[22]	Ackermann(0, Ackermann(1, 4))	[13]
2	[9]	Ackermann(0, Ackermann(1, 0))	[8]
3	[10]	Ackermann(0, Ackermann(1, 1))	[7]
4	[11]	Ackermann(0, Ackermann(1, 2))	[6]
2	[4]	Ackermann(0, Ackermann(1, 0))	[3]
3	[5]	Ackermann(1, Ackermann(2, 0))	[2]
5	[12]	Ackermann(1, Ackermann(2, 0))	[1]

FIGURA 4.8
Funcionamiento interno de la recursión: función de Ackermann.

dos. La solución encontrada a este problema puede servir tanto a la física de partículas como a la seguridad informática.

Según el *New Scientist*, Karl Mahlburg, estudiante de la Universidad de Wisconsin, ha dedicado un año para solucionar el problema que implica trabajar con patrones de números. "He llenado de cálculos y ecuaciones cuaderno tras cuaderno", dice Mahlburg.

Los patrones fueron descubiertos por primera vez por Ramanujan (1887-1920), un hindú que fue expulsado de la universidad por descuidar los estudios de todo lo que no fueran matemáticas. Su pasión por las matemáticas le llevó a seguir investigando y a escribir a matemáticos ingleses exponiendo sus teorías. Autodidacto, empezó a trabajar en el *Madras Port Trust*, y luego, en 1914, fue admitido en la Universidad de Cambridge. Posteriormente fue elegido miembro de la *Royal Society* y del *Trinity College*. Pocos años después regresó a la India, donde falleció tras una misteriosa enfermedad.

A continuación se presenta la fórmula correspondiente.

Fórmula 4.5

$$\text{Partición}(M, N) \begin{cases} 1 & \text{Si } M = 1 \\ 1 & \text{Si } N = 1 \\ \text{Partición}(M, M) & \text{Si } M < N \\ 1 + \text{Partición}(M, -1\ M) & \text{Si } M = N \\ \text{Partición}(M, N-1) + \text{Partición}(M-N, N) & \text{Si } M > N \end{cases}$$

A continuación se presenta el algoritmo correspondiente.

Algoritmo 4.10 Partición

Partición (*M*, *N*)

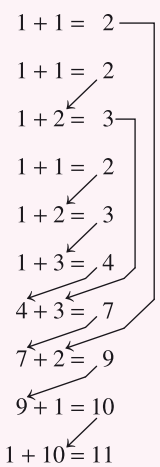
{Este algoritmo calcula de cuantas formas diferentes se puede descomponer un número entero positivo. *M* y *N* son valores numéricos enteros positivos. Al iniciar el algoritmo *M* = *N*}

```
1. Si ((M = 1) o (N = 1))
       entonces
           Hacer Partición ← 1  {Estado básico}
       si no
           1.1 Si (M < N)
                   entonces
                       Hacer Partición (M, M)
                   si no
                       1.1.1 Si (M = N)
                                 entonces
                                     Hacer Partición ← 1 + Partición (M, M – 1)
                                 si no
                                     Hacer Partición ← Partición (M, N – 1) + Partición (M – N, N)
                       1.1.2 {Fin del condicional del paso 1.1.1}
           1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

En la figura 4.9 se presenta el seguimiento del algoritmo para obtener el número de formas en que se puede descomponer el número 6, *M* = *N* = 6.

Observe nuevamente que el número entre corchetes tanto en la pila como en *Partición(M, N)* se utiliza para observar el orden con que se realizan las llamadas recursivas.

Partición (*M*, *N*)	
	6, 6
[1]	6, 5
[2]	6, 4
[3]	6, 3
[4]	6, 2
[5]	6, 1 → 1
[6]	4, 2
[7]	4, 1 → 1
[8]	2, 2
[9]	2, 1 → 1
[10]	3, 3
[11]	3, 2
[12]	3, 1 → 1
[13]	1, 2
$M < N$	1, 1 → 1
[14]	2, 4
$M < N$	2, 2
[15]	2, 1
[16]	1, 5
$M < N$	1, 1 → 1



	Pila	
	1 + Partición (2, 1)	[15]
[12]	Partición (3, 1) + Partición (1, 2)	[13]
	1 + Partición (3, 2)	[11]
	1 + Partición (2, 1)	[9]
[7]	Partición (4, 1) + Partición (2, 2)	[8]
[5]	Partición (6, 1) + Partición (4, 2)	[6]
[4]	Partición (6, 2) + Partición (3, 3)	[10]
[3]	Partición (6, 3) + Partición (2, 4)	[14]
[2]	Partición (6, 4) + Partición (1, 5)	[16]
	1 + Partición (6, 5)	[1]

FIGURA 4.9
Funcionamiento interno de la recursión: algoritmo de partición.

Ejemplo 4.9 Los números de Catalan

Estos números se utilizan en una gran variedad de problemas de combinatoria. Tienen varias aplicaciones; por ejemplo, dados como datos n matrices, permiten encontrar el número de formas en que se podrían multiplicar. Otra aplicación consiste en determinar el número de formas en que un polígono con $n + 2$ lados se puede descomponer en n triángulos.

En combinatoria los números de Catalan forman una secuencia de números naturales que aparece en varios problemas de conteo que habitualmente son recursivos. Obtienen su nombre del matemático belga Eugène Charles Catalan (1814-1894).

El enésimo número de Catalan se obtiene según la siguiente fórmula:

Fórmula 4.6

$$C_n = \frac{1}{n+1}\binom{2n}{n} \qquad \text{con } n \geq 0$$

Una manera recursiva de expresar los números de Catalan se observa en la siguiente fórmula.

Fórmula 4.7

$$\text{Catalan}(N) \begin{cases} 1 & \text{Si } N = 1 \\ \sum_{I=1}^{N-1} \text{Catalan}(I) * \text{Catalan}(N-I) & \text{En cualquier otro caso} \end{cases}$$

Los primeros números de Catalan siguiendo la fórmula recursiva son:

1, 1, 2, 5, 14, 42, 132, 429, 1 430,...

El problema, sin embargo, lo representa la gran cantidad de llamadas recursivas que se necesitan realizar para alcanzar estos resultados. Por ejemplo, si $N = 5$ que arrojaría como valor el número 14, se necesitaría casi una centena de llamadas recursivas.

A continuación se presenta el algoritmo que resuelve este problema.

Algoritmo 4.11 Catalan

Catalan (*N*)

{Este algoritmo obtiene el resultado de los números de Catalan. *N* es un valor numérico entero positivo.}

{*I* y *S* son variables de tipo entero}

1. *Si* (*N* = 1)
  entonces
    Hacer Catalan ← 1 {Estado básico}
  si no
    Hacer *S* ← 0
  1.1 Repetir con *I* desde 1 hasta *N*
      Hacer Catalan ← *S* + Catalan (*I*) * Catalan (*N* – *I*)
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del condicional del paso 1}

La figura 4.10 muestra el seguimiento del algoritmo de Catalan para $N = 3$.

Ejemplo 4.10

Coeficientes binomiales

El triángulo de Pascal es un triángulo de números enteros, infinito y simétrico cuyas diez primeras líneas se pueden observar en la figura 4.11:

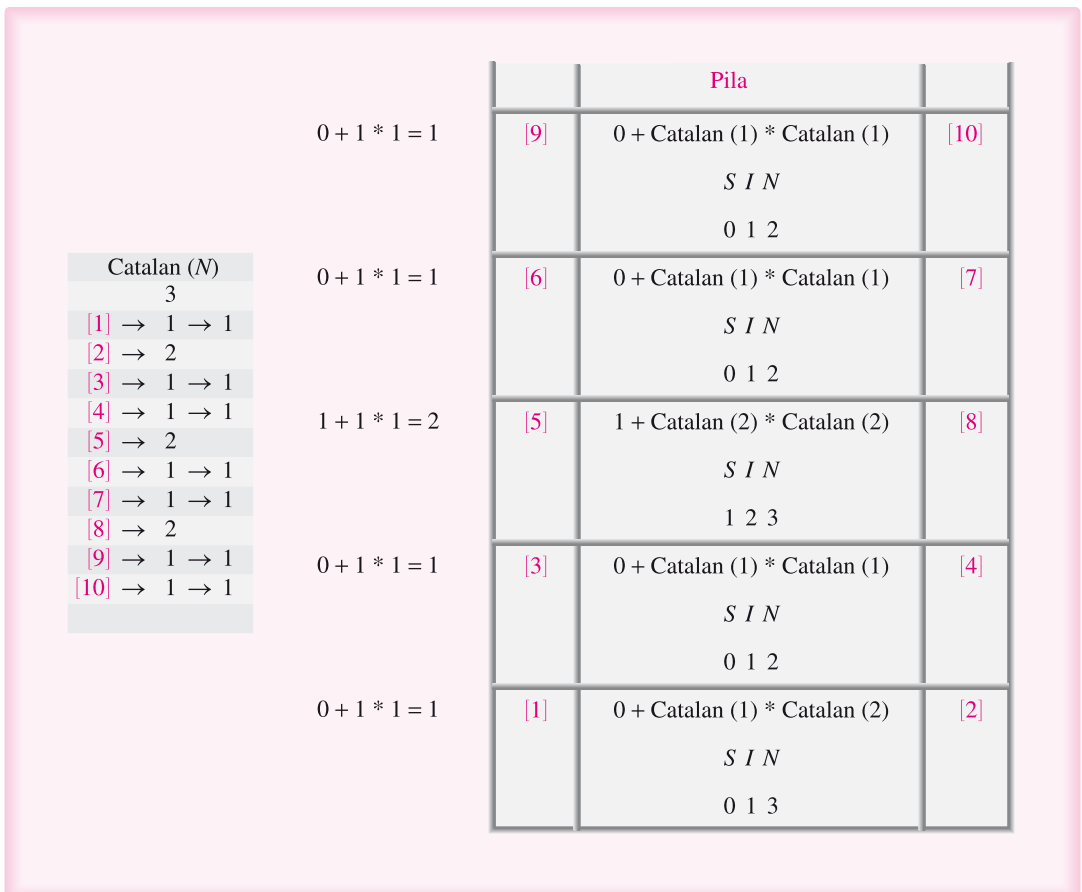
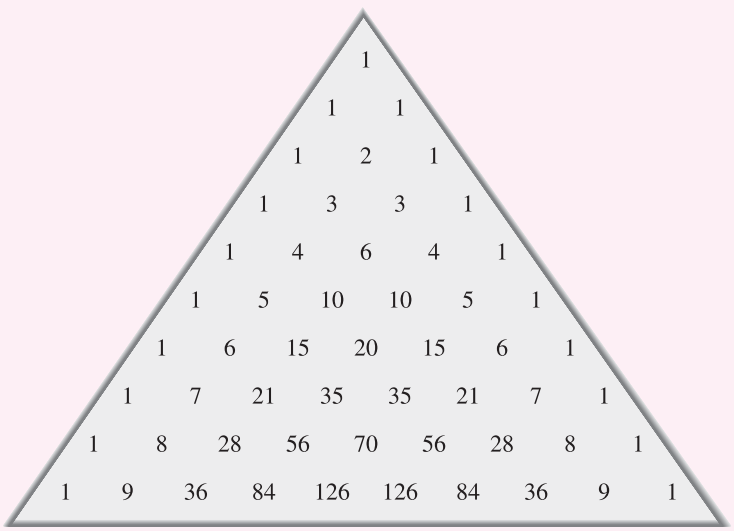


FIGURA 4.10
Funcionamiento interno de la recursión: los números de Catalan.

El triángulo se construye de la siguiente manera: primero se escribe el 1 del primer renglón, la cima del triángulo. A partir de la siguiente línea entre cada número se deja un número determinado de espacios en blanco, tres en este caso, para dar claridad. Cada número que se escribe en un renglón es la suma de los dos que se encuentran en el renglón de arriba. Por ejemplo, el 2 que se encuentra en medio del segundo renglón representa la suma de los dos números que se encuentran arriba de él. El 6 del quinto renglón se deriva de la suma de los dos números de arriba, 3 y 3. El primer 10 del sexto renglón se deriva de la suma de los dos números que se encuentran arriba, 4 y 6, y así sucesivamente.

Se puede observar, además, que los lados exteriores del triángulo están formados por 1. Si le quitamos el lado del costado izquierdo, entonces nos quedan los números naturales en orden creciente del 1 al 9. Podemos hacer lo mismo con el otro costado, ya que existe un eje de simetría vertical que pasa por el vértice del triángulo.

FIGURA 4.11
Triángulo de Pascal.



La fórmula que da el desarrollo de $(a + b)^2$ según las potencias crecientes de a y decrecientes de b se llama binomio de Newton. En esta expresión lo único que se desconoce son los coeficientes de los monomios $a^k\ b^{n-k}$. La definición habitual de los coeficientes binomiales se expresa en términos de factoriales, como se puede observar en la fórmula 4.8.

Fórmula 4.8

$$C_n^k = \frac{n!}{k!(n-k)!}(0 \le k \le n)$$

Sin embargo, también es posible presentar una definición recursiva, como se muestra en la siguiente fórmula.

Fórmula 4.9

$$CB(N, K) \begin{cases} \rightarrow 1 \quad \text{Si } CB(N, 0) \text{ o } C(N, N) \\ \rightarrow CB(N-1, K) + CB(N-1, K-1) \text{ Si } N > K > 0 \end{cases}$$

El algoritmo 4.12 describe la solución del problema de los coeficientes binomiales.

Algoritmo 4.12 CB

***CB* (*N*, *K*)**

{Este algoritmo obtiene el resultado de los coeficientes binomiales. *N* y *K* son valores numéricos enteros positivos o nulos}

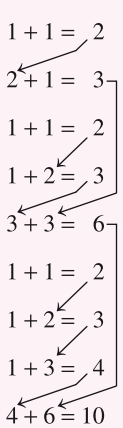
1. *Si* ((*K* = 0) o (*N* = *K*))
entonces
Hacer CB ← 1 {Estado básico}
si no
Hacer CB ← CB(*N* – 1, *K* – 1) + CB(*N* – 1, *K*)
2. {Fin del condicional del paso 1}

En la figura 4.12, se puede observar el seguimiento del algoritmo para *N* = 5 y *K* = 2.

No olvide que los números que aparecen entre corchetes son para ilustrar el orden en que se realizan las llamadas recursivas.

FIGURA 4.12
Funcionamiento interno de la recursión: coeficientes binomiales.

CB (*N*, *K*)			
	5	2	
[1]	4	1	
[2]	3	0	→ 1
[3]	3	1	
[4]	2	0	→ 1
[5]	2	1	
[6]	1	0	→ 1
[7]	1	1	→ 1
[8]	4	2	
[9]	3	1	
[10]	2	0	→ 1
[11]	2	1	
[12]	1	0	→ 1
[13]	1	1	→ 1
[14]	3	2	
[15]	2	1	
[16]	1	0	→ 1
[17]	1	1	→ 1
[18]	2	2	→ 1



	Pila	
[16]	CB(1, 0) + CB(1, 1)	[17]
[15]	CB(2, 1) + CB(2, 2)	[18]
[12]	CB(1, 0) + CB(1, 1)	[13]
[10]	CB(2, 0) + CB(2, 1)	[15]
[9]	CB(3, 1) + CB(3, 2)	[14]
[6]	CB(1, 0) + CB(1, 1)	[7]
[4]	CB(2, 0) + CB(2, 1)	[5]
[2]	CB(3, 0) + CB(3, 1)	[3]
[1]	CB(4, 1) + CB(4, 2)	[8]

4.2 EL PROBLEMA DE LAS TORRES DE HANOI

El problema de las Torres de Hanoi es un problema clásico de recursión, ya que pertenece a la clase de problemas cuya solución se simplifica notablemente al utilizar este concepto.

Se tienen tres torres y un conjunto de N discos de diferentes tamaños. Cada uno de ellos tiene una perforación en el centro que le permite deslizarse por cualquiera de las torres. Inicialmente los N discos están ordenados de mayor a menor en una de las torres.

El objetivo del problema consiste en pasar los N discos de la torre de origen a una torre destino, utilizando la otra torre disponible, como auxiliar. A continuación se presentan las reglas que se deben respetar en cada movimiento.

1. En cada movimiento sólo puede intervenir un disco; por lo tanto, siempre será el disco superior el que pueda moverse.
2. No puede quedar un disco sobre otro de menor tamaño.

Supongamos que las torres se identifican con los nombres A, B y C. Los discos inicialmente se encuentran en la torre A —origen—. El objetivo, como señalamos anteriormente, consiste en transferir todos los discos a la torre B —destino—, utilizando la torre C como auxiliar. En las figuras 4.13*a* y *b* se presentan el estado inicial y el estado final, respectivamente, del problema de las Torres de Hanoi para cinco discos.

Es importante observar, si se analiza detenidamente el problema, que éste se puede descomponer en tres subproblemas, uno de los cuales, el segundo, se considera trivialidad porque implica únicamente un movimiento. Se muestran a continuación los diferentes subproblemas:

1. Transferir $(N - 1)$ discos de la torre A —origen— a la torre C —auxiliar—.
2. Mover un disco de la torre A —origen— a la torre B —destino—.
3. Transferir $(N - 1)$ discos de la torre C —auxiliar— a la torre B —destino—.

En la figura 4.14 se presenta la solución al problema de las Torres de Hanoi para dos discos, $N = 2$.

Observe que se realizan tres movimientos:

Mover de A a C
Mover de A a B
Mover de C a B

En la figura 4.15, por otra parte, se ilustra la solución del problema de las Torres de Hanoi para tres discos, $N = 3$. Primero se transfieren dos $(N - 1)$ discos de la torre A a la torre auxiliar C (figuras 4.15*b*, *c* y *d*). Posteriormente se realiza el movimiento del disco de la torre A a la torre destino B (figura 4.15*e*). Finalmente se resuelve el tercer subproblema, se transfieren $(N - 1)$ discos de la torre C —auxiliar— a la torre A (figuras 4.15*f, g*, *h*).

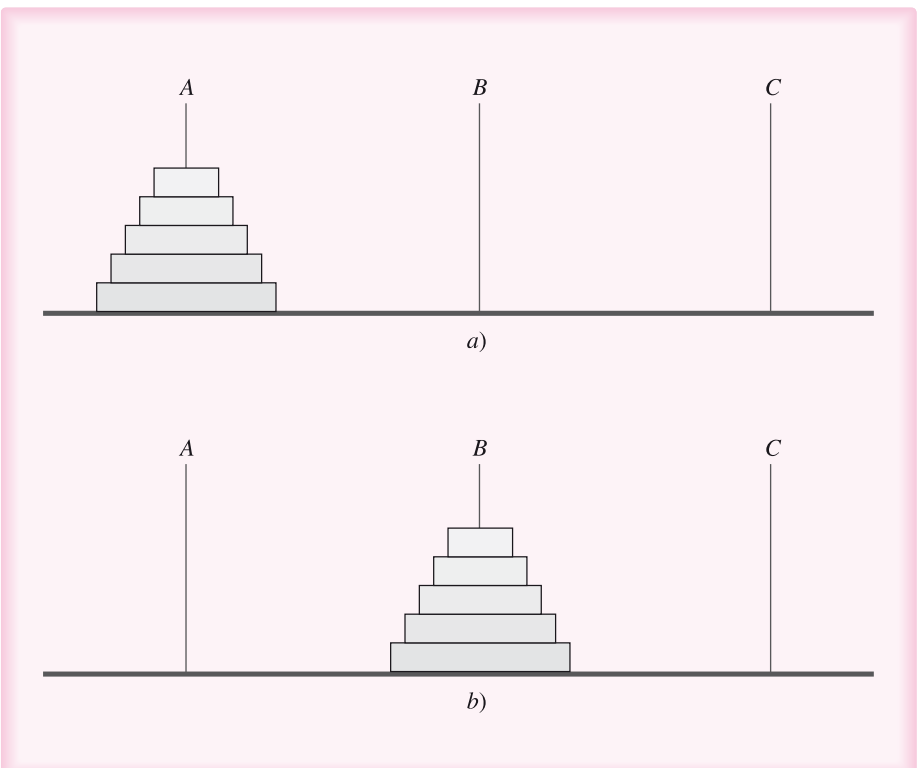


FIGURA 4.13
Torres de Hanoi.
a) Estado inicial.
b) Estado final.

Los movimientos que se realizan para resolver este problema son: mover de *A* a *C*

Mover de *A* a *B*
Mover de *B* a *C*
Mover de *A* a *B*
Mover de *C* a *A*
Mover de *C* a *B*
Mover de *A* a *B*

Luego de realizar numerosas pruebas para distintos valores de *N*, se puede concluir que para cualquier *N*, el número de movimientos (*NM*) está dado por la siguiente fórmula:

$$NM = 2^n - 1 \qquad \textbf{Fórmula 4.10}$$

Así, por ejemplo, para cinco discos se efectuarán 31 movimientos, para diez discos 1 023 movimientos y para 15 discos 32 767 movimientos.

A continuación se presenta el algoritmo recursivo que permite resolver este problema.

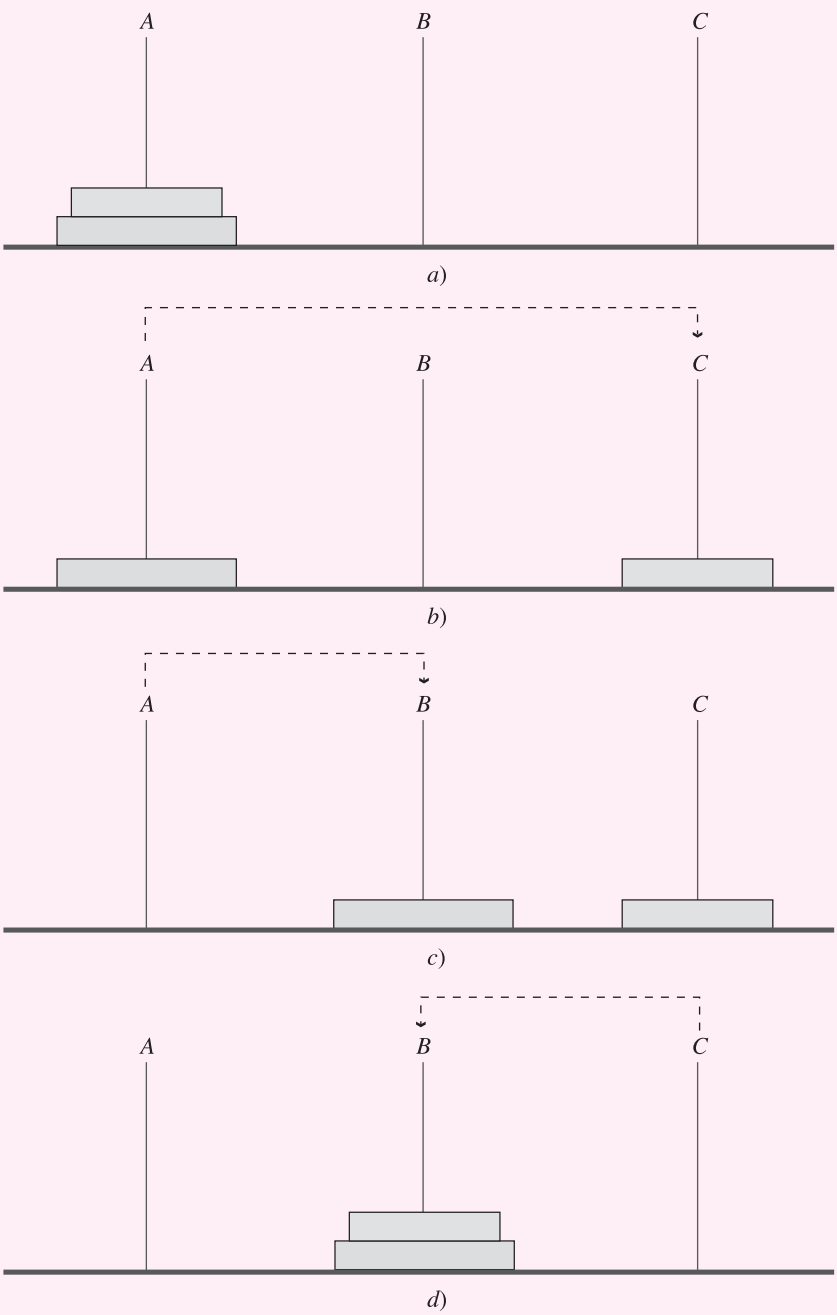


FIGURA 4.14

Torres de Hanoi ($N = 2$). *a*) Estado inicial. *b*) Luego de mover un disco de *A* a *C*. *c*) Luego de mover un disco de *A* a *B*. *d*) Estado final.

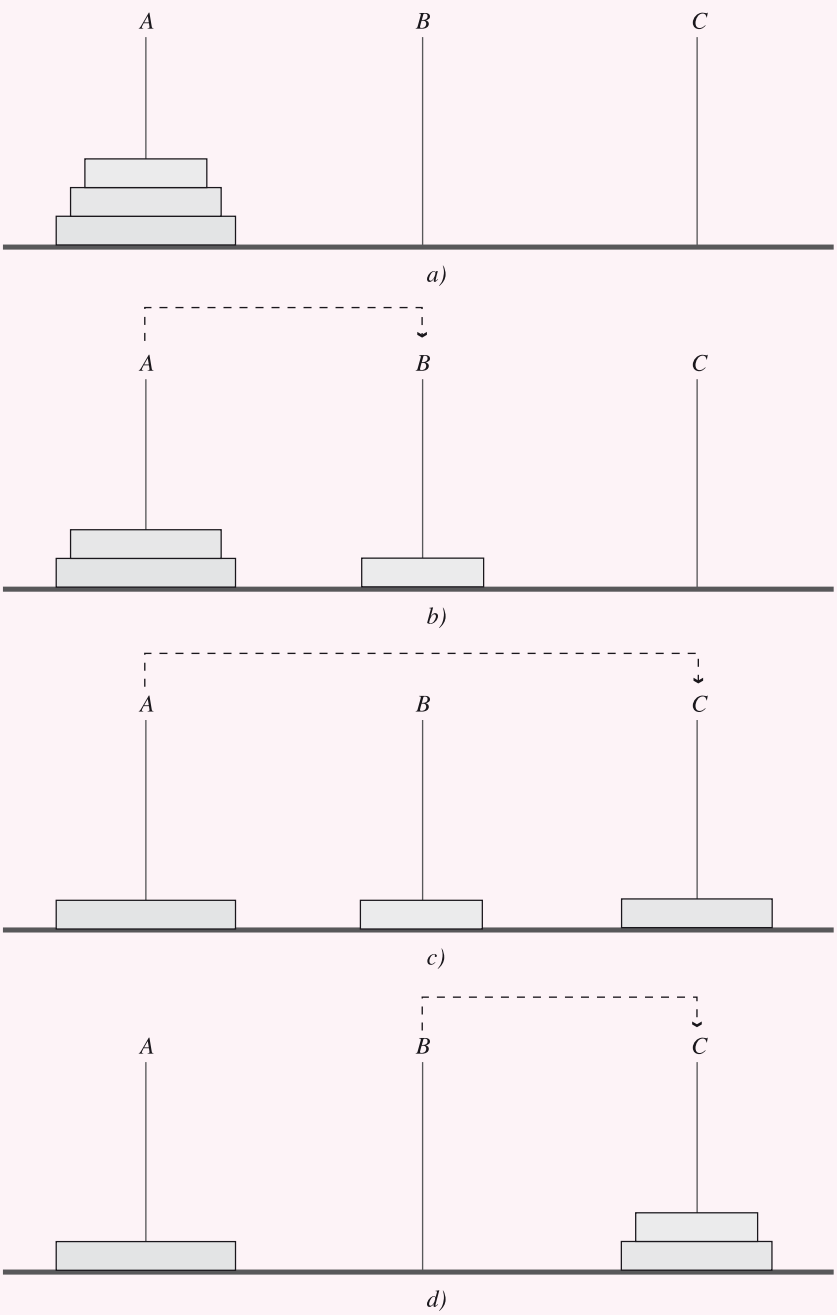


FIGURA 4.15

Torres de Hanoi ($N = 3$).
a) Estado inicial.
b) Luego de mover un disco de *A* a *B*.
c) Luego de mover un disco de A a *C*.
d) Luego de mover un disco de *B* a *C*.

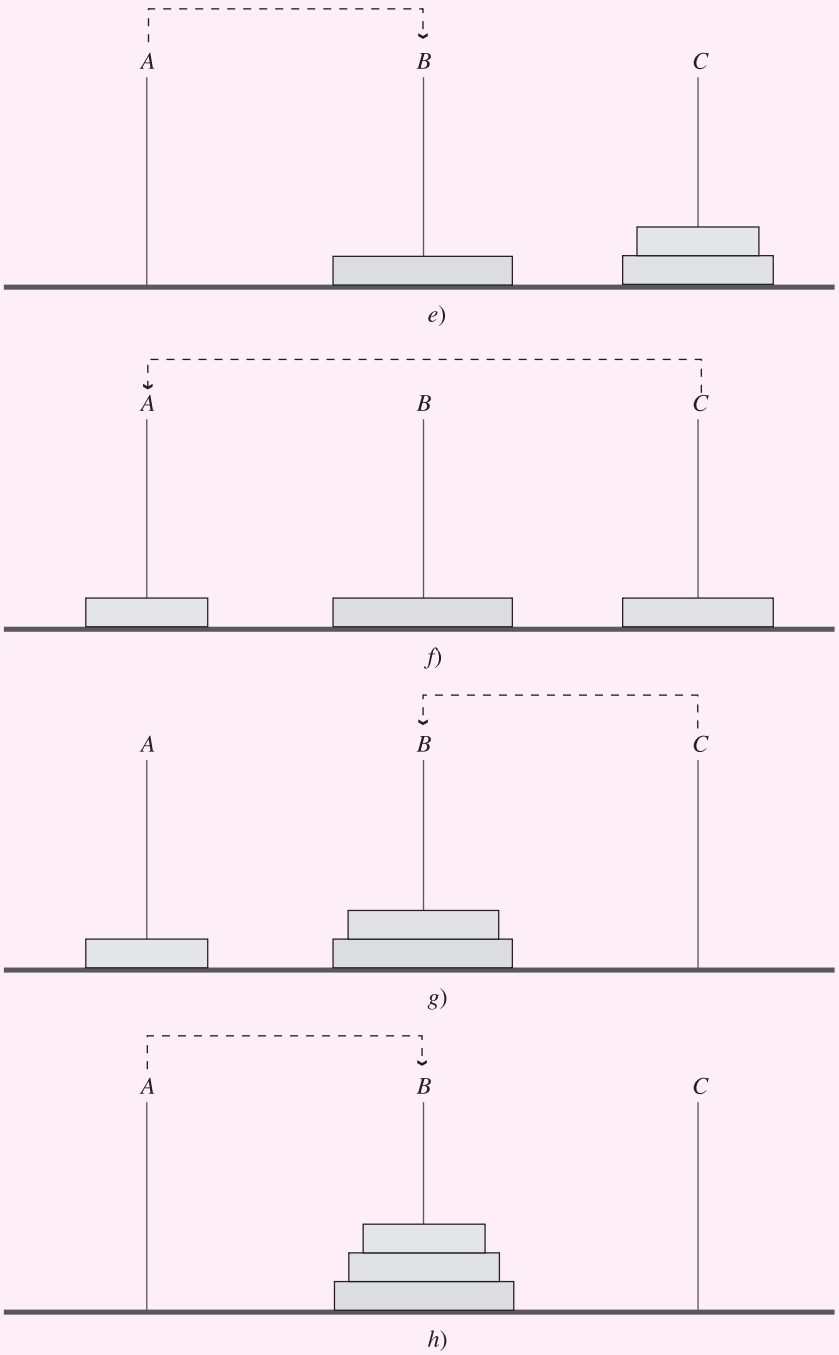


FIGURA 4.15
(*continuación*)
e) Luego de mover un disco de *A* a *B*.
f) Luego de mover un disco de *C* a *A*.
g) Luego de mover un disco de *C* a *B*.
h) Estado final.

Algoritmo 4.13 Hanoi

Hanoi (*N*, ORIGEN, DESTINO, AUXILIAR)

{Este algoritmo obtiene y escribe los movimientos que se deben realizar para transferir los *N* discos de la torre ORIGEN a la torre DESTINO, con ayuda de la torre AUXILIAR}

1. *Si* (*N* = 1)
entonces
Escribir "Mover un disco de ORIGEN a DESTINO" {Estado básico}
si no
{Mover *N* – 1 discos de la torre ORIGEN a la torre AUXILIAR}
Hanoi (*N* – 1, ORIGEN, AUXILIAR, DESTINO)
Escribir "Mover un disco de ORIGEN a DESTINO"
{Mover *N* – 1 discos de la torre AUXILIAR a la torre DESTINO}
Hanoi (*N* – 1, AUXILIAR, DESTINO, ORIGEN)
2. {Fin del condicional del paso 1}

En la figura 4.16 se muestra el seguimiento del algoritmo para cuatro discos.

FIGURA 4.16
Funcionamiento interno de la recursión: torres de Hanoi.

Hanoi (*N*, *A*, *B*, *C*)				
4	*A*	*B*	*C*	
3	*A*	*C*	*B*	
2	*A*	*B*	*C*	
1	*A*	*C*	*B*	[M1] : $A \rightarrow C$
[1] 1	*C*	*B*	*A*	[M3] : $C \rightarrow B$
[2] 2	*B*	*C*	*A*	
1	*B*	*A*	*C*	[M5] : $B \rightarrow A$
[3] 1	*A*	*C*	*B*	[M7] : $A \rightarrow C$
[4] 3	*C*	*B*	*A*	
2	*C*	*A*	*B*	
1	*C*	*B*	*A*	[M9] : $C \rightarrow B$
[5] 1	*B*	*A*	*C*	[M11] : $B \rightarrow A$
[6] 2	*A*	*B*	*C*	
1	*A*	*C*	*B*	[M13] : $A \rightarrow C$
[7] 1	*C*	*B*	*A*	[M15] : $C \rightarrow B$

Movimientos			
[M1]	$A \rightarrow C$	[M9]	$C \rightarrow B$
[M2]	$A \rightarrow B$	[M10]	$C \rightarrow A$
[M3]	$C \rightarrow B$	[M11]	$B \rightarrow A$
[M4]	$A \rightarrow C$	[M12]	$C \rightarrow B$
[M5]	$B \rightarrow A$	[M13]	$A \rightarrow C$
[M6]	$B \rightarrow C$	[M14]	$A \rightarrow B$
[M7]	$A \rightarrow C$	[M15]	$C \rightarrow B$
[M8]	$A \rightarrow B$		

	Pila	
[7]	Escribir $A \rightarrow B$ 1 *C* *B* *A*	M [14]
[5]	Escribir $C \rightarrow A$ 1 *B* *A* *C*	M [10]
[6]	Escribir $C \rightarrow$ B 2 *A* *B* *C*	M [12]
[3]	Escribir $B \rightarrow$ C 1 *A* *C* *B*	M [6]
[1]	Escribir $A \rightarrow B$ 1 *C* *B* *A*	M [2]
[2]	Escribir $A \rightarrow C$ 2 *B* *C* *A*	M [4]
[4]	Escribir $A \rightarrow B$ 3 *C* *B* *A*	M [8]

El algoritmo recursivo anterior ofrece una solución clara y compacta al problema de las Torres de Hanoi. Sin embargo, es posible también presentar una solución iterativa a este problema, pero es conveniente mencionar que en este caso es más complicada y extensa.

Un subprograma generalmente trabaja con variables locales y parámetros. Cuando se hace una llamada recursiva al subprograma, los valores actuales de variables y parámetros se deben conservar. Además se debe almacenar la dirección a la cual se tendrá que regresar el control una vez que se termina de ejecutar.

El algoritmo anterior trabaja con cuatro parámetros: *N*, ORIGEN, DESTINO y AUXILIAR. Para construir el algoritmo iterativo, si se quieren conservar los valores de los parámetros en cada llamada recursiva, se deberá definir una pila para cada uno de ellos. Otra alternativa sería definir una única pila, en la que cada elemento fuera capaz de almacenar los cuatro parámetros. En el algoritmo que se va a presentar se trabaja con cuatro pilas:

PILAN: para almacenar imágenes de *N*.
PILAO: para almacenar imágenes de ORIGEN.
PILAD: para almacenar imágenes de DESTINO.
PILAX: para almacenar imágenes de AUXILIAR.

También será necesario manejar un TOPE para las pilas. El siguiente algoritmo presenta una solución iterativa al problema de las Torres de Hanoi.

Algoritmo 4.14 Hanoi_ite

Hanoi_ite (*N*, ORIGEN, DESTINO, AUXILIAR)

{Este algoritmo resuelve el problema de las Torres de Hanoi de manera no recursiva. ORIGEN, DESTINO y AUXILIAR son parámetros que representan las tres torres. *N* simboliza el número de discos}
{PILAN, PILAO, PILAD y PILAX son estructuras de datos tipo pila. TOPE es una variable de tipo entero. VARAUX es una variable de tipo carácter. BAND es una variable de tipo booleano}

1. Hacer TOPE ← 0 y BAND ← FALSO
2. Mientras ((*N* > 0) y (BAND = FALSO)) *Repetir*
  2.1 Mientras (*N* > 1) *Repetir*
    Hacer TOPE ← TOPE + 1, PILAN[TOPE] ← *N*, PILAO[TOPE] ← ORIGEN,
      PILAD[TOPE] ← DESTINO, PILAX[TOPE] ← AUXILIAR
    {Simula llamada a Hanoi con *N* – 1, ORIGEN, AUXILIAR y DESTINO}
    Hacer *N* ← *N* – 1, VARAUX ← DESTINO, DESTINO ← AUXILIAR
      y AUXILIAR ← VARAUX
  2.2 {Fin del ciclo del paso 2.1}
    Escribir "Mover un disco de ORIGEN a DESTINO"
    Hacer BAND ← VERDADERO
  2.3 *Si* (TOPE > 0) {Las pilas no están vacías}

entonces
{Se extraen los elementos del TOPE de la pilas}
Hacer $N \leftarrow$ PILAN[TOPE], ORIGEN $\leftarrow$ PILAO[TOPE],
DESTINO $\leftarrow$ PILAD[TOPE], AUXILIAR $\leftarrow$ PILAA[TOPE]
y TOPE $\leftarrow$ TOPE – 1
Escribir "Mover un disco de ORIGEN a DESTINO"
{Simula llamada a Hanoi con N – 1, AUXILIAR, DESTINO y ORIGEN}
Hacer $N \leftarrow N$ – 1, VARAUX $\leftarrow$ ORIGEN, ORIGEN $\leftarrow$ AUXILIAR,
AUXILIAR $\leftarrow$ VARAUX y BAND $\leftarrow$ FALSO
2.4 {Fin del condicional del paso 2.3}
3. {Fin del ciclo del paso 2}

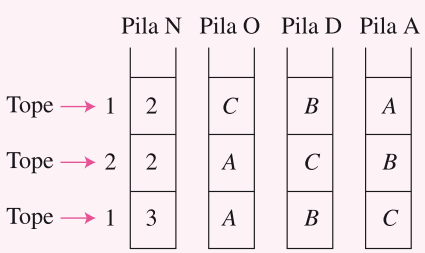
En la figura 4.17 se muestra el seguimiento del algoritmo para tres discos.

FIGURA 4.17.
Seguimiento del algoritmo iterativo: Torres de Hanoi.

N	ORIGEN	DESTINO	AUXILIAR	TOPE	BAND	VARAUX
3	*A*	*B*	*C*	0	F	
2		*C*	*B*	1		*B*
1		*B*	*C*	2	V	*C*
2	*A*	*C*	*B*	1		
1	*B*		*A*		F	*A*
3	*A*	*B*	*C*	0	V	
2	*C*		*A*		F	*A*
1		*A*	*B*	1	V	*B*
2	*C*	*B*	*A*	0		
1	*A*		*C*		F	*C*
					V	

Nota: en el lugar de VERDADERO se escribe V, en el lugar de FALSO F.

Movimientos
$A \rightarrow B$
$A \rightarrow C$
$B \rightarrow C$
$A \rightarrow B$
$C \rightarrow A$
$C \rightarrow B$
$A \rightarrow B$
$A \rightarrow C$



4.3 RECURSIVIDAD EN ÁRBOLES

Los árboles representan las estructuras de datos dinámicas y no lineales más interesantes de computación. La estructura de **árboles balanceados** o **AVL** es la estructura de datos más eficiente para trabajar en la memoria rápida de la computadora. Por otra parte, la estructura de datos **Árboles-B**, representa la estructura de datos más eficiente para trabajar en memoria secundaria —dispositivos de almacenamiento secundario—.

Los árboles son una estructura inherentemente recursiva y todas las operaciones que se realizan en árboles se deben programar en forma recursiva. A diferencia de otras estructuras de datos en las cuales las operaciones se pueden implementar tanto en forma recursiva como iterativa, independientemente de las diferencias que se pueden observar con respecto a la eficiencia, en los árboles sólo se puede trabajar de manera recursiva.

En el capítulo correspondiente a árboles el lector podrá practicar en forma intensa la recursividad.

4.4 RECURSIVIDAD EN ORDENACIÓN Y BÚSQUEDA

En los capítulos 8 y 9 el lector podrá aplicar nuevamente el concepto de recursividad. En el capítulo 8 se presenta el método más eficiente de ordenación, *Quicksort*, que funciona de manera recursiva.

En el capítulo 9, correspondiente a búsqueda, el lector podrá estudiar nuevamente gran cantidad de algoritmos recursivos.

EJERCICIOS

1. ***Inversión de capital.*** Se ha depositado en una institución bancaria un monto de capital *m* por el cual se recibe un X% de interés anual. El problema consiste en determinar el capital que se tendrá al cabo de *n* años. Escriba un subprograma recursivo que resuelva este problema. Recuerde que debe establecer los estados básico y recursivo del problema.

2. Retome el problema anterior y resuélvalo de manera iterativa. Compare sus soluciones, teniendo en cuenta la eficiencia en el manejo de memoria y la legibilidad del código generado.

3. Escriba un subprograma recursivo que invierta el orden de los elementos de un arreglo de *N* números enteros. Es decir, que el elemento que está en la posición 1 se intercambie con el que está en la posición *N*, el de la posición 2, con el de la $N - 1$, y así sucesivamente.

4. Retome el problema anterior y resuélvalo de manera iterativa. Compare sus soluciones, teniendo en cuenta la eficiencia en el manejo de memoria y la legibilidad del código generado.

5. Escriba un subprograma recursivo que invierta el orden de una cadena de caracteres. Por ejemplo, si la cadena de entrada es ROMA, el resultado que debe arrojar el programa es AMOR.

6. Se tienen tres arreglos: SUR, CENTRO y NORTE que almacenan los nombres de los países de Sur, Centro y Norteamérica, respectivamente. Los tres arreglos están ordenados en forma alfabética. Escriba un subprograma recursivo que mezcle los tres arreglos anteriores, formando un cuarto arreglo, AMÉRICA, en el cual aparezcan los nombres de todos los países del continente ordenados alfabéticamente. Compare esta solución con la desarrollada para el problema 8 del capítulo 1.

7. Dado como dato el siguiente programa, sígalo y diga qué imprime para los siguientes valores de *X*: $X = 38$, $X = 51$, $X = 24$

```
P1 (X)
{X es un parámetro de tipo entero positivo}
1. Si (X > 100)
       entonces
           P1 ← (X – 8)
       si no
           Hacer P1 ← P1 (P1 (X + 9))
2. {Fin del condicional del paso 1}
```

8. Escriba un subprograma recursivo que le permita calcular el determinante de una matriz cuadrada de dimensión n.

9. Escriba un subprograma recursivo que busque un valor X en un arreglo unidimensional de enteros, ordenado en forma decreciente.

10. Escriba un subprograma recursivo tal que dado como dato un arreglo unidimensional de enteros positivos de dimensión N, determine si las sumas de las dos mitades (del elemento 1 al $N/2$ y del elemento $N/2 + 1$ al N) son iguales.

11. Escriba un subprograma recursivo que quite todos los espacios en blanco de una cadena de caracteres.

12. Dados como datos dos números enteros positivos A y B —el segundo puede ser también nulo—, escriba un subprograma recursivo que calcule A^B.

13. Escriba un subprograma que resuelva la función de Ackermann en forma iterativa. Compare su solución con la analizada en este libro.

14. Escriba un subprograma recursivo que, dado como dato un número entero positivo, regrese 1 si todos los dígitos de dicho número son pares y 0 en otro caso.

15. Retome el problema anterior y resuélvalo de manera iterativa. Compare sus soluciones, teniendo en cuenta la eficiencia en el manejo de memoria y la legibilidad del código generado.

16. Escriba un subprograma recursivo que, dado como dato un número entero positivo, regrese 1 si el número es divisible por 11 y 0 en otro caso. El criterio que deberá usarse para determinar si es divisible es que la diferencia entre la suma de los dígitos que ocupan posiciones pares y la suma de los dígitos que ocupan posiciones impares sea un múltiplo de 11. Por ejemplo, si el número es 6 479, se tiene que 13 (6 + 7) – 13 (4 + 9) = 0 y 0 es múltiplo de 11.

17. Retome el problema anterior y resuélvalo de manera iterativa. Compare sus soluciones, teniendo en cuenta la eficiencia en el manejo de memoria y la legibilidad del código generado.

18. Escriba un subprograma recursivo que, dado como dato un número entero positivo, regrese como resultado la suma de sus divisores.

19. Retome el algoritmo 3.5 —*Conv_postfija*— o el 3.6 —*Conv_prefija*—, del capítulo 3, y desarrolle una versión recursiva del mismo. Compare su solución con la analizada en este libro. Identifique ventajas y desventajas de cada una de ellas.

20. Escriba un subprograma recursivo que invierta el orden de los elementos de una pila. Puede utilizar cualquier estructura de datos como auxiliar, si lo requiere.

21. Escriba un método recursivo, para la clase *Cola*, que imprima todos los elementos de una cola circular.

22. Retome el problema anterior y resuélvalo de manera iterativa. Compare sus soluciones, teniendo en cuenta la eficiencia en el manejo de memoria y la legibilidad del código generado.

23. Escriba un método recursivo, para la clase *Cola*, que invierta el orden de los elementos de una cola. Puede utilizar cualquier estructura de datos como auxiliar, si lo requiere.

Capítulo 5

LISTAS

5.1 INTRODUCCIÓN

Las estructuras de datos presentadas hasta el momento, arreglos y registros, se denominan estáticas. Reciben este nombre debido a que durante la compilación se les asigna un espacio de memoria, y éste permanece inalterable durante la ejecución del programa.

En este capítulo se presenta la estructura de datos **lista**. Este es un tipo de estructura **lineal** y **dinámica** de datos. Lineal porque a cada elemento le puede seguir sólo otro elemento; dinámica porque se puede manejar la memoria de manera flexible, sin necesidad de reservar espacio con antelación.

La principal ventaja de manejar un tipo dinámico de datos es que se pueden adquirir posiciones de memoria a medida que se necesitan; éstas se liberan cuando ya no se requieren. Así es posible crear estructuras dinámicas que se expandan o contraigan, según se les agregue o elimine elementos. El dinamismo de estas estructuras soluciona el problema de decidir cuál es la cantidad óptima de memoria que se debe reservar para un problema específico. Sin embargo, es importante destacar que las estructuras dinámicas no pueden reemplazar a los arreglos en todas sus aplicaciones. Existen numerosos casos que podrían fácilmente ser solucionados aplicando arreglos, mientras que si se utilizaran estructuras dinámicas, como las listas, la solución de estos problemas se complicaría.

Las listas ligadas son colecciones de elementos llamados nodos; el orden entre éstos se establece por medio de un tipo de datos denominado **punteros**, **apuntadores**, **direcciones** o referencias a otros nodos. Por tanto, siempre es importante distinguir entre un dato de tipo apuntador y el dato contenido en la celda al cual éste apunta. Se usará la notación ***P*** ← ***^D*** para indicar que *P* es un apuntador al nodo *D*, ***Crear(P)*** para señalar el proceso de asignación de memoria al nodo *P*, y ***Quitar(P)*** para indicar el proceso inverso; es decir, cuando se libera una posición de memoria apuntada por *P*.

Las operaciones más importantes que se realizan en las estructuras de datos son las de búsqueda, inserción y eliminación. Se utilizan también para comparar la eficiencia de las estructuras de datos y de esta forma observar cuál es la estructura que mejor se adapta al tipo de problema que se quiera resolver. La búsqueda, por ejemplo, es una operación que no se puede realizar en forma eficiente en las listas. Por otra parte, las operaciones de inserción y eliminación se efectúan de manera eficiente en este tipo de estructuras de datos.

Este capítulo se dedicará a las **estructuras dinámicas lineales llamadas listas**; entre ellas se distinguen tres tipos: **listas simplemente ligadas**, **listas doblemente ligadas**

y **listas circulares**. En el siguiente capítulo se presentarán las **estructuras dinámicas no lineales**, denominadas **árboles**.

5.2 LISTAS SIMPLEMENTE LIGADAS

Una **lista simplemente ligada** constituye una colección de elementos llamados nodos. El orden entre éstos se establece por medio de punteros; es decir, direcciones o referencias a otros nodos. Un tipo especial de lista simplemente ligada es la lista vacía. La figura 5.1 presenta la estructura de un nodo de una lista simplemente ligada.

En general, un nodo consta de dos partes:

- Un campo **INFORMACIÓN** que será del tipo de los datos que se quiera almacenar en la lista.
- Un campo **LIGA**, de tipo puntero, que se utiliza para establecer la liga o el enlace con otro nodo de la lista. Si el nodo fuera el último de la lista, este campo tendrá como valor NIL —vacío—. Al emplearse el campo liga para relacionar dos nodos, no será necesario almacenar físicamente a los nodos en espacios contiguos.

En la figura 5.2 se presenta un ejemplo de una lista simplemente ligada que almacena apellidos. El primer nodo de la lista es apuntado por una variable *P*, de tipo apuntador —*P* almacena la dirección del primer nodo—. El campo liga del último nodo de la lista tiene un valor NIL, que indica que dicho nodo no apunta a ningún otro. El apuntador al inicio de la lista es importante porque permite posicionarnos en el primer nodo de la misma y tener acceso al resto de los elementos. Si, por alguna razón, este apuntador se extraviara, entonces perderíamos toda la información almacenada en la lista. Por otra parte, si la lista simplemente ligada estuviera vacía, entonces el apuntador al inicio tendrá el valor NIL.

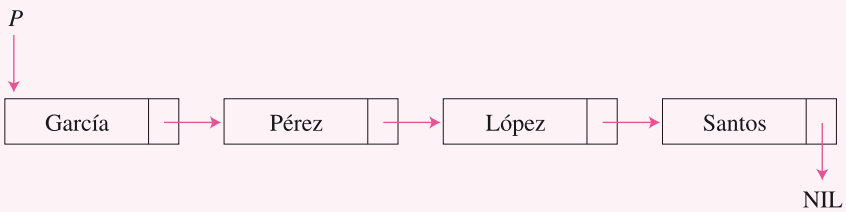
5.2.1 Operaciones con listas simplemente ligadas

Las operaciones que pueden efectuarse en una lista simplemente ligada son:

- Recorrido de la lista.
- Inserción de un elemento.
- Borrado de un elemento.
- Búsqueda de un elemento.

FIGURA 5.1
Estructura de un nodo.

FIGURA 5.2
Ejemplo de lista.



Antes de analizar cada una de estas operaciones, se presentará un algoritmo que permite crear una lista simplemente ligada, al incorporar cada nuevo nodo al inicio.

Algoritmo 5.1 Crea_inicio

Crea_inicio

{Este algoritmo permite crear una lista simplemente ligada, agregando cada nuevo nodo al inicio de la misma}
{*P* y *Q* son variables de tipo puntero. Los campos del nodo son INFO, que será del tipo de datos que se quiera almacenar en la lista, y LIGA de tipo apuntador. *P* apunta al inicio de la lista. RES es una variable de tipo entero}

1. Crear (*P*) {Se crea el primer nodo de la lista simplemente ligada}
2. Leer *P*^.INFO
3. Hacer *P*^.LIGA ← NIL
4. Escribir "¿Desea ingresar más nodos a la lista? Sí: 1, No: 0"
5. Leer RES
6. Mientras (RES = 1) *Repetir*
   Crear (*Q*)
   Leer *Q*^.INFO
   Hacer *Q*^.LIGA ← *P* y *P* ← *Q*
   Escribir "¿Desea ingresar más nodos a la lista? Sí: 1, No: 0"
   Leer RES
7. {Fin del ciclo del paso 6}

Veamos un ejemplo para ilustrar el funcionamiento de este algoritmo.

Ejemplo 5.1

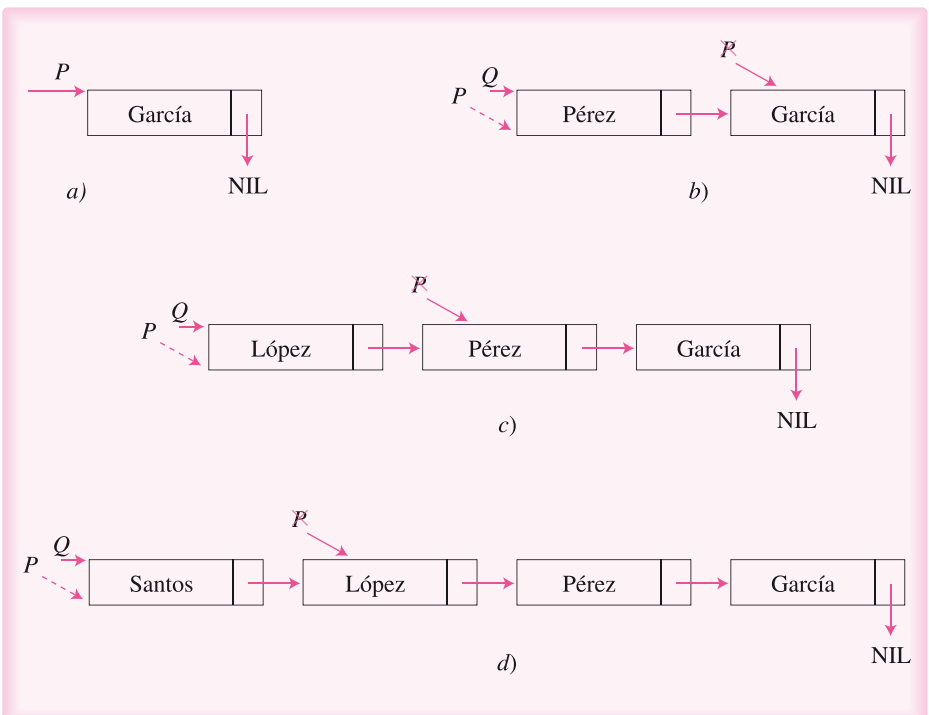
Dados los siguientes datos: García, Pérez, López y Santos, genere una lista simplemente ligada mediante el algoritmo 5.1. En la siguiente figura se puede observar, paso a paso, cómo se va construyendo la lista.

Como se aprecia en la figura 5.3*d*, la lista quedó en orden inverso con respecto al orden en el que fueron dados los datos. Para lograr que los datos queden en el orden en el que fueron dados, se debe agregar cada nodo al final de la lista. A continuación se presenta un algoritmo que permite crear una lista simplemente ligada, al incorporar cada nuevo nodo al final.

FIGURA 5.3

Creación de listas. *a*) Luego de crear el primer nodo. *b*) Luego de insertar a "Pérez". *c*) Luego de insertar a "López". *d*) Luego de insertar a "Santos".

Nota: Las flechas discontinuas indican los cambios originados al insertar un nuevo elemento al inicio de la lista.



Algoritmo 5.2 Crea_final

Crea_final

{Este algoritmo permite crear una lista simplemente ligada, agregando cada nuevo nodo al final de la misma}
{*P*, *Q* y *T* son variables de tipo apuntador. Los campos del nodo son INFO, que será del tipo de datos que se quiera almacenar en la lista, y LIGA de tipo apuntador. *P* apunta al inicio de la lista. RES es una variable de tipo entero}

1. Crear (*P*) {Se crea el primer nodo de la lista}
2. Leer *P*^.INFO
3. Hacer *P*^.LIGA ← NIL y *T* ← *P*
4. Escribir "¿Desea ingresar más nodos a la lista? Sí : 1, No: 0"
5. Leer RES
6. Mientras (RES = 1) *Repetir*
   Crear (*Q*)
   Leer *Q*^.INFO
   Hacer *Q*^.LIGA ← NIL, *T*^.LIGA ← *Q* y *T* ← *Q* {*T* apunta al último nodo}
   Escribir "¿Desea ingresar más nodos a la lista? Sí: 1, No: 0"
   Leer RES
7. {Fin del ciclo del paso 6}

Veamos un ejemplo para ilustrar el funcionamiento de este algoritmo.

Ejemplo 5.2

Se utilizan los datos del ejemplo anterior para crear una lista aplicando el algoritmo 5.2. Es importante observar que en este algoritmo se utiliza otra variable de tipo apuntador para mantener la dirección del último nodo de la lista, de tal manera que se pueda establecer el enlace entre éste y el nuevo nodo. En la figura 5.4 se puede observar, paso a paso, cómo se va construyendo esa lista.

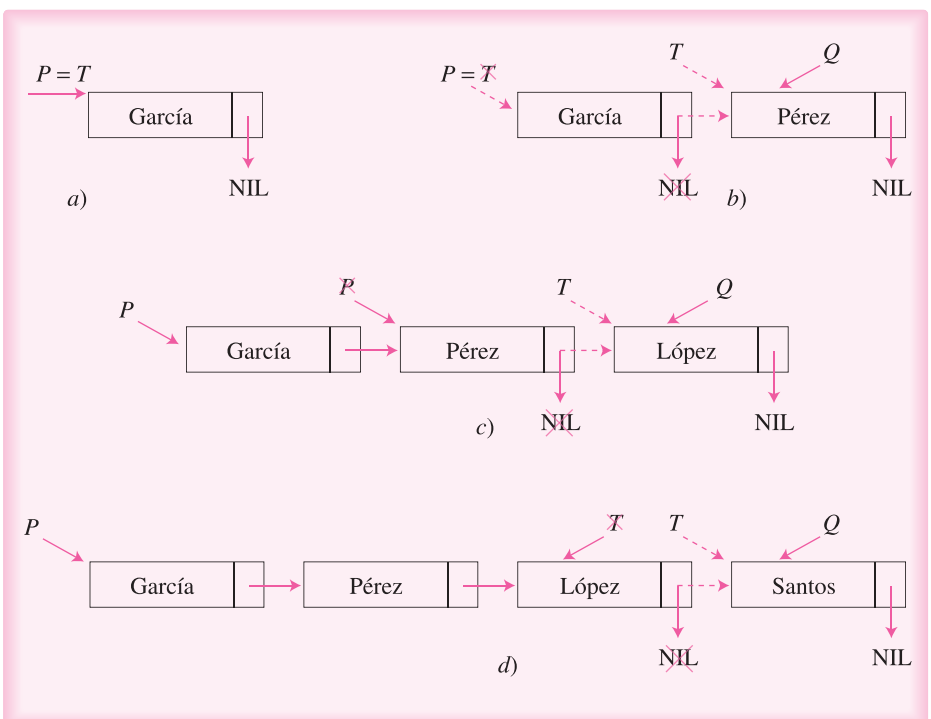


FIGURA 5.4

Creación de listas. *a*) Luego de crear el primer nodo. *b*) Luego de insertar a "Pérez". *c*) Luego de insertar a "López". *d*) Luego de insertar a "Santos".

Nota: Las flechas discontinuas indican los cambios originados al insertar un nuevo elemento al final de la lista.

5.2.2 Recorrido de una lista simplemente ligada

La operación de recorrido en una lista simplemente ligada consiste en visitar cada uno de los nodos que forman la lista. La visita puede implicar una operación simple; por ejemplo, imprimir la información del nodo, o una compleja, dependiendo del problema que se intente resolver.

Para recorrer todos los nodos de una lista simplemente ligada se comienza con el primero. Se toma el valor del campo LIGA de éste y se avanza al segundo, y así sucesivamente hasta llegar al último nodo, cuyo campo LIGA tiene el valor NIL. En general, la dirección de un nodo, excepto el primero, está dada por el campo LIGA de su predecesor.

El algoritmo 5.3 presenta los pasos necesarios para recorrer una lista en forma iterativa.

Algoritmo 5.3 Recorre_iterativo

Recorre_iterativo (*P*)

{Este algoritmo recorre una lista cuyo primer nodo está apuntado por *P*}
{*Q* es una variable de tipo apuntador. INFO y LIGA son los campos de cada nodo de la lista}

1. Hacer $Q \leftarrow P$
2. Mientras ($Q \neq$ NIL) *Repetir*
   Escribir *Q*^.INFO
   Hacer $Q \leftarrow$ *Q*^.LIGA {Apunta al siguiente nodo de la lista}
3. {Fin del ciclo del paso 2}

Las listas se pueden manejar fácilmente con procesos recursivos. El algoritmo 5.4 constituye una versión recursiva para recorrer una lista simplemente ligada.

Algoritmo 5.4 Recorre_recursivo

Recorre_recursivo (*P*)

{Este algoritmo recorre una lista simplemente ligada en forma recursiva. *P* es un apuntador al nodo que se va a visitar. La primera vez trae la dirección del primer nodo de la lista}
{INFO y LIGA son los campos de cada nodo de la lista}

1. *Si P* $\neq$ NIL *entonces*
   Escribir *P*^.INFO
   Llamar a Recorre_recursivo con *P*^.LIGA
   {Llamada recursiva con el apuntador al siguiente nodo de la lista}
2. {Fin del condicional del paso 1}

Veamos ahora la operación de inserción en listas simplemente ligadas.

5.2.3 Inserción en listas simplemente ligadas

La operación de inserción en listas simplemente ligadas consiste en agregar un nuevo nodo a la lista. Sin embargo, dependiendo de la posición en la que se deba insertar el nodo, se pueden presentar diferentes casos, como los que se señalan a continuación:

- Insertar un nodo al inicio de la lista.
- Insertar un nodo al final de la lista.
- Insertar un nodo antes que otro cuya información es *X*.
- Insertar un nodo después que otro cuya información es *X*.

No se considerará en estos algoritmos el caso de que la lista esté vacía; esta condición se puede incluir ya sea al inicio del algoritmo o en el programa principal. Se considerará entonces que la lista en la cual se va a insertar el nuevo nodo ya existe —por lo menos tiene un nodo—.

a) Inserción al inicio de una lista simplemente ligada

En este caso el nuevo nodo se coloca al principio de la lista simplemente ligada, convirtiéndose en el primero de ella. El proceso es relativamente simple, como se puede observar en el siguiente algoritmo.

Algoritmo 5.5 Inserta_inicio

Inserta_inicio (*P*, DATO)

{Este algoritmo inserta un nodo al inicio de una lista simplemente ligada. *P* es el apuntador al primer nodo de la misma, y DATO es la información que se almacenará en el nuevo nodo}
{*Q* es una variable de tipo apuntador. INFO y LIGA son los campos de cada nodo de la lista}

1. Crear (*Q*)
2. Hacer *Q*^.INFO ← DATO, *Q*^.LIGA ← *P* y *P* ← *Q*

Ejemplo 5.3

En la figura 5.5 se presenta un ejemplo de inserción al inicio de la lista.

b) Inserción al final de una lista simplemente ligada

En este caso el nuevo nodo se coloca al final de la lista simplemente ligada, convirtiéndose en el último. El algoritmo 5.6 describe este proceso.

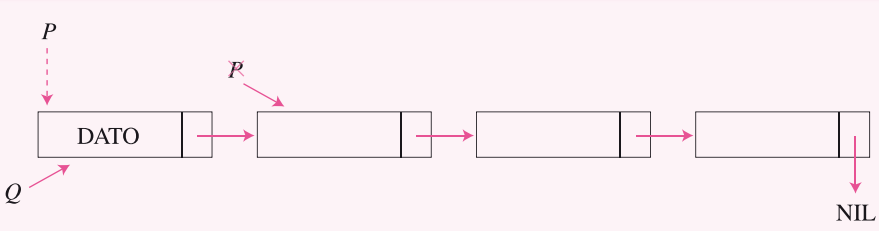


FIGURA 5.5
Inserción al inicio de la lista.

Nota: La flecha discontinua indica los cambios originados por la inserción de un nuevo nodo al inicio de la lista.

Algoritmo 5.6 Inserta_final

Inserta_final (*P*, DATO)

{Este algoritmo inserta un nodo al final de una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista, y DATO es la información que se almacenará en el nuevo nodo}
{*Q* y *T* son variables de tipo puntero. INFO y LIGA son los campos de cada nodo de la lista}

1. Hacer $T \leftarrow P$
2. Mientras (T^.LIGA $\neq$ NIL) *Repetir*
   {Recorre la lista hasta llegar al último elemento}
   Hacer $T \leftarrow T$^.LIGA
3. {Fin del ciclo del paso 2}
4. Crear (Q)
5. Hacer Q^.INFO $\leftarrow$ DATO, Q^.LIGA $\leftarrow$ NIL y T^.LIGA $\leftarrow Q$

Ejemplo 5.4

En la figura 5.6 se presenta un ejemplo de inserción al final de la lista.

Si en cada lista simplemente ligada se utilizaran dos apuntadores, uno al primer nodo y otro al último (fig. 5.7*a*), entonces el proceso de inserción al final se simplificaría, ya que no sería necesario recorrerla toda para llegar al final. El nuevo nodo se podría incorporar directamente (ver fig. 5.7*b*), como en el caso de inserción al inicio de la lista.

c) Inserción de un nodo antes que otro en una lista simplemente ligada

En este tipo de inserción en listas simplemente ligadas, el nuevo nodo se debe colocar antes de otro nodo dado como referencia. Se pueden presentar diferentes casos; por ejemplo, que el nodo dado como referencia no se encuentre en la lista o que el nuevo nodo a insertar se convierta en el primero. Se asume, como se ha señalado anteriormente, que la lista no está vacía.

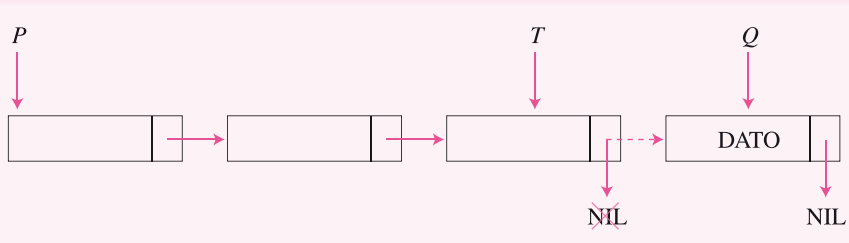


FIGURA 5.6
Inserción al final de la lista.

Nota: La flecha discontinua indican los cambios originados por la inserción de un nuevo nodo al final de la lista.

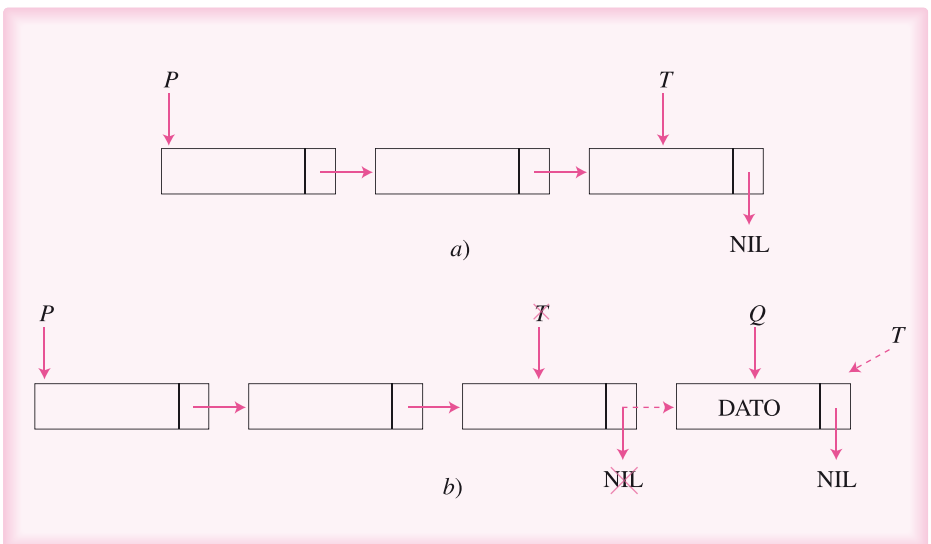


FIGURA 5.7
Inserción en una lista con punteros al inicio y al final de la misma. *a*) Lista con puntero al inicio, *P*, y al final, *T*. *b*) Lista luego de la inserción de un nuevo elemento al final de la misma.

Nota: Las flechas discontinuas indican los cambios originados por la inserción de un nuevo nodo al final de la lista.

Algoritmo 5.7 Inserta_antes_X

Inserta_antes_*X* (*P*, DATO, *X*)

{Este algoritmo inserta un nodo antes de un nodo dado como referencia en una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista, DATO indica la información que se almacenará en el nuevo nodo, y *X* representa el contenido —información— del nodo dado como referencia}
{*Q*, *X* y *T* son variables de tipo apuntador. INFO y LIGA son los campos de los nodos de la lista. BAND es una variable de tipo entero}

1. Hacer $Q \leftarrow P$ y BAND $\leftarrow 1$
2. Mientras ((Q^.INFO $\neq X$) y (BAND = 1)) Repetir
  2.1 *Si* (Q^.LIGA $\neq$ NIL)
    entonces
      Hacer $T \leftarrow Q$ y $Q \leftarrow Q$^.LIGA
    si no
      Hacer BAND $\leftarrow 0$
  2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. *Si* (BAND = 1)
    entonces
      Crear (*X*)
      Hacer *X*^.INFO $\leftarrow$ DATO
    4.1 *Si* ($P = Q$) {El nodo dado como referencia es el primero}
      entonces
        Hacer *X*^.LIGA $\leftarrow P$ y $P \leftarrow X$
      si no

```
            Hacer T^.LIGA ← X y X^.LIGA ← Q
      4.2   {Fin del condicional del paso 4.1}
       si no
            Escribir "El nodo dado como referencia no se encuentra en la lista"
5.    {Fin del condicional del paso 4}
```

Ejemplo 5.5

En la figura 5.8 se presenta un ejemplo de inserción de un nodo antes que otro dado como referencia en una lista simplemente ligada, aplicando el algoritmo anterior.

d) Inserción de un nodo después de otro en una lista simplemente ligada

En este tipo de inserción en listas simplemente ligadas, el nuevo nodo se debe colocar después de otro dado como referencia. Se pueden presentar diferentes casos; por ejemplo, que el nodo dado como referencia no se encuentre en la lista o que el nuevo se convierta en el último de la lista. Se asume, como se ha señalado, que la lista no está vacía. A continuación se presenta el algoritmo correspondiente.

Algoritmo 5.8 Inserta_después_X

Inserta_después_X (*P*, DATO, *X*)

{Este algoritmo inserta un nodo después de otro dado como referencia en una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista, DATO indica la información que se

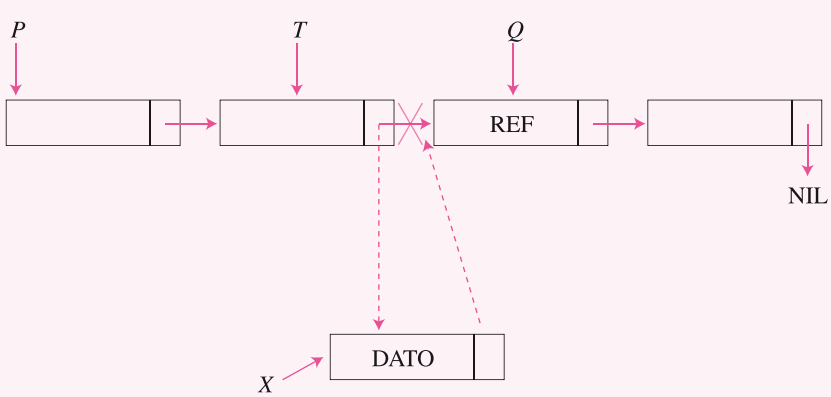


FIGURA 5.8
Inserción de nodos.

Nota: Las flechas discontinuas indican los cambios originados por la inserción de un nuevo nodo precediendo a otro, dado como referencia.

almacenará en el nuevo nodo, y *X* representa el contenido —información— del nodo dado como referencia}
{*Q* y *T* son variables de tipo apuntador. INFO y LIGA son los campos de los nodos de la lista. BAND es una variable de tipo entero}

1. Hacer $Q \leftarrow P$ y BAND $\leftarrow 1$
2. Mientras ((*Q*^.INFO $\neq X$) y (BAND = 1)) *Repetir*
  2.1 Si *Q*^.LIGA $\neq$ NIL
    entonces
      Hacer $Q \leftarrow$ *Q*^.LIGA
    si no
      Hacer BAND $\leftarrow 0$
  2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. *Si* (BAND = 1)
    entonces
      Crear (*T*)
      Hacer *T*^.INFO $\leftarrow$ DATO, *T*^.LIGA $\leftarrow$ *Q*^.LIGA y *Q*^.LIGA $\leftarrow$ *T*
    si no
      Escribir "El nodo dado como referencia no se encuentra en la lista"
5. {Fin del condicional del paso 4}

Ejemplo 5.6

En la figura 5.9 se presenta un ejemplo de inserción de un nodo después de otro dado como referencia, en listas simplemente ligadas aplicando el algoritmo anterior.

Cabe señalar que en las operaciones de listas tratadas hasta el momento, no se ha considerado el orden entre los elementos. Si se supone que la lista está ordenada, en el momento de insertar un nuevo nodo habrá que mantener el orden previamente establecido.

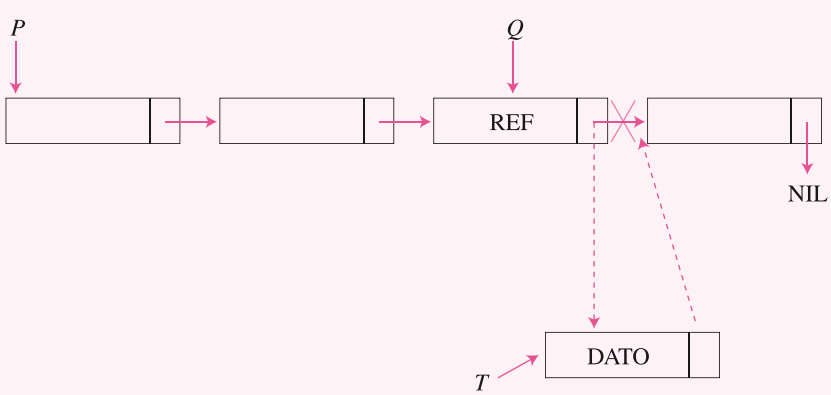


FIGURA 5.9
Inserción de nodos.

Nota: Las flechas discontinuas indican los cambios originados por la inserción de un nuevo nodo sucediendo a otro dado como referencia.

5.2.4 Eliminación en listas simplemente ligadas

La operación de eliminación en listas simplemente ligadas consiste en eliminar un nodo de la lista y liberar el espacio de memoria correspondiente. Dependiendo de la posición en la que éste se encuentre, se pueden presentar diferentes casos, como los que se señalan a continuación:

- Eliminar el primer nodo.
- Eliminar el último nodo.
- Eliminar un nodo con información *X*.
- Eliminar el nodo anterior al nodo con información *X*.
- Eliminar el nodo posterior al nodo con información *X*.

Cabe destacar que en los algoritmos que se presentan a continuación no se considera que la lista esté vacía. Esta condición se puede evaluar fácilmente al inicio del algoritmo o bien en el programa principal.

a) Eliminar el primer nodo de una lista simplemente ligada

Esta operación es muy sencilla, ya que sólo es necesario redefinir el apuntador al inicio de la lista. Si ésta quedara vacía (es decir, si la lista tenía sólo un elemento), entonces apuntaría a NIL. En el siguiente algoritmo se describe este caso.

Algoritmo 5.9 Elimina_inicio

Elimina_inicio (*P*)

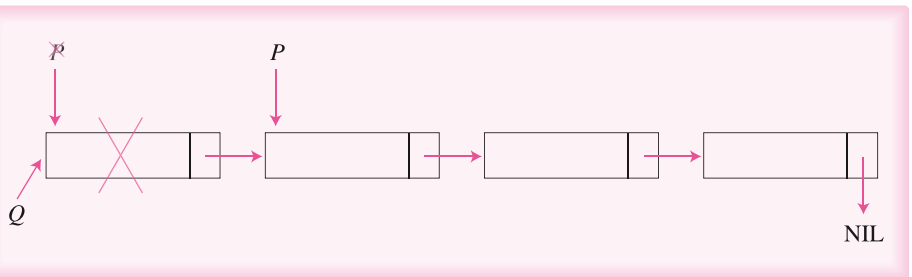
{Este algoritmo permite eliminar el primer elemento de una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista}
{*Q* es una variable de tipo apuntador. INFO y LIGA son los campos de los nodos de la lista}

1. Hacer $Q \leftarrow P$
   {Si la lista tuviera sólo un elemento entonces a *P* se le asignaría NIL, que es el valor de *Q*^.LIGA. En caso contrario, queda con la dirección del siguiente nodo}
2. Hacer $P \leftarrow Q$^.LIGA {Redefine el puntero al inicio de la lista}
3. Quitar (*Q*)

Ejemplo 5.7

En la figura 5.10 se presenta un ejemplo de eliminación del primer nodo de una lista simplemente ligada, aplicando el algoritmo anterior.

FIGURA 5.10
Eliminación del primer nodo de una lista.



b) Eliminar el último nodo de una lista simplemente ligada

En este caso se debe eliminar el último nodo de una lista simplemente ligada. Es importante observar que para alcanzar el último nodo, se debe recorrer toda la lista, excepto si se usara un apuntador que indique su final. A continuación se presenta un algoritmo de solución, considerando que solamente se tiene un apuntador al inicio de la lista.

Algoritmo 5.10 Elimina_último

Elimina_último (*P*)

{Este algoritmo permite eliminar el último nodo de una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista}
{*Q* y *T* son variables de tipo apuntador. INFO y LIGA son los campos de los nodos de la lista}

1. Hacer $Q \leftarrow P$
2. Si (P^.LIGA = NIL) {Se verifica si la lista tiene sólo un nodo}
entonces
Hacer $P \leftarrow$ NIL
si no
2.1 Mientras (Q^.LIGA $\neq$ NIL) *Repetir*
Hacer $T \leftarrow Q$ y $Q \leftarrow Q$^.LIGA
2.2 {Fin del ciclo del paso 2.1}
Hacer T^.LIGA $\leftarrow$ NIL
3. {Fin del condicional del paso 2}
4. Quitar (Q)

Ejemplo 5.8

En la figura 5.11 se presenta un ejemplo de eliminación del último nodo de una lista simplemente ligada.

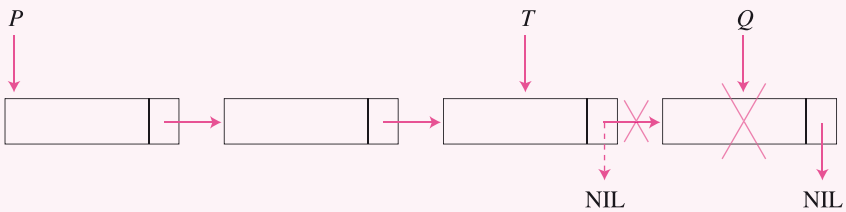


FIGURA 5.11
Eliminación del último nodo de una lista.

Nota: La flecha discontinua indica los cambios originados por la eliminación del último nodo de la lista.

c) Eliminar un nodo con información *X* de una lista simplemente ligada

La eliminación de un nodo con información *X* es uno de los casos complicados de esta operación, porque se pueden presentar diferentes variantes. Por ejemplo, el nodo puede ser el primero, el último, el único o no encontrarse en la lista. El algoritmo 5.11 describe este proceso.

Algoritmo 5.11 Elimina_X

Elimina_X (*P*, *X*)

{Este algoritmo elimina un nodo con información *X* de una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista}
{*Q* y *T* son variables de tipo apuntador. BAND es una variable de tipo entero. INFO y LIGA son los campos de los nodos de la lista}

```
1. Hacer Q ← P y BAND ← 1
2. Mientras ((Q^.INFO ≠ X) y (BAND = 1)) Repetir
   2.1 Si Q^.LIGA ≠ NIL
            entonces
              Hacer T ← Q y Q ← Q^.LIGA
            si no
              Hacer BAND ← 0
   2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (BAND = 0)
      entonces
         Escribir "El elemento con información X no se encuentra en la lista"
      si no
         4.1 Si (P = Q) {Se verifica si el elemento a eliminar es el primero}
               entonces
                  Hacer P ← Q^.LIGA
               si no
                  Hacer T^.LIGA ← Q^.LIGA
         4.2 {Fin del condicional del paso 4.1}
             Quitar (Q)
5. {Fin del condicional del paso 4}
```

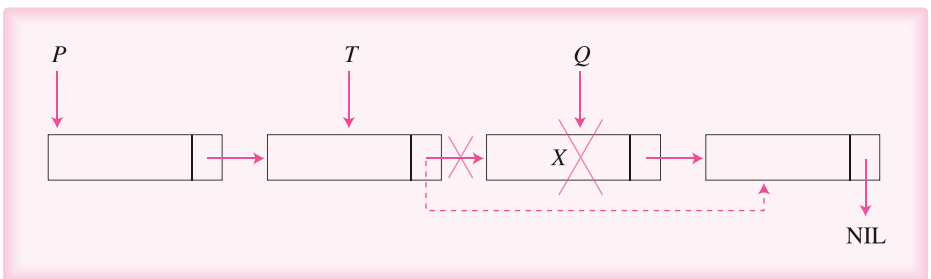


FIGURA 5.12
Eliminación de un nodo con información *X*.

Nota: La flecha discontinua indica los cambios originados por la eliminación del nodo.

Ejemplo 5.9

En la figura 5.12 se presenta un ejemplo de eliminación de un nodo con información *X* en una lista simplemente ligada.

d) Eliminar el nodo anterior al nodo con información *X* en una lista simplemente ligada

Este es el caso de eliminación más complicado en listas simplemente ligadas, porque tiene muchas variantes. Por ejemplo, el nodo con información *X* puede ser el primero —entonces no hay nada que eliminar—, el segundo —entonces hay que eliminar el primero de la lista—, estar en cualquier otra posición, o bien no encontrarse en la lista.

Algoritmo 5.12 Elimina_antes_*X*

Elimina_antes_X (*P*, *X*)

{Este algoritmo permite eliminar el nodo anterior al nodo que contiene la información *X* en una lista simplemente ligada. *P* es el apuntador al primer nodo de la lista}
{*Q*, *T* y *R* son variables de tipo apuntador. BAND es una variable de tipo entero. INFO y LIGA son los campos de los nodos de la lista}

1. *Si* (*P*^.INFO = *X*)
  entonces
    Escribir "No existe un nodo que preceda al que contiene a *X*"
  si no
    Hacer *Q* ← *P*, *T* ← *P* y BAND ← 1
  1.1 Mientras ((*Q*^.INFO ≠ *X*) y (BAND = 1)) *Repetir*
    1.1.1 *Si* (*Q*^.LIGA ≠ NIL)
      entonces
        Hacer *R* ← *T*, *T* ← *Q* y *Q* ← *Q*^.LIGA
      si no
        Hacer BAND ← 0
    1.1.2 {Fin del condicional del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
  1.3 *Si* (BAND = 0)

```
        entonces
            Escribir "El elemento no se encuentra en la lista"
        si no
        1.3.1 Si (P^.LIGA = Q) {El elemento a eliminar es el primero}
                entonces
                    Hacer P ← Q
                si no
                Hacer R^.LIGA ← Q
        1.3.2 {Fin del condicional del paso 1.3.1}
            Quitar (T)
    1.4 {Fin del condicional del paso 1.3}
2. {Fin del condicional del paso 1}
```

Ejemplo 5.10

En la figura 5.13 se presenta un ejemplo de eliminación del nodo anterior a otro dado como referencia, mediante el algoritmo 5.12.

Se deja al lector la construcción del algoritmo para eliminar un nodo después de otro dado como referencia. Este algoritmo es de menor complejidad que el presentado antes.

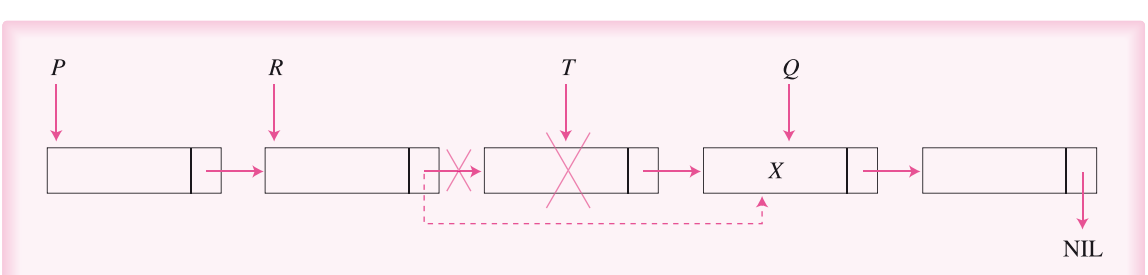
5.2.5 Búsqueda en listas simplemente ligadas

La operación de búsqueda de un elemento en una lista simplemente ligada es muy fácil de realizar, aunque ineficiente ya que se lleva a cabo de forma secuencial. Se debe ir recorriendo los nodos hasta encontrar el que estamos buscando o hasta que se llega al final de la lista. El algoritmo es similar a los que se desarrollaron para recorrer una lista en forma iterativa o recursiva.

Al igual que en el caso de las operaciones vistas anteriormente, existe diferencia en los algoritmos si las listas se encuentran ordenadas o desordenadas. Se comenzará, en primer término, con el algoritmo de búsqueda para listas simplemente ligadas que se encuentran desordenadas.

FIGURA 5.13
Eliminación de nodos.

Nota: La flecha discontinua indica los cambios originados por la eliminación del nodo anterior a un nodo dado como referencia.



Algoritmo 5.13 Búsqueda_desordenada

Búsqueda_desordenada (*P*, *X*)

{Este algoritmo permite buscar el elemento con información *X* en una lista simplemente ligada que se encuentra desordenada. *P* es el apuntador al primer nodo de la lista}
{*Q* es una variable de tipo apuntador. INFO y LIGA son campos de los nodos de la lista}

1. Hacer $Q \leftarrow P$
2. Mientras (($Q \neq$ NIL) y (Q^.INFO $\neq X$))
    Hacer $Q \leftarrow Q$^.LIGA
3. {Fin del ciclo del paso 2}
4. *Si* (Q = NIL)
    entonces
        Escribir "El elemento no se encuentra en la lista"
    si no
        Escribir "El elemento sí se encuentra en la lista"
5. {Fin del condicional del paso 4}

Es importante destacar que con una simple modificación en la condición del ciclo del paso 2 se adapte este algoritmo para la búsqueda de elementos en listas simplemente ligadas que se encuentran ordenadas. A continuación se presenta el algoritmo de búsqueda en listas simplemente ligadas que se encuentran ordenadas en forma ascendente.

Algoritmo 5.14 Búsqueda_ordenada

Búsqueda_ordenada (*P*, *X*)

{Este algoritmo permite buscar el elemento con información *X* en una lista simplemente ligada que se encuentra ordenada en forma ascendente. *P* es el apuntador al primer nodo de la lista}
{*Q* es una variable de tipo apuntador. INFO y LIGA son los campos de los nodos de la lista}

1. Hacer $Q \leftarrow P$
2. Mientras (($Q \neq$ NIL) y (Q^.INFO $< X$))
    Hacer $Q \leftarrow Q$^.LIGA
3. {Fin del ciclo del paso 2}
4. *Si* ((Q = NIL) o (Q^.INFO $> X$))
    entonces
        Escribir "El elemento no se encuentra en la lista"
    si no
        Escribir "El elemento sí se encuentra en la lista"
5. {Fin del condicional del paso 4}

Todos los algoritmos presentados tanto para la búsqueda, inserción y eliminación se pueden implementar de forma recursiva. A continuación se muestra una versión recursiva del algoritmo 5.13.

Algoritmo 5.15 Búsqueda_recursivo

Búsqueda_recursivo (*P*, *X*)

{Este algoritmo permite buscar recursivamente a un elemento con información *X* en una lista simplemente ligada que se encuentra desordenada. *P* es el apuntador al primer elemento de la lista}

```
1. Si (P ≠ NIL)
        entonces
        1.1 Si (P^.INFO = X)
                entonces
                    Escribir "El elemento se encuentra en la lista"
                si no
                    Llamar a Búsqueda_recursivo con P^.LIGA y X
        1.2 {Fin del condicional del paso 1.1}
        si no
                    Escribir "El elemento no se encuentra en la lista"
2. {Fin del condicional del paso 1}
```

5.3 LISTAS CIRCULARES

Las **listas circulares** son similares a las listas simplemente ligadas. Sin embargo, tienen la característica de que el último elemento de la lista apunta al primero, en lugar de apuntar al vacío o NIL.

Se define una **lista simplemente ligada circular** como una colección de elementos llamados nodos, en la cual el último nodo apunta al primero. En la figura 5.14 se presenta una gráfica de una lista circular.

Las operaciones en listas circulares son similares a las operaciones en listas lineales; por tanto, no se tratarán nuevamente en esta sección. Sin embargo, es importante

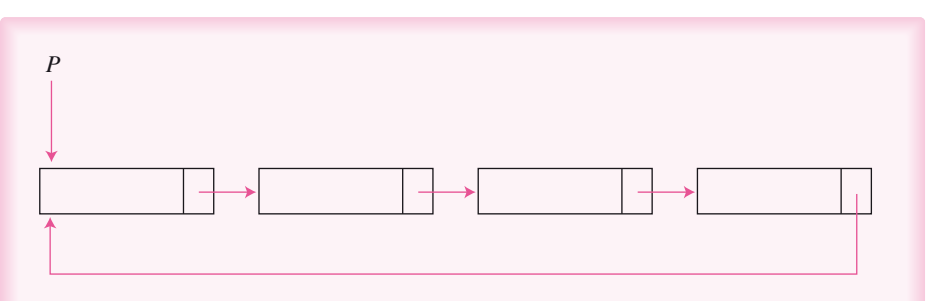


FIGURA 5.14
Lista circular.

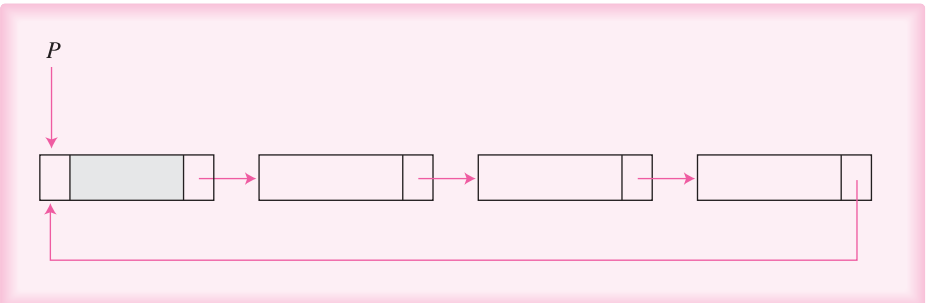


FIGURA 5.15
Lista circular con nodo de cabecera.

señalar que para el caso de la operación de recorrido de listas circulares se necesita considerar algún criterio para detectar cuándo se han visitado todos los nodos de la lista. Esto último con el propósito de evitar caer en ciclos infinitos. Una posible solución al problema planteado consiste en usar un nodo extra, llamado **nodo de cabecera**, para indicar el inicio de la lista. Este nodo contendrá información especial, de tal manera que se distinga de los demás y así podrá hacer referencia al principio de la lista. La figura 5.15 presenta una gráfica de una lista circular con nodo de cabecera.

En los algoritmos presentados para operar con listas simplemente ligadas se puede apreciar que sólo se tiene acceso a un nodo y al sucesor de éste. Si se necesitara su predecesor, por ejemplo, se tendrían que usar variables auxiliares (véase el algoritmo 5.12). Una manera de evitar esta situación, es teniendo acceso a los nodos en cualquier orden —antecesor o sucesor—, y además recorrer la lista del inicio al fin, o viceversa. Las listas que cuentan con esta facilidad son las doblemente ligadas. A continuación se presenta este tipo de estructuras.

5.4 LISTAS DOBLEMENTE LIGADAS

Una **lista doblemente ligada** es una colección de nodos, en la cual cada uno de ellos tiene dos apuntadores (fig. 5.16*a*), uno apuntando a su predecesor (LIGAIZQ) y otro a su sucesor (LIGADER). Por medio de estos punteros se podrá entonces avanzar o retroceder a través de la lista, según se tomen las direcciones de uno u otro apuntador. La figura 5.16*b* representa una lista doblemente ligada que almacena apellidos.

Para tener un fácil acceso a la información de la lista es recomendable usar dos apuntadores, *P* y *F*, que apunten al principio y al final de ésta, respectivamente.

5.4.1 Operaciones con listas doblemente ligadas

Las operaciones que se pueden llevar a cabo con este tipo de estructuras son las mismas que con listas simplemente ligadas. En esta sección se presentarán las operaciones de:

- Recorrido de la lista.
- Inserción de un elemento.
- Borrado de un elemento.

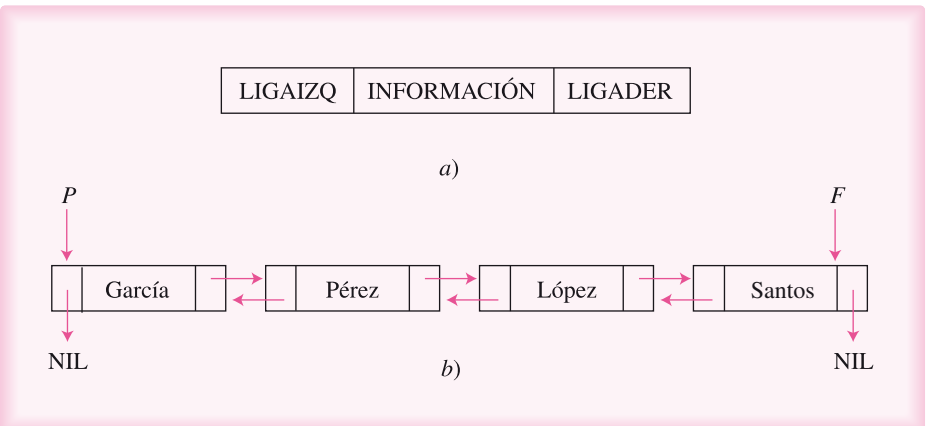


FIGURA 5.16
Ejemplo de lista doblemente ligada. *a*) Estructura de un nodo. *b*) Ejemplo de lista doblemente ligada.

5.4.2 Recorrido de una lista doblemente ligada

Al tener cada nodo una doble liga, la lista se puede recorrer tanto del inicio al final, mediante las ligas derechas, como en sentido inverso; es decir, del final al principio, con las ligas izquierdas. Cualquiera que sea la dirección del recorrido, el algoritmo es similar al que se presenta para listas simplemente ligadas. Se deja al lector su diseño.

5.4.3 Inserción en listas doblemente ligadas

La inserción de un elemento consiste en agregar un nuevo nodo a la lista y establecer los apuntadores correspondientes. No se considerará el caso de lista vacía. La inserción se puede llevar a cabo:

- Al inicio de la lista doblemente ligada.
- Al final de la lista doblemente ligada.
- Antes/después de un nodo con información *X*.

a) Inserción al inicio de la lista doblemente ligada

En este caso el nuevo nodo se coloca al principio de la lista y se establecen las ligas correspondientes. El nuevo nodo insertado se convierte, entonces, en el primero de la lista doblemente ligada. El algoritmo 5.16 describe este proceso.

Algoritmo 5.16 Inserta_principio

Inserta_principio (*P*, DATO)

{Este algoritmo inserta un nodo al inicio de una lista doblemente ligada. *P* es el apuntador al primer nodo de la lista y DATO es la información que se almacenará en el nuevo nodo}

{*Q* es una variable de tipo apuntador. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

1. Crear (*Q*)
   Hacer *Q*^.INFOR ← DATO, *Q*^.LIGADER ← *P*, *P*^.LIGAIZQ ← *Q*,
   Q^.LIGAIZQ ← NIL y *P* ← *Q*

Ejemplo 5.11

En la figura 5.17 se presenta un ejemplo de inserción al inicio de una lista doblemente ligada.

b) Inserción al final de una lista doblemente ligada

En este caso el nuevo nodo se coloca al final de la lista doblemente ligada, convirtiéndose en el último. El algoritmo 5.17 describe este proceso.

Algoritmo 5.17 Inserta_final

Inserta_final (*F*, DATO)

{Este algoritmo inserta un nodo al final de una lista doblemente ligada. *F* es el apuntador al último nodo de la lista, y DATO es la información que se almacenará en el nuevo nodo}
{*Q* es una variable de tipo puntero. INFOR, LIGAIZQ y LIGADER son los campos de cada nodo de la lista}

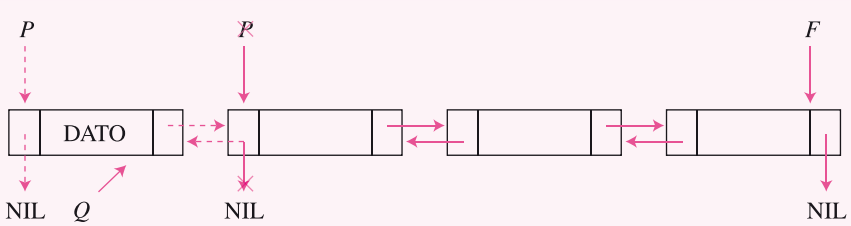
1. Crear(*Q*)
2. Hacer *Q*^.INFOR ← DATO, *F*^.LIGADER ← *Q*, *Q*^.LIGAIZQ ← *F*,
   Q^.LIGADER ← NIL y *F* ← *Q*

Ejemplo 5.12

En la figura 5.18 se presenta un ejemplo de inserción al final de una lista doblemente ligada.

FIGURA 5.17
Inserción al inicio de la lista.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada por la inserción de un nuevo nodo al inicio de la misma.



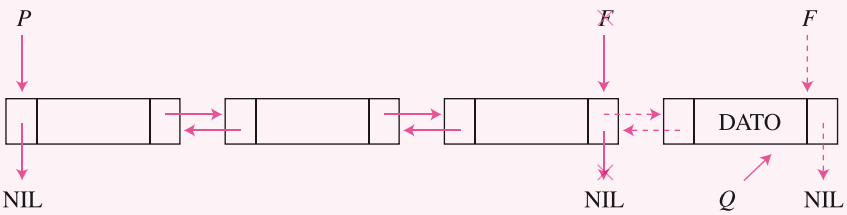


FIGURA 5.18
Inserción al final de la lista.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada, por la inserción de un nuevo nodo.

Al trabajar con un apuntador al último elemento de la lista, *F*, la operación de inserción se simplifica notablemente ya que se evita recorrer toda la lista.

c) Inserción de un nodo antes que otro en una lista doblemente ligada

En este caso el nuevo nodo se coloca precediendo a otro dado como referencia. Cabe señalar que sólo se presentará la operación de inserción de un nodo antes de otro dado como referencia, ya que las operaciones Antes_que_otro_ y Después_que_otro son simétricas.

Algoritmo 5.18 Inserta_antes_X

Inserta_antes_X (*P*, DATO, *X*)

{Este algoritmo inserta un nodo antes de otro dado como referencia, con información *X*. *P* es el apuntador al primer nodo de la lista, y DATO es la información que se almacenará en el nuevo nodo}
{*Q*, *T* y *R* son variables de tipo apuntador. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

```
1. Hacer Q ← P
2. Mientras ((Q^.LIGADER ≠ NIL) y (Q^.INFOR ≠ X)) Repetir
        Hacer Q ← Q^.LIGADER
3. {Fin del ciclo del paso 2}
4. Si (Q^.INFOR = X)
     entonces
        Crear (T) {Se crea el nuevo nodo}
        Hacer T^.INFOR ← DATO, T^.LIGADER ← Q, R ← Q^.LIGAIZQ
         y Q^.LIGAIZQ ← T
   4.1 Si (P = Q)
            entonces
               Hacer P ← T y T^.LIGAIZQ ← NIL
            si no
               Hacer R^.LIGADER ← T y T^.LIGAIZQ ← R
```

```
4.2 {Fin del condicional del paso 4.1}
    si no
        escribir "El elemento no se encuentra en la lista"
5. {Fin del condicional del paso 4}
```

Ejemplo 5.13

En la figura 5.19 se presenta un ejemplo de inserción, aplicando el algoritmo 5.18.

5.4.4 Eliminación en listas doblemente ligadas

La operación de eliminación de un nodo en una lista doblemente ligada, al igual que en el caso de las listas simplemente ligadas, consiste en eliminar un elemento de la lista, redefiniendo los apuntadores correspondientes y liberando el espacio de memoria ocupado por el nodo. En la eliminación se pueden presentar diferentes casos, aunque algunos de ellos son simétricos, ya que cada nodo tiene apuntadores hacia delante —derecha— y atrás —izquierda—.

- Eliminar el primer nodo.
- Eliminar el último nodo.
- Eliminar el nodo con información *X*.
- Eliminar el nodo anterior/posterior al nodo con información *X*.

En los algoritmos que presentan la solución a los diferentes casos de borrado de un elemento de una lista, no se considera que ésta se encuentre vacía. Este caso, como ya se ha repetido en varias ocasiones, se puede controlar en el programa principal o bien con una condición simple al inicio de cada algoritmo.

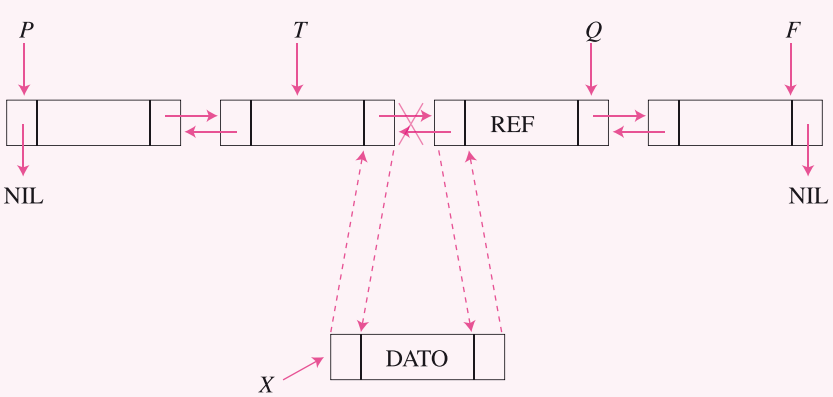


FIGURA 5.19
Inserción de nodos.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada, por la inserción de un nuevo nodo.

a) Eliminar el primer nodo de una lista doblemente ligada

Consiste en quitar el primer nodo de la lista, cualquiera que sea su información, redefiniendo el puntero al inicio de la misma. El algoritmo 5.19 describe este proceso.

Algoritmo 5.19 Elimina_inicio

Elimina_inicio (*P*, *F*)

{Este algoritmo elimina el primer elemento de una lista doblemente ligada. *P* y *F* son los apuntadores al primer y último nodos de la lista, respectivamente}
{*Q* es una variable de tipo apuntador. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

1. Hacer $Q \leftarrow P$
2. *Si* (Q^.LIGADER ≠ NIL) {Verifica si la lista tiene sólo un nodo}
   entonces
   Hacer $P \leftarrow Q$^.LIGADER y P^.LIGAIZQ ← NIL
   si no
   Hacer $P \leftarrow$ NIL y $F \leftarrow$ NIL
3. {Fin del condicional del paso 2}
4. Quitar (*Q*)

Ejemplo 5.14

En la figura 5.20 se presenta un ejemplo de eliminación del primer nodo de una lista doblemente ligada, mediante el algoritmo 5.19.

b) Eliminar el último nodo de una lista doblemente ligada

Este caso es simétrico al anterior; consiste en eliminar el último nodo de una lista doblemente ligada y redefinir el apuntador al final de ella.

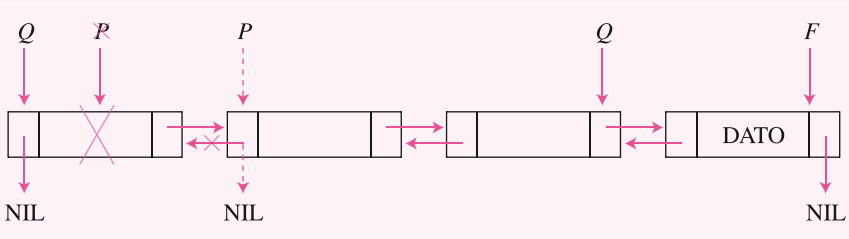


FIGURA 5.20
Eliminación del primer nodo de una lista.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada por la eliminación del primer nodo.

Algoritmo 5.20 Elimina_último

Elimina_último (*P*, *F*)

{Este algoritmo elimina el último elemento de una lista doblemente ligada. *P* y *F* son los apuntadores al primero y último nodos de la lista, respectivamente}
{*Q* es una variable de tipo puntero. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

1. Hacer $Q \leftarrow F$
2. *Si* (Q^.LIGAIZQ $\neq$ NIL) {Verifica si la lista tiene un solo nodo}
   entonces
   Hacer $F \leftarrow Q$^.LIGAIZQ y F^.LIGADER $\leftarrow$ NIL
   si no
   Hacer $F \leftarrow$ NIL y $P \leftarrow$ NIL
3. {Fin del condicional del paso 2}
4. Quitar (Q)

Ejemplo 5.15

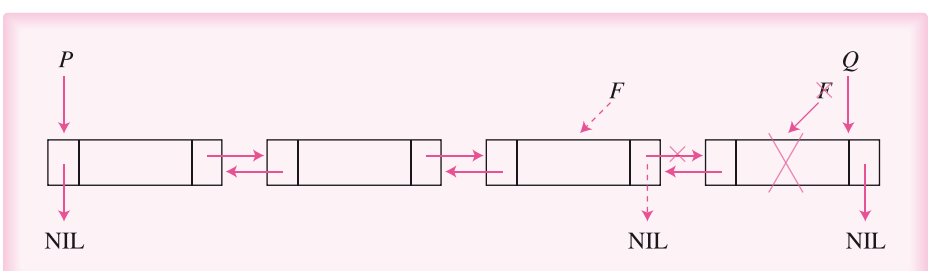
En la figura 5.21 se presenta un ejemplo de eliminación del último nodo de una lista doblemente ligada aplicando el algoritmo anterior.

c) Eliminar un nodo con información *X*

Este caso consiste en eliminar el nodo que contenga la información *X*, y establecer los apuntadores correspondientes entre su antecesor y su sucesor, respectivamente. Este caso tiene algunas variantes. El nodo que se quiere eliminar puede que no se encuentre en la lista, o bien que se halle y sea el primero, el último, el único, o que esté en cualquier posición intermedia de la estructura.

FIGURA 5.21
Eliminación del último nodo de una lista.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada por la eliminación del último nodo.



Algoritmo 5.21 Elimina_X

Elimina_X (*P*, *F*, *X*)

{Este algoritmo elimina el nodo con información *X* de una lista doblemente ligada. *P* y *F* son los apuntadores al primero y último nodos de la lista, respectivamente}
{*Q*, *T* y *R* son variables de tipo apuntador. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

```
1. Hacer Q ← P
2. Mientras ((Q^.LIGADER ≠ NIL) y (Q^.INFOR ≠ X)) Repetir
        Hacer Q ← Q^.LIGADER
3. {Fin del ciclo del paso 2}
4. Si (Q^.INFOR = X)
        entonces
        4.1 Si ((Q = P) y (Q = F)) {La lista tiene un solo nodo}
                entonces
                    Hacer P ← NIL y F ← NIL
                si no
          4.1.1 Si (Q = P) {Es el primero}
                    entonces
                        Hacer P ← Q^.LIGADER y P^.LIGAIZQ ← NIL
                    si no
         4.1.1.1 Si (Q = F) {Es el último}
                        entonces
                            Hacer F ← Q^.LIGAIZQ y F^.LIGADER ← NIL
                        si no {Es un nodo intermedio}
                            Hacer T ← Q^.LIGAIZQ, R ← Q^.LIGADER
                                  T^.LIGADER ← R y R^.LIGAIZQ ← T
          4.1.1.2 {Fin del condicional del paso 4.1.1.1}
        4.1.2 {Fin del condicional del paso 4.1.1}
        4.2 {Fin del condicional del paso 4.1}
                Quitar (Q)
        si no
                Escribir "El elemento con información X no se encuentra en la lista"
5. {Fin del condicional del paso 4}
```

Ejemplo 5.16

En la figura 5.22 se presenta un ejemplo de eliminación de un nodo con información *X* en una lista doblemente ligada.

d) Eliminar el nodo anterior al nodo con información *X*

En este caso se trata de eliminar el nodo anterior a uno dado como referencia que contenga la información *X*. El caso también tiene algunas variantes. Puede ser que el nodo con información *X* no se encuentre en la lista o bien se encuentre, y sea el primero —en

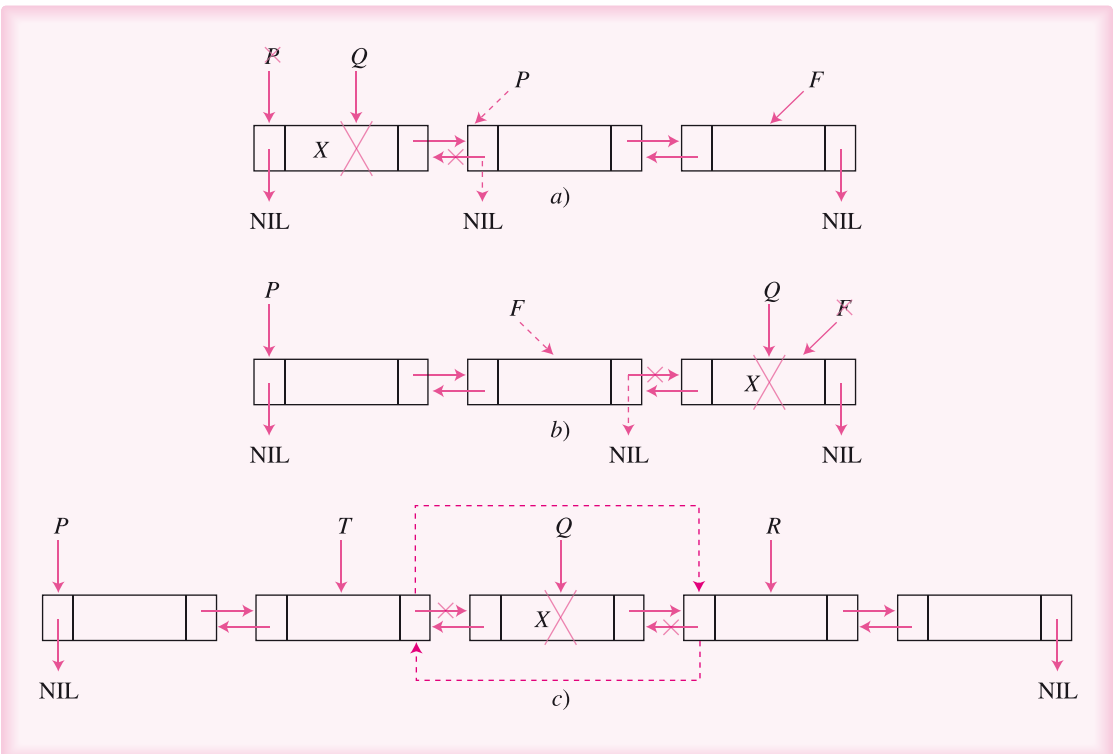


FIGURA 5.22

Eliminación de un nodo con información *X*. *a*) El nodo es el primero. *b*) El nodo es el último. *c*) El nodo es intermedio.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada, por la eliminación de un nodo.

ese caso no hay nada que eliminar—, el segundo —se elimina el primero de la lista—, o se encuentre en cualquier otra posición. El algoritmo 5.22 describe los pasos necesarios para llevar a cabo esta operación.

Algoritmo 5.22 Elimina_antes_X

Elimina_antes_X (*P*, *X*)

{Este algoritmo elimina, si se puede, el nodo anterior a aquel que contiene la información *X*. *P* es el apuntador al primer nodo de la lista}
{*Q*, *T* y *R* son variables de tipo apuntador. INFOR, LIGADER y LIGAIZQ son los campos de cada nodo de la lista}

```
1. Hacer Q ← P
2. Mientras ((Q^.LIGADER ≠ NIL) y (Q^.INFOR ≠ X)) Repetir
        Hacer Q ← Q^.LIGADER
3. {Fin del ciclo del paso 2}
4. Si (Q^.INFOR = X)
        entonces
        4.1 Si (P = Q)
                entonces
                    Escribir "No existe nodo anterior al primero"
                si no
                    Hacer T ← Q^.LIGAIZQ
            4.1.1 Si (P = T) {Es el primer nodo de la lista}
                    entonces
                        Hacer P ← Q y P^.LIGAIZQ ← NIL
                    si no
                        Hacer R ← T^.LIGAIZQ, Q^.LIGAIZQ ← R y
                         R^.LIGADER ← Q
            4.1.2 {Fin del condicional del paso 4.1.1}
                    Quitar (T)
        4.2 {Fin del condicional del paso 4.1}
            si no
                Escribir "El elemento con información X no se encuentra en la lista"
5. {Fin del condicional del paso 4}
```

Ejemplo 5.17 La figura 5.25 contiene un ejemplo de eliminación, mediante el algoritmo anterior.

FIGURA 5.23
Eliminación de nodos.

Nota: Las flechas discontinuas indican los cambios originados en la lista doblemente ligada, por la eliminación de un nodo.

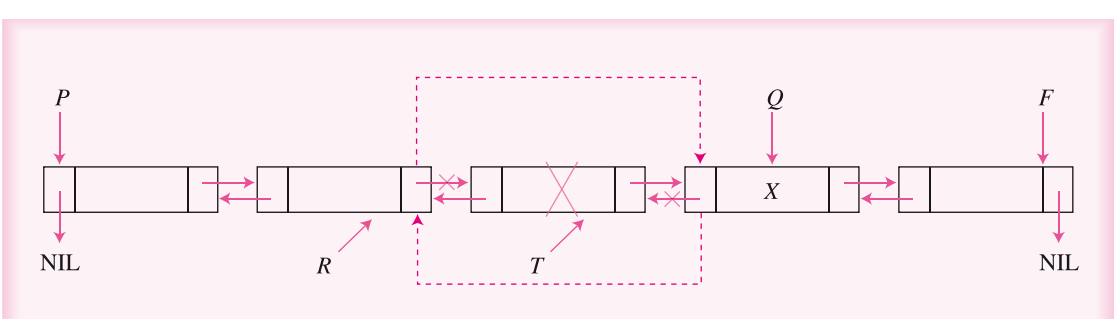
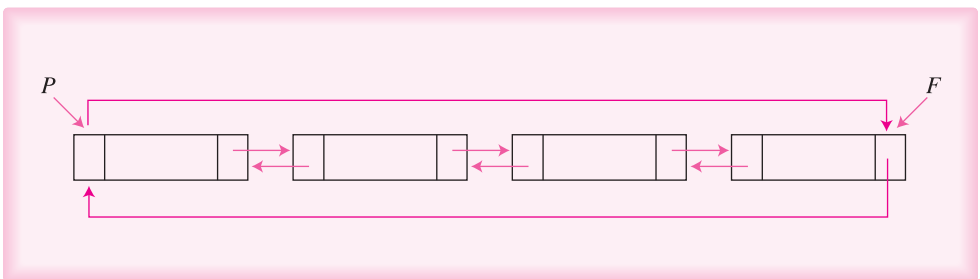


FIGURA 5.24
Lista doblemente ligada circular.



5.5 LISTAS DOBLEMENTE LIGADAS CIRCULARES

En las **listas doblemente ligadas circulares**, el campo liga izquierda del primer nodo de la lista apunta al último, y el campo liga derecha de éste apunta al primero. La figura 5.24 representa una estructura de este tipo.

La principal ventaja de las listas circulares es que permiten la navegación en cualquier sentido a través de la misma y, además, se puede recorrer toda la lista partiendo de cualquier nodo, siempre que tengamos un apuntador a éste. Sin embargo, debemos destacar que es necesario establecer condiciones adecuadas para detener el recorrido de una lista y evitar caer en ciclos infinitos. Al igual que en el caso de listas simplemente ligadas circulares, se suele utilizar un nodo de cabecera (fig. 5.25).

Este nodo tendrá las características descritas anteriormente y servirá como referencia para detectar cuándo se ha recorrido totalmente la lista.

Hasta este momento se han estudiado las principales características de la estructura tipo lista, considerando todas sus variantes (fig. 5.26):

FIGURA 5.25
Lista doblemente ligada circular con nodo de cabecera.

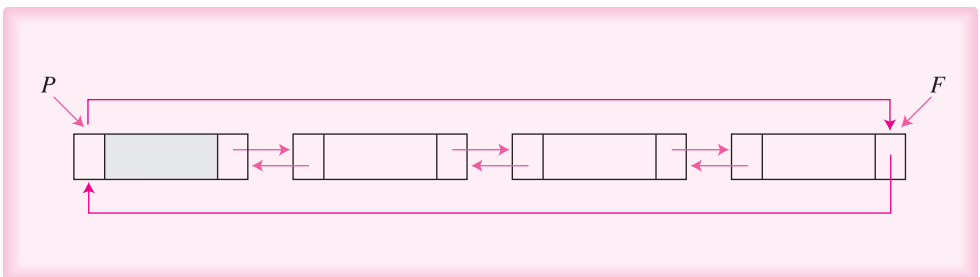
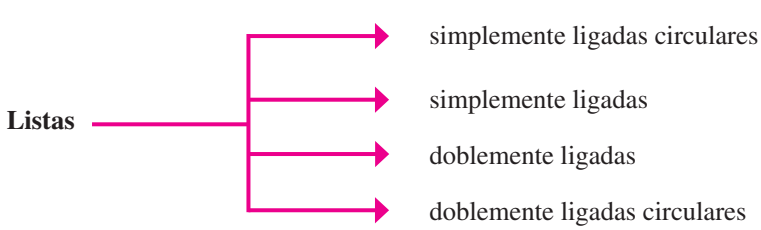


FIGURA 5.26



Además se han presentado los algoritmos utilizados para realizar las operaciones más importantes de listas. Las siguientes son algunas aplicaciones de listas.

5.6 APLICACIONES DE LISTAS

Dos de las aplicaciones más conocidas de listas son las siguientes:

- Representación de polinomios.
- Resolución de colisiones (*hash*).

En general se puede señalar que las listas son muy útiles para aquellas aplicaciones en las que se necesite dinamismo en el crecimiento y reducción de la estructura de datos usada para el almacenamiento de la información.

5.6.1 Representación de polinomios

Las listas se pueden emplear para almacenar los coeficientes diferentes de cero del polinomio, junto al exponente. Así, por ejemplo, dado el polinomio:

$$P(x) = 3X^4 + 0.5X^3 + 6X - 4$$

su representación mediante listas queda como se muestra en la figura 5.27.

El campo información de cada nodo de la lista contendrá dos campos: el campo COEFICIENTE y el campo EXPONENTE.

Cabe destacar que en el ejemplo anterior se utilizó una lista simplemente ligada, pero se pudo haber usado una circular o también una lista doblemente ligada.

5.6.2 Solución de colisiones (*hash*)

En el capítulo 9, al tratar el método de búsqueda por transformación de claves, se utilizaron listas para resolver colisiones —método de encadenamiento—. Con el objeto se evitar la reiteración y la redundancia de información, se sugiere remitirse a dicho capítulo.

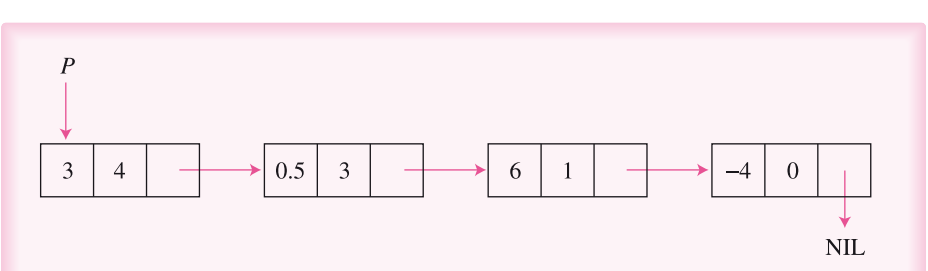


FIGURA 5.27
Representación de polinomios usando listas.

5.7 LA CLASE *LISTA*

La **clase *Lista*** tiene atributos y métodos, ambos dependen del tipo de lista que se esté definiendo. En esta sección se declara la clase correspondiente a una lista simplemente ligada. Consecuentemente, los atributos son los apuntadores al primero y último nodos de la lista, mientras que los métodos representan todas las operaciones analizadas anteriormente: Crea_inicio, Inserta_inicio, Inserta_final,…, Recorre_iterativo,…, Elimina_inicio, Elimina_final,…

La clase *Lista* utiliza la clase *Nodo* para declarar el tipo de sus atributos, la cual representa a los nodos de la lista. Es decir, cada nodo tiene dos atributos: uno para almacenar la información y el otro para guardar la dirección del siguiente nodo. Los métodos de esta clase son las operaciones válidas sobre sus miembros. En las figuras 5.28 y 5.29 se puede observar la representación gráfica de las clases *Nodo* y *Lista*, respectivamente.

Se tiene acceso a los miembros de un objeto de la clase *Lista* por medio de la notación de puntos. Cuando se asume que la variable LISTAOBJ es un objeto de la clase *Lista*, previamente creado, se puede hacer:

FIGURA 5.28
Clase *Nodo*.

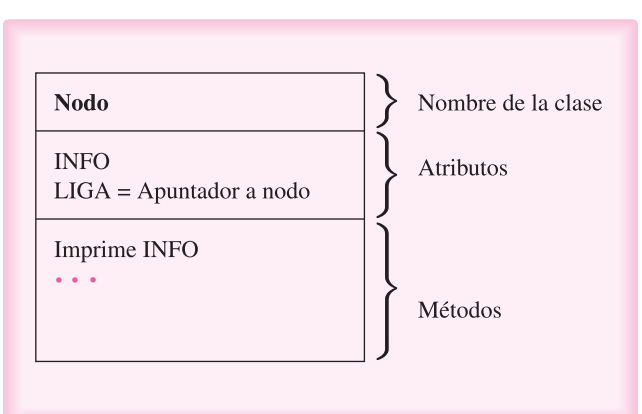
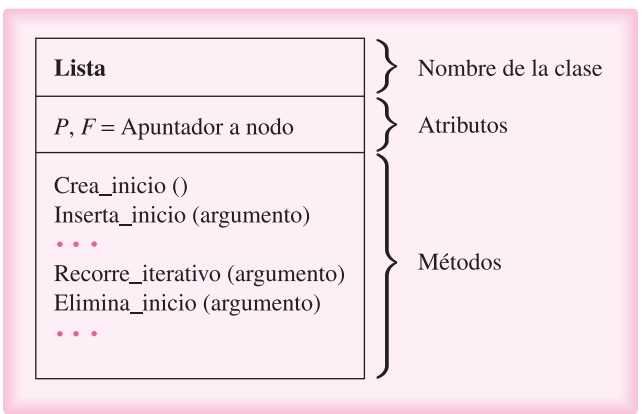


FIGURA 5.29
Clase *Lista*.



LISTAOBJ.Inserta_inicio(argumento) para invocar el método que inserta un nuevo elemento al inicio de la lista. En este método hay un solo argumento que representa el valor a guardar en el nuevo nodo. Todos los otros valores requeridos son miembros de la clase.

LISTAOBJ.Elimina_inicio(argumento) para invocar el método que elimina el primer nodo de la lista. En este método hay un único argumento, que es para regresar el valor eliminado; todos los otros valores requeridos son miembros de la clase.

EJERCICIOS

1. Defina un algoritmo para insertar, si es posible, un elemento antes de otro nodo dado como referencia en una lista ordenada.

2. Defina un algoritmo para insertar, si es posible, un elemento siguiendo a otro dado como referencia, en una lista ordenada.

3. Defina un algoritmo para insertar un elemento en una lista ordenada, de tal manera que no se altere el orden de la misma.

4. Defina un algoritmo para eliminar un nodo de una lista ordenada.

5. Escriba un subprograma que lea dos listas que se encuentran ordenadas y forme una tercera que resulte de la mezcla de los elementos de ambas listas.

6. Escriba un subprograma recursivo que, dadas dos listas ordenadas ascendentemente, las mezcle y genere una nueva lista ordenada en forma descendente.

7. Escriba un subprograma que, dada una lista que contiene números, la divida en dos listas independientes, una formada por los números positivos y otra por los números negativos.

8. Escriba un subprograma recursivo para imprimir toda la información de una lista.

9. Escriba un subprograma recursivo que busque un elemento *X* en una lista doblemente ligada.

10. Escriba un subprograma que elimine un elemento *X* de una lista circular.

11. Defina un algoritmo para insertar elementos en una lista circular.

12. Escriba los subprogramas “Mete_Pila” y “Saca_Pila” para insertar y eliminar, respectivamente, un elemento de una pila implementada por medio de una lista.

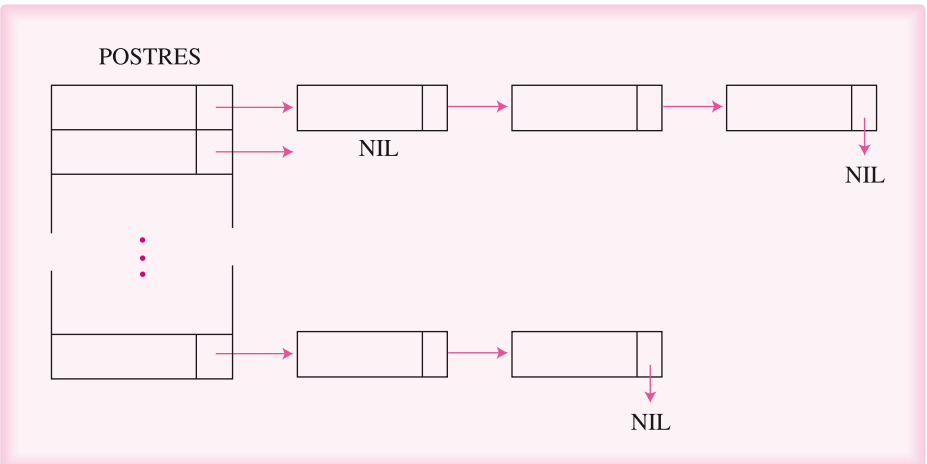
13. Escriba un subprograma recursivo que permita recorrer una lista doblemente ligada en ambos sentidos.

14. Defina un algoritmo recursivo para insertar un elemento siguiendo a otro nodo dado como referencia, en una lista doblemente ligada.

15. Escriba un subprograma recursivo para evaluar un polinomio representado por medio de una lista lineal.

16. Defina los algoritmos necesarios para implementar una estructura tipo cola mediante listas.

17. Se ha definido la siguiente estructura de datos:



En el arreglo "POSTRES" se almacenan nombres de postres, ordenados alfabéticamente. A su vez, cada elemento del arreglo tiene una lista de todos los ingredientes que requiere dicho postre.

Escriba un programa que:

a) Dado el nombre de un postre, imprima la lista de todos sus ingredientes.
b) Dado el nombre de un postre, inserte nuevos ingredientes a su correspondiente lista.
c) Dado el nombre de un postre, elimine alguno de sus ingredientes.
d) Dé de alta un postre con todos sus ingredientes.
e) Dé de baja un postre con todos sus ingredientes.

Nota: En cada uno de los puntos anteriores verifique todos los casos que pudieran presentarse.

18. Escriba un subprograma que elimine los elementos repetidos de una estructura tipo cola implementada por medio de listas.

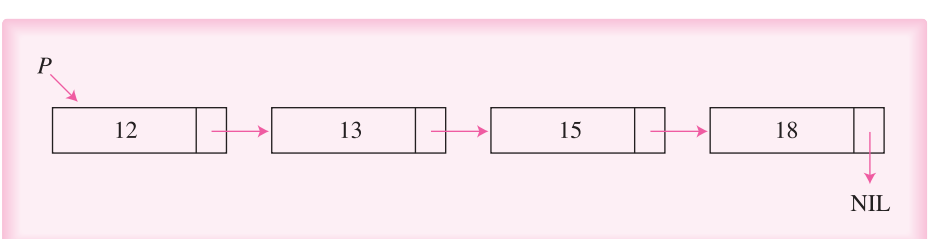
19. Retome la clase definida previamente para listas simplemente ligadas y prográmela en algún lenguaje orientado a objetos.

20. Retome la clase del problema anterior y adáptela para listas simplemente ligadas circulares con nodo de cabecera.

21. Defina una clase para listas doblemente ligadas. Incluya los atributos y todos los métodos que considere conveniente. Prográmela en algún lenguaje de programación orientado a objetos.

22. Retome la clase definida del problema anterior y adáptela para listas doblemente ligadas circulares con nodo de cabecera.

23. Considere que se tiene una lista de números enteros, ordenados crecientemente, como la que se muestra a continuación. Observe que faltan algunos números para tener todos los valores comprendidos entre el primero, 12, y el último, 18. Escriba un programa que "complete" la lista, de tal manera que la misma, una vez modificada, almacene todos los valores a partir del número del primer nodo hasta el número del último. Para el ejemplo, la lista guardará los números 12, 13, 14, 15, 16, 17 y 18. Utilice la clase de listas simplemente ligadas previamente definida.



Capítulo 6

ÁRBOLES

6.1 INTRODUCCIÓN

Hasta el momento sólo se han estudiado estructuras de datos lineales, tanto estáticas como dinámicas: a cada elemento siempre le sucede o le precede como máximo otro elemento. Al estudiar la estructura de datos árboles se introduce el concepto de ramificación entre componentes o nodos. Es decir, a un elemento le pueden preceder o suceder varios elementos.

Los **árboles** son las estructuras de datos **no lineales** y **dinámicas** de datos más importantes del área de computación. Dinámicas, puesto que las mismas pueden cambiar tanto de forma como de tamaño durante la ejecución del programa. No lineales, puesto que cada elemento del árbol puede tener más de un sucesor. Los **árboles balanceados** o **AVL** son la estructura de datos más eficiente para trabajar con la memoria principal —interna— del procesador, mientras que los **árboles *B*** y, especialmente la versión ***B*⁺**, representan la estructura de datos más eficiente para trabajar en memoria secundaria o externa.

En la tabla 6.1 se presentan las principales estructuras de datos clasificadas de acuerdo con su capacidad para cambiar en forma y tamaño, durante la ejecución del programa.

Es de observar que las pilas y colas no fueron consideradas en esta clasificación, puesto que dependen de la estructura que se utilice para implementarlas. Si se usan arreglos, se tratarán como estructuras estáticas. Si se implementan con listas, serán estructuras dinámicas. En ambos casos son lineales.

En la tabla 6.2 se presentan las principales estructuras de datos clasificadas según la distribución de sus elementos.

TABLA 6.1
Estructuras de datos estáticas y dinámicas

Estructuras estáticas	Estructuras dinámicas
Arreglos	Listas
Registros	Árboles
	Gráficas

TABLA 6.2
Estructuras de datos lineales y no lineales

Estructuras lineales	Estructuras no lineales
Arreglos	Árboles
Registros	Gráficas
Pilas	
Colas	
Listas	

6.2 ÁRBOLES EN GENERAL

Un **árbol** se puede definir como una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos, uno de los cuales es conocido como **raíz**. Además se crea una relación o parentesco entre los nodos dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ancestro, etcétera.

Formalmente se define un árbol de tipo *T* como una estructura homogénea resultado de la concatenación de un elemento de tipo *T* con un número finito de árboles disjuntos, llamados subárboles. Una forma particular de árbol es el árbol vacío. Los árboles son estructuras recursivas, ya que cada subárbol es a su vez un árbol.

Los árboles se pueden aplicar para la solución de una gran cantidad de problemas. Por ejemplo, se pueden utilizar para representar fórmulas matemáticas, para registrar la historia de un campeonato de tenis, para construir un árbol genealógico, para el análisis de circuitos eléctricos y para enumerar los capítulos y secciones de un libro.

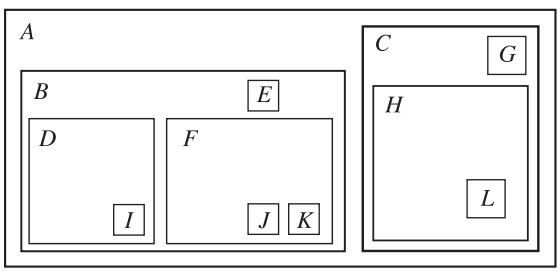
Un árbol se puede representar de diferentes formas y todas ellas se consideran equivalentes. En la figura 6.1 se presentan cinco notaciones diferentes correspondientes a un mismo árbol. En la figura 6.1*a* se utiliza un diagrama de Venn; en la figura 6.1*b* la anidación de paréntesis; en la figura 6.1*c* la notación decimal de Dewey; en la figura 6.1*d* la notación indentada, y, por último, en la figura 6.1*e* un grafo. Esta última representación es la que comúnmente se utiliza, y ha originado el término árbol por su parecido abstracto con el vegetal —raíz, ramas, hojas—, a pesar de que la raíz se dibuja arriba, aunque en el vegetal se encuentre abajo.

En el grafo se distinguen **nodos** —círculos— y **arcos** —líneas con flechas—. Los primeros se usan para almacenar la información y los últimos para establecer la relación entre los nodos. En el ejemplo de la figura 6.1*e* los nodos guardan letras y los arcos permiten ir de ciertos nodos a otros.

6.2.1 Características y propiedades de los árboles

La estructura tipo árbol tiene ciertas características y propiedades que la distinguen. A continuación se presentan las más importantes:

a) Todo árbol que no es vacío tiene un único nodo raíz.



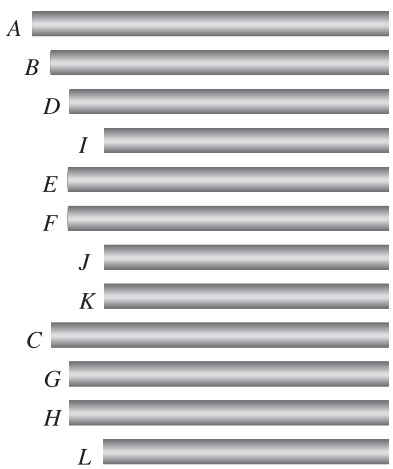
a)

(*A* (*B* (*D* (*I*), *E*, *F*, (*J*, *K*)), *C* (*G*, *H* (*L*))))

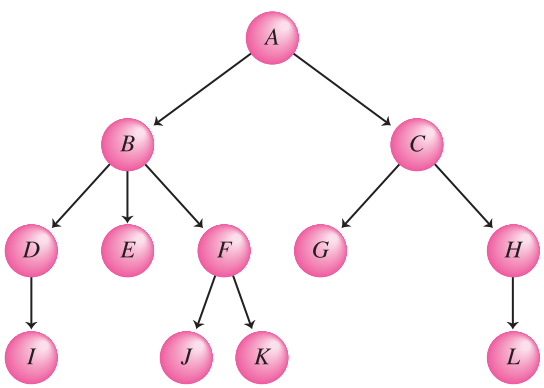
b)

1.*A*, 1.1.*B*, 1.1.1.*D*, 1.1.1.1.*I*, 1.1.2.*E*, 1.1.3.*F*, 1.1.3.1.*J*, 1.1.3.2.*K*, 1.2.*C*, 1.2.1.*G*, 1.2.2.*H*, 1.2.2.1.*L*

c)



d)



e)

FIGURA 6.1

Diferentes formas de representar una estructura de árbol. *a*) Diagramas de Venn. *b*) Anidación de paréntesis. *c*) Notación decimal de Dewey. *d*) Notación indentada. *e*) Grafo.

b) Un nodo *X* es descendiente directo de un nodo *Y*, si el nodo *X* es apuntado por el nodo *Y*. En este caso es común utilizar la expresión *X* **es hijo de** *Y*.

c) Un nodo *X* es antecesor directo de un nodo *Y*, si el nodo *X* apunta al nodo *Y*. En este caso es común utilizar la expresión *X* **es padre de** *Y*.

d) Se dice que todos los nodos que son descendientes directos —hijos— de un mismo nodo —padre— son **hermanos**.

e) Todo nodo que no tiene ramificaciones —hijos—, se conoce con el nombre de **terminal** u **hoja**.

f) Todo nodo que no es raíz ni terminal u hoja se conoce con el nombre de **interior**.

g) **Grado** es el número de descendientes directos de un determinado nodo.

h) **Grado del árbol** es el máximo grado de todos los nodos del árbol.

i) **Nivel** es el número de arcos que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene nivel 1.

j) **Altura** del árbol es el máximo número de niveles de todos los nodos del árbol.

A continuación se presenta un ejemplo para clarificar estos conceptos.

Ejemplo 6.1

Dado el árbol general de la figura 6.2, se puede afirmar lo siguiente:

1. *A* es la raíz del árbol.

2. *B* es hijo de *A*.
   C es hijo de *A*.
   D es hijo de *B*.
   E es hijo de *B*.
   L es hijo de *H*.

3. *A* es padre de *B*.
   B es padre de *D*.
   D es padre de *I*.
   C es padre de *G*.
   H es padre de *L*.

4. *B* y *C* son hermanos.
   D, *E* y *F* son hermanos.
   G y *H* son hermanos.
   J y *K* son hermanos.

5. *I*, *E*, *J*, *K*, *G* y *L* son nodos terminales u hojas.

6. *B*, *D*, *F*, *C* y *H* son nodos interiores.

7. El grado del nodo *A* es 2.
   El grado del nodo *B* es 3.
   El grado del nodo *C* es 2.
   El grado del nodo *D* es 1.
   El grado del nodo *E* es 0.
   El grado del árbol es 3.

8. El nivel del nodo *A* es 1.
   El nivel del nodo *B* es 2.
   El nivel del nodo *D* es 3.
   El nivel del nodo *C* es 2.
   El nivel del nodo *L* es 4.

9. La altura del árbol es 4.

6.2.2 Longitud de camino interno y externo

Se define la longitud de camino del nodo *X* como el número de arcos que se deben recorrer para llegar desde la raíz hasta el nodo *X*. Por definición la raíz tiene longitud de

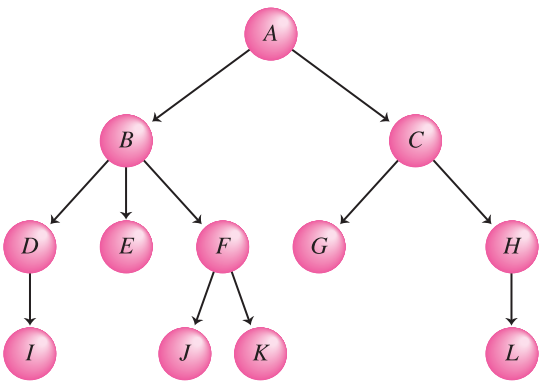


FIGURA 6.2
Árbol general.

camino 1, sus descendientes directos longitud de camino 2 y así sucesivamente. Considere el árbol de la figura 6.2. El nodo *B* tiene longitud de camino 2, el nodo *I* longitud de camino 4 y el nodo *H* longitud de camino 3.

Longitud de camino interno

La **longitud de camino interno** (LCI) del árbol es la suma de las longitudes de camino de todos los nodos del árbol. Esta medida es importante porque permite conocer los caminos que tiene el árbol. Se calcula por medio de la siguiente fórmula:

$$LCI = \sum_{i=1}^{h} n_i * i$$

Fórmula 6.1

donde i representa el nivel del árbol, h su altura y n_i el número de nodos en el nivel i.

La LCI del árbol de la figura 6.2 se calcula de esta forma:

$$LCI = 1 * 1 + 2 * 2 + 5 * 3 + 4 * 4 = 36$$

Ahora bien, la media de la longitud de camino interno (LCIM) se calcula dividiendo la LCI entre el número de nodos del árbol (n). La media es importante porque permite conocer, en promedio, el número de decisiones que se deben tomar para llegar a un determinado nodo partiendo desde la raíz. Se expresa:

$$LCIM = LCI/n$$

Fórmula 6.2

La LCIM del árbol de la figura 6.2 se calcula como se muestra a continuación:

$$LCIM = 36/3 = 3$$

Longitud de camino externo

Para definir la longitud de camino externo de un árbol es necesario primero explicar los conceptos de **árbol extendido** y **nodo especial**.

Un árbol extendido es aquel en el que el número de hijos de cada nodo es igual al grado del árbol. Si alguno de los nodos del árbol no cumple con esta condición, entonces deben incorporarse al mismo tantos nodos especiales como se requiera para llegar a cumplirla.

Los nodos especiales tienen como objetivo remplazar las ramas vacías o nulas, no pueden tener descendientes y normalmente se representan con la forma de un cuadrado. En la figura 6.3 se presenta el árbol extendido de la figura 6.2. El número de nodos especiales de este árbol es 25.

Se puede definir ahora la **longitud de camino externo** (LCE) de un árbol como la suma de las longitudes de camino de todos los nodos especiales del árbol. Se calcula por medio de la siguiente fórmula:

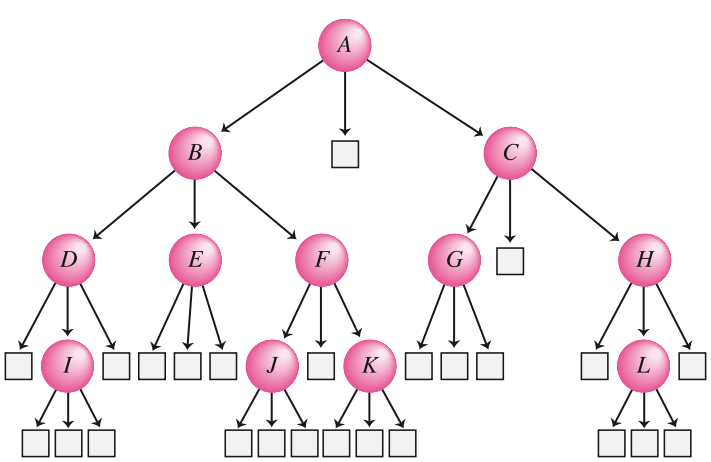
$$\text{LCE} = \sum_{i=2}^{h+1} n_{ei} * i \qquad \textbf{Fórmula 6.3}$$

donde i representa el nivel del árbol, h su altura y ne_i el número de nodos especiales en el nivel i. Observe que i comienza desde 2, puesto que la raíz se encuentra en el nivel 1 y no puede ser un nodo especial.

La LCE del árbol de la figura 6.3 se calcula de esta manera:

$$\text{LCE} = 1 * 2 + 1 * 3 + 11 * 4 + 12 * 5 = 109$$

FIGURA 6.3
Árbol extendido.



Ahora bien, la media de la longitud de camino externo (LCEM) se calcula dividiendo la LCE entre el número de nodos especiales del árbol (*ne*). Observe el lector la siguiente fórmula:

$$LCEM = LCE/ne \qquad \textbf{Fórmula 6.4}$$

e indica el número de arcos que se deben recorrer en promedio para llegar, partiendo desde la raíz, a un nodo especial cualquiera del árbol.

La LCEM del árbol de la figura 6.3 se calcula de la siguiente manera:

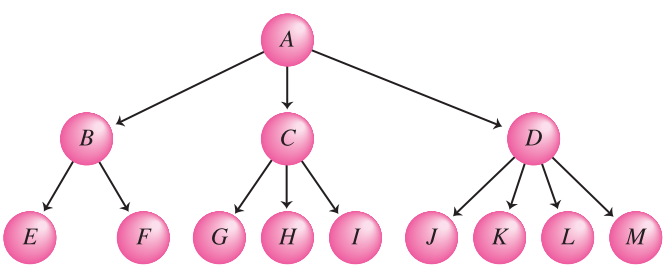
$$LCEM = 109/25 = 4.36$$

El siguiente ejemplo clarificará los conceptos de longitud de camino interno y externo.

Ejemplo 6.2

Dado el árbol general de la figura 6.4 y el árbol extendido de la figura 6.5 se calcula la longitud de camino interno:

FIGURA 6.4
Árbol general.

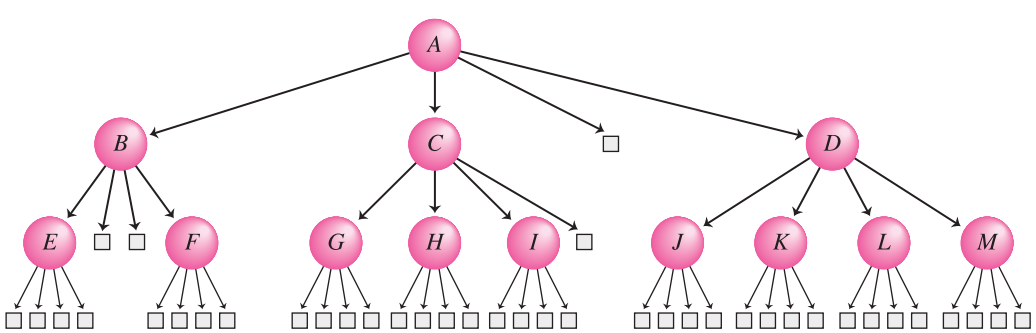


$$LCI = 1 * 1 + 3 * 2 + 9 * 3 = 34$$

La media de la longitud de camino interno:

$$LCIM = 34/13 = 2.61$$

FIGURA 6.5
Árbol extendido.



La longitud de camino externo:

$$LCE = 1 * 2 + 3 * 3 + 36 * 4 = 155$$

La media de la longitud de camino externo:

$$LCEM = 155/40 = 3.87$$

6.3 ÁRBOLES BINARIOS

Un árbol ordenado es aquel en el cual la distribución de las ramas sigue un cierto orden. Los árboles ordenados de grado 2 son de especial interés en el área de la computación porque permiten representar la información relacionada con la solución de muchos problemas. Estos árboles son conocidos con el nombre de **árboles binarios**.

En un árbol binario cada nodo puede tener como máximo dos subárboles y éstos se distinguen entre sí como el subárbol izquierdo y el subárbol derecho, según su ubicación con respecto al nodo raíz.

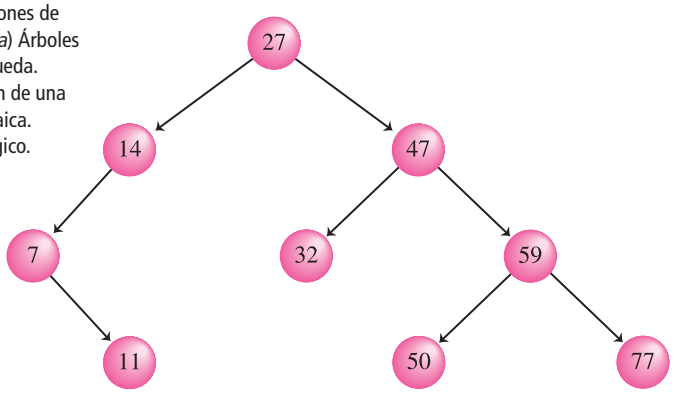
Formalmente se define un árbol binario tipo *T* como una estructura homogénea, resultado de la concatenación de un elemento de tipo *T*, llamado raíz, con dos árboles binarios disjuntos, llamados subárbol izquierdo y subárbol derecho. Un árbol binario especial es el árbol vacío.

Los árboles binarios tienen múltiples aplicaciones. Se les puede utilizar para representar la solución de un problema para el cual existen dos posibles alternativas (árbol de decisiones), para representar un árbol genealógico (construido en forma ascendente y donde se muestran los ancestros de un individuo dado), para representar la historia de un campeonato de tenis (construido en forma ascendente y donde existe un ganador, 2 finalistas, 4 semifinalistas y así sucesivamente) y para representar expresiones algebraicas construidas con operadores binarios. Esto sólo por citar algunos de sus múltiples usos.

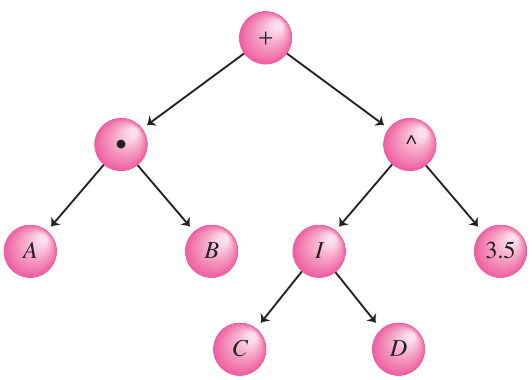
En la figura 6.6 se muestran tres diagramas correspondientes a una estructura de árbol binario. En la figura 6.6*a* hay un árbol binario de búsqueda (esta variante se presentará con detalle más adelante), en la figura 6.6*b* el árbol binario que representa la expresión (*A* * *B*) + (*C*/*D*) ^ 3.5 y en la figura 6.6*c* un árbol genealógico.

Los árboles ordenados de grado mayor a 2 representan también estructuras importantes. Se conocen con el nombre de árboles multicaminos y serán estudiados más adelante en este mismo capítulo.

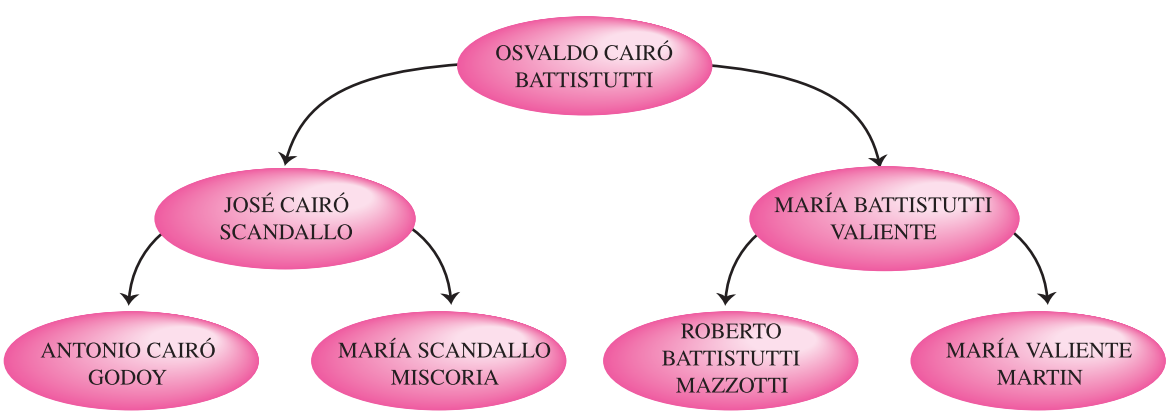
FIGURA 6.6
Distintas aplicaciones de árboles binarios. *a*) Árboles binarios de búsqueda. *b*) Representación de una expresión algebraica. *c*) Árbol genealógico.



a)



b)

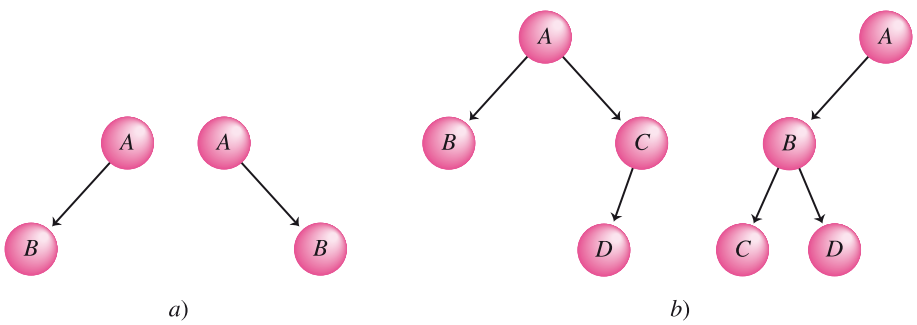


c)

6.3.1 Árboles binarios distintos, similares y equivalentes

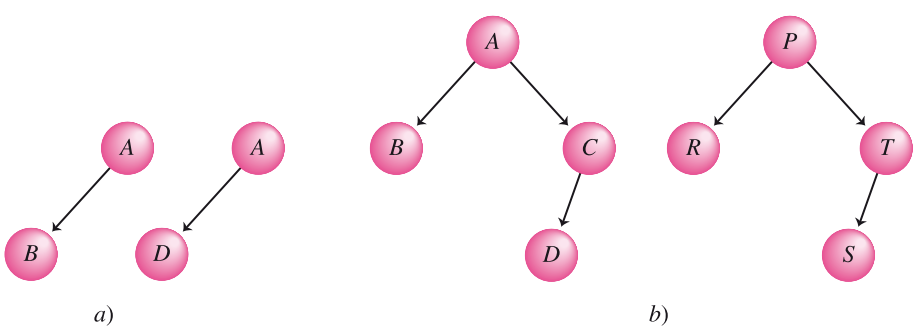
Dos árboles binarios son **distintos** cuando sus estructuras —la distribución de nodos y arcos— son diferentes. En la figura 6.7 se presentan dos ejemplos de árboles binarios distintos.

FIGURA 6.7
Árboles binarios distintos.



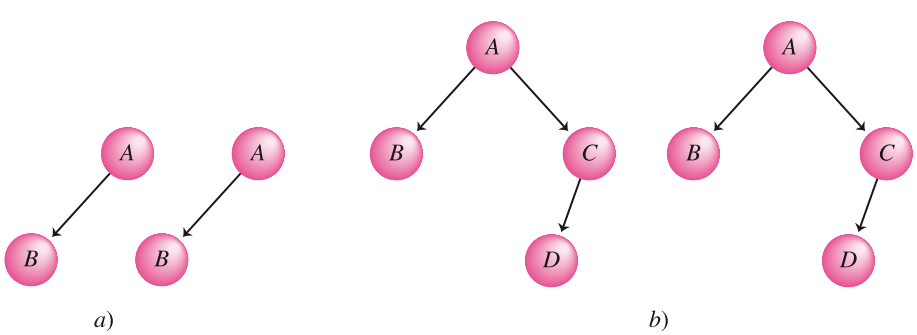
Dos árboles binarios son **similares** cuando sus estructuras son idénticas, pero la información que contienen sus nodos difiere entre sí. En la figura 6.8 se presentan dos ejemplos de árboles binarios similares.

FIGURA 6.8
Árboles binarios similares.



Por último, los árboles binarios **equivalentes** se definen como aquellos que son similares y además los nodos contienen la misma información. En la figura 6.9 se muestran dos ejemplos de árboles binarios equivalentes.

FIGURA 6.9
Árboles binarios equivalentes.



Con el fin de clarificar los conceptos anteriores, se presenta el siguiente ejemplo.

Ejemplo 6.3

Dados los árboles binarios de la figura 6.10, se puede afirmar lo siguiente:

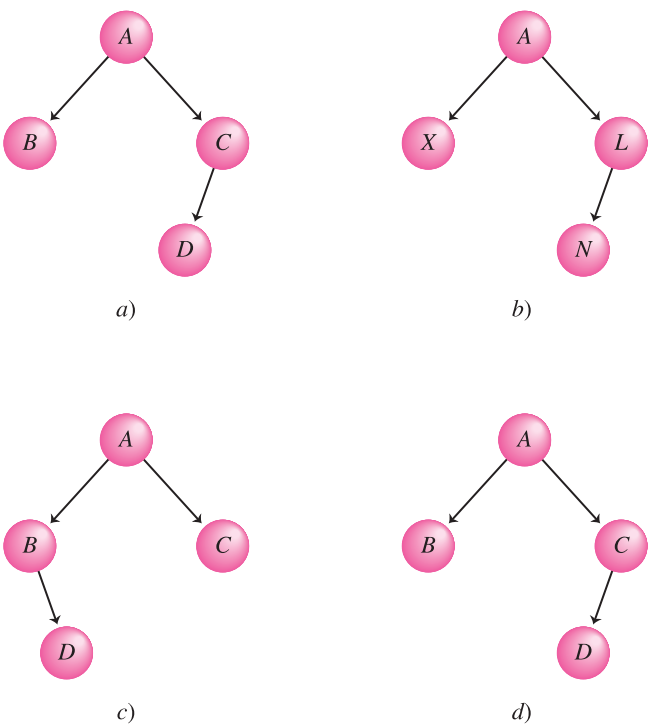


FIGURA 6.10
Árboles binarios distintos, similares y equivalentes.

- El árbol de la figura 6.10*c* es distinto de los árboles de la figura 6.10*a*, 6.10*b* y 6.10*d*.
- Los árboles de la figura 6.10*a*, 6.10*b* y 6.10*d* son similares.
- Los árboles de la figura 6.10*a* y 6.10*d* son equivalentes.

6.3.2 Árboles binarios completos

Se define un **árbol binario completo (ABC)** como un árbol en el que todos sus nodos, excepto los del último nivel, tienen dos hijos: el subárbol izquierdo y el subárbol derecho. En la figura 6.11 se presentan dos ejemplos de árboles binarios completos.

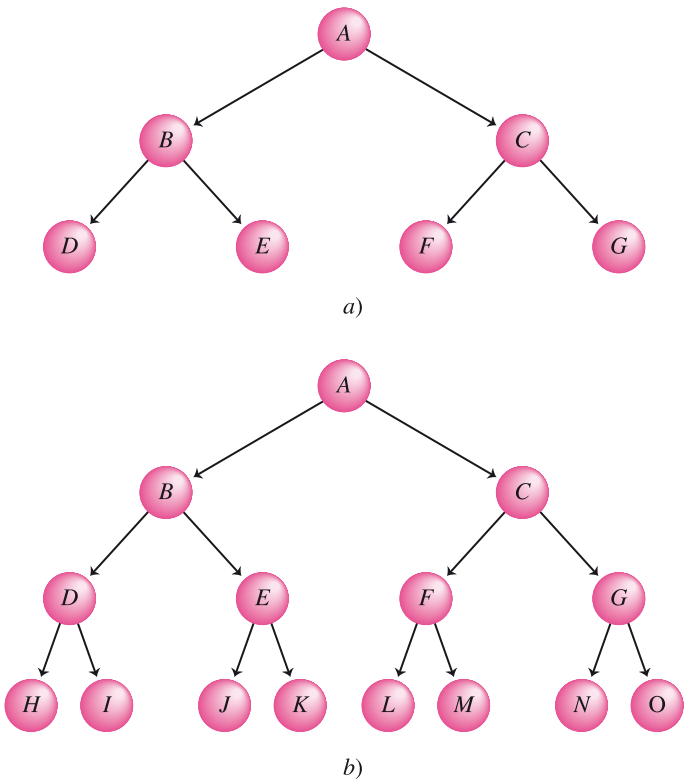


FIGURA 6.11
Árboles binarios completos. *a*) De altura 3. *b*) De altura 4.

El número de nodos de un árbol binario completo de altura h, se puede calcular aplicando la siguiente fórmula:

$$\text{NÚMERO DE NODOS(ABC)} = 2^h - 1 \qquad \textbf{Fórmula 6.5}$$

Así, por ejemplo, un árbol binario completo de altura 5 tendrá 31 nodos, uno de altura 9 tendrá 511 nodos y un árbol de altura 17 tendrá 131 071 nodos.

Cabe aclarar que existen algunos autores que definen un árbol binario completo de otra forma y otros que utilizan el término *lleno* para referirse a lo que en este libro se denomina *completo.*

6.3.3 Representación de árboles generales como binarios

Los árboles binarios, por las razones ya mencionadas, se aplican en la solución computacional de muchos problemas. Además, su uso se ve favorecido por su dinamismo, la no linealidad entre sus elementos y por su sencilla programación. Por lo tanto, resulta muy útil poder convertir árboles generales, con 0 a n hijos, en árboles binarios.

En esta sección se darán los pasos necesarios para lograrlo. Considere el árbol general de la figura 6.12*a*. Las operaciones que se deben aplicar para lograr la conversión del árbol general al árbol binario correspondiente son las siguientes:

1. Enlazar los hijos de cada nodo en forma horizontal —los hermanos—.
2. Relacionar en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además, se debe eliminar el vínculo de ese padre con el resto de sus hijos.
3. Rotar el diagrama resultante, aproximadamente 45 grados hacia la izquierda, y así se obtendrá el árbol binario correspondiente.

En la figura 6.12*b* se visualiza el árbol luego de aplicar los dos primeros pasos. En la figura 6.12*c* se observa el árbol binario, obtenido luego de aplicar el tercer paso.

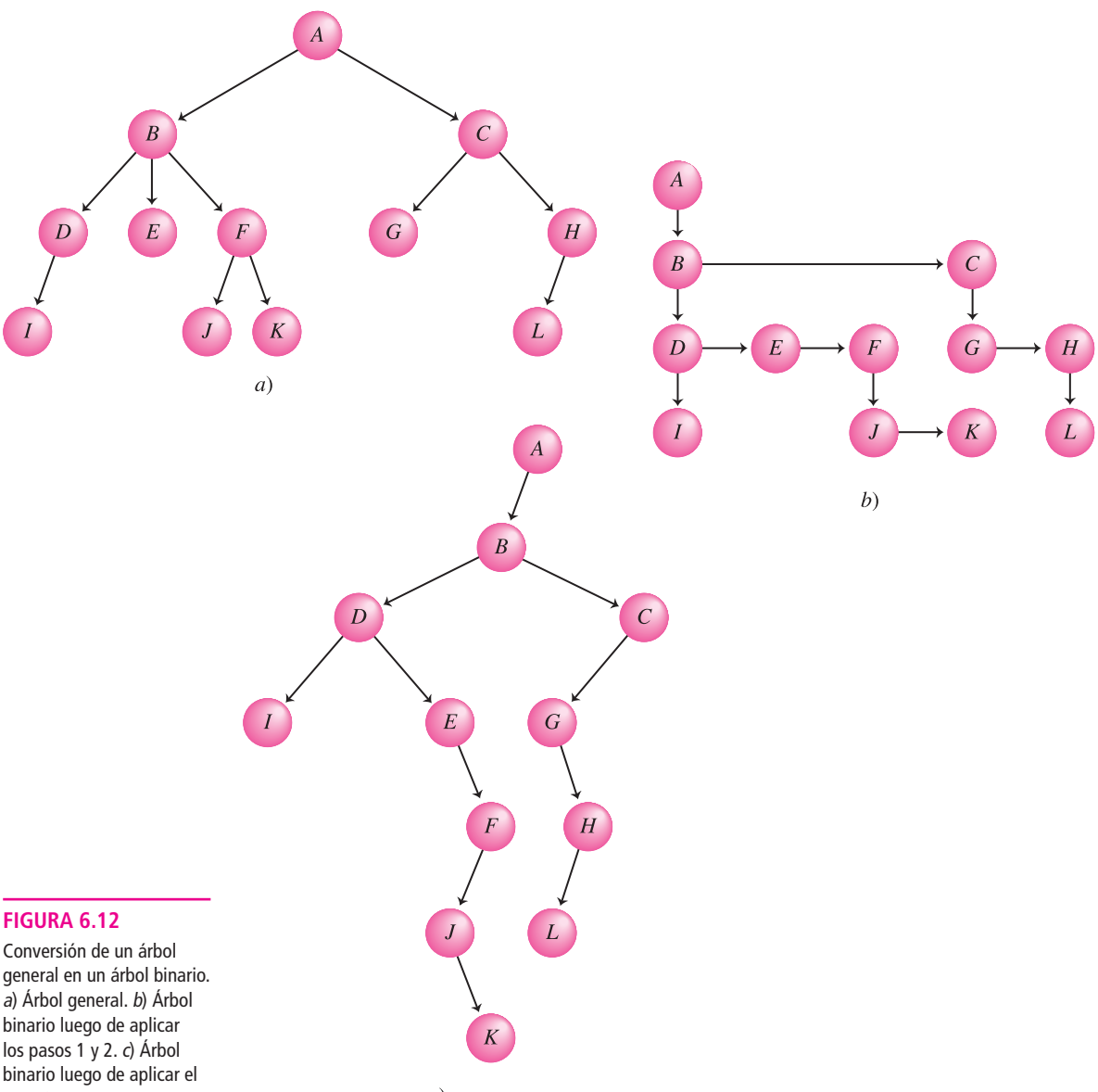


FIGURA 6.12
Conversión de un árbol general en un árbol binario. *a*) Árbol general. *b*) Árbol binario luego de aplicar los pasos 1 y 2. *c*) Árbol binario luego de aplicar el paso 3.

Ejemplo 6.4

Dado como dato el árbol general de la figura 6.13*a*, se debe convertir a un árbol binario. En la figura 6.13*b* se observa una gráfica del árbol luego de aplicar los dos primeros pasos. En la figura 6.13*c* se observa el árbol binario que se obtiene luego de que se aplica el tercer paso.

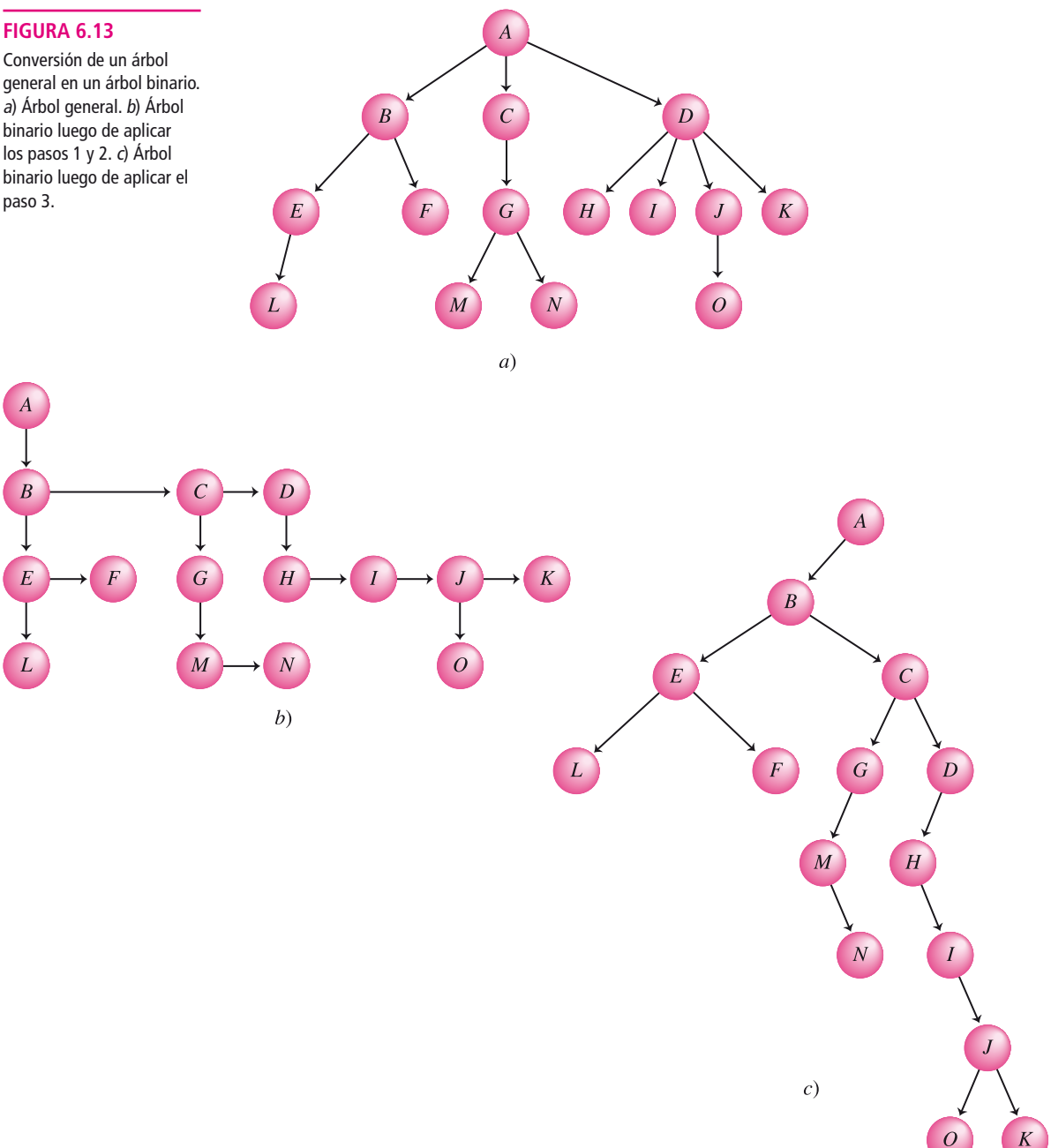


FIGURA 6.13
Conversión de un árbol general en un árbol binario. *a*) Árbol general. *b*) Árbol binario luego de aplicar los pasos 1 y 2. *c*) Árbol binario luego de aplicar el paso 3.

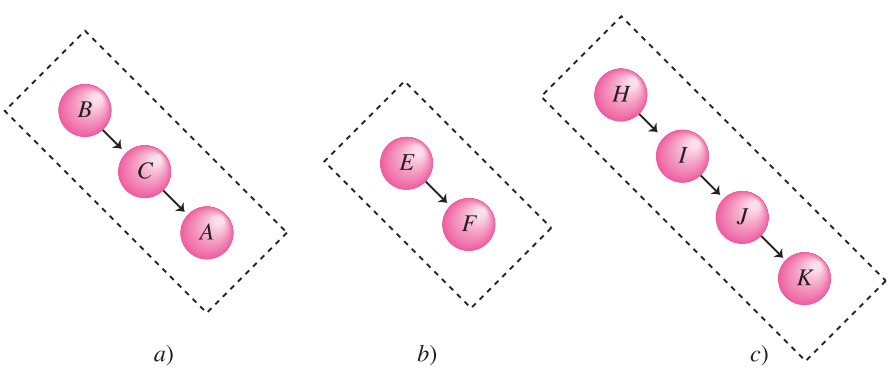
Observe que para todo nodo de un árbol binario, generado a partir de un árbol general, se debe cumplir lo siguiente:

1. Si la rama derecha de cada nodo, excepto el nodo raíz, es distinta de vacío se encuentra un nodo que era hermano de éste en el árbol general. De la figura 6.13*c* podemos deducir que *C* era hermano de *B*. Aplicando el mismo criterio deducimos también que *D* era hermano de *C* y, por lo tanto, por transitividad, hermano de *B*. Otras deducciones que se pueden realizar observando la figura 6.13*c* son las siguientes:

- *E* y *F* eran hermanos.
- *M* y *N* eran hermanos.
- *H*, *I*, *J* y *K* eran hermanos.

Es de notar que los hermanos se encuentran en la gráfica en una línea oblicua continua, orientada 45 grados hacia la derecha. En la figura 6.14 se presentan tres diagramas diferentes donde se pueden observar algunos ejemplos.

FIGURA 6.14
Nodos hermanos.



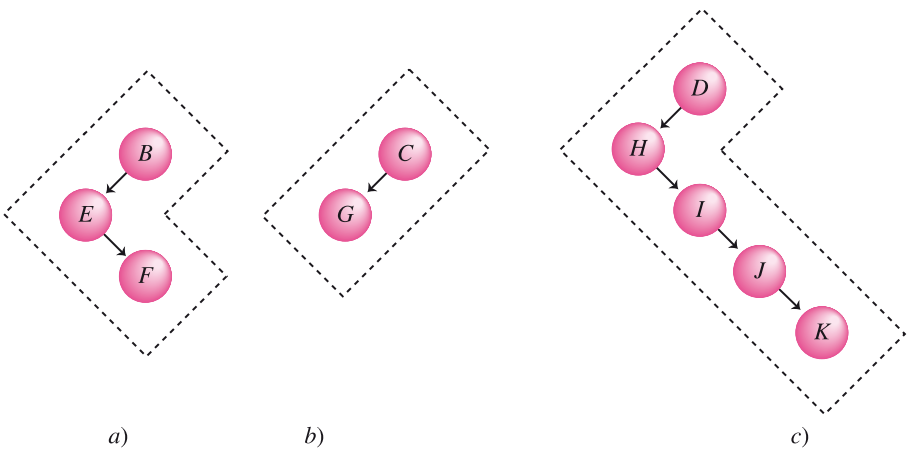
2. En la rama izquierda de cada nodo —si ésta es distinta de vacío— se encuentra un nodo que era hijo de éste en el árbol general. De la figura 6.13*c* podemos deducir que *E* era hijo de *B* y como por (1) *F* era hermano de *E*, podemos afirmar que *F* era también hijo de *B*. Otras deducciones que se pueden realizar observando la figura 6.13*c* son las siguientes:

- *B*, *C* y *D* eran hijos de *A*.
- *M* y *N* eran hijos de *G*.
- *G* era hijo de *C*.

Es de notar que los hijos de un nodo se encuentran en la gráfica, primero en una línea oblicua continua orientada 45 grados hacia la izquierda y luego en una línea continua oblicua orientada 45 grados hacia la derecha. En la figura 6.15 hay tres diagramas diferentes donde se observan algunos ejemplos.

En la figura 6.15*b* no existe línea oblicua orientada hacia la derecha porque *G* es el único hijo del nodo *C*.

FIGURA 6.15
Nodos hijos.

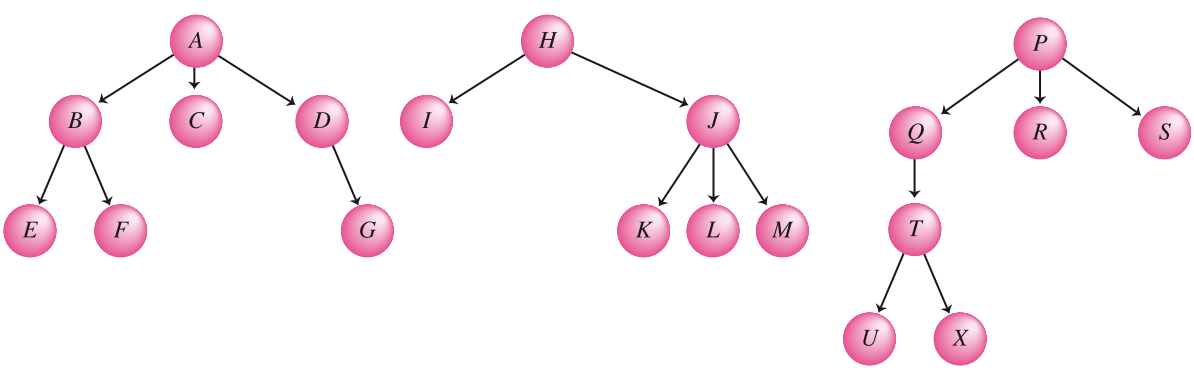


6.3.4 Representación de un bosque como árbol binario

Un **bosque** representa un conjunto normalmente ordenado de uno o más árboles generales. Es posible utilizar el algoritmo de conversión analizado en la sección anterior, con algunas modificaciones, para generar un árbol binario a partir de un bosque.

Considere por ejemplo el bosque, formado por tres árboles generales, de la figura 6.16. Los pasos que se deben aplicar para lograr la conversión del bosque a un árbol binario son los siguientes:

FIGURA 6.16
Bosque de árboles generales.

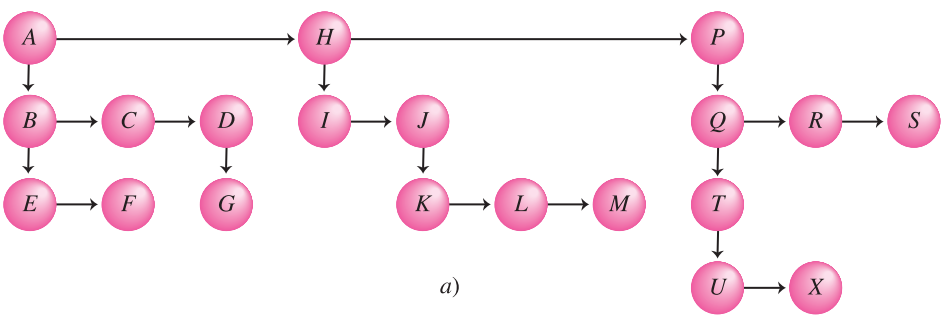


1. Enlazar en forma horizontal las raíces de los distintos árboles generales.
2. Relacionar los hijos de cada nodo —los hermanos— en forma horizontal.
3. Enlazar en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además, se debe eliminar el vínculo del padre con el resto de sus hijos.
4. Rotar el diagrama resultante aproximadamente 45 grados hacia la izquierda y así se obtendrá el árbol binario correspondiente.

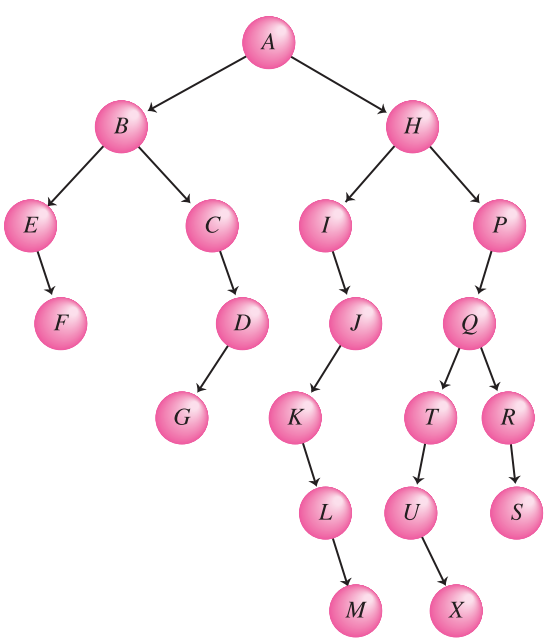
En la figura 6.17*a* se muestra el árbol luego de aplicar los tres primeros pasos. En la figura 6.17*b* se observa el árbol binario obtenido luego de que se realiza el cuarto paso.

FIGURA 6.17

Conversión de un bosque en árbol binario. *a*) Árbol binario luego de aplicar los pasos 1, 2 y 3. *b*) Árbol binario luego de aplicar el paso 4.



a)



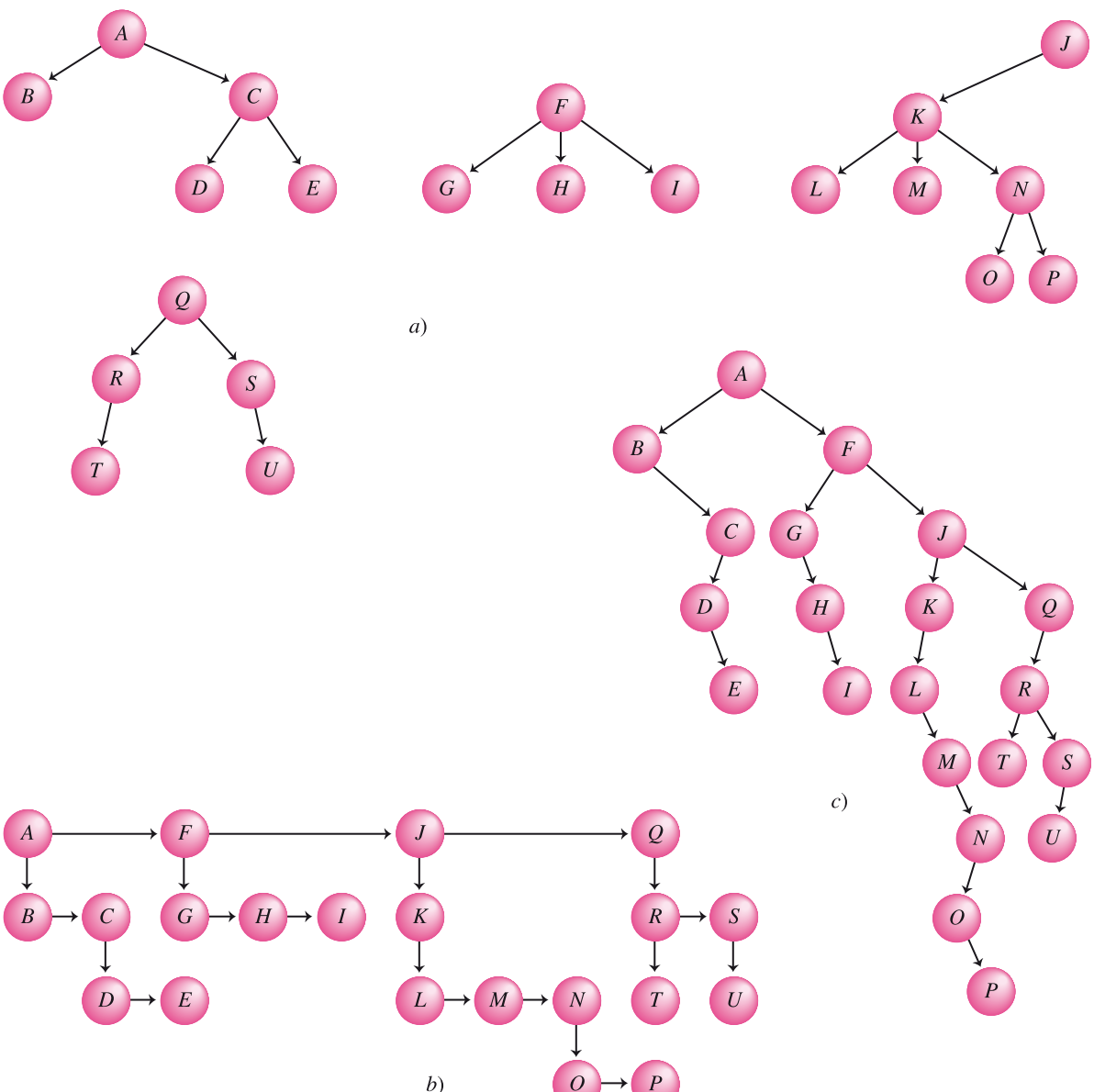
b)

Ejemplo 6.5

Dado como dato el bosque de la figura 6.18*a*, se desea convertirlo a un árbol binario. En la figura 6.18*b* se observa una gráfica del árbol luego de aplicar los tres primeros pasos. En la figura 6.18*c* se puede apreciar el árbol binario que se obtiene luego de que se aplica el cuarto paso.

FIGURA 6.18

Conversión de un bosque en árbol binario. *a*) Bosque. *b*) Árbol luego de aplicar el primero, segundo y tercer pasos. *c*) Árbol binario luego de aplicar el cuarto paso.



Es de notar que para todo nodo de un árbol binario, que se obtiene a partir de un bosque, se cumplen los dos incisos señalados en la conversión de un árbol general en un árbol binario.

6.3.5 Representación de árboles binarios en memoria

Las dos maneras más comunes de representar un árbol binario en memoria son:

1. Por medio de datos tipo puntero, también conocidos como variables dinámicas.
2. Por medio de arreglos.

En este libro se explicará y utilizará la primera forma, puesto que representa la más natural para tratar una estructura de datos de este tipo.

Al final del capítulo se presentará una breve introducción a los árboles, desde el punto de vista del paradigma orientado a objetos. Sin embargo, lo que se estudiará a continuación sigue siendo válido para las clases. Como en las otras estructuras de datos presentadas en este libro, los conceptos explicados son los fundamentos requeridos para el uso de las mismas, independientemente del paradigma y del lenguaje utilizado para su implementación.

Los nodos del árbol binario se representan como registros. Cada uno de ellos contiene como mínimo tres campos. En un campo se almacenará la información del nodo. Los dos restantes se utilizarán para apuntar los subárboles izquierdo y derecho, respectivamente, del nodo en cuestión.

Dado el nodo *T*:

En él se distinguen tres campos:

- **IZQ:** es el campo donde se almacena la dirección del subárbol izquierdo del nodo *T*.
- **INFO:** representa el campo donde se almacena la información del nodo. Normalmente en este campo y en el transcurso de este libro se almacenará un valor simple: número o carácter. Sin embargo, en la práctica es común almacenar en este campo cualquier tipo de dato.
- **DER:** es el campo donde se almacena la dirección del subárbol derecho del nodo *T*.

La definición de un árbol binario en lenguaje algorítmico es como sigue:

```
ENLACE = ^NODO
NODO   = REGISTRO
           IZQ: tipo ENLACE
           INFO: tipo de dato
           DER: tipo ENLACE
{Fin de la definición}
```

Nota: Es importante observar que se utiliza el símbolo ^ para representar el concepto de dato tipo puntero.

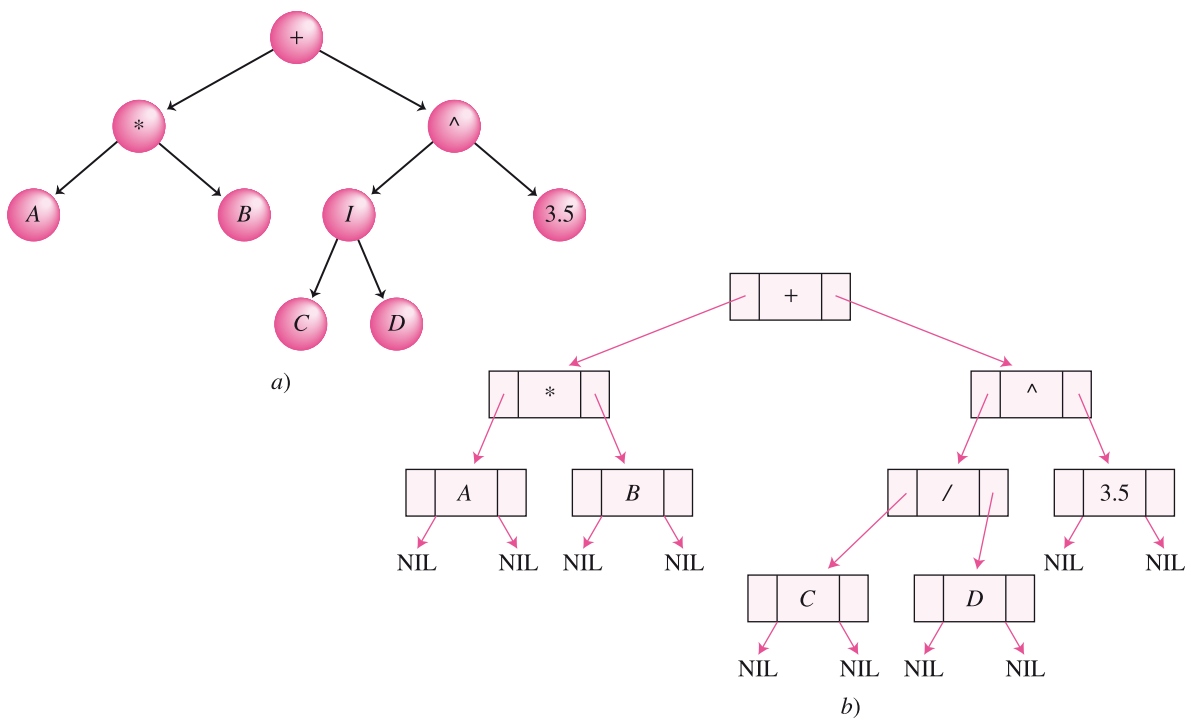
Ejemplo 6.6

Considere el árbol binario de la figura 6.19*a*, que representa la expresión algebraica (*A* * *B*) + (*C* / *D*) ^ 3.5. Su representación en memoria es como la que se muestra en la figura 6.19*b*.

Note el lector que en la figura 6.19*b* se utiliza el término NIL para hacer referencia al árbol vacío.

FIGURA 6.19

Representación de un árbol binario en memoria. *a*) Árbol binario. *b*) Su representación en memoria.



Como todas las estructuras de datos, los árboles tienen asociadas ciertas operaciones. A continuación se presentan los algoritmos de algunas de estas operaciones y más adelante, cuando se estudien otros tipos de árboles, se explicarán otras.

6.3.6 Operaciones en árboles binarios

Una de las operaciones básicas de un árbol binario es la creación del mismo en memoria. Un algoritmo muy simple para formar un árbol, por medio de la creación dinámica de nodos y la asignación a éstos de información, es el que se muestra a continuación:

Algoritmo 6.1 Crea_árbol

Crea_árbol (APNODO)

{El algoritmo crea un árbol binario en memoria. APNODO es una variable de tipo ENLACE —puntero a un nodo—. La primera vez APNODO se crea en el programa principal}
{INFO, IZQ y DER son campos del registro NODO. INFO es de tipo carácter. IZQ y DER son de tipo puntero. Las variables RESP y OTRO son de tipo carácter y de tipo ENLACE, respectivamente}

```
1. Leer APNODO^.INFO  {Lee la información y se guarda en el nodo}
2. Escribir "¿Existe nodo por izquierda: 1(Sí) – 0(No)?"
3. Leer RESP
4. Si (RESP = "Sí")
      entonces
         Crear(OTRO) {Se crea un nuevo nodo}
         Hacer APNODO^.IZQ ← OTRO
         Regresar a Crea_árbol con APNODO^.IZQ {Llamada recursiva}
      si no
         Hacer APNODO^.IZQ ← NIL
5. {Fin del condicional del paso 4}
6. Escribir "¿Existe nodo por derecha: 1(Sí) – 0(No)?"
7. Leer RESP
8. Si (RESP = "Sí")
      entonces
         Crear(OTRO) {Se crea un nuevo nodo}
         Hacer APNODO^.DER ← OTRO
         Regresar a Crea_árbol con APNODO^.DER  {Llamada recursiva}
      si no
         Hacer APNODO^.DER ← NIL
9. {Fin del condicional del paso 8}
```

Una vez que se crea el árbol binario, se pueden realizar otras operaciones sobre sus elementos: recorrer todos los nodos, insertar un nuevo nodo, eliminar alguno de los existentes o buscar un valor determinado.

Una de las operaciones más importantes que se realiza en un árbol binario es el recorrido de los mismos. Recorrer significa visitar los nodos del árbol en forma ordenada, de tal manera que todos los nodos del mismo sean visitados una sola vez. Existen tres formas diferentes de efectuar el recorrido y todas ellas de naturaleza recursiva; éstas son:

a) Recorrido en **preorden**
- Visitar la raíz
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho

b) Recorrido en **inorden**
- Recorrer el subárbol izquierdo

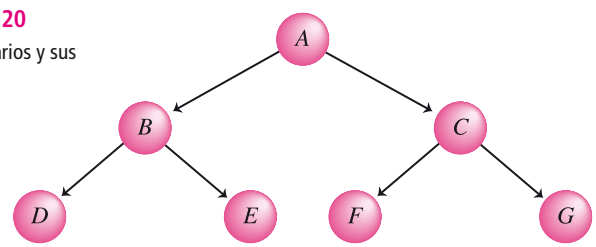
- Visitar la raíz
- Recorrer el subárbol derecho

c) Recorrido en **posorden**
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- Visitar la raíz

En la figura 6.20 se muestran tres árboles binarios con el resultado que se obtiene al efectuar los diferentes tipos de recorrido. En este ejemplo, la visita del nodo implicó la impresión de su contenido.

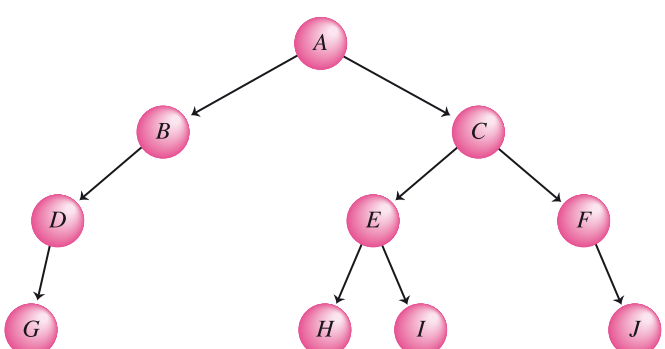
Note que en un árbol binario que representa una expresión algebraica, por ejemplo, el árbol de la figura 6.20*c*, la impresión de la información de sus nodos, usando el recorrido preorden, produce la notación polaca prefija. En el caso del recorrido inorden se obtiene la notación convencional y, por último, el recorrido posorden produce la notación polaca posfija. Aunque, cabe aclarar, sin los paréntesis respectivos que indican la precedencia de los distintos operadores.

FIGURA 6.20
Árboles binarios y sus recorridos.



a)

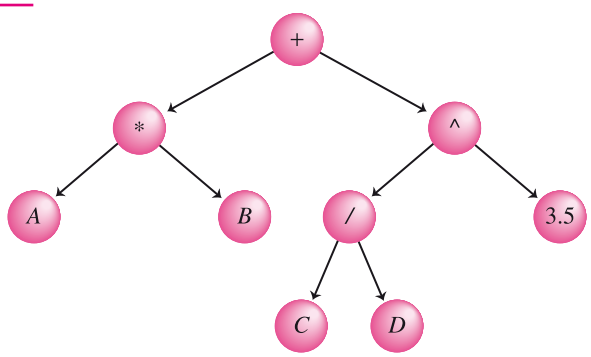
PREORDEN: *A B D E C F G*
INORDEN: *D B E A F C G*
POSORDEN: *D E B F G C A*



b)

PREORDEN: *A B D G K C E H I F J*
INORDEN: *G K D B A H E I C F J*
POSORDEN: *K G D B H I E J F C A*

FIGURA 6.20
(*continuación*)



c)

PREORDEN: + * *A B* ^ / *C D* 3.5
INORDEN: *A* * *B* + / *D* ^ 3.5
POSORDEN: *A B* * *C D* / 3.5 ^+

Se analizan a continuación los algoritmos que efectúan los diferentes tipos de recorridos en un árbol binario.

Algoritmo 6.2 Preorden

Preorden (APNODO)

{Este algoritmo realiza el recorrido preorden de un árbol binario. APNODO es un dato de tipo ENLACE —puntero a un nodo—}
{INFO, IZQ y DER son campos del registro nodo. INFO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero}
1. *Si* (APNODO ≠ NIL*) entonces*
Visitar el APNODO {Escribir NODO^.INFO}
Regresar a Preorden con APNODO^.IZQ
{Llamada recursiva a Preorden con la rama izquierda del nodo en cuestión}
Regresar a Preorden con APNODO^.DER
{Llamada recursiva a Preorden con la rama derecha del nodo en cuestión}
2. {Fin del condicional del paso 1}

Nota: Cabe destacar que el término *visitar* se puede reemplazar por cualquier otra instrucción válida, por ejemplo escribir, sumar o comparar la información del nodo. Note que esta aclaración se aplica también para los otros tipos de recorridos.

Ejemplo 6.7

En la siguiente tabla se presentan los pasos necesarios para obtener el recorrido preorden del árbol binario de la figura 6.20 *a*), utilizando el algoritmo 6.2.

En la columna *Pila: rama pendiente de visitar*, la llamada (*N*) indica el orden en el cual las ramas pendientes de visitar se introdujeron en la pila. En la columna *Nodo actual* la llamada (*N*) señala la rama que se extrajo de la pila. Observe el lector que el orden en que los nodos se visitaron es:

A - B - D - E - C - E - G

TABLA 6.3
Recorrido preorden

Paso	Nodo actual	Nodo visitado	Rama a visitar	Pila: rama pendiente de visitar
1	*A*	*A*	*A*^.IZQ → *B*	*A*^.DER → *C* (1)
2	*B*	*B*	*B*^.IZQ → *D*	*B*^.DER → *E* (2)
3	*D*	*D*	*D*^.IZQ → NIL	*D*^.DER → NIL (3)
4	NIL			
5	(3) NIL			
6	(2) *E*	*E*	*E*^.IZQ → NIL	*E*^.DER → NIL (4)
7	NIL			
8	(4) NIL			
9	(1) *C*	*C*	*C*^.IZQ → *F*	*C*^.DER → *G* (5)
10	*F*	*F*	*F*^.IZQ → NIL	*F*^.DER → NIL (6)
11	NIL			
12	(6) NIL			
13	(5) *G*	*G*	*G*^.IZQ → NIL	*G*^.DER → NIL (7)
14	NIL			
15	(7) NIL			

Algoritmo 6.3 Inorden

Inorden (APNODO)

{Este algoritmo realiza el recorrido inorden de un árbol binario. APNODO es un registro de tipo ENLACE —puntero a un nodo—}
{INFO, IZQ y DER son campos del registro nodo. INFO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero}

1. *Si* (APNODO ≠ NIL) *entonces*
Regresar a Inorden con APNODO^.IZQ
{Llamada recursiva a Inorden con la rama izquierda del nodo en cuestión}
Visitar el APNODO {Escribir APNODO^.INFO}
Regresar a Inorden con APNODO^.DER
{Llamada recursiva a Inorden con la rama derecha del nodo en cuestión}
2. {Fin del condicional del paso 1}

Ejemplo 6.8

En la tabla 6.4 se muestra la generación del recorrido inorden del árbol de la figura 6.20*a*, usando el algoritmo 6.3.

TABLA 6.4
Recorrido inorden

Paso	Nodo actual	Rama a visitar	Nodo visitado	Pila: rama pendiente de visitar	
1	*A*	*A*^.IZQ → *B*		*A*^.DER → *C* *A*	(1) (2)
2	*B*	*B*^.IZQ → *D*		*B*^.DER → *E* *B*	(3) (4)
3	*D*	*D*^.IZQ → NIL		*D*^.DER → NIL *D*	(5) (6)
4	NIL		*D* (6)		
5	(5) NIL		*B* (4)		
6	(3) *E*	*E*^.IZQ → NIL		*E*^.IZQ → NIL *E*	(7) (8)
7	NIL		*E* (8)		
8	(7) NIL		*A* (2)		
9	(1) *C*	*C*^.IZQ → *F*		*C*^.DER → *G* *C*	(9) (10)
10	*F*	*F*^.IZQ → NIL		*F*^.DER → NIL *F*	(11) (12)
11	NIL		*F* (12)		
12	(11) NIL		*C* (10)		
13	(9) *G*	*G*^.IZQ → NIL		*G*^.DER → NIL *G*	(13) (14)
14	NIL		*G* (14)		
15	(13) NIL				

En la columna *Pila: rama pendiente de visitar*, la llamada (*N*) indica el orden en el cual las ramas pendientes de visitar fueron introducidas a la pila. En las columnas *Nodo actual* y *Nodo visitado*, las llamadas (*N*) indican las instrucciones que se extrajeron de la pila. Note que esta observación también es válida para la tabla 6.5.

El orden en que se visitaron los nodos es:

D - B - E - A - F - C - G

Algoritmo 6.4 Posorden

Posorden (APNODO)

{Este algoritmo realiza el recorrido posorden de un árbol binario. APNODO es un dato de tipo ENLACE —puntero a un nodo—}
{INFO, IZQ y DER son campos del registro nodo. INFO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero}

```
1. Si (APNODO ≠ NIL) entonces
      Regresar a Posorden con APNODO^.IZQ
      {Llamada recursiva a Posorden con la rama izquierda del nodo en cuestión}
      Regresar a Posorden con APNODO^.DER
      {Llamada recursiva a Posorden con la rama derecha del nodo en cuestión}
      Visitar el APNODO {Escribir APNODO^.INFO}
2. {Fin del condicional del paso 1}
```

Ejemplo 6.9

En la tabla 6.5 se presentan los pasos necesarios que se siguieron para efectuar el recorrido posorden del árbol de la figura 6.20*a*, aplicando el algoritmo anterior.

TABLA 6.5
Recorrido posorden

Paso	Nodo actual	Rama a visitar	Nodo visitado	Pila: rama pendiente de visitar
1	*A*	*A*^.IZQ → *B*		*A* (1)
				A^.DER → *C* (2)
2	*B*	*B*^.IZQ → *D*		*B* (3)
				B^.DER → *E* (4)
3	*D*	*D*^.IZQ → NIL		*D* (5)
				D^.DER → NIL (6)
4	NIL			
5	(6) NIL		*D* (5)	
6	(4) *E*	*E*^.IZQ → NIL		*E* (7)
				E^.DER → NIL (8)
7	NIL			
8	(8) NIL		*E* (7)	
9			*B* (3)	
10	(2) *C*	*C*^.IZQ → *F*		*C* (9)
				C^.DER → *G* (10)
11	*F*	*F*^.IZQ → NIL		*F* (11)
				F^.DER → NIL (12)
12	NIL			
13	(12) NIL		*F* (11)	
14	(10) *G*	*G*^.IZQ → NIL		*G* (13)
				G^.DER → NIL (14)
15	NIL			
16	(14) NIL		*G* (13)	
17			*C* (9)	
18			*A* (1)	

6.3.7 Árboles binarios de búsqueda

En esta sección se presenta un tipo especial de árboles binarios. Su principal característica es que la información se almacena en los nodos cuidando de mantener cierto orden.

Formalmente se define un **árbol binario de búsqueda** de la siguiente manera: **Para todo nodo *T* del árbol se debe cumplir que todos los valores almacenados en el subárbol izquierdo de *T* sean menores o iguales a la información guardada en el nodo *T*. De forma similar, todos los valores almacenados en el subárbol derecho de *T* deben ser mayores o iguales a la información guardada en el nodo *T*.** Los valores a los que se hace referencia en la definición refieren al contenido del campo de información del nodo. Por ejemplo, si en el árbol se almacena información de los empleados de una empresa, los valores utilizados para generar el árbol de manera ordenada corresponderán al número de cada empleado —campo clave—.

El árbol binario de búsqueda es una estructura de datos obre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación. Comparando esta estructura con otras, se pueden observar ciertas ventajas. Por ejemplo, en un arreglo es posible localizar datos eficientemente si éstos se encuentran ordenados, pero las operaciones de inserción y eliminación resultan costosas, porque involucran movimiento de los elementos dentro del arreglo. En las listas, por otra parte, dichas operaciones se pueden llevar a cabo con facilidad, pero la operación de búsqueda, en este caso, es una operación que demanda recursos, pudiendo incluso requerir recorrer todos los elementos de ella para llegar a uno en particular.

Ejemplo 6.10

La figura 6.21 contiene un árbol binario de búsqueda.

Observe el lector que si en dicho árbol se sustituyen los valores 140 por 160, 99 por 105 y 43 por 55, el árbol continúa siendo un árbol binario de búsqueda.

Pero por otra parte, si en dicho árbol se reemplaza el valor 87 por 125, entonces el árbol deja de ser un árbol binario de búsqueda, puesto que viola el principio que dice: "Todos los nodos del subárbol izquierdo del nodo *T* deben almacenar valores menores o iguales al nodo´´—en este caso 125 no es menor a 120—.

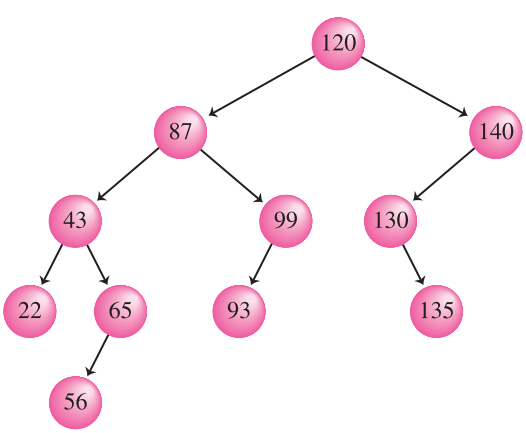


FIGURA 6.21
Árbol binario de búsqueda.

También es posible observar que si se efectúa un recorrido inorden sobre un árbol de búsqueda se obtendrá una clasificación de los nodos en forma ascendente. El recorrido inorden del árbol de la figura 6.21 produce el siguiente resultado:

22 - 43 - 56 - 65 - 87 - 93 - 99 - 120 - 130 - 135 - 140

Búsqueda

La operación de **búsqueda en un árbol binario de búsqueda** es mucho más eficiente que en un árbol binario general, ya que al comparar el valor buscado con la información del nodo visitado, si no es igual, se deberá continuar sólo por alguno de los dos subárboles. Por ejemplo, si se compara el valor buscado 90 con el valor del nodo visitado 81, la búsqueda sólo debe continuar por el camino de la derecha. El camino de la izquierda se desecha, porque contiene nodos cuyos valores serán menores o iguales a 85.

Algoritmo 6.5 Búsqueda_ABB

Búsqueda_ABB (APNODO, INFOR)

{Este algoritmo localiza el nodo del árbol binario de búsqueda que contiene la información, INFOR, que estamos buscando. APNODO es un parámetro de tipo ENLACE —la primera vez apunta a la raíz del árbol—. Se asume que el árbol no es vacío}

```
1. Si (INFOR < APNODO^.INFO)
     entonces
   1.1 Si (APNODO^.IZQ = NIL)
         entonces
             Escribir "La información no se encuentra en el árbol"
         si no
             Regresar a Búsqueda_ABB con APNODO^.IZQ e INFOR
             {Llamada recursiva}
   1.2 {Fin del condicional del paso 1.1}
     si no
   1.3 Si (INFOR > APNODO^.INFO)
           entonces
     1.3.1 Si (APNODO^.DER = NIL)
                 entonces
                     Escribir "La información no se encuentra en el árbol"
                 si no
                     Regresar a Búsqueda_ABB con APNODO^.DER e INFOR
                     {Llamada recursiva}
     1.3.2 {Fin del condicional del paso 1.3.1}
         si no
             Escribir "La información está en el árbol"
   1.4 {Fin del condicional del paso 1.3}
2. {Fin del condicional del paso 1}
```

Analice el algoritmo de búsqueda con el siguiente ejemplo.

Ejemplo 6.11

Supongamos que se desea localizar las claves 93 y 123 en el árbol binario de búsqueda de la figura 6.21. En las tablas 6.6 y 6.7 se presentan los pasos (*P*), número de comparaciones (*C*) y preguntas y acciones necesarias para localizar las claves 93 y 123, respectivamente.

TABLA 6.6
Localización de la clave 93 (INFOR ← 93)

P	*C*	Preguntas y acciones
1	1	¿Es 93 < 120?
		Sí. ¿Es el subárbol izquierdo de 120 (87) = NIL?
	2	No. Entonces se regresa a Búsqueda con el subárbol izquierdo de 120 (87) e INFOR
	3	¿Es 93 < 87?
2	4	No. ¿Es 93 > 87?
		Sí. ¿Es el subárbol derecho de 87 (99) = NIL?
	5	No. Entonces se regresa a Búsqueda con el subárbol derecho de 87 (99) e INFOR
	6	¿Es 93 < 99?
3		Sí. ¿Es el subárbol izquierdo de 99 (93) = NIL?
	7	No. Entonces se regresa a Búsqueda con el subárbol izquierdo de 99 (93) e INFOR
	8	¿Es 93 < 93?
4	9	No. ¿Es 93 > 93?
		No. Entonces ÉXITO

TABLA 6.7
Localización de la clave 123 (INFOR ← 123)

P	*C*	Preguntas y acciones
	1	¿Es 123 < 120?
1	2	No. ¿Es 123 > 120?
		Sí. ¿Es el subárbol derecho de 120 (140) = NIL?
	3	No. Entonces se regresa a Búsqueda con el subárbol derecho de 120 (140) e INFOR
	4	¿Es 123 < 140?
2		Sí. ¿Es el subárbol izquierdo de 140 (130) = NIL?
	5	No. Entonces se regresa a Búsqueda con el subárbol izquierdo de 140 (130) e INFOR
	6	¿Es 123 < 130?
3	7	Sí. ¿Es el subárbol izquierdo de 130 (NIL) = NIL?
		Sí. Entonces FRACASO

Otra forma de escribir el algoritmo de búsqueda presentado anteriormente es la que se muestra a continuación. En esta variante se contempla el caso de que el árbol esté vacío.

Algoritmo 6.6 Búsqueda_v1_ABB

Búsqueda_v1_ABB (APNODO, INFOR)

{El algoritmo localiza el nodo del árbol binario de búsqueda que contiene cierta información —INFOR—. Parámetro de tipo entero. APNODO es una variable de tipo ENLACE. La primera vez, apunta a la raíz del árbol}

```
1. Si (APNODO ≠ NIL)
     entonces
   1.1 Si (INFOR < APNODO^.INFO)
         entonces
           Regresar a Búsqueda_v1_ABB con APNODO^.IZQ e INFOR
           {Llamada recursiva}
         si no
     1.1.1 Si (INFOR > NODO^.INFO)
           entonces
             Regresar a Búsqueda_v1_ABB con APNODO^.DER e INFOR
             {Llamada recursiva}
           si no
             Escribir "La información se encuentra en el árbol"
     1.1.2 {Fin del condicional del paso 1.1.1}
   1.2 {Fin del condicional del paso 1.1}
     si no
       Escribir "La información no se encuentra en el árbol"
2. {Fin del condicional del paso 1}
```

Inserción en un árbol binario de búsqueda

La **inserción en un árbol binario de búsqueda** es una operación que se puede realizar eficientemente en este tipo de estructura de datos. La estructura crece conforme se insertan elementos al árbol. Los pasos que se deben realizar para agregar un nuevo nodo a un árbol binario de búsqueda son los siguientes:

1. Comparar la clave a insertar con la raíz del árbol. Si es mayor, se sigue con el subárbol derecho. Si es menor, se continúa con el subárbol izquierdo.
2. Repetir sucesivamente el paso 1 hasta que se cumpla alguna de las siguientes condiciones:
   - 2.1 El subárbol derecho, o el subárbol izquierdo, es igual a vacío, en cuyo caso se procederá a insertar el elemento en el lugar que le corresponde.

2.2 La clave que se quiere insertar está en el nodo analizado, por lo tanto no se lleva a cabo la inserción. Este caso es válido sólo cuando la aplicación exige que no se repitan elementos.

Ejemplo 6.12

Supongamos que se quiere insertar las siguientes claves en un árbol binario de búsqueda que se encuentre vacío:

120 - 87 - 43 - 65 - 140 - 99 - 130 - 22 - 56

Los resultados parciales que ilustran cómo funciona el procedimiento se presentan en la figura 6.22.

FIGURA 6.22

Inserción en un árbol binario de búsqueda.

Nota: Las líneas en color indican el elemento que acaba de insertarse.

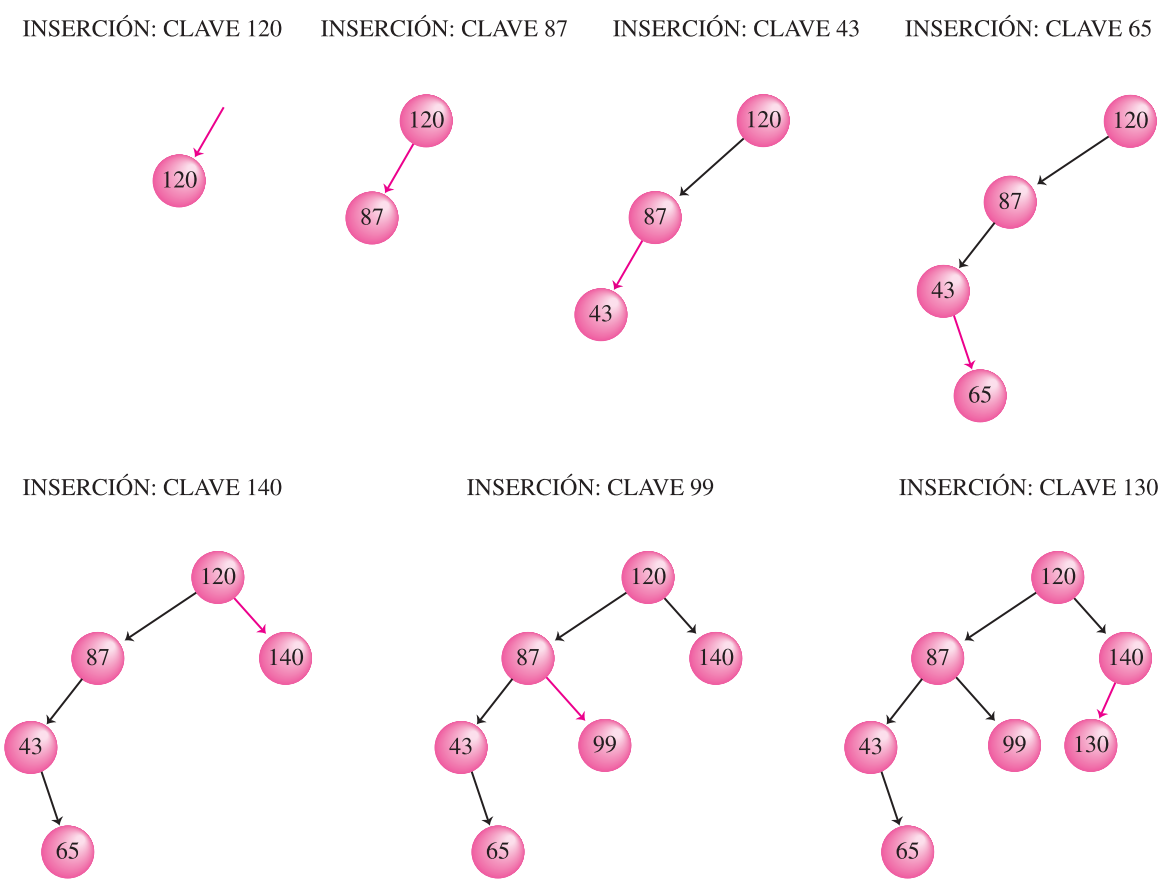
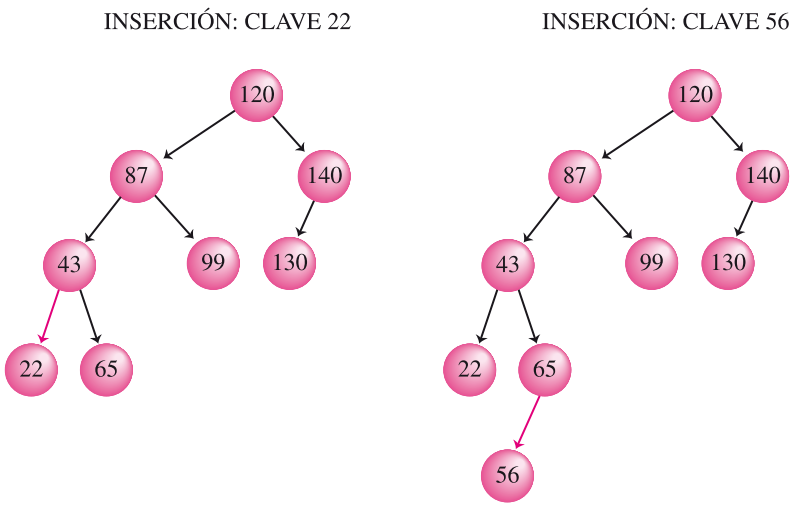


FIGURA 6.22

(*continuación*)



El algoritmo de inserción es el siguiente.

Algoritmo 6.7 Inserción_ABB

Inserción_ABB (APNODO, INFOR)

{El algoritmo realiza la inserción de un nodo en un árbol binario de búsqueda. APNODO es una variable de tipo puntero y la primera vez debe ser distinta de vacío. INFOR es un parámetro de tipo entero que contiene la información que se quiere insertar en un nodo}
{Se utiliza además, como auxiliar, la variable OTRO de tipo puntero}

```
1. Si (INFOR < APNODO^.INFO)
     entonces
  1.1 Si (APNODO^.IZQ = NIL)
        entonces
          Crear(OTRO) {Se crea un nuevo nodo}
          Hacer OTRO^.IZQ ← NIL, OTRO^.DER ← NIL, OTRO^.INFO ← INFOR
                y APNODO^.IZQ ← OTRO
        si no
          Regresar a Inserción_ABB con APNODO^.IZQ e INFOR
          {Llamada recursiva}
  1.2 {Fin del condicional del paso 1.1}
    si no
  1.3 Si (INFOR > APNODO^.INFO)
        entonces
    1.3.1 Si (APNODO^.DER = NIL)
            entonces
              Crear(OTRO) {Se crea un nuevo nodo}
              Hacer OTRO^.IZQ ← NIL, OTRO^.DER ← NIL,
                    OTRO^.INFO ← INFOR y NODO^.DER ← OTRO
```

```
                si no
                    Regresar a Inserción_ABB con NODO^.DER e INFOR
                    {Llamada recursiva}
        1.3.2 {Fin del condicional del paso 1.3.1}
            si no
                Escribir "El nodo ya se encuentra en el árbol"
    1.4 {Fin del condicional del paso 1.3}
2. {Fin del condicional del paso 1}
```

Ejemplo 6.13

Supongamos que se desea insertar las claves 93 y 135 en el árbol binario de búsqueda de la figura 6.22. En las tablas 6.8 y 6.9 se presentan los pasos (*P*), número de comparaciones (*C*) y preguntas y acciones necesarias para insertar estas claves.

TABLA 6.8
Inserción de la clave 93 (INFOR ← 93)

P	*C*	Preguntas y acciones
	1	¿Es 93 < 120?
1		Sí. ¿Es el subárbol izquierdo de 120 (87) = NIL?
	2	No. Entonces regresar a Inserción con el subárbol izquierdo de 120 (87) e Infor
	3	¿Es 93 < 87?
2	4	No. ¿Es 93 > 87?
		Sí. ¿Es el subárbol derecho de 87 (99) = NIL?
	5	No. Entonces regresar a Inserción con el subárbol derecho de 87 (99) e Infor
	6	¿Es 93 < 99?
3		Sí. ¿Es el subárbol izquierdo de 99 (NIL) = NIL?
	7	Sí. Entonces crear otro nodo, realizar los enlaces y cargar la información

TABLA 6.9
Inserción de la clave 135 (INFOR ← 135)

P	*C*	Preguntas y acciones
	1	¿Es 135 < 120?
1	2	No. ¿Es 135 > 120?
		Sí. ¿Es el subárbol derecho de 120 (140) = NIL?
	3	Sí. Entonces regresar a Inserción con el subárbol derecho de 120 (140) e Infor
	4	No. ¿Es 135 > 140?
2		Sí. ¿Es el subárbol izquierdo de 140 (130) = NIL?
	5	No. Entonces regresar a Inserción con el subárbol izquierdo de 140 (130) e Infor
	6	¿Es 135 < 130?
3	7	No. ¿Es 135 > 130?
		Sí. ¿Es el subárbol derecho de 130 (NIL) = NIL?
	8	No. Entonces crear otro nodo, realizar los enlaces y cargar la información

En la figura 6.23 se presenta el árbol binario de búsqueda luego de realizada esta operación.

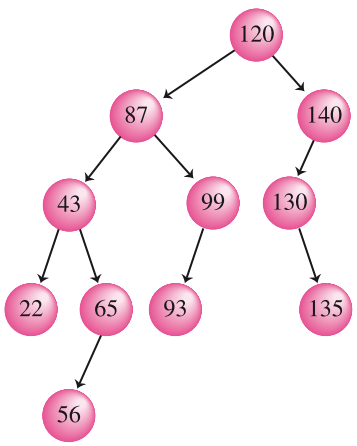


FIGURA 6.23
Árbol binario de búsqueda. Inserción de las claves 93 y 135.

Otra forma de expresar el algoritmo de inserción es la siguiente.

Algoritmo 6.8 Inserción_v1_ABB

Inserción_v1_ABB (APNODO, INFOR)

{El algoritmo realiza la inserción de un elemento en un árbol binario de búsqueda. APNODO es una variable de tipo ENLACE, la primera vez apunta a la raíz del árbol. INFOR es un parámetro de tipo entero que contiene la información del elemento que se quiere insertar. El algoritmo considera el caso de un árbol vacío}

```
1. Si (APNODO ≠ NIL)
     entonces
   1.1 Si (INFOR < APNODO^.INFO)
         entonces
            Regresar a Inserción_v1_ABB con APNODO^.IZQ e INFOR
            {Llamada recursiva}
         si no
     1.1.1 Si (INFOR > APNODO^.INFO)
            entonces
               Regresar a Inserción_v1_ABB con APNODO^.DER e INFOR
               {Llamada recursiva}
            si no
               Escribir "La información ya se encuentra en el árbol"
     1.1.2 {Fin del condicional del paso 1.1.1}
   1.2 {Fin del condicional del paso 1.1}
     si no
        Crear (OTRO) {Se crea un nuevo nodo}
        Hacer OTRO^.IZQ ← NIL, OTRO^.DER ← NIL, OTRO^.INFO ← INFOR
              y APNODO ← OTRO
2. {Fin del condicional del paso 1}
```

Eliminación en un árbol binario de búsqueda

La operación de **eliminación en un árbol binario de búsqueda** es un poco más complicada que la de inserción. Ésta consiste en eliminar un nodo sin violar los principios que definen un árbol binario de búsqueda. Se deben distinguir los siguientes casos:

1. Si el elemento a eliminar es terminal u hoja, simplemente se suprime redefiniendo el puntero de su predecesor.
2. Si el elemento a eliminar tiene un solo descendiente, entonces tiene que sustituirse por ese descendiente.
3. Si el elemento a eliminar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Cabe destacar que antes de eliminar un nodo, se debe localizar éste en el árbol. Para esto se utiliza el algoritmo de búsqueda presentado anteriormente.

Ejemplo 6.14

Supongamos que se desea eliminar las siguientes claves del árbol binario de búsqueda de la figura 6.23:

22 - 99 - 87 - 120 - 140 - 135 - 56

Los resultados parciales que ilustran cómo funciona el procedimiento se presentan en la figura 6.24.

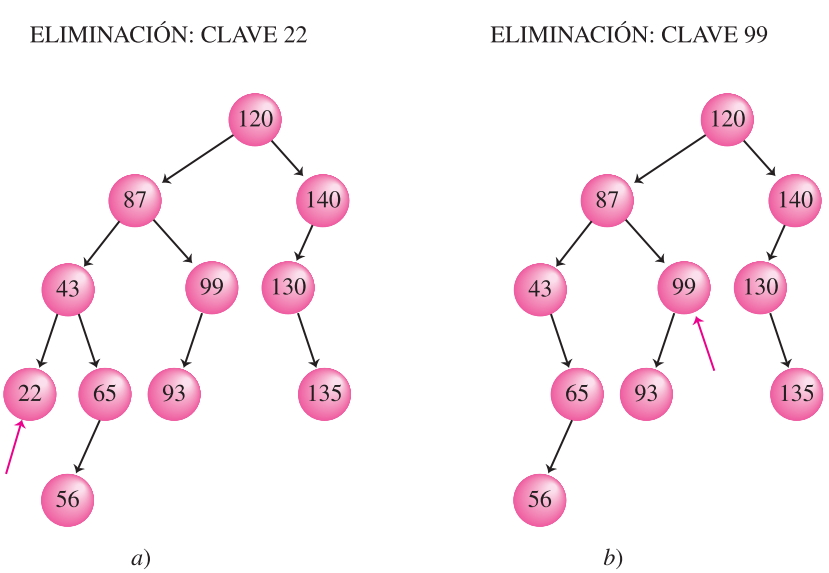


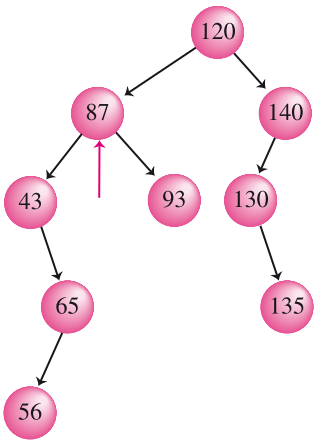
FIGURA 6.24
Eliminación en un árbol binario de búsqueda. *a*) y *f*) corresponden al primer caso; *b*) y *e*) corresponden al segundo caso; *c*) y *d*) corresponden al tercer caso. *g*) Estado final del árbol.

Nota: Las flechas en color indican el elemento que quiere eliminarse.

FIGURA 6.24

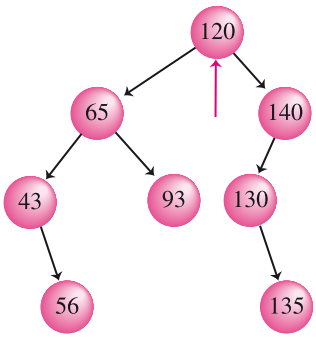
(*continuación*)

ELIMINACIÓN: CLAVE 87



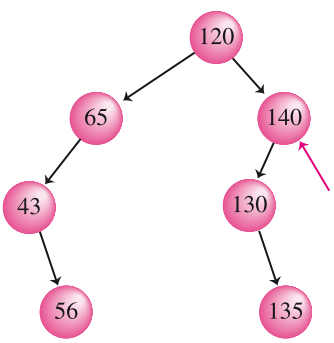
c)

ELIMINACIÓN: CLAVE 120



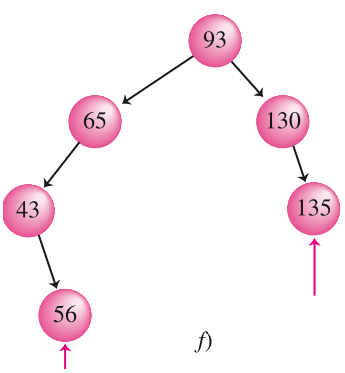
d)

ELIMINACIÓN: CLAVE 140

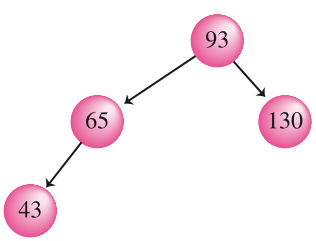


e)

ELIMINACIÓN: CLAVES 135 Y 56



f)



g)

A continuación se presenta el algoritmo correspondiente:

Algoritmo 6.9 Eliminación_ABB

Eliminación_ABB (APNODO, INFOR)

```
{El algoritmo realiza la eliminación de un elemento en un árbol binario de búsqueda. APNODO
es una variable, por referencia, de tipo ENLACE. INFOR es un parámetro de tipo entero que
contiene la información del nodo que se desea eliminar}
{AUX, AUXI y OTRO son variables auxiliares de tipo puntero. BO es una variable de tipo
booleano}

1. Si (APNODO ≠ NIL)
      entonces
      1.1 Si (INFOR < APNODO^.INFO)
            entonces
               Regresar a Eliminación_ABB con APNODO^.IZQ e INFOR
            si no
         1.1.1 Si (INFOR > APNODO^.INFO)
               entonces
                  Regresar a Eliminación_ABB con APNODO^.DER e INFOR
               si no
                  Hacer OTRO ← NODO
                     1.1.1.1 Si (OTRO^.DER = NIL)
                                 entonces
                                    Hacer APNODO ← OTRO^.IZQ
                                 sino
                           1.1.1.1.1 Si (OTRO^.IZQ = NIL)
                                       entonces
                                          Hacer APNODO ← OTRO^.DER
                                       sino
                                          Hacer AUX ← APNODO^.IZQ y BO ← FALSO
                                 1.1.1.1.1.A Mientras (AUX^.DER ≠ NIL) Repetir
                                             Hacer AUX1 ← AUX, AUX ← AUX^.DER
                                                   y BO ← VERDADERO
                                 1.1.1.1.1.B {Fin del ciclo del paso 1.1.1.1.1.A}
                                             Hacer APNODO^.INFO ← AUX^.INFO y
                                                   OTRO ← AUX
                                 1.1.1.1.1.C Si (BO = VERDADERO)
                                              entonces
                                                 Hacer AUX1^.DER ← AUX^.IZQ
                                              sino
                                                 Hacer APNODO^.IZQ ← AUX^.IZQ
                                 1.1.1.1.1.D {Fin del condicional del paso 1.1.1.1.1.C}
                           1.1.1.1.2 {Fin del condicional del paso 1.1.1.1.1}
                     1.1.1.2 {Fin del condicional del paso 1.1.1.1}
                              Quitar (OTRO) {Se libera el espacio de memoria}
               1.1.2 {Fin del condicional del paso 1.1.1}
         1.2 {Fin del condicional del paso 1.1}
          sino
             Escribir "La información a eliminar no se encuentra en el árbol"
2.    {Fin del condicional del paso 1}
```

Ejemplo 6.15

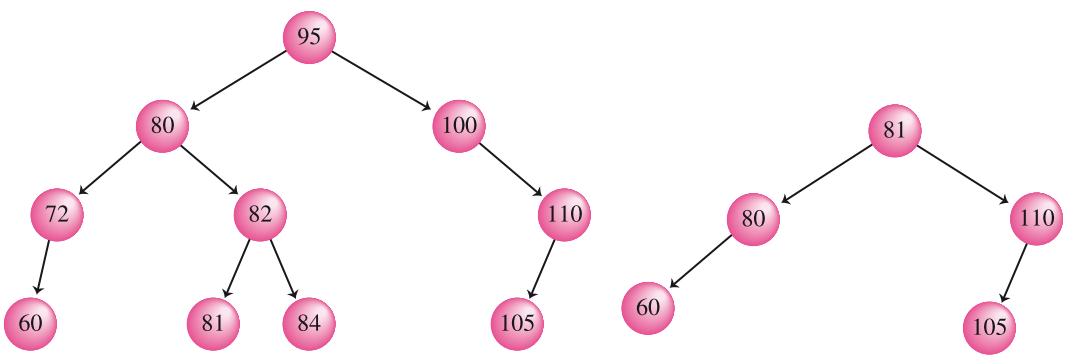
Dado el árbol de la figura 6.25*a*, verifique si queda igual al árbol de la figura 6.25*b* luego de eliminar, aplicando el algoritmo 6.9, las claves que se muestran a continuación:

95 - 72 - 84 - 100 - 82

FIGURA 6.25

Eliminación en un árbol binario de búsqueda. *a*) Antes de eliminar las claves. *b*) Después de eliminar las claves.

Nota: En el ejemplo, como el nodo a eliminar tiene dos descendientes, se sustituye por el nodo que se encuentra más a la derecha en el subárbol izquierdo.



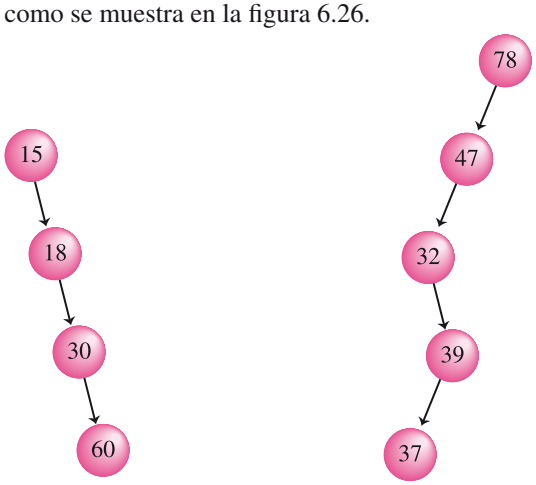
a) *b*)

6.4 ÁRBOLES BALANCEADOS

Cuando se estudiaron los árboles binarios de búsqueda se mencionó que es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación. Sin embargo, si el árbol crece o decrece descontroladamente, el rendimiento puede disminuir considerablemente. El caso más desfavorable se produce cuando se inserta un conjunto de claves ordenadas en forma ascendente o descendente, como se muestra en la figura 6.26.

FIGURA 6.26

Árboles binarios de búsqueda con crecimiento descontrolado.



Es de notar que el número promedio de comparaciones que se deben realizar para localizar una determinada clave en un árbol binario de búsqueda con crecimiento descontrolado es $N/2$, cifra que muestra un rendimiento muy pobre en la estructura.

Con el objeto de mantener la eficiencia en la operación de búsqueda surgen los **árboles balanceados**. La principal característica de éstos es la de realizar reacomodos o balanceos, después de inserciones o eliminaciones de elementos. Estos árboles también reciben el nombre de **AVL** en honor a sus inventores, dos matemáticos rusos, G. M. **A**delson-**V**elskii y E. M. Landis.

Formalmente se define un árbol balanceado como un árbol binario de búsqueda, en el cual se debe cumplir la siguiente condición: Para todo nodo *T* del árbol, la altura de los subárboles izquierdo y derecho no deben diferir en más de una unidad.

$$|H_{RI} - H_{RD}| \leq 1$$

donde H_{RI} es la altura de la rama o subárbol izquierdo y H_{RD} es la altura de la rama o subárbol derecho.

En la figura 6.27 se muestran dos ejemplos de árboles balanceados.

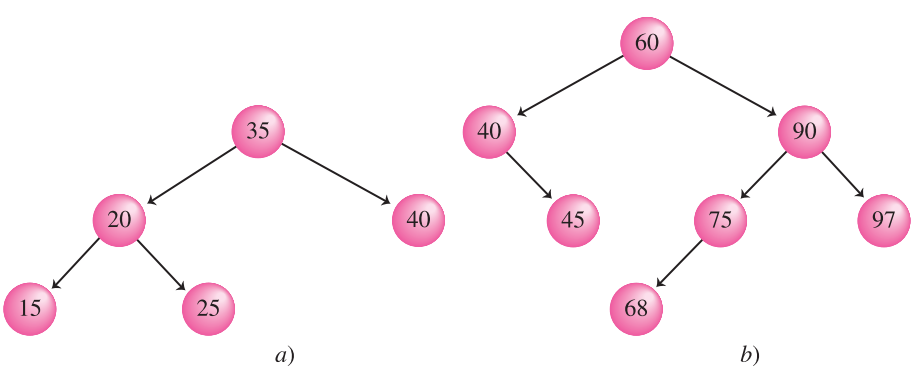


FIGURA 6.27
Dos árboles balanceados. *a*) Con altura 3. *b*) Con altura 4.

Observe el lector que si se insertan las claves 10, 18 o 27 en el árbol balanceado de la figura 6.27*a*, éste pierde el equilibrio. Sin embargo, si se insertan las claves 37 o 45 en el mismo árbol, éste mantiene el equilibrio; más aún, lo mejora.

Ahora bien, con respecto a la eliminación, si a dicho árbol se le quitan las claves 15, 20 o 25 el árbol continúa siendo balanceado; incluso podrían quitársele las tres claves y el árbol no perdería el equilibrio. Sin embargo, si se elimina la clave 40 el árbol pierde el equilibrio y es necesario reestructurarlo.

Los árboles balanceados se parecen mucho, en su mecanismo de formación, a los números Fibonacci. El árbol de altura 0 es vacío, el árbol de altura 1 tiene un único nodo y, en general, el número de nodos del árbol con altura $h > 1$ se calcula aplicando la siguiente fórmula recursiva:

$$K_h = K_{h-1} + 1 + K_{h-2}$$ **Fórmula 6.6**

donde K indica el número de nodos del árbol y h la altura.

Por otra parte, algunos estudios demuestran que la altura de un árbol balanceado de n nodos nunca excederá de 1.44 * log n.

Ejemplo 6.16

Supongamos que se desea calcular el número de nodos de un árbol balanceado con altura 5. La forma en que se efectúa el cálculo es la siguiente:

$$\begin{aligned}
K_5 &= K_4 + 1 + K_3 \\
K_4 &= K_3 + 1 + K_2 \\
K_3 &= K_2 + 1 + K_1 \\
K_2 &= K_1 + 1 + K_0 \\
K_1 &= 1 \\
K_0 &= 0 \\
K_2 &= 2 \\
K_3 &= 4 \\
K_4 &= 7 \\
K_5 &= 12
\end{aligned}$$

6.4.1 Inserción en árboles balanceados

Al insertar un elemento en un árbol balanceado se deben distinguir los siguientes casos:

1. Las ramas izquierda (*RI*) y derecha (*RD*) del árbol tienen la misma altura ($H_{RI} = H_{RD}$), por lo tanto:
- **1.1** Si se inserta un elemento en *RI*, entonces H_{RI} será mayor en una unidad a H_{RD}.
- **1.2** Si se inserta un elemento en *RD*, entonces H_{RD} será mayor en una unidad a H_{RI}.

Observe que en cualquiera de los dos casos mencionados (1.1 y 1.2), no se viola el criterio de equilibrio del árbol.

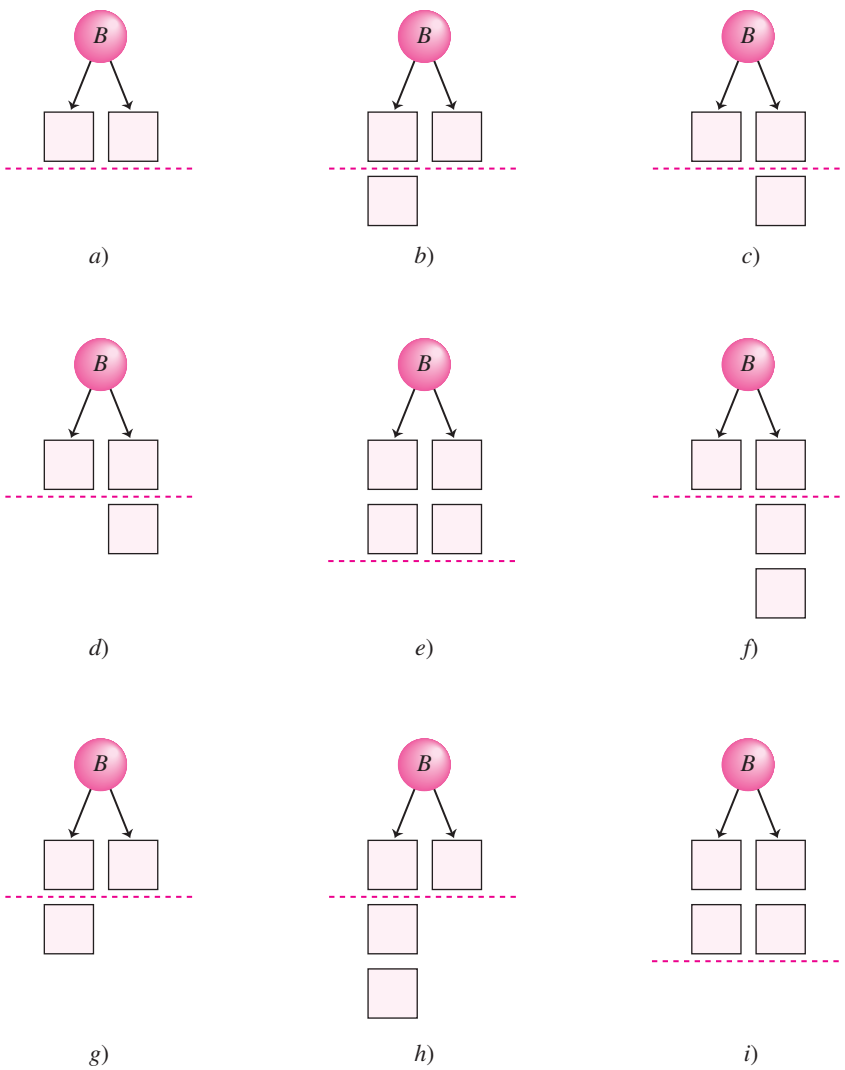
2. Las ramas izquierda (*RI*) y derecha (*RD*) del árbol tienen altura diferente ($H_{RI} \neq H_{RD}$):
- **2.1** Supongamos que $H_{RI} < H_{RD}$:
  - **2.1.1** Si se inserta un elemento en *RI*, entonces H_{RI} será igual a H_{RD}. {Las ramas tienen la misma altura, por lo que se mejora el equilibrio del árbol}
  - **2.1.2** Si se inserta un elemento en *RD*, entonces se rompe el criterio de equilibrio del árbol y es necesario reestructurarlo.
- **2.2** Supongamos que $H_{RI} > H_{RD}$:
  - **2.2.1** Si se inserta un elemento en *RI*, entonces se rompe el criterio de equilibrio del árbol y es necesario reestructurarlo.
  - **2.2.2** Si se inserta un elemento en *RD*, entonces H_{RD} será igual a H_{RI}. {Las ramas tienen la misma altura, por lo que se mejora el equilibrio del árbol}

En la figura 6.28 se muestran diagramas de los distintos casos que se presentan en la operación de inserción en árboles balanceados.

FIGURA 6.28

Diferentes casos de inserción en árboles balanceados. *a*) Caso 1. *b*) Caso 1.1. *c*) Caso 1.2. *d*) Caso 2.1. *e*) Caso 2.1.1. *f*) Caso 2.1.2. *g*) Caso 2.2. *h*) Caso 2.2.1. *i*) Caso 2.2.2.

Nota: La línea discontinua marca el equilibrio perfecto del árbol.

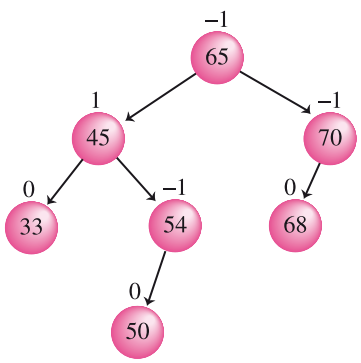


Ahora bien, para poder determinar si un árbol está balanceado o no, se debe manejar información relativa al equilibrio de cada nodo del árbol. Surge así el concepto de **factor de equilibrio** de un nodo (***FE***) que se define como la altura del subárbol derecho menos la altura del subárbol izquierdo.

$$FE = H_{RD} - H_{RI}$$ **Fórmula 6.7**

Los valores que puede tomar *FE* son –1, 0, 1. Si *FE* llegara a tomar los valores de –2 o 2, entonces debería reestructurarse el árbol. En la figura 6.29 se presenta un árbol balanceado con el correspondiente *FE* para cada nodo del árbol.

FIGURA 6.29
Árbol balanceado con el correspondiente *FE*.



Observe el lector que en la figura 6.29 el *FE* del nodo que almacena el 65 es –1, puesto que la altura del subárbol derecho es igual a 2 y la altura del subárbol izquierdo igual a 3.

$$FE_{65} = 2 - 3 = -1$$

El *FE* de 45 se calcula como:

$$FE_{45} = 2 - 1 = 1$$

A continuación se presenta la definición de un árbol balanceado en lenguaje algorítmico.

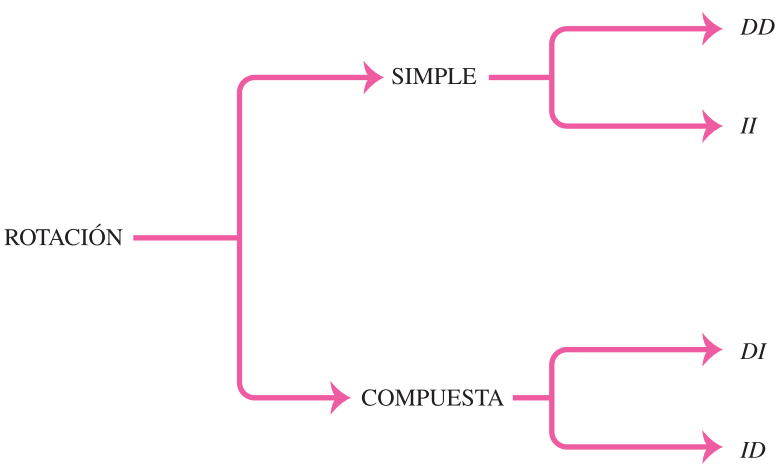
```
ENLACE = ^NODO
NODO   = REGISTRO
           IZQ, DER: tipo ENLACE
           INFO: tipo de dato
           FE: –1..1
{Fin de la definición}
```

6.4.2 Reestructuración del árbol balanceado

El proceso de inserción en un árbol balanceado es sencillo, sin embargo requiere de operaciones auxiliares que complican parcialmente el proceso. Primero se debe seguir el camino de búsqueda del árbol, hasta localizar el lugar donde hay que insertar el elemento. Luego se calcula su *FE*, que obviamente será 0, y regresamos por el camino de búsqueda calculando el *FE* de los distintos nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio entonces se debe reestructurar el árbol. El proceso termina

al llegar a la raíz del árbol, o cuando se realiza la reestructuración del mismo, en cuyo caso no es necesario determinar el *FE* de los nodos restantes.

Reestructurar el árbol significa rotar los nodos del mismo para llevarlo a un estado de equilibrio. La rotación puede ser simple o compuesta. El primer caso involucra dos nodos y el segundo caso afecta a tres. Si la rotación es simple se puede realizar por las ramas derechas (***DD***) o por las ramas izquierdas (***II***). Si por otra parte la rotación es compuesta se puede realizar por las ramas derecha e izquierda (***DI***) o por las ramas izquierda y derecha (***ID***).



Ejemplo 6.17

En la figura 6.30 se pueden observar gráficamente los diferentes tipos de rotaciones. En la figura 6.30*a* se presenta la rotación *II*, en la figura 6.30*b* la rotación *DD*, en la figura 6.30*c* la rotación *DI* y en la figura 6.30*d* la rotación *ID*.

La línea continua (_____) marca el estado de los nodos del árbol antes de realizar la inserción. La línea discontinua (_ _ _ _) indica el nuevo elemento insertado. La línea con puntos (.....) marca el camino de regreso hasta que se detecta el desequilibrio del árbol. La línea gruesa (_____) indica el movimiento de los nodos en la rotación.

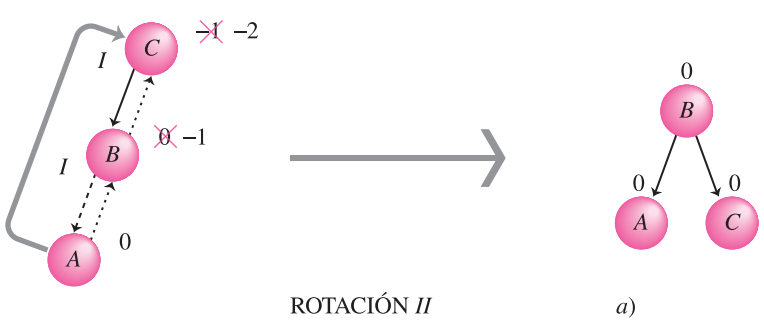
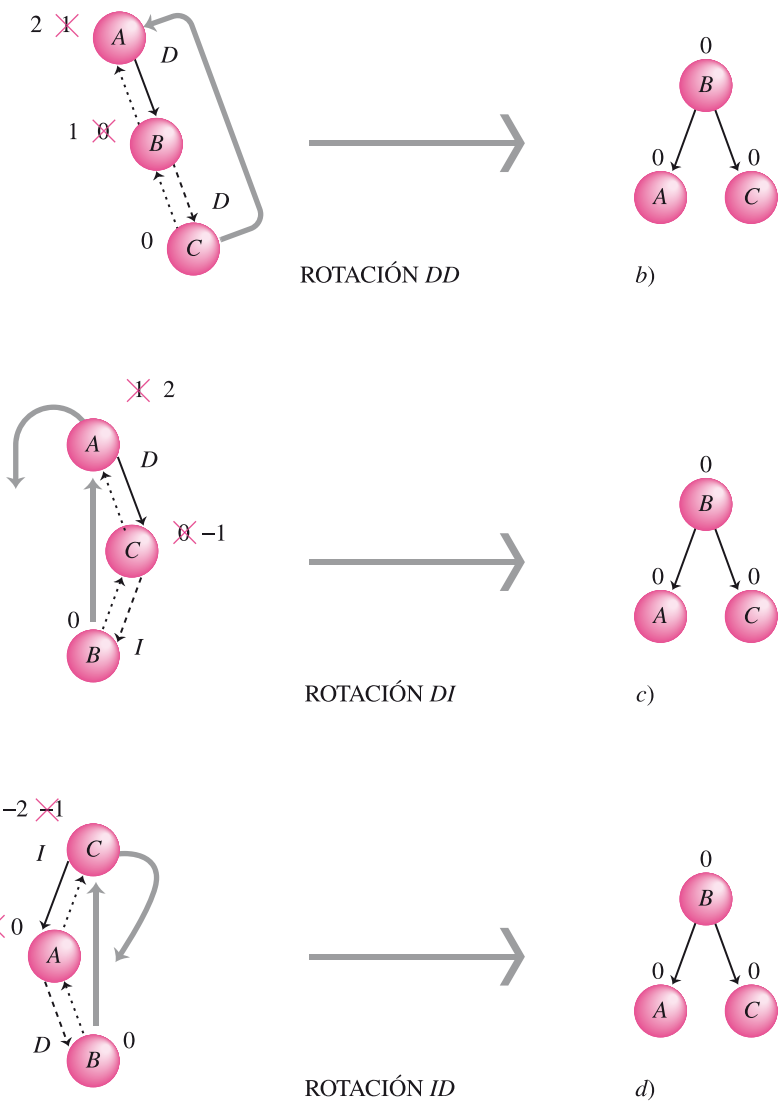


FIGURA 6.30
Rotaciones en árboles balanceados. *a*) Rotación *II*. *b*) Rotación *DD*. *c*) Rotación *DI*. *d*) Rotación *ID*.

FIGURA 6.30
(*continuación*)



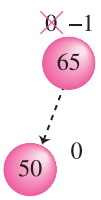
Ejemplo 6.18 Supongamos que se desea insertar las siguientes claves en un árbol balanceado que se encuentra vacío:

65 - 50 - 23 - 70 - 82 - 68 - 39

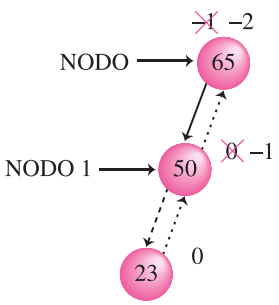
Las operaciones necesarias son las siguientes:

a) INSERCIÓN: CLAVE 65

b) INSERCIÓN: CLAVE 50



c) INSERCIÓN: CLAVE 23



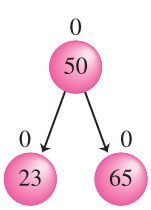
Al regresar, luego de insertar un nodo con el valor 23, siguiendo el camino de búsqueda se detecta que en la clave 65 se viola el criterio de equilibrio del árbol y se debe reestructurar. Se apunta con NODO la clave 65 y con NODO1 la rama izquierda de dicha clave. Luego se verifica el *FE* de NODO1; como en este caso es igual a –1 se puede realizar la rotación *II*. El movimiento de apuntadores para realizar la rotación *II* es el siguiente:

```
NODO^.IZQ   ← NODO1^.DER
NODO1^.DER  ← NODO
NODO        ← NODO1
```

Respecto al *FE* de los nodos afectados, éste será siempre 0 en el caso de rotaciones simples.

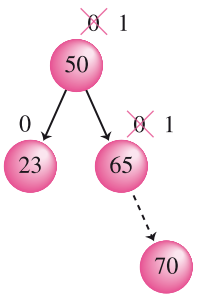
```
NODO^.FE    ← 0
NODO1^.FE   ← 0
```

Luego de efectuar el reacomodo, el árbol queda así:

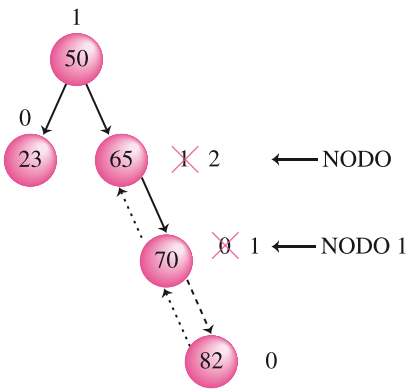


Al regresar siguiendo el camino de búsqueda se modifica el *FE* de los nodos 65 y 50, pero el equilibrio del árbol se mantiene.

d) INSERCIÓN: CLAVE 70



e) INSERCIÓN: CLAVE 82



ROTACIÓN *DD*

Al regresar, luego de insertar el valor 82, siguiendo el camino de búsqueda se observa una violación al criterio de equilibrio del árbol y se debe reestructurar. Se apunta con NODO la clave 65 y con NODO1 la rama derecha de dicha clave. Se verifica el *FE* de NODO1 y como en este caso es igual a 1 se puede realizar la rotación *DD*. El movimiento de apuntadores para realizar la rotación *DD* es:

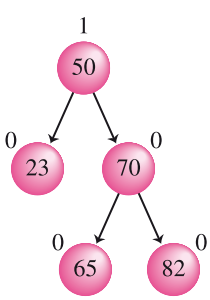
```
NODO^.DER   ← NODO1^.IZQ
NODO1^.IZQ  ← NODO
NODO        ← NODO1
```

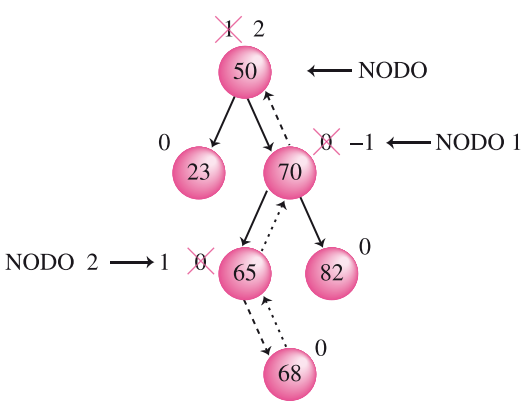
Respecto al *FE* de los nodos afectados, las asignaciones son las siguientes:

```
NODO^.FE    ← 0
NODO1^.FE   ← 0
```

Luego de volver a equilibrarlo, el árbol queda de esta forma:

f) INSERCIÓN: CLAVE 68





ROTACIÓN *DI*

Luego de insertar la clave 68 y al regresar siguiendo el camino de búsqueda se advierte que en la clave 50 se rompe el equilibrio del árbol. Se apunta con NODO la clave 50 y con NODO1 su rama derecha. Se calcula el *FE* de NODO1 y como en este caso es igual a –1, se realiza la rotación *DI.* Se apunta entonces con NODO2 la rama izquierda de NODO1. El movimiento de apuntadores para realizar la rotación *DI* es:

```
NODO1^.IZQ  ← NODO2^.DER
NODO2^.DER  ← NODO1
NODO^.DER   ← NODO2^.IZQ
NODO2^.IZQ  ← NODO
NODO        ← NODO2
```

El *FE* de los nodos involucrados se asigna de acuerdo con los valores establecidos en la tabla 6.10.

TABLA 6.10
Factores de equilibrio en la rotación *DI*

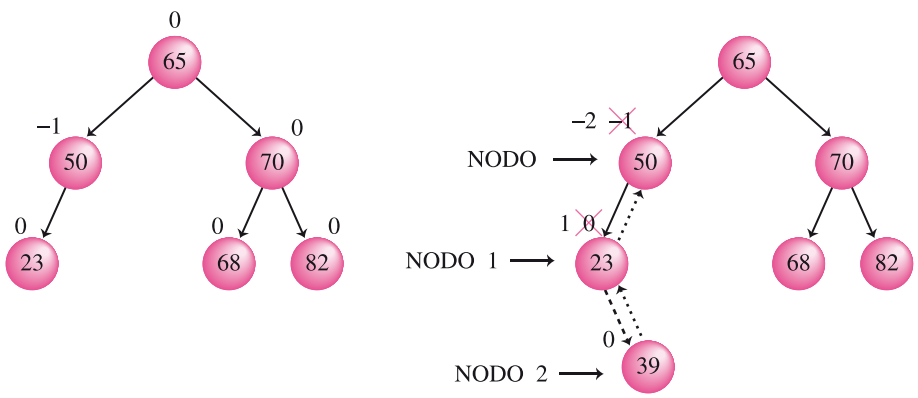
NODO2^.*FE* = –1	NODO^.*FE* = 0 NODO1^.*FE* = 1 NODO2^.*FE* = 0
NODO2^.*FE* = 0	NODO^.*FE* = 0 NODO1^.*FE* = 0 NODO2^.*FE* = 0
NODO2^.*FE* = 1	NODO^.*FE* = –1 NODO1^.*FE* = 0 NODO2^.*FE* = 0

Como en el ejemplo presentado el *FE* de NODO2 es igual a 1, se realizan las siguientes asignaciones:

```
NODO^.FE   ← –1
NODO1^.FE  ← 0
NODO2^.FE  ← 0
```

Luego de realizar el reacomodo, el árbol queda de la siguiente manera:

g) INSERCIÓN: CLAVE 39



ROTACIÓN *ID*

Regresando por el camino de búsqueda luego de insertar la clave 39, es evidente que en la clave 50 se rompe el equilibrio del árbol. Se apunta con NODO la clave 50 y con NODO1 su rama izquierda. Se verifica el *FE* de NODO1 y como en este caso es igual a 1, se realiza la rotación *ID*. Se apunta con NODO2 la rama derecha de NODO1. El movimiento de apuntadores para realizar la rotación *ID* es:

```
NODO1^.DER ← NODO2^.IZQ
NODO2^.IZQ ← NODO1
NODO^.IZQ  ← NODO2^.DER
NODO2^.DER ← NODO
NODO       ← NODO2
```

El *FE* de los nodos involucrados se asigna de acuerdo con los valores establecidos en la tabla 6.11.

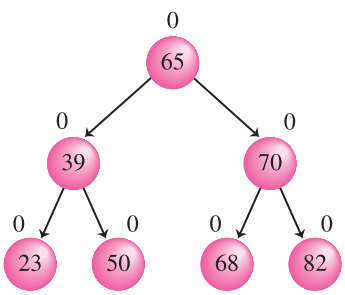
TABLA 6.11
Factores de equilibrio en la rotación *ID*

NODO2^.*FE* = –1	NODO^.*FE* = 1 NODO1^.*FE* = 0 NODO2^.*FE* = 0
NODO2^.*FE* = 0	NODO^.*FE* = 0 NODO1^.*FE* = 0 NODO2^.*FE* = 0
NODO2^.*FE* = 1	NODO^.*FE* = 0 NODO1^.*FE* = –1 NODO2^.*FE* = 0

Como en el ejemplo presentado el *FE* de NODO2 es igual a 0, se realizan las siguientes asignaciones:

NODO^.*FE* ← 0
NODO1^.*FE* ← 0
NODO2^.*FE* ← 0

Luego de volver a equilibrarlo, el árbol queda de la siguiente forma:



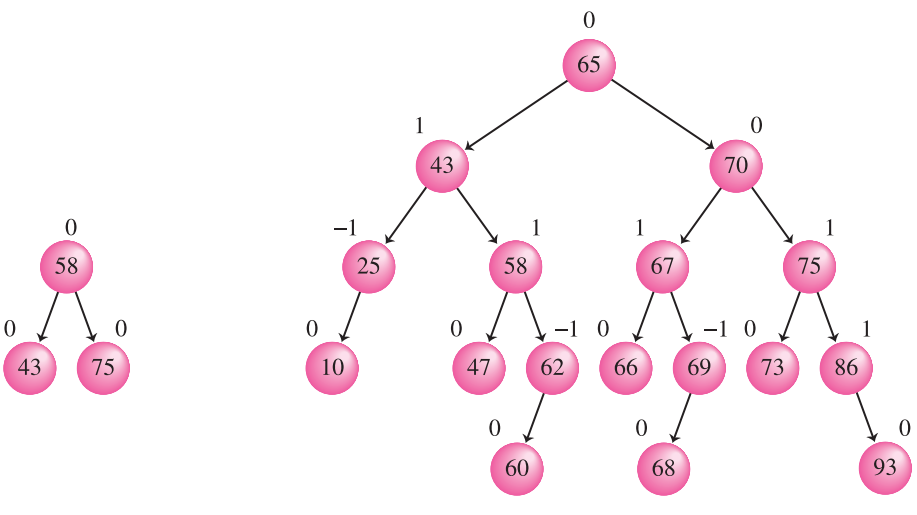
Nota: Observe que luego de realizar la inserción de un elemento y cuando se regresa por el camino de búsqueda, el *FE* del nodo visitado se incrementa en 1 si la inserción se hizo por su rama derecha y disminuye en 1 si la inserción se hizo por su rama izquierda.

Ejemplo 6.19

Dado como dato el árbol balanceado de la figura 6.31*a*, verifique si el mismo queda igual al de la figura 6.31*b* luego de insertar las siguientes claves:

86 - 65 - 70 – 67 - 73 - 93 - 69 - 25 - 66 - 68 - 47 - 62 - 10 - 60

FIGURA 6.31
Inserción en árboles balanceados. *a*) Antes de insertar las claves. *b*) Después de insertar las claves.



A continuación se presenta el algoritmo de inserción en árboles balanceados.

Algoritmo 6.10 Inserta_balanceado

Inserta_balanceado (NODO, BO, INFOR)

```
{El algoritmo inserta un elemento en un árbol balanceado. NODO es un parámetro de tipo
puntero, por referencia. BO es un parámetro de tipo booleano, por referencia. BO se utiliza para
indicar que la altura del árbol ha crecido, su valor inicial es FALSO. INFOR es un parámetro
de tipo entero que contiene la información del elemento que queremos insertar}
{OTRO, NODO1 y NODO2 son variables auxiliares de tipo puntero}
1. Si (NODO ≠ NIL)
     entonces
  1.1 Si (INFOR < NODO^.INFO)
        entonces
          Regresar a Inserta_balanceado con NODO^.IZQ, BO e INFOR
          {Llamada recursiva}
    1.1.1 Si (BO = VERDADERO)
            entonces
      1.1.1.1 Si (NODO^.FE)
                = 1: Hacer NODO^.FE ← 0 y BO ← FALSO
                = 0: Hacer NODO^.FE ← –1
                = –1: Hacer NODO1 ← NODO^.IZQ
                {Reestructuración del árbol}
        1.1.1.1.1 Si (NODO1^.FE ≤ 0)
                    entonces {Rotación II}
                      Hacer NODO^.IZQ ← NODO1^.DER,
                            NODO1^.DER, ← NODO,
                            NODO^.FE ← 0 y NODO ← NODO1
                      {Termina la rotación II}
                    si no {Rotación ID}
                      Hacer NODO2 ← NODO1^.DER,
                            NODO^.IZQ ← NODO2^.DER,
                            NODO2^.DER ← NODO,
                            NODO1^.DER ← NODO2^.IZQ y
                            NODO2^.IZQ ← NODO1
          1.1.1.1.1.A Si (NODO2^.FE = –1)
                        entonces
                          Hacer NODO^.FE ← 1
                        si no
                          Hacer NODO^.FE ← 0
          1.1.1.1.1.B {Fin del condicional del paso 1.1.1.1.1.A}
          1.1.1.1.1.C Si (NODO2^.FE = 1)
                        entonces
                          Hacer NODO1^.FE ← –1
                        si no
                          Hacer NODO1^.FE ← 0
          1.1.1.1.1.D {Fin del condicional del paso 1.1.1.1.1.C}
                      Hacer NODO ← NODO2
                      {Termina la rotación ID}
```

```
            1.1.1.1.2  {Fin del condicional del paso 1.1.1.1.1}
                       Hacer NODO^.FE ← 0 y BO ← FALSO
          1.1.1.2  {Fin del condicional del paso 1.1.1.1}
        1.1.2  {Fin del condicional del paso 1.1.1}
si no
        1.1.3  Si (INFOR > NODO^.INFO)
                 entonces
                   Regresar a Inserta_balanceado con NODO^.DER, BO e INFOR
                   {Llamada recursiva}
               1.1.3.1  Si (BO = VERDADERO)
                          entonces
               1.1.3.1.1  Si (NODO^.FE)

                                = –1: Hacer NODO^.FE ← 0 y BO ← FALSO
                                = 0: Hacer NODO1^.FE ← 1
                                = 1: Hacer NODO1 ← NODO^.DER
                                {Reestructuración de árbol}
                     1.1.3.1.1.A  Si (NODO1^.FE ≥ 0)
                                    entonces {Rotación DD}
                                      Hacer NODO^.DER ← NODO1^.IZQ,
                                            NODO1^.IZQ ← NODO,
                                            NODO^.FE ← 0 y NODO ← NODO1
                                      {Termina la rotación DD}
                                    si no {Rotación DI}
                                      Hacer NODO2 ← NODO1^.IZQ,
                                            NODO^.DER ← NODO2^.IZQ
                                            NODO2^.IZQ ← NODO,
                                            NODO1^.IZQ ← NODO2^.DER y
                                            NODO2^.DER ← NODO1
                                      Si (NODO2^.FE = 1)
                                        entonces
                                          Hacer NODO^.FE ← –1
                                        si no
                                          Hacer NODO^.FE ← 0
                                      {Fin del condicional interno}
                                      Si (NODO2^.FE = –1)
                                        entonces
                                          Hacer NODO1^.FE ← 1
                                        si no
                                          Hacer NODO1^.FE ← 0
                                        {Fin del condicional interno}
                                        Hacer NODO ← NODO2
                                        {Termina la rotación DI}
                       1.1.3.1.1.B  {Fin del condicional del paso 1.1.3.1.1.A}
                                    Hacer NODO^.FE ← 0 y BO ← FALSO
                     1.1.3.1.2  {Fin del condicional del paso 1.1.3.1.1}
                   1.1.3.2  {Fin del condicional del paso 1.1.3.1}
                 si no
                   Escribir "La información ya se encuentra en el árbol"
        1.1.4  {Fin del condicional del paso 1.1.3}
```

```
    1.2 {Fin del condicional del paso 1.1}
     si no
         Crear (NODO)
         Hacer NODO^.INFO ← INFOR, NODO^.IZQ ← NIL, NODO^.DER ← NIL,
               NODO^.FE ← 0 y BO ← VERDADERO
2. {Fin del condicional del paso 1}
```

Eliminación en árboles balanceados

La operación de **eliminación en árboles balanceados** es más compleja que la operación de inserción, como normalmente ocurre en casi todas las estructuras de datos. Consiste en quitar un nodo del árbol sin violar los principios que definen un árbol balanceado. Recuerde que se definió como una estructura en la cual, para todo nodo del árbol, se debe cumplir que: **la altura del subárbol izquierdo y la altura del subárbol derecho no deben diferir en más de una unidad.**

Eliminar nodos en un árbol balanceado resulta difícil a pesar de que se utiliza el mismo algoritmo de eliminación, idéntico en lógica pero diferente en implementación, que en los árboles binarios de búsqueda y las mismas operaciones de reacomodo que se utilizan en el algoritmo de inserción en árboles balanceados.

En la operación de eliminación en árboles balanceados se deben distinguir los siguientes casos:

1. Si el elemento a eliminar es terminal u hoja, simplemente se suprime.
2. Si el elemento a eliminar tiene un solo descendiente, entonces se tiene que sustituir por ese descendiente.
3. Si el elemento a eliminar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Para eliminar un nodo en un árbol balanceado lo primero que se debe hacer es localizar su posición en el árbol. Se elimina siguiendo los criterios establecidos anteriormente y se regresa por el camino de búsqueda calculando el *FE* de los nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio, entonces se debe reestructurar el árbol. El proceso termina cuando se llega a la raíz del árbol. Cabe aclarar que mientras que en el algoritmo de inserción una vez efectuada una rotación se podía detener el proceso, en este algoritmo se debe continuar puesto que se puede producir más de una rotación en el camino hacia atrás. Para comprender mejor la operación de eliminación en árboles balanceados, observe el siguiente ejemplo.

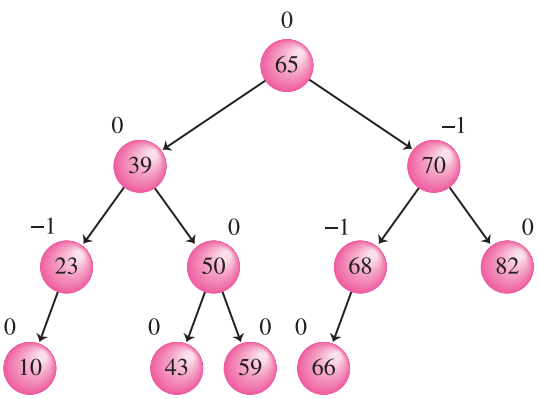
Ejemplo 6.20

Supongamos que se desea eliminar las siguientes claves del árbol balanceado de la figura 6.32:

82 - 10 - 39 - 65 - 70 - 23 - 50 - 59

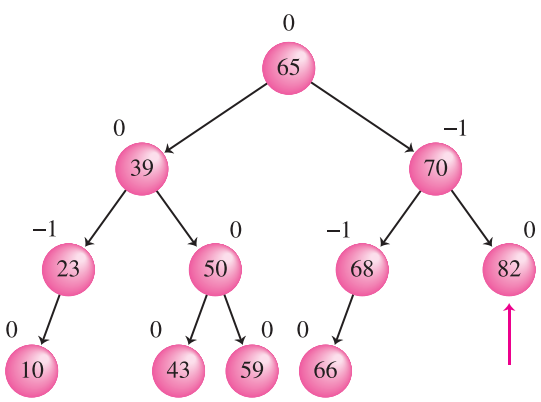
Cabe destacar que el movimiento de apuntadores y la reasignación de los *FE* no se presentarán en este ejemplo, porque son idénticos a los mostrados en el ejemplo 6.18.

FIGURA 6.32
Árbol balanceado.

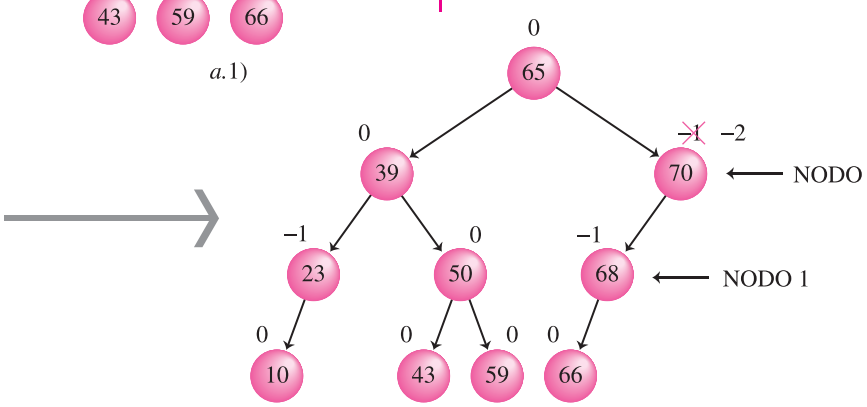


Las operaciones que se realizan son las siguientes:

a) ELIMINACIÓN: CLAVE 82

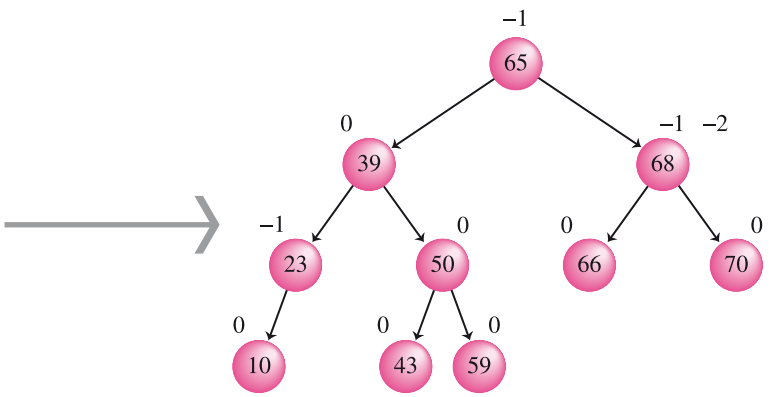


a.1)



a.2)

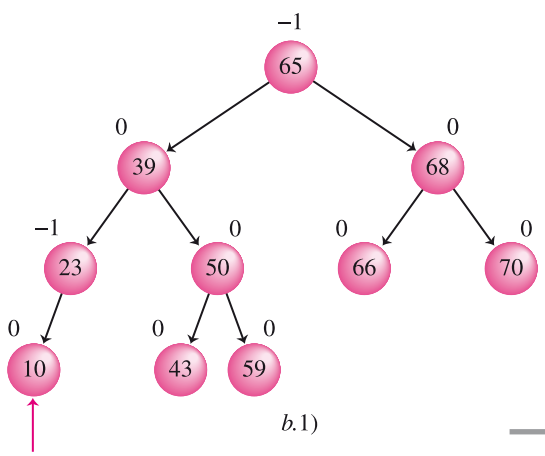
ROTACIÓN *II*



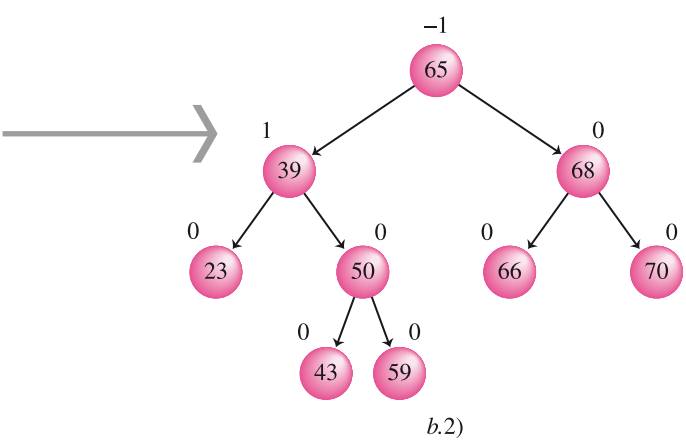
a.3)

La eliminación de la clave 82 es un proceso sencillo, ya que dicha clave no tiene descendientes (diagrama *a*.1). Al regresar siguiendo el camino de búsqueda es evidente que en la clave 70 se rompe el criterio de equilibrio y se debe reestructurar el árbol (diagrama *a*.2). Se apunta con NODO la clave 70 y con NODO1 la rama izquierda de NODO. Se verifica el *FE* de NODO1 y como éste es igual a –1, entonces se realiza la rotación *II*. Luego de la reestructuración, el árbol queda como en el diagrama *a*.3.

b) ELIMINACIÓN: CLAVE 10



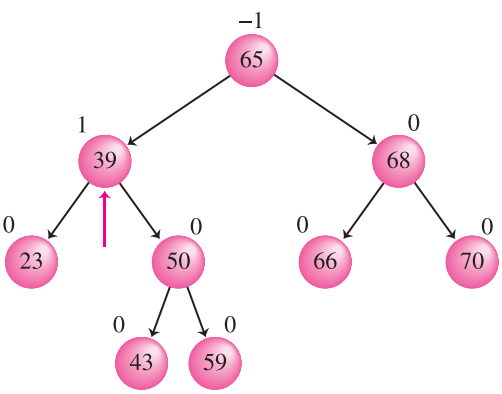
b.1)



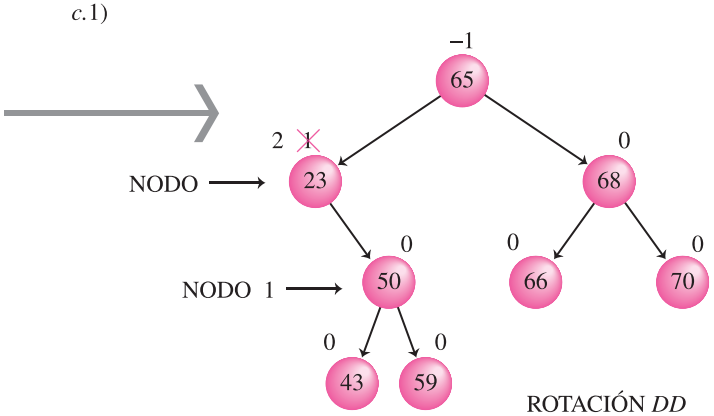
b.2)

La eliminación de la clave 10 es un proceso sencillo (diagrama *b*.1). No se debe reestructurar el árbol porque mantiene el equilibrio y sólo es necesario cambiar el *FE* de los nodos que almacenan al 23 y 39 (diagrama *b*.2).

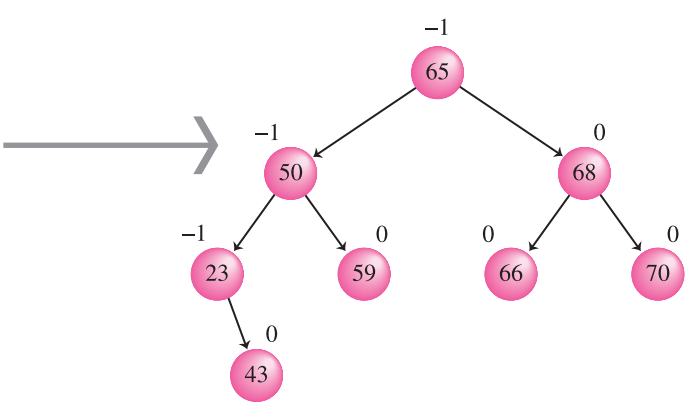
c) ELIMINACIÓN: CLAVE 39



c.1)



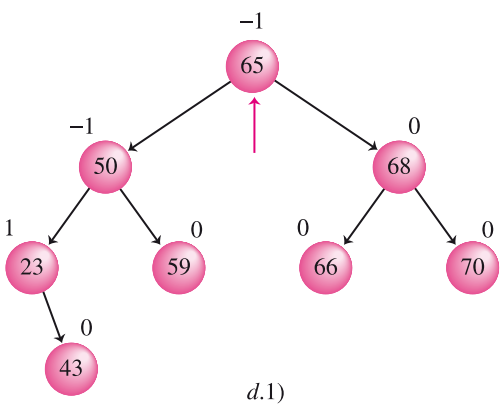
c.2)



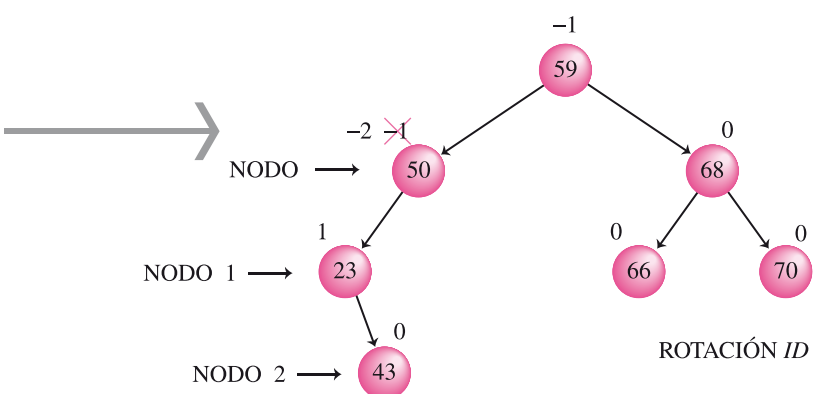
c.3)

Al eliminar la clave 39 se origina el caso más difícil de eliminación en árboles: la eliminación de una clave con dos descendientes (diagrama *c*.l). En este caso se opta por sustituir dicha clave por el nodo que se encuentra más a la derecha en el subárbol izquierdo (23). Luego de la sustitución, el árbol queda como se muestra en el diagrama *c*.2. Al realizar la sustitución se observa que en dicho nodo se viola el criterio de equilibrio y se debe reestructurar el árbol. Se apunta con NODO la clave 23 y con NODO1 la rama derecha de NODO. Se verifica el *FE* de NODO1 y como éste es igual a 0, se realiza la rotación *DD*. Luego del reacomodo, el árbol queda como en el diagrama *c*.3.

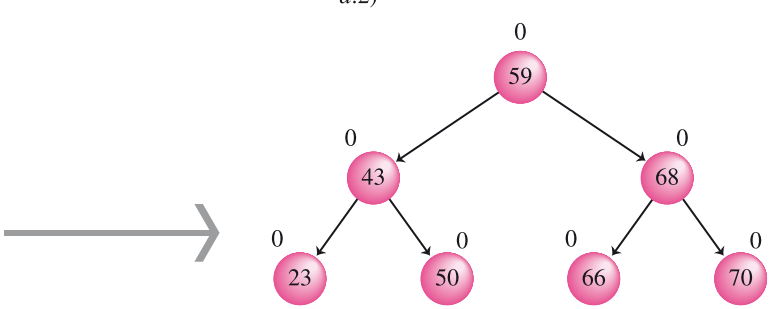
d) ELIMINACIÓN: CLAVE 65



d.1)



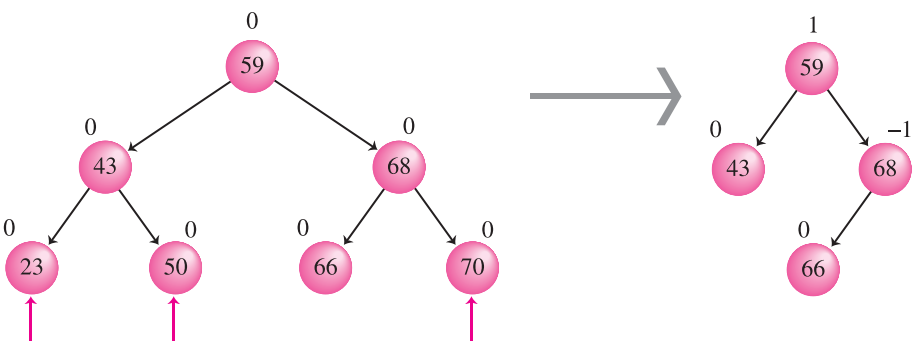
d.2)



d.3)

Al eliminar la clave 65 surge nuevamente el tercer caso de eliminación, que corresponde a una clave con dos descendientes (diagrama *d*.1). Se sustituye dicha clave por el nodo que se encuentra más a la derecha en el subárbol izquierdo (59). Luego de la sustitución, el árbol queda como se presenta en el diagrama *d*.2. Es evidente que, después de la sustitución, en el nodo con la clave 50 se viola el criterio de equilibrio y se debe reestructurar el árbol. Se apunta con NODO la clave 50 y con NODO1 la rama izquierda de NODO y se verifica el *FE*. Como en este caso es igual a 1, se apunta con NODO2 la rama derecha de NODO1 y se realiza la rotación *ID*. Luego de la reestructuración, el árbol queda como el presentado en el diagrama (*d*.3).

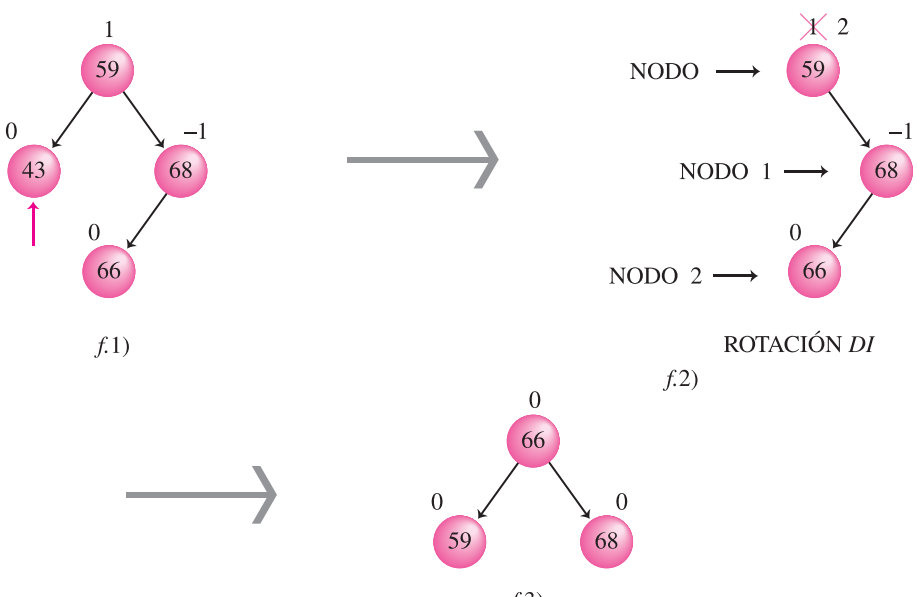
e) ELIMINACIÓN: CLAVES 70, 23 Y 50



e.1) *e*.2)

Luego de la eliminación de las claves, el árbol queda como en el diagrama *e*.2.

f) ELIMINACIÓN: CLAVE 43



f.1)

ROTACIÓN *DI*

f.2)

f.3)

La eliminación de la clave 43 corresponde al primer caso de borrado en árboles, es el caso más simple (diagrama *f*.1). Sin embargo, al verificar el *FE* de la clave 59 se advierte que se rompe el equilibrio del árbol y se debe reestructurar (diagrama *f*.2). Se apunta con NODO la clave 59 y con NODO1 la rama derecha de NODO, y se verifica el *FE* de NODO1. Como éste es igual a –1, se apunta con NODO2 la rama izquierda de NODO1 y se realiza la rotación *DI*. Luego de la reestructuración, el árbol queda como en el diagrama *f*.3.

Observe cuidadosamente que luego de realizar la eliminación de un elemento y cuando se regresa por el camino de búsqueda, el *FE* del nodo visitado disminuye en 1 si la eliminación se hizo por su rama derecha y se incrementa en 1 si la eliminación se hizo por su rama izquierda.

Ejemplo 6.21

Dado el árbol balanceado de la figura 6.33*a*, verifique si el mismo queda igual al de la figura 6.33*b* luego de eliminar las siguientes claves:

25 - 75 - 66 - 65 - 62 - 10 - 43 - 47

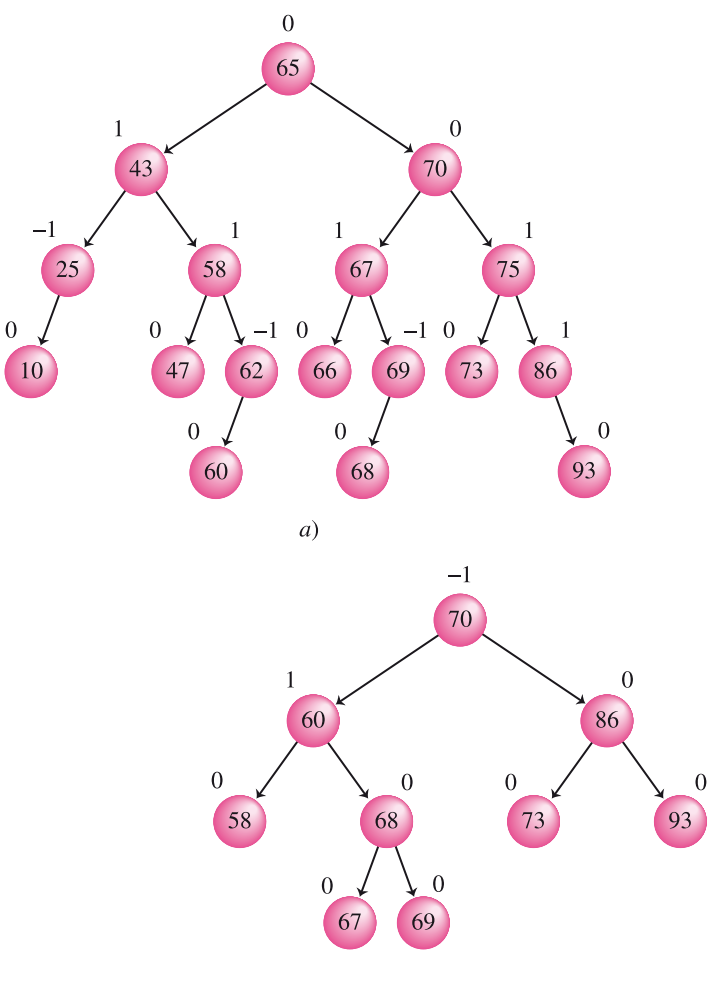


FIGURA 6.33
Eliminación en árboles balanceados. *a*) Antes de eliminar las claves. *b*) Después de eliminar las claves.

Con el fin de darle mayor modularidad al algoritmo de eliminación en árboles balanceados, se estudiarán dos algoritmos auxiliares. El primero, **Reestructura_izq**, se utiliza cuando la altura de la rama izquierda ha disminuido. El segundo, **Reestructura_der**, se emplea cuando la altura de la rama derecha ha disminuido.

Algoritmo 6.11 Reestructura_izq

Reestructura_izq (NODO, BO)

```
{Este algoritmo reestructura el árbol cuando la altura de la rama izquierda ha disminuido y el
FE de NODO es igual a 1. NODO es un parámetro por referencia de tipo puntero. BO es un
parámetro de tipo booleano, también por referencia. BO se utiliza para indicar que la altura de
la rama izquierda ha disminuido}
{NODO1 y NODO2 son variables auxiliares de tipo puntero}

1. Si (BO = VERDADERO)
     entonces
   1.1 Si (NODO^.FE)
         = –1: Hacer NODO^.FE ← 0
         = 0: Hacer NODO^.FE ← 1 y BO ← FALSO
         = 1: {Reestructuración del árbol}
              Hacer NODO1 ← NODO^.DER
       1.1.1 Si (NODO1^.FE ≥ 0)
               entonces {Rotación DD}
                  Hacer NODO^.DER ← NODO1^.IZQ y NODO1^.IZQ ← NODO
         1.1.1.1 Si NODO1^.FE
                   = 0: Hacer NODO^.FE ← 1, NODO1^.FE ← –1 y
                        BO ← FALSO
                   = 1: Hacer NODO^.FE ← 0 y NODO1^.FE ← 0
         1.1.1.2 {Fin del condicional 1.1.1.1}
                 Hacer NODO ← NODO1
                 {Termina la rotación DD}
               si no {Rotación DI}
                 Hacer NODO2 ← NODO1^.IZQ, NODO^.DER ← NODO2^.IZQ,
                    NODO2^.IZQ ← NODO, NODO1^.IZQ ← NODO2^.DER y
                    NODO2^.DER ← NODO1
         1.1.1.3 Si (NODO2^.FE = 1)
                   entonces
                      Hacer NODO^.FE ← –1
                   si no
                      Hacer NODO^.FE ← 0
         1.1.1.4 {Fin del condicional 1.1.1.3}
         1.1.1.5 Si (NODO2^.FE = –1)
                   entonces
                      Hacer NODO1^.FE ← 1
                   si no
                      Hacer NODO1^.FE ← 0
```

```
            1.1.1.6 {Fin del condicional 1.1.1.5}
                    Hacer NODO ← NODO2 y NODO2^.FE ← 0
                    {Termina la rotación DI}
          1.1.2 {Fin del condicional del paso 1.1.1}
    1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

Algoritmo 6.12 Reestructura_der

```
Reestructura_der

{Este algoritmo reestructura el árbol cuando la altura de la rama derecha ha disminuido y el
FE de NODO es igual a –1. NODO es un parámetro, por referencia, de tipo puntero. BO es un
parámetro de tipo booleano, también por referencia. BO se utiliza para indicar que la altura de
la rama derecha ha disminuido}
{NODO1 y NODO2 son variables auxiliares de tipo puntero}

1. Si (BO = VERDADERO) entonces
  1.1 Si NODO^.FE
        = 1 : Hacer NODO^.FE ← 0
        = 0 : Hacer NODO^.FE ← –1 y BO ← FALSO
        = –1: {Reestructuración del árbol}
              Hacer NODO1 ← NODO^.IZQ
        1.1.1 Si (NODO1^.FE ≤ 0)
                entonces {Rotación II)
                    Hacer NODO^.IZQ ← NODO1^.DER y NODO1^.DER ← NODO
          1.1.1.1 Si NODO1^.FE
                    = 0: Hacer NODO^.FE ← –1, NODO1^.FE ← 1 y
                            BO ← FALSO
                    = –1: Hacer NODO^.FE ← 0 y NODO1^.FE ← 0
          1.1.1.2 {Fin del condicional del paso 1.1.1.1}
                    Hacer NODO ← NODO1
                    {Termina la rotación II}
                si no {Rotación ID)
                    Hacer NODO2 ← NODO1^.DER, NODO^.IZQ ← NODO2^.DER,
                        NODO2^.DER ← NODO, NODO1^.DER ← NODO2^.IZQ y
                        NODO2^.IZQ ← NODO1
          1.1.1.3 Si (NODO2^.FE = –1)
                    entonces
                        Hacer NODO^.FE ← 1
                    si no
                        Hacer NODO^.FE ← 0
          1.1.1.4 {Fin del condicional del paso 1.1.1.3}
          1.1.1.5 Si (NODO2^.FE = 1)
                    entonces
                        Hacer NODO1^.FE ← –1
```

```
                    si no
                        Hacer NODO1^.FE ← 0
            1.1.1.6 {Fin del condicional del paso 1.1.1.5}
                    Hacer NODO ← NODO2 y NODO2^.FE ← 0
                    {Termina la rotación ID)
        1.1.2 {Fin del condicional del paso 1.1.1}
    1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

A continuación se presenta el algoritmo de eliminación en árboles balanceados, el cual hará uso de los previamente explicados.

Algoritmo 6.13 Elimina_balanceado

Elimina_balanceado (NODO, BO, INFOR)

```
{El algoritmo elimina un elemento en un árbol balanceado. Utiliza dos algoritmos auxiliares
Reestructura_izq y Reestructura_der. NODO es un parámetro por referencia de tipo puntero.
BO es un parámetro de tipo booleano, también por referencia, y se utiliza para indicar que la
altura del árbol ha disminuido, su valor inicial es FALSO. INFOR es un parámetro de tipo
entero que contiene la información del elemento que se quiere eliminar}
{OTRO, AUX, AUX1 son variables auxiliares de tipo puntero. BOOL es una variable de tipo
booleano}

1. Si (NODO ≠ NIL)
      entonces
   1.1 Si (INFOR < NODO^.INFO)
         entonces
            Regresar a Elimina_balanceado con NODO^.IZQ, BO e INFOR
            Llamar al algoritmo Reestructura_izq con NODO y BO
         si no
      1.1.1 Si (INFOR > NODO^.INFO)
               entonces
                  Regresar a Elimina_balanceado con NODO^.DER, BO e INFOR
                  Llamar al algoritmo Reestructura_der con NODO y BO
               si no
                  Hacer OTRO ← NODO y BO ← VERDADERO
            1.1.1.1 Si (OTRO^.DER = NIL)
                     entonces
                        Hacer NODO ← OTRO^.IZQ
                     si no
                  1.1.1.1.1 Si (OTRO^.IZQ = NIL)
                              entonces
                                 Hacer NODO ← OTRO^.DER
                              si no
```

```
                    Hacer AUX ← NODO^.IZQ y BOOL ← FALSO
1.1.1.1.1.A  Mientras (AUX^.DER ≠ NIL) Repetir
                        Hacer AUX1 ← AUX, AUX ← AUX^.DER
                          y BOOL ← VERDADERO
1.1.1.1.1.B  {Fin del ciclo del paso 1.1.1.1.1.A}
                      Hacer NODO^.INFO ← AUX^.INFO y
                        OTRO ← AUX
1.1.1.1.1.C  Si (BOOL = VERDADERO)
                        entonces
                          Hacer AUX1^.DER ← AUX^.IZQ
                        si no
                          Hacer NODO^.IZQ ← AUX^.IZQ
1.1.1.1.1.D  {Fin del condicional del paso 1.1.1.1.1.C}
                      Llamar al algoritmo Reestructura_der
                        con NODO^.IZQ y BO
          1.1.1.1.2  {Fin del condicional del paso 1.1.1.1.1}
        1.1.1.2  {Fin del condicional del paso 1.1.1.1}
                    Quitar (OTRO) {Libera la memoria del nodo}
      1.1.2  {Fin del condicional del paso 1.1.1}
    1.2  {Fin del condicional del paso 1.1}
      si no
        Escribir "La información  no se encuentra en el árbol"
2.  {Fin del condicional del paso 1}
```

El análisis matemático de los algoritmos de inserción —Inserta_balanceado— y eliminación —Elimina_balanceado— demuestra que es posible buscar, insertar y eliminar un elemento en un árbol balanceado de n nodos en O(log n) unidades de tiempo. Por otra parte, diversos análisis demuestran que son más frecuentes las rotaciones en las operaciones de inserción que en las de eliminación, ya que mientras se produce aproximadamente una rotación por cada dos inserciones, se produce una rotación por cada cinco eliminaciones.

Ejemplo 6.22

Supongamos que se desea eliminar la clave 65 del árbol balanceado de la figura 6.34:

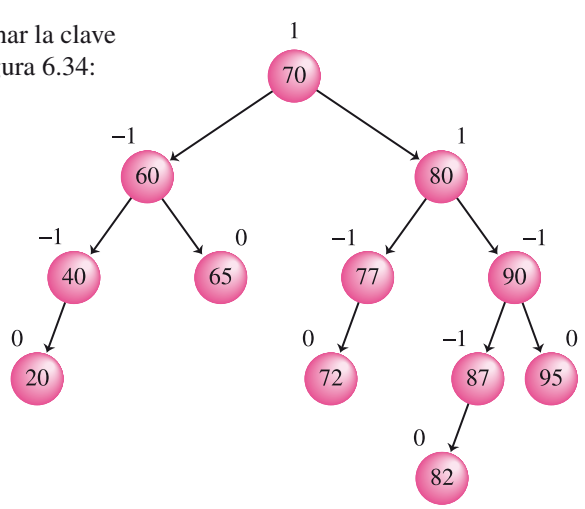
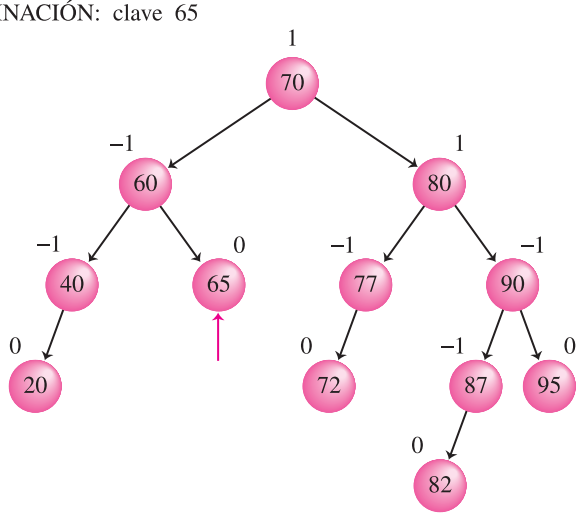


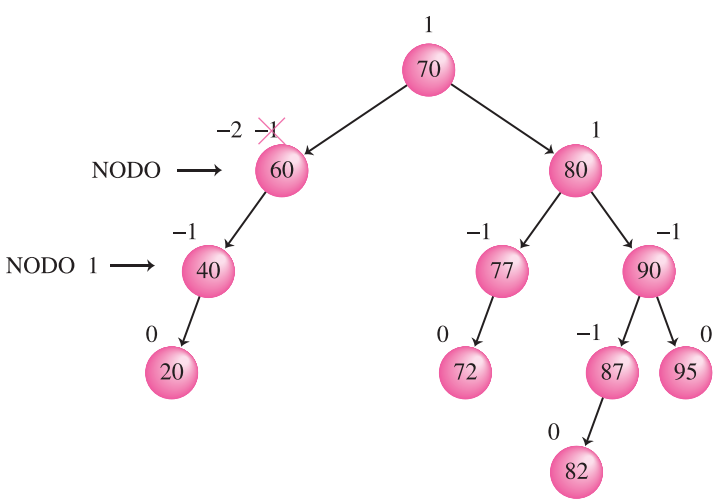
FIGURA 6.34
Árbol balanceado.

Las operaciones que se realizan son las siguientes:

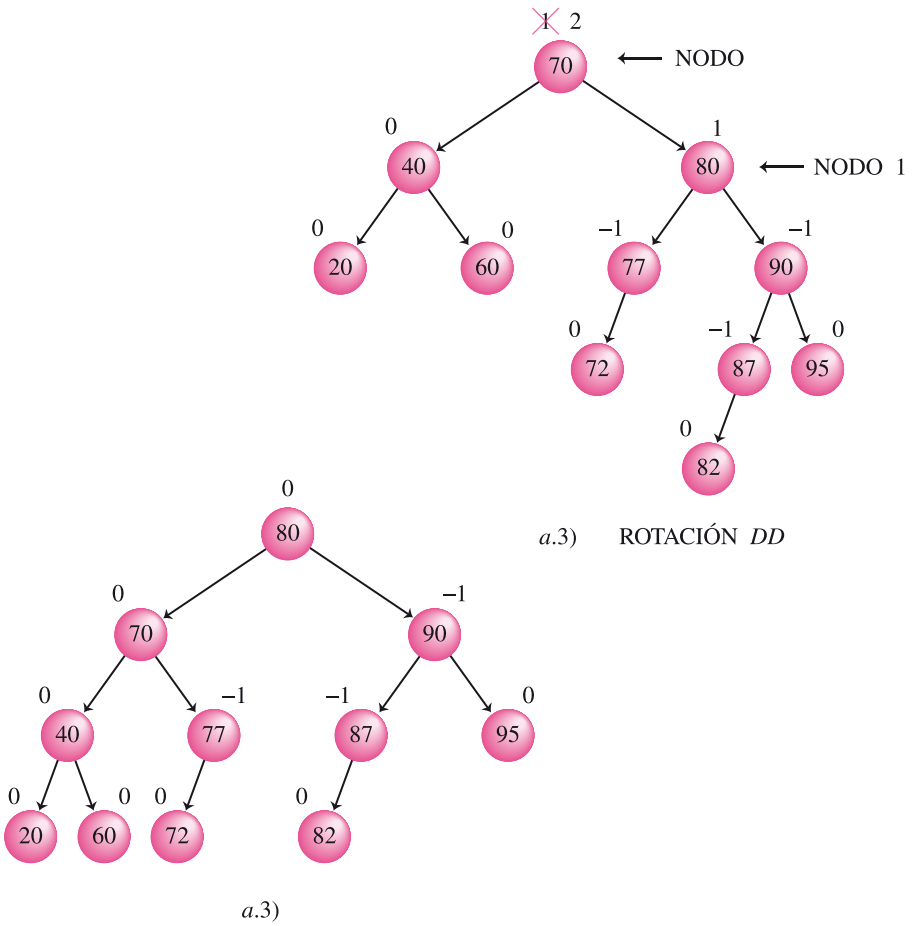
a) ELIMINACIÓN: clave 65



a.1)



a.2) ROTACIÓN *II*



a.3) ROTACIÓN *DD*

a.3)

Observe el lector que al eliminar la clave 65 se desbalancea el árbol y debemos efectuar la rotación *II*. Sin embargo, luego de balancear y modificar el factor de equilibrio del nodo que almacena la clave 70 nos damos cuenta que debemos efectuar un nuevo balanceo, ahora una rotación *DD*. Éste es un típico caso donde al eliminar una clave se produce una cadena de balanceos.

6.5 ÁRBOLES MULTICAMINOS

Los diferentes tipos de árboles binarios estudiados hasta el momento fueron desarrollados para funcionar en la memoria principal de la computadora. Sin embargo, existen muchas aplicaciones en las que el volumen de información es tal, que los datos no caben en la memoria principal y es necesario almacenarlos, organizados en archivos, en dispositivos de almacenamiento secundario. Esta organización de archivos debe ser suficientemente adecuada como para recuperar los datos en forma eficiente.

Es importante recordar que el tiempo necesario para localizar un registro en la memoria principal de la computadora se mide en microsegundos, mientras que el tiempo necesario para localizar una página (contiene varios registros) en memoria secundaria, por ejemplo disco, se mide en milisegundos. El tiempo de acceso, claro está, es miles de veces más rápido en la memoria principal que en la memoria secundaria.

Considere el caso de almacenar un árbol binario en disco. Se necesitará en promedio, para localizar alguno de los nodos, $\log_d n$ accesos a disco, donde n representa el número de nodos del árbol y del orden del mismo, que en este caso es igual a 2. Por ejemplo si el árbol contiene 1 000 000 de elementos, se necesitarían aproximadamente 20 accesos a disco. Ahora bien, si el árbol está organizado en páginas —nodos—, de tal manera que cada página contenga como mínimo 100 elementos, entonces se necesitarían como máximo tres accesos a disco ($\log_{100}$ 1 000 000). Note el lector que los accesos a disco disminuyen de modo considerable.

Existen diferentes técnicas para la organización de archivos indizados, sin embargo la organización en árboles-*B* y específicamente su variante, la organización en árboles-B^+, es la más utilizada.

6.5.1 Árboles-*B*

Los **árboles-*B*** son una generalización de los árboles balanceados. Éstos representan básicamente un método para almacenar y recuperar información en medios externos. Fueron propuestos por Bayer y McCreight en 1970. Su nombre árboles-*B* nunca fue explicado por los autores, aunque muchos sostienen que *B* proviene de Bayer, uno de sus inventores.

En este tipo de árboles, un grupo de nodos recibe el nombre de **página**. En cada página se almacena la información de un grupo de nodos y se identifica por medio de una clave o llave.

En general cada página de un árbol *B* de orden *d* contiene 2*d* claves como máximo y *d* claves como mínimo. Con esto se garantiza que cada página esté llena como mínimo hasta la mitad. Respecto al número de descendientes, cada página de un árbol-*B* de orden *d* tiene 2*d* + 1 hijos como máximo y *d* + 1 hijos como mínimo, excepto la página raíz que puede contener como mínimo 1 dato y por consiguiente solamente 2 hijos. Las páginas en general son almacenadas en dispositivos de almacenamiento secundario, a excepción de la página raíz que es conveniente mantenerla en memoria principal. Cabe mencionar, que básicamente por cuestiones de espacio, en los ejemplos y figuras, en cada nodo se almacena solamente un dato, la clave con la cual vamos a trabajar. En la figura 6.35 se presenta un diagrama correspondiente a un árbol-*B* de orden 2.

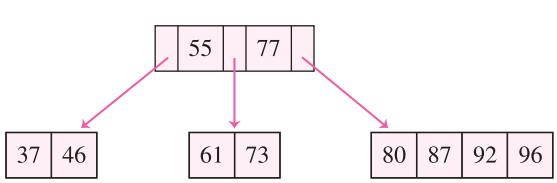
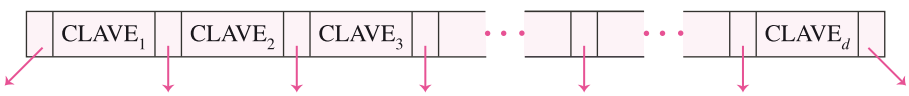


FIGURA 6.35
Árbol-*B* de orden 2.

En la figura 6.36 se observa una página de un árbol-*B* de orden *d*, con *d* claves y *d* + 1 hijos.

FIGURA 6.36
Página de un árbol-*B* de orden *d*.



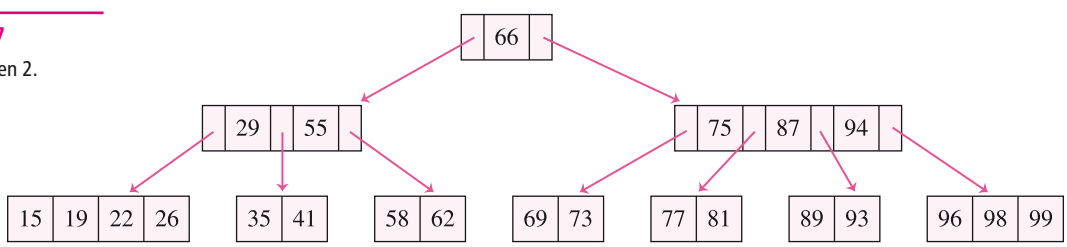
Formalmente un árbol-*B* se define de la siguiente manera:

1. Cada página, excepto la raíz, contiene entre *d* y 2*d* elementos, siendo *d* el grado del árbol.
2. La raíz puede almacenar entre *1* y *2d* elementos.
3. Cada página, excepto la página raíz y las páginas hoja, tiene entre *d* + 1 y 2*d* + 1 descendientes. Se utilizará *m* para expresar el número de elementos por página.
4. La página raíz tiene al menos dos descendientes.
5. Las páginas hoja están todas al mismo nivel.

Ejemplo 6.23

Luego de analizar el árbol-*B* de la figura 6.37 se puede afirmar lo siguiente respecto a éste:

FIGURA 6.37
Árbol-*B* de orden 2.



- Orden del árbol: 2
- Altura del árbol: 3
- Todas las páginas contienen 2, 3 o 4 elementos, excepto la raíz que contiene 1.
- Los elementos dentro de la página se encuentran ordenados en forma creciente, de izquierda a derecha.
- Todas las hojas están al mismo nivel.
- Todas las páginas tienen 3 o 4 descendientes.

Búsqueda en árboles-*B*

El proceso de **búsqueda en árboles-*B*** es una generalización del proceso de búsqueda en árboles binarios de búsqueda. Los pasos necesarios para localizar una clave *X* en un árbol-*B* son los que se presentan a continuación. Se utiliza NIL para indicar que la página está vacía.

1. Se debe tener en memoria la página sobre la cual se quiere trabajar.
  1.1 *Si* (página ≠ NIL)
    entonces
      Se avanza hacia el paso 2

    si no
      Se avanza hacia el paso 3
  1.2 {Fin del condicional del paso 1.1}

2. Se debe verificar si la clave buscada se encuentra en dicha página. Si *m* es pequeña se utilizará búsqueda secuencial, de otra manera se podrá utilizar búsqueda binaria.

  2.1 *Si* (la clave se encuentra en la página)
    entonces {La operación de búsqueda concluye cuando se encuentra ¡ÉXITO! el dato en la página visitada}

    si no
      Se deben distinguir los siguientes casos:
      Si ($X < CL_{1)}$) *entonces*
        Se debe localizar PAG_0
      Si ($CL_i < X < CL_m$) *entonces*
        Se debe localizar PAG_i
      Si ($X > CL_m$) *entonces*
        Se debe localizar PAG_m
  2.2 {Fin del condicional del paso 2.1}
  2.3 Regresar al paso 1.

  Nota: Se utiliza el término *CL* para hacer referencia a las claves de una determinada página, *X* para indicar la clave que se busca y *PAG* para expresar la página que debe localizarse en memoria secundaria.

3. ¡FRACASO! La página que se desea localizar está vacía, por lo tanto el proceso de búsqueda se interrumpe y se informa que la clave no se encuentra almacenada en el árbol.

Inserción en árboles-*B*

El proceso de **inserción en árboles-*B*** es relativamente sencillo, aunque requiere cierto tratamiento especial debido a las características propias de estos árboles. Los árboles-*B* tienen un comportamiento típico, diferente al resto de los árboles estudiados anteriormente. Todas las hojas están al mismo nivel y por lo tanto cualquier camino desde la raíz hasta alguna de las hojas tiene la misma longitud. Por otra parte, los árboles-*B* tienen una forma extraña de crecer, lo hacen de abajo hacia arriba, es decir, desde las hojas hacia la raíz. Los pasos para llevar a cabo la inserción de un nodo en un árbol-*B* son los siguientes:

1. Localizar la página donde corresponde —por el valor, para no alterar el orden— insertar la clave.

2. *Si* ($m < 2d$) {El número de elementos de la página es menor a $2d$}

entonces
La clave se inserta en el lugar que le corresponde
{En la figura 6.38 se presenta un ejemplo de este caso}
si no {El número de elementos de la página es igual a $2d$}
La página afectada se divide en 2 y se distribuyen las $m + 1$ claves equitativamente entre las mismas. La clave del medio sube a la página antecesora
{En la figura 6.39 se presenta un ejemplo de este caso}

3. {Fin del condicional del paso 2}

Los pasos anteriores se repiten mientras sea necesario. Si alguna de las páginas antecesoras se desborda nuevamente, entonces hay que ordenar las claves en la página, aplicar partición y la clave del medio sube a la página antecesora. El proceso de propagación puede llegar incluso hasta la raíz, en dicho caso la altura del árbol se incrementa en una unidad.

{En la figura 6.40 se presenta un ejemplo de este caso}

FIGURA 6.38

Inserción de la clave 15 en un árbol-*B*. *a*) Antes de insertar la clave. *b*) Después de insertar la clave.

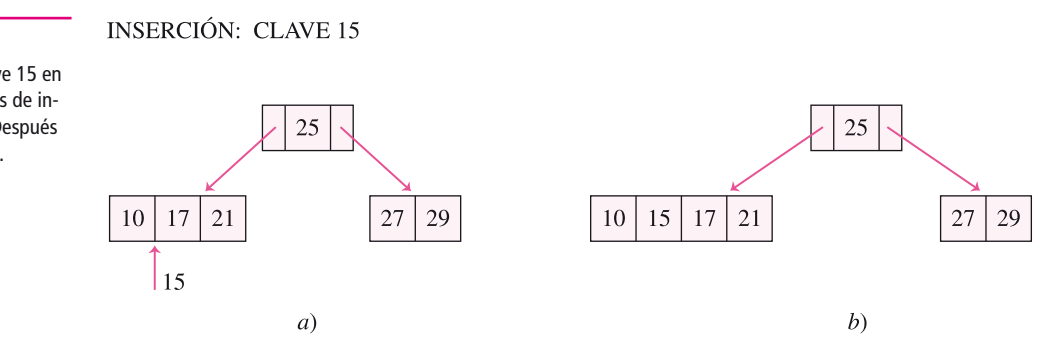


FIGURA 6.39

Inserción de la clave 13 en un árbol-*B*. *a*) Antes de insertar la clave. *b*) Después de insertarla.

Nota: Observe el lector que la inserción de la clave 13 provocó la división de la página *A* en dos páginas: *B* y *C*. Las claves se distribuyeron equitativamente entre las páginas citadas y la clave del medio (15) subió a la página antecesora.

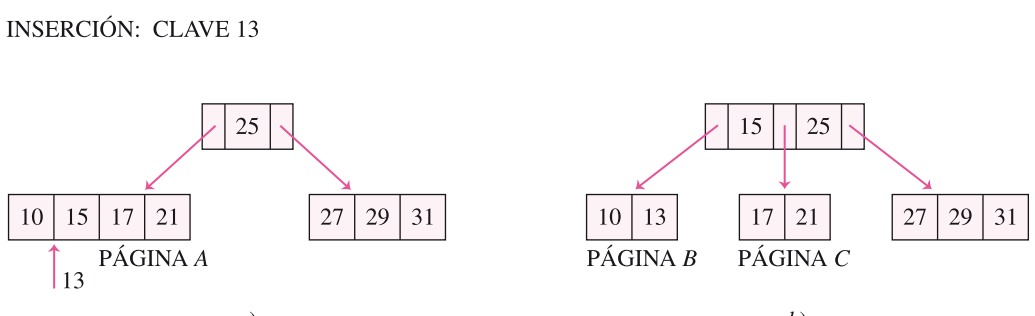
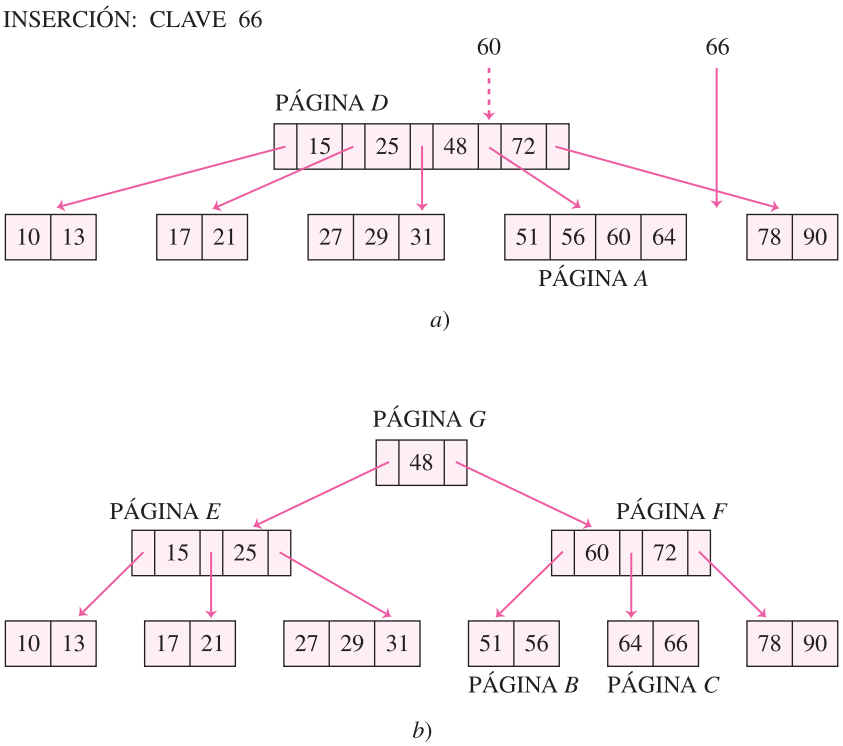


FIGURA 6.40
Inserción de la clave 66 en un árbol-*B*. *a*) Antes de insertar la clave. *b*) Después de insertarla.

Nota: Observe el lector que la inserción de la clave 66 provocó la división de la página *A* en dos páginas: *B* y *C*. Sin embargo, al subir la clave del medio (60) se produjo un nuevo desbordamiento que originó la partición de la página *D* en las páginas *E* y *F*. La clave 48 forma ahora parte de una nueva página (*G*) y representa la raíz del árbol.



Ejemplo 6.24

Supongamos que se desea insertar las siguientes claves en un árbol-*B* de orden 2 que se encuentra vacío:

10 - 27 - 29 - 17 - 25 - 21 - 15 - 31 - 13 - 51 - 20 - 24 - 48 - 19 - 60 - 35 - 66

Los resultados parciales que ilustran el crecimiento del árbol se presentan en los diagramas de la figura 6.41.

FIGURA 6.41
Inserciones en un árbol-*B* de orden 2.

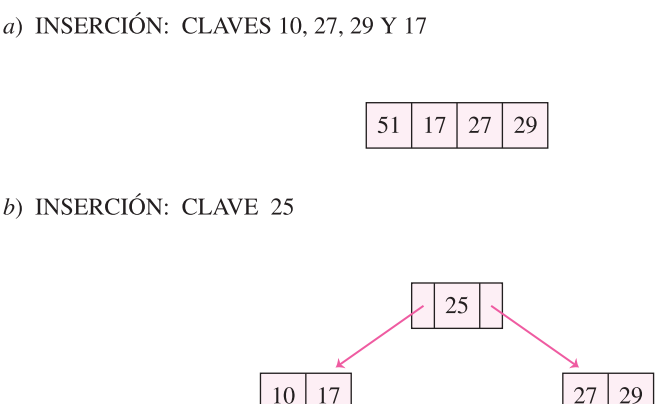
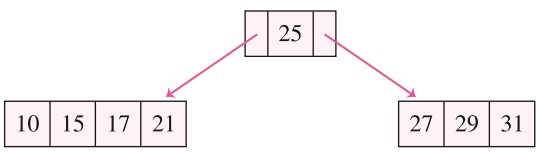
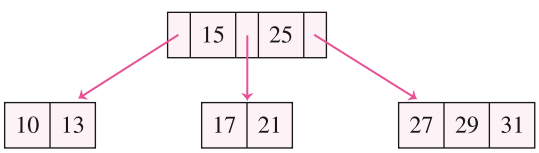


FIGURA 6.41
(*continuación*)

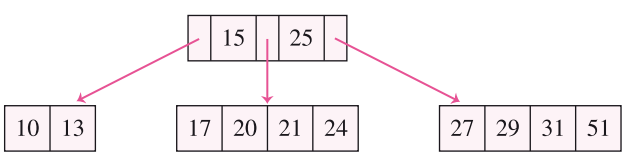
c) INSERCIÓN: CLAVES 21, 15 Y 31



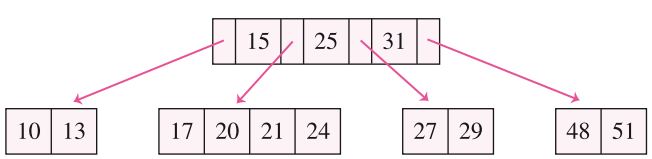
d) INSERCIÓN: CLAVE 13



e) INSERCIÓN: CLAVES 51, 20 Y 24



f) INSERCIÓN: CLAVE 48



g) INSERCIÓN: CLAVE 19

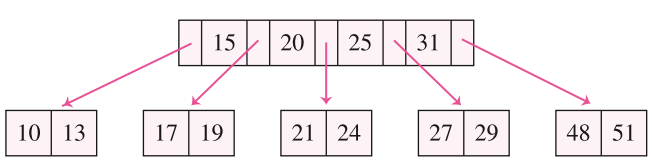
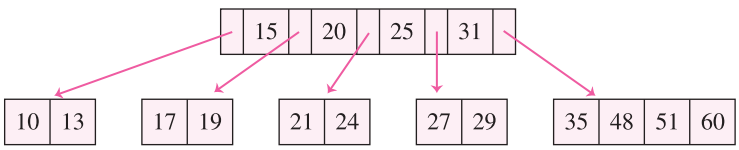
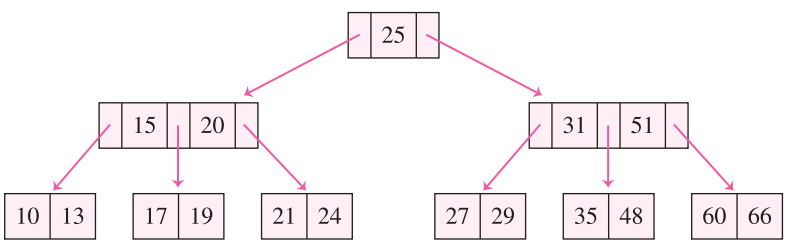


FIGURA 6.41
(*continuación*)

h) INSERCIÓN: CLAVES 60 Y 35



i) INSERCIÓN: CLAVE 66

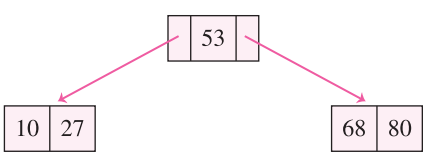


Ejemplo 6.25

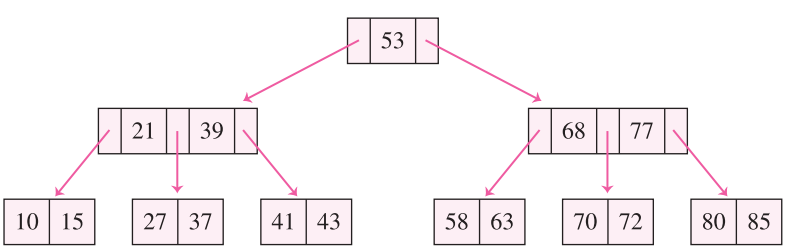
Dado como dato el árbol-*B* de orden 2 de la figura 6.42*a*, verifique si el mismo queda igual al de la figura 6.42*b* luego de insertar las siguientes claves:

43 - 21 - 77 - 58 - 63 - 15 - 37 - 41 - 72 - 39 - 95 - 70

FIGURA 6.42
Inserción en un árbol-*B* de orden 2. *a*) Antes de insertar las claves. *b*) Después de insertar las claves.



a)



b)

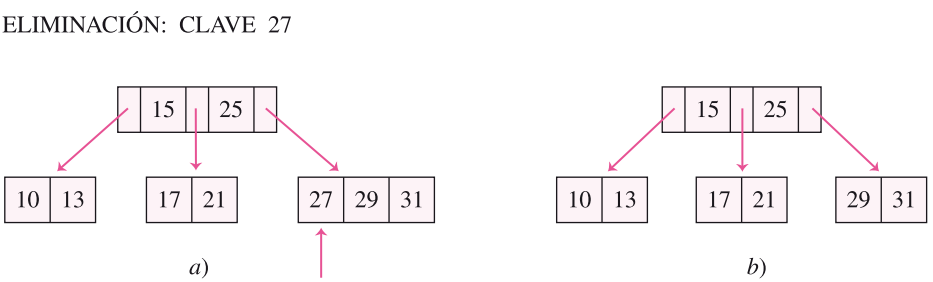
Eliminación en árboles-*B*

La operación de **eliminación en árboles-*B*** es una operación más complicada que la inserción. Consiste en quitar una clave del árbol sin violar la condición de que en una página, excepto la raíz, no puede haber menos de d claves ni más de $2d$ claves, siendo d el orden del árbol. En la operación de borrado se deben distinguir los siguientes casos:

1. *Si* la clave a eliminar se encuentra en una página hoja *entonces* simplemente se suprime.
  1.1 *Si* ($m \geq d$)
    {Se verifica que el número de elementos en la página sea válido}
      entonces
        Termina la operación de borrado.
        {Se presenta un ejemplo de este caso en la figura 6.43}
      si no
        Se debe bajar la clave lexicográficamente adyacente de la página antecesora y sustituir esta clave por la que se encuentre más a la derecha en el subárbol izquierdo o por la que se encuentre más a la izquierda en el subárbol derecho. Con este paso se logra que m, en esta página, siga siendo $\geq d$.
    {Se presenta un ejemplo en las figuras 6.44*a* y 6.44*b*}
      Si esto no es posible, por las m de las páginas involucradas, se deben fusionar las páginas que son descendientes directas de la clave que se baja.
    {Se presenta un ejemplo de este caso en las figuras 6.44*c* y 6.44*d*}
  1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

3. *Si* la clave a eliminar no se encuentra en una página hoja *entonces*
    Se debe sustituir por la clave que se encuentra más a la izquierda en el subárbol derecho o por la clave que se encuentra más a la derecha en el subárbol izquierdo.
  3.1 *Si* ($m \geq d$)
    {Se verifica que el número de elementos en la página sea válido}

FIGURA 6.43
Eliminación de la clave 27 en un árbol-*B* de orden 2. *a*) Antes de eliminar la clave. *b*) Después de eliminarla.



entonces
Termina la operación de borrado.
{Se presenta un ejemplo de este caso en la figura 6.45}
si no
Se debe bajar la clave lexicográficamente adyacente de la página antecesora y fusionar las páginas que son descendientes directas de dicha clave.
{En la figura 6.46 se presenta un ejemplo de este caso}
3.2 {Fin del condicional del paso 3.1}
4. {Fin del condicional del paso 3}

Cabe aclarar que el proceso de fusión de páginas se puede propagar incluso hasta la raíz, en cuyo caso la altura del árbol disminuye en una unidad. En la figura 6.47 se presentan dos ejemplos de este caso.

FIGURA 6.44
Eliminación de las claves 21 y 10 en un árbol-*B* de orden 2. *a*) Antes de eliminar la clave 21. *b*) Después de eliminarla. *c*) Antes de eliminar la clave 10. *d*) Después de eliminarla.

Notas: Al eliminar la clave 21 de la página *A*, baja la clave 25 de la página antecesora y ésta es sustituida por la que se encuentra más a la izquierda en la página derecha; es decir, la clave 27 de la página *B*.

Al eliminar la clave 10 de la página *A*, baja la clave 15 de la página antecesora y se fusionan las páginas *A* y *B*.

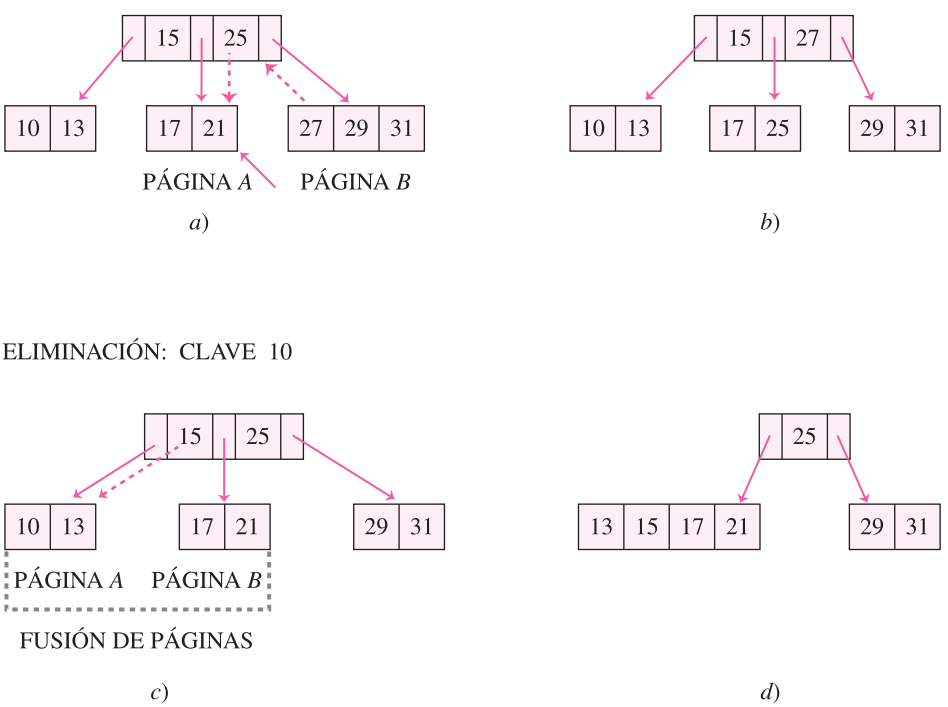


FIGURA 6.45
Eliminación de la clave 15 en un árbol-*B* de orden 2. *a*) Antes de eliminar la clave. *b*) Después de eliminarla.

Nota: Al eliminar la clave 15 se sustituye por la clave que se encuentra más a la izquierda en el subárbol derecho (17).

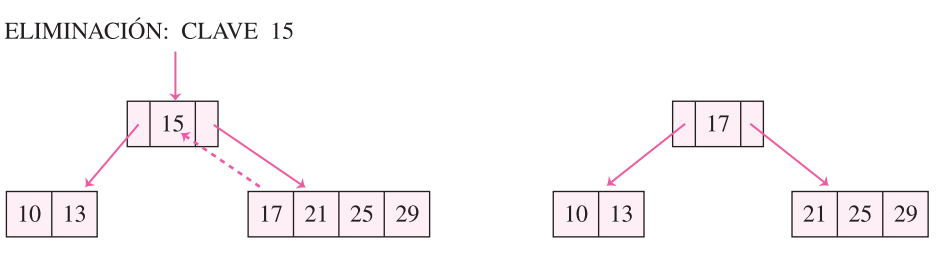


FIGURA 6.46

Eliminación de la clave 25 en un árbol-*B* de orden 2. *a*) Antes de eliminar la clave. *b*) Después de eliminarla.

Nota: Al eliminar la clave 25 se sustituye por la clave que se encuentra más a la derecha en el subárbol izquierdo (21). Sin embargo, al subir la clave 21, en la página *A*, *m* queda menor que *d*, por lo que es necesario realizar una fusión. Baja la clave correspondiente a la página antecesora (nuevamente 21), y se fusionan las páginas *A* y *B*.

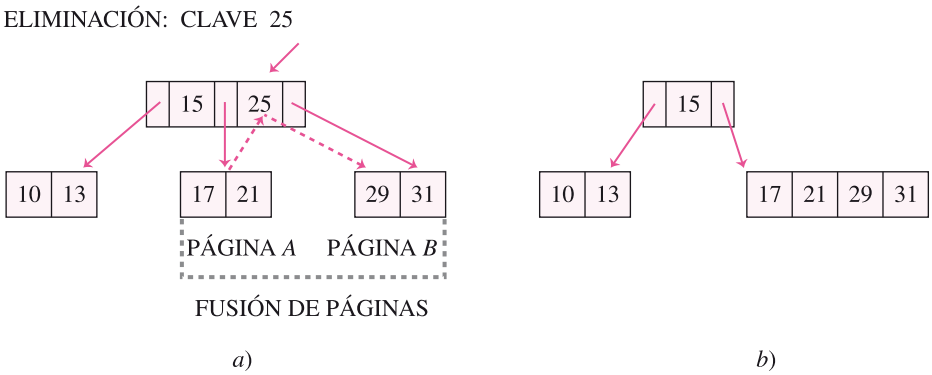


FIGURA 6.47

Eliminación de las claves 21 y 25 en un árbol-*B* de orden 2. *a*) Antes de eliminar la clave 21. *b*) Después de eliminarla.

Nota: Al eliminar la clave 21 de la página *A*, *m* queda menor a *d*, por lo que es necesario bajar la clave 20 de la página antecesora, produciéndose la fusión de las páginas *A* y *B*. Sin embargo, en la página *C* nuevamente *m* queda menor a *d*, por lo que es necesario bajar la clave 25 de la página *E*. Como esta página queda vacía, es necesaria entonces una nueva fusión, ahora de las páginas *C* y *D*. La altura del árbol disminuye en una unidad.

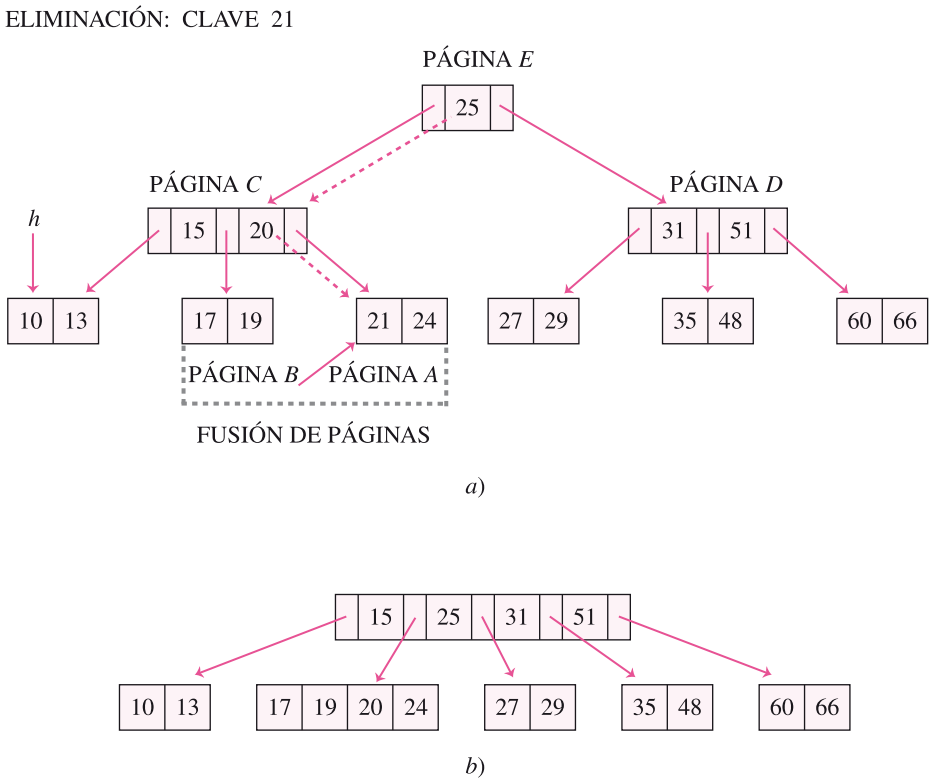
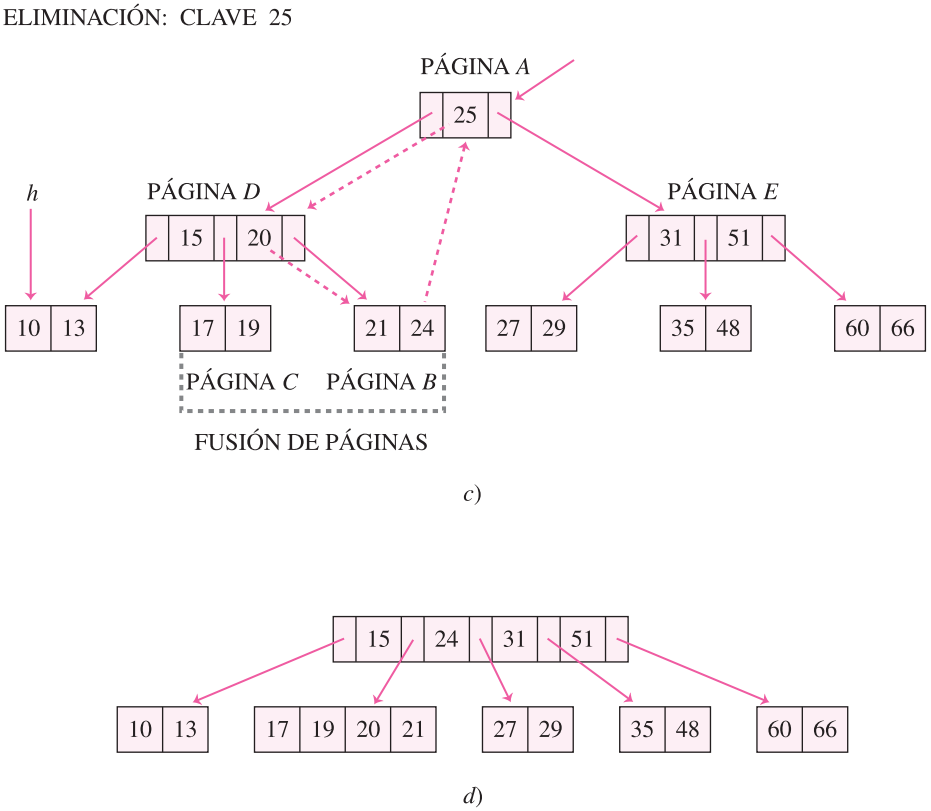


FIGURA 6.47
(*continuación*)
c) Antes de eliminar la clave 25. *d*) Después de eliminarla.

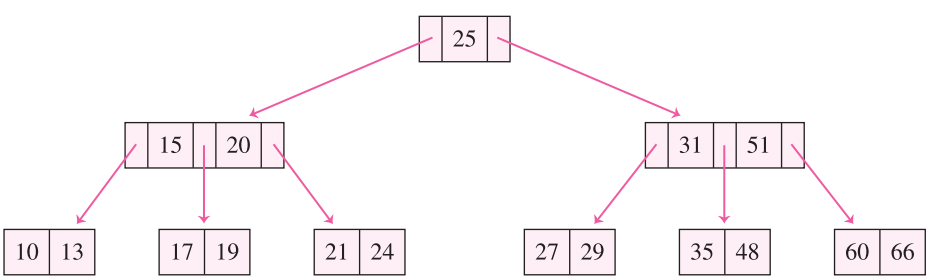
Nota: Al eliminar la clave 25 de la página *A*, se sustituye por la clave que se encuentra más a la derecha en el subárbol izquierdo (24 de la página *B*). Sin embargo, en la página *B*, *m* queda menor que *d*, por lo que es necesario bajar la clave 20 de la página *D* produciéndose la fusión de las páginas *B* y *C*. Nuevamente en la página *D*, *m* queda menor a d, por lo que ahora es necesario bajar la clave 24 de la página *A*. Como esta página queda vacía entonces necesita realizarse una fusión de las páginas *D* y *E*. La altura del árbol disminuye en una unidad.



Ejemplo 6.26

Supongamos que se desea eliminar las siguientes claves del árbol-*B* de orden 2 de la figura 6.48:

FIGURA 6.48
Árbol-*B* de orden 2.

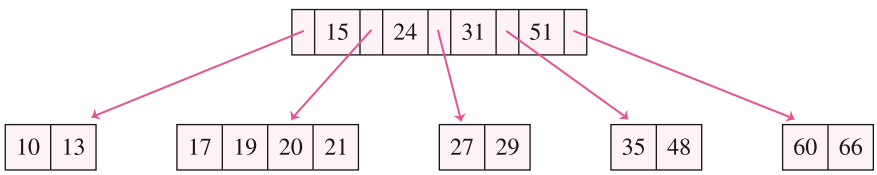


25 - 24 - 29 - 27 - 48 - 19 - 51 - 21 - 13 - 15 - 17 - 66 - 10

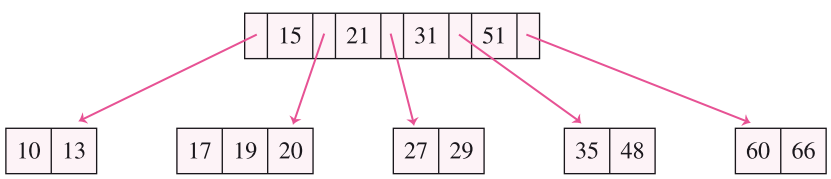
Los resultados parciales que ilustran cómo funciona el procedimiento se presentan en los diagramas de la figura 6.49.

FIGURA 6.49
Eliminaciones en un árbol-*B* de orden 2.

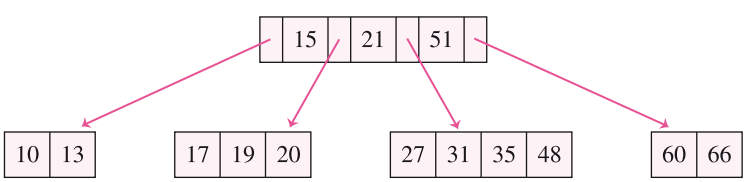
a) ELIMINACIÓN: CLAVE 25



b) ELIMINACIÓN: CLAVE 24



c) ELIMINACIÓN: CLAVE 29



d) ELIMINACIÓN: CLAVES 27, 48 Y 19

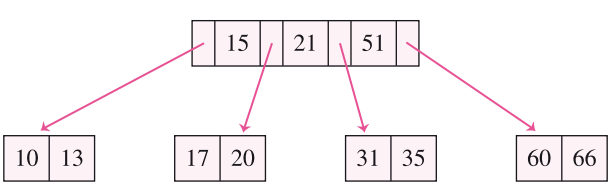
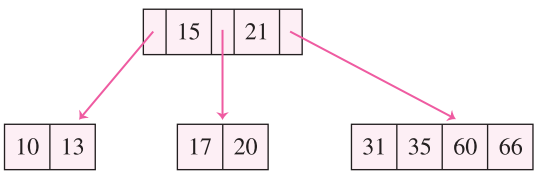
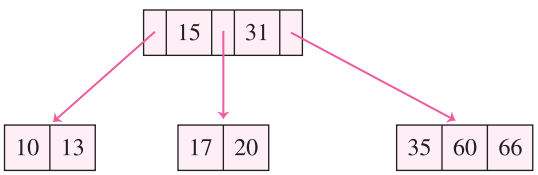


FIGURA 6.49
(*continuación*)

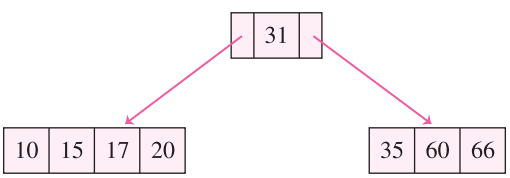
e) ELIMINACIÓN: CLAVE 51



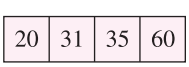
f) ELIMINACIÓN: CLAVE 21



g) ELIMINACIÓN: CLAVE 21



h) ELIMINACIÓN: CLAVES 15, 17, 66 Y 10

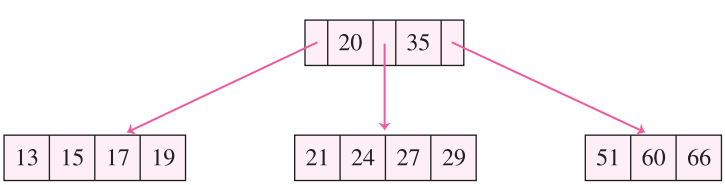


Ejemplo 6.27

Dado como dato el árbol-*B* de orden 2 de la figura 6.48, verifique si el mismo queda igual al de la figura 6.50, luego de eliminar las siguientes claves:

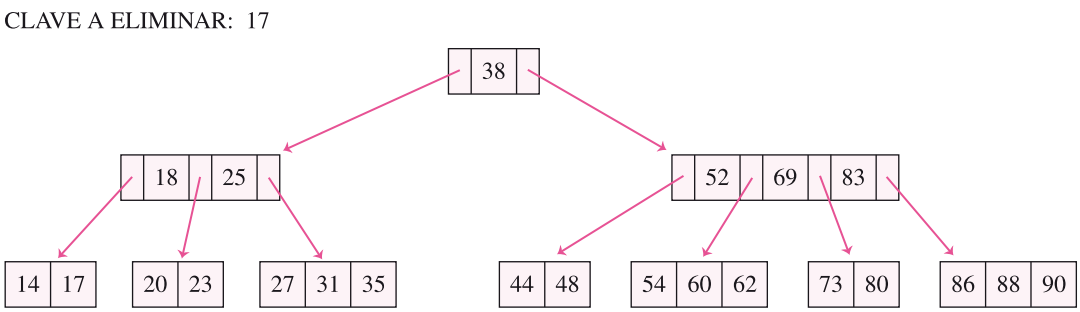
48 - 31 - 10 - 25

FIGURA 6.50
Árbol-*B* de orden 2 luego de eliminar las claves 48, 31, 10 y 25.



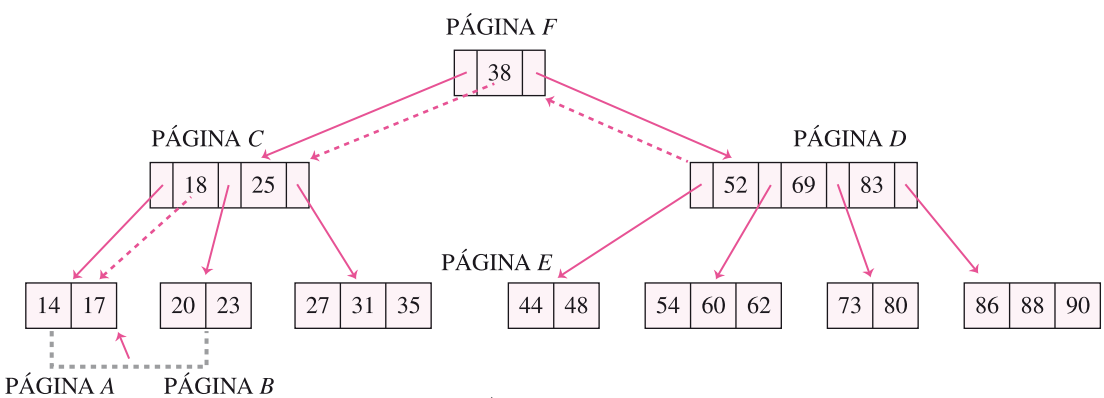
Ejemplo 6.28 Supongamos que se desea eliminar la clave 17 del árbol-*B* de orden 2 de la figura 6.51:

FIGURA 6.51
Árbol-*B* de orden 2.

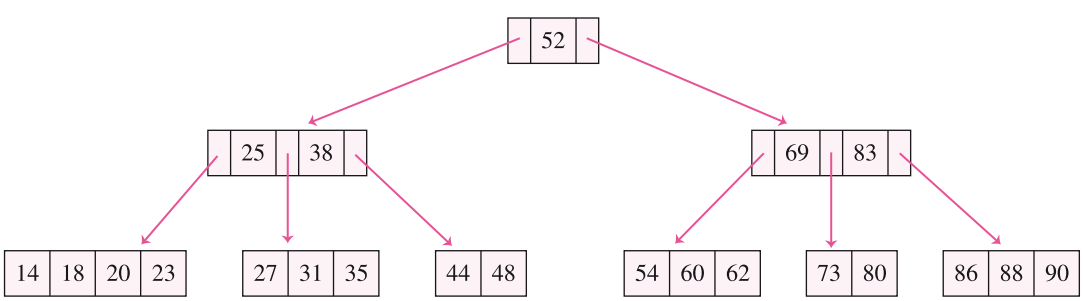


Las operaciones que se realizan son las siguientes:

a) ELIMINACIÓN: CLAVE 17



a)



b)

Nota: Al eliminar la clave 17 de la página *A*, *m* queda menor a *d*, por lo que es necesario bajar la clave 18 de la página *C,* produciéndose la fusión de las páginas *A* y *B.* Sin embargo, en la página *C* nuevamente *m* queda menor a *d,* por lo que es necesario bajar la clave 38 de la página *F.* Aquí es donde se produce uno de los casos más difíciles de borrado en árboles-*B*. En los ejemplos anteriores hacíamos fusión de las páginas *C* y *D*, disminuyendo la altura del árbol. Sin embargo, si hiciéramos esto *m* sería mayor a 2*d,* por lo que violaríamos los principios que definen un árbol-*B*. Es necesario entonces subir la clave 52 de la página *D* a la página *F*, y la página *E* pasa a ser el hijo derecho de la clave 38, ahora en la página *C*.

6.5.2 Árboles-B^+

Los **árboles-B^+** se han convertido en la técnica más utilizada para la organización de archivos indizados. La principal característica de estos árboles es que toda la información se encuentra en las hojas, mientras que los nodos raíz e interiores almacenan claves que se utilizan como índices. Debido a esta característica de los árboles-*B*, todos los caminos desde la raíz hasta cualquiera de los datos tienen la misma longitud. En la figura 6.52 presentamos un diagrama de un árbol-B^+ de orden 2.

Es de notar que los árboles-B^+ ocupan un poco más de espacio que los árboles-*B*, y esto ocurre al existir duplicidad en algunas claves. Sin embargo, esto es aceptable si el archivo se modifica frecuentemente, puesto que se evita la operación de reorganización del árbol que es tan costosa en los árboles-*B*.

Formalmente se define un árbol-B^+ de orden *d* de la siguiente manera:

1. Cada página, excepto la raíz, contiene *m* elementos, donde *m* es un valor entre *d* y 2*d*.
2. La raíz contiene de 1 a 2*d* elementos.
3. Cada página, excepto la raíz, tiene entre *d* + 1 y 2*d* + 1 descendientes.
4. La página raíz tiene al menos dos descendientes.
5. Las páginas hojas están todas al mismo nivel.
6. Toda la información, con las claves que las identifican, se encuentra en las páginas hoja.
7. Las claves almacenadas en las páginas raíz e interiores se utilizan como índices.

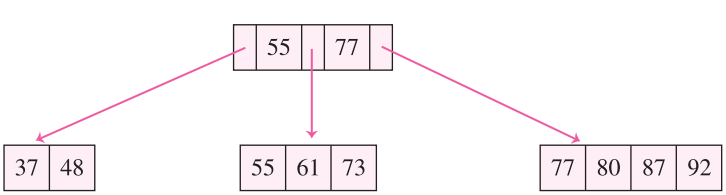


FIGURA 6.52
Árbol-B^+ de orden 2.

Búsqueda en árboles-B^+

La operación de **búsqueda en árboles-B^+** es similar a la operación de búsqueda en árboles-B. El proceso es simple, sin embargo puede suceder que al buscar una determinada clave la misma se encuentre en una página raíz o interior. En dicho caso no se debe detener el proceso porque en la página raíz o en las interiores sólo se almacenan claves que funcionan como índices. La búsqueda debe continuar en la página apuntada por la rama derecha de dicha clave.

Por ejemplo, al buscar la clave 55 en el árbol-B^+ de la figura 6.52 se advierte que ésta se encuentra en la página raíz. En este caso, se debe continuar el proceso de búsqueda en la página apuntada por la rama derecha de dicha clave.

Inserción en árboles-B^+

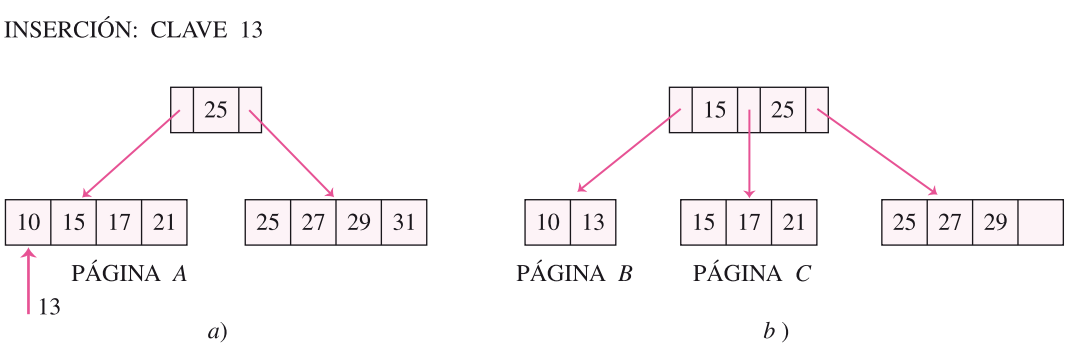
El proceso de **inserción en árboles-B^+** es relativamente simple, similar al proceso de inserción en árboles-B. La dificultad se presenta cuando se desea insertar una clave en una página que se encuentra llena ($m = 2d$). En este caso, la página afectada se divide en 2, distribuyéndose las $m + 1$ claves de la siguiente forma: "las d primeras claves en la página de la izquierda y las $d + 1$ restantes claves en la página de la derecha". Una copia de la clave del medio sube a la página antecesora.

En la figura 6.53 se muestran dos diagramas que ilustran cómo funciona este caso. Puede suceder que la página antecesora se desborde nuevamente, en dicho caso se debe repetir el proceso anterior. Es importante notar que el desbordamiento en una página que no es hoja no produce duplicidad de claves. El proceso de propagación puede llegar hasta la raíz, en cuyo caso la altura del árbol se puede incrementar en una unidad.

FIGURA 6.53

Inserción de la clave 13 en un árbol-B^+. *a*) Antes de insertar la clave. *b*) Después de insertarla.

Nota: Observe que la inserción de la clave 13 en la página *A* produce su división en dos páginas: *B* y *C*. Las *d* primeras claves se ubican en la página *B* (10 y 13). Las $d + 1$ claves restantes en la página *C* (15, 17 y 21). Una copia de la clave del medio (15) sube a la página antecesora.

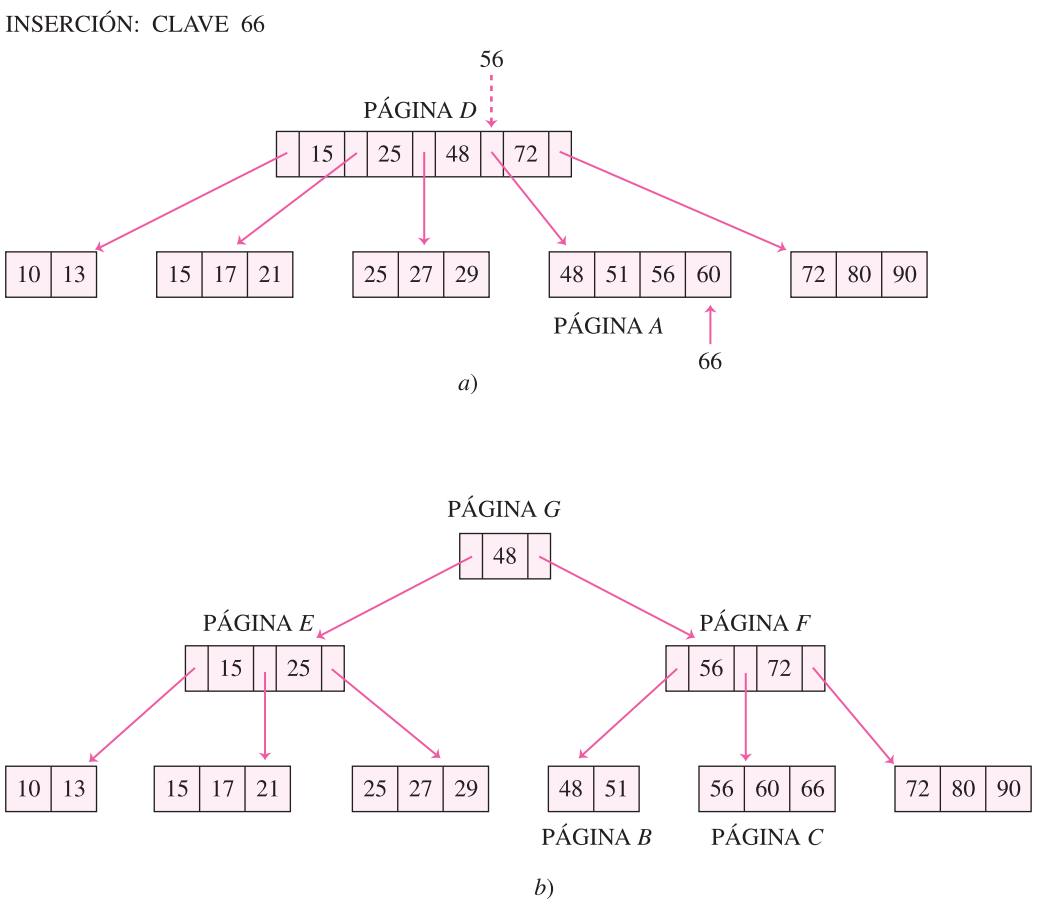


En la figura 6.54 se presentan dos diagramas que clarifican y resuelven este caso.

FIGURA 6.54

Inserción de la clave 66 en un árbol-B^+. *a*) Antes de insertar la clave. *b*) Después de insertarla.

Nota: La inserción de la clave 66 en la página *A* provocó la división de ésta en las páginas *B* y *C*. Sin embargo, al subir una copia de la clave del medio (56) se produce un nuevo desbordamiento en la página *D* que provoca su partición en las páginas *E* y *F*. La clave 48 forma ahora parte de la página *G* y representa la raíz del árbol. La altura del árbol se incrementa en una unidad.



Ejemplo 6.29

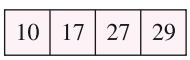
Supongamos que se desea insertar las siguientes claves en un árbol-B^+ de orden 2 que se encuentra vacío:

10 - 27 - 29 - 17 - 25 - 21 - 15 - 31 - 13 - 51 -
20 - 24 - 48 - 19- 60 - 35 - 66

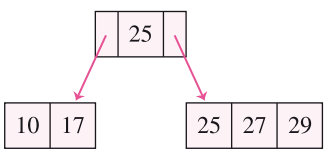
Los resultados parciales que ilustran el crecimiento del árbol se presentan en los diagramas correspondientes a la figura 6.55.

FIGURA 6.55
Inserciones en un árbol-B^+ de orden 2.

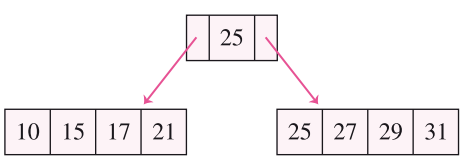
a) INSERCIÓN: CLAVES 10, 27, 29 Y 17



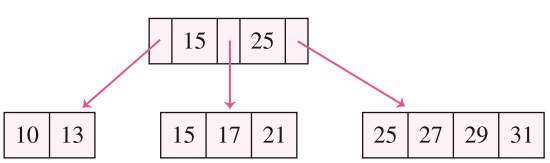
b) INSERCIÓN: CLAVE 25



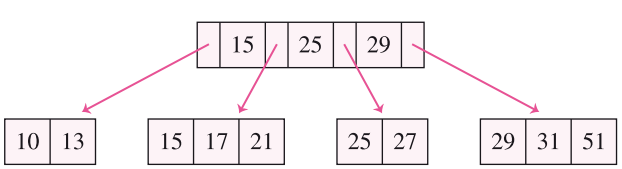
c) INSERCIÓN: CLAVES 21, 15 Y 31



d) INSERCIÓN: CLAVE 13



e) INSERCIÓN: CLAVE 51



f) INSERCIÓN: CLAVES 20, 24, 48 Y 19

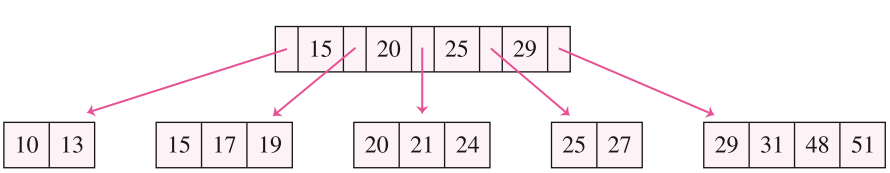
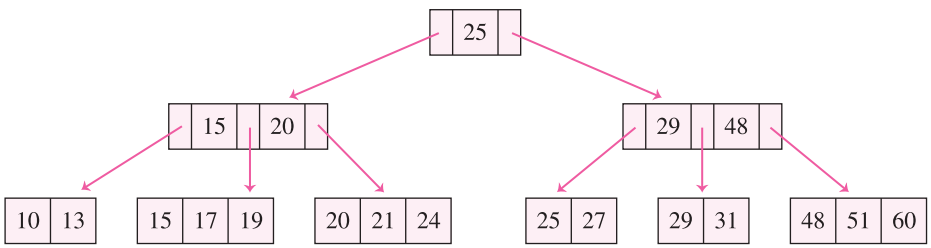
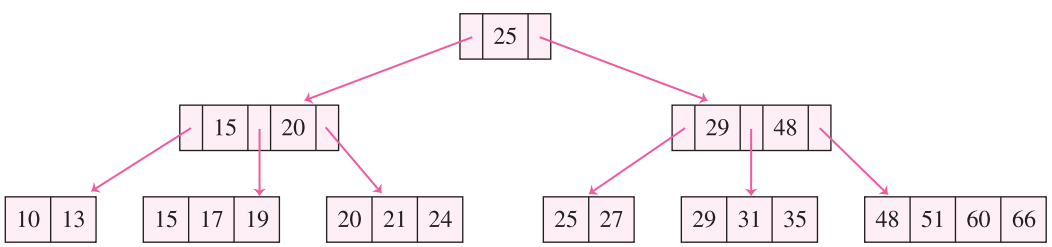


FIGURA 6.55
(*continuación*)

g) INSERCIÓN: CLAVE 60



h) INSERCIÓN: CLAVES 35 Y 66

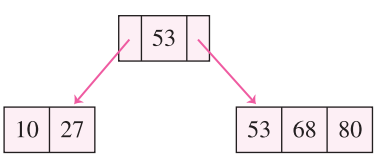


Ejemplo 6.30

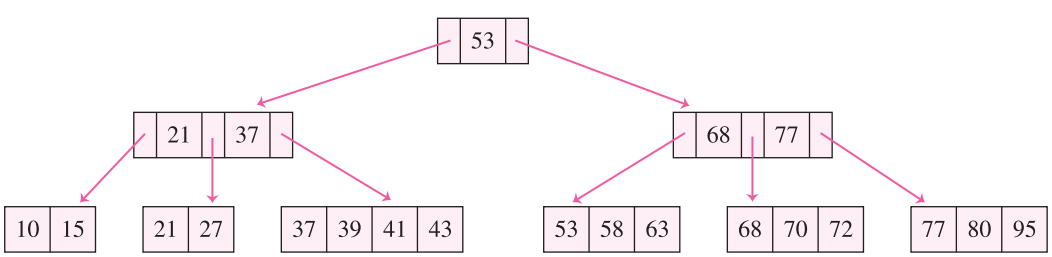
Dado como dato el árbol-B^+ de orden 2 de la figura 6.56*a*, verifique si el mismo queda igual al de la figura 6.56*b*, luego de insertar las siguientes claves:

43 - 21 - 77 - 58 - 63 - 15 - 37 - 41 - 72 - 39 - 95 - 70

FIGURA 6.56
Inserciones en un árbol-B^+ de orden 2. *a*) Antes de insertar las claves. *b*) Después de insertarlas.



a)



b)

Eliminación en árboles-B^+

La operación de **eliminación en árboles-B^+** es más simple que la operación de borrado en árboles-B. Esto ocurre porque las claves que se deben eliminar siempre se encuentran en las páginas hoja. En general se deben distinguir los siguientes casos:

1. *Si* al eliminar una clave *m* queda mayor o igual a *d*, *entonces* termina la operación de borrado. Las claves de las páginas raíz o internas no se modifican por más que sean una copia de la clave eliminada en las hojas. (Se presenta un ejemplo de este caso en la figura 6.57.)
2. *Si* al eliminar una clave *m* queda menor a *d*, *entonces* se debe realizar una redistribución de claves, tanto en el índice como en las páginas hojas. Cuando se cambia la estructura del árbol, se quitan aquellas claves que quedaron en los nodos interiores luego de haber eliminado su correspondiente información en los nodos hoja. Hay dos ejemplos que ilustran cómo funciona este caso en la figura 6.58.

Puede suceder que al eliminar una clave y al realizar una redistribución de las mismas, la altura del árbol disminuya en una unidad. En la figura 6.59 se presentan dos diagramas que corresponden a este caso.

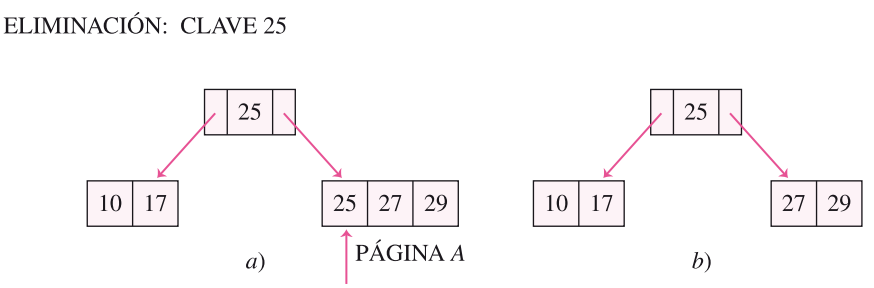


FIGURA 6.57
Eliminación de la clave 25 de un árbol-B^+ de orden 2. *a*) Antes de eliminar la clave. *b*) Después de eliminarla.

Nota: Al eliminar la clave 25 de la página *A*, la página raíz *B* que contiene como índice a la clave eliminada no se modifica.

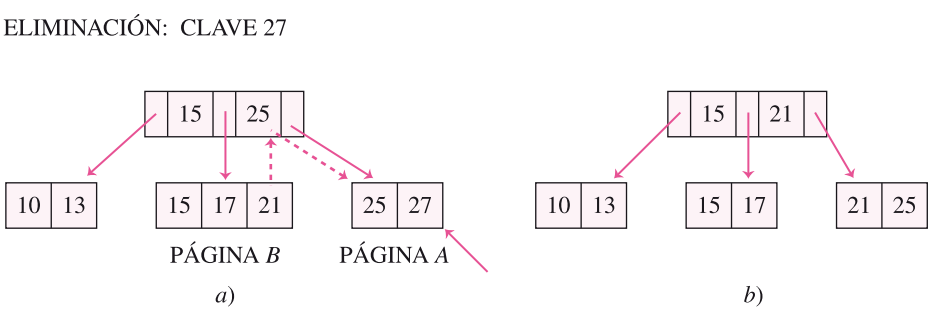


FIGURA 6.58
Eliminación de las claves 27 y 21 de un árbol-B^+ de orden 2. *a*) Antes de eliminar la clave 27. *b*) Después de eliminarla. *c*) Antes de eliminar la clave 21. *d*) Después de eliminarla.

FIGURA 6.58
(*continuación*)

Notas: Al eliminar la clave 27 de la página *A*, *m* queda menor a *d*, por lo que debe realizarse una redistribución de claves. Se toma la clave que se encuentra más a la derecha en la rama izquierda de 25 (21 de la página *B*). Se coloca dicha clave en la página *A* y una copia de la misma, como índice, en la página *C*.

Al eliminar la clave 21 de la página *A*, *m* queda menor a *d*, por lo que debe realizarse una redistribución de claves. Como no se puede tomar una clave de la página *B* puesto que *m* quedaría menor a *d*, entonces se realiza una fusión de las páginas *A* y *B*.

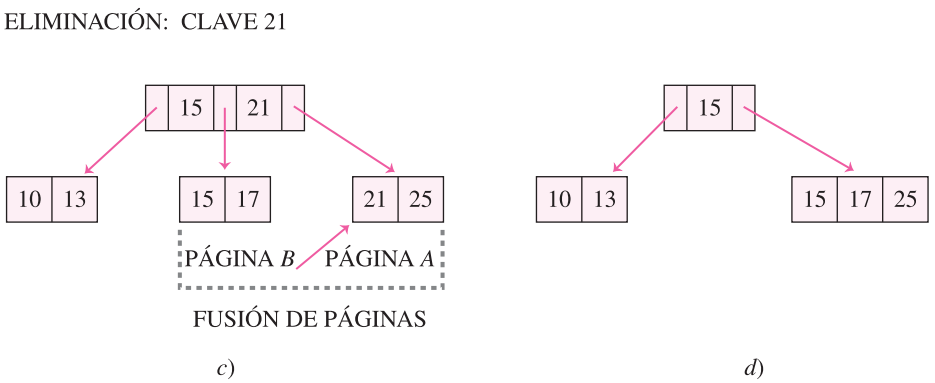
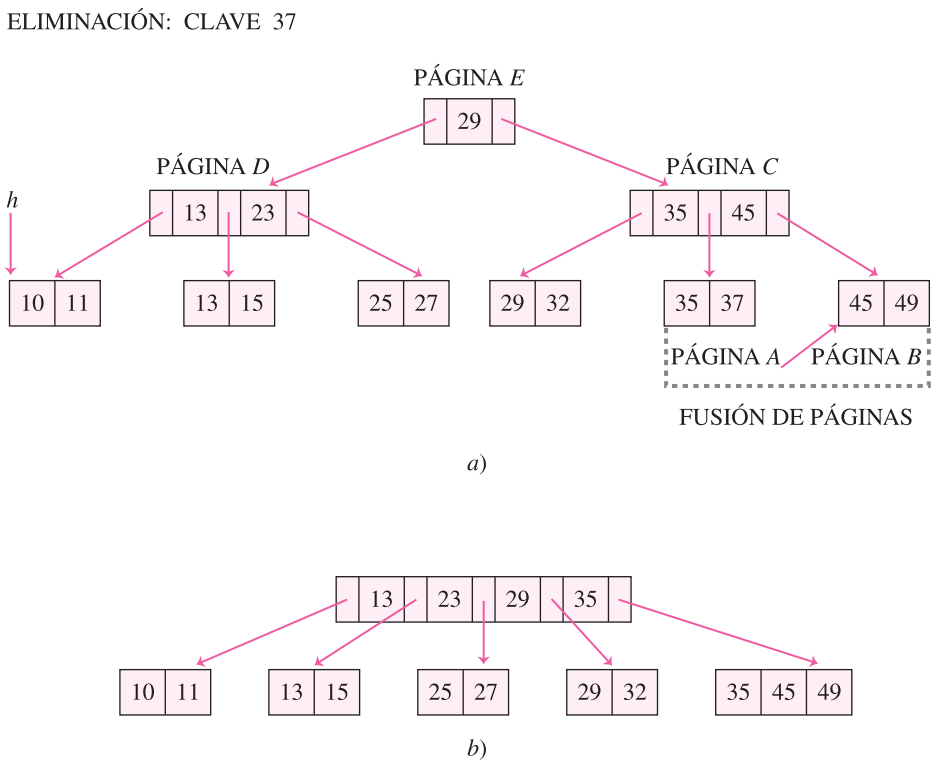


FIGURA 6.59
Eliminación de la clave 37 en un árbol-B^+ de orden 2. *a*) Antes de eliminar la clave. *b*) Después de eliminarla.

Nota: Al eliminar la clave 37 de la página *A*, *m* queda menor a *d*, por lo que debe realizarse una redistribución de claves. Como no puede tomarse una clave de la página *B*, puesto que *m* quedaría menor a *d*, entonces se realiza una fusión de las páginas *A* y *B*. Sin embargo, luego de esta fusión *m* queda menor a *d* en la página *C*, por lo que debe bajarse la clave 29 de la página *E* y realizarse una nueva fusión, ahora de las páginas *C* y *E*. La altura del árbol disminuye en una unidad.

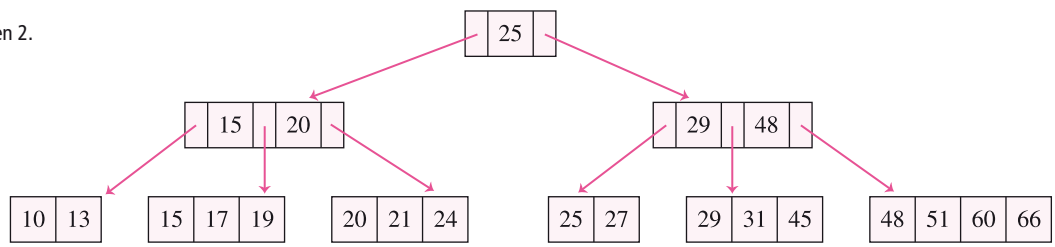


Ejemplo 6.31

Supongamos que se desea eliminar las siguientes claves del árbol-B^+ de orden 2 de la figura 6.60.

15 - 51 - 48 - 60 - 31 - 20 - 66 - 29 - 10 - 25 - 17 - 24

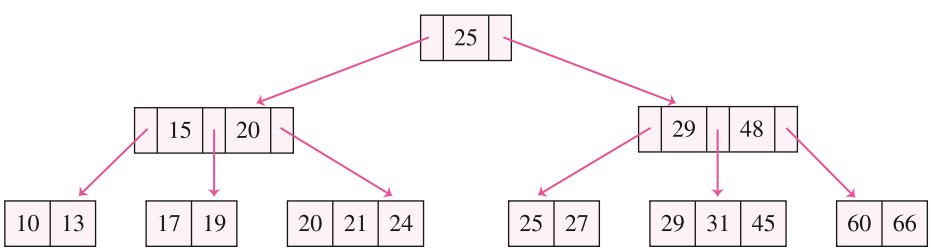
FIGURA 6.60
Árbol-B^+ de orden 2.



Los resultados parciales que ilustran cómo funciona el procedimiento se presentan en los diagramas de la figura 6.61.

FIGURA 6.61
Eliminaciones en un árbol-B^+ de orden 2.

a) ELIMINACIÓN: CLAVES 15, 51 Y 48



b) ELIMINACIÓN: CLAVE 60

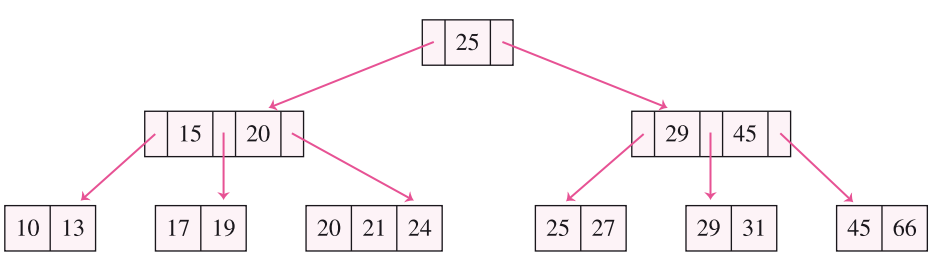
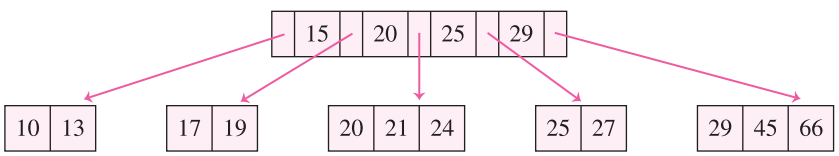
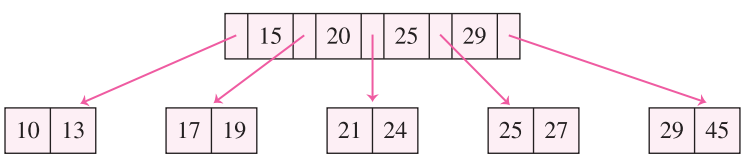


FIGURA 6.61
(*continuación*)

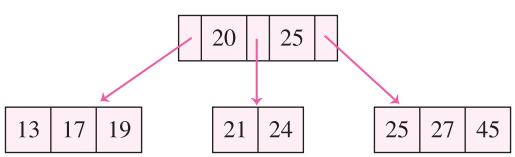
c) ELIMINACIÓN: CLAVE 31



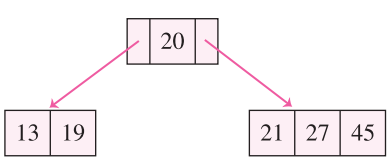
d) ELIMINACIÓN: CLAVES 20 Y 66



e) ELIMINACIÓN: CLAVES 29 y 10



f) ELIMINACIÓN: CLAVES 25, 17 y 24

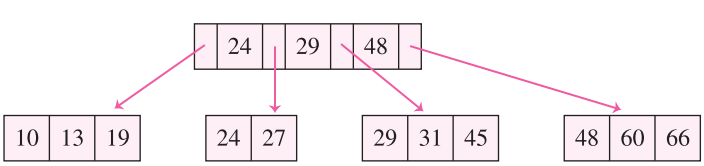


Ejemplo 6.32

Verifique si el árbol-B^+ de orden 2 de la figura 6.60 queda igual al de la figura 6.62, luego de eliminar las siguientes claves:

51 - 20 - 15 - 21 - 25 - 17

FIGURA 6.62
Árbol-B^+ de orden 2 luego de eliminar las claves 51, 20, 15, 21, 25 y 17.



6.5.3 Árboles 2-4

Los **árboles 2-4** son una variante de los árboles multicaminos. Éstos se caracterizan porque cada uno de sus nodos puede tener máximo 4 hijos y todos los nodos externos —hojas— están al mismo nivel. Es decir, en estos árboles se debe garantizar el tamaño y la altura de los mismos.

Como en el caso de los árboles multicaminos analizados en las secciones previas, las operaciones de inserción y eliminación pueden ocasionar, respectivamente, la partición o fusión de los nodos con el objeto de mantener las propiedades enunciadas. Debido a que se llevan a cabo de manera similar a lo presentado, se deja al lector el desarrollo de los correspondientes algoritmos.

6.6 LA CLASE *ÁRBOL*

La **clase *Árbol*** tiene como atributo a la raíz de la estructura y como métodos a todas las operaciones analizadas, según el tipo de árbol que se esté representando. Gráficamente una clase *Árbol* —para árboles binarios— se puede ver como se muestra en la figura 6.63. En este caso, los métodos permiten llevar a cabo todas las operaciones presentadas previamente: los tres tipos de recorridos, búsqueda, inserción y eliminación.

Se tiene acceso a los miembros de un objeto de la clase *Árbol* por medio de la notación de puntos. Asumiendo que la variable AROBJ es un objeto de la clase *Árbol*, previamente creado, se puede hacer:

AROBJ.Recorre_Preorden(argumento) para invocar el método que visita cada uno de los nodos del árbol siguiendo el recorrido preorden. En este método se requiere como argumento un puntero al nodo a visitar —la primera vez es la raíz—, ya que es un método recursivo.

AROBJ.Inserta(argumentos) para insertar un nuevo elemento en el árbol binario. En este método se requieren dos argumentos, uno para el nodo a visitar —la primera vez es la raíz— y otro para el dato a insertar.

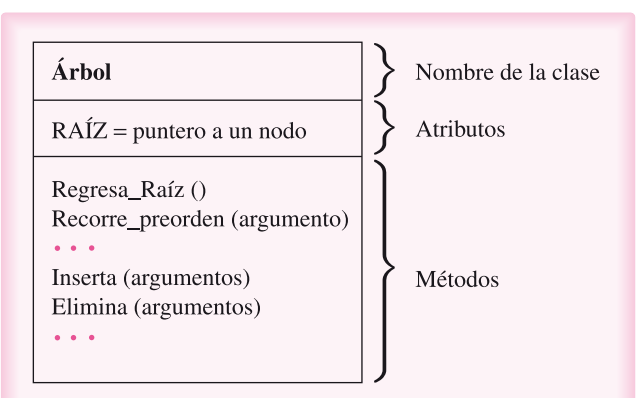
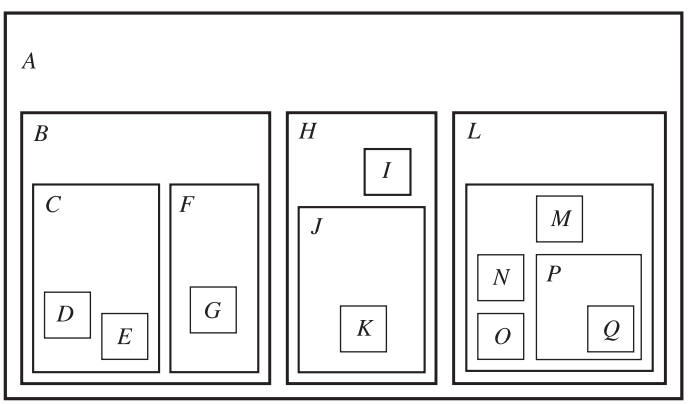


FIGURA 6.63
Clase *Árbol*.

EJERCICIOS

Árboles en general

1. Los árboles se pueden representar de diferentes formas. Dado el siguiente diagrama de Venn que corresponde a una estructura árbol, conviértalo a notación decimal de Dewey y notación indentada.



2. Dada la siguiente estructura del árbol representada como anidación de paréntesis:

$$(A(B(E(K), F), C(G(L, M(N))), D(H, I, (O, P, Q, R), J)))$$

Calcule lo siguiente:

a) Grado del árbol.
b) Grado del nodo *G*.
c) Altura del árbol.
d) Nodos terminales u hojas.
e) Nodos interiores.

3. Dada la siguiente estructura de árbol representada como notación decimal de Dewey:

1.*A*, 1.1.*B*, 1.1.1.*D*, 1.1.2.*E*, 1.1.2.1.*I*,1.1.2.2.J,
1.1.3.*F*, 1.1.4.*G*, 1.1.4.1.*K*, 1.1.4.1.1.*M*, 1.1.4.1.2.*N*,
1.2.*C*, 1.2.1.*H*, 1.2.1.1.*L*

Calcule las longitudes de camino interno y externo de dicho árbol.

4. Calcule cuál es el grado del nodo *T*, si *T* es padre del nodo *P* y éste tiene 4 hermanos.

Árboles binarios

5. Represente las siguientes expresiones algebraicas utilizando árboles binarios.

a) ((*C* * *Y*) ^ 0.8 + (*D*/*R*)) * *K*
b) *A* = *R* – (*C* + *L* / *D*) * *K*
c) *X* = ((*B* / *C* * *D*) ^ 0.5 + (*B* / *K* + *P*) ^ 0.8) ^ 0.5

6. Dados los siguientes árboles binarios representados como anidación de paréntesis:

a) (*K* (*B* (*A*, *F* (*D*)), *W* (*M* (*L*, *O* (*P*)), *Z*)))
b) (25(20(10(8)), 23(21)), 90(80(62(47(32))), 100))

Escriba los recorridos preorden, inorden y posorden de cada uno de ellos.

7. ¿Cuál es el máximo número de nodos de un árbol binario de altura *h*?

8. ¿Cuántos árboles binarios **distintos** se puede tener con 4 nodos? ¿Y cuántos con 7?

9. ¿Cuántos árboles binarios **similares** se puede tener con 4 nodos? ¿Y cuántos con 7?

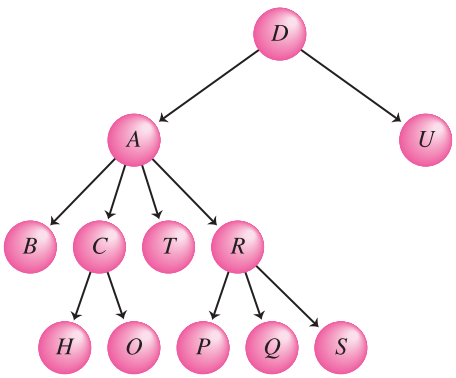
10. Dadas las siguientes secuencias de nodos obtenidas por los recorridos preorden, inorden y posorden, dibuje el correspondiente árbol binario.

- Preorden: *P* - *R* - *A* - *C* - *H* - *T* - *O* - *M*
- Inorden: *A* - *R* - *H* - *C* - *P* - *O* - *T* - *M*
- Postorden: *A* - *H* - *C* - *R* - *O* - *M* - *T* - *P*

Representación de árboles generales como árboles binarios

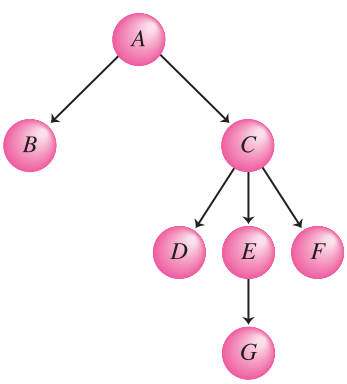
11. Dados los siguientes árboles generales, uno representado en forma de grafo, inciso *a*, y otro representado como anidación de paréntesis, inciso *b*), conviértalos a árboles binarios.

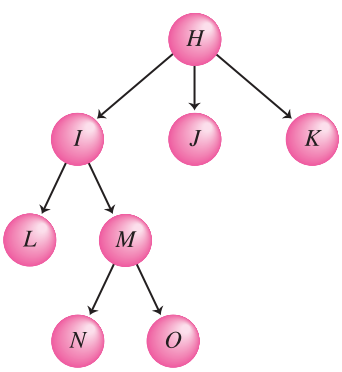
a)

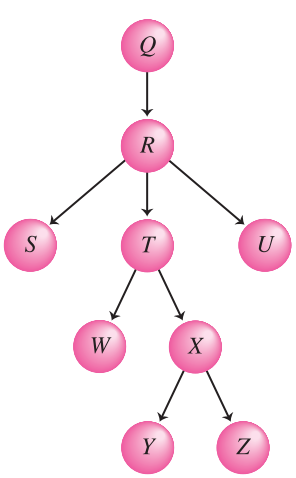


b) (*A* (*B* (*E*, *F* (*K*)), *C* (*G* (*L*, *M* (*Q*, *R*), *N*)), *D* (*H*, *I* (*O* (*S*), *P*))))

12. Dado el siguiente bosque, conviértalo en árbol binario.





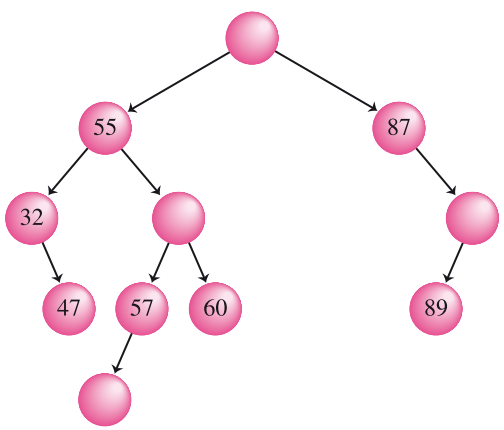


Árboles binarios de búsqueda

13. Dadas las siguientes claves que representan los signos del zodiaco, construya un árbol binario de búsqueda.

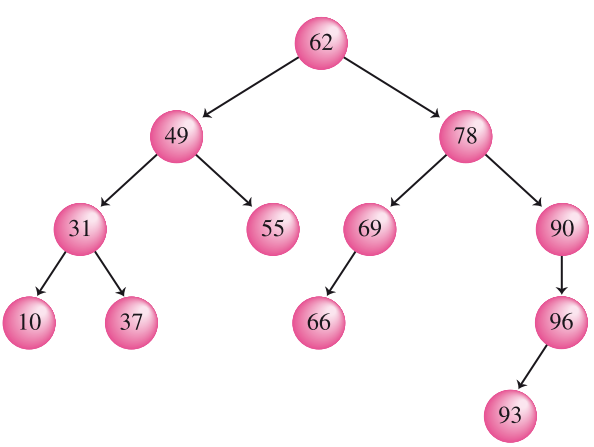
piscis - acuario - capricornio - cáncer - sagitario - virgo - leo -
escorpión - libra - géminis - aries - tauro

14. Dado el siguiente árbol binario de búsqueda, complete los nodos en blanco de tal forma que no se violen los principios que definen justamente un árbol binario de búsqueda.



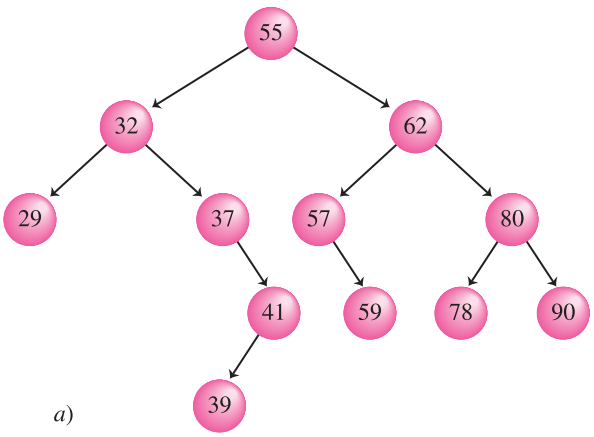
15. Dado el siguiente árbol binario de búsqueda, elimine las claves

49 - 37 - 62 - 90 - 78

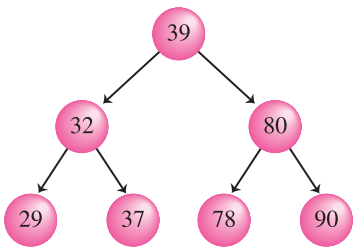


16. Verifique si el árbol binario de búsqueda del diagrama del inciso *a*) queda igual al del diagrama del inciso *b*), luego de eliminar las claves

55 - 62 - 57 - 59 - 41



a)



b)

Ejercicios de programación en árboles binarios

17. Escriba un programa que calcule e imprima cuántos nodos tiene un árbol binario.

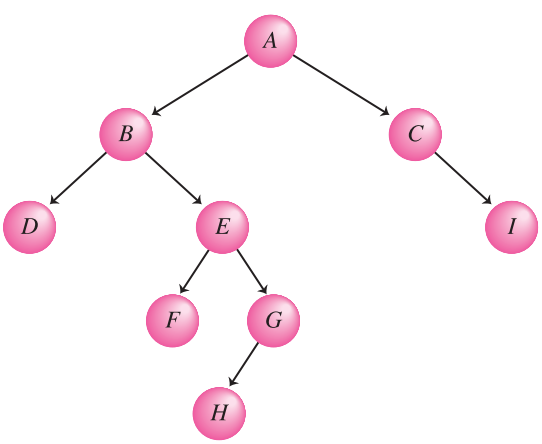
18. Escriba un programa que calcule e imprima el total de nodos internos que tiene un árbol binario.

19. Considerando que un árbol binario almacena números enteros, encuentre e imprima el máximo valor guardado en el árbol y el promedio de los mismos.

20. Escriba un procedimiento que realice lo siguiente:

a) Imprima la información almacenada en las hojas de un árbol binario.
b) Imprima la información almacenada en los nodos internos de un árbol binario.

21. Dado el siguiente árbol binario:



Escriba un programa que imprima los nodos del mismo de la siguiente forma:

- A - - B - - - D - - - E - - - - F - - - - G - - - - - H - - C - - - I

22. Dado el árbol binarlo del ejercicio 21, escriba un programa que imprima los nodos del mismo de la siguiente forma:

- A - - B - - C - - - D - - - E - - - I - - - - F - - - - G - - - - - H

23. Escriba un procedimiento que visite los nodos de un árbol binario de la siguiente forma:

- Raíz
- Rama derecha
- Rama izquierda

24. Escriba tres procedimientos que efectúen los recorridos en preorden, inorden y posorden en forma iterativa en lugar de recursiva, para lo cual se puede apoyar en una pila.

25. Escriba un procedimiento que elimine todas las hojas de un árbol binario.

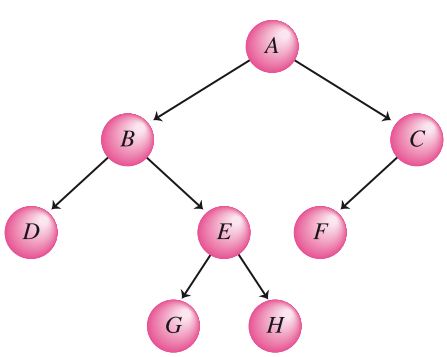
26. Escriba un programa que cargue los nodos de un árbol binario en un arreglo unidimensional. Cuide que se mantenga la relación padre-hijo entre los nodos.

27. Escriba una función que determine si dos árboles binarios son **similares**.

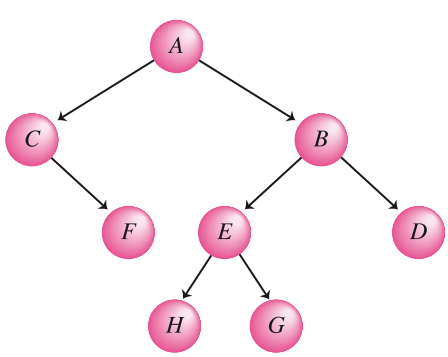
28. Escriba una función que determine si dos árboles binarios son **equivalentes**.

29. Escriba un procedimiento que intercambie los subárboles izquierdo y derecho de un árbol binario. Es de observar que este intercambio se debe realizar para todo nodo del árbol.

Ejemplo: Dado el árbol binario del diagrama del inciso *a*), el intercambio de ramas produce el árbol del diagrama del inciso *b*).

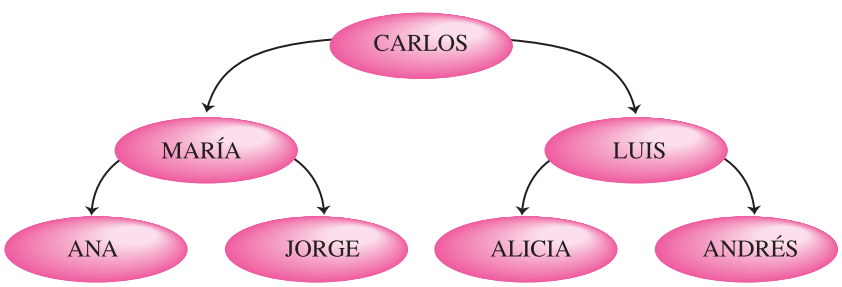


a)



b)

30. Se tiene almacenada toda la ascendencia de Carlos en un árbol binario. Se ha seguido el siguiente criterio para Carlos y todos sus progenitores: en la rama izquierda se ha guardado el nombre de la madre y en la rama derecha el nombre del padre. Observe la figura que se muestra a continuación.



Escriba un subprograma que imprima el nombre de todos los **progenitores femeninos** de Carlos.

31. Retome el problema anterior. Agregue una función que pueda insertar al árbol genealógico de Carlos tanto ascendientes femeninos como masculinos.

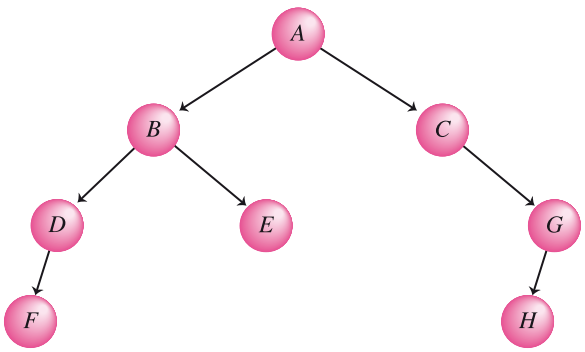
32. Defina la clase *Árbol binario* utilizando algún lenguaje de programación orientado a objetos, tomando como base para programar los métodos los algoritmos estudiados en este capítulo.

33. Retome el problema anterior. Agregue a la clase un método que muestre el contenido de un nodo.

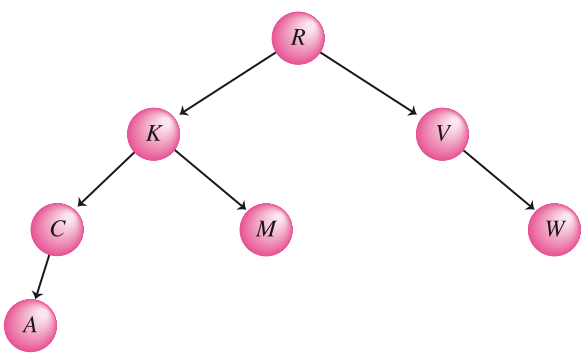
34. Escriba un programa de aplicación que dados dos objetos de la clase *Árbol binario*, previamente definida, imprima un mensaje adecuado según los mismos sean equivalentes o no. Determine si requiere definir nuevos métodos a la clase.

Árboles balanceados

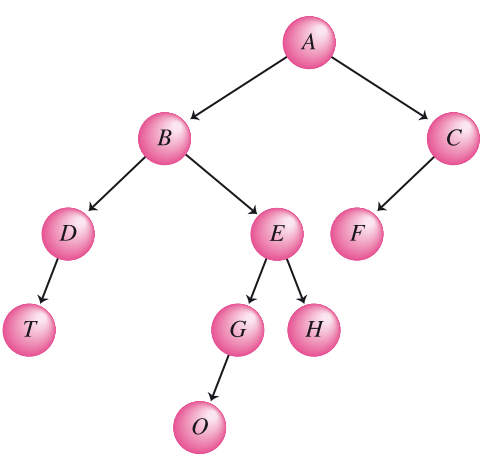
35. Determine si los siguientes árboles binarios son árboles balanceados.



a)



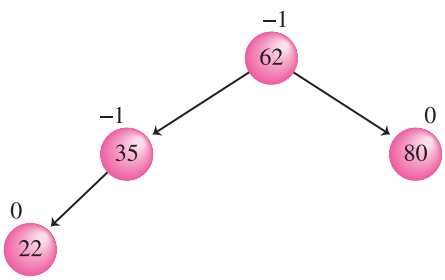
b)



c)

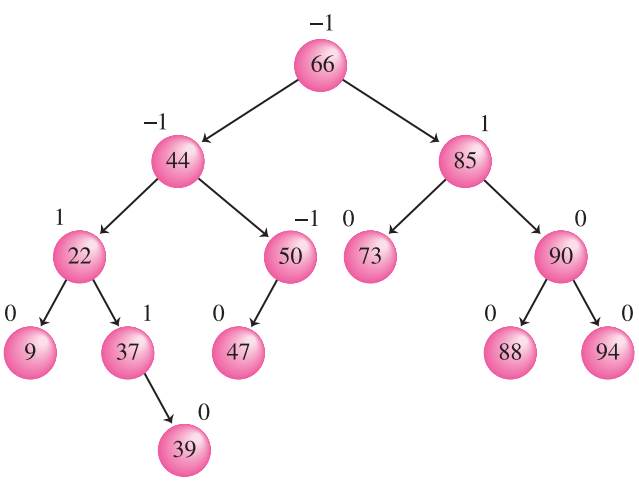
36. Calcule cuál es el máximo número de nodos de un árbol balanceado de altura 13. ¿Cuál es el mínimo?

37. Inserte las claves 10 - 47 - 38 - 06 - 55 - 90 - 49 en el árbol balanceado que se da a continuación.

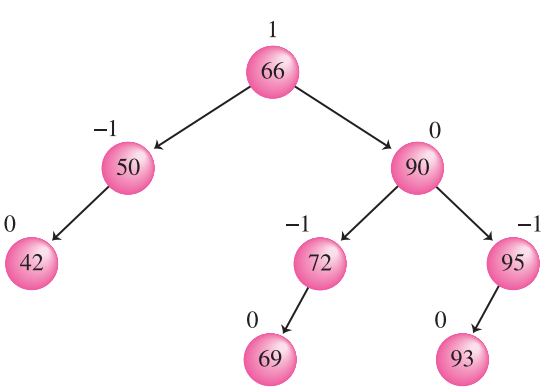


38. Elimine las siguientes claves del árbol balanceado del siguiente diagrama:

73 - 66 - 50 - 47 - 39 - 94



39. Escriba las instrucciones necesarias para equilibrar el árbol balanceado del siguiente diagrama, luego de eliminar la clave 50.



Árboles-*B* y árboles-*B*+

40. ¿Cuál es el número de claves (m) que puede tener como máximo un árbol-*B* de orden 50 y tres niveles? ¿Cuál el de un árbol-*B*+?

41. ¿Cuál es el número más pequeño de claves que, al ser insertadas, provocaría que un árbol-*B* de orden 50 tuviera tres niveles?

42. ¿Cuál es el número más pequeño de claves que, al ser insertadas, provocaría que un árbol-*B* de orden 100 tuviera cuatro niveles?

43. Un árbol-*B* de orden 100 y tres niveles tiene 5 800 000 claves. ¿Cuántas claves se podrían eliminar del árbol sin que éste tenga que disminuir su altura?

44. Realice los tres ejercicios anteriores, pero ahora aplicados a árboles-*B*+.

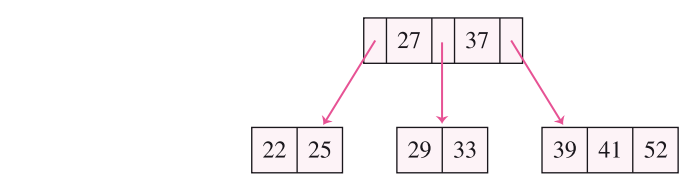
45. Inserte las siguientes claves:

08 - 77 - 36 - 68 - 90 - 37 - 41 - 52 - 57 - 13 - 30 - 28 - 24 - 86

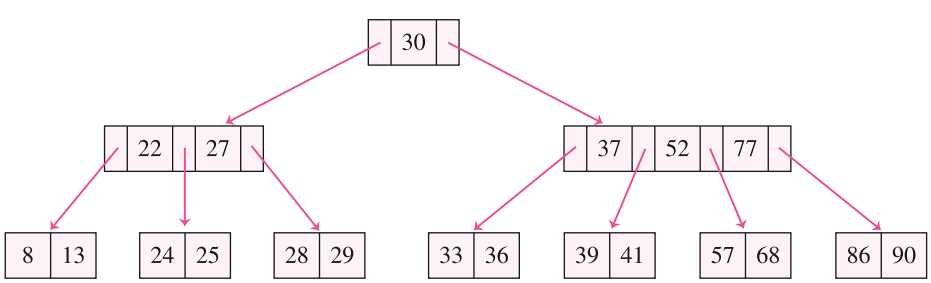
a) En un árbol-*B* de orden 2 que se encuentra vacío.
b) En un árbol-*B*+ de orden 2 que se encuentra vacío.

46. Verifique si el árbol-*B* de orden 2 del diagrama del inciso *a*) queda igual al del diagrama del inciso *b*), luego de insertar las siguientes claves:

90 - 08 - 77 - 68 - 36 - 30 - 13 - 57 - 24 - 28 - 86



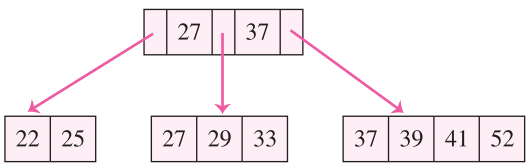
a)



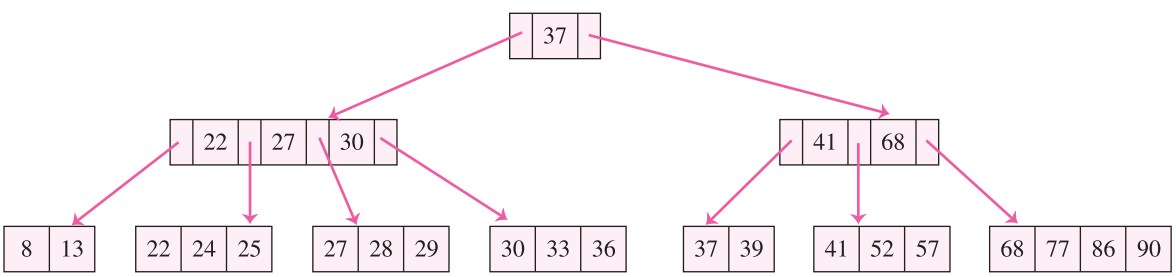
b)

47. Verifique si el árbol-B^+ de orden 2 del diagrama del inciso *a*) queda igual al del diagrama del inciso *b*), luego de insertar las siguientes claves:

90 - 08 - 77 - 68 - 36 - 30 - 13 - 57 - 24 - 28 - 86



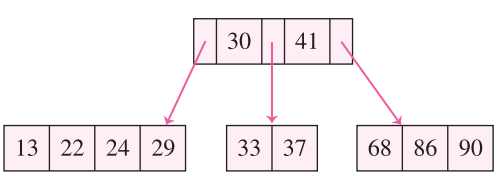
a)



b)

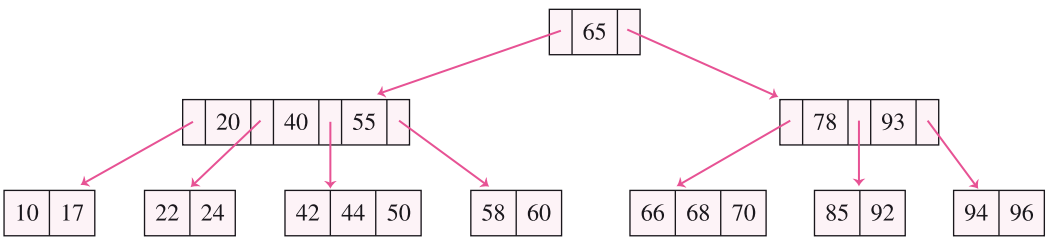
48. Verifique si el árbol-*B* de orden 2 del diagrama del inciso *b*) del ejercicio 46 queda igual al árbol del siguiente diagrama, luego de eliminar las claves:

52 - 77 - 36 - 08 - 57 - 39 - 28 - 25 - 27

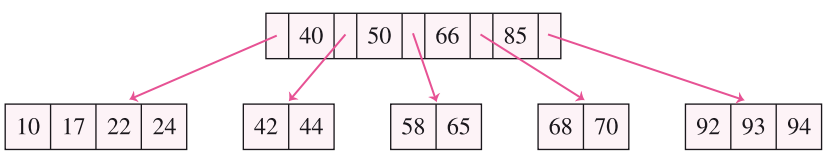


49. Verifique si el árbol *B* que se presenta en el inciso *a*) queda igual al árbol del diagrama del inciso *b*), luego de eliminar las siguientes claves:

96 - 55 - 60 - 20 - 78



a)



b)

Ejercicios de programación de árboles-*B* y árboles-*B*+

50. Escriba los subprogramas de inserción y eliminación en árboles-*B*.

51. Escriba los subprogramas de inserción y eliminación en árboles-*B*+.

Capítulo 7

GRÁFICAS

7.1 INTRODUCCIÓN

En el capítulo anterior se estudiaron las estructuras de datos tipo árboles, en donde cada nodo o elemento puede tener como máximo un nodo que le precede o raíz. Sin embargo, en la práctica existen problemas o situaciones en que la información que se debe almacenar no corresponde con una estructura de este tipo. Para estos problemas se necesita de una estructura en la cual se puedan representar otras relaciones entre los datos o componentes de la misma. Dedicaremos este capítulo al estudio de las gráficas.

Las **gráficas** son estructuras de datos no lineales donde cada componente puede tener uno o más predecesores y sucesores. En una gráfica se distinguen dos elementos: los nodos, mejor conocidos como **vértices**, y los arcos, llamados **aristas**, que conectan un vértice con otro. Los vértices almacenan información y las aristas representan relaciones entre dicha información.

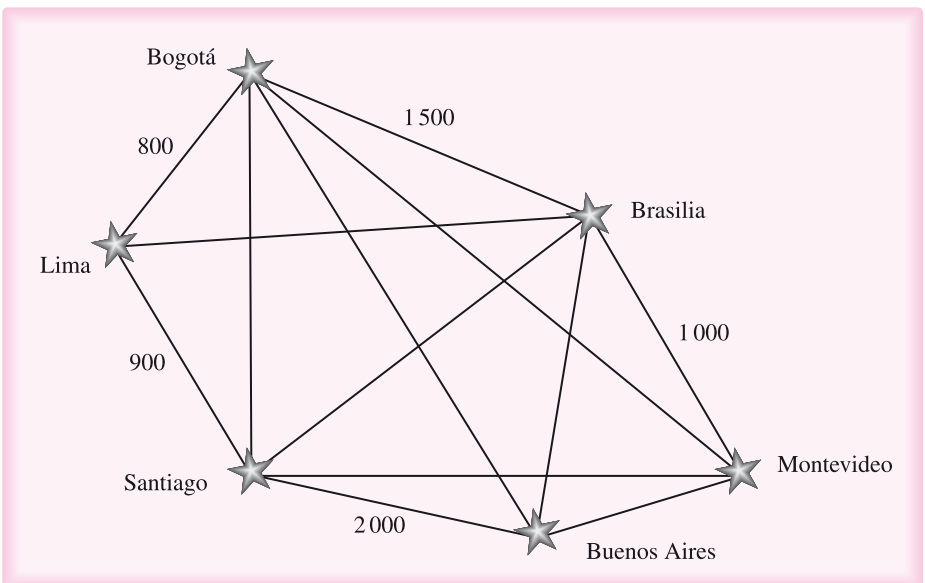
Estas estructuras tienen aplicaciones en diferentes dominios, entre ellos transporte —terrestre, aéreo y marítimo—, redes de computadoras, mapas —ubicación geográfica de varias ciudades—, asignación de tareas, etc. Considere, por ejemplo, la gráfica de la figura 7.1, donde se observan algunas de las principales capitales sudamericanas y la conexión entre ellas. En este caso los vértices representan a las ciudades, mientras que las aristas a las carreteras o algún otro medio de conexión entre ellas. Algunas aristas están etiquetadas, el valor que aparece en ellas constituye la distancia que existe entre las ciudades. En general, una etiqueta en la arista que une, por ejemplo, los vértices i y j se usa para representar el costo de ir del vértice i al vértice j.

En la figura 7.2 se presentan dos ejemplos de gráficas. La primera a) tiene cuatro vértices (a, b, c, d) y cinco aristas ((a, b), (b, c), (c, d), (d, a), (b, d)); mientras que la segunda b) tiene seis vértices (a, b, c, d, e, f) y seis aristas ((a, b), (b, c), (c, d), (d, a), (d, e), (e, f)).

7.2 DEFINICIÓN DE GRÁFICAS

Una gráfica G consta de dos conjuntos: $V(G)$ y $A(G)$. El primero lo integran elementos llamados nodos o vértices; el segundo, arcos o aristas. Por lo tanto, podemos denotar una gráfica G como:

FIGURA 7.1
Ejemplo de gráfica.

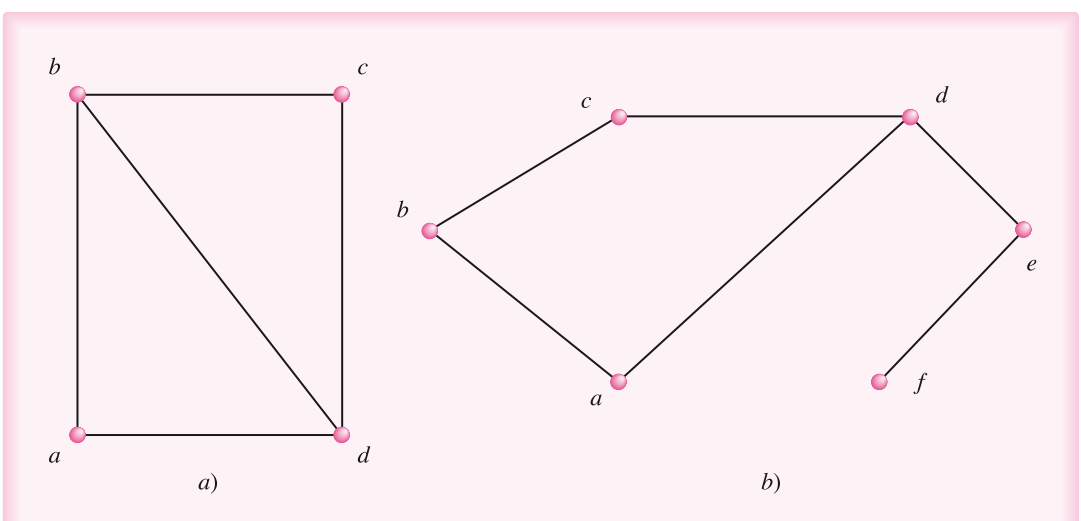


$$G = (V, A)$$

Donde V representa el conjunto de vértices de G y A el conjunto de aristas de G. Si no se hace ninguna especificación, los conjuntos V y A son finitos.

Cada arista está identificada por un único par de nodos del conjunto de vértices, que puede o no estar ordenado. Una arista que va del vértice u al v se denota mediante

FIGURA 7. 2
Elementos de una gráfica.



la expresión $a = (u, v)$, donde u y v son vértices adyacentes y los extremos de a. En este caso, u y v están conectados por a y se dice que a es incidente en u y v.

7.3 CONCEPTOS BÁSICOS DE GRÁFICAS

A continuación se presentan algunos de los conceptos más importantes relacionados con la teoría de gráficas.

- **Grado de un vértice**: El grado de un vértice v, escrito como grado(v), es el número de aristas que contienen a v; es decir, que tienen a v como extremo. Si el grado(v) = 0 (v no tiene aristas), se dice que v es un nodo aislado.
- **Lazo** o **bucle**: Un lazo o bucle es una arista que conecta a un vértice consigo mismo; es decir, $a = (u, u)$.
- **Camino**: Un camino P de longitud n se define como la secuencia de n vértices que se debe seguir para llegar del vértice v_1 —origen— al vértice v_n —destino—.

  $$P = (v_1, ..., v_n)$$

  De tal modo que v_i es adyacente a v_{i+1} para $i = 1, 2,..., n-1$.

- **Camino cerrado**: El camino P es cerrado si el primero y último vértices son iguales; es decir, si $v_1 = v_n$.
- **Camino simple**: El camino es simple si todos sus nodos son distintos, con excepción del primero y del último, que pueden ser iguales; es decir, P es simple si v_1, v_2, ..., son distintos.
- **Ciclo**: Un ciclo es un camino simple cerrado de longitud 3 o mayor. Un ciclo de longitud k se llama k-ciclo.
- **Gráfica conexa**: Se dice que una gráfica es conexa si existe un camino simple entre cualesquiera dos de sus nodos.
- **Gráfica árbol**: Se dice que una gráfica G es del tipo árbol o árbol libre si G es una gráfica conexa sin ciclos.
- **Gráfica completa**: Se dice que una gráfica es completa si cada vértice v de G es adyacente a todos los demás vértices de G. Una gráfica completa de n vértices tendrá $n(n-1)/2$ aristas.
- **Gráfica etiquetada**: Se dice que una gráfica G está etiquetada si sus aristas tienen asignado un valor. Es decir, si cada arista a tiene un valor numérico no negativo $c(a)$, llamado costo, peso o longitud de a, entonces G tiene peso o está etiquetada. En este caso, cada camino P de G tendrá asociado un peso o longitud que será la suma de los pesos de las aristas que forman el camino P.
- **Multigráfica**: Una gráfica se denomina multigráfica si al menos dos de sus vértices están conectados entre sí por medio de dos aristas. En este caso, las aristas reciben el nombre de **aristas múltiples** o paralelas.
- **Subgráfica**: Dada la gráfica $G = (V, A)$, $G' = (V', A')$ se denomina subgráfica de G si $V' \neq \phi$, $V' \subseteq V$ y $A' \subseteq A$, donde cada arista de A' es incidente con vértices de V'.

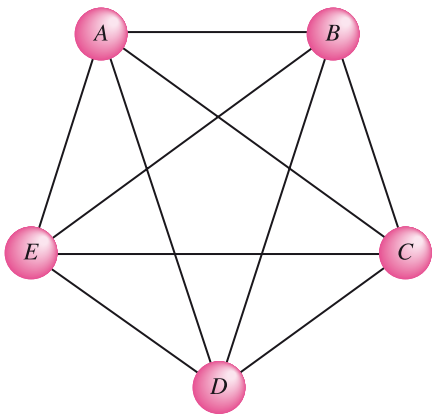


FIGURA 7.3
Conceptos de gráficas.

Luego de observar la figura 7.3 se pueden realizar las siguientes afirmaciones:

a) Todos los vértices tienen grado 4.
b) Un camino *P* para llegar del nodo *A* al *D* puede ser *A*-*B*-*C*-*D*. Otros pueden ser *A*-*E*-*D* o *A*-*D*.
c) El camino *A*-*C*-*D*-*A* es un camino cerrado, el *A*-*C*-*D* no lo es.
d) El camino *A*-*C*-*D*-*A* es un camino simple, el *A*-*C*-*B*-*D*-*C* no lo es.
e) El camino *A*-*C*-*D*-*A* es un ciclo.
f) Es una gráfica conexa, pues todos los nodos tienen al menos un camino a otro nodo.
g) Es una gráfica completa, pues todos los nodos se conectan con los demás.

Luego de observar la figura 7.4 se pueden realizar las siguientes afirmaciones:

a) En la gráfica de la figura 7.4*a,* existe un lazo o bucle en el vértice *d*. Es decir, $a = (d, d)$.
b) La gráfica de la figura 7.4*b,* es una multigráfica, ya que hay dos aristas que unen los vértices *c* y *d*. Es decir, las aristas $a_1 = (c, d)$ y $a_2 = (c, d)$ son aristas múltiples o aristas paralelas.

En las siguientes secciones se describen dos tipos de gráficas: dirigidas y no dirigidas.

7.4 GRÁFICAS DIRIGIDAS

En esta sección se tratará un tipo especial de gráficas llamadas **gráficas dirigidas**. Además de su definición y su representación, se presentarán los principales algoritmos usados para el cálculo de caminos. Es importante mencionar que existe gran cantidad de problemas de la vida real que son muy difíciles de resolver, y que, sin embargo, se podrían resolver fácilmente si se modelaran con gráficas y luego se aplicaran algunos métodos que funcionan sobre ellas.

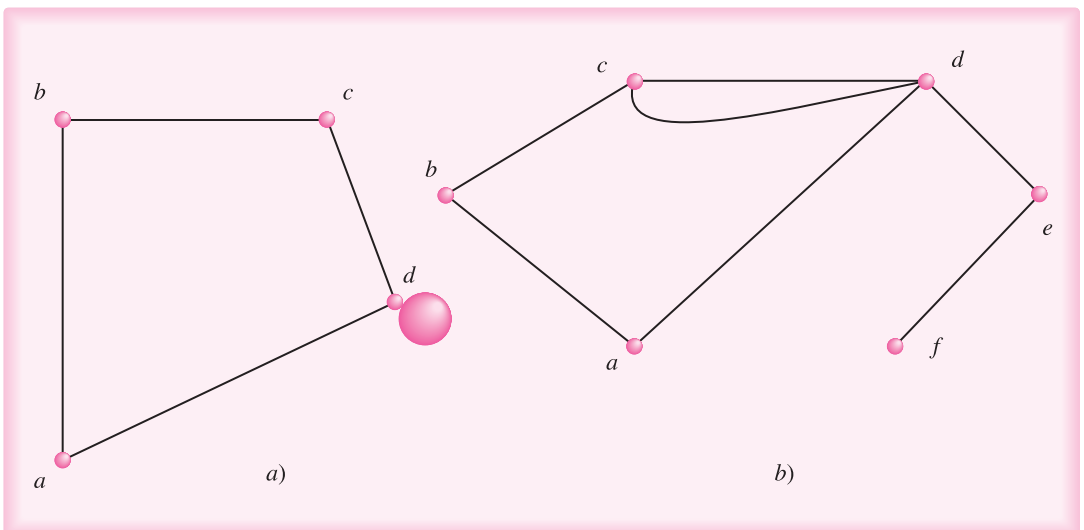


FIGURA 7.4
Otros conceptos de gráficas.

Cada vez que solucionamos un problema, en realidad estamos encontrando la solución a un modelo del problema. Todos los modelos son simplificaciones, de alguna forma, del mundo real, de otra manera serían extremadamente complejos y difíciles de manejar.

El proceso de solución de un problema consta de dos etapas importantes: el desarrollo de un modelo de un problema y el uso del modelo para generar la solución. La solución, finalmente, es en términos del modelo. Si el nuestro tiene un alto grado de fidelidad y el método que empleamos es adecuado, entonces nuestra solución será buena. Por el contrario, si nuestro modelo no representa fidedignamente al problema, entonces los resultados no serán satisfactorios. La teoría de gráficas proporciona los conceptos para modelar muchos problemas de la vida real, utilizando justamente gráficas. Luego, existen muy buenos métodos que se pueden aplicar a estas gráficas, que proporcionarán como resultado final la solución del problema inicial.

Las gráficas dirigidas se caracterizan porque sus aristas tienen asociada una dirección; es decir, son pares ordenados. Los vértices se utilizan para representar información, mientras que las aristas representan una relación con dirección o jerarquía entre aquéllos. Una posible aplicación de este tipo de gráficas puede ser la representación de ciudades en los vértices, y la duración de los vuelos en las aristas, asumiendo que el tiempo necesario para ir de la ciudad $C1$ a la ciudad $C2$ no es el mismo —teniendo en cuenta razones como los vientos— que el requerido para ir de la ciudad $C2$ a la ciudad $C1$. A continuación se define formalmente el concepto de gráfica dirigida.

Una **gráfica dirigida** G, también llamada **digráfica**, se caracteriza porque cada arista a tiene una dirección asignada; es decir, cada arista está asociada a un par ordenado (u, v) de vértices de G. Una arista dirigida $a = (u, v)$ se llama arco, y generalmente se expresa como $u \rightarrow v$. Para las aristas de las digráficas se aplica la siguiente terminología:

FIGURA 7.5
Representación de una arista dirigida.

a) *a* empieza en *u* y termina en *v*.
b) *u* es el origen o punto inicial de *a*, y *v* es el destino o punto terminal de *a*.
c) *u* es un predecesor de *v* y *v* es un sucesor o vecino de *u*.
d) *u* es adyacente hacia *v* y *v* es adyacente desde *u*.

En la figura 7.5 se presenta un ejemplo de una arista de una digráfica. Observe que el arco que une a los dos vértices tiene dirección, indicada por medio de la flecha.

7.4.1 Representación de gráficas dirigidas

Las digráficas son estructuras de datos abstractas; por lo tanto, los lenguajes de programación no cuentan con herramientas que permitan su manejo. Para su representación se requiere usar otras estructuras de datos. Existen varias opciones para realizar esto último; la elección de la más adecuada depende del uso que se le vaya a dar a la información almacenada en los vértices y en las aristas. Las representaciones más utilizadas son las matrices y listas de adyacencia, que se describen a continuación. Es importante señalar que algunos lenguajes de programación, como LISP o SCHEME, no utilizan arreglos bidimensionales —matrices— como estructuras de datos estándar; por lo tanto, se usan árboles o listas para la representación de digráficas.

Matriz de adyacencia

Una **matriz de adyacencia** es una matriz booleana, de orden *n*, donde *n* indica el número de vértices de G. Los renglones y columnas de la matriz representan a los vértices y su contenido la existencia o no de arcos entre ellos. Por lo tanto, cada elemento *i*, *j* de la matriz almacena un **1** o un **0**, dependiendo de si existe o no un arco entre los vértices *i* y *j*.

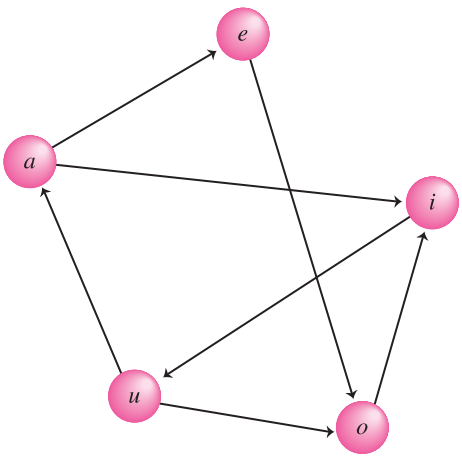
Para generar la matriz de adyacencia correspondiente a una digráfica se le da un orden arbitrario a sus vértices, y se asigna a los renglones y a las columnas de una matriz el mismo orden. Un elemento de la matriz será 1 si los vértices correspondientes al renglón y a la columna están unidos por una arista —son adyacentes—, y 0 en caso contrario.

Si $G = (V, A)$ y $V = \{1, 2, 3, ..., n\}$, la matriz de adyacencia M que representa a G tiene $n \times n$ elementos donde $M[i, j]$(con $1 \leq i \leq n$ y $1 \leq j \leq n$) es 1 sólo si existe un arco que vaya del nodo i al j, y es 0 en otro caso.

Una ventaja de las matrices de adyacencia es que el tiempo de acceso al elemento requerido es independiente del tamaño de V y A. El tiempo de búsqueda es del orden de $0(n)$. Sin embargo, su principal desventaja es que requiere un espacio de almacenamiento de n^2 posiciones, aunque el número de arcos de G no sobrepase ese número. La matriz de adyacencia es útil en los algoritmos diseñados para conocer si existe una arista entre dos nodos dados.

En las figuras 7.6 y 7.7 se presentan dos ejemplos de gráficas dirigidas con sus respectivas representaciones por medio de matrices de adyacencia.

FIGURA 7.6
Ejemplo de representación de gráficas. *a*) Gráfica dirigida. *b*) Matriz de adyacencia de la gráfica dirigida.



a)

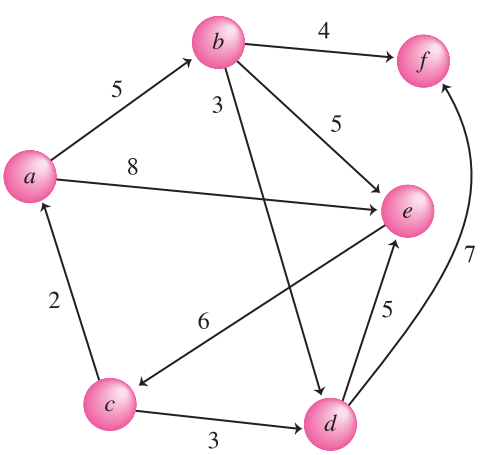
	a	*e*	*i*	*o*	*u*
a	0	1	1	0	0
e	0	0	0	1	0
i	0	0	0	0	1
o	0	0	1	0	0
u	1	0	0	1	0

b)

Una variante de la matriz de adyacencia es la matriz de adyacencia etiquetada, en donde $M[i, j]$ representa la etiqueta o costo asociado al arco. Si la arista no existe, entonces el valor de $M[i, j]$ será cero. Estas matrices también se denominan **matrices de costos** o **de distancias**. En la figura 7.7 se presenta un ejemplo de este caso.

Como ya se mencionó, la principal desventaja de las matrices de adyacencia es el espacio que requieren para almacenar la información. Una alternativa para optimizar el uso de la memoria es por medio de las listas de adyacencia.

FIGURA 7.7
Ejemplo de representación de gráficas. *a*) Gráfica dirigida con costos. *b*) Matriz de adyacencia etiquetada o con costos.



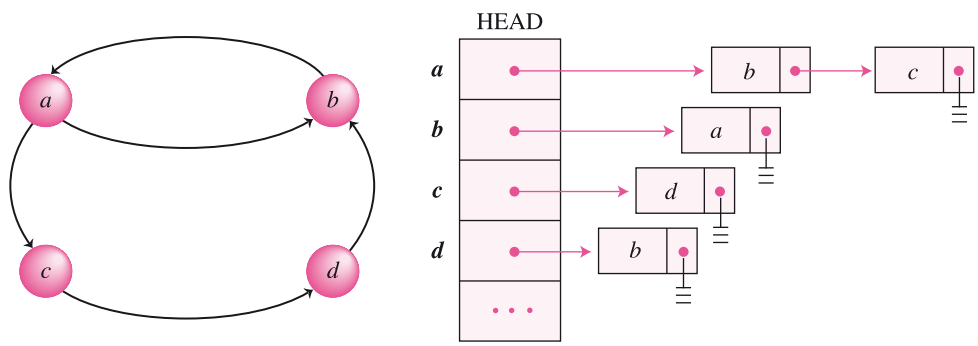
a)

	a	*b*	*c*	*d*	*e*	*f*
a	0	5	0	0	8	0
b	0	0	0	3	5	4
c	2	0	0	3	0	0
d	0	0	0	0	5	7
e	0	0	6	0	0	0
f	0	0	0	0	0	0

b)

FIGURA 7.8

Ejemplo de representación de gráficas. *a*) Digráfica. *b*) Lista de adyacencia de la digráfica.



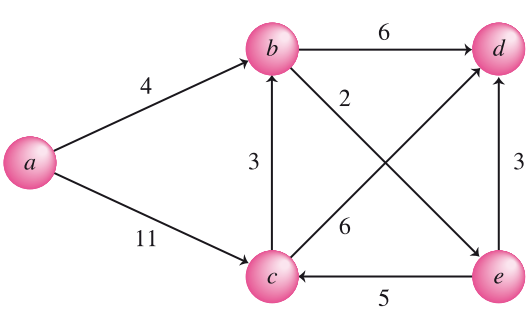
Lista de adyacencia

Una **lista de adyacencia** para un vértice a es una lista ordenada de todos los vértices adyacentes de a. Por lo tanto, una lista de adyacencia para representar una gráfica dirigida estará formada por tantas listas como vértices tenga G. Para guardar los vértices de G se puede utilizar otra lista o un arreglo. En este libro se usa un arreglo al que llamamos HEAD, donde HEAD[i] es un apuntador a la lista de vértices adyacentes al vértice i. La lista de adyacencia requiere un espacio de almacenamiento proporcional a la suma del número de vértices más el número de arcos.

Este tipo de representación se recomienda cuando el número de aristas es menor a n^2. El uso de la lista de adyacencia permite ahorrar espacio de almacenamiento. Sin embargo, usar una lista en lugar de una matriz tiene la desventaja de que el tiempo de búsqueda de las aristas puede ser mayor, ya que se pierde el acceso directo que permite la matriz. La operación de búsqueda será del orden de $O(n)$.

FIGURA 7.9

Ejemplo de aplicación del algoritmo de Dijkstra. *a*) Digráfica. *b*) Matriz de distancias de la digráfica.



	a	*b*	*c*	*d*	*e*
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

b)

En la figura 7.8 se observa que en el arreglo, en la posición correspondiente a cada uno de los cuatro vértices, se guardó un puntero a la lista de adyacencia de los respectivos vértices. Así, en la posición del nodo *a* hay un puntero a la lista formada por los vértices *b* y *c*, ambos adyacentes desde *a*.

7.4.2 Obtención de caminos dentro de una digráfica

Al buscar una estructura de datos que se ajuste a las características de un problema, se busca también que sobre dicha estructura se puedan realizar operaciones que faciliten el manejo de la información almacenada en ella. Para el caso de las gráficas dirigidas, generalmente resulta de interés encontrar los caminos, directos o indirectos, entre sus vértices. A su vez, al trabajar con digráficas etiquetadas se requiere encontrar el camino más corto entre dos vértices dados o entre todos sus vértices. Es decir, interesan aquellos caminos que nos permitan llegar desde un vértice origen a un vértice destino recorriendo la menor distancia o con el menor costo. Los algoritmos más usados para este fin son: **Dijkstra**, **Floyd** y **Warshall**. Los tres algoritmos utilizan una matriz de adyacencia etiquetada, donde

$M[i, j] = 0$ si $i = j$.
$M[i, j] = \infty$ si no existe un camino de i a j, donde $i \neq j$.
$M[i, j]$ = costo de ir del vértice i al vértice j, si existe a(i, j).

A partir de este punto, a la matriz de adyacencia etiquetada la llamaremos **matriz de distancias** o **matriz de costos**. En las siguientes secciones se presentarán los algoritmos mencionados.

7.4.3 Algoritmo de Dijkstra

El **algoritmo de Dijkstra** encuentra el camino más corto de un vértice elegido a cualquier otro vértice de la digráfica, donde la longitud de un camino es la suma de los pesos de las aristas que lo forman. Las aristas deben tener un peso no negativo.

Una posible aplicación de este algoritmo se presenta cuando se desea encontrar la ruta más corta entre dos ciudades; cada vértice representa una ciudad y el peso de las aristas indica la duración de los vuelos.

A continuación se describen los principales elementos que se consideran cuando se aplica el algoritmo.

- ***S*** es un arreglo formado por los vértices de los cuales ya conocemos la distancia mínima entre ellos y el origen. Este arreglo, inicialmente, sólo almacena al nodo origen.
- ***D*** es un arreglo formado por la distancia del vértice origen a cada uno de los otros. Es decir, $D[i]$ almacena la menor distancia, o costo, entre el origen y el vértice *i.* A este camino se le conoce como *especial.* Este arreglo se forma en cada paso del algoritmo. Al terminar el algoritmo, *D* contendrá la distancia mínima entre el origen y cada uno de los otros vértices de la gráfica.

- ***M*** es una matriz de distancias de $n \times n$ elementos, tal que $M[i, j]$ almacena la distancia o costo entre los vértices i y j, si entre ambos existe una arista. En caso contrario, $M[i, j]$ será un valor muy grande (∞).

El algoritmo de Dijkstra es el siguiente:

Algoritmo 7.1 Dijkstra

Dijkstra (*N*)

{Este algoritmo encuentra la distancia mínima entre un vértice origen y cada uno de los otros vértices de una gráfica dirigida. Se considera al vértice 1 como el vértice origen. *N* es el número de vértices de la gráfica dirigida. *S* y *D* son arreglos de *N* elementos y *M* es una matriz de $N \times N$ elementos, según lo descrito anteriormente}

1. Agregar el vértice 1 a *S*.
2. *Repetir* con *i* desde 2 hasta *N*.
  Elegir un vértice *v* en $(V - S)$ tal que $D[v]$ sea el mínimo valor
  Agregar *v* a *S*
  2.1 *Repetir* para cada vértice *w* en $(V - S)$
    Hacer $D[w] \leftarrow$ mínimo $(D[w], D[v] + M[v, w])$
  2.2 {Fin del ciclo del paso 2.1}
3. {Fin del ciclo del paso 2}

Dada una gráfica $G = (V, A)$, donde V está formado por n vértices, si se usa una matriz de distancias para representarla, cada ciclo toma un tiempo de $O(n)$ y son ejecutados $n - 1$ veces; por lo tanto, el algoritmo es del orden de $O(n^2)$. Si A es menor que n^2, entonces es más eficiente usar una lista de adyacencia para representar la digráfica. En este caso el tiempo de recorrido será del orden de $O(\log n)$, y el de los ciclos será del orden de $O(A \log n)$.

Ejemplo 7.1

A continuación se presenta un ejemplo de aplicación del algoritmo de Dijkstra para encontrar el camino más corto desde uno de los vértices a cualquiera de los otros vértices de una gráfica dirigida, formada por cinco vértices ($N = 5$).

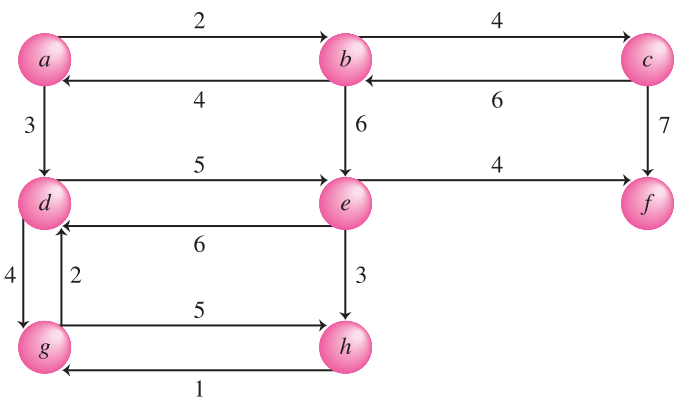
En la tabla 7.1 se presenta el seguimiento del algoritmo para la digráfica de la figura 7.9. La primera columna es para *S*, arreglo en el cual se almacena en cada paso del algoritmo el vértice seleccionado. Las columnas etiquetadas con $D[a]$, $D[b]$, ..., $D[e]$ se utilizan para mostrar el valor mínimo del camino encontrado entre el vértice origen y

TABLA 7.1
Aplicación del algoritmo de Dijkstra

S	*D[a]*	*D[b]*	*D[c]*	*D[d]*	*D[e]*	Comentario
$\{a\}$	0	**4**	11	∞	∞	Estado inicial
$\{a, b\}$	0	4	11	10	**6**	Se encontró: *a*, *b*, *d* y *a*, *b*, *e*
$\{a, b, e\}$	0	4	11	**9**	6	Se encontró: *a*, *b*, *e*, *d*
$\{a, b, e, d\}$	0	4	**11**	9	6	No hay cambios
$\{a, b, e, d, c\}$	0	4	11	9	6	Estado final: *a*, *b*, *e*, *d*, *c*

FIGURA 7.10
Ejemplo de aplicación del algoritmo de Dijkstra.

a) Digráfica. *b*) Matriz de distancias de la digráfica.



a)

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*
a	0	2	∞	3	∞	∞	∞	∞
b	4	0	4	∞	6	∞	∞	∞
c	∞	6	0	∞	∞	7	∞	∞
d	∞	∞	∞	0	5	∞	4	∞
e	∞	∞	∞	6	0	4	∞	3
f	∞	∞	∞	∞	∞	0	∞	∞
g	∞	∞	∞	2	∞	∞	0	5
h	∞	∞	∞	∞	∞	∞	1	0

b)

los vértices *a*, *b*, ..., *e,* respectivamente. Por último, se tiene una columna en la cual se comenta cada paso del algoritmo.

- Se eligió el vértice *a* como vértice origen.
- Una vez que el vértice elegido se agrega a *S*, su valor correspondiente en *D* no cambia.
- Los valores finales de *D* indican la distancia mínima entre el vértice origen y cada uno de los otros vértices. Por ejemplo, $D[d] = 9$ representa el costo de ir del vértice *a* al *d* pasando por *b* y *e*; $D[e] = 6$ es el costo de ir del vértice *a* al *e* pasando por *b*.
- La sombra se utiliza para indicar, en cada paso, cuál es el mínimo valor en *D,* lo que implica la elección del vértice correspondiente para ser incluido en *S*.

Ejemplo 7.2

En la figura 7.10 se presenta otro ejemplo de aplicación del algoritmo de Dijkstra para encontrar el camino más corto desde uno de los vértices a cualquiera de los otros vértices de una gráfica dirigida, con $N = 8$.

TABLA 7.2
Aplicación del algoritmo de Dijkstra

S	*D[a]*	*D[b]*	*D[c]*	*D[d]*	*D[e]*	*D[f]*	*D[g]*	*D[h]*	Comentario
{*a*}	0	**2**	∞	3	∞	∞	∞	∞	Estado inicial
{*a*, *b*}	0	2	6	**3**	8	∞	∞	∞	Se encontró: *a*, *b*, *c* y *a*, *b*, *e*
{*a*, *b*, *d*}	0	2	**6**	3	8	∞	7	∞	Se encontró: *a*, *b*, *d*, *g*
{*a*, *b*, *d*, *c*}	0	2	6	3	8	13	**7**	∞	Se encontró: *a*, *b*, *c*, *f*
{*a*, *b*, *d*, *c*, *g*}	0	2	6	3	**8**	13	7	12	Se encontró: *a*, *d*, *g*, *h*
{*a*, *b*, *d*, *c*, *g*, *e*}	0	2	6	3	8	12	7	**11**	Se encontró: *a*, *b*, *e*, *f* y *a*, *b*, *e*, *h*
{*a*, *b*, *d*, *c*, *g*, *e*, *h*}	0	2	6	3	8	**12**	7	11	No hay cambio
{*a*, *b*, *d*, *c*, *g*, *e*, *h*, *f*}	0	2	6	3	8	12	7	11	Estado final

En la tabla 7.2 se presenta el seguimiento del algoritmo para la digráfica de la figura 7.10. La primera columna es para *S*, arreglo en el cual se almacena en cada paso del algoritmo el vértice seleccionado. Las columnas etiquetadas con *D*[*a*], *D*[*b*], ... ,*D*[*h*] se utilizan para mostrar el valor mínimo del camino encontrado entre el vértice origen y cada uno de los otros vértices. Por último, se tiene una columna en la cual se comenta cada paso del algoritmo.

- Se eligió el vértice *a* como origen.
- Una vez que el vértice elegido se agrega a *S*, su valor correspondiente en *D* no cambia.
- Los valores finales de *D* indican la distancia mínima entre el vértice origen y cada uno de los otros vértices. Por ejemplo, *D*[*f*] = 12 representa el costo de ir del vértice *a* al vértice *f*, pasando por los vértices *b* y *e*.
- La sombra se utiliza para indicar, en cada paso, cuál es el mínimo valor en *D*, lo que implica la elección del vértice correspondiente para ser incluido en *S*.

7.4.4 Algoritmo de Floyd

El **algoritmo de Floyd** encuentra el camino más corto entre todos los vértices de la digráfica. Sea la gráfica dirigida $G = (V, A)$ donde cada arco $u \rightarrow v$ tiene asociado un peso. El algoritmo de Floyd permitirá encontrar el camino más corto entre cada par ordenado *u* y *v*.

La matriz de distancias sirve como punto de partida para este algoritmo. Se realizan *k* iteraciones sobre la matriz buscando el camino más corto; por lo tanto, en la *k*-ésima iteración, *M*[*i*, *j*] tendrá el camino de menor costo para llegar de *i* a *j*, pasando por un número de vértices menor a *k*, el cual se calculará según la siguiente expresión:

$$M_k[i,j]\text{mín} = \begin{cases} M_{k-1}[i,j] \\ M_{k-1}[i,k] + M_{k-1}[k,j] \end{cases}$$

Se elegirá el camino más corto entre el valor obtenido en la iteración $(k-1)$ y el que resulta de pasar por el vértice k. En el algoritmo se usa la matriz de costos, M, donde $M[i, j]$ será igual al costo de ir de i a j, a un valor muy grande (∞) si no existe camino de i a j, o a cero si $i = j$.

Algoritmo 7.2 Floyd

Floyd (*N*)

{Este algoritmo encuentra la distancia mínima entre todos los vértices de la gráfica dirigida. N es el número de vértices de la gráfica dirigida. M es una matriz de $N \times N$ elementos, y se inicia con los costos de la digráfica. k, i, j son variables enteras}

1. *Repetir* con K desde 1 hasta N
  1.1 *Repetir* con I desde 1 hasta N
    1.1.1 *Repetir* con J desde 1 hasta N
      1.1.1.1 *Si* $(M_{IK} + M_{KJ} < M_{IJ})$ *entonces*
        Hacer $M_{IJ} \leftarrow M_{IK} + M_{KJ}$
      1.1.1.2 {Fin del condicional del paso 1.1.1.1}
    1.1.2 {Fin del ciclo del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Para todo vértice de la digráfica se prueba si el camino más corto, para ir desde dicho vértice a los otros, es a través de un vértice intermedio k. En caso afirmativo, el costo que tiene asociado se reemplaza por la suma de los costos de ir del vértice origen al intermedio y del intermedio al destino. En otro caso, el valor de $M[i, j]$ no se modifica. Una vez probados todos los vértices de la digráfica como nodos intermedios, la matriz resultante almacena la menor distancia entre cada par de nodos. La complejidad del algoritmo es del orden de $O(N^3)$, ya que se utilizan tres ciclos anidados de orden N.

Ejemplo 7.3

A continuación se presenta un ejemplo de aplicación del algoritmo de Floyd para encontrar la mínima distancia entre todos los vértices de una gráfica dirigida. La matriz de distancias es la correspondiente a la digráfica del ejemplo 7.1. En la figura 7.11 se presentan los diferentes estados de la matriz de distancias de la digráfica dada, obtenidos a partir de la aplicación del algoritmo. Así, la matriz mostrada en el inciso *a)* es la matriz de distancias de la digráfica dada —estado inicial—. La matriz de costos mostrada en el inciso *b*) es la obtenida usando el vértice *b* como vértice intermedio. En ese paso, $K = 2$, se encontraron los siguientes caminos: *a*, *b*, *d* (con distancia igual a 10); *a*, *b*, *e* (con distancia igual a 6) y *c*, *b*, *e* (con distancia igual a 5). La figura de *c*) corresponde a la

matriz de distancias, resultado de usar el vértice *c* como vértice intermedio. En ese paso, $K = 3$, se encontró el siguiente camino: *e*, *c*, *b* (con distancia igual a 8). Finalmente, la figura presentada en el inciso *d)* muestra la matriz obtenida de distancias mediante el vértice *e* como vértice intermedio. Para $K = 5$ se encontraron los siguientes caminos: *a*, *e*, *d* (con distancia igual a 9), *b*, *e*, *c* (con distancia igual a 7) y *b*, *e*, *d* (con distancia igual a 5). El estado final de esta matriz almacena las distancias mínimas entre cada uno de los vértices de la digráfica dada.

Si además de obtener la menor distancia entre todos los vértices de la gráfica dirigida se requiere conocer la trayectoria encontrada para cada vértice, se deberá ir guardando dicha trayectoria —los vértices intermedios—. Esta variante del algoritmo de Floyd utiliza un arreglo auxiliar (*T*) de $N \times N$ elementos, donde $T[i, j]$ será igual a *k* si *k* es un nodo intermedio entre *i* y *j*.

FIGURA 7.11
Ejemplo de aplicación del algoritmo de Floyd.

a)

	a	b	c	d	e
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

b)

	a	b	c	d	e
a	0	4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

c)

	a	b	c	d	e
a	0	4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	8	5	3	0

d)

	a	b	c	d	e
a	0	4	11	9	6
b	∞	0	7	5	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	8	5	3	0

Algoritmo 7.3 Floyd_guarda_vértices

Floyd_guarda_vértices (*N*)

{Este algoritmo encuentra la distancia mínima entre todos los vértices de la gráfica dirigida. Además, almacena el conjunto de nodos intermedios que forman las trayectorias encontradas. N es el número de vértices de la gráfica dirigida. M es una matriz de $N \times N$ elementos, y se inicia con los costos de la digráfica. T es una matriz de $N \times N$ elementos, y almacenará en cada elemento i, j el vértice intermedio usado para ir de i a j. K, I, J son variables enteras}

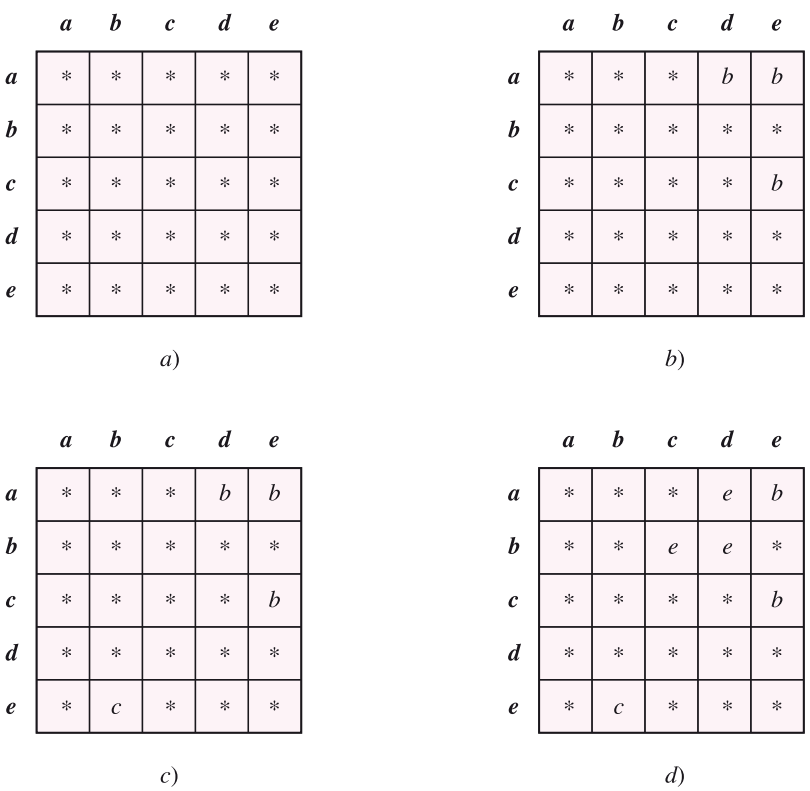
1. *Repetir* con K desde 1 hasta N
  1.1 *Repetir* con I desde 1 hasta N
    1.1.1 *Repetir* con J desde 1 hasta N
      1.1.1.1 *Si* $(M_{IK} + M_{KJ} < M_{ij})$ *entonces*
        Hacer $M_{IJ} \leftarrow M_{IK} + M_{KJ}$ y $T_{IJ} \leftarrow K$
      1.1.1.2 {Fin del condicional del paso 1.1.1.1}
    1.1.2 {Fin del ciclo del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Ejemplo 7.4

A continuación se presenta un ejemplo de aplicación del algoritmo de Floyd_guarda_vértices. Como resultado se obtiene una matriz con las distancias mínimas entre todos los vértices de una gráfica dirigida y los vértices intermedios utilizados para alcanzar esas distancias. La matriz de distancias es la correspondiente a la digráfica del ejemplo 7.1. Con respecto a la obtención de la matriz de distancias mínimas, el algoritmo genera una matriz igual a la presentada en la figura 7.12, donde se observa cómo se van asignando valores a la matriz (T) a medida que se van encontrando vértices intermedios que reducen la distancia entre dos vértices.

En el inciso *a*) se muestra el estado inicial de T. En este caso se asignó un * a cada componente del arreglo. En la figura del inciso *b*) se presenta el arreglo una vez registrado b (para $K = 2$) como vértice intermedio entre a y d, a y e, c y e. En *c*) se puede observar T luego de encontrar al nodo c (para $K = 3$) como intermedio entre e y b. Por último, se llega al estado final de T, inciso *d*), luego de encontrar a e (para $K = 5$) como vértice intermedio entre los vértices a y d, b y c, b y d.

FIGURA 7.12
Ejemplo de aplicación del algoritmo Floyd_guarda_vértices.



7.4.5 Algoritmo de Warshall

El **algoritmo de Warshall** encuentra, si es posible, un camino entre cada uno de los vértices de la gráfica dirigida. Es decir, la solución encontrada por el algoritmo no presenta las distancias entre los vértices, sólo muestra si hay o no camino entre ellos.

El algoritmo de Warshall se basa en un concepto llamado **cerradura transitiva** de la matriz de adyacencia. Sea la gráfica dirigida $G(V, A)$ y su matriz de adyacencia M, donde $M[i, j] = 1$ si hay un arco de i a j, y 0 si no lo hay. La cerradura transitiva de M es la matriz C tal que $C[i, j] = 1$ si hay un camino de longitud mayor o igual que 1 de i a j, o 0 en otro caso. Para generar la matriz C se establece que existe un camino del vértice i al j que no pasa por un número de vértices mayor que k si:

a) Ya existe un camino de i a j que no pasa por un número de vértices mayor que $k - 1$.
b) Hay un camino de i a k que no pasa por un número de vértices mayor que $k - 1$, y hay un camino de k a j que no pasa por un número de vértices mayor que $k - 1$.

Algoritmo 7.4 Warshall

Warshall (*N*)

{Este algoritmo encuentra, si es posible, un camino de longitud mayor o igual a uno entre cada uno de los vértices de la gráfica dirigida. *N* es el número de vértices de la digráfica. *C* es una matriz de $N \times N$ elementos. Inicialmente es igual a *M*. Al terminar el algoritmo contendrá la cerradura transitiva de *M*, *k*, *I*, *J* son variables enteras}

1. *Repetir* con *K* desde 1 hasta *N*
  1.1 *Repetir* con *I* desde 1 hasta *N*
    1.1.1 *Repetir* con *J* desde 1 hasta *N*
      1.1.1.1 *Si* (*A*[*I*, *J*] = 0) *entonces*
        A[*I*, *J*] ← *A*[*I*, *K*] y *A*[*K*, *J*]
      1.1.1.2 {Fin del condicional del paso 1.1.1.1}
    1.1.2 {Fin del ciclo del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Ejemplo 7.5

Se presenta un ejemplo de aplicación del algoritmo de Warshall para determinar si existe o no un camino entre todos los vértices de una gráfica dirigida. Se toma la digráfica del ejemplo 7.1; la matriz *C*, que contendrá la cerradura transitiva al finalizar el algoritmo, se inicia con los valores de la matriz de adyacencia correspondiente.

En la figura 7.13 se presentan los diferentes estados de *C*, obtenidos a partir de la aplicación del algoritmo. En la figura del inciso *a*) se muestra el estado inicial de la matriz. Por su parte, en *b*) se presenta la correspondiente a la matriz *C*, resultado de usar el vértice *b* como vértice intermedio. Se encontraron los siguientes caminos: *a*, *b*, *d*, *a*, *b*, *e* y *c*, *b*, *e*. La figura del inciso *c)* corresponde a la matriz *C* obtenida por medio del vértice *c* como vértice intermedio. Se formaron los caminos *e*, *c*, *b* y *e*, *c*, *e*. Finalmente, en *d*) se presenta a la matriz *C* luego de usar el vértice *e* como vértice intermedio. Se obtuvieron los caminos *a*, *e*, *a*, *b*, *e*, *b*, *b*, *e*, *c* y *c*, *e*, *c*.

7.5 GRÁFICAS NO DIRIGIDAS

En esta sección se presentará el concepto de **gráficas no dirigidas** o simplemente **gráficas**, cuya característica principal es que sus aristas son pares no ordenados de vértices. Es decir, si existe un camino del vértice *i* al *j*, será exactamente el mismo camino del vértice *j* al *i*.

Estas gráficas se utilizan para modelar relaciones simétricas entre diferentes objetos, que se representan por medio de los vértices, mientras que las aristas se usan para indicar las relaciones entre ellos. Por ejemplo, el costo de un boleto de avión para ir de la ciudad de México a Guadalajara será el mismo en cualquiera de las direcciones que se realice el viaje.

	a	b	c	d	e
a	0	1	1	0	0
b	0	0	0	1	1
c	0	1	0	1	0
d	0	0	0	0	0
e	0	0	1	1	0

a)

	a	b	c	d	e
a	0	1	1	**1**	**1**
b	0	0	0	1	1
c	0	1	0	1	**1**
d	0	0	0	0	0
e	0	0	1	1	0

b)

	a	b	c	d	e
a	0	1	1	**1**	**1**
b	0	0	0	1	1
c	0	1	0	1	**1**
d	0	0	0	0	0
e	0	**1**	1	1	**1**

c)

	a	b	c	d	e
a	**1**	1	1	**1**	**1**
b	0	**1**	**1**	1	1
c	0	1	**1**	1	**1**
d	0	0	0	0	0
e	0	**1**	1	1	**1**

d)

FIGURA 7.13
Ejemplo de aplicación del algoritmo de Warshall.

Una gráfica no dirigida $G = (V, A)$ consta de un conjunto finito de vértices V y de otro finito de aristas A. Se diferencia de una gráfica dirigida en que cada arista en A es un par no ordenado de vértices. Si (u, v) es una arista no dirigida, entonces $(u, v) = (v, u)$.

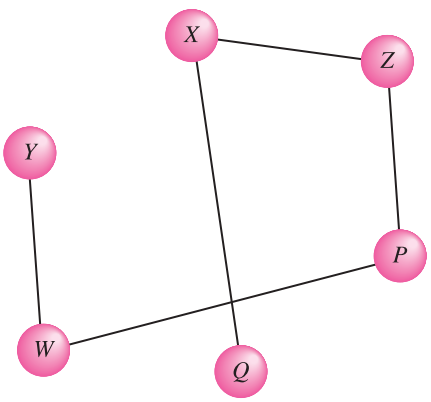
7.5.1 Representación de gráficas no dirigidas

Las gráficas no dirigidas son estructuras de datos abstractas; por lo tanto, se deben apoyar en otras estructuras para su representación en memoria. Las dos representaciones más utilizadas son **matriz de adyacencia** y **lista de adyacencia**, ambas explicadas en la sección correspondiente a las gráficas dirigidas.

Considerando la simetría de las relaciones entre los elementos de la gráfica, se requiere cambiar cada arista no dirigida entre u y v por dos aristas dirigidas, una de u a v y otra de v a u; por lo tanto, la matriz de adyacencia resultará una matriz simétrica, y en la lista de adyacencia el vértice u estará en la lista de adyacencia del vértice v, y viceversa.

En la figura 7.14*a* se puede observar un ejemplo de una gráfica y en *b*) su representación por medio de una matriz de adyacencia. En la matriz se ha sombreado la diagonal

FIGURA 7.14
Gráfica y su representación por medio de una matriz de adyacencia.

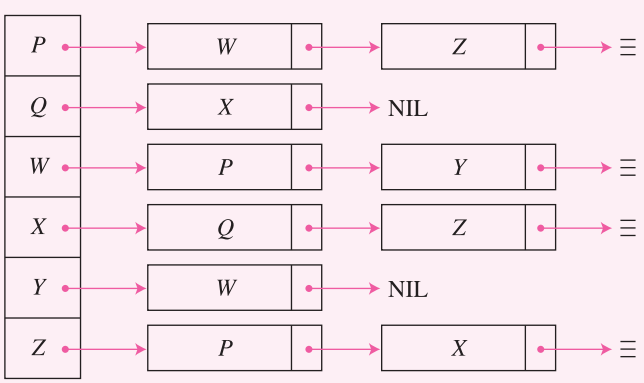


a)

	P	Q	W	X	Y	Z
P	0	0	1	0	0	1
Q	0	0	0	1	0	0
W	1	0	0	0	1	0
X	0	1	0	0	0	1
Y	0	0	1	0	0	0
Z	1	0	0	1	0	0

b)

FIGURA 7.15
Representación de una gráfica por medio de una lista de adyacencia.



principal para ilustrar más claramente la simetría. Es decir, la matriz triangular inferior es igual a la matriz triangular superior.

En la figura 7.15 se muestra la representación de la gráfica de la figura 7.14*a* por medio de una lista de adyacencia.

7.5.2 Construcción del árbol abarcador de costo mínimo

Sea $G = (V, A)$ una gráfica conexa, es decir, una gráfica en la cual existe un camino simple entre cualesquiera dos de sus vértices. Además, cada arista (u, v) tiene asociado un peso o costo, $c(u, v)$.

Considerando lo anterior, un **árbol abarcador** de una gráfica G se define como un árbol libre que conecta todos los vértices de V. El costo del árbol abarcador resulta de la suma de las aristas incluidas en él. Por lo tanto, un **árbol abarcador de costo mínimo** se forma a partir de las aristas de menor costo.

Una aplicación típica de árboles abarcadores de costo mínimo es el diseño de redes de comunicación. Por ejemplo, los vértices de la gráfica pueden representar ciudades, y las aristas posibles canales de comunicación entre ellas. El costo asociado a cada arista representa el costo de comunicar una ciudad con otra (en tiempo, dinero, medios, etc.). Por lo tanto, el árbol abarcador representará la red de comunicación que conecta a todas las ciudades a un costo mínimo.

Los árboles abarcadores de costo mínimo gozan de una propiedad que sirve como base para todos los algoritmos utilizados para su construcción. Esta propiedad establece que si $G = (V, A)$ es una gráfica conexa, U es un subconjunto propio del conjunto de vértices V, y (u, v) es una arista de costo mínimo tal que $u \in U$ y $v \in V - U$, entonces existe un árbol abarcador de costo mínimo que incluye a (u, v) entre sus aristas.

En las siguientes secciones se presentarán los algoritmos de Prim y de Kruskal utilizados para obtener el árbol abarcador de una gráfica.

7.5.3 Algoritmo de Prim

El **algoritmo de Prim** permite encontrar el árbol abarcador de costo mínimo de una gráfica. Para ello utiliza dos conjuntos: V, conjunto de todos los vértices, y U, conjunto auxiliar iniciado con el primer vértice. En cada iteración del algoritmo se busca la arista (u, v) que conecte U con la subgráfica V, U. Luego se agrega el nodo v, perteneciente a V, U, a U. Este proceso se repite hasta que $U = V$.

Considerando el recorrido que se debe hacer en la gráfica, el tiempo de ejecución será del orden de $O(n^2)$ si se usa una matriz de adyacencia para representarla. En cambio, si la gráfica fue representada por medio de una lista de adyacencia, la complejidad del algoritmo será del orden de $O(A * \log n)$, donde A es el número de aristas.

Antes de presentar el algoritmo, resulta conveniente explicar los elementos que se usarán en él.

- ***V*** es el conjunto de vértices de G: $V = \{1, 2, ..., n\}$. Se usan los números enteros del 1 en adelante para identificar los vértices. Sin embargo, en cada aplicación se podrá usar la manera que se considere más adecuada.
- ***U*** es un subconjunto propio del conjunto V, siendo su valor inicial el del primer vértice. Para nuestra implementación: $U = \{1\}$.
- ***L*** es una lista de aristas que se va formando con las aristas de menor costo que se van seleccionando. Inicialmente L está vacía: $L = \varnothing$.

Algoritmo 7.5 Prim

Prim (*N*)

{Este algoritmo encuentra el árbol abarcador de costo mínimo de una gráfica G de N vértices. U, V y L son estructuras de datos —arreglos o listas— que permiten guardar los nombres de los vértices y las aristas seleccionadas}

1. Mientras ($V \neq U$) Repetir
   Elegir una arista $(u, v) \in A(G)$ tal que su costo sea mínimo, siendo $u \in U$ y $v \in (V - U)$
   Agregar la arista (u, v) a L
   Agregar el nodo v a U
2. {Fin del ciclo del paso 1}

Ejemplo 7.6

A continuación se presenta un ejemplo de uso del algoritmo de Prim para encontrar el árbol abarcador de costo mínimo de una gráfica. El algoritmo se aplica a la gráfica de la figura 7.16, obteniendo como resultado el árbol que se muestra en la figura 7.17.

El seguimiento del algoritmo se presenta en la tabla 7.3.

En la primera iteración se elige la arista (1,2) por ser la de menor costo, en este caso igual a **1**. El vértice **2** se agrega al conjunto U. En la siguiente iteración se elige la arista (2,3), con un costo de 3 y el vértice **3** pasa a formar parte de U. Así se sigue hasta alcanzar la condición final de ($V = U$), y consecuentemente termina el proceso.

FIGURA 7.16
Ejemplo de aplicación del algoritmo de Prim.

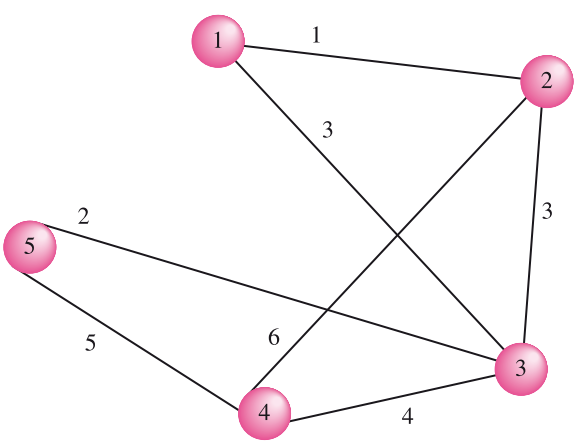


TABLA 7.3
Aplicación del algoritmo de Prim

Iteración	*U*	(*u*, *v*)	Costo	*v*	*L*	*U actualizado*
1	{1}	(1, 2)	1	2	{(1, 2)}	{1, 2}
2	{1, 2}	(2, 3)	3	3	{(1, 2), (2, 3)}	{1, 2, 3}
3	{1, 2, 3}	(3, 5)	2	5	{(1, 2), (2, 3), (3, 5)}	{1, 2, 3, 5}
4	{1, 2, 3, 5}	(3, 4)	4	4	{(1, 2), (2, 3), (3, 5), (3, 4)}	{1, 2, 3, 5, 4}

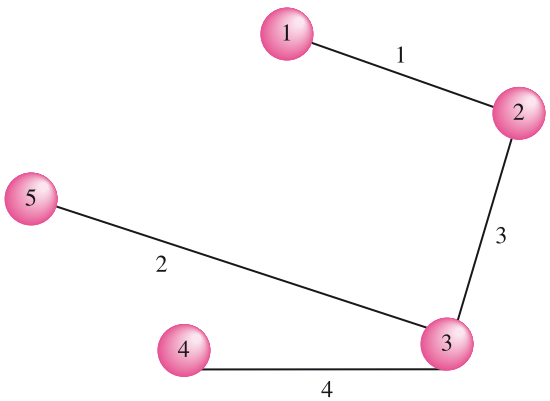


FIGURA 7.17
Árbol abarcador de costo mínimo, obtenido aplicando el algoritmo de Prim.

7.5.4 Algoritmo de Kruskal

El **algoritmo de Kruskal**, al igual que el de Prim, permite encontrar el árbol abarcador de costo mínimo de una gráfica. La construcción del árbol abarcador de costo mínimo se lleva a cabo seleccionando la arista de menor costo y agregándola al árbol abarcador. Para ello se utiliza un proceso relativamente sencillo.

Primero, se debe generar una serie de particiones a partir del conjunto de vértices *V*. Inicialmente, las particiones tienen tamaño uno. Es decir:

$$P_0 = \{\{V_1\}, \{V_2\}, \{V_3\}, ..., \{V_n\}\}$$

donde P_0 indica la partición inicial, y cada $\{V_i\}$ es una partición formada por el vértice *i*.

A partir de este paso se busca la arista de menor costo, y si ésta une dos vértices que pertenecen a particiones diferentes, dichas particiones se reemplazan por su unión. Para el caso contrario, la arista no forma parte del árbol abarcador de costo mínimo, ya que produciría un ciclo. Se continúa eligiendo la arista (u, v) de menor costo y uniendo las particiones a las cuales pertenecen u y v, respectivamente, hasta que se tenga una sola partición formada por todos los vértices de la gráfica. Es decir:

$$P_k = \{V_1, V_2, V_3, ..., V_n\}$$

donde P_k es la partición final, luego de k iteraciones, la cual está formada por los N vértices de G.

Este algoritmo requiere, en el peor de los casos, un tiempo de $O(A * \log A)$ donde A es el número de aristas de la gráfica. Si A es menor que n^2, entonces el algoritmo de Kruskal es más eficiente que el de Prim. Si A tiene un valor cercano a n^2, entonces es más conveniente usar el de Prim.

El algoritmo utiliza algunos elementos auxiliares. A continuación se describen los mismos:

- ***L*** es un conjunto formado por las aristas y sus respectivos costos.
- ***P*** representa las particiones generadas a partir de *V*. Inicialmente $P = \{\{1\}, \{2\}, ..., \{n\}\}$

Para estos dos elementos —L y P— se pueden usar arreglos o listas para su representación en memoria.

Algoritmo 7.6 Kruskal

Kruskal (N)

{Este algoritmo encuentra el árbol abarcador de costo mínimo de una gráfica G de N vértices. L y P son estructuras de datos —arreglos o listas— que permiten guardar las aristas y las particiones, respectivamente}

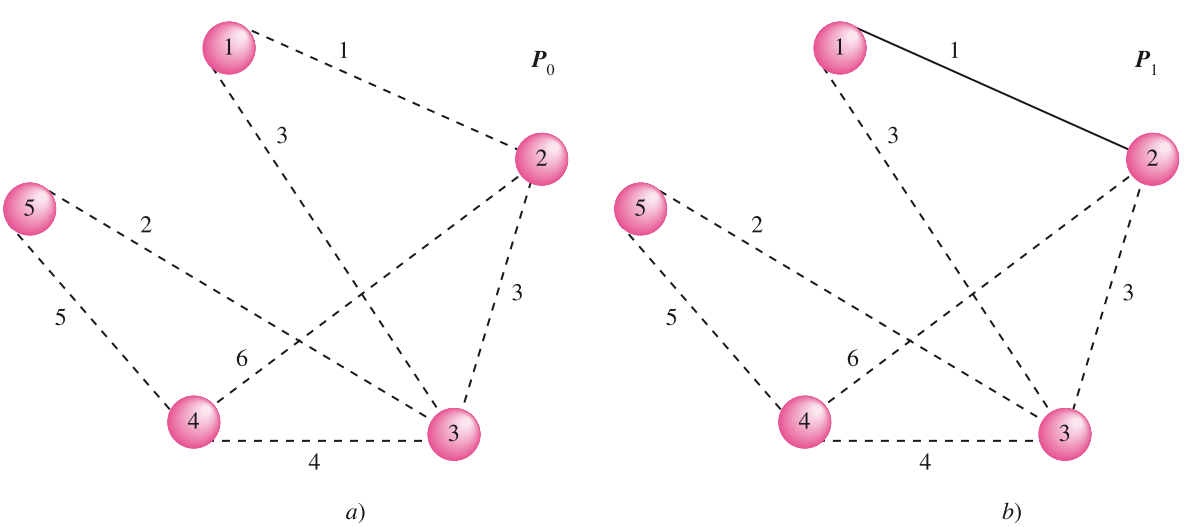
1. Mientras haya vértices en P que pertenezcan a particiones distintas *Repetir*
  De L seleccionar la arista (u, v) que tenga el menor costo
  1.1 *Si* (u y v se encuentran en particiones diferentes) *entonces*
    Unir las particiones a las cuales pertenecen u y v
  1.2 {Fin del condicional del paso 1.1}
2. {Fin del ciclo del paso 1}

Ejemplo 7.7

Se presenta un ejemplo de uso del algoritmo de Kruskal para encontrar el árbol abarcador de costo mínimo de una gráfica, el cual se aplica a la gráfica de la figura 7.18, obteniendo como resultado el árbol que se muestra en la figura 7.19.

FIGURA 7.18

Árbol abarcador de costo mínimo obtenido aplicando el algoritmo de Kruskal. *a)* Gráfica inicial: en ella aparecen todas las aristas con su peso asociado. Partición inicial (P_0). *b)* Luego de seleccionar la arista de menor peso (1), que une los vértices 1 y 2.



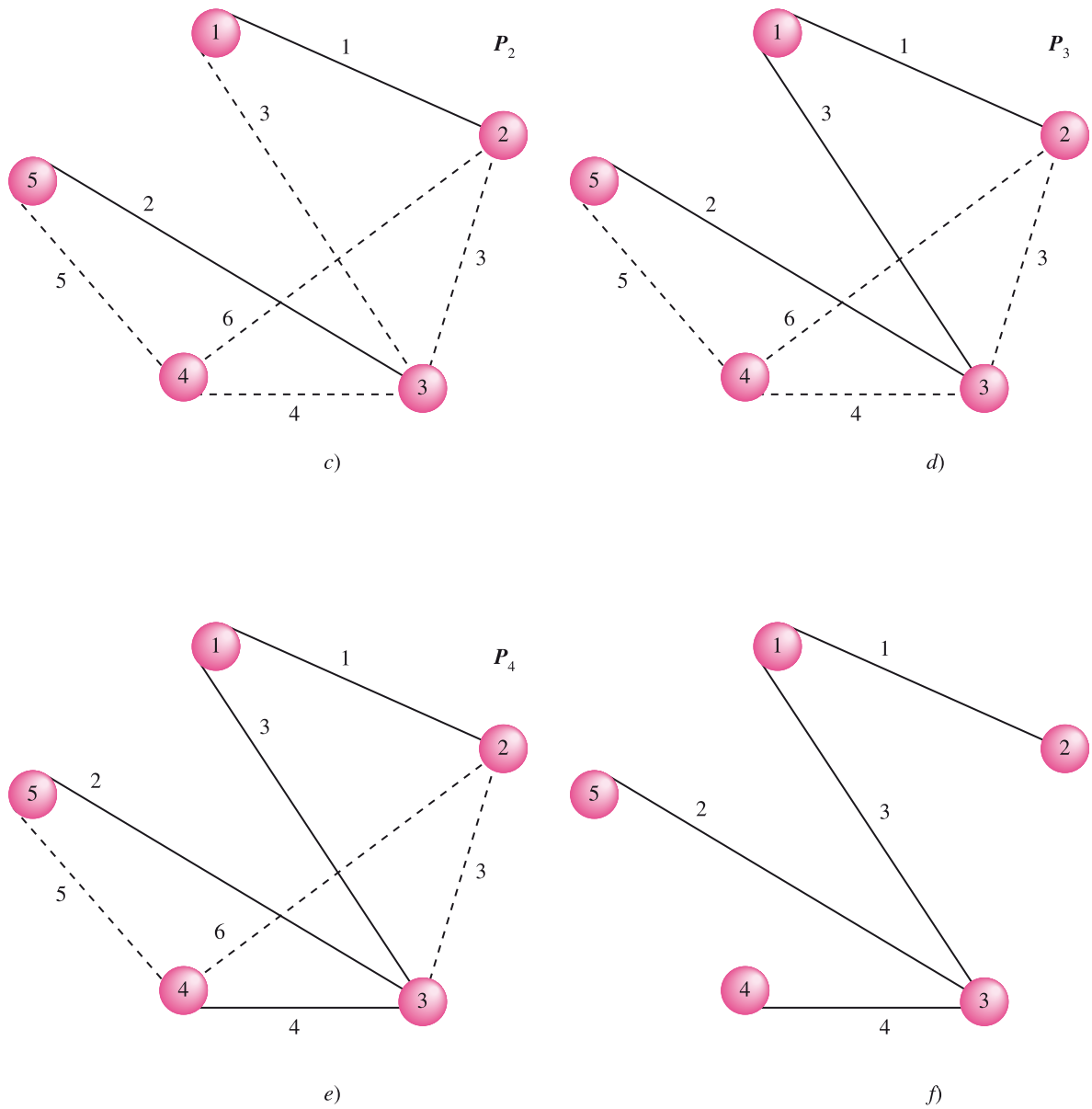
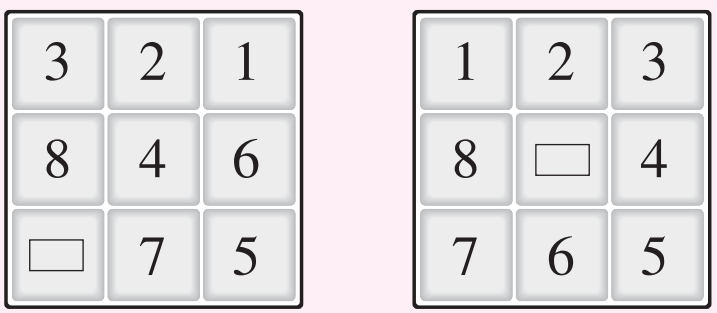


FIGURA 7.18

(*continuación*)

c) Luego de seleccionar la siguiente arista de menor peso (2), que une los vértices 3 y 5.
d) Luego de seleccionar la siguiente arista de menor peso (3), que une los vértices 1 y 3.
e) Luego de seleccionar la siguiente arista de menor peso (4), que une los vértices 3 y 4.
f) Árbol abarcador al que se llega luego de aplicar el algoritmo de Kruskal.

FIGURA 7.19



Estado inicial Estado final

La lista L formada por las aristas y sus costos $(u, v)c$, donde u y v son vértices y c representa el costo asociado a dicha arista, es la siguiente:

$$L = \{(1, 2)\mathbf{1}, (3, 5)\mathbf{2}, (1, 3)\mathbf{3}, (2, 3)\mathbf{3}, (3, 4)\mathbf{4}, (4, 5)\mathbf{5}, (2, 6)\mathbf{6}\}$$

En la tabla 7.4 se presenta el seguimiento del algoritmo.

En el paso 1 se elige la arista (1, 2) porque es la que tiene asociado el menor costo, y, en consecuencia, se unen las particiones correspondientes a los vértices 1 y 2. La partición resultante se sombrea. Se sigue de la misma manera en los pasos 2 y 3. En el paso 4 se elige la arista (3, 4) con un costo de 4, y no la arista (2, 3) con un costo de 3, ya que los vértices de esta última no cumplen con la condición de pertenecer a particiones distintas.

7.6 RESOLUCIÓN DE PROBLEMAS

Los problemas básicamente se clasifican en dos grandes subgrupos: los que cuentan con una solución **determinística** para su solución, expresable por medio de un algoritmo, y los que requieren de una **búsqueda** para su solución. La inteligencia artificial se preocupa de este tipo de problemas, sin importar si son más o menos complejos que los anteriores —determinísticos—.

TABLA 7.4
Aplicación del algoritmo de Kruskal

Paso	Arista elegida	Costo	Particiones
0			P_0= {{1}, {2}, {3}, {4}, {5}} //Estado inicial
1	(1, 2)	1	P_1 = {{1, 2}, {3}, {4}, {5}}
2	(3, 5)	2	P_2 = {{1, 2}, {3, 5}, {4}}
3	(1, 3)	3	P_3 = {{1, 2, 3, 5}, {4}}
4	(3, 4)	4	P_4 = {1, 2, 3, 5, 4}

En la vida existe una cantidad de problemas que se pueden resolver aplicando métodos de búsqueda, si éstos fueron modelados mediante gráficas. Uno de los problemas más estudiado y utilizado en el campo de la solución de problemas es el del *puzzle-8*. Este es un juego que consiste en ordenar un conjunto de fichas en un tablero de $N \times N$ posiciones, usando sólo un lugar libre, de tal manera que aquéllas queden en una secuencia de 1 a $N^2 - 1$. Se utilizan frecuentemente los juegos en el área de resolución de problemas, porque proporcionan una rica fuente de ejemplos para comparar y probar distintos métodos de búsqueda. En la figura 7.20 se presenta un ejemplo de un puzzle-8.

En el puzzle existe un estado inicial y otro final definidos. Por otra parte, hay un conjunto de movimientos que permiten cambiar de una configuración a otra; estos movimientos se denominan operadores. En los diferentes estados parciales siempre existen operadores prohibidos, es decir, que no se pueden aplicar porque los movimientos representan estados ilegales.

En la figura 7.20 se observan tanto los estados inicial y final del problema a resolver. La celda vacía, en este caso, sólo se puede intercambiar con las celdas que contienen al 7 y al 8, respectivamente.

En el puzzle-8, que se representa como una matriz de 3 × 3, el número posible de combinaciones que se podrían generar es 9!, lo cual implica que el puzzle tenga 362 880 estados legales:

$$(3 \times 3)! = 9! = 362\,880$$

Es prudente destacar que para un estado final sólo existen 9!/2 estados iniciales posibles. La complejidad de este problema es similar a la que tiene un cartero que debe distribuir nueve cartas en nueve direcciones diferentes, y quiere encontrar la trayectoria óptima. Si el problema que tuviéramos que resolver fuera en cambio el puzzle-15, que se representa en una matriz de de 4 × 4, los estados legales del problema son 16! = 2.09*E*13. Por otra parte, cabe mencionar que existen puzzles-3, que se representan en una matriz de 2 × 2, cuyo número de estados legales es 4!; sin embargo, muchos de ellos no se pueden resolver.

Para que el lector observe la complejidad del problema, se presentan las tablas 7.5 y 7.6, donde se muestra el tiempo que se necesita para generar todos los estados legales del puzzle-8 y puzzle-15, respectivamente.

En la primera columna se presenta el número de nodos generados por segundo, mientras que en la segunda es el tiempo necesario para procesar todos los nodos. En la tabla 7.5 se observa que si se pudieran procesar 1 000 nodos por segundo, se requerirían 362.88 segundos para alcanzar la solución del problema. En cambio, si la capacidad de procesamiento fuera de 1 000 000 de nodos por segundo, sólo se necesitaría 0.36 segundos para llegar a la solución.

TABLA 7.5
Puzzle-8

Nodos por segundo	Tiempo de solución
1 000	362.88 segundos 6.04 minutos
1 000 000	0.36 segundos

TABLA 7.6
Puzzle de 4 × 4

Nodos por segundo	Tiempo de solución
1 000	2.09 E10 segundos 3.48 E08 minutos 5 805 555 horas 241 898 días 662.73 años
1 000 000	20 900 000 segundos 348 333 minutos 5805.55 horas 241.89 días

En la tabla 7.6 se presentan los resultados para el puzzle-15. Si la velocidad con que se generan los nodos fuera de 1 000 nodos por segundo, entonces se necesitarían **663 años** para generar todos los nodos; si, en cambio, la velocidad fuera de 1 000 000 de nodos por segundo, se necesitarían **243 días**. Observe que los problemas son fácilmente entendibles, pero la solución es muy compleja.

Los puzzles constituyen un excelente campo para aplicar y probar métodos de búsqueda. El puzzle más frecuentemente utilizado es el puzzle-8; es decir, aquel de dimensión 3 × 3, donde se deben acomodar los números 1 al 8. En la figura 7.20 se presenta otro ejemplo de puzzle-8. El problema queda definido en función de:

- Un estado inicial y un estado final.
- Un conjunto de movimientos (u operadores) permitidos para cambiar de una configuración a otra. Es decir, un operador está asociado al concepto de movimiento y es el que permite transformar un estado en otro.
- Un conjunto de operadores prohibidos.

Para el problema del puzzle los operadores válidos se muestran en la figura 7.21, mientras que los operadores prohibidos se presentan en la figura 7.22.

FIGURA 7.20
Ejemplo de un puzzle-8.

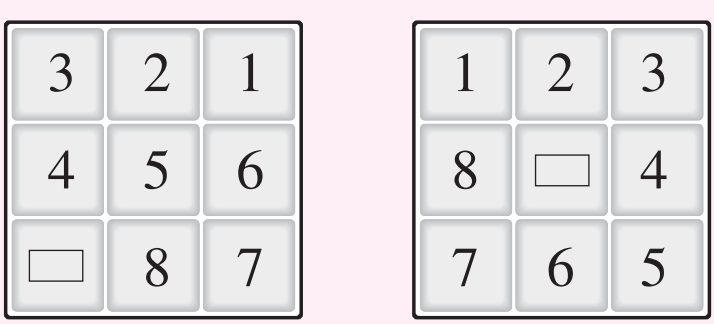


FIGURA 7.21
Operadores válidos para el puzzle.

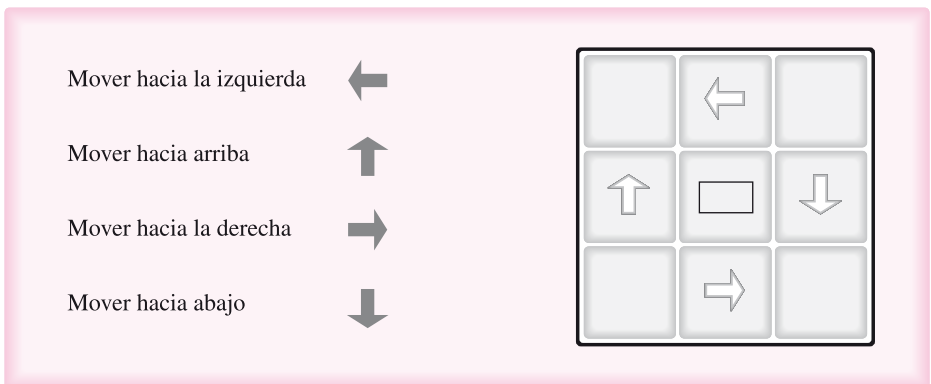
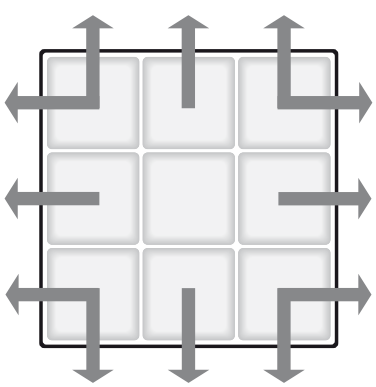


FIGURA 7.22
Operadores prohibidos para el puzzle.



En lenguaje de estados y operadores, una solución al problema consiste en obtener una secuencia apropiada de operadores que permitirán transformar en inicial el estado final.

La solución de un problema requiere de un orden de búsqueda para su solución, pero antes de comenzar la búsqueda es necesario modelar o representar el problema de alguna forma. Las alternativas son:

- Espacio/estado
- Reducción de problemas

En este capítulo sólo analizaremos la representación espacio/estado porque es la que se relaciona con las gráficas.

7.6.1 Espacio-estado

Un paso importante en la formulación de un problema **espacio/estado** consiste en la selección de una forma de representar los estados del problema. Las estructuras de datos más usadas para describir los estados son: arreglos unidimensionales, arreglos bidimen-

sionales, listas ligadas, árboles y gráficas. En el problema del puzzle-8, una matriz de 3 × 3 parece ser la estructura más natural para representar los estados del problema. Sin embargo, hay que ser cuidadosos en la elección de la estructura por dos razones:

1. La solución debe ser computable. Es decir, se debe poder desarrollar un método que se pueda *ejecutar* en una computadora en un tiempo razonable.
2. Debe permitir almacenar y manipular los operadores que transforman un estado en otro.

Considerando que la matriz es una estructura de datos estática, la lista resulta ser la estructura de datos más adecuada para representar al puzzle-8. Por el dinamismo de la misma, se pueden almacenar fácilmente los estados y los operadores del problema.

7.6.2 Métodos de búsqueda en espacio-estado

En el planteamiento de problemas espacio/estado, una solución se obtiene mediante la aplicación de operadores desde el estado inicial hasta alcanzar el estado final o meta. Es importante considerar los siguientes puntos:

- Un nodo inicial se asocia con la descripción de un estado inicial.
- Los sucesores de un nodo se calculan usando los operadores que son aplicables a ese estado.
- *X* es un operador que calcula todos los sucesores de un nodo. El proceso de aplicar *X* a un nodo se denomina **expander** un nodo.
- Se utilizan apuntadores para ligar el nodo padre con su hijo y, de esta manera, poder obtener la trayectoria cuando se encuentra el estado meta.

Los métodos de búsqueda se caracterizan por el orden en el cual se expanden los nodos; los dos básicos más ampliamente conocidos son:

- ***Breadth-first*** o **búsqueda a lo ancho**: Se expanden los nodos en el orden en que han sido generados.
- ***Depth-first*** o **búsqueda en profundidad**: Se expanden los nodos que han sido generados recientemente.

Es importante señalar que éstos son métodos de búsqueda elementales para encontrar las trayectorias, pero son exhaustivos porque expanden demasiados nodos y debemos recordar que siempre existen límites, tanto de tiempo como de espacio, para encontrar la solución del problema. Dentro del área de resolución de problemas de inteligencia artificial existen otros métodos más eficientes que incorporan conocimiento, estrategia y heurística, y permiten no sólo encontrar una trayectoria, sino también la óptima.

7.6.3 Método de búsqueda *breadth-first*

En el método de búsqueda ***breadth-first*** o **búsqueda a lo ancho** los nodos se expanden en el orden en el que han sido generados. Se avanza por niveles; es decir, primero se generan todos los nodos del primer nivel, luego los del segundo, posteriormente los del tercero, y así sucesivamente hasta encontrar el estado meta, siempre que sea posible en cuanto a tiempo y espacio ocupado en memoria. El método trabaja con dos listas auxiliares, una llamada **ABIERTO** y la otra **CERRADO**. La primera se utiliza para almacenar los nodos que están pendientes de ser expandidos; cada nuevo nodo generado se coloca siempre al final de ABIERTO. En CERRADO se colocan los nodos que ya han sido expandidos. A continuación se presenta el algoritmo correspondiente.

Algoritmo 7.7 *Breadth-first*

Breadth_first

{Este método permite encontrar la trayectoria que hace posible ir de un estado inicial a un estado final, usando los operadores permitidos. ABIERTO y CERRADO son dos listas ligadas en las cuales se almacenan los nodos pendientes de ser expandidos y los nodos ya expandidos, respectivamente}

```
1. Insertar el nodo inicial en la lista ABIERTO
2. Mientras (ABIERTO no esté VACÍA) y (no se haya alcanzado el estado final). Repetir:
        Quitar el primer nodo X de ABIERTO
   2.1 Si (el nodo X no se encuentra en CERRADO) entonces
                Poner el nodo X en la lista CERRADO
                Expandir el nodo X obteniendo todos sus sucesores
       2.1.1 Si (hay sucesores y no son el estado final) entonces
                  Almacenarlos al final de ABIERTO y proveer apuntadores para regresar a X
       2.1.2 {Fin del condicional del paso 2.1.1}
   2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. Si (se generó el estado final)
        entonces {Éxito}
            Desplegar trayectoria del estado inicial al estado final
         si no {Fracaso}
            Escribir "No se alcanzó el estado final"
5. {Fin del condicional del paso 4}
```

Ejemplo 7.8

A continuación se presenta un ejemplo de aplicación del método de búsqueda *breadth-first* para encontrar una trayectoria entre dos nodos. El problema consiste en encontrar una trayectoria de *S* a *K*, de la gráfica presentada en la figura 7.23, tomando el orden alfabético como base para el orden de aplicación de los operadores.

En la figura 7.24 se observa que para este problema, la solución alcanzada es:

S - A - C - E - J - K

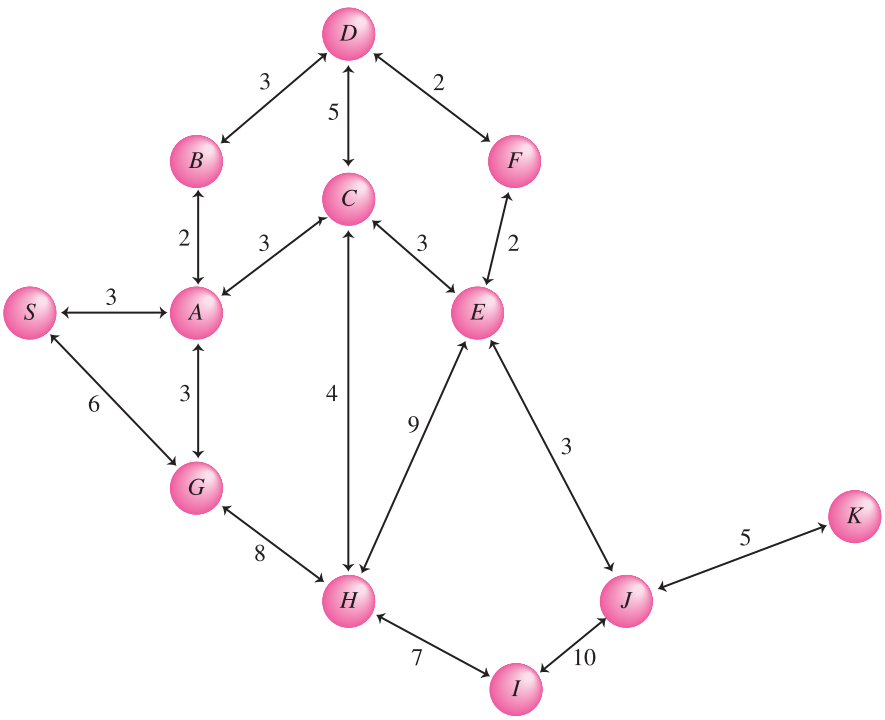


FIGURA 7.23
Búsqueda *breadth-first*.

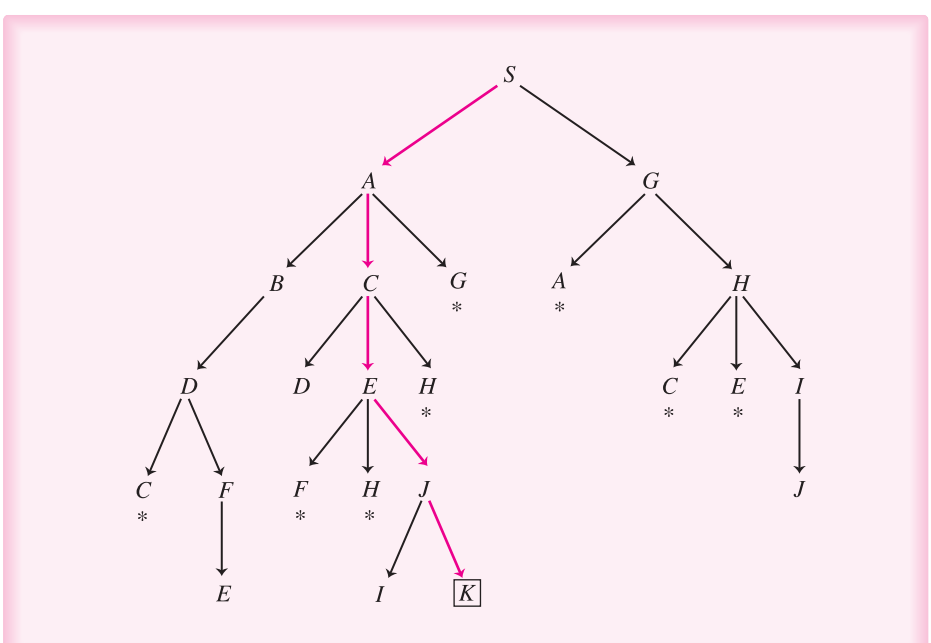
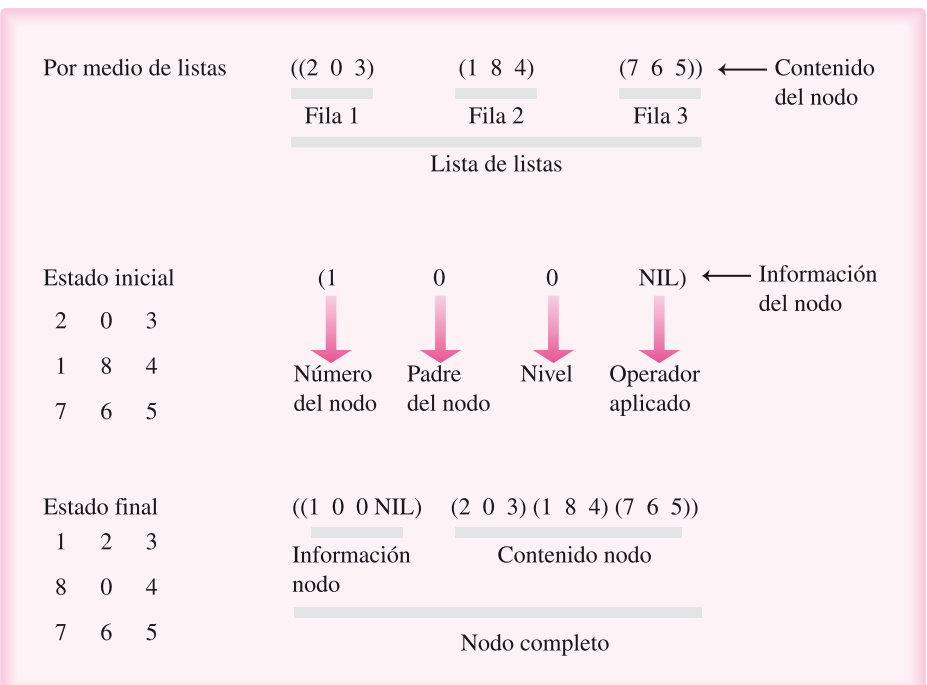


FIGURA 7.24
Trayectoria encontrada:
S - A - C - E - J - K.
* Ya fueron expandidos.

la cual se indica con una flecha en color. En el estado inicial, *S*, se obtienen todos sus sucesores, los cuales, según la gráfica de la figura 7.23, son *A* y *G*. Como ninguno de los dos es el estado meta, se continúa expandiendo cada uno de estos nodos. Así, para *A* se obtienen *B*, *C* y *G*, mientras que para *G* se generaron *A* y *H*. En ambos casos se obtuvieron nodos (*G* y *A*) con los que ya se contaba por expansiones anteriores; por lo tanto, se ignoran. En la gráfica, estos casos se señalan con *. En la figura se observa claramente que todas las ramas del árbol crecen, en profundidad, de igual manera.

Ejemplo 7.9

A continuación se presenta otro ejemplo de aplicación del método de búsqueda *breadth-first*, pero ahora para resolver el puzzle-8. En este caso se usan listas para representar los nodos.



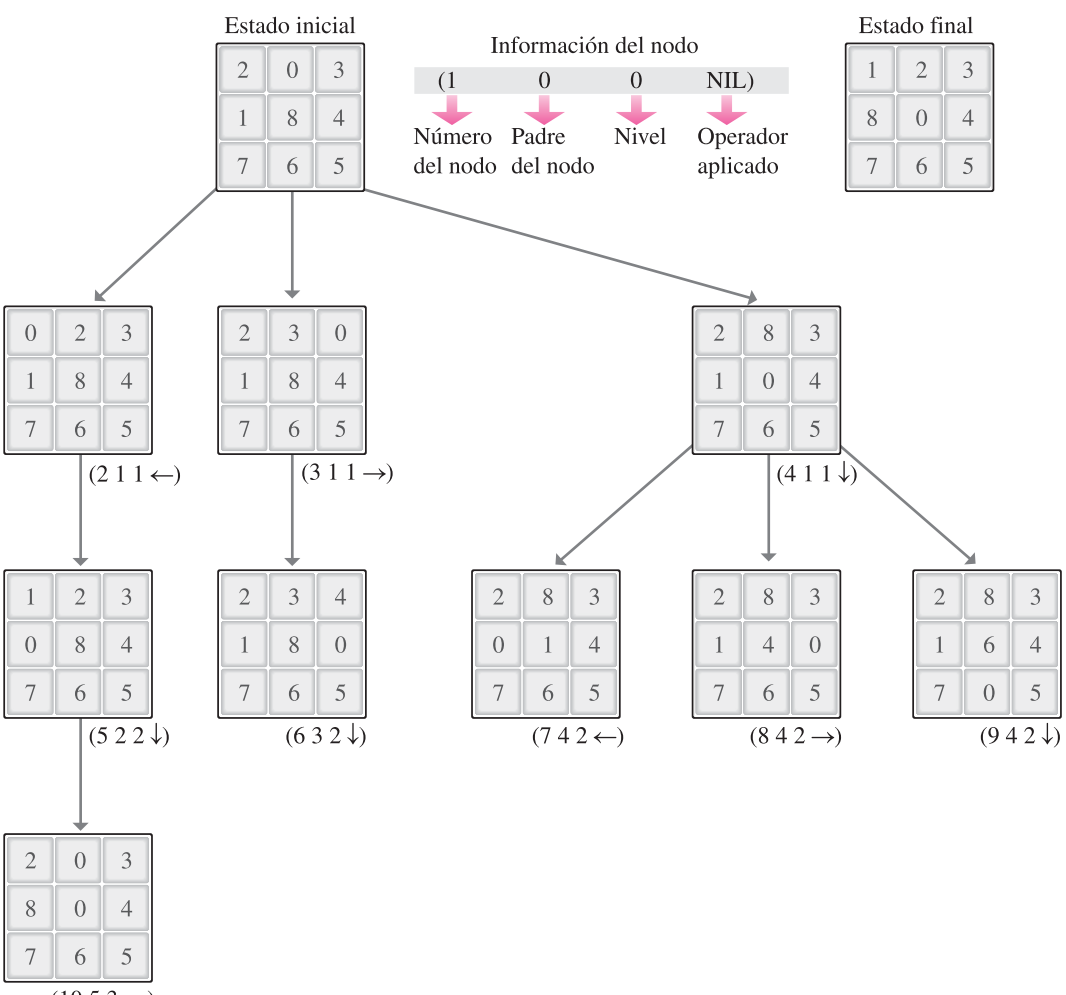
Nota: Se utiliza el 0 en lugar del rectángulo para indicar celda vacía, simplemente para mantener la homogeneidad de los datos.

Cabe recordar que se usan las listas ABIERTO y CERRADO para almacenar los nodos que se van a expandir en algún momento y los que ya han sido expandidos, respectivamente. Esta última lista también se utiliza para recuperar la trayectoria desde el estado inicial al estado meta, una vez que se alcanza esta última. También es importante remarcar que cada vez que se expande un nodo es necesario verificar que su contenido no se encuentre en la lista CERRADO, para evitar caer en ciclos infinitos. Si expandemos un nodo que ya se encuentra en CERRADO, entonces caemos en un ciclo, y además de ser muy difícil salir de él, es casi imposible encontrar la solución del problema —estado meta—.

La figura 7.25 presenta la solución al problema del puzzle-8. Los operadores se aplican siguiendo la forma de las manecillas del reloj: izquierda, arriba, derecha y abajo, ←, ↑, →, ↓. Se presentan a continuación las listas ABIERTO y CERRADO formadas durante la solución del problema. Las líneas horizontales sobre la información de los nodos indican que dicho nodo se quitó de la lista ABIERTO. Como se observa, en las dos listas se incorpora solamente la información del nodo por problemas de espacio. En una aplicación real es absolutamente necesario incluir el contenido completo del nodo.

Observe que para el estado inicial la información de ese nodo es (1 0 0 NIL). El primer número, 1, indica el número del nodo; el segundo, 0, indica su padre, el tercero, 0, el nivel en que nos encontramos y el último, NIL, el operador aplicado.

FIGURA 7.25
Solución al problema del puzzle-8.



ABIERTO	CERRADO
((~~1 0 0 NIL~~))	((1 0 0 NIL))
((~~2 1 1~~ ←)(3 1 1 →)(4 1 1 ↓)	((2 1 1 ←)(1 0 0 NIL))
((~~3 1 1~~ →)(4 1 1 ↓)(5 2 2 ↓))	((3 1 1 →)(2 1 1 ←)(1 0 0 NIL))
((~~4 1 1~~ ↓)(5 2 2 ↓)(6 3 2 ↓))	((4 1 1 ↓)(3 1 1 →)(2 1 1 ←)(1 0 0 NIL))
((~~5 2 2~~ ↓)(6 3 2 ↓)(7 4 2 ←)(8 4 2 →)(9 4 2 ↓))	((5 2 2 ↓)(4 1 1 ↓)(3 11 →)(2 1 1 ←)(1 0 0 NIL))

Una vez que se alcanza el estado meta se puede obtener información muy valiosa del campo *Información* del nodo. En este ejemplo: (10 5 3 →), el número **10** indica que se han generado (10 – 1) nodos. Asimismo, el **3** indica que el estado meta se encontró en el nivel 3 y, por lo tanto, se necesita aplicar tres operadores para resolver el problema. Por otra parte, la trayectoria se debe obtener de CERRADO de la siguiente manera:

(10 5 3 →)	padre de 10 → 5 (se busca en CERRADO el número de nodo 5)
(5 2 2 ↓)	padre de 5 → 2 (se busca en CERRADO el número de nodo 2)
(2 1 1 ←)	padre de 2 → 1 (se busca en CERRADO el número de nodo 1)
(1 0 0 NIL)	padre de 1 → 0 (se busca en CERRADO el número de nodo 0)

La solución en lenguaje de estados y operadores es:

$$\leftarrow \downarrow \rightarrow$$

Recuerde que en el lenguaje de estados y operadores, la solución a un problema consiste en encontrar la secuencia de operadores que permiten transformar el estado inicial en el final.

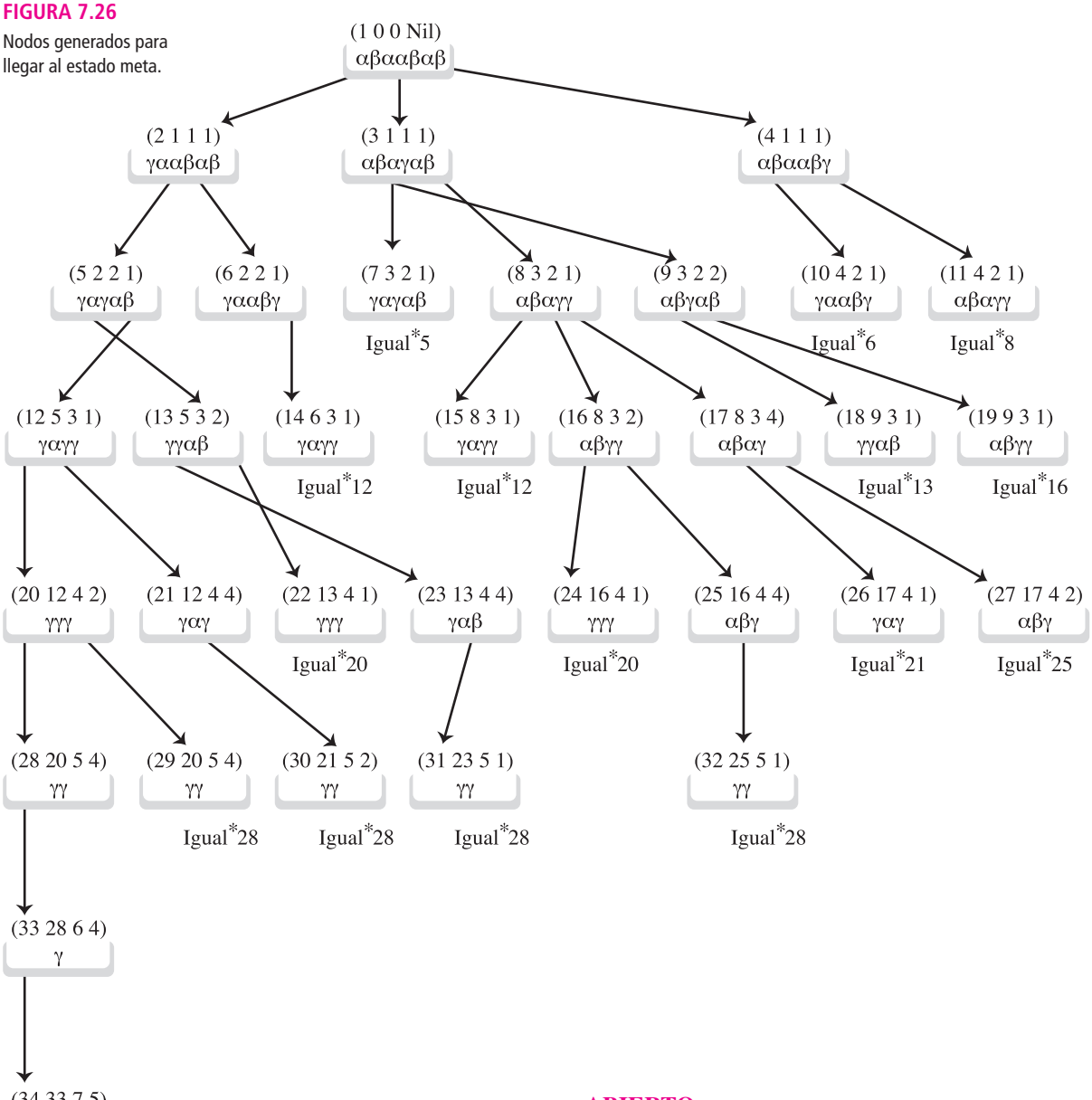
Ejemplo 7.10

A continuación se presenta otro ejemplo de uso del método de búsqueda *breadth-first*. En este caso, el método permite probar si una cadena dada como entrada fue generada o no por cierta gramática. Para este problema los operadores se definen en término de las siguientes reglas de reescritura:

1. $\alpha\beta \rightarrow \gamma$
2. $\alpha\gamma \rightarrow \gamma$
3. $\gamma\beta \rightarrow \gamma$
4. $\gamma\gamma \rightarrow \gamma$
5. $\gamma \rightarrow \Sigma$

En la figura 7.26 se presenta la gráfica con los nodos generados a partir del estado inicial $\boldsymbol{\alpha\beta\alpha\alpha\beta\alpha\beta}$ hasta alcanzar el estado meta $\boldsymbol{\Sigma}$. El * debajo de un nodo indica que dicho nodo ya existe y, por lo tanto, se elimina para evitar los ciclos. Como en el ejemplo anterior, las listas ABIERTO y CERRADO almacenan los nodos pendientes de ser expandidos y aquellos que ya han sido expandidos. Las líneas horizontales sobre algunos nodos indican que éstos fueron eliminados de ABIERTO por haber sido expandidos anteriormente, para no caer en ciclos infinitos.

FIGURA 7.26
Nodos generados para llegar al estado meta.



ABIERTO

((~~1 0 0 Nil~~))
((~~2 1 1 1~~)(3 1 1 1)(4 1 1 1))
((~~3 1 1 1~~)(4 1 1 1)(5 2 2 1)(6 2 2 1))
((~~4 1 1 1~~)(5 2 2 1)(6 2 2 1)(7 3 2 1)(8 3 2 1)(9 3 2 2))
((~~5 2 2 1~~)(6 2 2 1)(7 3 2 1)(8 3 2 1)(9 3 2 2)(10 4 2 1)(11 4 2 1))
((~~6 2 2 1~~)(7 3 2 1)(8 3 2 1)(9 3 2 2)(10 4 2 1)(11 4 2 1)(12 5 3 1)(13 5 3 2))
((~~7 3 2 1~~)(~~8 3 2 1~~)(9 3 2 2)(10 4 2 1)(11 4 2 1)(12 5 3 1)(13 5 3 2)(14 6 3 1))
*

((~~9 3 2 2~~) (10 4 2 1)(11 4 2 1)(12 5 3 1)(13 5 3 2)(14 6 3 1)(15 8 3 1)(16 8 3 2)(17 8 3 4))
((~~10 4 2 1~~)(~~11 4 2 1~~)(~~12 5 3 1~~)(13 5 3 2)(14 6 3 1)(15 8 3 1)(16 8 3 2)(17 8 3 4)(18 9 3 1)(19 9 3 1))
* *
((~~13 5 3 2~~)(14 6 3 1)(15 8 3 1)(16 8 3 2)(17 8 3 4)(18 9 3 1)(19 9 3 1)(20 12 4 2)(21 12 44))
((~~14 6 3 1~~)(~~15 8 3 1~~)(~~16 8 3 2~~)(17 8 3 4)(18 9 3 1)(19 9 3 1)(20 12 4 2)(21 12 4 4)(22 13 4 1)(23 13 4 4))
* *
((~~17 8 3 4~~)(18 9 3 1)(19 9 3 1)(20 12 4 2)(21 12 4 4)(22 13 4 1)(23 13 4 4)(24 16 4 1)(25 16 4 4))
((~~18 9 3 1~~)(~~19 9 3 1~~)(~~20 12 4 2~~)(21 12 4 4)(22 13 4 1)(23 13 4 4)(24 16 4 1)(25 16 4 4)(26 17 4 1)(27 17 4 2))
* *
((~~21 12 4 4~~)(22 13 4 1)(23 13 4 4)(24 16 4 1)(25 16 4 4)(26 17 4 1)(27 17 4 2)(28 20 5 4)(29 20 5 4))
((~~22 13 4 1~~)(~~23 13 4 4~~)(24 16 4 1)(25 16 4 4)(26 17 4 1)(27 17 4 2)(28 20 5 4)(29 20 5 4)(30 21 5 2))
*
((~~24 16 4 1~~)(25 16 4 4)(26 17 4 1)(27 17 4 2)(28 20 5 4)(29 20 5 4)(30 21 5 2)(31 23 5 1))
*
((~~26 17 4 1~~)(~~27 17 4 2~~)(~~28 20 5 4~~)(29 20 5 4)(30 21 5 2)(31 23 5 1)(32 25 5 1))
* *
((~~29 20 5 4~~)(~~30 21 5 2~~)(~~31 23 5 1~~)(~~32 25 5 1~~)(~~33 28 6 4~~))
* * *
↓
(34 33 7 5) Solución

CERRADO

((1 0 0 NIL))
((2 1 1 1)(1 0 0 NIL))
((3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((20 12 4 2)(17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((21 12 4 4)(20 12 4 2)(17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((23 13 4 4)(21 12 4 4)(20 12 4 2)(17 2 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((25 16 4 4)(23 13 4 4)(21 12 4 4)(20 12 4 2)(17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((28 20 5 4)(25 16 4 4)(23 13 4 4)(21 12 4 4)(20 12 4 2)(17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))
((33 28 6 4)(28 20 5 4)(25 16 4 4)(23 13 4 4)(21 12 4 4)(20 12 4 2)(17 8 3 4)(16 8 3 2)(13 5 3 2)(12 5 3 1)(9 3 2 2)(8 3 2 1)(6 2 2 1)(5 2 2 1)(4 1 1 1)(3 1 1 1)(2 1 1 1)(1 0 0 NIL))

La trayectoria que permite llegar a la solución es la siguiente:

(34 33 7 5), (33 28 6 4), (28 20 5 4), (20 12 4 2), (12 5 3 1), (2 5 2 1), (2 1 1 1), (1 0 0 NIL)

El estado meta es el nodo (34 33 7 5). El total de nodos generados es 33 y fueron necesarios siete niveles para alcanzar la solución. Los operadores —reglas de reescritura— aplicados para alcanzar el estado meta partiendo del estado inicial son:

$$
\begin{array}{ll}
1 & \alpha\beta \rightarrow \gamma \\
1 & \alpha\beta \rightarrow \gamma \\
1 & \alpha\beta \rightarrow \gamma \\
2 & \alpha\gamma \rightarrow \gamma \\
4 & \gamma\gamma \rightarrow \gamma \\
4 & \gamma\gamma \rightarrow \gamma \\
5 & \gamma \rightarrow \Sigma
\end{array}
$$

Ejemplo 7.11

Otro ejemplo de aplicación del método de búsqueda *breadth-first* es el conocido como *problema de las jarras de agua.* Se tienen dos jarras, una con capacidad para cuatro litros y otra para tres. Ninguna de ellas tiene marcas de medición. Además, se dispone de una bomba de agua que permite llenar las jarras de agua tantas veces como se requiera. El problema consiste en encontrar la forma de colocar exactamente dos litros de agua en la jarra de cuatro litros. En este problema se parte de un estado inicial y existen varias maneras de alcanzar el estado final, particularidad que lo convierte en un caso muy interesante.

Los elementos del problema se almacenan en una lista formada por las variables *X* y *Y*, las cuales a su vez representarán a las jarras de cuatro y tres litros de capacidad, respectivamente.

X: Jarra de cuatro litros, que puede tomar los valores: 0, 1, 2, 3, 4.
Y: Jarra de tres litros, que puede tomar los valores: 0, 1, 2, 3.

Para este problema los operadores válidos son los que se muestran en la tabla 7.7.

TABLA 7.7
Operadores para el problema de la jarra

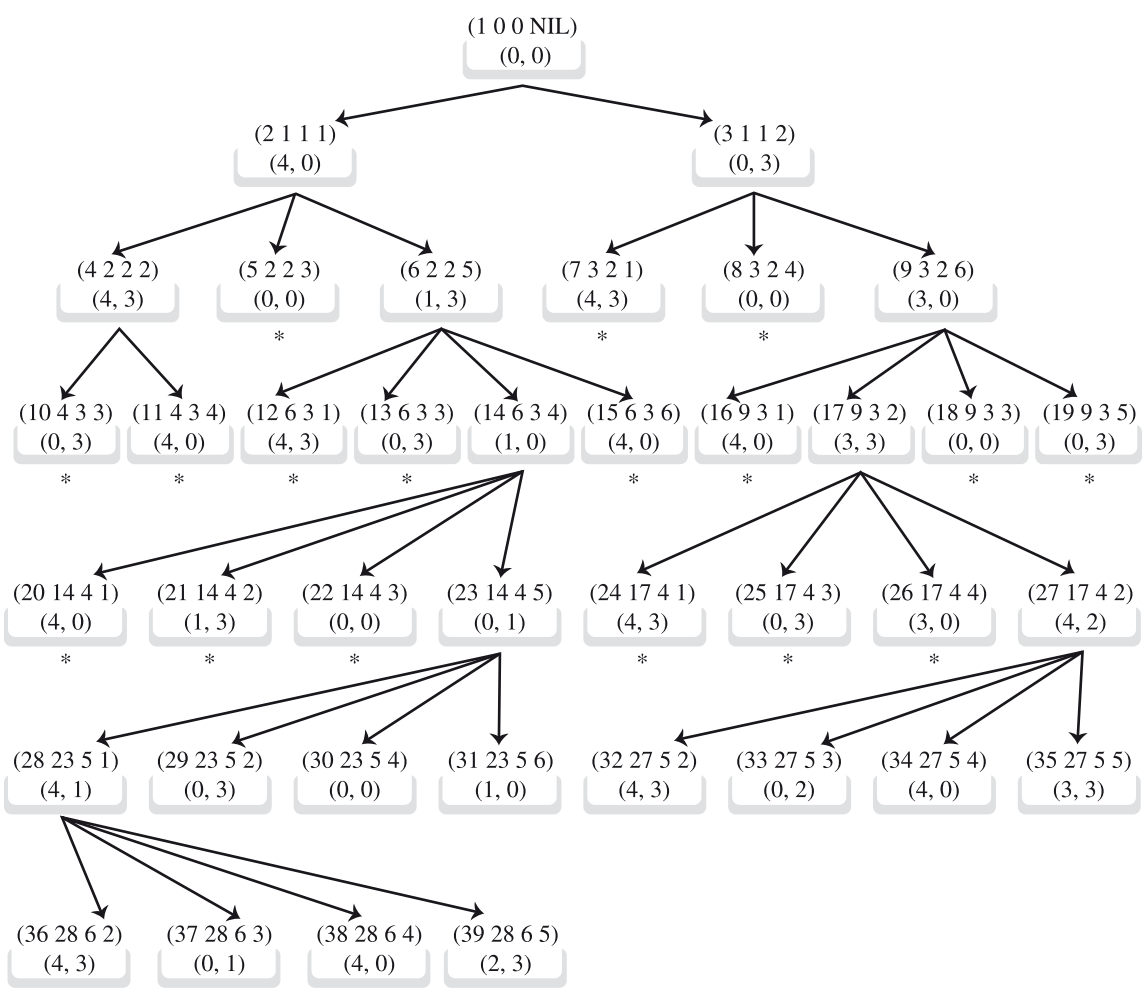
Identificador del operador	Descripción de la operación
1	Llenar la jarra de cuatro litros.
2	Llenar la jarra de tres litros.
3	Vaciar en el suelo la jarra de cuatro litros.
4	Vaciar en el suelo la jarra de tres litros.
5	Verter de la jarra de cuatro litros a la de tres, hasta que la segunda se llene o la primera quede vacía.
6	Verter de la jarra de tres litros a la de cuatro, hasta que la segunda se llene o la primera quede vacía.

El estado inicial y el final del problema se definen como:

Estado inicial: (0, 0) Ambas jarras están vacías.
Estado final: (2, *N*) La jarra *X* tiene dos litros y la jarra *Y* tiene *N*, donde *N* = 0, 1, 2, 3.

Observe que este problema muestra la particularidad de tener un estado inicial y múltiples estados finales —4—. En la figura 7.27 se presenta la gráfica con los nodos generados para alcanzar la solución. Como en los ejemplos anteriores, el asterisco se utiliza para indicar que dicho nodo ya fue generado y, por lo tanto, se ignora.

FIGURA 7.27
Nodos generados para alcanzar la solución.



A continuación se presentan las listas ABIERTO y CERRADO donde se guardan, respectivamente, los nodos generados y expandidos. Las líneas atravesadas sobre algunos nodos indican que éstos fueron eliminados por haber sido expandidos anteriormente. El asterisco debajo de un nodo indica que éste ya existe y, por lo tanto, se elimina para evitar caer en ciclos. La igualdad se detecta al comparar el contenido del nodo que se quiere expandir con el de aquellos que ya se encuentran en CERRADO.

ABIERTO

((~~1 0 0 NIL~~))
((~~2 1 1 1~~)(3 1 1 2))
((~~3 1 1 2~~)(4 2 2 2)(5 2 2 3)(6 2 2 5))
((~~4 2 2 2~~)(5 2 2 3)(6 2 2 5)(7 3 2 1)(8 3 2 4)(9 3 2 6))
((~~5 2 2 3~~)(~~6 2 2 5~~)(7 3 2 1)(8 3 2 4)(9 3 2 6)(10 4 3 3)(11 4 3 4))
*
((~~7 3 2 1~~)(~~8 3 2 4~~)(~~9 3 2 6~~)(10 4 3 3)(11 4 3 4)(12 6 3 1)(13 6 3 3)(14 6 3 4)(15 6 3 6))
* *
((~~10 4 3 3~~)(~~11 4 3 4~~)(~~12 6 3 1~~)(~~13 6 3 3~~)(~~14 6 3 4~~)(15 6 3 6)(16 9 3 1)(17 9 3 2)(18 9 3 3)(19 9 3 5))
* * * *
((~~15 6 3 6~~)(~~16 9 3 1~~)(~~17 9 3 2~~)(18 9 3 3)(19 9 3 5)(20 14 4 1)(21 14 4 2)(22 14 4 3)(23 14 4 5))
* *
((~~18 9 3 3~~)(~~19 9 3 5~~)(~~20 14 4 1~~)(~~21 14 4 2~~)(~~22 14 4 3~~)(~~23 14 4 5~~)(24 17 4 1)(25 17 4 3)(26 17 4 4)(27 17 4 6))
* * * * *
((~~24 17 4 1~~)(~~25 17 4 3~~)(~~26 17 4 4~~)(~~27 17 4 6~~)(28 23 5 1)(29 23 5 2)(30 23 5 4)(31 23 5 6))
* * *
((~~28 23 5~~ 1)(29 23 5 2)(30 23 5 4)(31 23 5 6)(32 27 5 2)(33 27 5 3)(34 27 5 4)(35 27 5 5))
↓
(39 28 6 5) Estado meta

CERRADO

((1 0 0 NIL)
((2 1 1 1)(1 0 0 NIL))
((3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((14 6 3 4)(9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((17 9 3 2)(14 6 3 4)(9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((23 14 4 5)(17 9 3 2)(14 6 3 4)(9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((27 17 4 6)(23 14 4 5)(17 9 3 2)(14 6 3 4)(9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))
((28 23 5 1)(27 17 4 6)(23 14 4 5)(17 9 3 2)(14 6 3 4)(9 3 2 6)(6 2 2 5)(4 2 2 2)(3 1 1 2)(2 1 1 1)(1 0 0 NIL))

El estado meta se encuentra en el nodo (39 28 6 5). El total de nodos generados es 38 (39 – 1) y el total de niveles es **6**. La trayectoria que describe la solución se recupera de la lista CERRADO:

(39 28 6 5), (28 23 5 1), (23 14 4 5), (14 6 3 4), (6 2 2 5), (2 5 2 1), (2 1 1 1), (1 0 0 NIL)

Considerando la descripción de las operaciones asociadas a los operadores —tabla 7.7—, la solución en lenguaje de estados y operadores es:

1 5 4 5 1 5

Esta secuencia de operadores indica que primero se debe llenar la jarra *X* —requiere cuatro litros— y luego verter esa agua en la jarra *Y* —sólo se usarán tres de los cuatro litros—. El siguiente operador indica que se debe vaciar la jarra *Y*, arrojando el agua que contiene al piso. Posteriormente se coloca el litro restante de la jarra *X* en la jarra *Y*, quedando la primera vacía y la segunda con un litro. Luego se llena la jarra *X* —usando cuatro litros—. Por último, se pasa agua de la jarra *X* a la jarra *Y* hasta que ésta se llene —lo cual se logra con sólo dos litros—; por lo tanto, la jarra *X* se queda con los otros dos litros, alcanzando así el estado meta.

Complejidad del método *breadth-first*

La complejidad del método *breadth-first* es $O(b^d)$, donde b representa el factor de ramificación del nodo y d la profundidad del árbol. Suponiendo que $b = 10$, la velocidad de expansión de 1 000 nodos por segundo y la capacidad de almacenamiento 100 bytes por nodo, en la tabla 7.8 se observa en tiempo y espacio la complejidad del método *breadth-first*.

Es importante destacar que cuando se utiliza el método *breadth-first* se debe encontrar la solución en los primeros seis niveles, porque de otra forma aparecerán serios problemas en cuanto a tiempo y espacio. Observe que en el nivel 8 ya se necesitan 31 horas para resolver el problema y 11 gigabytes.

7.6.4 Método de búsqueda *depth-first*

En el método de búsqueda ***depth-first***, conocido con el nombre de **búsqueda en profundidad** en el mundo de habla hispana, se expande el nodo más recientemente generado;

TABLA 7.8
Complejidad *breadth-first* $O(b^d)$

Profundidad	Nodos	Tiempo	Memoria
0	1	1 milisegundo	100 bytes
2	111	0.1 segundo	11Kb
4	11 111	11 segundos	1Mb
6	10^6	18 minutos	111Mb
8	10^8	31 horas	11Gb
10	10^{10}	128 días	1Tb
12	10^{12}	35 años	111Tb
14	10^{14}	3 500 años	11 111Tb

esto último permite realizar una búsqueda en profundidad en lugar de hacerlo en forma horizontal como en el método de búsqueda a lo ancho. La profundidad del nodo inicial es cero y la de un nodo que no es inicial es igual a uno más la profundidad de su padre.

Normalmente se establece un límite máximo de profundidad permitido, que a su vez establece el número máximo de niveles que se pueden generar en la búsqueda de la solución. Si se llega al límite establecido sin haber alcanzado el estado meta, entonces se considera que el problema no tiene solución. A continuación se presenta el algoritmo correspondiente.

Algoritmo 7.8 *Depth-first*

Depth-First

{Este método permite encontrar el estado meta de un problema, a partir de un estado inicial y usando los operadores permitidos para dicho problema. *P* es un entero que indica el límite de profundidad permitido. ABIERTO y CERRADO son dos listas lineales simplemente ligadas}

1. Insertar el nodo inicial en la lista llamada ABIERTO.
2. *Mientras* (ABIERTO tenga elementos) y (no se haya llegado al estado final). *Repetir*
Quitar el primer nodo *N* de ABIERTO.
2.1 *Si* (*N* no está en CERRADO) y (su profundidad es ≤ *P*) *entonces*
Insertar el nodo *N* en la lista CERRADO
Expandir el nodo *N* obteniendo todos sus sucesores
2.1.1 Si (hay sucesores y no son el estado meta) *entonces*
Almacenarlos al inicio de la lista ABIERTO
2.1.2 {Fin del condicional del paso 2.1.1}
2.2 {Fin del condicional del paso 2.1}
3. {Fin del ciclo del paso 2}
4. *Si* (alguno de los nodos generados es el estado meta)
entonces {Éxito}
Desplegar la trayectoria desde el estado inicial al final
si no {Fracaso}
No se encontró el estado final
5. {Fin del condicional del paso 4}

En este algoritmo se maneja un valor adicional, *P*, que representa la profundidad máxima permitida. Al resolver el problema se verifica si el nodo ya tiene esa profundidad. En caso afirmativo, se elimina de la lista ABIERTO y se aplica *backtracking*; es decir, se continúa el análisis con el nodo inmediatamente anterior del árbol de derivación. Además, es importante destacar que mientras en el método *breadth-first* los nodos generados se almacenan al final de ABIERTO, en el método *depth-first* se colocan al inicio.

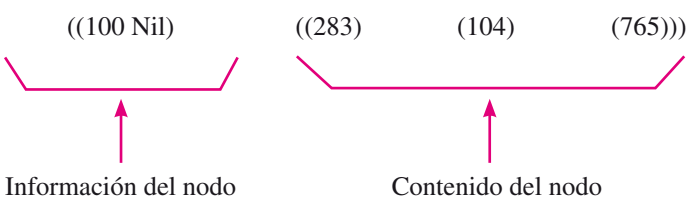
Ejemplo 7.12

A continuación se presenta un ejemplo de aplicación del método de búsqueda *depth-first*. Se retoma el problema del puzzle-8, pero ahora se soluciona por medio de este método. Se parte de un estado inicial y se define un estado final al cual se quiere llegar.

Estado inicial		
2	8	3
1	0	4
7	6	5

Estado final		
1	2	3
8	0	4
7	6	5

El estado inicial se representa de la siguiente manera:



Además se establece un límite de profundidad igual a **4**. En la figura 7.28 se presenta la gráfica con los nodos generados hasta llegar a la solución. Las líneas punteadas indican *backtracking*, un método que deshace parte de la trayectoria generada cuando ésta no permite llegar a la solución.

A continuación se presentan las listas ABIERTO y CERRADO donde se guardan, respectivamente, los nodos generados y los expandidos. Las líneas horizontales indican que dichos nodos fueron eliminados ya sea por haber sido expandidos anteriormente o por haber llegado a la profundidad límite establecida. El asterisco debajo de un nodo indica que éste ya existe o que su profundidad es igual a la máxima establecida; por lo tanto, se elimina para evitar caer en ciclos.

ABIERTO

((~~1 0 0 NIL~~))
((~~2 1 1 ←~~)(3 1 1 ↑)(4 1 1 →)(5 1 1 ↓))
((~~6 2 2 ↑~~)(7 2 2 ↓)(3 1 1 ↑)(4 1 1 →)(5 1 1 ↓))
((~~8 6 3 →~~)(7 2 2 ↓)(3 1 1 ↑)(4 1 1 →)(5 1 1 ↓))
((~~9 8 4 →~~)(~~10 8 4 ↓~~)(~~7 2 2 ↓~~)(3 1 1 ↑)(4 1 1 →)(5 1 1 ↓))
* *
((~~11 7 3 →~~)(3 1 1 ↑)(4 1 1 →)(5 1 1 ↓))
((~~12 11 4 →~~)(~~13 11 4 →~~)(~~3 1 1 ↑~~)(4 1 1 →)(5 1 1 ↓))
* *
((~~14 3 2 ←~~)(15 3 2 →)(4 1 1 →)(5 1 1 ↓))
((~~16 14 3 ↓~~)(15 3 2 →)(4 1 1 →)(5 1 1 ↓))
↓
(17 16 4 →) Estado meta

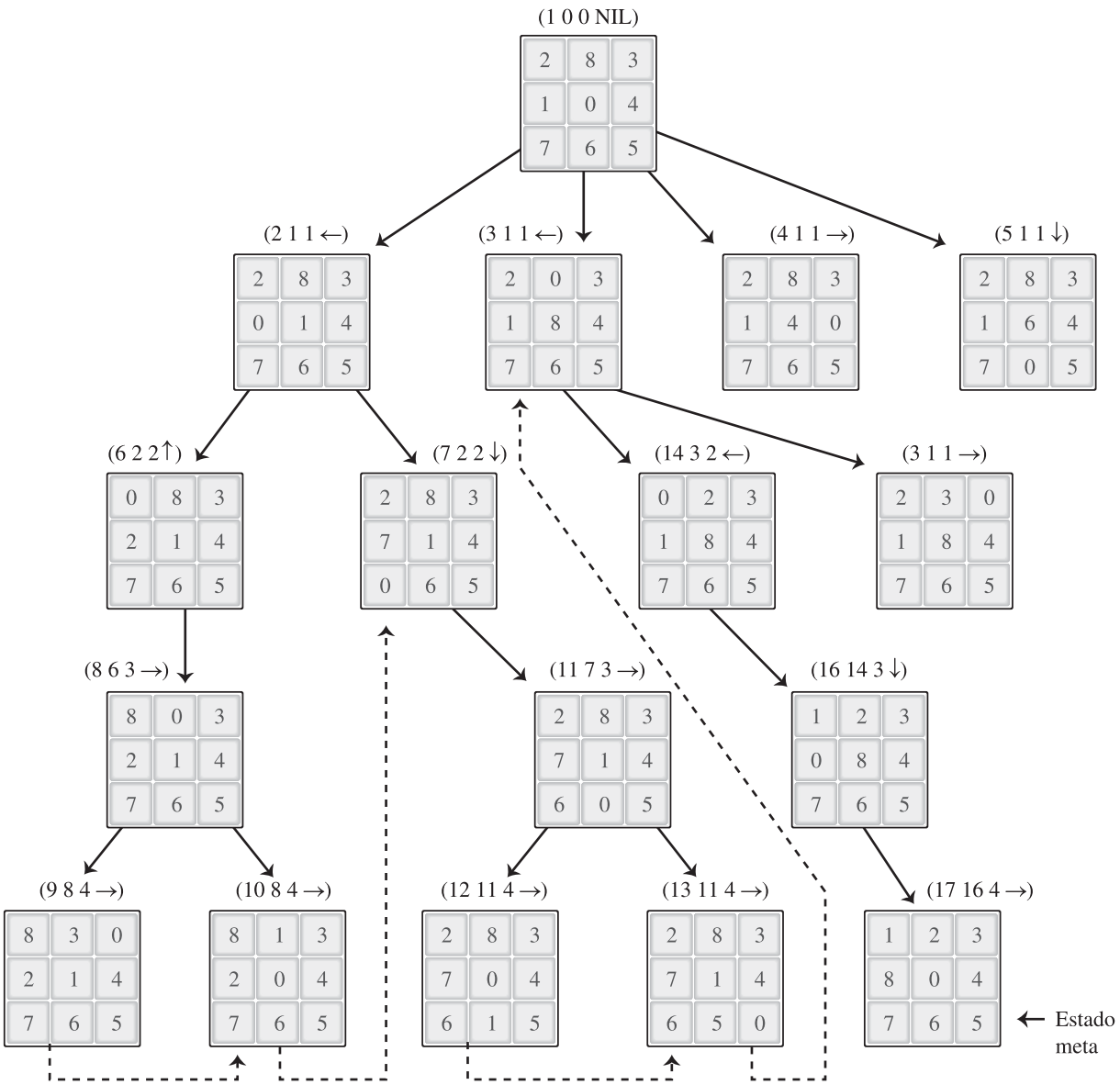


FIGURA 7.28
Solución al problema del puzzle-8 aplicando *depth-first*.

CERRADO
((1 0 0 NIL))
((2 1 1 ←)(1 0 0 NIL))
((6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((7 2 2 ↓)(8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((11 7 3 →)(7 2 2 ↓)(8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((3 1 1 ↑)(11 7 3 →)(7 2 2 ↓)(8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((14 3 2 ←)(3 1 1 ↑)(11 7 3 →)(7 2 2 ↓)(8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))
((16 14 3 ↓)(14 3 2 ←)(3 1 1 ↑)(11 7 3 →)(7 2 2 ↓)(8 6 3 →)(6 2 2 ↑)(2 1 1 ←)(1 0 0 NIL))

El estado meta se encuentra en el nodo (17 16 4 →). El total de nodos generados es **16** (17 – 1) y **4** es el nivel donde encontramos el estado meta. La trayectoria que describe la solución se recupera de la lista CERRADO:

(17 16 4 →),(16 14 3 ↓),(14 3 2 ←),(3 1 1 ↑),(1 0 0 NIL)

La solución expresada en lenguaje de estados y operadores queda de la siguiente manera:

↑ ← ↓ →

Los métodos de búsqueda analizados, *breadth-first* y *depth-first*, se conocen como ciegos, ya que son métodos exhaustivos. En principio, estos métodos proporcionan una solución para encontrar una trayectoria, pero son poco prácticos porque expanden demasiados nodos; además, sabemos que existen límites en cuanto a tiempo y espacio —memoria—. Usando información especial del problema y su representación se puede aumentar la velocidad. Esa información se denomina información heurística.

En el campo de la solución de problemas, **heurística** significa acelerar el proceso de búsqueda hacia la meta mediante la expansión de los nodos más promisorios. Uno de los temas que ocupa a la inteligencia artificial es precisamente el estudio de métodos heurísticos.

En resumen, los métodos analizados pueden mejorar considerablemente su desempeño si les incorporamos conocimiento y heurística. Uno de los métodos heurísticos de mejor comportamiento se conoce como A*.

7.7 LA CLASE GRÁFICA

Para definir la **clase gráfica** se requiere determinar sus atributos y los métodos necesarios para su manejo. Los atributos son los vértices y aristas, indicando en estas últimas si tienen dirección y costo. Para su representación se puede utilizar cualquiera de las estructuras presentadas. En cuanto a los métodos, éstos serán los que permitan encontrar un vértice, imprimir la información de vértices y aristas, así como encontrar trayectorias según el tipo de gráfica que se esté representando.

Es recomendable que se defina una clase por tipo de gráfica. Es decir, una para las gráficas dirigidas y otra para las gráficas no dirigidas. En cada una de ellas se deberá incluir, como métodos, los algoritmos estudiados en este capítulo.

EJERCICIOS

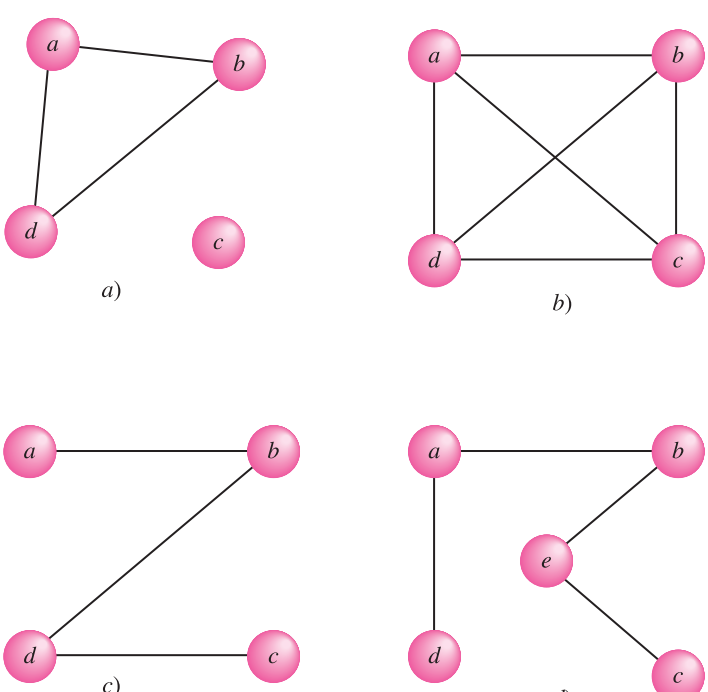
1. Para cada una de las gráficas de la figura 7.29 indique:

a) Aquellas que son gráficas conectadas.
b) Aquellas que son gráficas cíclicas.
c) Aquellas que son gráficas conexas.
d) Aquellas que son gráficas completas.
e) Todos los pares de vértices adyacentes.
f) Un camino entre los vértices *a* y *c*, si es posible.
g) Un camino cerrado entre cualquier par de vértices, si es posible.
h) Un camino simple entre cualquier par de vértices, si es posible.
i) El grado de cada vértice.

2. Utilice una matriz de adyacencia y una lista similar para representar las gráficas de la figura 7.30*a* y *b*.

3. Utilice una matriz de adyacencia y una lista de adyacencia para representar la digráfica de la figura 7.31.

FIGURA 7.29



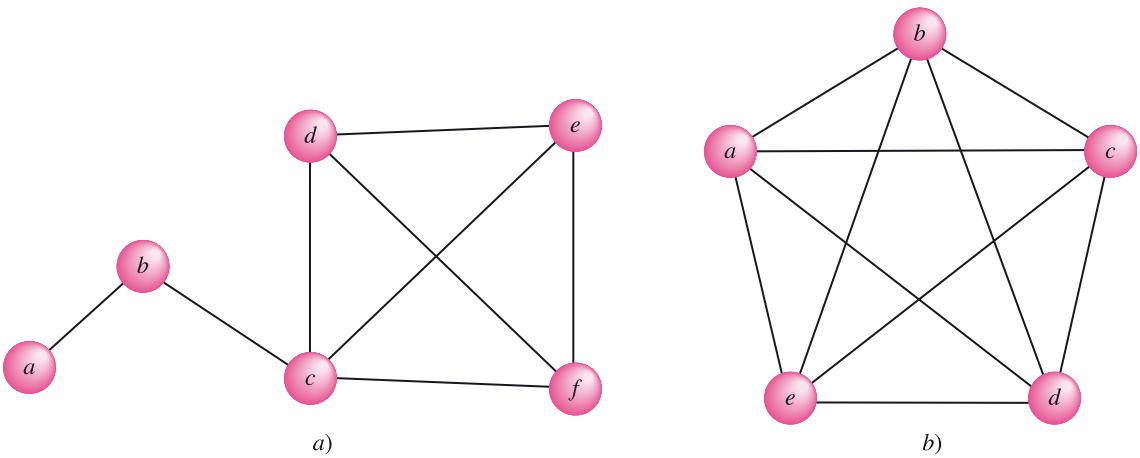


FIGURA 7.30

4. Utilice una matriz de costos para representar la gráfica de la figura 7.32.

5. Utilice una matriz de costos para representar las digráficas de la figura 7.33*a* y *b*.

6. Dada la digráfica representada en la figura 7.34, indique cuáles de las siguientes sucesiones de índices describen un camino en ella:

a) 1 2 5 3
b) 5 3 4 1
c) 3 4 2 3
d) 1 2 5 3
e) 2 5 1 4 3

FIGURA 7.31

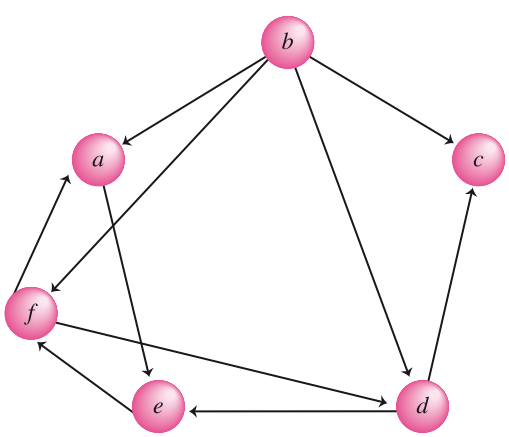
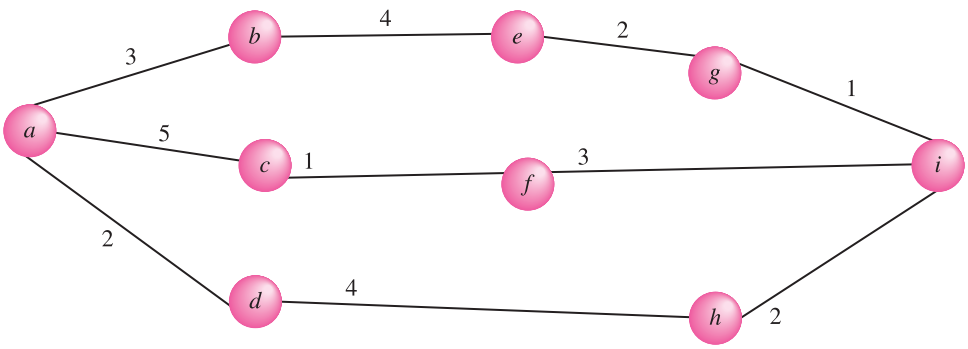


FIGURA 7.32



7. Dada la digráfica de la figura 7.34, encuentre un camino acíclico de:

a) 1 a 5
b) 2 a 1
c) 3 a 2
d) 4 a 3
e) 5 a 4

8. Dada la digráfica de la figura 7.35:

a) Encuentre la trayectoria más corta del vértice *a* a todos los otros vértices.
b) Utilice una matriz de costos para representarla.

FIGURA 7.33

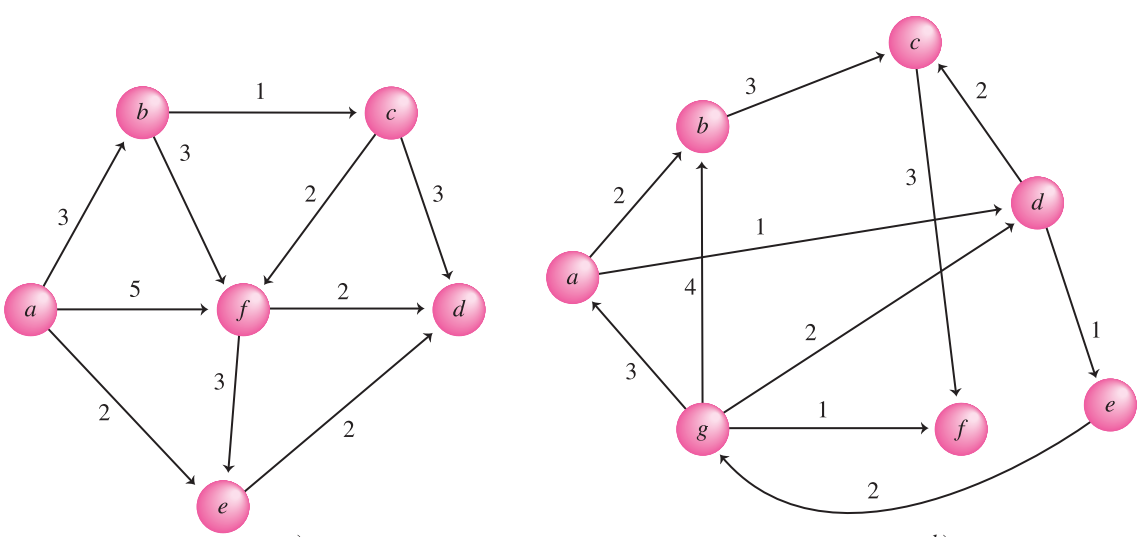
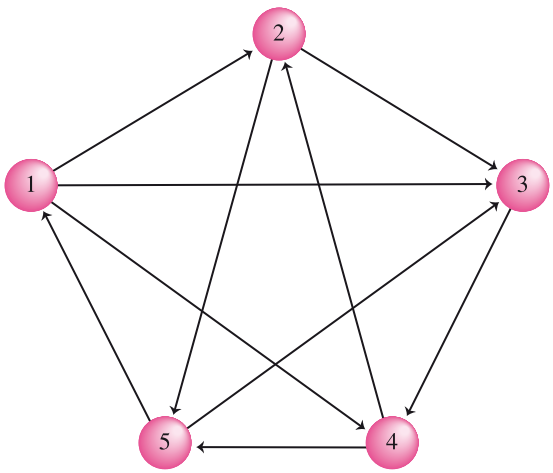


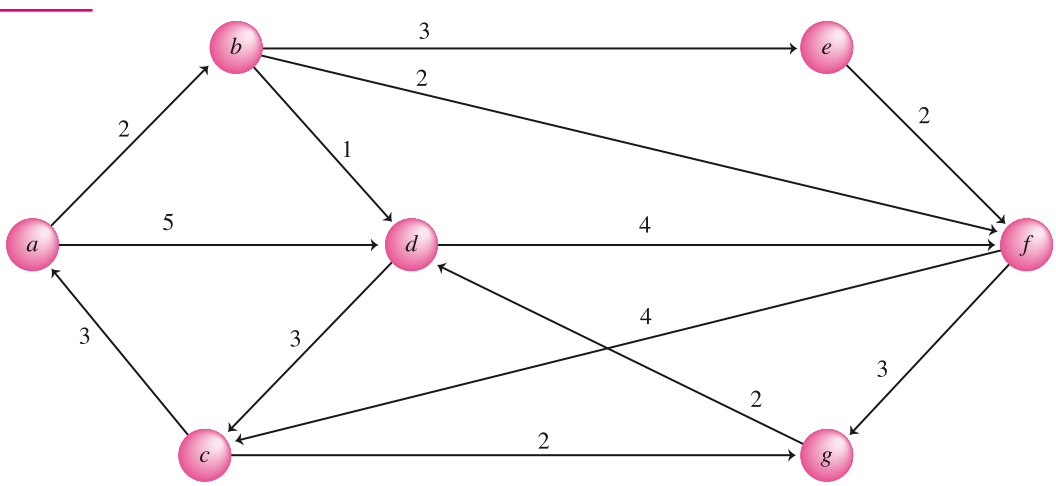
FIGURA 7.34



9. Dada la siguiente matriz de adyacencia, dibuje la gráfica correspondiente.

	a	*b*	*c*	*d*	*e*
a	1	1	1	1	1
b	1	0	1	1	1
c	1	1	0	1	0
d	1	1	1	0	1
e	1	1	1	1	0

FIGURA 7.35



10. Dada la siguiente matriz de adyacencia, dibuje la digráfica correspondiente.

	a	*b*	*c*	*d*	*e*	*f*
a	0	0	1	0	1	0
b	1	0	1	1	0	1
c	0	0	0	0	1	0
d	0	0	1	0	1	0
e	0	0	0	0	0	1
f	1	0	0	1	0	0

11. Dada la siguiente matriz de costos, dibuje la digráfica correspondiente.

	a	*b*	*c*	*d*	*e*	*f*	*g*
a	0	2	∞	5	∞	∞	∞
b	∞	0	∞	1	3	2	∞
c	3	∞	0	∞	∞	3	2
d	∞	∞	3	0	∞	4	∞
e	∞	5	∞	∞	0	2	∞
f	∞	∞	4	∞	∞	0	3
g	∞	∞	∞	2	∞	∞	0

12. Aplique el algoritmo de Dijkstra a la digráfica de la figura 7.33*b*. Tome el vértice *a* como vértice origen. Construya la tabla correspondiente al seguimiento del algoritmo.

13. Aplique el algoritmo de Floyd a la digráfica de la figura 7.35. Construya la tabla correspondiente al seguimiento del algoritmo.

14. Aplique el algoritmo de Warshall a la digráfica de la figura 7.35. Construya la tabla correspondiente al seguimiento del algoritmo.

15. Aplique el algoritmo de Prim a la gráfica de la figura 7.32. Construya la tabla correspondiente al seguimiento del algoritmo.

16. Aplique el algoritmo de Kruskal a la gráfica de la figura 7.32. Construya la tabla correspondiente al seguimiento del algoritmo.

17. Escriba un algoritmo que permita almacenar una gráfica por medio de:

a) Una matriz de adyacencia.
b) Una lista de adyacencia.

18. Escriba un algoritmo que permita almacenar una digráfica por medio de una matriz de costos.

19. Escriba un algoritmo que permita, dada una gráfica almacenada por medio de una matriz de adyacencia, imprimir todos los pares de vértices adyacentes.

20. Escriba una versión modificada del algoritmo de Dijkstra que permita elegir el camino con el menor número de aristas, en caso de trayectorias con igual costo.

21. Escriba un algoritmo que permita eliminar las aristas necesarias para obtener, como resultado, una gráfica acíclica.

22. Considere que hay cuatro tipos de sangre: *A*, *B*, *AB* y *O*. Además, se sabe que el tipo *O* es compatible ("puede donar a") con cualquiera de los cuatro tipos; el tipo *A* es compatible con su tipo y con el tipo *AB*; el tipo *B* es compatible con su tipo y con el tipo *AB*, y el tipo *AB* sólo puede donar a su mismo tipo. Utilice una gráfica para representar esta información. ¿Qué tipo de gráfica será la más apropiada? Justifique su respuesta.

23. Piense en la receta para preparar su platillo favorito. Las operaciones involucradas, junto con el tiempo requerido para su realización, pueden representarse por medio de una gráfica. En ella se pueden indicar las tareas a realizar simultáneamente, así como aquellas que deben seguir cierta seriación. Utilice una gráfica y represéntela, considerando que:

a) Usted no cuenta con ayuda y, por lo tanto, va a llevar a cabo cada una de las actividades requeridas.
b) Usted cuenta con tantos ayudantes como desee y, por lo tanto, ciertas tareas se pueden realizar paralelamente.

24. Se quiere representar una topología de una red telefónica, donde se distinguen centrales de conmutación y enlaces entre las centrales. Los enlaces entre las centrales son bidireccionales y con diferentes capacidades de transmisión (canales de voz). Utilice una gráfica para representar una red de este tipo. ¿Qué tipo de gráfica será la más apropiada? ¿Cómo se modifica la gráfica si cambian las capacidades? ¿Qué operaciones sobre esta gráfica podrían ser de interés? Justifique sus respuestas.

25. Se tiene un alfabeto que consiste en todas las palabras binarias de tres bits. Al transmitirlos a través de un canal con ruido se originan cambios, y, por lo tanto, se origina transición entre las palabras transmitidas. Por ejemplo, si la palabra transmitida es 010, la recibida podría ser 000. Suponga que en cada palabra transmitida puede haber sólo un dígito con error. Con una gráfica represente todas las palabras del alfabeto y las transiciones que pueden originarse a otras palabras al transmitirlas a través del canal con ruido. ¿Qué tipo de gráfica será la más apropiada? Justifique su respuesta.

26. Se tienen tres jarras de agua con capacidades de cinco, tres y siete litros. Ninguna de ellas presenta marcas de medición. Se tiene una bomba que permite llenar las jarras de agua. ¿Cómo se pueden colocar exactamente cuatro (4) litros de agua en la jarra de cinco litros de capacidad?

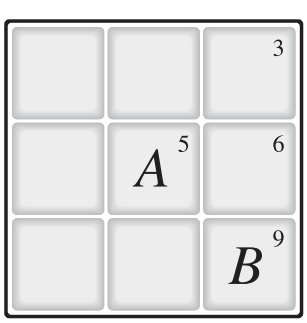
27. Tres misioneros y tres caníbales se encuentran sobre la orilla de un río. Todos quieren llegar a la otra orilla, pero únicamente hay un bote para dos personas. Los misioneros, para no correr el riesgo de ser comidos, quieren que su número nunca sea menor que el de caníbales en el mismo lado del río. ¿Cómo pueden cruzar todos sin que los misioneros estén en peligro?

28. Muestre que la cadena (((), ()), (), ((), ())) pertenece al lenguaje generado por la gramática G, aplicando las siguientes reglas de reescritura:

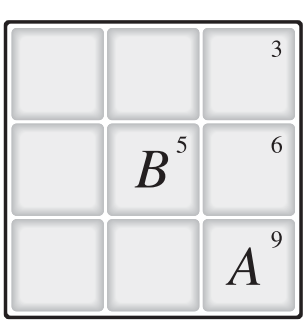
1. $S \leftarrow ()$
2. $A \leftarrow S$
3. $A \leftarrow A, A$
4. $S \leftarrow (A)$

29. Considere el estado inicial, el estado final y los operadores que se dan a continuación. Encuentre la solución al problema.


Operadores



Estado inicial



Estado final

30. Defina la clase *Gráfica*, correspondiente a una gráfica no dirigida, utilizando una matriz de costos para almacenar las aristas y sus costos.

31. Retome el problema anterior e incluya los métodos necesarios para implementar los algoritmos de Prim y Kruskal.

32. Defina la clase *Digráfica*, correspondiente a una gráfica dirigida, utilizando una lista de adyacencia para almacenar las aristas y sus costos. Puede reusar la clase *Listas* del capítulo 5.

33. Retome el problema anterior e incluya en la clase los métodos necesarios para implementar los algoritmos de Dijkstra, Floyd y Warshall.

Capítulo 8

MÉTODOS DE ORDENACIÓN

8.1 INTRODUCCIÓN

Ordenar significa reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica. Los procesos de ordenación y búsqueda —este último se estudiará en el siguiente capítulo— son frecuentes en nuestra vida. Vivimos en un mundo desarrollado, automatizado, acelerado, donde la información representa un elemento de vital importancia. La sociedad debe estar informada y, por lo tanto, constantemente se necesita buscar y recuperar información.

La operación de búsqueda —recuperación— de información normalmente se efectúa sobre elementos ordenados, lo que demuestra que, en general, donde haya objetos que se deban buscar y recuperar estará presente el proceso de ordenación.

Los objetos ordenados aparecen por doquier. Directorios telefónicos, registros de pacientes de un hospital, registros de huéspedes de un hotel, índices de libros de una biblioteca, son tan sólo algunos ejemplos de objetos ordenados con los que el ser humano se encuentra frecuentemente. Incluso y de manera informal se puede señalar que desde niño se nos enseña a ser organizado, a poner las cosas en orden.

La ordenación es una actividad fundamental y relevante en la vida. Imagine el lector qué ocurriría si se deseara encontrar un libro en una biblioteca con más de 100 000 volúmenes y éstos estuvieran desordenados o registrados en los índices en el orden en el cual fueron recibidos; o, por ejemplo, si se quisiera hablar por teléfono con una persona y se encontrara que en el directorio los abonados están ordenados según su número telefónico, en forma ascendente o descendente. La tarea sería mayúscula, pero sin ningún sentido.

Formalmente se define **ordenación** de la siguiente manera:

Sea A una lista de N elementos:

$$A_1, A_2, A_3, ..., A_N$$

Ordenar significa permutar estos elementos de tal forma que queden de acuerdo con una distribución preestablecida.

- Ascendente: $A_1 \leq A_2 \leq A_3 \leq ... \leq A_N$
- Descendente: $A_1 \geq A_2 \geq A_3 \geq ... \geq A_N$

En el procesamiento de datos a los métodos de ordenación se les clasifica en dos grandes categorías, según donde hayan sido almacenados:

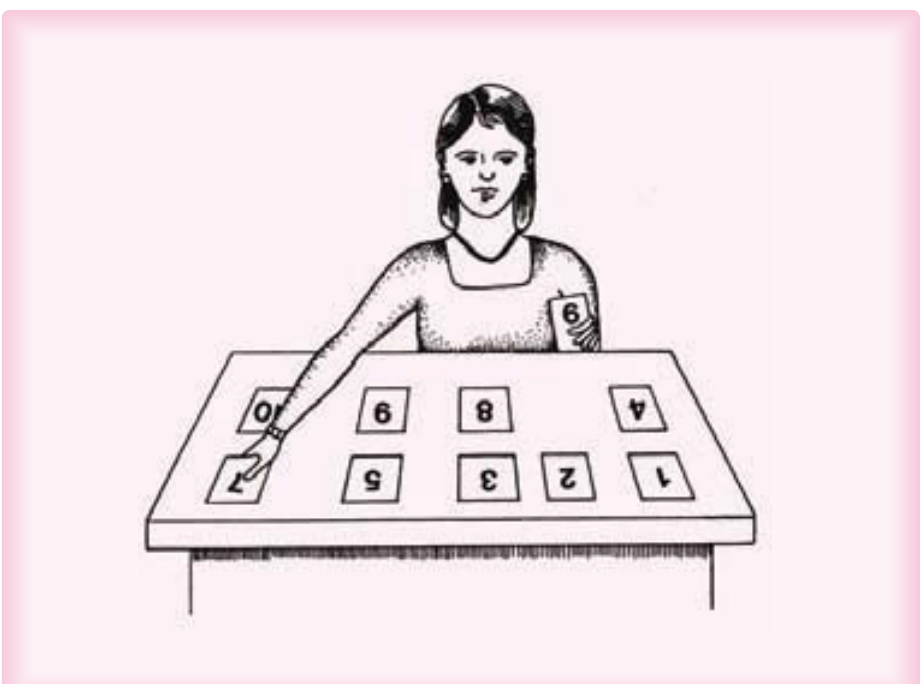
- Ordenación de arreglos.
- Ordenación de archivos.

La primera categoría se denomina también **ordenación interna**, ya que los elementos o componentes del arreglo se encuentran en la memoria principal de la computadora. La segunda categoría se llama **ordenación externa**, ya que los elementos se encuentran en archivos almacenados en dispositivos de almacenamiento secundario, como discos, cintas, tambores, etcétera.

Si se buscara una analogía entre los métodos de ordenación y la vida real, se podría mencionar que para la máquina, la ordenación interna representa lo que para un humano significa ordenar un conjunto de tarjetas que se encuentran visibles y extendidas todas sobre una mesa. La ordenación externa, en cambio, representa para la máquina lo que para un humano significa ordenar un conjunto de tarjetas que están dispuestas una debajo de otra y en donde sólo se visualiza la primera.

En la primera parte de este capítulo se estudiarán los métodos más importantes de ordenación interna y posteriormente los más interesantes de ordenación externa.

FIGURA 8.1
Ordenación interna.



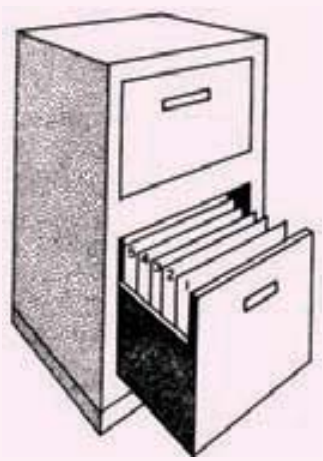


FIGURA 8.2
Ordenación externa.

8.2 ORDENACIÓN INTERNA

Los métodos de **ordenación interna** se explicarán con arreglos unidimensionales, pero su uso puede extenderse a otros tipos de arreglos y estructuras de datos. Es importante señalar, además, que se trabajará con métodos de ordenación *in situ*, es decir, métodos que no requieren de arreglos auxiliares para su ordenación. Los métodos que requieren de arreglos auxiliares son generalmente ineficientes e intrínsecamente de menor interés.

Los métodos de ordenación interna a su vez se pueden clasificar en dos tipos:

- Métodos directos (n^2).
- Métodos logarítmicos ($n * \log n$).

Los **métodos directos** tienen la característica de que su implementación es relativamente sencilla y son fáciles de comprender, aunque son ineficientes cuando N —el número de elementos del arreglo— es de tamaño mediano o grande. Los **métodos logarítmicos**, por su parte, son más complejos que los directos. Su elaboración es más sofisticada y, al ser menos intuitivos, resultan más difíciles de entender. Sin embargo, son más eficientes ya que requieren de menos comparaciones y movimientos para ordenar sus elementos.

Es importante destacar que una buena medida de eficiencia entre los distintos métodos la constituye el tiempo de ejecución del algoritmo y éste depende fundamentalmente del número de comparaciones y movimientos que se realicen entre sus elementos.

Como conclusión se puede señalar que cuando N es pequeño se deben utilizar métodos directos y cuando N es mediano o grande se usarán métodos logarítmicos.

Los métodos directos más conocidos son:

- Ordenación por intercambio.
- Ordenación por inserción.
- Ordenación por selección.

8.2.1 Ordenación por intercambio directo (burbuja)

El método de **intercambio directo**, conocido coloquialmente como **burbuja**, es el más utilizado entre los estudiantes principiantes de computación por su fácil comprensión y programación. Pero es preciso señalar que es quizás el método más ineficiente.

El método de intercambio directo puede trabajar de dos maneras diferentes: llevando los elementos más pequeños hacia la parte izquierda del arreglo o trasladando los elementos más grandes hacia su parte derecha. La idea básica de este algoritmo consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados. Se realizan $(n - 1)$ pasadas transportando en cada una de ellas el menor o mayor de elementos —según sea el caso— a su posición ideal. Al final de las $(n - 1)$ pasadas los elementos del arreglo estarán ordenados.

Ejemplo 8.1

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A, transportando en cada pasada el menor elemento hacia la parte izquierda del arreglo.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[7] > A[8]$	$(12 > 35)$	no hay intercambio
$A[6] > A[7]$	$(27 > 12)$	sí hay intercambio
$A[5] > A[6]$	$(44 > 12)$	sí hay intercambio
$A[4] > A[5]$	$(16 > 12)$	sí hay intercambio
$A[3] > A[4]$	$(08 > 12)$	no hay intercambio
$A[2] > A[3]$	$(67 > 08)$	sí hay intercambio
$A[1] > A[2]$	$(15 > 08)$	sí hay intercambio

Luego de la primera pasada el arreglo queda así:

A: 08 15 67 12 16 44 27 35

Observe que el elemento más pequeño, en este caso 08, fue situado en la parte izquierda del arreglo.

SEGUNDA PASADA

$A[7] > A[8]$	$(27 > 35)$	no hay intercambio
$A[6] > A[7]$	$(44 > 27)$	sí hay intercambio
$A[5] > A[6]$	$(16 > 27)$	no hay intercambio

$A[4] > A[5]$ (12 > 16) no hay intercambio
$A[3] > A[4]$ (67 > 12) sí hay intercambio
$A[2] > A[3]$ (15 > 12) sí hay intercambio

Luego de la segunda pasada el arreglo queda así:

A: 08 12 15 67 16 27 44 35

y el segundo elemento más pequeño del arreglo, en este caso 12, fue situado en la segunda posición.

En la tabla 8.1 se presenta el resultado de las pasadas restantes.

TABLA 8.1

3a. pasada:	08	12	15	16	67	27	35	44
4a. pasada:	08	12	15	16	27	67	35	44
5a. pasada:	08	12	15	16	27	35	67	44
6a. pasada:	08	12	15	16	27	35	44	67
7a. pasada:	08	12	15	16	27	35	44	67

El algoritmo de ordenación por el método de intercambio directo que transporta en cada pasada el *menor elemento* hacia la parte izquierda del arreglo es el siguiente:

Algoritmo 8.1 Burbuja_menor

Burbuja_menor (*A*, *N*)

{Este algoritmo ordena los elementos del arreglo unidimensional utilizando el método de la burbuja. Transporta en cada pasada el elemento más pequeño hacia la parte izquierda del arreglo. *A* es un arreglo unidimensional de *N* elementos}
{*I*, *J* y AUX son variables de tipo entero}

1. *Repetir* con *I* desde 2 hasta *N*
  1.1 *Repetir* con *J* desde *N* hasta *I*
    1.1.1 *Si* $A(J-1) > A[J]$ *entonces*
      Hacer AUX ← $A[J-1]$, $A[J-1] \leftarrow A[I]$ y $A[I] \leftarrow$ AUX
    1.1.2 {Fin del condicional del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Ejemplo 8.2

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional *A* transportando en cada pasada el mayor elemento hacia la parte derecha del arreglo:

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[1] > A[2]$	(15 > 67)	no hay intercambio
$A[2] > A[3]$	(67 > 08)	sí hay intercambio
$A[3] > A[4]$	(67 > 16)	sí hay intercambio
$A[4] > A[5]$	(67 > 44)	sí hay intercambio
$A[5] > A[6]$	(67 > 27)	sí hay intercambio
$A[6] > A[7]$	(67 > 12)	sí hay intercambio
$A[7] > A[8]$	(67 > 35)	sí hay intercambio

A: 15 08 16 44 27 12 35 [67]

Observe que el elemento más grande, en este caso 67, fue situado en la última posición del arreglo.

SEGUNDA PASADA

$A[1] > A[2]$	(15 > 08)	sí hay intercambio
$A[2] > A[3]$	(15 > 16)	no hay intercambio
$A[3] > A[4]$	(16 > 44)	no hay intercambio
$A[4] > A[5]$	(44 > 27)	sí hay intercambio
$A[5] > A[6]$	(44 > 12)	sí hay intercambio
$A[6] > A[7]$	(44 > 35)	sí hay intercambio

A: 08 15 16 27 12 35 [44] [67]

y el segundo elemento más grande del arreglo, en este caso 44, fue situado en la penúltima posición. En la tabla 8.2 se ve el resultado de las pasadas restantes.

TABLA 8.2

3a. pasada:	08	15	16	12	27	35	44	67
4a. pasada:	08	15	12	16	27	35	44	67
5a. pasada:	08	12	15	16	27	35	44	67
6a. pasada:	08	12	15	16	27	35	44	67
7a. pasada:	08	12	15	16	27	35	44	67

El algoritmo de ordenación por el método de intercambio directo que transporta en cada pasada el elemento mayor hacia la parte derecha del arreglo es:

Algoritmo 8.2 Burbuja_mayor

Burbuja_mayor (*A*, *N*)

{El algoritmo ordena los elementos del arreglo unidimensional *A*. Transporta en cada pasada el elemento más grande hacia la parte derecha del arreglo. *A* es un arreglo de *N* elementos}
{*I*, *J* y AUX son variables de tipo entero}

1. *Repetir* con *I* desde *N* – 1 hasta 1
  1.1 *Repetir* con *J* desde 1 hasta *I*
    1.1.1 *Si* *A*[*J*] > *A* [*J* + 1] *entonces*
      Hacer AUX ← *A*[*J*], *A*[*J*] ← *A*[*J* + 1] y *A*[*J* + 1] ← AUX
    1.1.2 {Fin del condicional del paso 1.1.1}
  1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

Análisis de eficiencia del método de intercambio directo

El número de comparaciones que se realizan en el método de la burbuja se puede contabilizar fácilmente. En la primera pasada se realizan $(n - 1)$ comparaciones, en la segunda pasada $(n - 2)$ comparaciones, en la tercera pasada $(n - 3)$ comparaciones y así sucesivamente hasta llegar a 2 y 1 comparaciones entre claves, siendo n el número de elementos del arreglo. Por lo tanto, tenemos que el número de comparaciones es:

$$C = (n-1)+(n-2)+\ldots+2+1 = \frac{n*(n-1)}{2}$$

que es igual a:

$$C = \frac{n^2 - n}{2} \qquad \textbf{Fórmula 8.1}$$

Como ya se mencionó en el capítulo 2, se hace uso del principio de inducción matemática para desarrollar ciertas fórmulas.

Respecto del número de movimientos, éstos dependen fundamentalmente de si el arreglo se encuentra ordenado, desordenado o en orden inverso. Los movimientos para cada uno de estos casos son:

$$M_{mín} = 0 \qquad M_{med} = 0.75 * (n^2 - n) \qquad M_{máx} = 1.5 * (n^2 - n) \qquad \textbf{Fórmula 8.2}$$

Así, por ejemplo, si se tiene que ordenar un arreglo que contiene 500 elementos, el número de movimientos que se tendrá que realizar es:

a) Si el arreglo se encuentra ordenado:
- 124 750 comparaciones
- 0 movimientos

b) Si los elementos del arreglo se encuentran dispuestos en forma aleatoria:
- 124 750 comparaciones
- 187 125 movimientos

c) Si los elementos del arreglo se encuentran en orden inverso:
- 124 750 comparaciones
- 374 250 movimientos

Ahora bien, el tiempo necesario para ejecutar el algoritmo de la burbuja es proporcional a n^2, $O(n^2)$, donde n es el número de elementos del arreglo.

A continuación se presentarán dos variantes del método de intercambio directo: método de intercambio directo con señal y método de *shaker sort*.

8.2.2 Ordenación por el método de intercambio directo con señal

Este método es una modificación del método de intercambio directo analizado en la sección anterior. La idea central de este algoritmo consiste en utilizar una marca o señal para indicar que no se ha producido ningún intercambio en una pasada. Es decir, se comprueba si el arreglo está totalmente ordenado después de cada pasada, terminando su ejecución en caso afirmativo. El algoritmo de ordenación por el método de la burbuja con señal es:

Algoritmo 8.3 Burbuja_señal

Burbuja_señal (*A*, *N*)

{El algoritmo ordena los elementos del arreglo utilizando el método de la burbuja con señal. *A* es un arreglo unidimensional de *N* elementos}
{*I*, *J* y AUX son variables de tipo entero. BAND es una variable de tipo booleano}

1. Hacer $I \leftarrow 1$ y BAND ← FALSO
2. *Mientras* (($I \leq N - 1$) y (BAND = FALSO)) *Repetir*
Hacer BAND ← VERDADERO
2.1 *Repetir* con *J* desde 1 hasta $N - 1$
2.1.1 *Si* ($A[J] > A[J + 1]$) *entonces*
Hacer AUX ← $A[J]$, $A[J] \leftarrow A[J + 1]$, $A[J + 1]$ ← AUX
y BAND ← FALSO
2.1.2 {Fin del condicional del paso 2.1.1}
2.2 {Fin del ciclo del paso 2.1}
Hacer $I \leftarrow I + 1$
3. {Fin del ciclo del paso 2}

A continuación presentamos la segunda variante del método de intercambio directo.

8.2.3 Ordenación por el método de la sacudida (*shaker sort*)

El método de *shaker sort,* más conocido como el método de la sacudida, es una optimización del método de intercambio directo. La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método de la burbuja.

En este algoritmo cada pasada tiene dos etapas. En la primera etapa, de derecha a izquierda, se trasladan los elementos más pequeños hacia la parte izquierda del arreglo, almacenando en una variable la posición del último elemento intercambiado. En la segunda etapa, de izquierda a derecha, se trasladan los elementos más grandes hacia la parte derecha del arreglo, almacenando en otra variable la posición del último elemento intercambiado. Las sucesivas pasadas trabajan con los componentes del arreglo comprendidos entre las posiciones almacenadas en las variables auxiliares. El algoritmo termina cuando en una etapa no se producen intercambios, o bien cuando el contenido de la variable que guarda el extremo izquierdo del arreglo es mayor que el contenido de la variable que almacena el extremo derecho.

Ejemplo 8.3

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional *A* utilizando el método de la sacudida.

A: 15 67 08 16 44 27 12 35

PRIMERA PASADA

Primera etapa (de derecha a izquierda)

$A[7] > A[8]$	(12 > 35)	no hay intercambio
$A[6] > A[7]$	(27 > 12)	sí hay intercambio
$A[5] > A[6]$	(44 > 12)	sí hay intercambio
$A[4] > A[5]$	(16 > 12)	sí hay intercambio
$A[3] > A[4]$	(08 > 12)	no hay intercambio
$A[2] > A[3]$	(67 > 08)	sí hay intercambio
$A[1] > A[2]$	(15 > 08)	sí hay intercambio

Última posición de intercambio de derecha a izquierda: 2.

Luego de la primera etapa de la primera pasada, el arreglo queda así:

A: 08 15 67 12 16 44 27 35

Segunda etapa (de izquierda a derecha)

$A[2] > A[3]$	(15 > 67)	no hay intercambio
$A[3] > A[4]$	(67 > 12)	sí hay intercambio
$A[4] > A[5]$	(67 > 16)	sí hay intercambio
$A[5] > A[6]$	(67 > 44)	sí hay intercambio
$A[6] > A[7]$	(67 > 27)	sí hay intercambio
$A[7] > A[8]$	(67 > 35)	sí hay intercambio

Última posición de intercambio de izquierda a derecha: 8.

Luego de la segunda etapa de la primera pasada, el arreglo queda así:

A: 08 15 12 16 44 27 35 67

SEGUNDA PASADA

Primera etapa (de derecha a izquierda)

$A[6] > A[7]$	(27 > 35)	no hay intercambio
$A[5] > A[6]$	(44 > 27)	sí hay intercambio
$A[4] > A[5]$	(16 > 27)	no hay intercambio
$A[3] > A[4]$	(12 > 16)	no hay intercambio
$A[2] > A[3]$	(15 > 12)	sí hay intercambio

Última posición de intercambio de derecha a izquierda: 3.

A: 08 12 15 16 27 44 35 67

Segunda etapa (de izquierda a derecha)

$A[3] > A[4]$	(15 > 16)	no hay intercambio
$A[4] > A[5]$	(16 > 27)	no hay intercambio
$A[5] > A[6]$	(27 > 44)	no hay intercambio
$A[6] > A[7]$	(44 > 35)	sí hay intercambio

Última posición de intercambio de izquierda a derecha: 7.

A: 08 12 15 16 27 34 44 67

Al realizar la primera etapa de la tercera pasada se observa que no se realizaron intercambios; por lo tanto, la ejecución del algoritmo se termina. El algoritmo de ordenación por el método de la sacudida es el siguiente:

Algoritmo 8.4 Sacudida

Sacudida (*A*, *N*)

{El algoritmo ordena los elementos de un arreglo unidimensional utilizando el método de la sacudida. *A* es un arreglo de *N* elementos}
{*I*, IZQ, DER, *K* y AUX son variables de tipo entero}

1. Hacer IZQ ← 2, DER ← *N* y *K* ← *N*
2. *Mientras* (DER ≥ IZQ) *Repetir*
  2.1 *Repetir* con *I* desde DER hasta IZQ {Ciclo descendente}

```
    2.1.1  Si (A[I – 1] > A[I]) entonces
               Hacer AUX ← A[I – 1], A[I – 1] ← A[I], A[I] ← AUX y K ← I
    2.1.2  {Fin del condicional del paso 2.1.1}
2.2 {Fin del ciclo del paso 2.1}
    Hacer IZQ ← K + 1
2.3 Repetir con I desde IZQ hasta DER {Ciclo ascendente}
  2.3.1 Si (A[I – 1] > A[I]) entonces
            Hacer AUX ← A[I – 1], A[I – 1] ← A[I], A[I] ← AUX y K ← I
  2.3.2 {Fin del condicional del paso 2.2.1}
2.4 {Fin del ciclo del paso 2.3}
    Hacer DER ← K – 1
3. {Fin del ciclo 2}
```

Análisis de eficiencia del método de la sacudida

El análisis del método de la sacudida, y en general el de los métodos mejorados y logarítmicos, es muy complejo. Para el análisis de este método es necesario tener en cuenta tres factores que afectan directamente al tiempo de ejecución del algoritmo: las comparaciones entre las claves, los intercambios entre ellas y las pasadas que se realizan. Encontrar fórmulas que permitan calcular cada uno de estos factores es una tarea muy difícil de realizar.

Los estudios que se han efectuado sobre el método de la sacudida demuestran que en él sólo se pueden reducir las dobles comparaciones entre claves, pero se debe recordar que la operación de intercambio es una tarea más complicada y costosa que la de comparación. Por lo tanto, es posible afirmar que las hábiles mejoras realizadas sobre el método de intercambio directo sólo producen resultados apreciables si el arreglo está parcialmente ordenado, lo cual resulta difícil saber de antemano; pero si el arreglo está desordenado el método se comporta, incluso, peor que otros métodos directos, como los de inserción y selección.

8.2.4 Ordenación por inserción directa

El método de ordenación por **inserción directa** es el que utilizan generalmente los jugadores de cartas cuando las ordenan, de ahí que también se conozca con el nombre de método de la baraja.

La idea central de este algoritmo consiste en insertar un elemento del arreglo en su parte izquierda, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el *n*-ésimo elemento. Observemos un ejemplo.

Ejemplo 8.4

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional *A* utilizando el método de inserción directa:

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[2] < A[1]$ (67 < 15) no hay intercambio

A: 15 67 08 16 44 27 12 35

SEGUNDA PASADA

$A[3] < A[2]$ (08 < 67) sí hay intercambio
$A[2] < A[1]$ (08 < 15) sí hay intercambio

A: 08 15 67 16 44 27 12 35

TERCERA PASADA

$A[4] < A[3]$ (16 < 67) sí hay intercambio
$A[3] < A[2]$ (16 < 15) no hay intercambio

A: 08 15 16 67 44 27 12 35

Observe que una vez que se determina la posición correcta del elemento se interrumpen las comparaciones. Por ejemplo, para el caso anterior no se realizó la comparación $A[2] < A[1]$. En la tabla 8.3 se presenta el resultado de las pasadas restantes.

El algoritmo de ordenación por el método de inserción directa es:

Algoritmo 8.5 Inserción

Inserción (*A*, *N*)

{Este algoritmo ordena los elementos del arreglo utilizando el método de inserción directa. *A* es un arreglo unidimensional de *N* elementos}
{*I*, AUX y *K* son variables de tipo entero}

1. *Repetir* con *I* desde 2 hasta *N*
  Hacer AUX ← *A*[*I*] y *K* ← *I* – 1
 1.1 Mientras ((*K* ≥ 1) y (AUX < *A*[*K*])) *Repetir*
  Hacer *A*[*K* + 1] ← *A*[*K*] y *K* ← *K* – 1
 1.2 {Fin del ciclo del paso 1.1}
  Hacer *A*[*K* + 1] ← AUX
2. {Fin del ciclo del paso 1}

TABLA 8.3

4a. pasada:	08	15	16	44	67	27	12	35
5a. pasada:	08	15	16	27	44	67	12	35
6a. pasada:	08	12	15	16	27	44	67	35
7a. pasada:	08	12	15	16	27	35	44	67

Análisis de eficiencia del método de inserción directa

El número mínimo de comparaciones y movimientos entre claves se produce cuando los elementos del arreglo ya están ordenados. Analice el siguiente caso:

Ejemplo 8.5

Sea *A* un arreglo formado por los elementos:

A: 15 20 45 52 86

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[2] < A[1]$ (20 < 15) no hay intercambio

SEGUNDA PASADA

$A[3] < A[2]$ (45 < 20) no hay intercambio

TERCERA PASADA

$A[4] < A[3]$ (52 < 45) no hay intercambio

CUARTA PASADA

$A[5] < A[4]$ (86 < 52) no hay intercambio

Luego de las comparaciones realizadas, el arreglo correspondiente queda así:

A: 15 20 45 52 86

Observe que para este ejemplo se efectuaron cuatro comparaciones. En general, podemos afirmar que si el arreglo se encuentra ordenado se efectúan como máximo $n - 1$ comparaciones y 0 movimientos entre elementos.

$$C_{mín} = n - 1 \qquad \textbf{Fórmula 8.3}$$

El número máximo de comparaciones y movimientos entre elementos se produce cuando los elementos del arreglo están en orden inverso.

Ejemplo 8.6

Sea *A* un arreglo formado por los siguientes elementos:

A: 86 52 45 20 15

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[2] < A[1]$ (52 < 86) sí hay intercambio

SEGUNDA PASADA

$A[3] < A[2]$ (45 < 86) sí hay intercambio
$A[2] < A[1]$ (45 < 52) sí hay intercambio

TERCERA PASADA

$A[4] < A[3]$ (20 < 86) sí hay intercambio
$A[3] < A[2]$ (20 < 52) sí hay intercambio
$A[2] < A[1]$ (20 < 45) sí hay intercambio

CUARTA PASADA

$A[5] < A[4]$ (15 < 86) sí hay intercambio
$A[4] < A[3]$ (15 < 52) sí hay intercambio
$A[3] < A[2]$ (15 < 45) sí hay intercambio
$A[2] < A[1]$ (15 < 20) sí hay intercambio

Observe que en la primera pasada se realizó una comparación; en la segunda, dos comparaciones; en la tercera, tres comparaciones, y así sucesivamente hasta $n - 1$ comparaciones entre elementos. Por lo tanto:

$$M_{\text{máx}} = 1 + 2 + 3 + \ldots + (n-1) = \frac{n*(n-1)}{2}$$

que es igual a

$$C_{\text{máx}} = \frac{(n^2 - n)}{2}$$ **Fórmula 8.4**

Ahora bien, el número de comparaciones promedio, que es cuando los elementos aparecen en el arreglo en forma aleatoria, se puede calcular mediante la suma de las comparaciones mínimas y máximas dividida entre 2.

$$C_{\text{med}} = \frac{\left[(n-1) + \frac{(n^2 - n)}{2}\right]}{2}$$

Al hacer la operación queda:

$$C_{\text{med}} = \frac{\left(n^2 + n - 2\right)}{4}$$ **Fórmula 8.5**

Respecto del número de movimientos, si el arreglo se encuentra ordenado no se realiza ninguno. Por lo tanto:

$$M_{\text{mín}} = 0$$ **Fórmula 8.6**

El número máximo de movimientos se presenta cuando el arreglo está en orden inverso. Observe el ejemplo anterior. En la primera pasada se realizó un movimiento, en la segunda dos y así sucesivamente hasta $n - 1$ movimientos entre los elementos en la última pasada. Por lo tanto:

$$M_{\text{máx}} = 1 + 2 + 3 + \ldots + \left(n - 1\right) = \frac{n * \left(n - 1\right)}{2}$$

que es igual a

$$M_{\text{máx}} = \frac{\left(n^2 - n\right)}{2}$$ **Fórmula 8.7**

El número de movimientos promedio, que se da cuando los elementos se encuentran en el arreglo en forma aleatoria, se calcula como la suma de los movimientos mínimos y máximos dividida entre 2. Por lo tanto:

$$M_{\text{med}} = \frac{0 + \dfrac{\left(n^2 - n\right)}{2}}{2}$$

Al hacer la operación queda:

$$M_{\text{med}} = \frac{n^2 - n}{4}$$ **Fórmula 8.8**

Así, por ejemplo, si se tiene que ordenar un arreglo que contiene 500 elementos:

a) Si el arreglo se encuentra ordenado serán necesarias:
- 499 comparaciones
- 0 movimientos

b) Si los elementos del arreglo se encuentran dispuestos en forma aleatoria se realizarán:

- 62 624 comparaciones, en promedio.
- 62 375 movimientos, en promedio.

c) Si los elementos del arreglo se encuentran en orden inverso serán necesarias:
- 124 750 comparaciones.
- 124 750 movimientos.

Es importante señalar que el tiempo requerido para ejecutar el algoritmo de inserción directa es proporcional a n^2, $O(n^2)$, donde n es el número de elementos del arreglo.

A pesar de ser un método ineficiente y recomendable sólo cuando n es pequeño, el método de inserción directa se comporta mejor que los métodos de intercambio directo analizados anteriormente.

8.2.5 Ordenación por el método de inserción binaria

El método de ordenación por **inserción binaria** es una mejora del método de inserción directa presentado anteriormente. La mejora consiste en realizar una búsqueda binaria en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado. El proceso, al igual que en el método de inserción directa, se repite desde el segundo hasta el n-ésimo elemento. Analicemos un ejemplo.

Ejemplo 8.7

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional *A* utilizando el método de inserción binaria.

A: 15 67 08 16 44 27 12 35

A continuación se presentan las comparaciones que se llevan a cabo:

PRIMERA PASADA

$A[2] < A[1]$ (67 < 15) no hay intercambio

A: [15] [67] 08 16 44 27 12 35

SEGUNDA PASADA

$A[3] < A[1]$ (08 < 15) sí hay intercambio

A: [08] [15] [67] 16 44 27 12 35

TERCERA PASADA

$A[4] < A[2]$ (16 < 15) no hay intercambio
$A[4] < A[3]$ (16 < 67) sí hay intercambio

A: [08] [15] [16] [67] 44 27 12 35

CUARTA PASADA

$A[5] < A[2]$ (44 < 15) no hay intercambio
$A[5] < A[3]$ (44 < 16) no hay intercambio
$A[5] < A[4]$ (44 < 67) sí hay intercambio

A: 08 15 16 44 67 27 12 35

QUINTA PASADA

$A[6] < A[3]$ (27 < 16) no hay intercambio
$A[6] < A[4]$ (27 < 44) sí hay intercambio

A: 08 15 16 27 44 67 12 35

SEXTA PASADA

$A[7] < A[3]$ (12 < 16) sí hay intercambio
$A[7] < A[1]$ (12 < 08) no hay intercambio
$A[7] < A[2]$ (12 < 15) sí hay intercambio

A: 08 12 15 16 27 44 67 35

SÉPTIMA PASADA

$A[8] < A[4]$ (35 < 16) no hay intercambio
$A[8] < A[6]$ (35 < 44) sí hay intercambio
$A[8] < A[5]$ (35 < 27) no hay intercambio

A: 08 12 15 16 27 35 44 67

El algoritmo de ordenación por el método de inserción binaria es:

Algoritmo 8.6 Inserción_binaria

Inserción_binaria (*A*, *N*)

{Este algoritmo ordena los elementos de un arreglo unidimensional utilizando el método de inserción binaria. *A* es un arreglo unidimensional de *N* elementos}
{*I*, AUX, IZQ, DER, *M* y *J* son variables de tipo entero}

1. *Repetir* con *I* desde 2 hasta *N*
Hacer AUX ← *A*[*I*], IZQ ← 1 y DER ← *I* – 1
1.1 Mientras (IZQ ≤ DER) *Repetir*
Hacer M ← parte entera ((IZQ + DER) entre 2)

```
        1.1.1 Si (AUX ≤ A[M])
                  entonces
                      Hacer DER ← M – 1
                  si no
                      Hacer IZQ ← M + 1
        1.1.2 {Fin del condicional del paso 1.1.1}
    1.2 {Fin del ciclo del paso 1.1}
        Hacer J ← I – 1
    1.3 Mientras (J ≥ IZQ) Repetir
              Hacer A[J + 1] ← A[J] y J ← J – 1
    1.4 {Fin del ciclo del paso 1.3}
        Hacer A[IZQ] ← AUX
2. {Fin del ciclo del paso 1}
```

Análisis de eficiencia del método de inserción binaria

Al analizar el método de ordenación por inserción binaria se advierte la presencia de un caso antinatural. El método efectúa el menor número de comparaciones cuando el arreglo está totalmente desordenado y el máximo cuando se encuentra ordenado.

Es posible suponer que mientras en una búsqueda secuencial se necesitan K comparaciones para insertar un elemento, en una binaria se necesitará la mitad de las K comparaciones. Por lo tanto, el número de comparaciones promedio en el método de ordenación por inserción binaria se puede calcular como:

$$C = \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \ldots + \frac{(n-1)}{2} = \frac{n*(n-1)}{4}$$

que es igual a:

$$C = \frac{(n^2 - n)}{4} \qquad \textbf{Fórmula 8.9}$$

Éste es un algoritmo de comportamiento antinatural y, por lo tanto, es necesario ser muy cuidadoso cuando se hace un análisis de él. Las hábiles mejoras introducidas producen un efecto negativo cuando el arreglo está ordenado y resultados apenas satisfactorios cuando las claves están desordenadas. De todas maneras, se debe recordar que no se reduce el número de movimientos que es una operación más complicada y costosa que la operación de comparación. Por lo tanto, el tiempo de ejecución del algoritmo sigue siendo proporcional a n^2, $O(n^2)$.

8.2.6 Ordenación por selección directa

El método de ordenación por **selección directa** es más eficiente que los métodos analizados anteriormente. Pero, aunque su comportamiento es mejor que el de aquéllos y su

programación es fácil y comprensible, no se recomienda utilizarlo cuando el número de elementos del arreglo es mediano o grande. La idea básica de este algoritmo consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se lo coloca en la segunda posición. El proceso continúa hasta que todos los elementos del arreglo hayan sido ordenados. El método se basa en los siguientes principios:

1. Seleccionar el menor elemento del arreglo.
2. Intercambiar dicho elemento con el primero.
3. Repetir los pasos anteriores con los $(n - 1)$, $(n - 2)$ elementos, y así sucesivamente hasta que sólo quede el elemento mayor.

Ejemplo 8.8

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional *A* utilizando el método de selección directa.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

Se realiza la asignación: MENOR ← *A*[1] (15)

(MENOR < *A*[2])	(15 < 67)	sí se cumple la condición
(MENOR < *A*[3])	(15 < 08)	no se cumple la condición
		MENOR ← *A*[3] (8)
(MENOR < *A*[4])	(08 < 16)	sí se cumple la condición
(MENOR < *A*[5])	(08 < 44)	sí se cumple la condición
(MENOR < *A*[6])	(08 < 27)	sí se cumple la condición
(MENOR < *A*[7])	(08 < 12)	sí se cumple la condición
(MENOR < *A*[8])	(08 < 35)	sí se cumple la condición

Luego de la primera pasada, el arreglo queda de la siguiente forma:

A: 08 67 15 16 44 27 12 35

Observe que el menor elemento del arreglo *A*[3](08) se intercambió con el primer elemento *A*[1](15), realizando solamente un movimiento.

SEGUNDA PASADA

Se realiza la siguiente asignación: MENOR ← *A*[2](67)

(MENOR < *A*[3])	(67 < 15)	no se cumple la condición
		MENOR ← *A*[3] (15)
(MENOR < *A*[4])	(15 < 16)	sí se cumple la condición
(MENOR < *A*[5])	(15 < 44)	sí se cumple la condición

(MENOR < A[6]) (15 < 27) sí se cumple la condición
(MENOR < A[7]) (15 < 12) no se cumple la condición
MENOR ← A[7](12)
(MENOR < A[8]) (12 < 35) sí se cumple la condición

A: 08 12 15 16 44 27 67 35

Observe que el segundo elemento más pequeño del arreglo A[7](12) se intercambió con el segundo elemento, A[2](67).

En la tabla 8.4 se observa el resultado de las pasadas restantes.

TABLA 8.4

3a. pasada:	08	12	15	16	44	27	67	35
4a. pasada:	08	12	15	16	44	27	67	35
5a. pasada:	08	12	15	16	27	44	67	35
6a. pasada:	08	12	15	16	27	35	67	44
7a. pasada:	08	12	15	16	27	35	44	67

El algoritmo de ordenación por el método de selección directa es:

Algoritmo 8.7 Selección

Selección (A, N)

{Este algoritmo ordena los elementos de un arreglo unidimensional utilizando el método de selección directa. A es un arreglo unidimensional de N elementos}
{I, MENOR, K y J son variables de tipo entero}

1. *Repetir* con I desde 1 hasta N – 1
Hacer MENOR ← $A[I]$ y K ← I
1.1 *Repetir* con J desde I + 1 hasta N
1.1.1 *Si* ($A[J]$ < MENOR) *entonces*
Hacer MENOR ← $A[J]$ y K ← J
1.1.2 {Fin del condicional del paso 1.1.1}
1.2 {Fin del ciclo del paso 1.1}
Hacer $A[K]$ ← $A[I]$ y $A[I]$ ← MENOR
2. {Fin del ciclo del paso 1}

Análisis de eficiencia del método de selección directa

El análisis del método de selección directa es relativamente simple. Se debe considerar que el número de comparaciones entre elementos es independiente de la disposición inicial de éstos en el arreglo. En la primera pasada se realizan (n – 1) comparaciones, en la segunda pasada (n – 2) comparaciones y así sucesivamente hasta 2 y 1 comparaciones, en la penúltima y última pasadas, respectivamente. Por lo tanto:

$$C = (n-1)+(n-2)+\ldots+2+1 = \frac{n*(n-1)}{2}$$

que es igual a:

$$C = \frac{n^2 - n}{2} \qquad \textbf{Fórmula 8.10}$$

Respecto del número de intercambios, siempre será $n - 1$, a excepción de que se tenga incorporada en el algoritmo alguna técnica para prevenir el intercambio de un elemento consigo mismo. Por lo tanto:

$$M = n - 1 \qquad \textbf{Fórmula 8.11}$$

Así, por ejemplo, si se tiene que ordenar un arreglo que contiene 500 elementos, se efectuarán 124 750 comparaciones y 499 movimientos.

El tiempo de ejecución del algoritmo es proporcional a n^2, $O(n^2)$, aun cuando es más rápido que los métodos presentados con anterioridad.

8.2.7 Análisis de eficiencia de los métodos directos

La tabla 8.5 contiene las fórmulas necesarias para obtener el número de comparaciones y movimientos para ordenar un arreglo con los tres métodos directos analizados. Las columnas indican si los elementos del arreglo se encuentran en forma ordenada, desordenada o en orden inverso.

En la tabla 8.6, por otra parte, se observan los números de comparaciones y movimientos necesarios para ordenar un arreglo con los tres métodos directos analizados.

TABLA 8.5

		Ordenada	Desordenada	Orden inverso
Intercambio directo	C	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$
	M	0	$0.75*(n^2-n)$	$1.5*(n^2-n)$
Inserción directa	C	$(n-1)$	$\frac{(n^2+n-2)}{4}$	$\frac{n^2-n}{2}$
	M	0	$\frac{n^2-n}{4}$	$\frac{n^2-n}{2}$
Selección directa	C	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$
	M	$n-1$	$n-1$	$n-1$

TABLA 8.6

		Ordenada		Desordenada		Orden inverso	
Intercambio directo	*C*	124 750	499 500	124 750	499 500	124 750	499 500
	M	0	0	187 125	749 250	374 250	1 498 500
Inserción directa	*C*	499	999	62 624	250 249	124 750	499 500
	M	0	0	62 375	249 750	124 750	499 500
Selección directa	*C*	124 750	499 500	124 750	499 500	124 750	499 500
	M	499	999	999	999	499	999

Las columnas indican si los elementos del arreglo se encuentran en forma ordenada, desordenada o en orden inverso. Observe que estas columnas se encuentran divididas en dos. La subcolumna izquierda representa un arreglo de 500 elementos y la subcolumna derecha representa un arreglo de 1 000 elementos.

Es fácil observar que el método de selección directa es el mejor y sólo es superado por el método de inserción directa cuando los elementos del arreglo ya se encuentran ordenados. El peor método, sin duda, es el de intercambio directo.

En los siguientes incisos se analizarán los métodos logarítmicos más importantes.

8.2.8 Ordenación por el método de Shell

El **método de Shell** es una versión mejorada del método de inserción directa. Recibe ese nombre en honor de su autor, Donald L. Shell, quien lo propuso en 1959. Este método también se conoce como **inserción con incrementos decrecientes**.

En el método de ordenación por inserción directa cada elemento se compara para su ubicación correcta en el arreglo con los elementos que se encuentran en su parte izquierda. Si el elemento a insertar es más pequeño que el grupo de elementos que se encuentran a su izquierda, será necesario efectuar varias comparaciones antes de su ubicación.

Shell propone que las comparaciones entre elementos se efectúen con saltos de mayor tamaño, pero con incrementos decrecientes; así, los elementos quedarán ordenados en el arreglo más rápidamente. Para comprender mejor este algoritmo analice el siguiente caso.

Consideremos un arreglo que contenga 16 elementos. En primer lugar, se dividirán los elementos del arreglo en ocho grupos, teniendo en cuenta los elementos que se encuentran a ocho posiciones de distancia entre sí y se ordenarán por separado. Quedarán en el primer grupo los elementos ($A[1]$, $A[9]$); en el segundo, ($A[2]$, $A[10]$); en el tercero, ($A[3]$, $A[11]$), y así sucesivamente. Después de este primer paso se dividirán los elementos del arreglo en cuatro grupos, teniendo en cuenta ahora los elementos que se encuentren a cuatro posiciones de distancia entre sí y se les ordenará por separado. Que-

darán en el primer grupo los elementos ($A[1]$, $A[5]$, $A[9]$, $A[13]$); en el segundo ($A[2]$, $A[6]$, $A[10]$, $A[14]$), y así sucesivamente. En el tercer paso se dividirán los elementos del arreglo en grupos, tomando en cuenta los elementos que se encuentran ahora a dos posiciones de distancia entre sí y nuevamente se les ordenará por separado. En el primer grupo quedarán ($A[1]$, $A[3]$, $A[5]$, $A[7]$, $A[9]$, $A[11]$, $A[13]$, $A[15]$) y en el segundo ($A[2]$, $A[4]$, $A[6]$, $A[8]$, $A[10]$, $A[12]$, $A[14]$, $A[16]$).

Finalmente se agruparán y ordenarán los elementos de manera normal; es decir, de uno en uno. Se presenta a continuación un ejemplo.

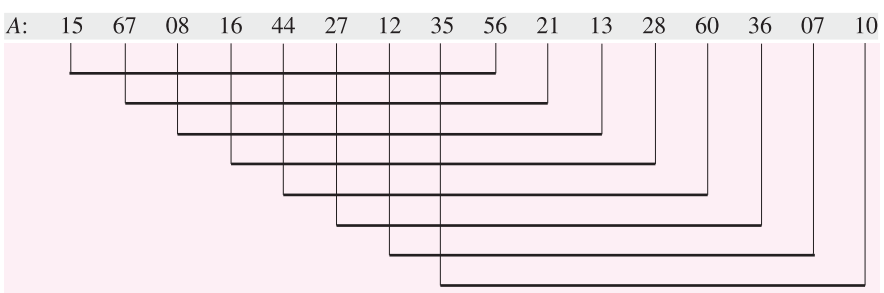
Ejemplo 8.9

Supongamos que se desea ordenar los elementos que se encuentran en el arreglo unidimensional A utilizando el método de Shell.

A:	15	67	08	16	44	27	12	35	56	21	13	28	60	36	07	10

PRIMERA PASADA

Se dividen los elementos en 8 grupos:

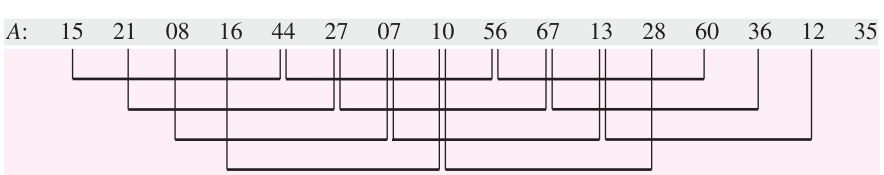


La ordenación produce:

A:	15	21	08	16	44	27	07	10	56	67	13	28	60	36	12	35

SEGUNDA PASADA

Se dividen los elementos en 4 grupos:

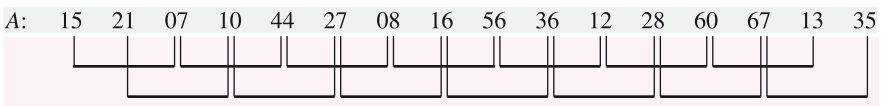


La ordenación produce:

A:	15	21	07	10	44	27	08	16	56	36	12	28	60	67	13	35

TERCERA PASADA

Se dividen los elementos en 2 grupos:

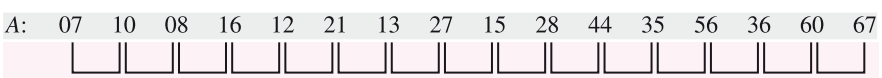


La ordenación produce

A: 07 10 08 16 12 21 13 27 15 28 44 35 56 36 60 67

CUARTA PASADA

Se dividen los elementos en un solo grupo:



La ordenación produce:

A: 07 08 10 12 13 15 16 21 27 28 35 36 44 56 60 67

A continuación se presenta el algoritmo de ordenación por el método de Shell.

Algoritmo 8.8 Shell

Shell (*A*, *N*)

{Este algoritmo permite ordenar los elementos de un arreglo unidimensional utilizando el método de Shell. *A* es un arreglo unidimensional de *N* elementos}
{INT, *I* y AUX son variables de tipo entero. BAND es una variable de tipo booleano}

1. Hacer INT ← *N* + 1
2. Mientras (INT > 1) *Repetir*
Hacer INT ← parte entera (INT / 2) y BAND ← VERDADERO
2.1 Mientras (BAND = VERDADERO) *Repetir*
Hacer BAND ← FALSO e *I* ← 1
2.1.1 Mientras ((*I* + INT) ≤ *N*) *Repetir*
2.1.1.1 *Si A*[*I*] > *A*[*I* + INT] *entonces*
Hacer AUX ← *A*[*I*], *A*[*I*] ← *A*[*I* + INT], *A*[*I* + INT] ← AUX
y BAND ← VERDADERO
2.1.1.2 {Fin del condicional del paso 2.1.1.1}
Hacer *I* ← *I* + 1
2.1.2 {Fin del ciclo del paso 2.1.1}
2.2 {Fin del ciclo del paso 2.1}
3. {Fin del ciclo del paso 2}

Análisis de eficiencia del método de Shell

El análisis de eficiencia del método de Shell es un problema muy complicado y aún no resuelto. Hasta el momento no se ha podido establecer la mejor secuencia de incrementos cuando n es grande. Cabe recordar que cada vez que se propone una secuencia de intervalos, es necesario *correr* el algoritmo para analizar su tiempo de ejecución.

En 1969, Pratt descubrió que el tiempo de ejecución del algoritmo es del orden de $n * (\log n)^2$. Unas pruebas exhaustivas realizadas para obtener la mejor secuencia de intervalos cuando el número de elementos del arreglo es igual a 8 arrojaron como resultado que la mejor secuencia corresponde a un intervalo de 1, que no es más que el método de inserción directa estudiado previamente. Estas pruebas también determinaron que el menor número de movimientos se registraba con la secuencia 3, 2, 1. Cabe aclarar que las pruebas exhaustivas corresponden al análisis de (8!) posibilidades; es decir, 40 320 casos diferentes.

En la tabla 8.7 se muestran las diez mejores secuencias obtenidas al evaluar las 40 320 posibilidades de secuencias que se presentan cuando se tiene un arreglo de 8 elementos.

Para concluir con el análisis de eficiencia de método de Shell, se menciona que estudios de Peterson y Russell, en la Universidad de Stanford, en 1971, muestran que las mejores secuencias para valores de N comprendidos entre 100 y 60 000 son las que se presentan en la tabla 8.8, donde $k = 0, 1, 2, 3,...$

Por último, y para aclarar aún más los conceptos vertidos sobre el método de Shell, se incluye un segundo ejemplo.

Ejemplo 8.10

Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A utilizando el método de Shell. La secuencia de intervalos que se utilizará corresponde a la fórmula $(2k - 1)$ presentada por Peterson y Russell.

A:	15	67	08	16	44	27	12	35	56	21	13	28	60	36	07	10

TABLA 8.7

Posición	Secuencia		
1		1	
2		6	1
3		5	1
4		7	1
5		4	1
6		3	1
7		2	1
8	5	3	1
9	4	2	1
10	3	2	1

TABLA 8.8

Secuencias	1, 3, 5, 9, ..., $2^k + 1$
	1, 3, 7, 15, ..., $2^k - 1$
	1, 3, 5, 11, ..., $(2^k \pm 1)/3$
	1, 4, 13, 40, ..., $(3^k + 1)/2$

Los resultados parciales de cada pasada, así como el resultado final, se observan en la tabla 8.9, donde PAS representa el número de pasada e INT representa el intervalo en el cual se está trabajando.

8.2.9 Ordenación por el método *quicksort*

El método de ordenación ***quicksort*** es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es también conocido como **método rápido** y de **ordenación por partición**. Este método es una mejora sustancial del método de intercambio directo y se denomina *quicksort* —rápido— por la velocidad con que ordena los elementos del arreglo. Su autor, C. A. Hoare, lo llamó así. La idea central de este algoritmo consiste en lo siguiente:

1. Se toma un elemento *X* de una posición cualquiera del arreglo.
2. Se trata de ubicar a *X* en la posición correcta del arreglo, de tal forma que todos los elementos que se encuentren a su izquierda sean menores o iguales a *X* y todos los que se encuentren a su derecha sean mayores o iguales a *X*.
3. Se repiten los pasos anteriores, pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición de *X* en el arreglo.
4. El proceso termina cuando todos los elementos se encuentran en su posición correcta en el arreglo.

Se debe seleccionar, entonces, un elemento *X* cualquiera. En este caso se seleccionará *A*[1]. Se empieza a recorrer el arreglo de derecha a izquierda comparando si los elementos son mayores o iguales a *X*. Si un elemento no cumple con esta condición, se intercambian aquéllos y se almacena en una variable la posición del elemento intercambiado —se acota el arreglo por la derecha—. Se inicia nuevamente el recorrido, pero ahora de izquierda a derecha, comparando si los elementos son menores o iguales a *X*.

TABLA 8.9

PAS	Arreglo *A*																INT
1	15	67	08	16	44	27	12	35	56	21	13	28	60	36	07	10	15
2	10	67	08	16	44	27	12	35	56	21	13	28	60	36	07	15	07
3	07	15	08	13	28	27	12	10	56	21	16	44	60	36	35	67	03
4	07	10	08	12	15	27	13	16	35	21	28	44	60	36	56	67	01
	07	08	10	12	13	15	16	21	27	28	35	36	44	56	60	67	

Si un elemento no cumple con esta condición, entonces se intercambian aquéllos y se almacena en otra variable la posición del elemento intercambiado —se acota el arreglo por la izquierda—. Se repiten los pasos anteriores hasta que el elemento X encuentra su posición correcta en el arreglo. Analicemos a continuación el siguiente ejemplo.

Ejemplo 8.11

Supongamos que se desea ordenar los elementos que se encuentran en el arreglo A utilizando el método.

A: 15 67 08 16 44 27 12 35

Se selecciona $A[1]$, por lo tanto, $X \leftarrow 15$.
Se llevan a cabo las comparaciones que se muestran a continuación:

PRIMERA PASADA

Recorrido de derecha a izquierda

$A[8] \geq X$	$(35 \geq 15)$	no hay intercambio
$A[7] \geq X$	$(12 \geq 15)$	sí hay intercambio

A: 12 67 08 16 44 27 15 35
↑ ↑

Recorrido de izquierda a derecha

$A[2] \leq X$	$(67 \leq 15)$	sí hay intercambio

A: 12 15 08 16 44 27 67 35
↑ ↑

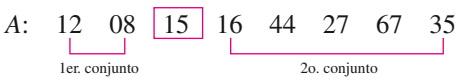
SEGUNDA PASADA

Recorrido de derecha a izquierda

$A[6] \geq X$	$(27 \geq 15)$	no hay intercambio
$A[5] \geq X$	$(44 \geq 15)$	no hay intercambio
$A[4] \geq X$	$(16 \geq 15)$	no hay intercambio
$A[3] \geq X$	$(08 \geq 15)$	sí hay intercambio

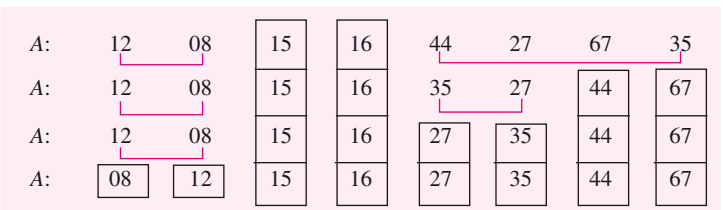
A: 12 08 15 16 44 27 67 35
↑ ↑

Como el recorrido de izquierda a derecha se debería iniciar en la misma posición donde se encuentra el elemento X, el proceso termina ya que se detecta que el elemento X se encuentra en la posición correcta. Observe que los elementos que forman parte del primer conjunto son menores o iguales a X, y los del segundo conjunto son mayores o iguales a X.



Este proceso de particionamiento aplicado para localizar la posición correcta de un elemento *X* en el arreglo se repite cada vez que queden conjuntos formados por dos o más elementos. El método se puede aplicar de manera iterativa o recursiva. En la tabla 8.10 se presenta la ubicación del resto de los elementos en el arreglo.

TABLA 8.10



El algoritmo de ordenación por el método *quicksort* en su versión recursiva es:

Algoritmo 8.9 Rápido_recursivo

Rápido_recursivo (*A*, *N*)

{Este algoritmo ordena los elementos del arreglo unidimensional utilizando el método rápido, de manera recursiva. *A* es un arreglo unidimensional de *N* elementos}

1. Llamar al algoritmo Reduce_recursivo con 1 y *N*

Observe que el algoritmo *Rápido_recursivo* requiere para su funcionamiento de otro algoritmo, el cual se presenta en la tabla 8.10.

Algoritmo 8.10 Reduce_recursivo

Reduce_recursivo (INI, FIN)

{INI y FIN representan las posiciones del extremo izquierdo y derecho, respectivamente, del conjunto de elementos a ordenar}
{IZQ, DER, POS y AUX son variables de tipo entero. BAND es una variable de tipo booleano}

1. Hacer IZQ ← INI, DER ← FIN, POS ← INI y BAND ← VERDADERO
2. Mientras (BAND = VERDADERO) *Repetir*
Hacer BAND ← FALSO
2.1 Mientras ((*A*[POS] ≤ *A*[DER]) y (POS ≠ DER)) *Repetir*
Hacer DER ← DER – 1
2.2 {Fin del ciclo del paso 2.1}
2.3 *Si* (POS ≠ DER) *entonces*
Hacer AUX ← *A*[POS], *A*[POS] ← *A*[DER], *A*[DER] ← AUX y POS ← DER

```
        2.3.1 Mientras ((A[POS] ≥ A[IZQ]) y (POS ≠ IZQ)) Repetir
                  Hacer IZQ ← IZQ + 1
        2.3.2 {Fin del ciclo del paso 2.3.1}
        2.3.3 Si (POS ≠ IZQ) entonces
                  Hacer BAND ← VERDADERO, AUX ← A[POS], A[POS] ← A[IZQ],
                    A[IZQ] ← AUX y POS ← IZQ
        2.3.4 {Fin del condicional del paso 2.3.3}
    2.4 {Fin del condicional del paso 2.3}
3.  {Fin del ciclo del paso 2}
4.  Si ((POS – 1) > INI) entonces
            Regresar a Reduce_recursivo con INI y (POS – 1) {Llamada recursiva}
5.  {Fin del condicional del paso 4}
6.  Si (FIN > (POS + 1)) entonces
            Regresar a Reduce_recursivo con (POS + 1) y FIN {Llamada recursiva}
7.  {Fin del condicional del paso 6}
```

Aun cuando el algoritmo del *quicksort* presentado resulte claro, es posible aumentar su velocidad de ejecución eliminando las llamadas recursivas. La recursividad es un instrumento muy poderoso, pero la eficiencia de ejecución es un factor muy importante en un proceso de ordenación que es necesario cuidar y administrar muy bien. Estas llamadas recursivas se pueden sustituir utilizando pilas, dando lugar entonces a la iteratividad.

Consideremos el arreglo unidimensional *A* del ejemplo 8.11. Luego de la primera partición, *A* queda de la siguiente manera:

A:	12	08	15	16	44	27	67	35
	1er. conjunto			2o. conjunto				

Al utilizar la iterativdad, se deben almacenar en las pilas los índices de los dos conjuntos de datos que falta tratar. Se utilizarán dos pilas, PILAMENOR y PILAMAYOR. En la primera se almacenará el extremo izquierdo y en la otra se almacenará el extremo derecho de los conjuntos de datos que falta tratar. En la figura 8.3 se observa el estado de las pilas, luego de cargar los extremos de los conjuntos que falta tratar.

Los índices del primer conjunto quedaron almacenados en la primera posición de PILAMENOR y PILAMAYOR, respectivamente. La posición del extremo izquierdo del primer conjunto (1) en PILAMENOR y la del extremo derecho del mismo conjunto (2) en PILAMAYOR. Las posiciones de los extremos izquierdo y derecho del segundo conjunto (4 y 8) fueron almacenados en la cima de PILAMENOR y PILAMAYOR, respectivamente.

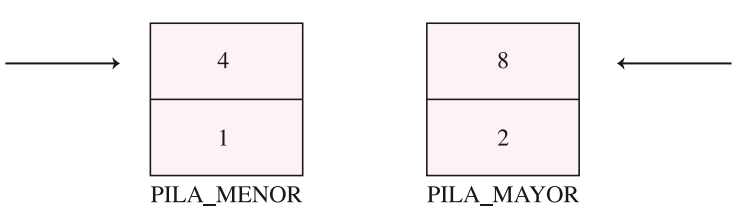


FIGURA 8.3
Pilas para sustituir la recursividad.

A continuación se presenta el algoritmo de ordenación por el método del *quicksort* utilizando iteratividad en lugar de recursividad.

Algoritmo 8.11 Rápido_iterativo

Rápido_iterativo (*A*, *N*)

{Este algoritmo ordena los elementos de un arreglo unidimensional utilizando el método rápido, de manera iterativa. *A* es un arreglo unidimensional de *N* elementos}
{TOPE, INI, FIN y POS son variables de tipo entero. PILAMENOR y PILAMAYOR son arreglos unidimensionales, que funcionan como pilas}

1. Hacer TOPE ← 1, PILAMENOR[TOPE] ← 1 y PILAMAYOR[TOPE] ← *N*
2. Mientras (TOPE > 0) *Repetir*
Hacer INI ← PILAMENOR[TOPE], FIN ← PILAMAYOR[TOPE] y TOPE ← TOPE – 1
Llamar al algoritmo Reduce_iterativo con INI, FIN y POS.
2.1 *Si* (INI < (POS – 1)) *entonces*
Hacer TOPE ← TOPE + 1, PILAMENOR[TOPE] ← INI y PILAMAYOR[TOPE] ← POS – 1
2.2 {Fin del condicional del paso 2.1}
2.3 *Si* (FIN > (POS + 1)) *entonces*
Hacer TOPE ← TOPE + 1, PILAMENOR[TOPE] ← POS + 1 y PILAMAYOR[TOPE] ← FIN
2.4 {Fin del condicional del paso 2.3}
3. {Fin del ciclo del paso 2}

Note que el algoritmo **Rápido_iterativo** necesita para su funcionamiento de otro algoritmo, el cual se presenta a continuación.

Algoritmo 8.12 Reduce_iterativo

Reduce_iterativo (INI, FIN, POS)

{INI y FIN representan las posiciones de los extremos izquierdo y derecho, respectivamente, del conjunto de elementos a evaluar. POS es una variable donde se almacenará el resultado de este algoritmo}
{IZQ, DER y AUX son variables de tipo entero. BAND es una variable de tipo booleano}

1. Hacer IZQ ← INI, DER ← FIN, POS ← INI y BAND ← VERDADERO
2. Mientras (BAND = VERDADERO) *Repetir*
2.1 Mientras ((*A*[POS] ≤ *A*[DER]) y (POS ≠ DER)) *Repetir*
Hacer DER ← DER – 1
2.2 {Fin del ciclo del paso 2.1}
2.3 *Si* (POS = DER)
entonces

```
            Hacer BAND ← FALSO
        si no
            Hacer AUX ← A[POS], A[POS] ← A[DER], A[DER] ← AUX y
              POS ← DER
    2.3.1 Mientras ((A[POS] ≥ A[IZQ]) y (POS ≠ IZQ)) Repetir
                    Hacer IZQ ← IZQ + 1
    2.3.2 {Fin del ciclo del paso 2.3.1}
    2.3.3 Si (POS = IZQ)
                entonces
                    Hacer BAND ← FALSO
                si no
                    Hacer AUX ← A[POS], A[POS] ← A[IZQ],
                    A[IZQ] ← AUX y POS ← IZQ
    2.3.4 {Fin del condicional del paso 2.3.3}
  2.4 {Fin del condicional del paso 2.3}
3. {Fin del ciclo del paso 2}
```

Ejemplo 8.12

En la tabla 8.11 se exponen los pasos necesarios para ordenar las claves del arreglo unidimensional *A*.

A: 15 67 08 16 44 27 12 35 56 21 13 28 60 36 07 10

La columna EXTREMO contiene los extremos *izquierdo* y *derecho* del conjunto de elementos a evaluar. En PILAMENOR se almacena el extremo izquierdo y en PILAMAYOR el extremo derecho de los conjuntos pendientes de tratar.

Una leve mejora en el funcionamiento del método rápido se puede producir si el primer elemento posicionado en el arreglo se encuentra en la mitad o muy próximo a su mitad. Para lograr esto último se necesita permutar los componentes del arreglo de tal forma que para el valor *X* todos los elementos que se encuentren a su izquierda desde *A*[1] hasta *A*[*i*], donde *i* es igual a ((*n*/2) – 1), sean menores o iguales a él y todos los que se encuentren a su derecha desde *A*[*i* + 1] hasta *A*[*N*] sean mayores o iguales a él. Por ejemplo, en el siguiente arreglo unidimensional *A*:

A: 45 21 76 08 17 96 55 36 43

43 es el elemento que ocupa la posición central del arreglo ordenado y divide a éste en dos mitades iguales. Queda la tarea de construir el algoritmo que encuentra al elemento *X* que ocupa la posición central del arreglo.

Análisis de eficiencia del método *quicksort*

El método *quicksort* es el más rápido de ordenación interna que existe en la actualidad. Esto es sorprendente, porque el método tiene su origen en el método de intercambio directo, el peor de todos los métodos directos. Diversos estudios realizados sobre su comportamiento demuestran que si se escoge en cada pasada el elemento que ocupa la

TABLA 8.11

POSICIONES																		
01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	EXTREMO	PILA MENOR	PILA MAYOR
15	67	08	16	44	27	12	35	56	21	13	28	60	36	07	10	(01, 16)	NIL	NIL
10	07	08	13	12	[15]	27	35	56	21	44	28	60	36	16	67	(07, 16)	01	05
																	07	08
10	07	08	13	12	[15]	16	21	[27]	56	44	28	60	36	35	67	(10, 16)	01	05
																	10	13
10	07	08	13	12	[15]	16	21	[27]	35	44	28	36	[56]	60	67	(15, 16)	07	08
																	01	05
10	07	08	13	12	[15]	16	21	[27]	35	44	28	36	[56]	[60]	[67]	(10, 13)	07	08
																	01	05
10	07	08	13	12	[15]	16	21	[27]	[28]	[35]	44	36	[56]	[60]	[67]	(12, 13)	07	08
																	01	05
10	07	08	13	12	[15]	16	21	[27]	[28]	[35]	[36]	[44]	[56]	[60]	[67]	(07, 08)	01	05
10	07	08	13	12	[15]	[16]	[21]	[27]	[28]	[35]	[36]	[44]	[56]	[60]	[67]	(01, 05)	NIL	NIL
08	07	[10]	13	12	[15]	[16]	[21]	[27]	[28]	[35]	[36]	[44]	[56]	[60]	[67]	(04, 05)	01	02
08	07	[10]	[12]	[13]	[15]	[16]	[21]	[27]	[28]	[35]	[36]	[44]	[56]	[60]	[67]	(01, 02)	NIL	NIL
[07]	[08]	[10]	[12]	[13]	[15]	[16]	[21]	[27]	[28]	[35]	[36]	[44]	[56]	[60]	[67]	NIL	NIL	NIL

posición central del conjunto de datos a analizar, el número de pasadas necesarias para ordenarlo es del orden de log n. Respecto del número de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada realizará $(n - 1)$ comparaciones, en la segunda $(n - 1)/2$ comparaciones, pero en dos conjuntos diferentes, en la tercera realizará $(n - 1)/4$ comparaciones, pero en cuatro conjuntos diferentes y así sucesivamente. Por lo tanto:

$$C = (n-1) + 2 * \frac{(n-1)}{2} + 4 * \frac{(n-1)}{4} + \ldots + (n-1) * \frac{(n-1)}{(n-1)}$$

lo cual es lo mismo que:

$$C = (n - 1) + (n - 1) + (n - 1) + ... + (n - 1)$$

Si se considera a cada uno de los componentes de la sumatoria como un término y el número de términos de la sumatoria es igual a m, entonces se tiene que:

$$C = (n - 1) * m$$

Considerando que el número de términos de la sumatoria (m) es igual al número de pasadas, y que éste es igual a log n, la expresión anterior queda:

$$C = (n - 1) * \log n \qquad \textbf{Fórmula 8.12}$$

Sin embargo, encontrar el elemento que ocupe la posición central del conjunto de datos que se van a analizar es una tarea difícil, ya que existen $1/n$ posibilidades de lograrlo. Además, el rendimiento medio del método es aproximadamente (2 * ln 2) inferior al caso óptimo, por lo que Hoare, el autor del método, propone como solución que el elemento X se seleccione arbitrariamente o bien entre una muestra relativamente pequeña de elementos del arreglo.

El peor caso ocurre cuando los elementos del arreglo ya se encuentran ordenados, o bien cuando se encuentran en orden inverso. Supongamos, por ejemplo, que se debe ordenar el siguiente arreglo unidimensional que ya se encuentra ordenado:

A: 08 12 15 16 27 35 44 67

Si se escoge arbitrariamente el primer elemento (08), entonces se particionará el arreglo en dos mitades, una de 0 y otra de $(n - 1)$ elementos.

Si se continúa con el proceso de ordenación y se escoge nuevamente el primer elemento (12) del conjunto de datos que se analizarán, entonces se dividirá el arreglo en dos nuevos conjuntos, nuevamente uno de 0 y otro de $(n - 2)$ elementos. Por lo tanto, el número de comparaciones que se realizarán será:

$$C_{máx} = n + (n-1) + (n-2) + \ldots + 2 = \frac{n*(n+1)}{2} - 1$$

que es igual a:

$$C_{máx} = \frac{n^2 + n}{2} - 1 \qquad \textbf{Fórmula 8.13}$$

Como conclusión, se puede afirmar que el tiempo promedio de ejecución del algoritmo es proporcional a (n * log n), $O(n * \log n)$. En el peor caso, el tiempo de ejecución es proporcional a n^2, $O(n^2)$.

8.2.10 Ordenación por el método *heapsort* (montículo)

El método de ordenación ***heapsort*** se conoce también como **montículo**. Su nombre se debe a su autor, J. W. Williams, quien lo llamó así. Es el más eficiente de los métodos de ordenación que trabajan con árboles. La idea central de este algoritmo se basa en dos operaciones:

1. Construir un montículo.
2. Eliminar la raíz del montículo en forma repetida.

Es importante señalar que un montículo se define como: *Para todo nodo del árbol se debe cumplir que su valor sea mayor o igual que el valor de cualquiera de sus hijos.*

Ejemplo 8.13

En la figura 8.4 se muestra un montículo. Allí se puede observar que para cada nodo K del árbol, su valor es mayor o igual que el valor de cualquiera de sus hijos.

Ahora bien, para representar un montículo en un arreglo lineal se debe tener en cuenta para todo nodo K lo siguiente:

1. El nodo K se almacena en la posición K correspondiente del arreglo.
2. El hijo izquierdo del nodo K se almacena en la posición 2 * K.
3. El hijo derecho del nodo K se almacena en la posición 2 * K + 1.

La figura 8.5 contiene la representación del montículo de la figura 8.4 en un arreglo unidimensional A.

Observe que si se desea obtener el hijo izquierdo del nodo $A[4]$, cuyo valor es 28, se hace $A[4 * 2] = A[8]$ y su contenido es 27. Si deseamos obtener en cambio el hijo derecho de $A[4]$, hacemos $A[4 * 2 + 1] = A[9]$ y su contenido es 16.

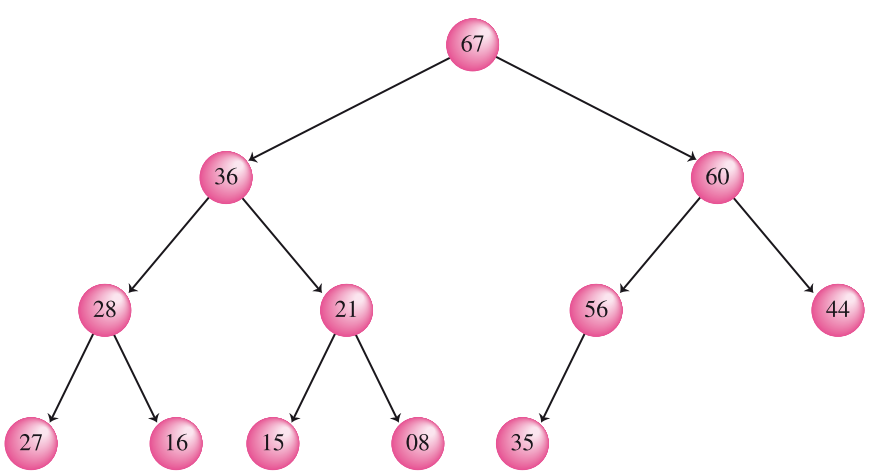


FIGURA 8.4
Montículo con 12 elementos.

FIGURA 8.5
Representación de un montículo en un arreglo lineal.

A	67	36	60	28	21	56	44	27	16	15	08	35
	1	2	3	4	5	6	7	8	9	10	11	12

A su vez, es posible calcular el padre de un nodo no raíz K cualquiera, tomando la parte entera de (K entre 2). Así, por ejemplo, si se desea obtener el padre del nodo $A[11]$, cuyo valor es 8, se hace A[parte entera(11 entre 2)] = $A[5]$ y su contenido es 21.

Inserción de un elemento en un montículo

La inserción de un elemento en un montículo se lleva a cabo por medio de los siguientes pasos:

1. Se inserta el elemento en la primera posición disponible.
2. Se verifica si su valor es mayor que el de su padre. Si se cumple esta condición, entonces se efectúa el intercambio. Si no se cumple esta condición, entonces el algoritmo se detiene y el elemento queda ubicado en su posición correcta en el montículo.

   Cabe aclarar que el paso 2 se aplica de manera recursiva y desde abajo hacia arriba.

Ejemplo 8.14

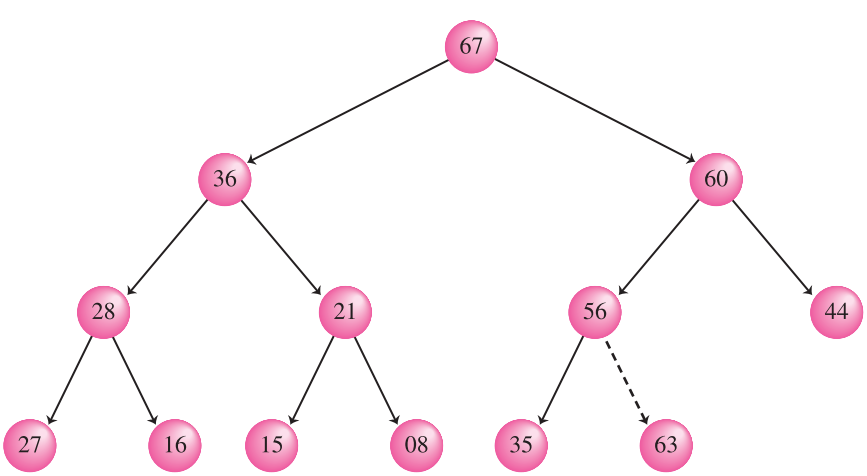
Supongamos que se quiere incorporar al montículo de la figura 8.6 el elemento 63. Las comparaciones que realizamos son:

63 > 56 sí hay intercambio
63 > 60 sí hay intercambio
63 > 67 no hay intercambio

Luego de haber insertado el elemento 63, el montículo queda como se muestra en la figura 8.7.

FIGURA 8.6
Montículo con 12 elementos.

Nota: Se utiliza flecha discontinua para indicar la posición inicial donde se inserta el elemento 63.



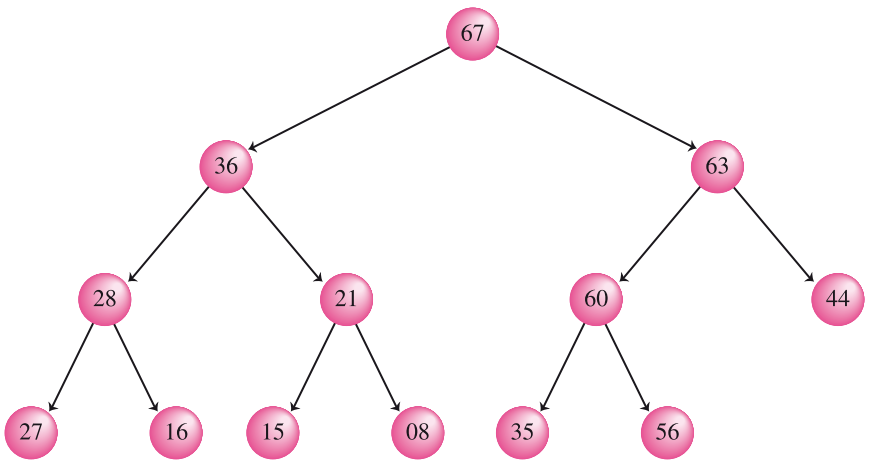


FIGURA 8.7
Montículo luego de haber insertado la clave 63.

Ejemplo 8.15

Supongamos que se desea insertar las siguientes claves en un montículo que se encuentra vacío.

15 60 08 16 44 27 12 35

Los resultados parciales que ilustran cómo funciona el procedimiento se observan en la figura 8.8. El montículo se representa como árbol y también como arreglo. Las flechas discontinuas indican la posición inicial donde se inserta el o los elementos.

Ejemplo 8.16

Dado el montículo de la figura 8.8*f*, verifique si éste queda igual al montículo, representado como arreglo, de la figura 8.9, luego de insertar las siguientes claves:

56 21 13 28 67 36 07 10

a) INSERCIÓN: CLAVE 15

b) INSERCIÓN: CLAVE 60

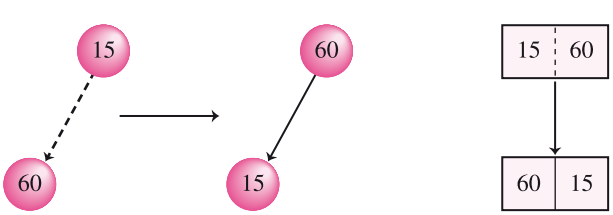
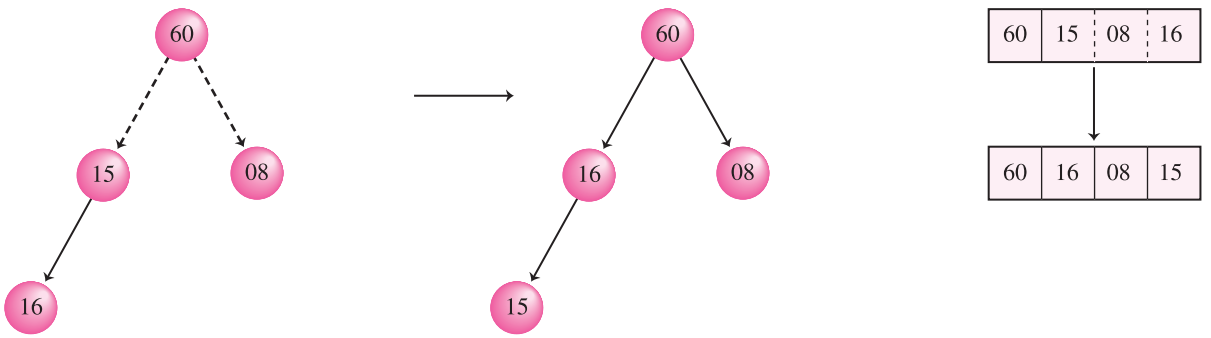


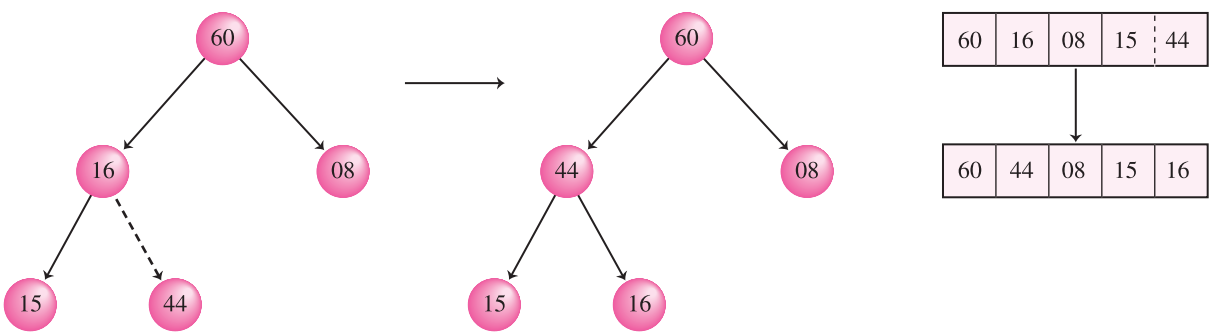
FIGURA 8.8
Inserciones en un montículo.

Nota: Se utiliza la flecha discontinua para indicar la posición inicial donde se inserta el o los elementos.

c) INSERCIÓN: CLAVES 08 y 16



d) INSERCIÓN: CLAVE 44



e) INSERCIÓN: CLAVE 27

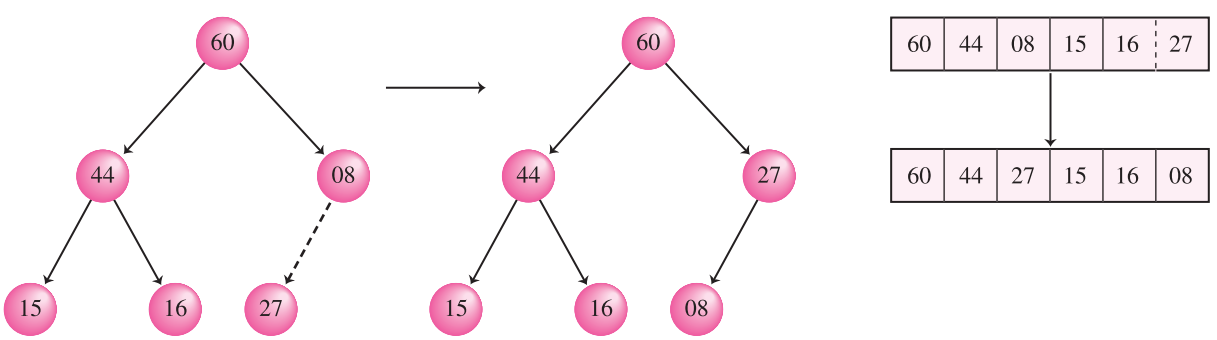


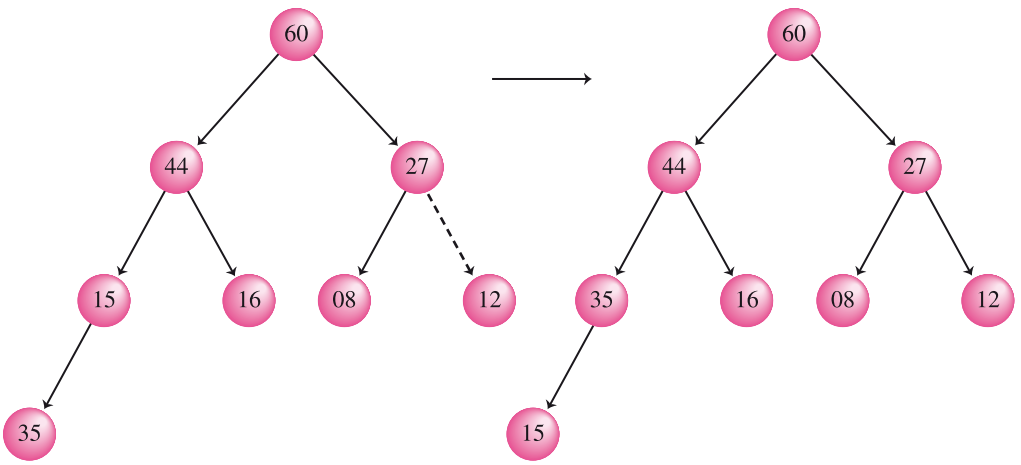
FIGURA 8.8

(*continuación*)

Inserciones en un montículo.

Nota: Se utiliza la flecha discontinua para indicar la posición inicial donde se inserta el o los elementos.

f) INSERCIÓN: CLAVES 12 y 35



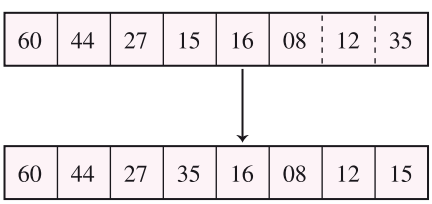


FIGURA 8.8
(*continuación*)
Inserciones en un montículo.

Nota: Se utiliza la flecha discontinua para indicar la posición inicial donde se inserta el o los elementos.

El algoritmo para insertar elementos en un montículo es:

Algoritmo 8.13 Inserta_montículo

Inserta_montículo (*A*, *N*)

{El algoritmo inserta los elementos en un montículo representado como arreglo. *A* es un arreglo unidimensional de *N* elementos}
{*I*, *K* y AUX son variables de tipo entero. BAND es una variable de tipo booleano}

1. *Repetir* con *I* desde 2 hasta *N*
Hacer *K* ← *I* y BAND ← VERDADERO
1.1 Mientras ((*K* > 1) y (BAND = VERDADERO)) *Repetir*
Hacer BAND ← FALSO
1.1.1 *Si* (A[*K*] > *A*[parte entera(*K* entre 2))] *entonces*
Hacer AUX ← *A*[parte entera(*K* entre 2)],
A[parte entera(*K* entre 2)] ← *A*[*K*], *A*[*K*] ← AUX,
K ← parte entera(*K* entre 2) y BAND ← VERDADERO
1.1.2 {Fin del condicional del paso 1.1.1}
1.2 {Fin del ciclo del paso 1.1}
2. {Fin del ciclo del paso 1}

67	56	60	44	21	28	36	15	35	16	13	08	27	12	17	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

FIGURA 8.9
Montículo representado como arreglo unidimensional.

Eliminación de un montículo

El proceso para obtener los elementos ordenados se efectúa eliminando la raíz del montículo en forma repetida. Ahora bien, los pasos necesarios para lograr la eliminación de la raíz del montículo son:

1. Se reemplaza la raíz con el elemento que ocupa la última posición del montículo.
2. Se verifica si el valor de la raíz es menor que el valor más grande de sus hijos. Si se cumple la condición, entonces se efectúa el intercambio. Si no se cumple la condición, entonces el algoritmo se detiene y el elemento queda ubicado en su posición correcta en el montículo.

   Cabe aclarar que el paso 2 se aplica de manera recursiva y desde arriba hacia abajo.

Ejemplo 8.17

Supongamos que se desea eliminar la raíz del montículo (67) de la figura 8.10.

Reemplazamos la raíz, 67, por el elemento que ocupa la última posición del montículo, 35. A continuación efectuamos las siguientes comparaciones:

35 < 60 sí hay intercambio
60 es el mayor de los hijos de 35

35 < 56 sí hay intercambio
56 es el mayor de los hijos de 35

El montículo, luego de haberse eliminado la raíz, queda como el que se presenta en la figura 8.11.

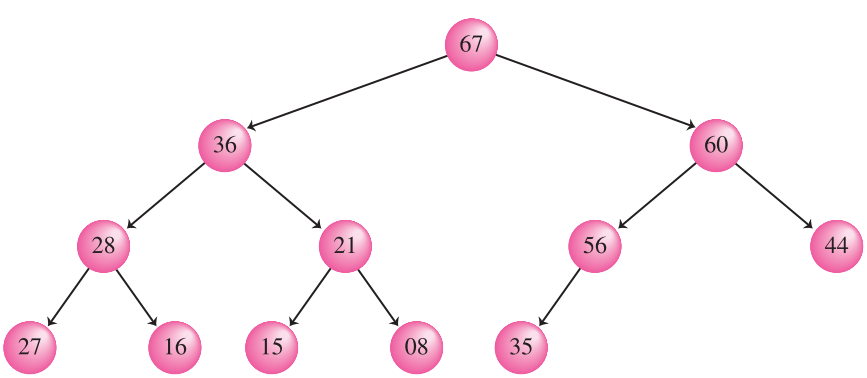
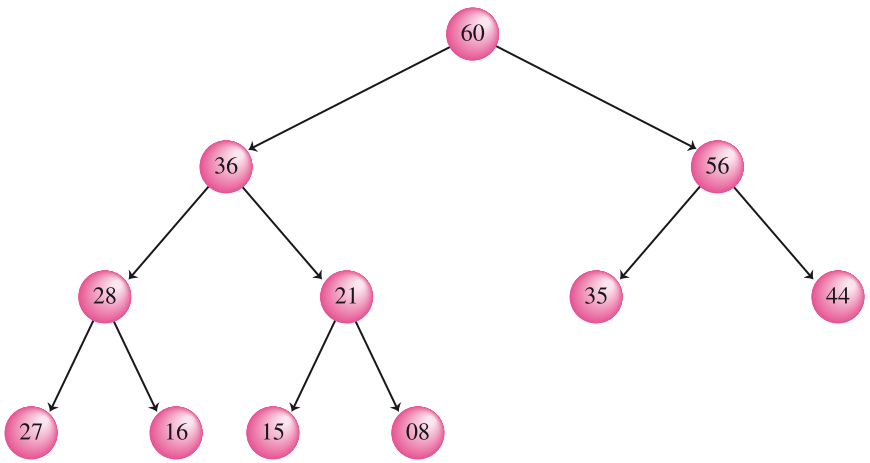


FIGURA 8.10
Montículo representado como árbol.

FIGURA 8.11
Montículo representado como árbol.



Ejemplo 8.18

Supongamos que se desea eliminar la raíz del montículo, presentado como arreglo, de la figura 8.12, en forma repetida.

Cabe aclarar que al reemplazar la raíz por el último elemento del montículo, ésta se coloca en la posición del último elemento. Es decir, la primera vez la raíz será colocada en la posición n, la segunda vez en la posición $(n - 1)$, la tercera vez en la posición $(n - 2)$ y así sucesivamente hasta que quede colocada en las posiciones 2 y 1, en forma respectiva. Los pasos a realizar son:

PRIMERA ELIMINACIÓN DE LA RAÍZ

Se intercambia la raíz, 67, con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:

$A[1] < A[3]$ $(10 < 60)$ sí hay intercambio
$A[3]$ es el mayor de los hijos de $A[1]$

$A[3] < A[7]$ $(10 < 36)$ sí hay intercambio
$A[7]$ es el mayor de los hijos de $A[3]$

$A[7] < A[14]$ $(10 < 12)$ sí hay intercambio
$A[14]$ es el mayor de los hijos de $A[7]$

Luego de eliminar la primera raíz, el montículo queda así:

60 56 36 44 21 28 12 15 35 16 13 08 27 10 07 67

FIGURA 8.12
Montículo representado como arreglo unidimensional.

67	56	60	44	21	28	36	15	35	16	13	08	27	12	07	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Observe que el elemento más grande se ubicó en la última posición del arreglo.

SEGUNDA ELIMINACIÓN DE LA RAÍZ

Se intercambia la raíz, 60, con el elemento que ocupa la última posición del montículo, 07. Las comparaciones que se realizan son:

$A[1] < A[2]$ (07 < 56) sí hay intercambio
$A[2]$ es el mayor de los hijos de $A[1]$

$A[2] < A[4]$ (07 < 44) sí hay intercambio
$A[4]$ es el mayor de los hijos de $A[2]$

$A[4] < A[9]$ (07 < 35) sí hay intercambio
$A[9]$ es el mayor de los hijos de $A[4]$

Luego de eliminar la segunda raíz, el montículo queda así:

56 44 36 35 21 28 12 15 07 16 13 08 27 10 [60] [67]

TERCERA ELIMINACIÓN DE LA RAÍZ

Se intercambia la raíz, 56, con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:

$A[1] < A[2]$ (10 < 44) sí hay intercambio
$A[2]$ es el mayor de los hijos de $A[1]$

$A[2] < A[4]$ (10 < 35) sí hay intercambio
$A[4]$ es el mayor de los hijos de $A[2]$

$A[4] < A[8]$ (10 < 15) sí hay intercambio
$A[8]$ es el mayor de los hijos de $A[4]$

Luego de eliminar la tercera raíz, el montículo queda así:

44 35 36 15 21 28 12 10 07 16 13 08 27 [56] [60] [67]

En la tabla 8.12 se presenta el resultado de las restantes eliminaciones. Observe que luego de eliminar la raíz del montículo, en forma repetida, el arreglo queda ordenado.

A continuación se presenta el algoritmo que elimina sucesivamente la raíz del montículo. Cabe aclarar que para efecto de aumentar la eficiencia del algoritmo se eliminan los intercambios parciales utilizando una variable auxiliar, en la que se almacena el último elemento del montículo.

TABLA 8.12

Eliminación	Montículo															
4	36	35	28	15	21	27	12	10	07	16	13	08	44	56	60	67
5	35	21	28	15	16	27	12	10	07	08	13	36	44	56	60	67
6	28	21	27	15	16	13	12	10	07	08	35	36	44	56	60	67
7	27	21	13	15	16	08	12	10	07	28	35	36	44	56	60	67
8	21	16	13	15	07	08	12	10	27	28	35	36	44	56	60	67
9	16	15	13	10	07	08	12	21	27	28	35	36	44	56	60	67
10	15	12	13	10	07	08	16	21	27	28	35	36	44	56	60	67
11	13	12	08	10	07	15	16	21	27	28	35	36	44	56	60	67
12	12	10	08	07	13	15	16	21	27	28	35	36	44	56	60	67
13	10	07	08	12	13	15	16	21	27	28	35	36	44	56	60	67
14	08	07	10	12	13	15	16	21	27	28	35	36	44	56	60	67
15	07	08	10	12	13	15	16	21	27	28	35	36	44	56	60	67

Algoritmo 8.14 Elimina_montículo

Elimina_montículo (*A*, *N*)

{El algoritmo elimina la raíz del montículo en forma repetida. *A* es un arreglo unidimensional de *N* elementos}
{*I*, AUX, IZQ, DER, *K* y AP son variables de tipo entero. BOOL es una variable de tipo booleano}

1. *Repetir* con *I* desde *N* hasta 2 {Ciclo descendente}
Hacer AUX ← *A*[*I*], *A*[*I*] ← *A*[1], IZQ ← 2, DER ← 3, *K* ← 1 y BOOL ← VERDADERO
1.1 Mientras ((IZQ < *I*) y (BOOL = VERDADERO)) *Repetir*
Hacer MAYOR ← *A*[IZQ] y AP ← IZQ
1.1.1 *Si* ((MAYOR < *A*[DER]) y (DER ≠ *I*)) *entonces*
Hacer MAYOR ← *A*[DER] y AP ← DER
1.1.2 {Fin del condicional del paso 1.1.1}
1.1.3 *Si* (AUX < MAYOR)
entonces
Hacer *A*[*K*] ← *A*[AP] y *K* ← AP
si no
Hacer BOOL ← FALSO
1.1.2 {Fin del condicional del paso 1.1.3}
Hacer IZQ ← *K* * 2 y DER ← IZQ + 1
1.2 {Fin del ciclo del paso 1.1}
Hacer *A*[*K*] ← AUX
2. {Fin del ciclo del paso 1}

El proceso de ordenación por el método del montículo consta de dos partes:

1. Construir el montículo. Esta operación se basa en la de inserción presentada en el algoritmo 8.13.
2. Eliminar repetidamente la raíz del montículo. Esta operación se basa en la de eliminación presentada en el algoritmo 8.14.

El algoritmo de ordenación, resulta entonces de la siguiente manera:

Algoritmo 8.15 Montículo

Montículo (*A*, *N*)

{El algoritmo ordena los elementos del arreglo utilizando el método del montículo. *A* es un arreglo unidimensional de *N* elementos}

1. Llamar al algoritmo Inserta_montículo con *A* y *N*.
2. Llamar al algoritmo Elimina_montículo con *A* y *N*.

Análisis de eficiencia del método del montículo

El análisis del método del montículo, como el de los métodos logarítmicos, es complejo. Es importante tener en cuenta tanto la fase de construcción del montículo como la fase donde se elimina repetidamente su raíz, para finalmente obtener el arreglo ordenado.

A diferencia de lo que se pudiera pensar, ya que en la fase de construcción del montículo los elementos mayores se cargan hacia la izquierda y en la fase de eliminación de la raíz los elementos mayores se cargan hacia la derecha, éste es un método muy rápido, sobre todo para valores grandes de *N*. Los estudios realizados al respecto demuestran que el tiempo de ejecución del algoritmo en ambas fases es de $O(n * \log n)$.

Aunque el método del montículo puede ser un poco más lento que el *quicksort* (se estima en 70%), es el único que garantiza que aun en el peor caso su tiempo de ejecución es proporcional a $(n * \log n)$, $O(n * \log n)$. Recuerde que el tiempo de ejecución del método *quicksort*, en el peor caso, es proporcional a n^2, $O(n^2)$.

8.3 ORDENACIÓN EXTERNA

En la actualidad es muy común procesar tales volúmenes de información que los datos no se pueden almacenar en la memoria principal de la computadora. Estos datos, organizados en archivos, se guardan en dispositivos de almacenamiento secundario, como cintas, discos, etcétera.

El proceso de ordenar los datos almacenados en varios archivos se conoce como **fusión** o **mezcla**; se entiende por este concepto a la combinación o intercalación de dos o más secuencias ordenadas en una única secuencia ordenada. Se debe hacer hincapié en

que sólo se colocan en la memoria principal de la computadora los datos que se pueden acceder en forma directa.

8.3.1 Intercalación de archivos

Por **intercalación de archivos** se entiende la unión o fusión de dos o más archivos, ordenados de acuerdo con un determinado campo clave, en un solo archivo. Analicemos el siguiente ejemplo.

Ejemplo 8.19

Suponga que se tienen dos archivos, *F*1 y *F*2, cuya información está ordenada de acuerdo con un campo clave:

F1: 06 09 18 20 35
F2: 10 16 25 28 66 82 87

Se debe producir, entonces, un archivo *F*3 ordenado, como resultado de la mezcla de *F*1 y *F*2. Sólo se pueden acceder en forma directa dos claves, la primera del archivo *F*1 y la segunda del archivo *F*2. Las comparaciones que se realizan para producir el archivo *F*3 son:

(06 < 10) sí se cumple la condición
Se escribe 06 en el archivo de salida *F*3 y se vuelve a leer otra clave de *F*1(09)

(09 < 10) sí se cumple la condición
Se escribe 09 en el archivo de salida *F*3 y se vuelve a leer otra clave de *F*1(18)

(18 < 10) no se cumple la condición
Se escribe 10 en el archivo de salida *F*3 y se vuelve a leer otra clave de *F*2(16)

El estado de los archivos *F*1, *F*2 y *F*3, hasta el momento, es como se muestra más abajo. La flecha señala el último elemento leído de los archivos que se están intercalando.

F1: 06 09 ↓18 20 35
F2: 10 ↓16 25 28 66 82 87

F3: 06 09 10

El proceso continúa hasta que en uno u otro archivo se detecte su final; en tal caso sólo se tendrá que copiar la información del archivo no vacío al archivo de salida *F*3. El resultado final de la intercalación entre los archivos *F*1 y *F*2 es:

***F*3:** 06 09 10 16 18 20 25 28 35 66 82 87

A continuación se muestra el algoritmo de intercalación de archivos.

Algoritmo 8.16 Intercalación

Intercalación (*F*1, *F*2, *F*3)

{El algoritmo intercala los elementos de dos archivos ya ordenados *F*1 y *F*2 y almacena el resultado en el archivo *F*3}
{R1 y R2 son variables de tipo entero. BAN1 y BAN2 son variables de tipo booleano}

1. Abrir los archivos *F*1 y *F*2 para lectura.
2. Abrir el archivo *F*3 para escritura.
3. Hacer BAN1 ← VERDADERO y BAN2 ← VERDADERO
4. Mientras (((no sea el fin de archivo de *F*1) o (BAN1 = FALSO)) y ((no sea el fin de archivo de *F*2) o (BAN2 = FALSO))) *Repetir*
  4.1 *Si* (BAN1 = VERDADERO) *entonces* {Se debe leer R1 de *F*1}
    Leer R1 de *F*1
    Hacer BAN1 ← FALSO
  4.2 {Fin del condicional del paso 4.1}
  4.3 *Si* (BAN2 = VERDADERO) *entonces* {Se debe leer R2 de *F*2}
    Leer R2 de *F*2
    Hacer BAN2 ← FALSO
  4.4 {Fin del condicional del paso 4.3}
  4.5 *Si* (R1 < R2)
    entonces
      Escribir R1 en *F*3
      Hacer BAN1 ← VERDADERO
    si no
      Escribir R2 en *F*3
      Hacer BAN2 ← VERDADERO
  4.6 {Fin del condicional del paso 4.5}
5. {Fin del ciclo del paso 4}
  {Verifica si se leyó un elemento de *F*1 y no se copió en *F*3}
6. *Si* (BAN1 = FALSO) *entonces*
  Escribir R1 en *F*3
  6.1 Mientras (no sea el fin de archivo de *F*1) *Repetir*
    Leer R1 de *F*1
    Escribir R1 en *F*3
  6.2 {Fin del ciclo del paso 6.1}
7. {Fin del condicional del paso 6}
  {Verifica si se leyó un elemento de *F*2 y no se copió en *F*3}
8. *Si* (BAN2 = FALSO) *entonces*
  Escribir R2 en *F*3
  8.1 Mientras (no sea el fin de archivo de *F*2) *Repetir*

```
            Leer R2 de F2
            Escribir R2 en F3
      8.2   {Fin del ciclo del paso 8.1}
   9.  {Fin del condicional del paso 8}
  10.  {Cerrar los archivos F1, F2 y F3}
```

8.3.2 Ordenación de archivos

La ordenación de archivos se efectúa cuando el volumen de los datos es demasiado grande y éstos no caben en la memoria principal de la computadora. Al ocurrir esta situación no se pueden aplicar los métodos de ordenación interna estudiados en la primera parte de este capítulo, de modo que se debe pensar en otro tipo de algoritmos para ordenar datos almacenados en archivos.

Por ordenación de archivos se entiende, entonces, la ordenación o clasificación de éstos, ascendente o descendentemente, de acuerdo con un campo determinado al que se denominará campo clave. La principal desventaja de esta ordenación es el tiempo de ejecución, debido a las sucesivas operaciones de lectura y escritura a/de archivo.

Los dos métodos de ordenación externa más importantes son los basados en la mezcla directa y en la mezcla equilibrada.

8.3.3 Ordenación por mezcla directa

El método de ordenación por **mezcla directa** es probablemente el más utilizado por su fácil comprensión. La idea central de este algoritmo consiste en la realización sucesiva de una partición y una fusión que produce secuencias ordenadas de longitud cada vez mayor. En la primera pasada, la partición es de longitud 1 y la fusión o mezcla produce secuencias ordenadas de longitud 2. En la segunda pasada, la partición es de longitud 2 y la fusión o mezcla produce secuencias ordenadas de longitud 4. Este proceso se repite hasta que la longitud de la secuencia para la partición sea:

$$\text{Parte entera } ((n+1)/2)$$

Donde n representa el número de elementos del archivo original.

Ejemplo 8.20

Supongamos que se desea ordenar las claves del archivo F. Para realizar tal actividad se utilizan dos archivos auxiliares a los que se les denominará $F1$ y $F2$.

***F*:** 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

PRIMERA PASADA

Partición en secuencias de longitud 1.

F1: 09′ 14′ 29′ 31′ 04′ 13′ 72′ 61′
F2: 75′ 68′ 17′ 25′ 05′ 18′ 46′

Fusión en secuencias de longitud 2.

F: 09 75′ 14 68′ 17 29′ 25 31′ 04 05′ 13 18′ 46 72′ 61′

SEGUNDA PASADA

Partición en secuencias de longitud 2.

F1: 09 75′ 17 29′ 04 05′ 46 72′
F2: 14 68′ 25 31′ 13 18′ 61′

Fusión en secuencias de longitud 4.

F: 09 14 68 75′ 17 25 29 31′ 04 05 13 18′ 46 61 72′

TERCERA PASADA

Partición en secuencias de longitud 4.

F1: 09 14 68 75′ 04 05 13 18′
F2: 17 25 29 31′ 46 61 72′

Fusión en secuencias de longitud 8.

F: 09 14 17 25 29 31 68 75′ 04 05 13 18 46 61 72′

CUARTA PASADA

Partición en secuencias de longitud 8.

F1: 09 14 17 25 29 31 68 75′
F2: 04 05 13 18 46 61 72′

Fusión en secuencias de longitud 16.

F: 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75

A continuación se presenta el algoritmo de ordenación de archivos por el método de mezcla directa.

Algoritmo 8.17 Mezcla_directa

Mezcla_directa (*F*, *F*1, *F*2, *N*)

{El algoritmo ordena los elementos del archivo *F* por el método de mezcla directa. Utiliza dos archivos auxiliares *F*1 y *F*2. *N* es el número de elementos del archivo *F*}
{PART es una variable de tipo entero}

1. Hacer PART ← 1
2. Mientras (PART < parte entera ((*N* + 1) / 2)) *Repetir*
    Llamar al algoritmo Particiona con *F*, *F*1, *F*2 y PART
    Llamar al algoritmo Fusiona con *F*, *F*1, *F*2 y PART
    Hacer PART ← PART * 2
3. {Fin del ciclo del paso 2}

Observe que el algoritmo requiere para su funcionamiento de dos algoritmos auxiliares, los cuales se presentan a continuación.

Algoritmo 8.18 Particiona

Particiona (*F*, *F*1, *F*2, PART)

{El algoritmo genera dos archivos auxiliares, *F*1 y *F*2, a partir del archivo *F*. PART es la longitud de la partición que se va a realizar}
{*K*, *L* y *R* son variables de tipo entero}

1. Abrir el archivo *F* para lectura.
2. Abrir los archivos *F*1 y *F*2 para escritura.
3. Mientras (no sea el fin de archivo de *F*) *Repetir*
    Hacer *K* ← 0
  3.1 Mientras ((*K* < PART) y (no sea el fin de archivo de *F*)) *Repetir*
      Leer *R* de *F*
      Escribir *R* en *F*1
      Hacer *K* ← *K* + 1
  3.2 {Fin del ciclo del paso 3.1}
      Hacer *L* ← 0
  3.3 Mientras ((*L* < PART) y (no sea el fin de archivo de *F*)) *Repetir*
      Leer *R* de *F*
      Escribir *R* en *F*2
      Hacer *L* ← *L* + 1
  3.4 {Fin del ciclo del paso 3.3}
4. {Fin del ciclo del paso 3}

Algoritmo 8.19 Fusiona

Fusiona (*F*, *F*1, *F*2, PART)

{El algoritmo fusiona los archivos *F*1 y *F*2 en el archivo *F*. PART es la longitud de la partición que se realizó previamente}
{*R*1, *R*2, *K* y *L* son variables de tipo entero. *B*1 y *B*2 son variables de tipo booleano}

1. Abrir el archivo *F* para escritura.
2. Abrir los archivos *F*1 y *F*2 para lectura.
3. Hacer *B*1 ← VERDADERO y *B*2 ← VERDADERO
4. *Si* (no es el fin de archivo de *F*1) *entonces*
    Leer *R*1 de *F*1
    Hacer *B*1 ← FALSO
5. {Fin del condicional del paso 4}
6. *Si* (no es el fin de archivo de *F*2) *entonces*
    Leer *R*2 de *F*2
    Hacer *B*2 ← FALSO
7. {Fin del condicional del paso 6}
8. Mientras ((no sea el fin de archivo de *F*1) o (*B*1 = FALSO)) y ((no sea el fin de archivo de *F*2) o (*B*2 = FALSO)) *Repetir*
    Hacer *K* ← 0 y *L* ← 0
8.1 Mientras ((*K* < PART) y (*B*1 = FALSO)) y ((*L* < PART) y ((*B*2 = FALSO))
    Repetir
8.1.1 *Si* (*R*1 ≤ *R*2)
    entonces
        Escribir *R*1 en *F*
        Hacer *B*1 ← VERDADERO y *K* ← *K* + 1
8.1.1.1 *Si* (no es el fin de archivo de *F*1) *entonces*
            Leer *R*1 de *F*1
            Hacer *B*1 ← FALSO
8.1.1.2 {Fin del condicional del paso 8.1.1.1}
    si no
        Escribir *R*2 en *F*
        Hacer *B*2 ← VERDADERO y *L* ← *L* + 1
8.1.1.3 *Si* (no es el fin de archivo de *F*2) *entonces*
            Leer *R*2 de *F*2
            Hacer *B*2 ← FALSO
8.1.1.4 {Fin del condicional del paso 8.1.1.3}
8.1.2 {Fin del condicional del paso 8.1.1}
8.2 {Fin del ciclo del paso 8.1}
8.3 Mientras ((*K* < PART) y (*B*1 = FALSO)) *Repetir*
        Escribir *R*1 en *F*
        Hacer *B*1 ← VERDADERO y *K* ← *K* + 1
8.3.1 *Si* (no es el fin de archivo de *F*1) *entonces*
            Leer *R*1 de *F*1
            Hacer *B*1 ← FALSO
8.3.2 {Fin del condicional del paso 8.3.1}
8.4 {Fin del condicional del paso 8.3}

```
8.5  Mientras ((L < PART) y (B2 = FALSO)) Repetir
          Escribir R2 en F
          Hacer B2 ← VERDADERO y L ← L + 1
     8.5.1  Si (no es el fin de archivo de F2) entonces
               Leer R2 de F2
               Hacer B2 ← FALSO
     8.5.2  {Fin del condicional del paso 8.5.1}
8.6  {Fin del ciclo del paso 8.5}
9.   {Fin del ciclo del paso 8}
10.  Si (B1 = FALSO) entonces
          Escribir R1 en F
11.  {Fin del condicional del paso 10}
12.  Si (B2 = FALSO) entonces
          Escribir R2 en F
13.  {Fin del condicional del paso 12}
14.  Mientras (no sea el fin de archivo de F1) Repetir
          Leer R1 de F1
          Escribir R1 en F
15.  {Fin del condicional del paso 14}
16.  Mientras (no sea el fin de archivo de F2) Repetir
          Leer R2 de F2
          Escribir R2 en F
17.  {Fin del ciclo del paso 16}
18.  {Cerrar los archivos F, F1 y F2}
```

Ejemplo 8.21

Supongamos que se desea ordenar las claves del archivo *F* utilizando el método de mezcla directo.

F: 25 33 15 18 21 07 12 36 84 90 19 38 40 22
64 77 29 36 11

Los pasos que se realizan son:

PRIMERA PASADA

Partición en secuencias de longitud 1.

F1: 25′ 15′ 21′ 12′ 84′ 19′ 40′ 64′ 29′ 11′
F2: 33′ 18′ 07′ 36′ 90′ 38′ 22′ 77′ 36′

Fusión en secuencias de longitud 2.

F: 25 33′ 15 18′ 07 21′ 12 36′ 84 90′ 19 38′
22 40′ 64 77′ 29 36′ 11′

SEGUNDA PASADA

Partición en secuencias de longitud 2.

F1: 25 33′ 07 21′ 84 90′ 22 40′ 29 36′
F2: 15 18′ 12 36′ 19 38′ 64 77′ 11′

Fusión en secuencias de longitud 4.

F: 15 18 25 33′ 07 12 21 36′ 19 38 84 90′
22 40 64 77′ 11 29 36′

TERCERA PASADA

Partición en secuencias de longitud 4.

F1: 15 18 25 33′ 19 38 84 90′ 11 29 36′
F2: 07 12 21 36′ 22 40 64 77′

Fusión en secuencias de longitud 8.

F: 07 12 15 18 21 25 33 36′ 19 22 38 40 64′
77 84 90′ 11 29 36′

CUARTA PASADA

Partición en secuencias de longitud 8.

F1: 07 12 15 18 21 25 33 36′ 11 29 36′
F2: 19 22 38 40 64 77 84 90′

Fusión en secuencias de longitud 16.

F: 07 12 15 18 19 21 22 25 33 36 38 40 64
77 84 90′ 11 29 36′

QUINTA PASADA

Partición en secuencias de longitud 16.

F1: 07 12 15 18 19 21 22 25 33 36 38 40 64 77 84 90′
F2: 11 29 36′

Fusión en secuencias de longitud 32.

F: 07 11 12 15 18 19 21 22 25 29 33 36 36 38 40
64 77 84 90

8.3.4 Ordenación por el método de mezcla equilibrada

El método de ordenación por **mezcla equilibrada**, conocido también cómo **mezcla natural**, es una optimización del método de mezcla directa.

La idea central de este algoritmo consiste en realizar las particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias de tamaño fijo previamente determinadas. Luego se realiza la fusión de las secuencias ordenadas, en forma alternada, sobre dos archivos. Aplicando estas acciones en forma repetida se logrará que el archivo original quede ordenado. Para la realización de este proceso de ordenación se necesitarán cuatro archivos. El archivo original *F* y tres archivos auxiliares a los que se denominará *F*1, *F*2 y *F*3. De estos archivos, dos serán considerados de entrada y dos de salida; esto, de manera alternada, con el objeto de realizar la fusión-partición. El proceso termina cuando en la realización de una fusión-partición el segundo archivo quede vacío.

Ejemplo 8.22

Supongamos que se desea ordenar las claves del archivo *F* utilizando el método de mezcla equilibrada.

***F*:** 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

Los pasos que se realizan son:

PARTICIÓN INICIAL

***F2*:** 09 75′ 29′ 25′ 46 61′
***F3*:** 14 68′ 17 31′ 04 05 13 18 72′

PRIMERA FUSIÓN-PARTICIÓN

***F*:** 09 14 68 75′ 04 05 13 18 25 46 61 72′
***F1*:** 17 29 31′

SEGUNDA FUSIÓN-PARTICIÓN

***F2*:** 09 14 17 29 31 68 75′
***F3*:** 04 05 13 18 25 46 61 72′

TERCERA FUSIÓN-PARTICIÓN

***F*:** 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75
***F1*:**

Observe que al realizar la tercera fusión-partición el segundo archivo queda vacío; por lo tanto, se puede afirmar que el archivo ya se encuentra ordenado. A continuación se presenta la descripción formal del algoritmo de mezcla equilibrada.

Algoritmo 8.20 Mezcla_equilibrada

Mezcla_equilibrada (*F*, *F*1, *F*2, *F*3)

{El algoritmo ordena los elementos del archivo *F* por el método de mezcla equilibrada. Utiliza tres archivos auxiliares *F*1, *F*2 y *F*3}
{BAND es una variable de tipo booleano}

1. Llamar al algoritmo Partición_inicial con *F*, *F*2 y *F*3
2. Llamar al algoritmo Partición_fusión con *F*2, *F*3, *F* y *F*1
3. Hacer BAND ← FALSO
4. Mientras ((F1 ≠ VACÍO) o (F3 ≠ VACÍO)) *Repetir*
4.1 *Si* (BAND = VERDADERO)
entonces
Llamar al algoritmo Partición_fusión con *F*2, *F*3, *F* y *F*1
Hacer BAND ← FALSO
si no
Llamar al algoritmo Partición_fusión con *F*, *F*1, *F*2 y *F*3
Hacer BAND ← VERDADERO
4.2 {Fin del condicional del paso 4.1}
5. {Fin del ciclo del paso 4}

El algoritmo 8.20 requiere para su funcionamiento de dos algoritmos auxiliares, **Partición_inicial** y **Partición_fusión**, los cuales se presentan a continuación.

Algoritmo 8.21 Partición_inicial

Partición_inicial (*F*, *F*2, *F*3)

{El algoritmo produce la partición inicial del archivo *F* en dos archivos auxiliares, *F*2 y *F*3}
{AUX y *R* son variables de tipo entero. BAND es una variable de tipo booleano}

1. Abrir el archivo *F* para lectura.
2. Abrir los archivos *F*2 y *F*3 para escritura.
3. Leer *R* de *F*.
4. Escribir *R* en *F*2.
5. Hacer BAND ← VERDADERO y AUX ← *R*
6. Mientras (no sea el fin de archivo de *F*) *Repetir*
Leer *R* de *F*
6.1 *Si* (R ≥ AUX)
entonces
Hacer AUX ← *R*
6.1.1 *Si* (BAND = VERDADERO)
entonces
Escribir *R* en *F*2
si no

```
                    Escribir R en F3
        6.1.2  {Fin del condicional del paso 6.1.1}
          si no
              Hacer AUX ← R
        6.1.3  Si (BAND = VERDADERO)
                  entonces
                    Escribir R en F3
                    Hacer BAND ← FALSO
                  si no
                    Escribir R en F2
                    Hacer BAND ← VERDADERO
        6.1.4  {Fin del condicional del paso 6.1.3}
    6.2  {Fin del condicional del paso 6.1}
7.  {Fin del ciclo del paso 6}
8.  {Cerrar los archivos F, F2 y F3}
```

Algoritmo 8.22 Partición_fusión

Partición_fusión (*FA, FB, FC, FD*)

{El algoritmo produce la partición y la fusión de los archivos *FA* y *FB*, en los archivos *FC* y *FD*}
{*R*1, *R*2 y AUX son variables de tipo entero. BAN1, BAN2 y BAN3 son variables de tipo booleano}

```
1.  Abrir los archivos FA y FB para lectura.
2.  Abrir los archivos FC y FD para escritura.
3.  Hacer BAN1 ← VERDADERO, BAN2 ← VERDADERO, BAN3 ← VERDADERO y
    AUX ← –32 768  {AUX se inicializa con un valor negativo alto}
4.  Mientras ((no sea el fin de archivo de FA) o (BAN1 = FALSO)) y ((no
    sea el fin de archivo de FB) o ((BAN2 = FALSO)) Repetir
    4.1  Si (BAN1 = VERDADERO) entonces
                Leer R1 de FA
                Hacer BAN1 ← FALSO
    4.2  {Fin del condicional del paso 4.1}
    4.3  Si (BAN2 = VERDADERO) entonces
                Leer R2 de FB
                Hacer BAN2 ← FALSO
    4.4  {Fin del condicional del paso 4.3}
    4.5  Si (R1 < R2)
            entonces
        4.5.1  Si (R1 ≥ AUX)
                  entonces
            4.5.1.1  Si (BAN3 = VERDADERO)
                        entonces
                            Escribir R1 en FC
                        si no
```

```
                    Escribir R1 en FD
          4.5.1.2  {Fin del condicional del paso 4.5.1.1}
                    Hacer BAN1 ← VERDADERO y AUX ← R1
               si no
          4.5.1.3  Si (BAN3 = VERDADERO)
                    entonces
                    Escribir R2 en FC
                    Hacer BAN3 ← FALSO
                    si no
                    Escribir R2 en FD
   Hacer BAN3 ← VERDADERO
          4.5.1.4  {Fin del condicional del paso 4.5.1.3}
      Hacer BAN2 ← VERDADERO y AUX ← –32 768
      4.5.2  {Fin del condicional del paso 4.5.1}
         si no
      4.5.3  Si (R2 ≥ AUX)
               entonces
          4.5.3.1  Si (BAN3 = VERDADERO)
                    entonces
                    Escribir R2 en FC
                      si no
                    Escribir R2 en FD
          4.5.3.2  {Fin del condicional del paso 4.5.3.1}
                    Hacer BAN2 ← VERDADERO y AUX ← R2
               si no
          4.5.3.3  Si (BAN3 = VERDADERO)
                    entonces
                    Escribir R1 en FC
                    Hacer BAN3 ← FALSO
                    si no
                    Escribir R1 en FD
                    Hacer BAN3 ← VERDADERO
          4.5.3.4  {Fin del condicional del paso 4.5.3.3}
                    Hacer BAN1 ← VERDADERO y AUX ← –32 768
      4.5.4  {Fin del condicional del paso 4.5.3}
   4.6  {Fin del condicional del paso 4.5}
5. {Fin del ciclo del paso 4}
6. Si (BAN1 = FALSO) entonces
   6.1  Si (BAN3 = VERDADERO)
          entonces
            Escribir R1 en FC
      6.1.1  Mientras (no sea el fin de archivo de FA) Repetir
                    Leer R1 de FA
                    Escribir R1 en FC
      6.1.2  {Fin del ciclo del paso 6.1.1}
         si no
            Escribir R1 en FD
      6.1.3  Mientras (no sea el fin de archivo de FA) Repetir
                    Leer R1 de FA
                    Escribir R1 en FD
```

```
      6.1.4 {Fin del ciclo del paso 6.1.3}
   6.2 {Fin del condicional del paso 6.1}
7. {Fin del condicional del paso 6}
8. Si (BAN2 = FALSO) entonces
   8.1 Si (BAN3 = VERDADERO)
         entonces
            Escribir R2 en FC
      8.1.1 Mientras (no sea el fin de archivo de FB) Repetir
               Leer R2 de FB
               Escribir R2 en FC
      8.1.2 {Fin del ciclo del paso 8.1.1}
         si no
            Escribir R2 en FD
      8.1.3 Mientras (no sea el fin de archivo de FB) Repetir
               Leer R2 de FB
               Escribir R2 en FD
      8.1.4 {Fin del ciclo del paso 8.1.3}
   8.2 {Fin del condicional del paso 8.1}
9. {Fin del condicional del paso 8}
10. {Cerrar los archivos FA, FB, FC y FD}
```

Ejemplo 8.23

Supongamos que se desea ordenar las claves del archivo *F* utilizando el método de mezcla equilibrada.

***F*:** 25 33 15 18 21 07 12 36 84 90 19 38 40 22 64
77 29 36 11

Los pasos que se realizan son:

PARTICIÓN INICIAL

***F2*:** 25 33′ 07 12 36 84 90′ 22 64 77′ 11′
***F3*:** 15 18 21′ 19 38 40′ 29 36′

PRIMERA FUSIÓN-PARTICIÓN

***F*:** 15 18 21 25 33′ 22 29 36 64 77′
***F1*:** 07 12 19 36 38 40 84 90′ 11′

SEGUNDA FUSIÓN-PARTICIÓN

***F2*:** 07 12 15 18 19 21 25 33 36 38 40 84 90′
***F3*:** 11 22 29 36 64 77′

TERCERA FUSIÓN-PARTICIÓN

***F*:** 07 11 12 15 18 19 21 22 25 29 33 36 36 38 40 64
77 84 90
***F*1:**

EJERCICIOS

Ordenación interna

1. En un arreglo se guardan los apellidos de *N* alumnos. Aplique el método de la burbuja —ordenación por el método de intercambio directo— para ordenar el arreglo en forma ascendente, de manera que:

$$Ap_1 \leq Ap_2 \leq \ldots \leq Ap_n$$

2. Resuelva el problema 1 aplicando el método de la burbuja con señal.

3. Resuelva el problema 1 aplicando el método *shakersort*.

4. Compare el tiempo de ejecución de las soluciones 1, 2 y 3 para distintos valores de *N*.

5. Dado un arreglo unidimensional de *N* números enteros, ordénelo en forma descendente aplicando:

a) Inserción directa
b) Inserción binaria
c) Selección directa

Compare el tiempo de ejecución de las soluciones *a*, *b* y *c* para distintos valores de *N*.

6. Se tienen tres arreglos paralelos. El primero de ellos almacena las matrículas de *N* alumnos; el segundo, las calificaciones de los *N* alumnos obtenidas en un examen final, y el tercero, el número total de materias aprobadas por cada alumno. Los elementos de los arreglos se corresponden.

a) Aplique el método de inserción directa para ordenar los arreglos, de manera que queden ordenados en forma ascendente por matrícula.
b) Aplique el método *quicksort* para ordenar los arreglos, de manera que queden ordenados en forma descendente por el número total de materias aprobadas.

7. En cierta empresa se maneja una lista de precios de los *N* artículos que se venden. De cada artículo se tiene la siguiente información:

- Clave del artículo
- Nombre del artículo
- Precio del artículo

a) Ordene el arreglo en forma ascendente según el campo "clave". Aplique el método *quicksort*.

b) Ordene el arreglo en forma descendente según el campo "precio". Aplique el método del *montículo*.

8. Resuelva el inciso *a*) del problema 7 aplicando el método de selección directa. Compare el tiempo de ejecución de esta solución con la obtenida en el ejercicio anterior, para distintos valores de *N*.

9. Una compañía propietaria de una cadena de hoteles quiere que se ordene su información. De cada hotel se tienen los siguientes datos:

- Nombre del hotel
- Ciudad en la que se encuentra el hotel
- Número de estrellas
- Número de cuartos

En una misma ciudad puede haber varios hoteles de la compañía.

Escriba un programa que ordene el arreglo según el campo "ciudad", en primer término, y luego, según el campo, "nombre". Es decir, el arreglo debe quedar:

$Nombre_1$	$Ciudad_1$	$Núm.estrellas_1$	$Núm.cuartos_1$
$Nombre_2$	$Ciudad_1$	$Núm.estrellas_2$	$Núm.cuartos_2$
...			
$Nombre_i$	$Ciudad_2$	$Núm.estrellas_i$	$Núm.cuartos_i$
...			
$Nombre_n$	$Ciudad_k$	$Núm.estrellas_k$	$Núm.cuartos_n$

Donde:

- $Ciudad_1 < Ciudad_2 < ... < Ciudad_k$
- $Nombre_1 < Nombre_2 < ... < Nombre_{i-1}$
- $Nombre_i < Nombre_{i+1} < ... < Nombre_{j-1}$
  ...
- $Nombre_t < Nombre_{t+1} < ... < Nombre_n$

10. Retome el problema 8 del capítulo 1. Ordene los arreglos SUR, CENTRO y NORTE, aplicando:

a) Inserción directa
b) Inserción binaria
c) Shell

11. Una escuela tiene almacenados los principales datos de cada alumno. Éstos son:

- Nombre del alumno
- Matrícula
- Número de materias aprobadas
- Promedio

a) Aplique el método de selección directa para ordenar el arreglo de *N* alumnos en forma ascendente, según el campo "nombre".
b) Aplique el método *quicksort* para ordenar el arreglo de *N* alumnos en forma ascendente, según el campo "número de materias aprobadas".

12. Se tiene un arreglo de *N* números enteros.

a) ¿Cuántas comparaciones y cuántos intercambios se deben realizar si se ordena el arreglo con el método de la burbuja?
b) ¿Cuántas comparaciones y cuántos intercambios se deben realizar si se ordena el arreglo con el método de selección directa?

13. Dado un arreglo unidimensional de *N* números enteros que debe ser ordenado en forma ascendente, conteste las siguientes preguntas:

a) Cuántas comparaciones e intercambios se deben realizar, aplicando el método de inserción directa, si:

- El arreglo ya está ordenado.
- El arreglo está ordenado en forma descendente.

b) Conteste la pregunta del inciso anterior, para el método *quicksort*.
c) Conteste la pregunta del inciso anterior, para el método del montículo.

14. Retome el problema 7, considerando que se definió la clase *Artículo* y se tiene un arreglo de objetos de este tipo. Escriba una función que ordene el arreglo de objetos, utilizando el algoritmo de *quicksort*.

15. Defina una clase *Arreglo*, según lo visto en el capítulo 1. En la clase debe incluir por lo menos 2 métodos de los estudiados en este capítulo para ordenar los elementos del arreglo.

16. Escriba una función que anime el proceso de ordenación usando el algoritmo de *quicksort*. Es decir, en la medida en que se va ordenando el arreglo, en la pantalla se debe ir reflejando gráficamente su cambio de estado.

Ordenación externa

17. Dado un archivo de números enteros, ordénelo e imprímalo.

18. En el archivo EMPLEADOS se tiene información sobre los empleados de una empresa. Los datos almacenados por cada empleado son:

- Nombre
- Estado civil
- Antigüedad

- Categoría
- Sueldo

Ordene el archivo según el campo "nombre".

a) Aplique mezcla directa.
b) Aplique mezcla equilibrada.

19. Se tiene un archivo con información sobre huéspedes de un hotel:

- Número de habitación
- Nombre del huésped
- Fecha de llegada

Ordene el archivo aplicando el método de mezcla directa.

a) Según el campo "número de habitación".
b) Según el campo "nombre".

20. Dado un archivo de cadenas de caracteres, ordénelo en forma descendente aplicando el método de mezcla equilibrada.

21. Se tienen dos archivos ordenados con los nombres de los estudiantes de una escuela. No se han actualizado de la misma manera los dos archivos, habiéndose dado de alta a algunos alumnos en un archivo pero no en el otro. Escriba un programa que obtenga un tercer archivo, también ordenado, intercalando la información de los dos archivos existentes. (No deben quedar elementos repetidos.)

22. Se tienen tres archivos A1, A2 y A3 con información sobre recitales efectuados en un teatro, a lo largo de los tres últimos años. Cada registro de los archivos tiene los siguientes campos:

- Nombre del cantante u orquesta que ofreció el recital
- Número de presentaciones
- Fechas de las presentaciones

Los tres archivos están ordenados en forma ascendente según el primer campo.
Escriba un programa que intercale los tres archivos, formando el archivo RECITALES.

Capítulo 9

MÉTODOS DE BÚSQUEDA

9.1 INTRODUCCIÓN

Este capítulo se dedica al estudio de una de las operaciones más importantes en el procesamiento de información: la **búsqueda**. Esta operación se utiliza básicamente para recuperar datos que se habían almacenado con anticipación. El resultado puede ser de éxito si se encuentra la información deseada, o de fracaso, en caso contrario.

La búsqueda ocupa una parte importante de nuestra vida. Prácticamente todo el tiempo estamos *buscando* algo. El mundo en que se vive hoy día es desarrollado, automatizado, y la información representa un elemento de vital importancia. Es fundamental estar informados y, por lo tanto, buscar y recuperar información. Por ejemplo, se buscan números telefónicos en un directorio, ofertas laborales en un periódico, libros en una biblioteca, etcétera.

FIGURA 9.1
Ejemplo práctico de búsqueda.

FIGURA 9.2
Ejemplo práctico de búsqueda.

En los ejemplos mencionados, la búsqueda se realiza, generalmente, sobre elementos que están ordenados. Los directorios telefónicos están organizados alfabéticamente, las ofertas laborales están ordenadas por tipo de trabajo y los libros de una biblioteca están clasificados por tema. Sin embargo, puede suceder que la búsqueda se realice sobre una colección de elementos no ordenados. Por ejemplo, cuando se busca la localización de una ciudad dentro de un mapa.

Se concluye que la operación de búsqueda se puede llevar a cabo sobre elementos ordenados o desordenados. Cabe destacar que la búsqueda es más fácil y ocupa menos tiempo cuando los datos se encuentran ordenados.

Los métodos de búsqueda se pueden clasificar en **internos** y **externos**, según la ubicación de los datos sobre los cuales se realizará la búsqueda. Se denomina **búsqueda interna** cuando todos los elementos se encuentran en la memoria principal de la computadora; por ejemplo, almacenados en arreglos, listas ligadas o árboles. Es **búsqueda externa** si los elementos están en memoria secundaria; es decir, si hubiera archivos en dispositivos como cintas y discos magnéticos.

En la siguiente sección se estudiarán los métodos internos, posteriormente en la sección 9.3 se hará lo propio con los externos.

9.2 BÚSQUEDA INTERNA

La **búsqueda interna** trabaja con elementos que se encuentran almacenados en la memoria principal de la máquina. Éstos pueden estar en estructuras estáticas —arreglos— o dinámicas —listas ligadas y árboles—. Los métodos de búsqueda interna más importantes son:

- Secuencial o lineal
- Binaria

- Por transformación de claves
- Árboles de búsqueda

9.2.1 Búsqueda secuencial

La **búsqueda secuencial** consiste en revisar elemento tras elemento hasta encontrar el dato buscado, o llegar al final del conjunto de datos disponible.

Primero se tratará sobre la búsqueda secuencial en arreglos, y luego en listas enlazadas. En el primer caso, se debe distinguir entre arreglos ordenados y desordenados.

Esta última consiste, básicamente, en recorrer el arreglo de izquierda a derecha hasta que se encuentre el elemento buscado o se termine el arreglo, lo que ocurra primero. Normalmente cuando una función de búsqueda concluye con éxito, interesa conocer en qué posición fue hallado el elemento que se estaba buscando. Esta idea se puede generalizar para todos los métodos de búsqueda.

A continuación se presenta el algoritmo de búsqueda secuencial en arreglos desordenados.

Algoritmo 9.1 Secuencial_desordenado

Secuencial_desordenado (*V*, *N*, *X*)

{Este algoritmo busca secuencialmente el elemento *X* en un arreglo unidimensional desordenado *V*, de *N* componentes}
{*I* es una variable de tipo entero}

1. Hacer $I \leftarrow 1$
2. Mientras (($I \leq N$) y ($V[I] \neq X$)) *Repetir*
Hacer $I \leftarrow I + 1$
3. {Fin del ciclo del paso 2}
4. *Si* ($I > N$)
entonces
Escribir "La información no está en el arreglo"
si no
Escribir "La información se encuentra en la posición", *I*
5. {Fin del condicional del paso 4}

Son dos los posibles resultados que se pueden obtener al aplicar este algoritmo: la posición en la que encontró el elemento, o un mensaje de fracaso si el elemento no se halla en el arreglo. Si hubiera dos o más ocurrencias del mismo valor, se encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias del dato buscado.

A continuación se presenta una variante de este algoritmo, pero utilizando recursividad, en lugar de iteratividad.

Algoritmo 9.2 Secuencial_desordenado_recursivo

Secuencial_desordenado_recursivo (*V*, *N*, *X*, *I*)

{Este algoritmo busca secuencialmente, y de forma recursiva, al elemento *X* en el arreglo unidimensional desordenado *V*, de *N* componentes}
{*I* es un parámetro de tipo entero que inicialmente se encuentra en 1}

1. *Si* (I > *N*)
entonces
Escribir "La información no se encuentra en el arreglo"
si no
1.1 *Si* (*V*[*I*] = *X*)
entonces
Escribir "La información se encuentra en la posición", *I*
si no
Regresar a Secuencial_desordenado_recursivo con *V*, *N*, *X* e *I* + 1
1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

La búsqueda secuencial en arreglos ordenados es similar al caso anterior. Sin embargo, el orden entre los elementos del arreglo permite incluir una nueva condición que hace más eficiente al proceso. A continuación analicemos el algoritmo de búsqueda secuencial en arreglos ordenados.

Algoritmo 9.3 Secuencial_ordenado

Secuencial_ordenado (*V*, *N*, *X*)

{Este algoritmo busca secuencialmente al elemento *X* en un arreglo unidimensional ordenado *V*, de *N* componentes. *V* se encuentra ordenado crecientemente: *V*[1] ≤ *V*[2] ≤ ... ≤ *V*[*N*]}
{*I* es una variable de tipo entero}

1. Hacer *I* ← 1
2. Mientras ((*I* ≤ *N*) y (*X* > *V*[*I*])) *Repetir*
Hacer *I* ← *I* + 1
3. {Fin del ciclo del paso 2)
4. *Si* ((*I* > *N*) o (*X* < *V*[*I*]))
entonces
Escribir "La información no se encuentra en el arreglo"
si no
Escribir "La información se encuentra en la posición", *I*
5. {Fin del condicional del paso 4}

Como el arreglo está ordenado, se establece una nueva condición: el elemento buscado tiene que ser mayor que el del arreglo. Cuando el ciclo se interrumpe, se evalúa cuál de las condiciones es falsa. Si ($I > N$) o si se comparó el elemento con un valor mayor a sí mismo ($X < V[I]$), se está ante un caso de fracaso: el elemento no está en el arreglo. Si $X = V[I]$ entonces se encontró al elemento en el arreglo.

A continuación se presenta el algoritmo de búsqueda en arreglos ordenados, pero en forma recursiva.

Algoritmo 9.4 Secuencial_ordenado_recursivo

Secuencial_ordenado_recursivo (V, N, X, I)

{Este algoritmo busca en forma secuencial y recursiva al elemento X en un arreglo unidimensional ordenado V, de N componentes. V se encuentra ordenado de manera creciente: $V[1] \leq V[2] \leq ... \leq V[N]$. I inicialmente tiene el valor de 1}

```
1. Si ((I ≤ N) y (X > V[I]))
       entonces
           Llamar a Secuencial_ordenado_recursivo con V, N, X e I + 1
       si no
       1.1 Si ((I > N) o (X < V[I]))
               entonces
                   Escribir "La información no se encuentra en el arreglo"
               si no
                   Escribir "La información se encuentra en la posición", I
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

El método de búsqueda secuencial también se puede aplicar a listas ligadas. Consiste en recorrer la lista nodo tras nodo, hasta encontrar al elemento buscado —éxito—, o hasta que lleguemos al final de la lista —fracaso—. La lista, como en el caso de arreglos, se puede encontrar ordenada o desordenada. El orden en el cual se puede recorrer la lista depende de sus características; puede ser simplemente ligada, circular o doblemente ligada. En este capítulo se presentará el caso de búsqueda secuencial en listas simplemente ligadas desordenadas. El lector, con los conocimientos que tiene sobre listas y búsqueda, puede implementar fácilmente los otros algoritmos.

Algoritmo 9.5 Secuencial_lista_desordenada

Secuencial_lista_desordenada (P, X)

{Este algoritmo busca en forma secuencial al elemento X en una lista simplemente ligada, que almacena información que está desordenada. P es un apuntador al primer nodo de la lista. INFO y LIGA son los campos de cada nodo}
{Q es una variable de tipo apuntador}

```
1. Hacer Q ← P
2. Mientras ((Q ≠ NIL) y (Q^.INFO ≠ X)) Repetir
        Hacer Q ← Q^.LIGA
3. {Fin del ciclo del paso 2}
4. Si (Q = NIL)
        entonces
            Escribir "La información no se encuentra en la lista"
        si no
            Escribir "La información se encuentra en la lista"
5. {Fin del condicional del paso 4}
```

Si la lista estuviera ordenada se modificaría este algoritmo, incluyendo una condición similar a la que se escribió en el algoritmo 9.3. Esto último con el objetivo de disminuir el número de comparaciones.

A continuación se presenta la variante recursiva de este algoritmo de búsqueda secuencial en listas simplemente ligadas desordenadas.

Algoritmo 9.6 Secuencial_lista_desordenada_recursivo

Secuencial_lista_desordenada_recursivo (*P*, *X*)

```
{Este algoritmo busca de manera secuencial y en forma recursiva al elemento X en una lista
simplemente ligada, que almacena información que está desordenada. P es un apuntador al
primer nodo de la lista. INFO y LIGA son los campos de cada nodo}

1. Si ((P ≠ NIL) y (P^.INFO ≠ X))
        entonces
            Regresar a Secuencial_lista_desordenada_recursivo con P^.LIGA y X
        si no
   1.1  Si (P = NIL)
            entonces
                Escribir "La información no se encuentra en la lista"
            si no
                Escribir "La información se encuentra en la lista"
   1.2  {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

Análisis de la búsqueda secuencial

El número de comparaciones es uno de los factores más importantes que se utilizan para determinar la complejidad de los métodos de búsqueda. Para analizar la complejidad de la búsqueda secuencial, se deben establecer los casos más favorable o desfavorable que se presenten.

Al buscar, por ejemplo, un elemento en un arreglo unidimensional desordenado de N componentes, puede suceder que ese valor no se encuentre; por lo tanto, se harán N

comparaciones al recorrer todo el arreglo. Por otra parte, si el elemento se encuentra en el arreglo, éste puede estar en la primera posición, en la última o en alguna intermedia. Si es el primero, se hará una comparación; si se trata del último, se harán N comparaciones; y si se encuentra en la posición i ($1 < i < N$), entonces se realizarán i comparaciones.

Ahora bien, el número de comparaciones que se llevan a cabo si trabajamos con arreglos ordenados será el mismo que para desordenados, siempre y cuando el elemento se encuentre en el arreglo. Si no fuera éste el caso, entonces el número de comparaciones disminuirá sensiblemente en arreglos ordenados, siempre que el valor buscado esté comprendido entre el primero y el último elementos del arreglo.

Por otra parte, el número de comparaciones en la búsqueda secuencial en listas simplemente ligadas es el mismo que para arreglos. En la fórmula 9.1 se presentan los números mínimo, mediano y máximo de comparaciones que se ejecutan cuando se trabaja con la búsqueda secuencial.

Fórmula 9.1

$$C_{\text{mín}} = 1 \qquad C_{\text{med}} = \frac{(1+n)}{2} \qquad C_{\text{máx}} = N$$

La tabla 9.1 presenta, para distintos valores de N, los números mínimo, mediano y máximo de comparaciones que se requieren para buscar secuencialmente un elemento en un arreglo o lista ligada.

9.2.2 Búsqueda binaria

La **búsqueda binaria** consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. Para el caso de que no fueran iguales se redefinen los extremos del intervalo, según el elemento central sea mayor o menor que el elemento buscado, disminuyendo de esta forma el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado, o cuando el intervalo de búsqueda se anula, es vacío.

El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas —no podríamos retroceder para establecer intervalos de búsqueda— ni con arreglos desordenados. Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de com-

TABLA 9.1
Complejidad del método de búsqueda secuencial

N	$C_{\text{mín}}$	C_{med}	$C_{\text{máx}}$
10	1	5.5	10
100	1	50.5	100
500	1	250.5	500
1 000	1	500.5	1 000
10 000	1	5 000.5	10 000

paraciones a realizar disminuye notablemente. Esta disminución resulta significativa cuanto más grande sea el tamaño del arreglo. A continuación se presenta el algoritmo de búsqueda binaria.

Algoritmo 9.7 Binaria

Binaria (*V*, *N*, *X*)

{Este algoritmo busca al elemento *X* en un arreglo unidimensional ordenado *V* de *N* componentes}
{IZQ, CEN y DER son variables de tipo entero. BAN es una variable de tipo booleano}

```
1. Hacer IZQ ← 1, DER ← N y BAN ← FALSO
2. Mientras ((IZQ ≤ DER) y (BAN = FALSO)) Repetir
   2.1 Si (X = V[CEN])
            entonces
                Hacer BAN ← VERDADERO
            si no {Se redefine el intervalo de búsqueda}
       2.1.1 Si (X > V[CEN])
                    entonces
                        Hacer IZQ ← CEN + 1
                    si no
                        Hacer DER ← CEN – 1
       2.1.2 {Fin del condicional del paso 2.1.1)
   2.2 {Fin del condicional del paso 2.1)
3. {Fin del ciclo del paso 2)
4. Si (BAN = VERDADERO)
       entonces
           Escribir "La información está en la posición", CEN
       si no
           Escribir "La información no se encuentra en el arreglo"
5. {Fin del condicional del paso 4}
```

Analicemos ahora un ejemplo para ilustrar el funcionamiento de este algoritmo.

Ejemplo 9.1

Sea *V* un arreglo unidimensional de números enteros, ordenado de manera creciente, como se muestra en la figura 9.3.

En la tabla 9.2 se presenta el seguimiento del algoritmo 9.7 cuando *X* es igual a 325 (*X* = 325).

En la figura 9.4 se observa gráficamente, para este caso en particular, cómo se va reduciendo el intervalo de búsqueda.

FIGURA 9.3

V

101	215	325	410	502	507	600	610	612	670
1	2	3	4	5	6	7	8	9	10

TABLA 9.2
Búsqueda binaria

Paso	BAN	IZQ	DER	CEN	$X = V[CEN]$	$X > V[CEN]$
1	Falso	1	10	5	325 = 502 ? No	325 > 502 ? No
2	Falso	1	4	2	325 = 215 ? No	325 > 215 ? Sí
3	Falso	3	4	3	325 = 325 ? Sí	
4	Verdadero					

La tabla 9.3, por otra parte, muestra nuevamente el seguimiento del algoritmo 9.7 para $X = 615$, valor que no se encuentra en el arreglo.

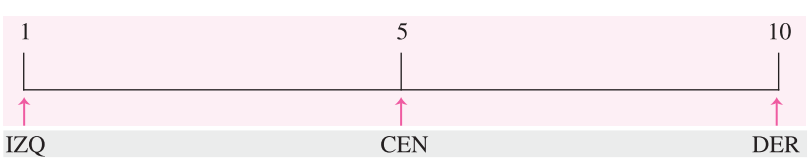
La figura 9.5 representa gráficamente cómo se va reduciendo el intervalo de búsqueda hasta anularse (DER < IZQ).

A continuación se presenta una variante del algoritmo de búsqueda binaria que no utiliza bandera —BAN—.

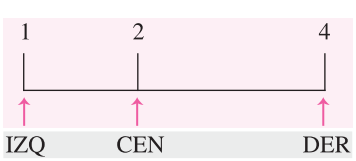
TABLA 9.3
Búsqueda binaria

Paso	BAN	IZQ	DER	CEN	$X = V[CEN]$	$X > V[CEN]$
1	Falso	1	10	5	615 = 502 ? No	615 > 502 ? Sí
2	Falso	6	10	8	615 = 610 ? No	615 > 610 ? Sí
3	Falso	9	10	9	615 = 612 ? No	615 > 612 ? Sí
4	Falso	10	10	10	615 = 670 ? No	615 > 670 ? No
5	Falso	10	9			

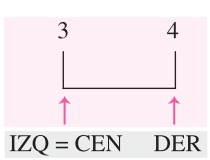
FIGURA 9.4
Reducción del intervalo de búsqueda. *a*) Paso 1. *b*) Paso 2. *c*) Paso 3 de la tabla 9.2.



a)



b)



c)

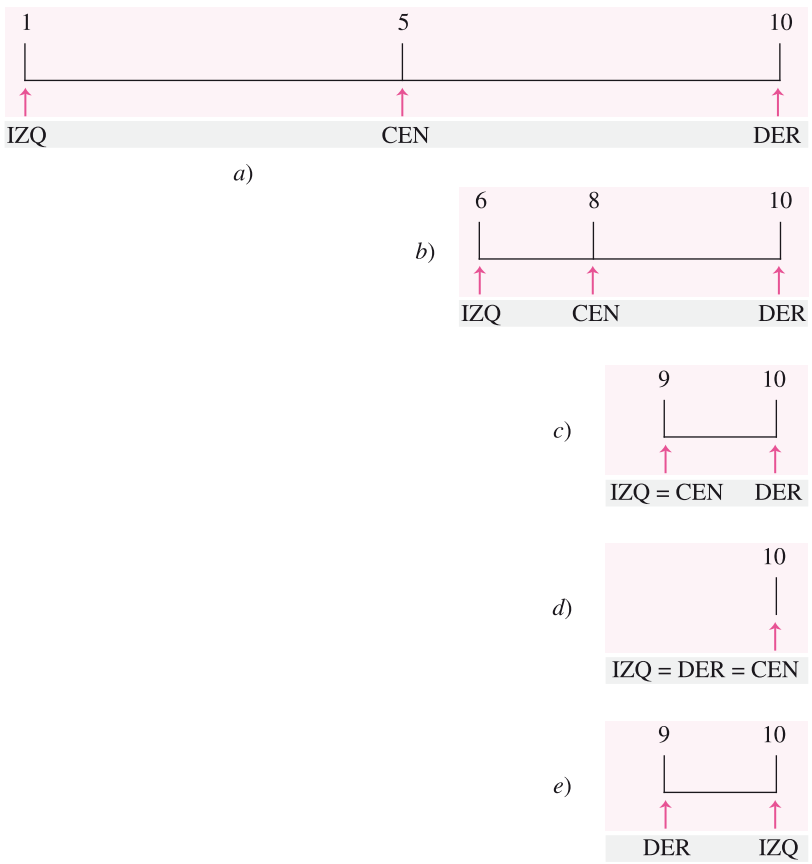


FIGURA 9.5
Reducción del intervalo de búsqueda. *a*) Paso 1. *b*) Paso 2. *c*) Paso 3. *d*) Paso 4. *e*) Paso 5 de la tabla 9.3.

Algoritmo 9.8 Binaria_sin_bandera

Binaria_sin_bandera (*V*, *N*, *X*)

{Este algoritmo busca al elemento *X* en el arreglo unidimensional ordenado *V* de *N* componentes}
{IZQ, DER y CEN son variables de tipo entero}

1. Hacer IZQ ← 1, DER ← *N* y CEN ← PARTE ENTERA ((IZQ + DER)/2)
2. Mientras ((IZQ ≤ DER) y (*X* ≠ *V*[CEN])) *Repetir*
  2.1 *Si X* > *V*[CEN]
    entonces
      Hacer IZQ ← CEN + 1
    si no
      Hacer DER ← CEN – 1
  2.2 {Fin del condicional del paso 2.1}
    Hacer CEN ← PARTE ENTERA ((IZQ + DER)/2)
3. {Fin del ciclo del paso 2}
4. *Si* (IZQ > DER)

```
        entonces
            Escribir "La información no se encuentra en el arreglo"
        si no
            Escribir "La información se encuentra en la posición", CEN
5. {Fin del condicional del paso 4}
```

Finalmente, se presenta una versión recursiva de este algoritmo de búsqueda binaria.

Algoritmo 9.9 Binaria_recursivo

```
Binaria_recursivo (V, IZQ, DER, X)

{Este algoritmo busca al elemento X en el arreglo unidimensional ordenado V de N componentes. IZQ ingresa inicialmente al algoritmo con el valor de 1. DER, por otra parte, ingresa con el valor de N}
{CEN es una variable de tipo entero}

1. Si (IZQ ≥ DER)
        entonces
            Escribir X, "No se encuentra en el arreglo"
        si no
            Hacer CEN ← PARTE ENTERA ((DER + IZQ)/2)
   1.1 Si (X = V[CEN])
          entonces
             Escribir "El dato buscado se encuentra en la posición", CEN
          si no
       1.1.1 Si (X > V[CEN])
                entonces
                   Regresar a Binaria_recursivo con V, CEN + 1, DER, X
                si no
                   Regresar a Binaria_recursivo con V, IZQ, CEN – 1, X
       1.1.2 {Fin del condicional del paso 1.1.1}
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

Análisis de la búsqueda binaria

Para analizar la complejidad del método de búsqueda binaria es necesario establecer los casos más favorables y desfavorables que se pudieran presentar en el proceso de búsqueda. El primero sucede cuando el elemento buscado es el central, en dicho caso se hará una sola comparación; el segundo sucede cuando el elemento no se encuentra en el arreglo; entonces se harán aproximadamente $\log_2 (n)$ comparaciones, ya que con cada comparación el número de elementos en los cuales se debe buscar se reduce en un factor de 2. De esta forma, se determinan los números mínimo, mediano y máximo de comparaciones que se deben realizar cuando se utiliza este tipo de búsqueda.

$$C_{\text{mín}} = 1 \qquad C_{\text{med}} = \frac{\left(1 + \log_2(N)\right)}{2} \qquad C_{\text{máx}} = \log_2(N)$$

Fórmula 9.2

En la tabla 9.4 se presentan, para distintos valores de *N*, los números mínimo, mediano y máximo de comparaciones requeridas para buscar un elemento en un arreglo, aplicando el método de búsqueda binaria.

Si se comparan los valores de la tabla 9.1 con los de la tabla 9.4 resulta claro que el método de búsqueda binaria es más eficiente que el método de búsqueda secuencial. Además, la diferencia se hace más significativa conforme más grande sea el tamaño del arreglo. Sin embargo, no hay que olvidar que el método de búsqueda binaria trabaja solamente con arreglos ordenados; por lo tanto, si el arreglo estuviera desordenado antes de emplear este método, aquél debería ordenarse.

Cabe destacar, sin embargo, que la ordenación de un arreglo también implica comparaciones y movimientos de elementos, así que si se va a realizar sólo una búsqueda sobre un arreglo desordenado conviene utilizar el método secuencial. En cambio, si se realizan búsquedas en forma continua, convendría ordenarlo para poder aplicar el método de búsqueda binaria.

9.2.3 Búsqueda por transformación de claves

Los dos métodos analizados anteriormente permiten encontrar un elemento en un arreglo. En ambos casos el tiempo de búsqueda es proporcional a su número de componentes. Es decir, a mayor número de elementos se debe realizar mayor número de comparaciones. Se mencionó además que si bien el método de búsqueda binaria es más eficiente que el secuencial, existe la restricción de que el arreglo se debe encontrar ordenado.

Esta sección se dedica a un nuevo método de búsqueda. Este método, conocido como **transformación de claves** o ***hash***, permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta con la ventaja de que el tiempo de búsqueda es independiente del número de componentes del arreglo.

Supongamos que se tiene una colección de datos, cada uno de ellos identificado por una clave. Es claro que resulta atractivo tener acceso a ellos de manera directa; es decir, sin recorrer algunos datos antes de localizar al buscado. El método por transformación de claves permite realizar justamente esta actividad; es decir, localizar el dato en forma

TABLA 9.4
Complejidad del método de búsqueda binaria

N	$C_{\text{mín}}$	C_{med}	$C_{\text{máx}}$
10	1	2.5	4
100	1	4	7
500	1	5	9
1 000	1	5.5	10
10 000	1	7.5	14

directa. El método trabaja utilizando una función que convierte una clave dada en una dirección —índice— dentro del arreglo.

$$\text{dirección} \leftarrow H\ (\text{clave})$$

La función *hash* (H) aplicada a la clave genera un índice del arreglo que permite acceder directamente al elemento. El caso más trivial se presenta cuando las claves son números enteros consecutivos.

Supongamos que se desea almacenar la información relacionada con 100 alumnos cuyas matrículas son números del 1 al 100. En este caso conviene definir un arreglo de 100 elementos con índices numéricos comprendidos entre los valores 1 y 100. Los datos de cada alumno ocuparán la posición del arreglo que se corresponda con el número de la matrícula; de esta manera se podrá acceder directamente a la información de cada alumno.

Consideremos ahora que se desea almacenar la información de 100 empleados. La clave de cada empleado corresponde al número de su seguro social. Si la clave está formada por 11 dígitos, resulta por completo ineficiente definir un arreglo con 99 999 999 999 elementos para almacenar solamente los datos de los 100 empleados. Utilizar un arreglo tan grande asegura la posibilidad de acceder directamente a sus elementos; sin embargo, el costo en memoria resulta tanto ridículo como excesivo. Siempre es importante equilibrar el costo del espacio de memoria con el costo por tiempo de búsqueda.

Cuando se tienen claves que no se corresponden con índices —por ejemplo, por ser alfanuméricas—, o cuando las claves representen valores numéricos muy grandes o no se corresponden con los índices de los arreglos, se utilizará una función *hash* que permita transformar la clave para obtener una dirección apropiada. Esta función *hash* debe ser simple de calcular y asignar direcciones de la manera más uniforme posible. Es decir, debe generar posiciones diferentes dadas dos claves también diferentes. Si esto último no ocurre ($H(K_1) = d$, $H(K_2) = d$ y $K_1 \neq K_2$) hay una **colisión**, que se define como la asignación de una misma dirección a dos o más claves distintas.

En este contexto, para trabajar con este método de búsqueda se debe seleccionar previamente:

- Una función *hash* que sea fácil de calcular y distribuya uniformemente las claves.
- Un método para resolver colisiones. Si éstas se presentan, se contará con algún método que genere posiciones alternativas.

Estos dos casos se tratarán en forma separada. Como ya se mencionó, seleccionar una buena función *hash* es muy importante, pero es difícil encontrarla. Básicamente porque no existen reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en su distribución. Realizar un análisis de las principales características de las claves siempre ayuda en la elección de una función de este tipo. A continuación se explican algunas de las funciones *hash* más utilizadas.

9.2.4 Función *hash* por módulo: división

La función ***hash* por módulo** o **división** consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Supongamos, por ejemplo, que se

tiene un arreglo de N elementos, y K es la clave del dato a buscar. La función *hash* queda definida por la siguiente fórmula:

$$H(K) = (K \bmod N) + 1 \qquad \textbf{Fórmula 9.3}$$

En la fórmula 9.3 se observa que al residuo de la división se le suma 1, esto último con el objetivo de obtener un valor comprendido entre 1 y N.

Para lograr mayor uniformidad en la distribución, es importante que N sea un número primo o divisible entre muy pocos números. Por lo tanto, si N no es un número primo, se debe considerar el valor primo más cercano.

En el ejemplo 9.2 se presenta un caso de función *hash* por módulo.

Ejemplo 9.2

Supongamos que $N = 100$ es el tamaño del arreglo, y las direcciones que se deben asignar a los elementos (al guardarlos o recuperarlos) son los números del 1 al 100. Consideremos además que $K_1 = 7\,259$ y $K_2 = 9\,359$ son las dos claves a las que se deben asignar posiciones en el arreglo. Si aplicamos la fórmula 9.3 con $N = 100$, para calcular las direcciones correspondientes a K_1 y K_2, obtenemos:

$$H(K_1) = (7\,259 \bmod 100) + 1 = 60$$
$$H(K_2) = (9\,359 \bmod 100) + 1 = 60$$

Como $H(K_1)$ es igual a $H(K_2)$ y K_1 es distinto de K_2, se está ante una colisión que se debe resolver porque a los dos elementos le correspondería la misma dirección.

Observemos, sin embargo, que si aplicáramos la fórmula 9.3 con un número primo cercano a N, el resultado cambiaría:

$$H(K_1) = (7\,259 \bmod 97) + 1 = 82$$
$$H(K_2) = (9\,359 \bmod 97) + 1 = 48$$

Con $N = 97$ se ha eliminado la colisión.

9.2.5 Función *hash* cuadrado

La función ***hash* cuadrado** consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos que se debe considerar se encuentra determinado por el rango del índice. Sea K la clave del dato a buscar, la función *hash* cuadrado queda definida, entonces, por la siguiente fórmula:

$$H(K) = \text{dígitos\_centrales}\ (K^2) + 1 \qquad \textbf{Fórmula 9.4}$$

La suma de una unidad a los dígitos centrales es útil para obtener un valor comprendido entre 1 y N.

En el ejemplo 9.3 se presenta un caso de función *hash* cuadrado.

Ejemplo 9.3

Sea $N = 100$ el tamaño del arreglo, y sus direcciones los números comprendidos entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.4 para calcular las direcciones correspondientes a K_1 y K_2:

$K_1^2 = 52\,693\,081$
$K_2^2 = 87\,590\,881$

$H(K_1)$ = dígitos_centrales (52 693 081) + 1 = 94
$H(K_2)$ = dígitos_centrales (87 590 881) + 1 = 91

Como el rango de índices en nuestro ejemplo varía de 1 a 100, se toman solamente los dos dígitos centrales del cuadrado de las claves.

9.2.6 Función *hash* por plegamiento

La **función *hash* por plegamiento** consiste en dividir la clave en partes, tomando igual número de dígitos aunque la última puede tener menos, y operar con ellas, asignando como dirección los dígitos menos significativos. La operación entre las partes se puede realizar por medio de sumas o multiplicaciones. Sea K la clave del dato a buscar. K está formada por los dígitos $d_1, d_2, ..., d_n$. La función *hash* por plegamiento queda definida por la siguiente fórmula:

$$H(K) = \text{dígmensig}((d_1 \ldots d_i) + (d_{i+1} \ldots d_j) + \ldots + (d_1 \ldots d_n)) + 1 \qquad \textbf{Fórmula 9.5}$$

El operador que aparece en la fórmula operando las partes de la clave es el de suma, pero, como ya se aclaró, puede ser el de la multiplicación. En este contexto, la suma de una unidad a los dígitos menos significativos —dígmensig— es para obtener un valor comprendido entre 1 y N.

En el ejemplo 9.4 se presenta un caso de función *hash* por plegamiento.

Ejemplo 9.4

Sea $N = 100$ el tamaño del arreglo, y las direcciones que deben tomar sus elementos los números comprendidos entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.5 para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32
$H(K_2)$ = dígmensig (93 + 59) + 1 = dígmensig (152) + 1 = 53

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

9.2.7 Función *hash* por truncamiento

La **función *hash* por truncamiento** consiste en tomar algunos dígitos de la clave y formar con ellos una dirección. Este método es de los más sencillos, pero es también de los que ofrecen menos uniformidad en la distribución de las claves.

Sea K la clave del dato a buscar. K está formada por los dígitos d_1, d_2, ..., d_n. La función *hash* por truncamiento se representa con la siguiente fórmula:

$$H(K) = \text{elegirdígitos } (d_1, d_2 \ldots d_n) + 1 \qquad \textbf{Fórmula 9.6}$$

La elección de los dígitos es arbitraria. Se podrían tomar los de las posiciones impares o de las pares. Luego se podrían unir de izquierda a derecha o de derecha a izquierda. La suma de una unidad a los dígitos seleccionados es útil para obtener un valor entre 1 y 100.

En el ejemplo 9.5 se muestra un caso de función *hash* por truncamiento.

Ejemplo 9.5

Sea $N = 100$ el tamaño del arreglo, y las direcciones de sus elementos los números entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.6 para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1) = \text{elegirdígitos } (7\,259) + 1 = 75 + 1 = 76$
$H(K_2) = \text{elegirdígitos } (9\,359) + 1 = 95 + 1 = 96$

En este ejemplo se toman el primero y tercer números de la clave y se unen de izquierda a derecha.

Es importante destacar que en todos los casos anteriores se presentaron ejemplos de claves numéricas. Sin embargo, en la práctica las claves pueden ser alfabéticas o alfanuméricas. En general, cuando aparecen letras en las claves se suele asociar a cada una un entero con el propósito de convertirlas en numéricas.

A	B	C	D	... Z
01	02	03	04	... 27

Si, por ejemplo, la clave fuera **ADA**, su equivalente numérica sería **010401**. Si hubiera combinación de letras y números, se procedería de la misma manera. Por ejemplo, dada una clave **Z4F21**, su equivalente numérica sería **2740621**. Otra alternativa sería tomar el valor decimal asociado para cada carácter según el código ASCII. Una vez obtenida la clave en su forma numérica, se puede utilizar normalmente cualesquiera de las funciones antes mencionadas. El ejemplo 9.11 ilustra un caso de clave alfabética.

9.2.8 Solución de colisiones

La elección de un método adecuado para resolver **colisiones** es tan importante como la elección de una buena función *hash*. Cuando ésta obtiene una misma dirección para dos claves diferentes, se está ante una colisión. Normalmente, cualquiera que sea el método

elegido resulta costoso tratar las colisiones. Es por ello que se debe hacer un esfuerzo importante para encontrar la función que ofrezca la mayor uniformidad en la distribución de las claves.

La manera más natural de resolver el problema de las colisiones es reservar una casilla por clave; es decir, aquellas que se correspondan una a una con las posiciones del arreglo. Pero, como ya se mencionó, esta solución puede tener un alto costo en memoria; por lo tanto, se deben analizar otras alternativas que permitan equilibrar el uso de memoria con el tiempo de búsqueda.

En adelante se estudiarán algunos de los métodos más utilizados para resolver colisiones, que se pueden clasificar en:

- Reasignación
- Arreglos anidados
- Encadenamiento

9.2.9 Reasignación

Existen varios métodos que trabajan bajo el principio de comparación y reasignación de elementos. A continuación se analizarán tres de ellos:

- Prueba lineal
- Prueba cuadrática
- Doble dirección *hash*

Prueba lineal

El método de **prueba lineal** consiste en que una vez que se detecta la colisión, se recorre el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o cuando se encuentra una posición vacía. El arreglo se trata como una estructura circular: *el siguiente elemento después del último es el primero*.

A continuación se expone el algoritmo de solución de colisiones por medio de la prueba lineal.

Algoritmo 9.10 Prueba_lineal

Prueba_lineal (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve el problema de las colisiones por medio del método de prueba lineal}
{*D* y *DX* son variables de tipo entero}

1. Hacer $D \leftarrow H(K)$ {Genera dirección}
2. *Si* (($V[DX] \neq$ VACÍO) y ($V[D] = K$))
   entonces

```
        Escribir "La información está en la posición", D
    si no
        Hacer DX ← D + 1
2.1 Mientras ((DX ≤ N) y (V[DX] ≠ VACÍO) y (V[DX] ≠ K) y (DX ≠ D))
        Repetir
        Hacer DX ← DX + 1
    2.1.1 Si (DX = N + 1) entonces
            Hacer DX ← 1
    2.1.2 {Fin del condicional del paso 2.1.1}
2.2 {Fin del ciclo del paso 2.1}
2.3 Si ((V[DX] = VACÍO) o (DX = D))
      entonces
        Escribir "La información no se encuentra en el arreglo"
      si no
        Escribir "La información está en la posición", DX
2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

La cuarta condición del ciclo del punto 2.1, ($DX \neq X$), es para evitar caer en un ciclo infinito si el arreglo estuviera lleno y el elemento a buscar no se encontrara en él.

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo podrían permanecer vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial, perdiendo así las ventajas del método *hash*. El ejemplo 9.6 ilustra el funcionamiento del algoritmo 9.10.

Ejemplo 9.6

Sea V un arreglo unidimensional de 10 elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 fueron asignadas según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.6 se aprecia el estado de arreglo (9.6*a*) y la tabla con $H(K)$ para cada clave (9.6*b*).

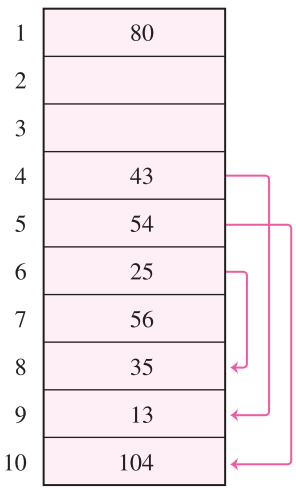
En la tabla 9.5 se presenta el seguimiento de las variables importantes del algoritmo 9.10 para el caso del ejemplo anterior. El dato a buscar es igual a 35.

Al aplicar la función hash a la clave 35, se obtiene una dirección (D) igual a 6. Sin embargo, en esa posición no se encuentra el elemento buscado, por lo que se comienza a recorrer secuencialmente el arreglo a partir de la posición (DX) igual a 7. En este caso la búsqueda concluye cuando se encuentra al valor buscado en la posición 8.

TABLA 9.5
Solución de colisiones por la prueba lineal. $K = 35$

D	*DX*
6	7
	8

FIGURA 9.6
Solución de colisiones por la prueba lineal. *a*) Arreglo. *b*) Tabla con $H(K)$.



a)

K	$H(K)$
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

En la tabla 9.6 se presenta ahora el seguimiento de las variables importantes del algoritmo 9.10, pero ahora para un caso más complejo del ejemplo anterior. El dato a buscar es igual a 13.

Prueba cuadrática

El método de la **prueba cuadrática** es similar al anterior. La diferencia consiste en que en el de la prueba cuadrática las direcciones alternativas se generarán como $D + 1$, $D + 4$, $D + 9$, ..., $D + i^2$ en vez de $D + 1$, $D + 2$, ..., $D + i$. Esta variación permite una mejor distribución de las claves que colisionan.

A continuación se presenta el algoritmo de solución de colisiones por medio de la prueba cuadrática.

TABLA 9.6
Solución de colisiones por la prueba lineal. $K = 13$

D	DX
4	5
	6
	7
	8
	9

Algoritmo 9.11 Prueba_cuadrática

Prueba_cuadrática (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve el problema de las colisiones por medio de la prueba cuadrática}
{*D*, *DX* e *I* son variables de tipo entero}

```
1. Hacer D ← H(K) {Genera dirección}
2. Si ((V[DX] ≠ VACÍO) y (V[D] = K))
      entonces
         Escribir "La información está en la posición", D
      si no
         Hacer I ← 1 y DX ← (D + (I * I))
      2.1 Mientras ((V[DX] ≠ VACÍO) y (V[DX] ≠ K)) Repetir
               Hacer I ← I + 1 y DX ← (D + (I * I))
        2.1.1 Si (DX > N) entonces
                  Hacer I ← 0, DX ← 1 y D ← 1
        2.1.2 {Fin del condicional del paso 2.1.1}
      2.2 {Fin del ciclo del paso 2.1}
      2.3 Si (V[DX] = VACÍO)
            entonces
               Escribir "La información no está en el arreglo"
            si no
               Escribir "La información está en la posición", DX
      2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

A continuación se presenta un ejemplo que ilustra el funcionamiento del algoritmo 9.11.

Ejemplo 9.7

Sea *V* un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80, 104 y 55 se asignaron según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

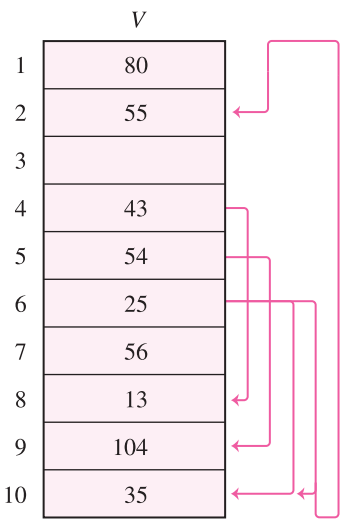
En la figura 9.7 se presenta el estado del arreglo (9.7*a*) y la tabla con $H(K)$ para cada clave (9.7*b*).

La tabla 9.7 contiene el seguimiento de las variables importantes del algoritmo 9.11 para el caso del ejemplo anterior, y el dato a buscar es igual a 35.

TABLA 9.7
Solución de colisiones por la prueba cuadrática. $K = 35$

D	*I*	*DX*
6	1	7
	2	10

FIGURA 9.7
Solución de colisiones por la prueba cuadrática. *a*) Arreglo. *b*) Tabla con *H*(*K*).



a)

K	*H*(*K*)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5
55	6

b)

Al aplicar la función *hash* a la clave 35, se obtiene una dirección (*D*) igual a 6; sin embargo, en esa dirección no se encuentra el elemento buscado. Se calcula posteriormente *DX*, como la suma *D* + (*I* * *I*), obteniéndose de esta forma la dirección 7. El algoritmo de búsqueda concluye cuando se encuentra el valor deseado en la décima posición del arreglo.

En la tabla 9.8 se presenta el seguimiento de las variables importantes del algoritmo 9.11 para un caso más complejo que el anterior. El dato a buscar es 55.

Doble dirección *hash*

El método de **doble dirección *hash*** consiste en que una vez que se detecta la colisión, se genera otra dirección aplicando la misma función *hash* a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o cuando se encuentra una posición vacía.

TABLA 9.8
Solución de colisiones por la prueba cuadrática. *K* = 55

D	*I*	*DX*
6	1	7
	2	10
	3	15
1	0	1
	1	2

$DH(K)$
$D'(H(D))$
$D''(H(D'))$
...

La función *hash* que se aplica no necesariamente tiene que ser la misma que originalmente se aplicó a la clave; podría ser cualquier otra. Sin embargo, no existe ningún estudio que precise cuál es la mejor función que se debe utilizar en el cálculo de las direcciones sucesivas.

Analicemos ahora el algoritmo de solución de colisiones por medio del método de la doble dirección *hash*.

Algoritmo 9.12 Doble_dirección

Doble_dirección (*V*, *N*, *K*)

{Este algoritmo busca al dato con la clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve el problema de las colisiones por medio de la doble dirección *hash*}
{*D* y *DX* son variables de tipo entero}

1. Hacer $D \leftarrow H(K)$
2. *Si* (($V[DX] \neq$ VACÍO) y ($V[D] = K$))
entonces
Escribir "La información se encuentra en la posición", *D*
si no
Hacer $DX \leftarrow H'(D)$
2.1 Mientras (($DX \leq N$) y ($V[DX] \neq$ VACÍO) y ($V[DX] \neq K$) y ($DX \neq D$)) *Repetir*
Hacer $DX \leftarrow H'(DX)$
2.2 {Fin del ciclo del paso 2.1}
2.3 *Si* (($V[DX] =$ VACÍO) o ($V[DX] \neq K$))
entonces
Escribir "La información buscada no está en el arreglo"
si no
Escribir "La información está en la posición", *DX*
2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}

El siguiente ejemplo ilustra el funcionamiento de este algoritmo.

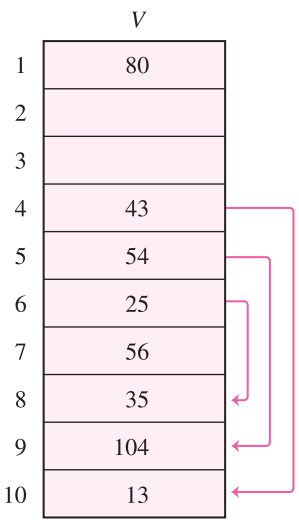
Ejemplo 9.8

Sea *V* un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 fueron asignadas según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

Además se definió una función H' para calcular direcciones alternativas en caso de haber colisión.

FIGURA 9.8
Solución de colisiones por el método de doble dirección *hash*. *a*) Arreglo. *b*) Tabla con $H(K)$, $H'(D)$, $H'(D')$, $H'(D'')$.



a)

K	$H(K)$	$H'(D)$	$H'(D')$	$H'(D'')$
25	6	–	–	–
43	4	–	–	–
56	7	–	–	–
35	6	8	–	–
54	5	–	–	–
13	4	6	8	10
80	1	–	–	–
104	5	7	9	–

b)

$$H'(D) = ((D + 1) \bmod 10) + 1$$

En la figura 9.8 se presenta el estado del arreglo (9.8*a*) y la tabla (9.8*b*) con $H(K)$ para cada clave, y $H'(D)$ en caso de colisión.

En la tabla 9.9 se presenta el seguimiento del algoritmo 9.12 para el caso del ejemplo anterior. El dato a buscar es igual a 13.

Al aplicar la función *hash* (H) a la clave 13, se obtuvo una dirección (D) igual a 4. Como en esa posición no se encuentra el elemento buscado, se aplica reiteradamente H', generando direcciones hasta localizar el valor deseado. En este ejemplo fue preciso aplicar tres veces la función H', obteniéndose las direcciones 6, 8 y 10, en la que finalmente se encontró el dato buscado.

9.2.10 Arreglos anidados

El método de **arreglos anidados** consiste en que cada elemento del arreglo tenga otro arreglo, en el cual se almacenen los elementos que colisionan. Si bien la solución parece ser sencilla, es claro que resulta ineficiente. Al trabajar con arreglos se depende del espacio que se haya asignado a éstos, lo cual conduce a un nuevo problema difícil

TABLA 9.9
Solución de colisiones por el método de doble dirección *hash*. $K = 13$

D	DX
4	6
	8
	10

de solucionar: elegir un tamaño adecuado de arreglo que permita un equilibrio entre el costo de memoria y el número de valores —que colisionan— que pudiera almacenar. Analicemos un ejemplo.

Ejemplo 9.9

Sea *V* un arreglo unidimensional de diez elementos. Los elementos con claves 25, 43, 56, 35, 54, 13, 80 y 104 se almacenaron en el arreglo unidimensional *V* utilizando la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.9 se presenta el estado del arreglo anidado (9.9*a*) y la tabla con $H(K)$ para cada clave (9.9*b*).

9.2.11 Encadenamiento

El método de **encadenamiento** consiste en que cada elemento del arreglo tenga un apuntador a una lista ligada, la cual se irá generando y almacenará los valores que colisionan. Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones que se presenten, se podrán resolver sin inconvenientes.

Como desventajas del método de encadenamiento se menciona el hecho de que ocupa espacio adicional al de la tabla y que exige el manejo de listas ligadas. Además, si las listas crecen demasiado se perderá la facilidad de acceso directo del método *hash*.

La figura 9.10 muestra la estructura de datos necesaria para resolver colisiones por medio del método de encadenamiento.

A continuación se presenta el algoritmo de solución de colisiones por encadenamiento.

FIGURA 9.9
Solución de colisiones con arreglos anidados.
a) Arreglo anidado.
b) Tabla con $H(K)$.

V

1	80			• • •	
2					
3					
4	43	13			
5	54	104			
6	25	35			
7	56				
8					
9					
10					

a)

K	*H*(*K*)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

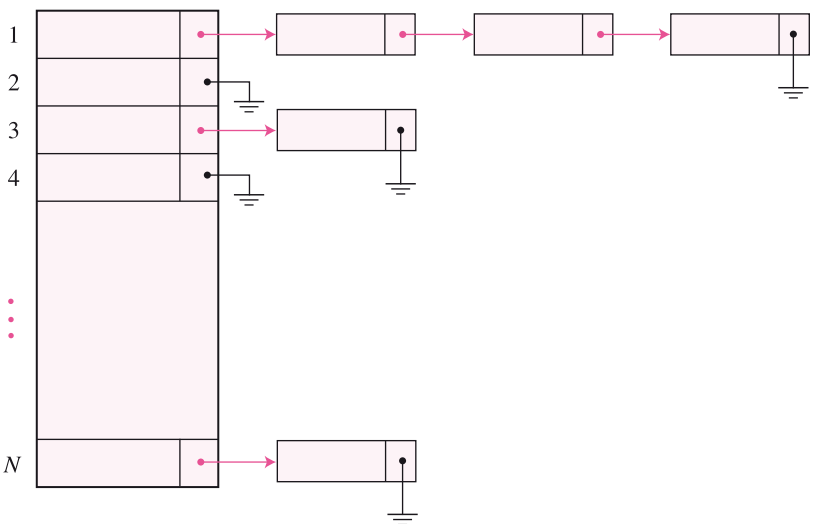


FIGURA 9.10
Solución de colisiones por encadenamiento.

Algoritmo 9.13 Encadenamiento

Encadenamiento (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve las colisiones por medio de encadenamiento en listas ligadas. SIG e INFO son los campos de cada nodo de la lista}
{*D* es una variable de tipo entero. *Q* es una variable de tipo puntero}

1. Hacer $D \leftarrow H(K)$ {Genera dirección}
2. *Si* ((*V*[*D*] ≠ VACÍO) y (*V*[*D*] = *K*))
    entonces
        Escribir "La información está en la posición", *D*
    si no
        Hacer $Q \leftarrow V[D]$.SIG {Apuntador a la lista}
    2.1 Mientras ((*Q* ≠ VACÍO) y (*Q*^.INFO ≠ *K*))
            Hacer $Q \leftarrow Q$^.SIG
    2.2 {Fin del ciclo del paso 2.1}
    2.3 *Si* (*Q* = VACÍO)
            entonces
                Escribir "La información no se encuentra en la lista"
            si no
                Escribir "La información se encuentra en la lista"
    2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}

El funcionamiento de este algoritmo queda más claro con el siguiente ejemplo.

Ejemplo 9.10

Sea *V* un arreglo unidimensional de diez elementos. Los elementos con claves 25, 43, 56, 35, 54, 13, 80 y 104 se almacenaron en el arreglo unidimensional *V* utilizando la siguiente función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.11 se presenta el estado del arreglo con encadenamiento (9.11*a*) y la tabla con $H(K)$ para cada clave (9.11*b*).

Una vez detectada la colisión en una cierta posición del arreglo, se debe recorrer la lista asociada a ella hasta encontrar el elemento buscado o llegar a su final.

En el ejemplo 9.11 se presenta otro caso de solución de colisiones por encadenamiento donde las claves son alfabéticas.

Ejemplo 9.11

Sea *P* un arreglo unidimensional de diez elementos, en el cual se almacenan los datos de algunos pinos mexicanos. Se utiliza como clave el nombre de los pinos para asignar a cada uno de ellos una dirección en el arreglo *P*. Para ello primero se obtendrá un número que resultará de sustituir cada letra por un dígito (del 01 al 27), y a este número se le aplicará la función *hash* (*H*) definida de la siguiente manera:

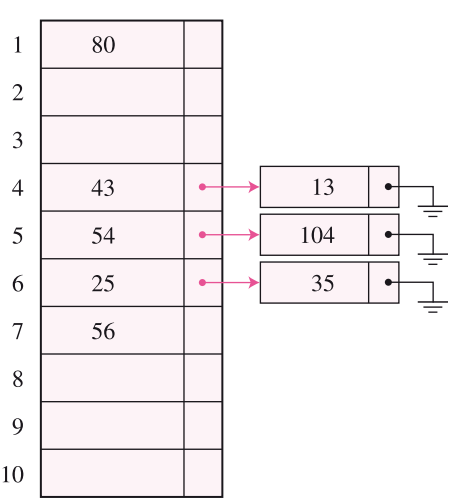
$$H\ (\text{clave}) = (\text{clave} \bmod 10) + 1$$

La tabla 9.10 contiene los nombres de los pinos, el valor numérico asociado (clave) y la dirección en *P* que le corresponde.

Como se puede apreciar en la tabla, ha habido colisiones. Para resolverlas, se aplicará el método de encadenamiento. La estructura resultante se muestra en la figura 9.12.

Cabe destacar que cualquiera que sea el método seleccionado para resolver las colisiones, se debe tener en cuenta en qué estado queda la estructura al insertar y, sobre todo, al eliminar elementos. La eliminación es la operación que más afecta cuando se

FIGURA 9.11
Solución de colisiones por encadenamiento.



a)

K	*H*(*K*)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

TABLA 9.10

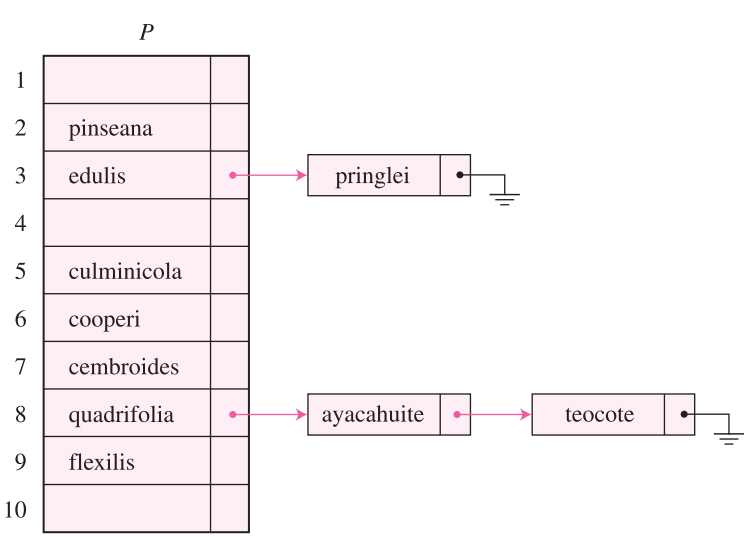
Nombre	Valor numérico	Dirección
Cembroides	96	7
Edulis	72	3
Culminicola	114	5
Quadrifolia	117	8
Pinseana	81	2
Flexilis	98	9
Ayacahuite	97	8
Teocote	87	8
Cooperi	85	6
Pringlei	92	3

tienen colisiones, por lo que se le debe dedicar especial atención para no perder eficiencia en la búsqueda.

Análisis del método por transformación de claves

Para analizar la complejidad de este método es necesario realizar varios cálculos probabilísticos, que no se estudiarán en esta obra. La dificultad del análisis se debe principalmente a que no sólo interviene la función *hash* sino también el método utilizado para resolver las colisiones. Por lo tanto, se debería analizar cada una de las posibles combinaciones que se pudieran presentar.

FIGURA 9.12
Solución de colisiones por encadenamiento.



Sea λ el factor de ocupación de un arreglo, definido como *M*/*N*, donde *M* es el número de elementos en el arreglo y *N* es su tamaño. Según Lipschutz, la probabilidad de llevar a cabo una búsqueda con éxito (*S*) y otra sin éxito (*Z*), quedan determinadas por las siguientes fórmulas:

Fórmulas 9.7

***a*) Búsqueda con éxito**

$$S(\lambda) = \frac{(1 + 1/(1-\lambda))}{2}$$

***b*) Búsqueda sin éxito**

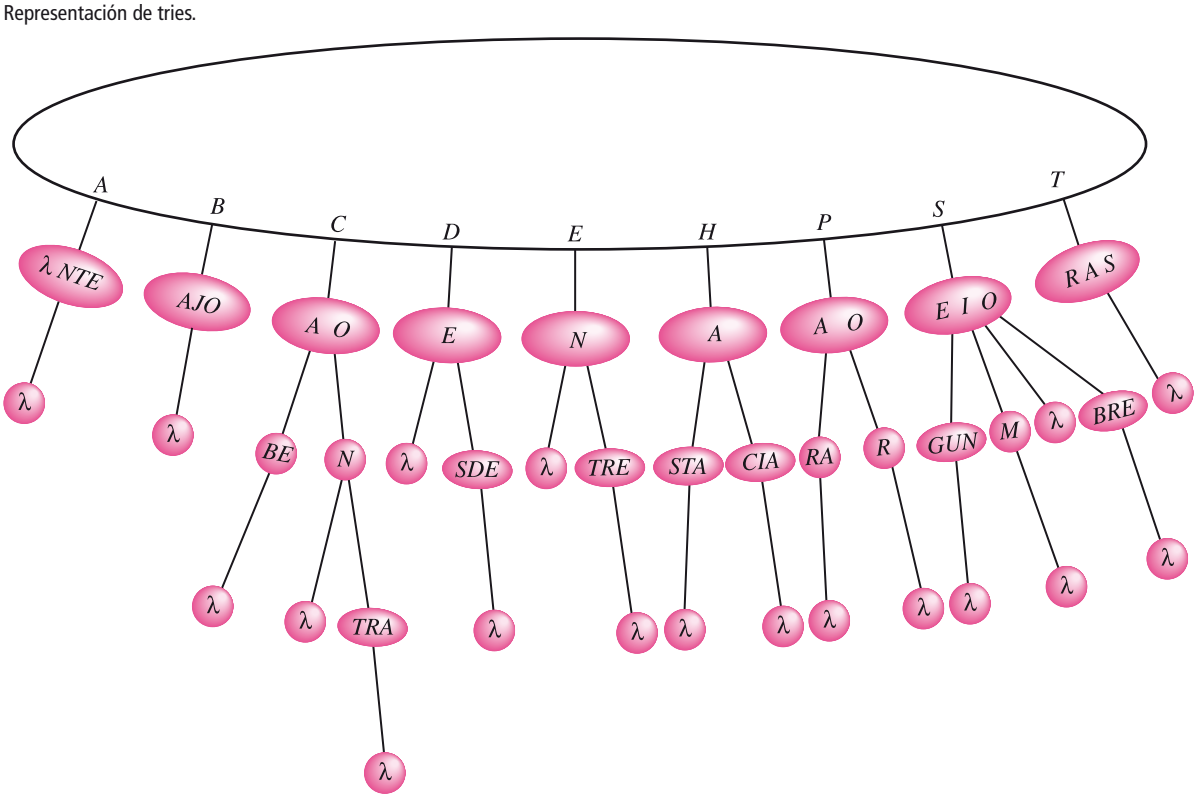
$$Z(\lambda) = \frac{(1 + 1/(1-\lambda)^2)}{2}$$

Cabe aclarar que estas fórmulas son válidas solamente en caso de funciones *hash* con el método lineal de solución de colisiones.

9.2.12 Árboles de búsqueda

En el capítulo 6 se presentaron los árboles como una estructura poderosa y eficiente para almacenar y recuperar información. Debido al dinamismo que caracteriza a los árboles, el beneficio de utilizarlos es mayor cuanto más variable sea el número de datos a tratar.

FIGURA 9.13
Representación de tries.



En esta sección sólo se hablará de la estructura **trie**, que es una variante de la estructura tipo árbol.

Un trie es una estructura similar a un árbol con *N* raíces, con la particularidad de que cada nodo del árbol puede ser nuevamente un trie. En la figura 9.13 se presenta un diagrama correspondiente a un trie que contiene las proposiciones del castellano.

Un trie puede representar una estructura sumamente útil para búsqueda. Las raíces del árbol tienen como objetivo dirigir el camino de búsqueda hacia la meta. La profundidad de una estructura de este tipo depende de la discriminación en la clave de búsqueda que realice el usuario. En la figura 9.13 se puede observar un trie cuya profundidad es variable para cada raíz. De esta forma se localiza la información buscada directamente en el nodo terminal, sin tener que realizar búsqueda secuencial. En la figura 9.14 el lector puede observar un trie con profundidad tres; la discriminación en la clave de búsqueda es igual a 2.

Con el propósito de instrumentar esta estructura en un lenguaje de alto nivel, podemos representar un trie como un bosque. Posteriormente, aplicando las reglas necesarias —analizadas en el capítulo 6—, se debe convertir esta estructura en árbol binario. En la figura 9.15 se muestra el bosque que representa al trie de la figura 9.13.

Finalmente, en la figura 9.16 se muestra al árbol binario que representa al bosque de la figura 9.15.

FIGURA 9.14

Representación de un trie con discriminación 2.

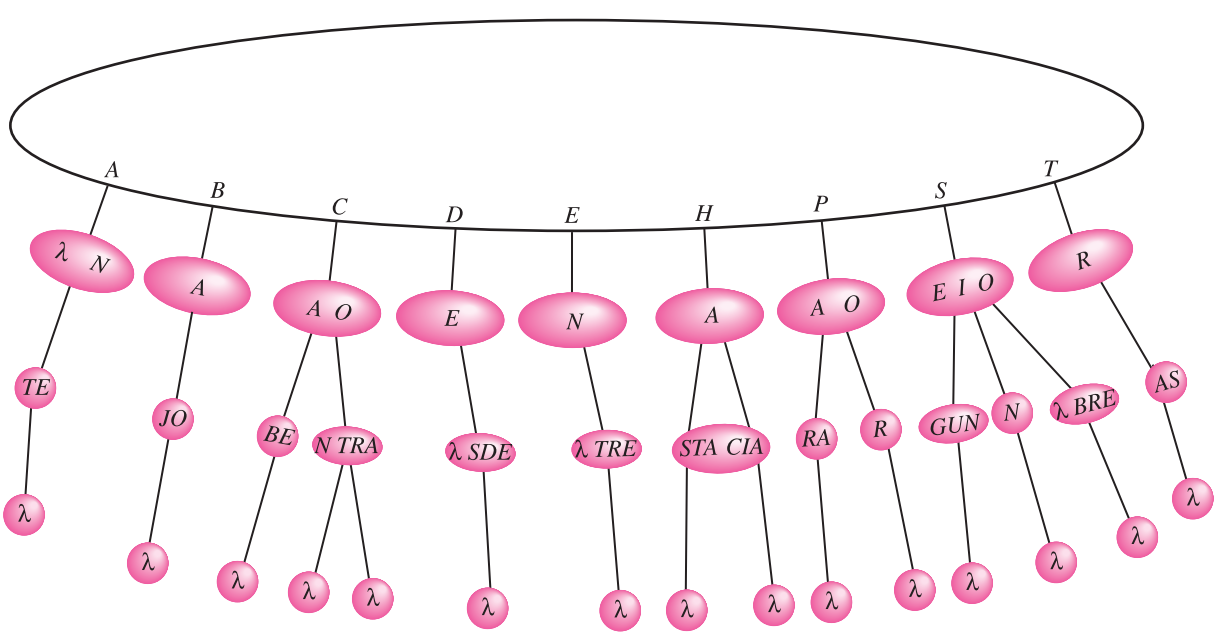
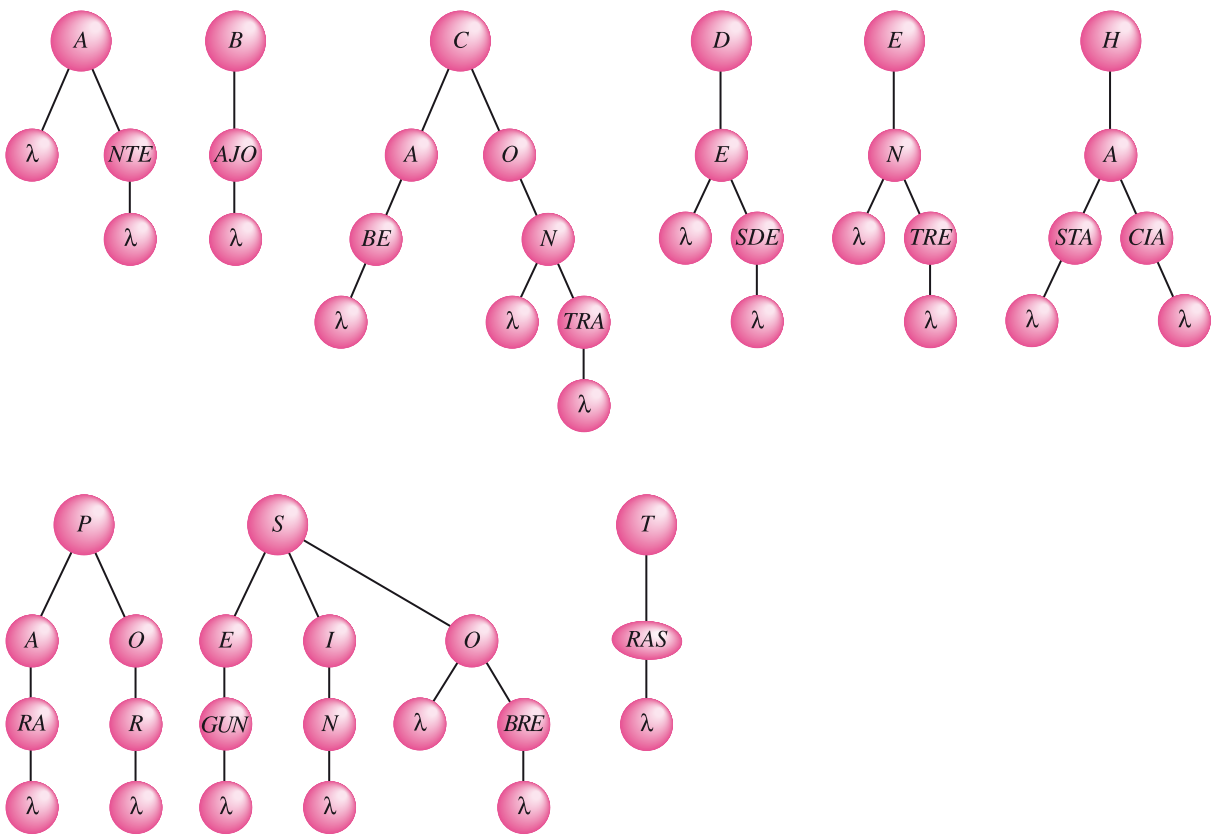


FIGURA 9.15
Representación del trie de la figura 9.13 como bosque.



9.3 BÚSQUEDA EXTERNA

En la sección anterior se estudiaron las técnicas de búsqueda que son aplicables cuando la información reside en la memoria principal de la computadora. En particular, se analizó la operación de búsqueda en estructuras estáticas —arreglos— y dinámicas —listas y árboles— de información. Sin embargo, existen casos en los cuales no se puede manejar toda la información en memoria principal, sino que es necesario trabajar con información almacenada en archivos. Este tipo de búsqueda se denomina **búsqueda externa**.

Los archivos se usan normalmente cuando el volumen de datos es significativo, o cuando la aplicación exige la permanencia de los datos, aun después de que ésta se termine de ejecutar. Como los archivos se encuentran almacenados en dispositivos periféricos —cintas, discos, etc.—, las operaciones de escritura y lectura de datos tienen un alto costo en cuanto a tiempo, por los accesos a estos periféricos. Para disminuir el

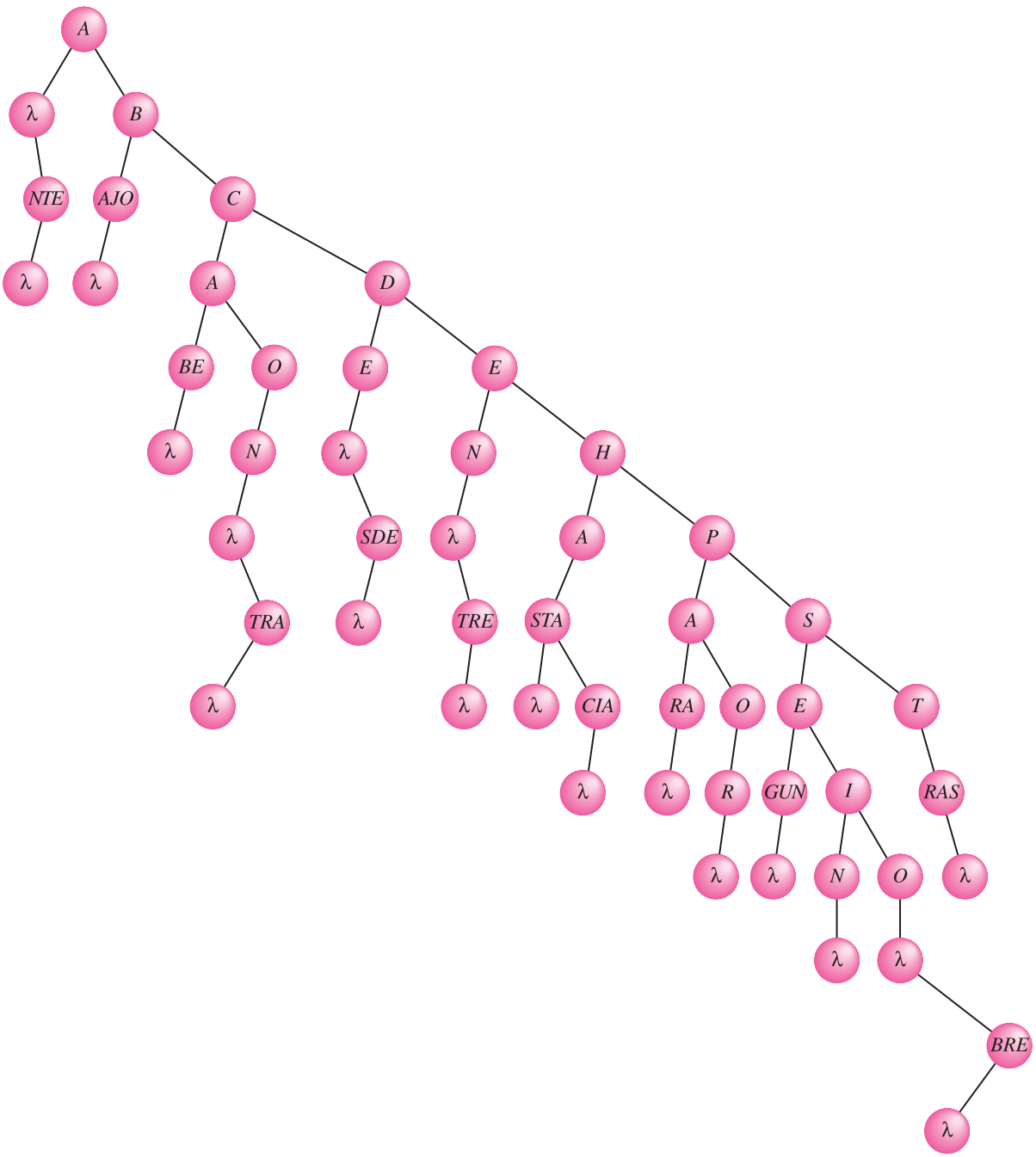


FIGURA 9.16

Representación del bosque de la figura 9.15 como árbol binario.

tiempo de acceso es muy importante optimizar las operaciones de búsqueda, inserción y eliminación en archivos. Una forma de hacerlo es trabajar con archivos ordenados. A continuación se describen algunos de los métodos más utilizados en búsqueda externa.

9.3.1 Búsqueda en archivos secuenciales

Los **archivos secuenciales** son aquellos cuyos componentes o registros ocupan posiciones relativas consecutivas. Todo componente o registro de un archivo tiene generalmente un campo que lo identifica, llamado **campo clave**. Éste se encuentra formado por un conjunto de caracteres o dígitos. Además, ocupa la misma posición relativa en todos los registros de un mismo archivo. Algunos ejemplos de campos clave son el número de cliente —archivo de clientes—, el número de contribuyente —archivo de hacienda—, la matrícula de un alumno —archivo de alumnos—, el número de empleado —archivo de empleados—, etc. Puede suceder que la clave de un registro esté formada por más de un campo. Por ejemplo, en un sistema de inventarios cada pieza se podría identificar por un campo que haga referencia al departamento al cual pertenece, y otro campo para la pieza en sí.

Enseguida se describen algunos métodos de búsqueda en archivos secuenciales.

9.3.2 Búsqueda secuencial

El método de **búsqueda secuencial** consiste en recorrer el archivo comparando la clave buscada con la clave del registro en curso. El recorrido lineal del archivo termina cuando se encuentra el elemento, o cuando se alcanza el final del archivo. Se pueden presentar algunas variantes dentro de este método, dependiendo sobre todo de si el archivo está ordenado o desordenado.

A continuación se detalla el algoritmo de búsqueda lineal en un archivo secuencial desordenado.

Algoritmo 9.14 Archivo_secuencial_desordenado

Archivo_secuencial_desordenado (*FA*, *K*)

{Este algoritmo busca secuencialmente en un archivo desordenado *FA*, un registro con clave *K*}
{BAN es una variable de tipo booleano. *R* es una variable de tipo registro. CLAVE es un campo del registro}

```
1. Abrir el archivo FA para lectura
2. Hacer BAN ← FALSO
3. Mientras ((no sea el fin de archivo de FA) y (BAN = FALSO)) Repetir
        Leer R de FA
   3.1 Si (R.CLAVE = K) entonces
            Escribir "La información se encuentra en el archivo"
```

```
            Hacer BAN ← VERDADERO
   3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Si (BAN = FALSO) entonces
        Escribir "La información no se encuentra en el archivo"
6. {Fin del condicional del paso 5}
```

Este algoritmo es similar al 9.1. En general, tiene las mismas características que el método secuencial en arreglos desordenados.

El algoritmo de búsqueda en archivos ordenados se estudiará considerando, en particular, archivos ordenados en forma creciente.

Algoritmo 9.15 Archivo_secuencial_ordenado

Archivo_secuencial_ordenado (*FA*, *K*)

```
{Este algoritmo busca secuencialmente en un archivo FA ordenado en forma creciente, un
registro con clave K}
{BAN es una variable de tipo booleano. R es una variable de tipo registro. CLAVE es un
campo del registro}

1. Abrir el archivo FA para lectura
2. Hacer BAN ← FALSO
3. Mientras ((no sea el fin de archivo de FA) y (BAN = FALSO)) Repetir
        Leer R de FA
   3.1 Si (R.CLAVE ≥ K) entonces
            Hacer BAN ← VERDADERO
   3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Si (R.CLAVE = K)
        entonces
            Escribir "La información se encuentra en el archivo"
        si no
            Escribir "La información no se encuentra en el archivo"
6. {Fin del condicional del paso 5}
```

La diferencia entre este algoritmo y el anterior consiste en que la búsqueda también se detiene cuando la clave de *R* es mayor que *K*. Esto último se debe a que si el archivo está ordenado, ya no se encontrará el registro con clave *K* entre los registros aún no visitados.

9.3.3 Búsqueda secuencial mediante bloques

La **búsqueda secuencial mediante bloques** consiste en tomar bloques de registros en vez de registros aislados. Un **bloque** es un conjunto de registros. Su tamaño es arbitrario y depende del número de elementos del archivo. Generalmente se define el tamaño del bloque igual a $\sqrt{N}$, donde $\sqrt{N}$ es el número de registros del archivo —la demostración de por qué es $\sqrt{N}$ se presenta más adelante—. El archivo debe estar ordenado. La búsqueda se realiza al comparar la clave en cuestión con el último registro de cada bloque. Si la clave resulta menor, entonces se busca en forma secuencial a través de los registros salteados en el bloque. En caso contrario se continúa con el siguiente bloque. En promedio, el número de comparaciones requeridas para encontrar un valor dado será igual a $\sqrt{N}$.

A continuación se presenta un algoritmo de búsqueda secuencial usando bloques.

Algoritmo 9.16 Archivo_secuencial_bloques

Archivo_secuencial_bloques (*FA*, *N*, *K*)

{Este algoritmo busca secuencialmente en un archivo ordenado *FA* de *N* elementos, un registro con clave *K*}
{*I* y *TB* son variables de tipo entero. BAN es una variable de tipo booleano}

1. Abrir el archivo *FA* para lectura
2. Hacer BAN ← FALSO, *I* ← 1 y *TB* ← Parte Entera (sqrt (*N*)) {Calcula el tamaño del bloque como la raíz cuadrada de *N*}
3. Mientras ((*TB* * *I* ≤ *N*) y (BAN = FALSO)) *Repetir*
Leer *R* de *FA* en la posición *TB* * *I*
3.1 *Si* (*R*.CLAVE ≥ *K*)
entonces
Hacer BAN ← VERDADERO
si no
Hacer *I* ← *I* + 1
3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. *Si* (BAN = VERDADERO)
entonces
5.1 *Si* (*R*.CLAVE = *K*)
entonces
Escribir "La información se encuentra en el archivo"
si no
Realizar búsqueda secuencial en los registros salteados: del registro (*TB* * (*I* – 1) + 1) al registro (*TB* * *I* – 1)
Reposicionar el puntero del archivo, y aplicar el algoritmo 9.15 para ejecutar la búsqueda elemento por elemento
5.2 {Fin del condicional del paso 5.1}
si no {Si *TB* no es múltiplo de *N*, quedaron elementos sin revisar}
Realizar búsqueda secuencial en los registros comprendidos entre (*TB* * (*I* – 1) + 1) y *N*
6. {Fin del condicional del paso 5}

En este algoritmo se lee el último registro de cada bloque, y de la comparación del elemento buscado con él se decide cómo continuar con la búsqueda. El siguiente ejemplo ilustra mejor el funcionamiento de este algoritmo.

Ejemplo 9.12

Sea *FA* un archivo ordenado de 20 registros. Los registros ocupan posiciones consecutivas con direcciones relativas del 1 al 20. Las claves de los registros almacenados en *FA* son:

204, 311, 409, 415, 439, 450, 502, 507, 600, 623, 679, 680, 691, 692, 695, 698, 730, 850, 870, 889.

Dado que se conoce *N*, se calcula el tamaño del bloque de la siguiente manera:

$$TB = \sqrt{20} \cong 4$$

204, 311, 409, **415**, 439, 450, 502, **507**, 600, 623, 679, **680**, 691, 692, 695, **698**, 730, 850, 870, **889**.

La tabla 9.11 presenta el seguimiento del algoritmo 9.16 para $K = 623$.

En la columna *Registro leído* aparece el último registro del bloque, pasos 1, 2 y 3. En el paso 3, cuando se cumple la condición de que R.CLAVE ≥ K, entonces se comienza la búsqueda secuencial a partir del elemento ($TB * (I - 1) + 1$) —elemento 9—, en este caso el 600, hasta que se encuentra el valor deseado —éxito— o hasta el elemento ($TB * I - 1$) —elemento 11—. Observe que en el paso 5 se encuentra el registro buscado.

9.3.4 Búsqueda secuencial con índices

El método de **búsqueda secuencial con índices** trabaja con bloques y con archivos de índices. En el archivo de índices se almacenan las claves que hacen referencia a cada bloque y la dirección de los bloques en el archivo. La búsqueda de un elemento comienza recorriendo el archivo de índices, comparando las claves allí almacenadas con la clave del elemento en cuestión. Una vez que se determina el bloque en el cual se puede encontrar el registro buscado, se continúa la búsqueda ahora recorriendo secuencialmente dicho bloque.

TABLA 9.11
Búsqueda secuencial con bloque

Paso	*I*	Registro leído	Compara	Bandera
1	1	415	415 ≥ 623 ?	*F*
2	2	507	507 ≥ 623 ?	*F*
3	3	680	680 ≥ 623 ?	*V*
4		600	600 = 623 ?	
5		623	623 = 623 ?	

La desventaja de este método es que requiere más espacio de memoria, ya que se trabaja con dos archivos: el principal, en el cual se almacenan los registros, y el de índices. Una forma de acelerar el proceso de búsqueda consiste en mantener en memoria principal el archivo de índices.

En la figura 9.17 se presenta un esquema de un archivo con su correspondiente archivo de índices.

El archivo de índices se recorre secuencialmente hasta encontrar la clave que sea mayor o igual a la clave buscada. Cuando esto último suceda, se tomará la dirección del bloque apuntado por dicha clave y se aplicará búsqueda secuencial (algoritmo 9.15) en dicho bloque.

Determinación del tamaño del bloque

El tamaño del bloque se debe elegir de tal forma que permita reducir el número de comparaciones. Sea N el número de registros en el archivo y TB el tamaño del bloque. La probabilidad de encontrar un registro en un bloque es igual para todos los bloques; por lo tanto, el número medio de bloques examinados será:

$$\sum_{i=1}^{N/TB}\left(i * \frac{1}{(N/TB)}\right) = \frac{N/TB+1}{2} \qquad (1)$$

Donde $1/(N/TB)$ representa la probabilidad de encontrar un registro en un bloque. Considere, además, que todos los registros tienen la misma probabilidad de ser el buscado; por lo tanto, el número medio de registros examinados será:

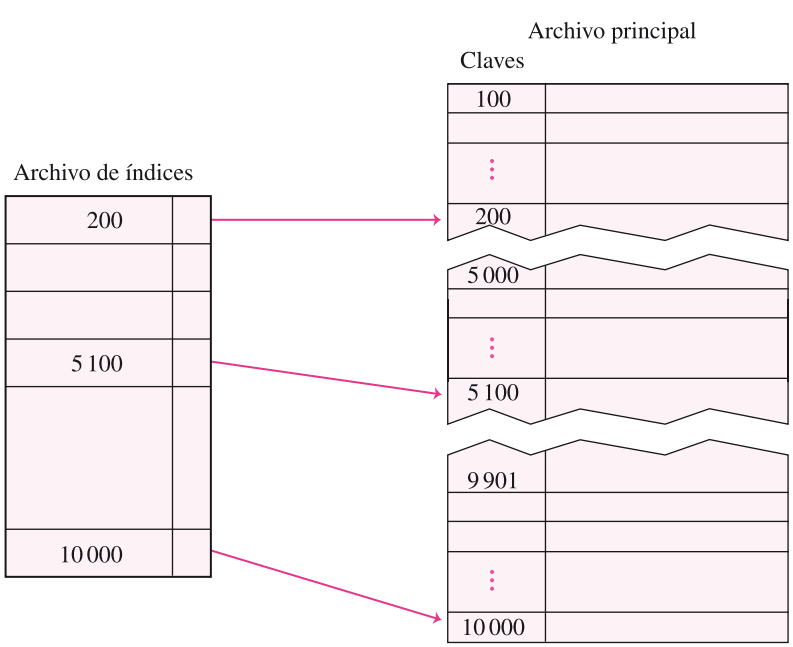


FIGURA 9.17
Búsqueda secuencial con índices.

$$\sum_{i=0}^{TB-1}\left(i * \frac{1}{TB}\right) = \frac{TB-1}{2} \tag{2}$$

Donde $1/TB$ es la probabilidad de que el registro examinado sea el buscado.

Se suman las expresiones 1 y 2 para obtener el número total medio de comparaciones (TC) que se deben hacer para encontrar un elemento en el archivo.

$$TC = \frac{(N / TB)+1}{2} + \frac{TB-1}{2}$$

Operando se obtiene:

$$TC = \frac{N}{2 * TB} + \frac{TB}{2} \tag{3}$$

Al minimizar TC se podrá determinar cuál es el tamaño adecuado para definir los bloques; es decir, el problema se reduce a encontrar un valor tal para TB que minimice el valor de TC.

$$\frac{d(TC)}{d(TB)} = \frac{-N}{2(TB)^2} + \frac{1}{2} \tag{4}$$

Se iguala a cero la expresión 4 y se hacen las operaciones:

$$\frac{-N}{2(TB)^2} + \frac{1}{2} = 0 \qquad \frac{N}{2(TB)^2} = \frac{1}{2} \tag{5}$$

De la expresión 5 se puede afirmar que el valor de TB que minimiza a TC es:

$$TB = \sqrt{N} \qquad \textbf{Fórmula 9.8}$$

Los archivos de índices, por otra parte, se pueden definir a distintos niveles; es decir, se pueden definir índices de índices. Si bien este tipo de organización optimiza el tiempo de búsqueda, tiene el inconveniente de que ocupa mucho espacio de almacenamiento.

9.3.5 Búsqueda binaria

El principio que rige el método de **búsqueda binaria** en la búsqueda externa es el mismo que se explicó en búsqueda binaria interna, sección 9.2.2 de este capítulo. El archivo debe estar ordenado y se debe conocer su número de elementos (N) para aplicar este método. El lector puede desarrollarlo fácilmente, ya que conoce el método de búsqueda binaria en memoria principal —interna—.

Cabe destacar que un gran inconveniente de la búsqueda binaria externa es que requiere accesos a diferentes posiciones del dispositivo periférico en el cual está almacenado el archivo; ello produce un alto costo en tiempo de acceso, que hace muy impráctica esta búsqueda.

9.3.6 Búsqueda por transformación de claves (*hash*)

El método de búsqueda externa por **transformación de claves** tiene básicamente las mismas características que el presentado en la sección 9.2.3. Los archivos normalmente se encuentran organizados en áreas llamadas **cubetas**. Éstas se encuentran formadas por cero, uno o más bloques de registros. Por lo tanto, la función *hash*, aplicada a una clave, dará como resultado un valor que hace referencia a una cubeta en la cual se puede encontrar el registro buscado.

Tal como se mencionó en búsqueda interna, la elección de una adecuada función *hash* y de un método para resolver colisiones es fundamental para lograr mayor eficiencia en la búsqueda.

Antes de presentar algunas funciones *hash* se hará un comentario sobre las colisiones. Los bloques contienen un número fijo de registros. Con respecto a las cubetas, no se establece un límite en cuanto al número de bloques que pueden almacenar. Esta característica permite solucionar, al menos parcialmente, el problema de las colisiones. Sin embargo, si el tamaño de las cubetas crece considerablemente, se perderán las ventajas propias de este método. Es decir, si el número de bloques que se deben recorrer en una cubeta es grande, el tiempo necesario para ello será significativo; por lo tanto, ya no se contará con la ventaja del acceso directo que caracteriza al método por transformación de claves. En la figura 9.18 se presenta una estructura de archivo organizado en cubetas, las que a su vez están formadas por bloques.

Como se muestra en la figura 9.17, cada cubeta puede tener un apuntador a un bloque. Si una cubeta tiene dos o más bloques se establecen ligas entre ellos. Dada la clave de un registro buscado, se aplicará una función *hash*, la cual dará como resultado un número de cubeta. Una vez localizada ésta, habrá que recorrer sus bloques hasta encontrar el registro, o llegar a un bloque con puntero nulo, lo cual indicará que no existen otros bloques.

Es importante elegir una función *hash* que distribuya las claves en forma homogénea a través de las cubetas, de manera que se evite la concentración de numerosas claves

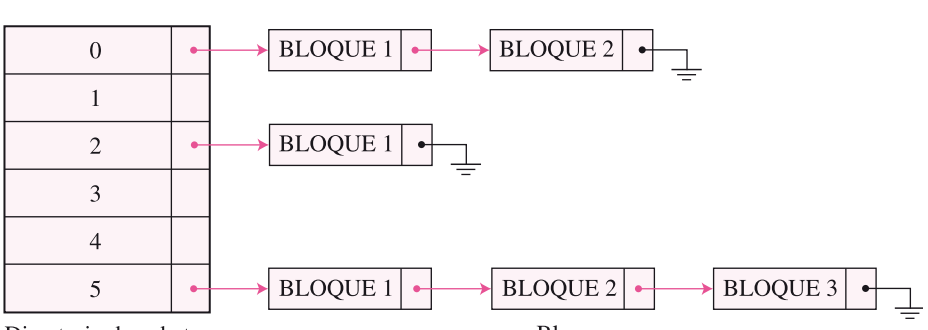


FIGURA 9.18
Archivo organizado con cubetas de bloques.

en una cubeta mientras otras permanecen vacías. A continuación se presentan algunas de las funciones *hash* más comunes.

Funciones *hash*

Una función *hash* se puede definir como una transformación de clave a una dirección. Al aplicar una función *hash* a una clave se obtiene el número de cubeta en la cual se puede encontrar el registro con dicha clave.

La función debe transformar las claves para que la dirección resultante sea un número comprendido entre los posibles valores de las cubetas. Por ejemplo, si se tienen 10 000 cubetas numeradas de 0 a 9 999, las direcciones producidas por la función deben ser valores comprendidos entre 0 y 9 999. Si las claves fueran alfabéticas o alfanuméricas, primero deberán convertirse en numéricas, tratando de no perder información, para luego ser transformadas en una dirección. Es importante que la función distribuya homogéneamente las claves entre los números de cubetas disponibles.

Las funciones módulo, cuadrado, plegamiento y truncamiento presentadas anteriormente para búsqueda interna son válidas también para búsqueda externa. Otra función que se puede utilizar para el cálculo de direcciones es la de conversión de bases, aunque no proporciona mayor homogeneidad en la distribución. De todas, la función módulo es, sin embargo, la que ofrece mayor uniformidad.

Conversiones de bases

La **conversión de bases** consiste en modificar de manera arbitraria la base de la clave obteniendo un número que corresponda a una cubeta. Si el número de dígitos del valor resultante excede el orden de las direcciones, entonces se suprimirán los dígitos más significativos.

Ejemplo 9.13

Supongamos que se tienen 100 cubetas, cada una de ellas referenciada por un número entero comprendido entre 1 y 100. Sea $K = 7\ 259$ la clave del registro que se busca. Se elige el 9 como base a la cual se convierte la clave.

$$H(7\ 259) = \text{dígmensig}\ (7 * 9^3 + 2 * 9^2 + 5 * 9^1 + 9 * 9^0)$$
$$H(7\ 259) = \text{dígmensig}(5\ 319) = 19$$

Se toma entonces como dirección el 19 y los dígitos más significativos, 5 y 3, se desprecian.

9.3.7 Solución de colisiones

Como se mencionó anteriormente cuando se trató búsqueda interna, uno de los aspectos que siempre se deben de considerar en el método por transformación de claves es la solución de colisiones. Cuando dos o más elementos con distintas claves tienen una misma dirección, se origina una colisión.

Para evitar las colisiones se debe elegir un tamaño adecuado de cubetas y de bloques. Con respecto a las cubetas, si se definen muy pequeñas el número de colisiones

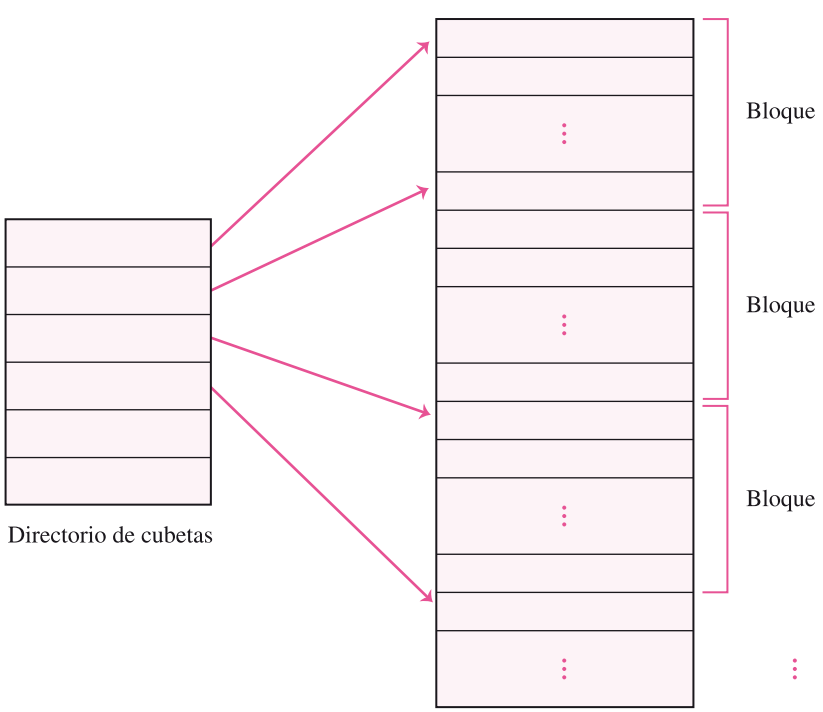
aumenta, mientras que si se definen muy grandes se pierde eficiencia en cuanto a espacio de almacenamiento. Además, si se necesitara copiar una cubeta en memoria principal y ésta fuera muy grande, ocasionaría problemas por falta de espacio. Otro inconveniente que se presenta en el caso de cubetas muy grandes es que se requiere mucho tiempo para recorrerlas.

Con respecto al tamaño de los bloques, es importante considerar la capacidad de éstos para almacenar registros. Un bloque puede almacenar uno, dos o más registros. Normalmente los tamaños de las cubetas y los bloques dependen de las capacidades del equipo con el que se esté trabajando.

Cabe destacar que utilizando una estructura como la de la figura 9.18 no se tendría problemas de colisiones, debido principalmente a que por más que la cubeta esté ocupada, es posible seguir enlazando tantos bloques como fueran necesarios. Este esquema de solución se corresponde con el presentado en búsqueda interna, bajo el nombre de encadenamiento. Sin embargo, no siempre es posible definir una estructura de este tipo. Considere, por ejemplo, un archivo organizado en cubetas como el que se muestra en la figura 9.19.

En este archivo cada cubeta tiene un bloque y, por lo tanto, una capacidad máxima determinada por el tamaño del bloque asociado con ella. Una vez que se satura la capacidad de la cubeta, cualquier registro asignado a ella producirá una colisión. A continuación se analizarán dos maneras diferentes de enfrentar esta situación.

FIGURA 9.19
Solución de colisiones.

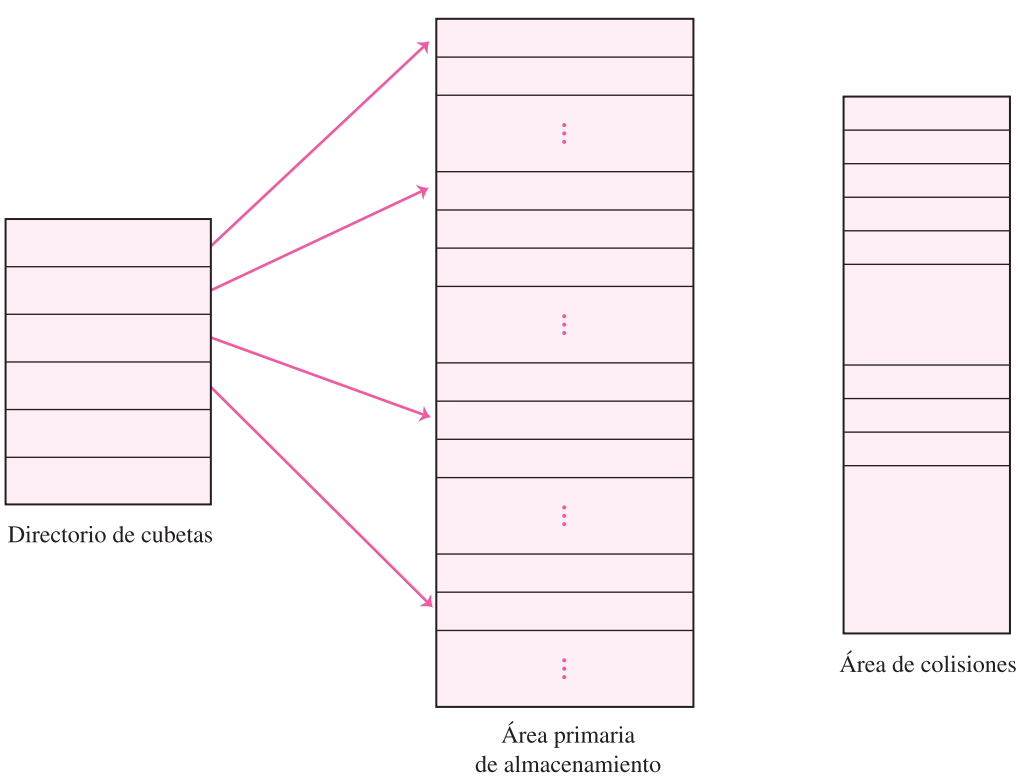


Uso de áreas independientes para colisiones

El **uso de áreas independientes para colisiones** consiste en definir áreas separadas —secundarias— de las áreas primarias de almacenamiento, en las que se almacenarán todos los registros que hayan colisionado. El área de colisiones puede estar organizada de diferentes maneras. Una alternativa consiste en tener el área común a todas las cubetas. En consecuencia, si se produce una colisión habrá que buscar a lo largo del área secundaria hasta encontrar el elemento deseado, según la figura 9.20.

Otra forma de organizar el área de colisiones consiste en dividirla en bloques, asociando cada uno de ellos a uno del área primaria. Esta alternativa optimiza el tiempo de búsqueda en el área de colisiones, pero tiene el inconveniente de que estos bloques podrían, a su vez, saturarse, ocasionando nuevamente colisiones. El esquema correspondiente a esta estructura se muestra en la figura 9.21.

FIGURA 9.20
Solución de colisiones mediante un área común de colisiones.



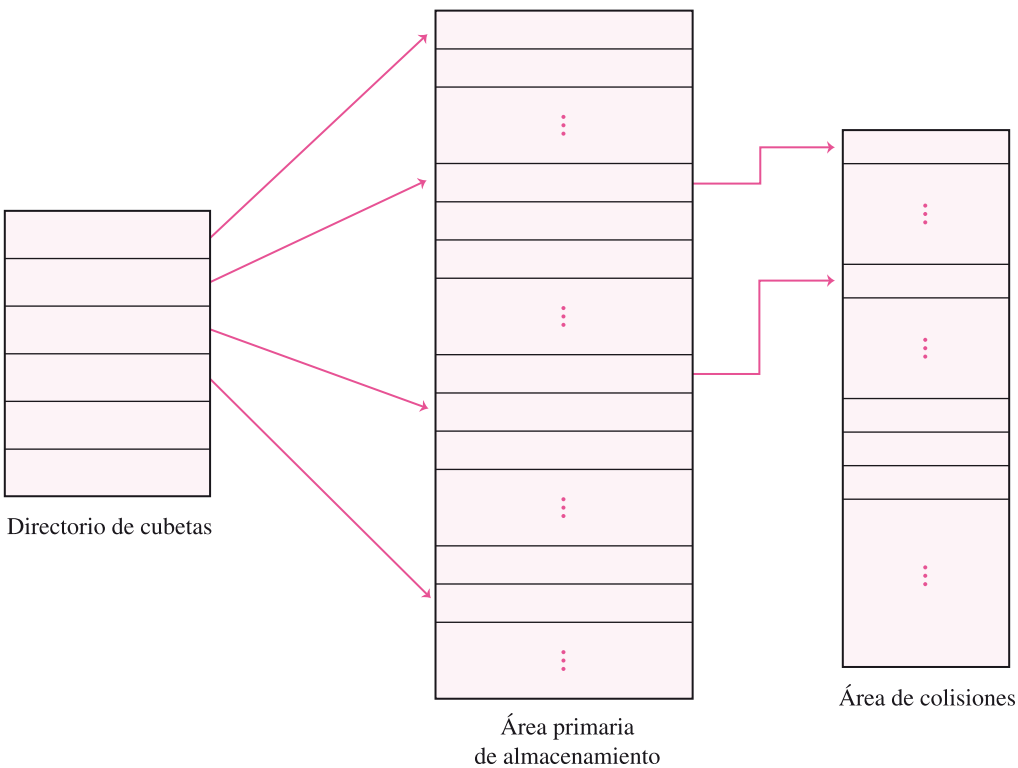


FIGURA 9.21
Solución de colisiones mediante un área de colisiones organizada en bloques.

Uso de áreas de colisiones entre los bloques de almacenamiento primario

El **uso de áreas de colisiones entre los bloques de almacenamiento primario** consiste en definir áreas de colisiones entre los bloques de almacenamiento primario. Este método es similar al presentado en búsqueda interna bajo el nombre de reasignación. Una vez detectada una colisión en un bloque se debe buscar en el área de colisiones inmediata a dicho bloque. Si el elemento no se encuentra y el área de colisiones está llena, se continuará la búsqueda a través de las otras áreas de colisiones. El proceso termina cuando el elemento se encuentra o bien cuando existen espacios vacíos en un bloque —el elemento buscado no se encuentra en el archivo—. El esquema correspondiente a este esquema se muestra en la figura 9.22.

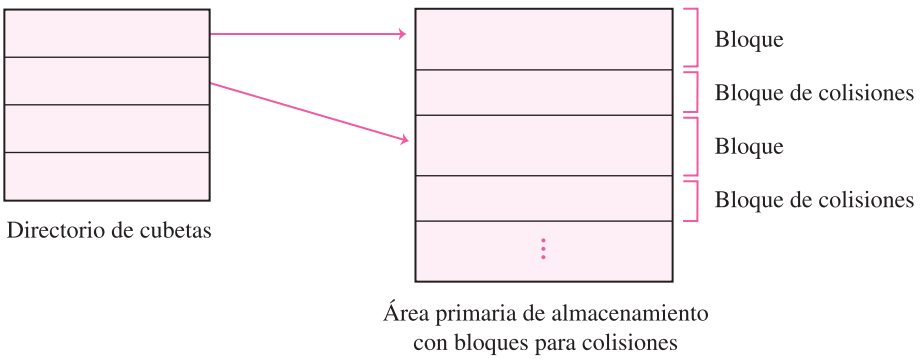


FIGURA 9.22
Solución de colisiones mediante bloques para colisiones entre los bloques primarios.

9.3.8 *Hashing* dinámico: búsqueda dinámica por transformación de claves

La principal característica del ***hashing* dinámico** es su dinamismo para variar el número de cubetas en función de su densidad de ocupación. Se comienza a trabajar con un número determinado de cubetas, y a medida que éstas se van llenando se asignan nuevas cubetas al archivo. Existen básicamente dos formas de trabajar con el *hashing* dinámico:

- Por medio de expansiones totales
- Por medio de expansiones parciales

9.3.9 Método de las expansiones totales

El método de **expansiones totales** es probablemente el más utilizado. Consiste en duplicar el número de cubetas en la medida en que éstas superan la densidad de ocupación previamente establecida. Así, por ejemplo, si el número inicial de cubetas es N y se hace una expansión total, el valor resultante —nuevo número de cubetas— será $2N$. Si se hace una segunda expansión total, se tendrá $4N$, y así sucesivamente.

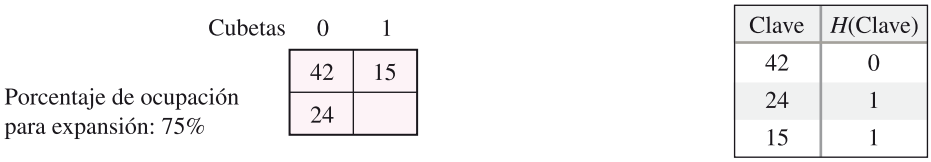
El dinamismo de este método también se da en sentido contrario; es decir, que a medida que la densidad de ocupación de las cubetas disminuye, se reduce el número de éstas. Así, se gana flexibilidad en cuanto a que se pueden incrementar los espacios de almacenamiento, pero también se pueden reducir si la demanda de espacio así lo indica.

Ejemplo 9.14

Supongamos que se tiene un archivo organizado en dos cubetas ($N = 2$), y se ha fijado una densidad de ocupación de 80%. La **densidad de ocupación** se calcula como el cociente entre el número de registros ocupados y el de registros disponibles. Cada cubeta tiene dos registros, y la función *hash* que transforma claves en direcciones se define de la siguiente manera:

$$H\,(\text{clave}) = \text{clave MOD } \textit{Número de cubetas}$$

FIGURA 9.23
Hash dinámico ($N = 2$): expansión total.



Clave	*H*(Clave)
42	0
24	1
15	1

Los valores 42, 24, 15 y 53 son las claves de los registros que se desea almacenar. Inicialmente el archivo está vacío. En la figura 9.23 se presenta un esquema de cómo quedan las cubetas, después de insertar las tres primeras claves.

Cuando se quiere insertar la clave 53, se supera la densidad de ocupación establecida, ya que se alcanzaría 100% de llenado. Por lo tanto, se deben expandir y reasignar los registros considerando ahora que el número de cubetas es igual a 2**N*, figura 9.24.

Supongamos ahora que se desea incorporar los registros con claves 21, 12, 14, 18, 49, 128, 22, 23 y 67 en este orden. El resultado, después de insertar las dos primeras claves, se puede observar en la figura 9.25.

Cuando se inserta el registro con clave 14, la densidad de ocupación supera el 80% fijado. Se vuelven, entonces, a expandir y a reasignar los registros almacenados (figura 9.26*a*), y luego se continúa con la inserción del resto de los elementos. La figura 9.26*b* presenta el estado de las cubetas luego de realizar todas las inserciones, excepto la última.

Cuando se inserta la última clave, 67, se supera nuevamente la densidad de ocupación y hay que volver a expandir las cubetas. Por lo tanto, ahora *N* será igual a 16 (figura 9.27).

Es importante señalar que en este método también se pueden producir colisiones, las cuales podrían tratarse según alguno de los esquemas propuestos anteriormente. Por ejemplo, si en el caso anterior (figura 9.26) luego de insertar los registros con claves 24 y 128 se tratara de agregar el registro con clave 192, se produciría una colisión, ya que la cubeta 0 está llena.

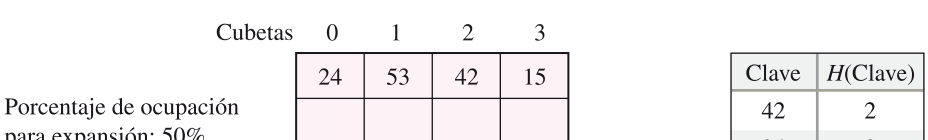
Ejemplo 9.15

Dado un archivo organizado en dos cubetas ($N = 2$), donde cada una de ellas tiene tres registros, se quiere almacenar las siguientes claves:

115, 96, 48, 79, 35, 26, 57, 81, 70, 64, 107, 45, 62, 98, 33, 28 y 38.

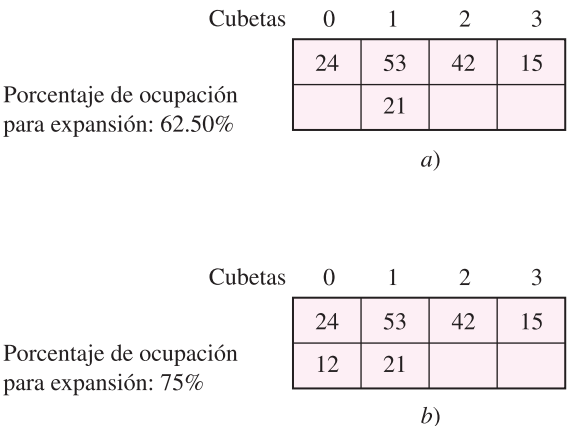
Se ha establecido una densidad de ocupación mayor a 82% para expansión y menor a 125% para reducción. Es importante remarcar que el porcentaje de ocupación, para el

FIGURA 9.24
Hash dinámico ($N = 4$): expansión total.



Clave	*H*(Clave)
42	2
24	0
15	3
53	1

FIGURA 9.25
Hash dinámico ($N = 4$): expansión total. *a*) Luego de insertar 21. *b*) Luego de insertar 12.



Clave	*H*(Clave)
21	1
12	2

caso de reducción, se calcula como el cociente entre el número de registros ocupados y el número de cubetas.

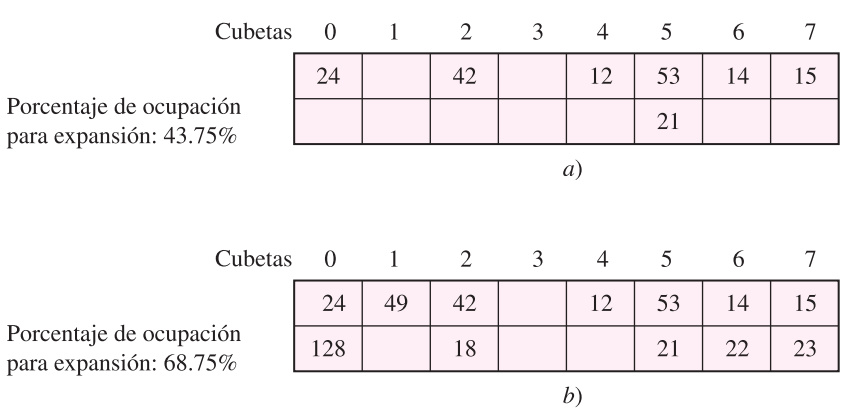
A continuación se presenta la función *hash* que se utiliza:

$$H \text{ (clave)} = \text{clave MOD } \textit{Número de cubetas}$$

Las claves se almacenan en el orden en que se dan. La representación final se puede observar en la figura 9.28.

En el siguiente ejemplo se aclara el concepto de reducción del número de cubetas en el método dinámico por transformación de claves, con expansiones totales.

FIGURA 9.26
Hash dinámico ($N = 4$): expansión total. *a*) Luego de insertar 21. *b*) Luego de insertar 12.



Clave	*H*(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

Cubetas	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	128	49	18	67		53	22	23	24		42		12		14	15
						21										

Porcentaje de ocupación para reducción: 81.25%

Porcentaje de ocupación para expansión: 40.62%

Clave	*H*(Clave)
42	10
24	8
15	15
53	5
21	5
12	12
14	14
18	2
49	1
128	0
22	6
23	7
67	3

FIGURA 9.27
Hash dinámico ($N = 16$): expansión total.

Ejemplo 9.16

Supongamos que se tiene el archivo en el estado que muestra la figura 9.27. Se desean eliminar ahora los registros con claves:

53, 18, 128, 23, 14, 49 y 22.

Al eliminar el registro con clave 53, la densidad de ocupación disminuye de tal manera que permite reducir el número de cubetas ($N/2$). Luego de la reducción y de la reasignación de registros, las cubetas quedan como se muestra en la figura 9.29.

Una vez eliminados los otros registros, la densidad de ocupación permite reducir nuevamente el número de cubetas. En la figura 9.30 se presenta su estado luego de la reducción de N y de la reasignación de los registros.

Ejemplo 9.17

Dado el archivo de la figura 9.28 y las especificaciones dadas en el ejemplo 9.15, elimine las siguientes claves:

FIGURA 9.28
Hash dinámico ($N = 8$): expansión total.

Cubetas	0	1	2	3	4	5	6	7
Porcentaje de ocupación para expansión: 70.83%	96	57	26	115	28	45	70	79
	48	81	98	35			62	
Porcentaje de ocupación para reducción: 212.50%	64	33		107			38	

Clave	*H*(Clave)
42	2
24	0
15	7
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7
67	3

Cubetas 0 1 2 3 4 5 6 7

Porcentaje de ocupación para expansión: 150%	24	49	42	67	12	21	14	15
	128		18				22	23

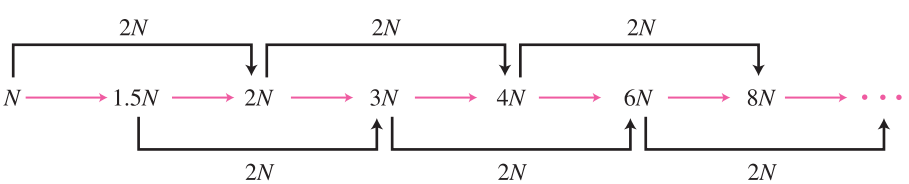
FIGURA 9.29
Hash dinámico ($N = 8$): reducción.

48, 35, 81, 70, 45, 33, 38 y 115

y verifique que el esquema final, luego de realizar las eliminaciones, quede igual al de la figura 9.31.

9.3.10 Método de las expansiones parciales

El método de las **expansiones parciales** consiste en incrementar en 50% el número de cubetas, haciendo de esta forma que dos expansiones parciales equivalgan a una total. Así, por ejemplo, si el número inicial de cubetas es N, y se hace una expansión parcial, el valor resultante será $1.5N$. Si se hacen otras expansiones parciales se tendrá $2N$, luego $3N$, y así sucesivamente.



A continuación se presenta un ejemplo de *hash* dinámico con expansiones parciales.

Clave	*H*(Clave)
42	2
24	0
15	3
21	1
12	0
67	3

Cubetas 0 1 2 3

Porcentaje de ocupación para expansión: 150%	24	21	42	15
	12			67

FIGURA 9.30
Hash dinámico ($N = 4$): reducción.

FIGURA 9.31
Hash dinámico ($N = 4$): reducción.

Cubetas 0 1 2 3

Porcentaje de ocupación para expansión: 75%

96	57	26	
64		62	79
28		98	107

Porcentaje de ocupación para reducción: 225%

Ejemplo 9.18

Retome el ejemplo 9.14. Supongamos que hasta el momento se han almacenado los registros con claves 42, 24 y 15. Cuando se quiere insertar el registro con clave 53, el número de registros supera el máximo permitido ya que la densidad de ocupación supera 80%; por tal razón se realiza una expansión parcial. La figura 9.32 muestra el estado de las cubetas luego de expandir y reasignar los registros.

Observe que en este caso el valor de N no fue muy adecuado para distribuir uniformemente los registros a través de las cubetas. En la cubeta 0 se tiene una colisión, mientras que la cubeta 1 permanece vacía.

Supongamos ahora que se desea incorporar los registros con claves

21, 12, 14, 18, 49, 128, 22 y 23.

Al insertar el registro con clave 21 se supera la densidad de ocupación, por lo que se deben expandir nuevamente las cubetas y reasignar los registros. Se inserta a continuación el registro con clave 12, como se ve en la figura 9.33.

Al insertar el registro con clave 14, otra vez se supera el porcentaje de ocupación permitido. Se vuelven a expandir las cubetas y a reasignar los registros. El resultado final, luego de insertar todas las claves, se muestra en la figura 9.34.

Ejemplo 9.19

Dado un archivo organizado en dos cubetas ($N = 2$), donde cada cubeta tiene tres registros, se quiere almacenar las siguientes claves:

115, 96, 49, 79, 35, 27, 57, 89, 70, 64, 107, 45, 67, 98, 33, 28, 38,
104, 42 y 15.

Para este ejemplo se ha establecido una densidad de ocupación mayor a 82% para expansión y menor a 125% para reducción. A continuación se presenta la función *hash* que se utiliza:

$$H\ (\text{clave}) = \text{clave MOD } \textit{Número de cubetas}$$

FIGURA 9.32
Hash dinámico ($N = 4$): reducción.

Cubetas 0 1 2

Porcentaje de ocupación para expansión: 66.66%

42		53
24		

↓

15

Clave	*H*(Clave)
42	0
24	0
15	0
53	2

FIGURA 9.33
Hash dinámico ($N = 4$): expansión parcial.

Cubetas 0 1 2 3

24	53	42	15
12	21		

Porcentaje de ocupación para expansión: 75%

Clave	H(Clave)
42	2
24	0
15	3
53	1
21	1
12	0

Observe si la estructura que obtiene es igual a la que se presenta en la figura 9.35.

A continuación se presenta un ejemplo para ilustrar la reducción del número de cubetas en el método dinámico por transformación de claves, con expansiones parciales.

Ejemplo 9.20

Supongamos que se tiene un archivo en el estado que muestra la figura 9.34*b*. Elimine los registros con claves:

53, 18 y 128

y verifique si las cubetas y registros quedan igual a la gráfica que se muestra en la figura 9.36.

Ejemplo 9.21

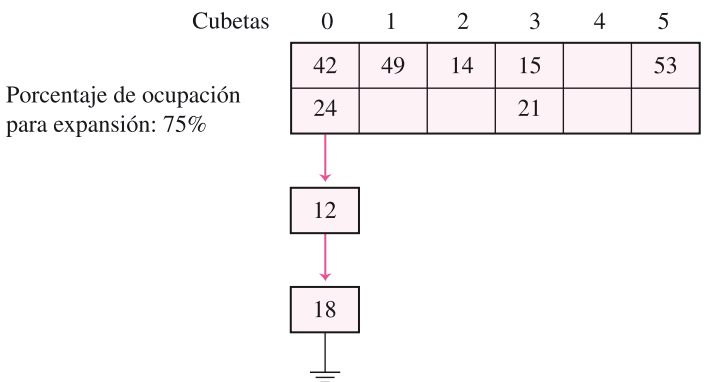
Dado el archivo de la figura 9.35 y las especificaciones dadas en el ejemplo 9.19, elimine las siguientes claves:

67, 104, 15, 45, 33, 79, 70 y 107.

Verifique si el esquema final que obtiene es igual al de la figura 9.37.

Finalmente, es importante señalar que el tamaño de las cubetas se debe establecer de acuerdo con el problema que esté intentando resolver. En los ejemplos presentados se han considerado inicialmente dos registros por cubeta. Sin embargo, este número es para que el lector observe el funcionamiento de los métodos al realizar expansiones y reducciones. Si el número de registros que utilizáramos fuera grande, entonces habría que ingresar gran cantidad de números para observar la expansión de cubetas.

Indudablemente, en la práctica se debe considerar un número mucho más grande de registros por cubeta. El número dependerá principalmente del tamaño de cada registro, de tal forma que una cubeta se pueda cargar en la memoria principal. En aplicaciones grandes, el número de registros por cubeta podría variar de 250 a 500. Si el número de registros por cubeta es pequeño y en forma continua se realizan inserciones y eliminaciones, entonces podría ocurrir que frecuentemente se deban realizar expansiones o reducciones, con la consabida pérdida de tiempo y alto costo, por la reasignación de los registros. Es el usuario quien debe definir entonces el número de registros por cubeta dependiendo del problema y de las actualizaciones que se realicen.



a)

Clave	*H*(Clave)
42	0
24	0
15	3
53	5
21	3
12	0
14	2
18	0
49	1

Cubetas 0 1 2 3 4 5 6 7

24	49	42		12	53	14	15
128		18			21	22	23

Porcentaje de ocupación para expansión: 75%

b)

Clave	*H*(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

FIGURA 9.34
Hash dinámico: expansión parcial. *a*) Luego de expandir e insertar los registros con claves 14, 18 y 49. *b*) Luego de expandir e insertar los registros con claves 128, 22 y 23.

9.3.11 Listas invertidas

Las **listas invertidas** trabajan sobre algunos de los atributos —campos— de los registros. Los atributos pueden estar o no invertidos; es decir, pueden ser o no campos clave. Los atributos invertidos generan listas ordenadas de registros, lo cual facilita las búsquedas que se hagan en ellas. Los atributos no invertidos generan el universo, o sea para encontrar un determinado elemento —registro—, se deberá realizar una búsqueda secuencial.

Porcentaje de ocupación para expansión: 55.55%

Porcentaje de ocupación para reducción: 166.66%

Cubetas	0	1	2	3	4	5	6	7	8	9	10	11
	96	49	98	27	64	89	42	115	104	57	70	35
			38	15	28			79		45		107
								67		33		

FIGURA 9.35
Hash dinámico ($N = 2$): expansión parcial.

Las listas invertidas son muy recomendables cuando se trabaja sobre combinaciones de campos clave. Cuando se requiere una combinación de atributos en la búsqueda, este método resulta muy conveniente, ya que con una secuencia óptima de operadores AND y OR la búsqueda se puede llevar a cabo de forma eficiente.

La desventaja del método es que requiere de una estructura muy complicada para operar. Básicamente trabaja sobre árboles B^+ con prefijo. Analicemos a continuación un ejemplo.

Ejemplo 9.22

Supongamos que se tiene un archivo en el cual cada registro almacena la siguiente información:

Nombre	Profesión	Edad

Se tienen los datos de seis personas:

Juan	matemático	32
Daniel	físico	40
José	matemático	25

FIGURA 9.36
Hash dinámico ($N = 6$): reducción.

Porcentaje de ocupación para expansión: 150%

Cubetas	0	1	2	3	4	5
	42	49	14	15	22	53
	24			21		
	12					

Clave	H(Clave)
42	0
24	0
15	3
21	3
12	0
14	2
49	1
22	4
23	5

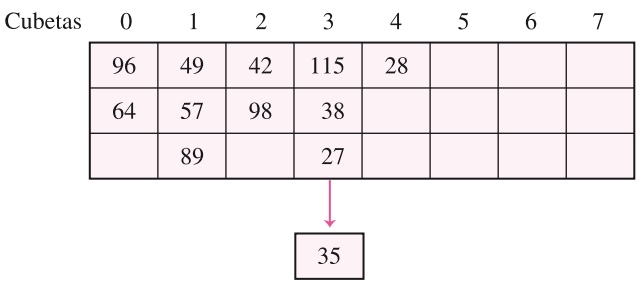


FIGURA 9.37
Hash dinámico ($N = 8$): reducción.

Porcentaje de ocupación para reducción: 150%

Pascual	ingeniero	38
Miguel	ingeniero	43
Felipe	abogado	35

Considerando que los atributos *profesión* y *edad* están invertidos, a continuación se presentan algunas operaciones de búsqueda con sus correspondientes resultados, para que el lector observe el funcionamiento del método.

a) Lista de personas por *profesión*.

matemáticos	{Juan, José}
físicos	{Daniel}
ingenieros	{Pascual, Miguel}
abogados	{Felipe}

b) Lista de todas las personas con *profesión* matemático o físico, y con más de 25 años de *edad*.

(((profesión = matemático) OR (profesión = físico)) AND (edad > 25))

La lista formada según el atributo *profesión* es:

{Juan, José, Daniel}

Sobre esta lista se aplicará la segunda condición planteada en la búsqueda, de lo que resulta:

{Juan, Daniel}

c) Lista de todos los ingenieros menores de 50 años y mayores de 40.

(profesión = ingeniero) AND ((edad < 50) AND (edad > 40))

La lista formada según el atributo *profesión* es:

{Pascual, Miguel}

A partir de esta lista, se buscarán los registros que cumplan con las condiciones impuestas sobre el atributo *edad*. La lista resultante será:

{Miguel}

Considerando que solamente el atributo *profesión* está invertido, se presentan algunas operaciones de búsqueda con sus correspondientes resultados.

a) Lista de todas las personas con *profesión* matemático o físico, y con más de 25 años de edad.

((profesión = matemático) OR (profesión = físico)) y búsqueda secuencial en la lista de los registros marcados para localizar aquellos con edad > 25.

{Juan, José, Daniel} y sobre esta lista una búsqueda secuencial para encontrar a los individuos mayores de 25 años.

b) Lista de todos los ingenieros menores de 50 años y mayores de 40.

(profesión = ingeniero) y búsqueda secuencial en la lista de los registros marcados para localizar aquellos con edad > 40 y edad < 50.

{Pascual, Miguel} y búsqueda secuencial sobre esta lista para encontrar a los individuos menores de 50 y mayores de 40.

c) Lista de todos los abogados mayores de 40 años.

(profesión = abogado) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos con edad > 40.

{Felipe} y búsqueda secuencial sobre esta lista para encontrar a los individuos mayores de 40 años.

En este caso la solución es la lista vacía. No hay ningún registro que tenga los atributos pedidos.

Ejemplo 9.23

La Dirección General de Reclusorios ha decidido crear una base de datos con información sobre sus presos. El esquema que se considera es el siguiente:

nombre_reo	clave_reo	edad	escolaridad	cod_delito	nacionalidad

La escolaridad está codificada como:

1. Analfabeto
2. Primaria
3. Secundaria

4. Preparatoria
5. Universidad
6. Posgrado

Los códigos de delito (cod_delito) están codificados como:

1. Delito contra la salud
2. Robo con arma de fuego
3. Acoso sexual
4. Otros

Considerando que los atributos *escolaridad* y *cod_delito* están invertidos, se presentan algunas operaciones de búsqueda con sus correspondientes resultados:

a) Los reclusos analfabetos con menos de 20 años de edad.

(escolaridad = 1) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos con edad < 20.

b) Los reclusos con posgrado, cuya edad está comprendida entre 20 y 50 años, y que cometieron el delito calificado como acoso sexual.

((escolaridad = 6) AND (cod_delito = 3)) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos cuya edad está comprendida entre 20 y 50 años de edad.

c) Los reos estadounidenses.

(búsqueda secuencial en todo el archivo para localizar a los reos de nacionalidad estadounidense.)

d) Los reos que cometieron robo con arma de fuego, menores de 22 años, o los que cometieron delito contra la salud, menores de 30 años.

((cod_delito = 2) y (búsqueda secuencial en la lista de los registros marcados para localizar edad < 22)) OR ((cod_delito = 1) y (búsqueda secuencial en la lista de los registros marcados para localizar edad < 30)).

Se ha mencionado que las listas generadas por atributos invertidos están ordenadas; por lo tanto, el tiempo de procesamiento está determinado por la lista de mayor tamaño. Una secuencia adecuada de operadores AND y OR puede ayudar a disminuir el tiempo de procesamiento. Analicemos el siguiente ejemplo.

Ejemplo 9.24

Supongamos que se tienen las listas $L1$, $L2$ y $L3$ de 1 000, 5 y 100 elementos, respectivamente. Si se necesitara unir las tres listas, el orden en el cual se hiciera la unión sería determinante en cuanto al número total de elementos con los cuales se trabaja.

1. $(L1 \cup L2) \cup L3 = (1\,000 + 5) \cup L3$
$= 1\,005 + (1\,005 + 100) = 2\,110$

2. $(L1 \cup L3) \cup L2 = (1\,000 + 100) \cup L2$
$= 1\,100 + (1\,100 + 5) = 2\,205$

3. $(L2 \cup L3) \cup L1 = (5 + 100) \cup L1$
$= 105 + (105 + 1\,000) = 1\,210$

Es fácil observar que la mejor secuencia es la tercera y el resultado es 1 210; y que la peor secuencia es la segunda y el resultado es 2 205.

Ejemplo 9.25

Sean A, B, C y D listas de 100, 300, 250 y 80 elementos, respectivamente. Si se necesitara su unión, algunas de las distintas secuencias que se tendrían son:

1. $((A \cup B) \cup C) \cup D = ((100 + 300) \cup C) \cup D$
$= (400 + (400 + 250)) \cup D$
$= (1\,050 + (650 + 80))$
$= 1\,780$

2. $((B \cup C) \cup A) \cup D = ((300 + 250) \cup A) \cup B$
$= (550 + (550 + 100)) \cup B$
$= (1\,200 + (650 + 80))$
$= 1\,930$

3. $((A \cup D) \cup B) \cup C = ((100 + 80) \cup B) \cup C$
$= (180 + (180 + 300)) \cup C$
$= (660 + (480 + 250))$
$= 1\,390$

4. $((A \cup D) \cup C) \cup B) = ((100 + 80) \cup C) \cup B$
$= (180 + (180 + 250)) \cup B$
$= (610 + 730)$
$= 1\,340$

Con los ejemplos queda demostrado cómo influye el tamaño de las listas en el número total de elementos a procesar. Es posible concluir, entonces, que resulta mucho más eficiente dejar las listas de mayor tamaño para unirlas al final.

9.3.12 Multilistas

El método de búsqueda **multilistas** permite acceder a la información que se encuentra ordenada utilizando campos clave. A un registro se puede llegar por diferentes caminos. Cada camino se establece en función del campo clave sobre el cual se haga la búsqueda.

La forma más eficiente de representar multilistas es utilizando listas. A continuación se presenta un ejemplo de este método.

Ejemplo 9.26

Supongamos que se tiene un archivo en el cual cada registro almacena la siguiente información:

Nombre	Profesión	Categoría
Juan	matemático	1
Daniel	físico	2
José	matemático	2
Pascual	ingeniero	3
Miguel	ingeniero	1
Felipe	abogado	2

La figura 9.38 representa las multilistas correspondientes a los datos dados. En este caso, la información de cada individuo puede ser accesada por medio de su profesión y de su categoría, que son justamente los atributos que permiten realizar búsqueda directa en el archivo. Como se puede observar en la siguiente figura, se tiene una lista por profesión y otra por categoría.

En general, las multilistas son recomendables cuando la búsqueda se hace sobre un solo atributo. En caso de necesitarse una combinación de atributos es preferible usar listas invertidas.

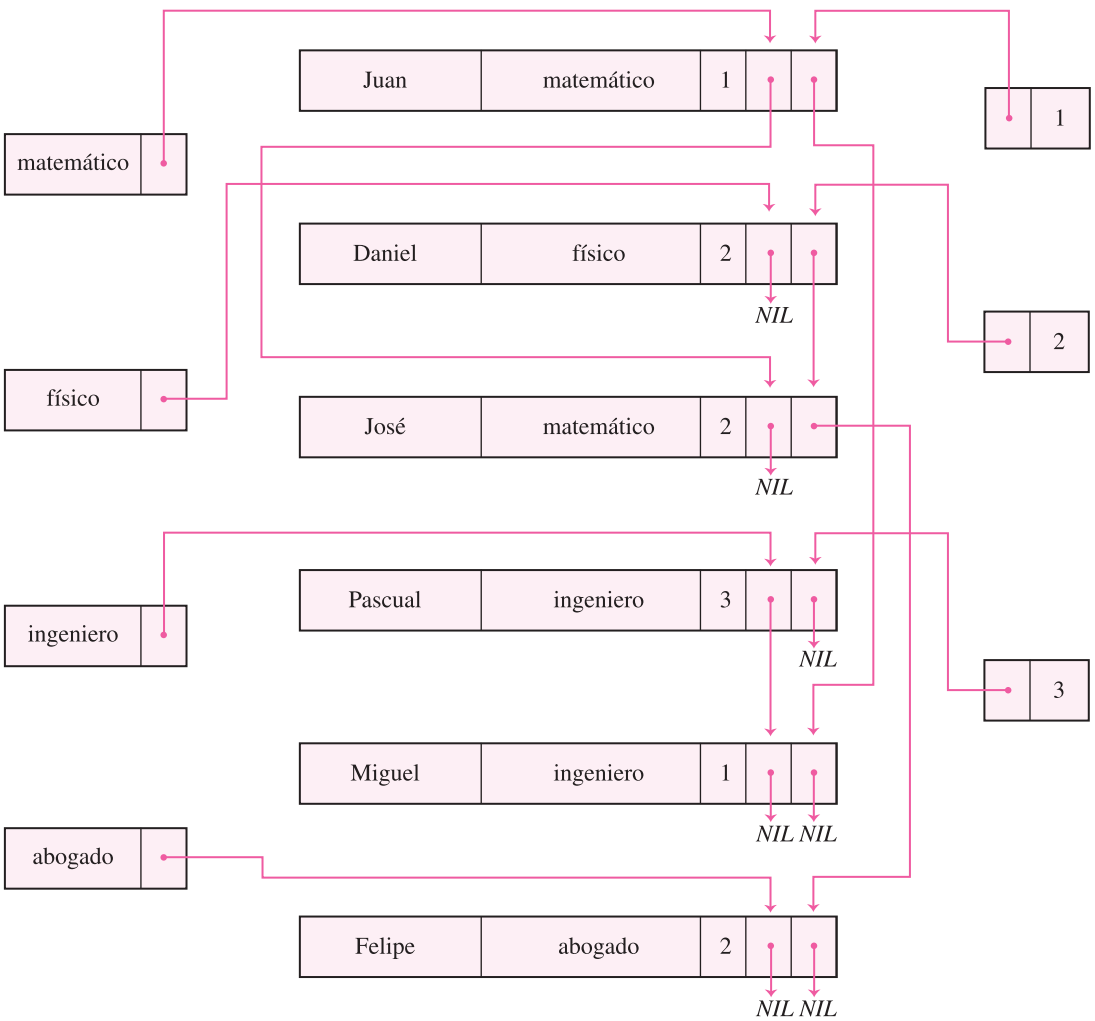


FIGURA 9.38
Multilistas.

EJERCICIOS

Búsqueda interna

1. Escriba un programa para búsqueda secuencial en un arreglo desordenado, que obtenga todas las ocurrencias de un dato dado.

2. Dado un arreglo que contiene los nombres de *N* alumnos ordenados alfabéticamente, escriba un programa que encuentre un nombre dado en el arreglo. Si lo encuentra debe dar como resultado la posición en la que lo encontró. En caso contrario, debe enviar un mensaje adecuado.

3. Dado un arreglo de *N* componentes que contienen la siguiente información:

- Nombre del alumno
- Promedio
- Número de materias aprobadas

Escriba un programa que lea el nombre de un alumno y obtenga como resultado el promedio y el número de materias aprobadas por dicho alumno. Si el nombre dado no está en el arreglo, envíe un mensaje adecuado.

a) Considere que el arreglo está desordenado.
b) Considere que el arreglo está ordenado.

4. Escriba un programa para búsqueda secuencial en arreglos ordenados de manera descendente.

5. Escriba un programa para búsqueda secuencial en listas simplemente ligadas que se encuentran desordenadas. Si el elemento se encuentra en la lista, indique el número de nodo en el cual se encontró. En caso contrario, emita un mensaje adecuado.

6. Escriba un programa para búsqueda secuencial en listas simplemente ligadas, ordenadas de manera descendente.

7. Escriba un programa de búsqueda binaria en arreglos ordenados.

a) De manera ascendente.
b) De manera descendente.

8. Resuelva el inciso *b* del problema 3 utilizando el algoritmo de búsqueda binaria.

9. Defina una clase *Arreglo*, según lo visto en el capítulo 1. En la clase debe incluir por lo menos dos métodos —de los estudiados en este capítulo— para buscar un elemento almacenado en el arreglo.

10. Dado que se requiere almacenar los registros con clave

23, 42, 5, 66, 14, 43, 59, 81, 37, 49, 28, 55, 94, 80 y 64

en un arreglo de 20 elementos, defina una función *hash* que distribuya los registros en el arreglo. Si hubiera colisiones, resuélvalas aplicando el método de reasignación lineal.

11. De un grupo de *N* alumnos se tienen los siguientes datos:

- Matrícula: valor entero comprendido entre 1 000 y 4 999
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres

El campo clave es matrícula. Los *N* registros han sido almacenados en un arreglo, aplicando la siguiente función *hash*:

$$H\,(\text{clave}) = \text{dígitos\_centrales}(\text{clave}^2) + 1$$

Las colisiones han sido tratadas con el método de doble dirección *hash*.

Escriba un subprograma que lea la matrícula de un alumno y regrese como resultado su nombre y dirección. En caso de no encontrarlo, emita un mensaje adecuado.

12. Se quiere almacenar en un arreglo los siguientes datos de *N* personas:

- Clave de contribuyente: alfanumérico, de longitud 6.
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres
- Saldo: real

Defina una función *hash* que permita almacenar en un arreglo los datos mencionados. Utilice el método de encadenamiento para resolver las colisiones.

13. Presente y explique una función *hash* que permita almacenar en un arreglo los elementos de la tabla periódica de los elementos de química y sus propiedades, de manera uniforme. La clave está dada por el nombre de los elementos.

14. Utilice la función definida en el ejercicio anterior para insertar y eliminar los elementos que se presentan a continuación:

Insertar: sodio, oro, osmio, litio, boro, cobre, plata, radio.
Eliminar: oro, osmio, boro, cobre, plata.

15. Dados los 12 signos del zodiaco (capricornio, acuario, piscis, aries, tauro, géminis, cáncer, leo, virgo, libra, escorpión, sagitario).

a) Escriba un subprograma para almacenarlos en una estructura de tries.
b) Escriba un subprograma de búsqueda para los signos, almacenados según lo especificado en el inciso anterior.

Búsqueda externa

16. Se han almacenado en un archivo secuencial los datos de los empleados de un supermercado:

- Nombre
- Registro Federal de Contribuyentes
- Fecha de ingreso
- Sueldo

Escriba un programa para buscar secuencialmente los datos de un empleado, dado su nombre como entrada.

a) Considere que el archivo está desordenado.
b) Considere que el archivo está ordenado.

17. Escriba un programa de búsqueda binaria en archivos secuenciales ordenados.

18. Defina una función *hash* que permita almacenar y posteriormente recuperar los elementos de la tabla periódica de los elementos de química en un archivo. La clave está dada por el nombre de los elementos. Resuelva las colisiones utilizando un área independiente para almacenar los elementos colisionados.

19. Se desea crear un archivo con información sobre pinos mexicanos. Cada registro contiene los siguientes datos:

- Nombre del pino
- Tipo de hojas
- Tipo de cono

El campo clave es *Nombre del pino*. Defina una función *hash* para almacenar, y posteriormente buscar, los siguientes pinos: *Cembroides, Monophylla, Nelsonii, Flexilis, Lumholtzii, Leiophylla, Douglasiana, Teocote, Herrerai, Montezumae, Cooperi, Contorta, Pondarosa, Arizonica, Caribaea, Patula, Radiata, Muricata, Remorata.*

Resuelva las colisiones utilizando un área común para almacenar los elementos colisionados.

20. Utilice la función definida en el ejercicio 13 para insertar y eliminar los elementos que se indican a continuación:

Insertar: sodio, oro, osmio, litio, boro, cobre, plata, radio.
Eliminar: oro, osmio, boro, cobre, plata.

El número de cubetas es dos ($N = 2$) y cada cubeta tiene dos registros. La densidad de ocupación permitida es 80%; en caso de superar este porcentaje se aplicarán expansiones totales.

a) Dibuje un esquema de la organización después de insertar los elementos osmio y plata; y luego de eliminar oro, boro y plata.
b) Diga qué claves originaron que el número de cubetas se expandiera o redujera.

21. Resuelva el problema anterior, pero ahora aplicando expansiones parciales, en caso de tener un porcentaje de ocupación mayor al permitido.

22. Sea $N = 2$ el número de cubetas. Cada cubeta tiene dos registros y se establece una densidad de ocupación permitida de 85%. Una vez superada esta densidad, se aplicarán expansiones parciales.

H (clave) = clave MOD *N*
Claves a insertar: 36, 11, 48, 06, 75, 65, 38, 88, 23, 14, 12

a) Dibuje un esquema de la organización después de insertar los elementos 06, 38, 23, 12.
b) Diga qué claves originaron que el número de cubetas se expandiera.

23. Considere el archivo del problema anterior. Elimine los registros con claves 75, 06, 65, 14, 12, 36, 23.

a) Dibuje un esquema de la organización después de eliminar los elementos 75, 14, 12.
b) Diga qué claves originaron que el número de cubetas se redujera.

24. Sea $N = 4$ el número de cubetas. Cada cubeta tiene dos registros, y se establece una densidad de ocupación permitida de 80%. Defina una función *hash* para:

Insertar las claves 77, 34, 23, 26, 39, 60, 19, 43, 70, 51, 17, 28
Eliminar las claves 23, 39, 60, 43, 17

a) Aplique expansiones totales.
b) Aplique expansiones parciales.

25. Determine cuál es el número de cubetas necesario para almacenar en un archivo nombre, apellido, edad, escolaridad y delito cometido por reos del Reclusorio Norte. El reclusorio tiene 2 700 presos.

Nota: Utilice el método de las expansiones totales. Cada cubeta tiene 50 registros. Al tener 80% de llenado se expande.

26. Determine cuál es el número de cubetas necesario para almacenar en un archivo los registros de los 5 000 000 de clientes que maneja una empresa de tarjetas de crédito.

Nota: Utilice el método de las expansiones parciales. Cada cubeta tiene 500 registros. Al tener 85% de llenado se expande.

27. Se tiene un archivo con registros que almacenan información sobre clientes de distintas sucursales bancarias. Los datos que se manejan por cada cliente son:

- Clave de la sucursal
- Nombre del titular
- Número de cuenta
- Saldo
- Número de préstamo
- Importe

Se tiene inversión sobre el campo clave *sucursal*.

a) Obtenga los registros de los clientes que tengan un préstamo mayor a $5 000 en la sucursal *Lima*.
b) Obtenga los registros de los clientes que tengan un préstamo mayor a $5 000 en la sucursal *Lima* y un saldo en su cuenta mayor a $3 000 en la sucursal *Río*.
c) Obtenga los registros de los clientes de la sucursal *Río* que tengan en su cuenta un saldo mayor a $6 000, o los registros de los clientes de la sucursal *Quito* que tengan un préstamo menor a $1 000 y un saldo en su cuenta mayor a $2 000.
d) Obtenga los registros de los clientes de la sucursal *Córdoba* que tengan un saldo mayor a $5 000 o un préstamo menor a $1 000.
e) Si se quiere determinar:

(sucursal = “Lima”) OR (sucursal = “Quito”) OR
(sucursal = “Río”) OR (sucursal = “Córdoba”)

y las correspondientes listas son de 100, 50, 120 y 200 elementos, respectivamente, ¿cuál será la secuencia óptima para alcanzar un costo mínimo?

28. En un archivo se ha almacenado la tabla periódica de los elementos químicos, junto con sus propiedades:

- Nombre
- Número atómico (*NA*)
- Peso atómico (*PA*)
- Punto de ebullición (*PE*)
- Punto de fusión (*PF*)
- Densidad (*DEN*)
- Electronegatividad (*EO*)
- Conductancia eléctrica (*CE*)
- Conductancia térmica (*CT*)

Se tiene inversión sobre los campos *punto de ebullición* y *punto de fusión*.

a) Obtenga los registros de los elementos alcalinotérreos. Éstos se determinan por las siguientes características: $EO = 2$; $1.54 < DEN < 5.01$ y su *PF* está comprendido entre los valores 922 y 1 560.
b) Obtenga los registros de los elementos del grupo 6B. Éstos se determinan por las siguientes características: $EO = -2, 4$ o 6; su *PE* está comprendido entre los valores 90.18 y 12.61.

29. En un archivo se han almacenado los datos de *N* profesionales.

- Clave de contribuyente
- Nombre
- Profesión
- Nacionalidad

Se tiene inversión sobre los campos *profesión* y *nacionalidad*:

a) Obtenga los registros de todos los ingenieros mexicanos.
b) Obtenga los registros de todos los ingenieros mexicanos de más de 60 años de edad.
c) Obtenga los registros de todos los ingenieros mexicanos de más de 60 años de edad o los pintores uruguayos.
d) Obtenga los registros de todos los abogados peruanos o los médicos chilenos de menos de 30 años.
e) Si se quiere determinar:

(profesión = ingeniero) OR (profesión = pintor) OR (profesión = médico) y las listas son de 100, 200 y 300 claves, respectivamente, ¿cuál será la secuencia óptima para alcanzar un costo mínimo?

BIBLIOGRAFÍA

La bibliografía que se presenta a continuación es fragmentaria, en el sentido de que sólo se incluyen obras que han servido de base para esta exposición o que están directamente vinculadas con ella.

Ackerman, A. F. *Quadratic Search for Hash Tables of Size P.* Comm. ACM 17, 1974.

Adelson-Velskii, G. y Landis, E. *An Algorithm for the Organization of Information*. Dokl. Akad Nauk SSSR, Mathemat, 146, 1962.

Aho, A., Hopcroft, J. y Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

————, *Data Structures and Algoritms.* Addison-Wesley, Publishing Company, 1983.

Albizuri, M. *Estructuras de datos.* Editorial Limusa, 1989.

Amble, O. y Knuth, D. *Ordered Hash Tables.* Computer J. 18, 1975.

Anderson, M. R. y Anderson, M. G. *Comments on Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets.* Comm. ACM 22, 1979.

Augenstein, M. y Tenenbaum, A. *A Lesson in Recursion and Structured Programming.* SIGCSE Bulletin, 8, 1976.

Baase, S. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, Mass., 1978.

Baer, J. y Schwab, B. *A Comparison of Tree Balancing Algorithms.* Comm. ACM 20, 1977.

Barron, D. *Recursive Techniques in Programming.* American-Elsevier, Nueva York, 1968.

Batagelj, V. *The Quadratice Hash Method When the Table Size is not a Prime Number.* Comm. ACM 18, 1975.

Bayer, R. *Binary B-trees for Virtual Memory.* Proc. 1971 ACM SIGFIDET Workshop, ACM, Nueva York.

————, *Symmetric Binary B-trees: Data Structure and Maintenance Algorithms.* Acta Informática, 1, 1972.

————, y Metzger, J. *On Encipherment of Search Trees and Random Access Files.* ACM. Trans. Database Syst. 1, 1976.

————, y Unterauer, K. *Prefix B-trees.* ACM Trans. Database Syst. 2, 1977.

————, y Schkolnick, N. *Concurrency of operations on B-tree.* Acta Inf., 9, 1977.

Bell, J. *The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering.* Comm. ACM 13, 1970.
————, y Kaman, C. *The Linear Quotient Hash Code.* Comm. ACM 13, 1970.
Bellman, R. *Dynamic Programming.* Princeton University Press, Princeton, N. J., 1957.
Bentley, J. *Multidimensional Binary Search Trees Used for Associative Searching.* Comm. ACM 18, 1975.
————, y Friedman, J. *Algorithms and Data Structure for Range Searching.* ACM Computing Surveys 11, 1979.
Berliner, H. *The B-tree Search Algorithm: A Best-First Proof Procedure.* Tech. Rep. CMU-CA-78-112, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, 1978.
Berztiss, A. *Data Structures, Theory and practice,* 2a. ed., Academic Press, Nueva York, 1977.
Bird, R. *Improving Programs by the Introduction of Recursion.* Comm. ACM 20, 1977.
————, *Notes on Recursion Elimination.* Comm. ACM 20, 1977.
Boothroyd, J. *Algorithm 201 (Shellsort).* Comm. ACM 6, 1963.
————, *Sort of a Section of the Elements of an Array by Determining the Rank of Each Element: Algorithm 25.* Comp. J. 10, 1967.
Brillinger, P. y Cohen, D. *Introduction to Data Structures and Non-numeric Computation.* Prentice-Hall, Englewood Cliffs, N. J., 1972.
Brown, M. *A Storage Scheme for Height-Balanced Trees.* Inf. Proc. Lett., 7, 1978.
Bruno, J. y Coffman, E. *Nearly Optimal Binary Search Trees.* Proc. IFIP Congress 71, North-Holland, Amsterdam, 1972.
Burkhard, W. *Hashing and Trie Algorithms for Partial Match Retrieval.* ACM Trans., Vol. 1, 1976.
Carter, J. y Wegman, M. *Universal Classes of Hash Functions.* IBM Research Report RC 6495, Thomas J. Watson Research Center, Yorktown Heights, Nueva York, 1977.
————, *Universal Classes of Hash Functions.* Proc. Ninth Annual ACM SYMP. on Theory of Computing, 1977.
Clampett, H. *Randomized Binary Searching With Tree Structures.* Comm. ACM 7, 1964.
Comer, D. *The Ubiquitous B-tree.* ACM Computing Surveys 11, 1979.
————, *A Note on Median Split Trees.* ACM Trans. Prog. Lang. and Sys. 2, 1980.
Dijkstra, E. W. *Notes on Structured Programming.* Structured Programming. Academic Press, Nueva York, 1972.
D'Imperio, M. *Data Structures and their Representation in Storage.* Annual Review Automatic Programming, 5, Pergamon Press, Elmsford, Nueva York, 1969.
Driscoll, J. y Lien, Y. *A Selective Trasversal Algorithm for Binary Search Trees.* Comm. ACM 21, 1978.
Elson, M. *Data Structures.* Science Research Associates, Palo Alto, Ca., 1975.
Finkel, R. y Bentley, J. *Quad Trees: A Date Structure for Retrieval on Composite Keys.* Acta Informática, 4, 1975.
Flores, I. *Computer Sorting.* Prentice-Hall, Englewood Cliffs, N. J., 1969.
————, y Madpis G. *Average Binary Search Lenghts for Dense Ordered Lists.* Comm. ACM 14, 1971.
Floyd, R. *Algorithm 113 (Treesort).* Comm. ACM 5, 1962.

Floyd, R. *Algorithm 243 (Treesort).* Comm. ACM 7, 1964.
————, *Algorithm 245 (Treesort 3).* Comm. ACM 7, 1964.
Foster, C. *Information Storage and Retrieval Using AVL Trees.* Proc. ACM 20th National Conf., ACM, Nueva York, 1965.
————, *A Generalization of AVL Trees.* Comm. ACM 16, 1973.
Frazer, W. y McKellar, A. *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting.* J. ACM 17, 1970.
Garey, M. *Optimal Binary Search Trees With Restricted Maximal Depth.* SIAM J. Comp. 2, 1974.
Garsia, A. y Wachs, M. *A New Algorithm for Minimum Cost Binary Trees.* SIAM J. Comp. 6, 1977.
Ghosh, S. y Lum, V. *Analysis of Collisions when Hashing by Division.* Inf. Syst., 1, 1975.
Gilstad, R. *Polyphase Merge Sorting ... An Advanced Technique.* Proc. AFIPS Eastern Jt. Comp. Conf., 18, 1960.
Gonnet G. y Rogers, L. *The Interpolation-Sequential Search Algorithm.* Inf. Proc. Lett., 6, 1977.
————, y Munro, J. *Efficient Ordering of Hash Tables.* SIAM J. Comp. 8, 1979.
Gotlieb, C. y Gotlieb, L. *Data Types and Data Structures.* Prentice-Hall, Englewood Cliffs, Nueva York, 1978.
Greene, D. y Knuth, D. *Mathematics for the Analysis of Algorithms.* Birkhauser, Boston, Mass, 1983.
Grimaldi Ralph. *Matemáticas Discretas y Combinatorio.* Addison-Wesley Iberoamericana, 1977.
Gudes, E. y Tsur, S. *Experiments with B-tree Reorganization.* ACM SIGMOD Symposium on Management of Data, 1980.
Guibas, L. McCreight E. Plass, M. y Roberts, J. *A New Representation for Linear Lists.* Proc. 9th ACM Symp. Theory of Comp., Nueva York, 1977.
Harel, D. *Algorithmics.* Addison-Wesley, 1987.
Harrison, M. *Data Structures and Programming.* Scott-Foresman, Glenville, Ill., 1973.
Held, G. y Stonebraker, M. *B-trees Re-examined.* Comm. ACM 21, 1978.
Hirschberg, D. *An Insertion Technique for One-sided Height-Balanced Trees.* Comm. ACM 19, 1976.
Hoare, C. *Partition, Algorithm 63: Quicksort, Algorithm 64, Find, Algorithm 65.* Comm. ACM 4, 1961.
————, *Quicksort.* Comp. J. 5, 1962.
Hoare, C. A. R. *Notes on Data Structuring.* Structured Programming. Academic Press, Nueva York, 1972.
————, y Dahl, O. *Structured Programming.* Academic Press, 1972.
Hopgood, F y Davenport, J. *The Quadratic Hash Method Where the Table Size is a Power of* 2. Comp. J. 15, 1972.
Horowitz, E. y Sahni, S. *Algorithms: Design and analysis.* Computer Science Press, MD, 1977.
————, S. *Fundamentals of Computer Algorithms.* Computer Science Press, Inc., 1978.
Hu, T. y Tucker, A. *Optimum Computer Search Trees.* SIAM J. Appl. Math., 21, 1971.
Jaime, A. *Estructuras de Información.* McGraw-Hill, 1989.

Joyanes Aguilar, L. *Fundamentos de Programación. Algoritmos y Estructuras de Datos.* McGraw-Hill, 1988.
Karlton, P., Fuller, S., Scroggs, R. y Kachler, E. *Performance of Height Balanced Trees.* Comm. ACM 19, 1976.
Knott, G. *Hashing Functions.* Computer Journal, 18, 1975.
Knuth, D. *The Art of Computer Programming.* Vol. 1 / Fundamental Algorithms. Reading, Mass: Addison-Wesley, 1968.
————, *Optimum Binary Search Trees.* Acta Informática, 1, 1971.
————, *The Art of Computer Programming.* Vol. 3 / Sorting and Searching. Reading, Mass: Addison-Wesley, 1973.
————, *El Arte de Programar Ordenadores.* Vol. 1 / Algoritmos fundamentales. Editorial Reverté, 1980.
Kolman, Bernard, *et al. Estructuras de Matemáticas Discretas para la Computación.* Prentice-Hall, 1986.
Lewis, T. y Smith, M. *Applying Data Structures.* Houghton Mifflin, Boston, 1976.
————, *Estructuras de Datos.* Paraninfo, 1985.
Lipschutz, S. *Estructura de Datos.* Serie Schaum. McGraw-Hill, México, 1989.
Lorin, H. *A Guided Bibliography to Sorting.* IBM Syst. J., 10, 1971.
Luccio, F y Pagli, L. *Qn the Height of Height-Balanced Trees.* IEEE Trans. Comptrs., 1976.
————, *Power Trees.* Comm. ACM 21, 1978.
Lucas, Reyrin y School: *Algorítmica y Representación de Datos.* Vol. 1, Secuencias, Autómatas de Estados Finitos. Masson, 1985.
Martin, W. *Sorting.* Comp. Surveys, 3, 1971.
Maurer, H. y Lewis, T. *Hash Table Methods.* Comp. Surveys, 7, 1975.
————, Ottmann, T. y Six, H. *Implementing Dictionaries Using Binary Trees of Very Small Height.* Inform. Proc. Letters, 5, 1976.
————, *Data Structures and Programming Techniques.* Prentice-Hall, Englewood, Cliffs, N. J., 1977.
————, y Ottmann, T. *Tree Structures for Set Manipulation Problems.* In Mathematical Foundations of Computer Science, Springer-Verlag, Nueva York, 1977.
McCreight, E. *Pagination of B*-trees with Variable-Lenght Records.* Comm. ACM 20, 1977.
————, *Priority Search Trees.* SIAM J. of Comp., 1985.
Melhorn, K. *Nearly Optimal Binary Search Trees.* Acta Informática, 5, 1975.
————, *Dynamic Binary Search.* SIAM J. Comp. 8, 1979.
————, *Data Structures and Algorithms.* Vol. 1, Sorting and Searching. Springer-Verlag, 1984.
Miller, R., Pippenger, N., Rosenberg, A. y Snyder, *L. Optimal 2-3 Trees.* IBM Research Rep. RC 6505, IBM Research Lab., Yorktown Heights, Nueva York, 1977.
Nievergelt, J. y Wong, C. *On Binary Search Trees.* Proc. IFIP Congress 71, North-Holland, 1972.
————, y Reingold, E. *Binary Search Trees of Bounded Balance.* SIAM J. Comp. 2, 1973.
————, *Binary, Trees and File Organization.* ACM Computing Surveys, 1974.
Pfaltz, J. *Computer Data Structures.* McGraw-Hill, Nueva York, 1977.
Pohl, I. *A Sorting Problem and its Complexity.* Comm. ACM 15, 1972.

Pratt, V. *Shellsort and Sorting Networks.* Garland, Nueva York, 1979.
Raiha, K. y Zweben, S. *An Optimal Insertion Algorithm for One-Sided Height-Balanced Binary Search Trees.* Comm. ACM 22, 1979.
Rosenberg, A. y Snyder, L. *Minimal Comparison 2-3 Trees.* SIAM J. Comput. 7, 1978.
Saxe, J. y Bentley, J. *Transforming Static Data Structures to Dynamic Structures.* Research Report CNUS-CS-79-141, Carnegie-Mellon University, Pittsburgh, 1979.
School, P. *Algorítmica y Representación de Árboles.* Vol. 2, Recursividades y Árboles. Masson, 1986.
Scowen, R. *Quicksort: Algorithm 271.* Comm. ACM 8, 1965.
Sedgewick, R. *Quicksort.* Report No. STAN-CS-75-492, Dept. of Computer Science, Stanford University, Ca., 1975.
————, *The Analysis of Quicksort Programs.* Acta Informática, 7, 1977.
————, *Implementing Quicksort Programs.* Comm. ACM 21, 1978.
Shell, D. *A Highspeed Sorting Procedure.* Comm. ACM 2, 1959.
————, Optimizing: *Optimizing the Polyphase Sort.* Comm. ACM 14, 1971.
Shneiderman, B. *Polynomial Search.* Software-Practice and Experience, 3, 1973.
————, *Jump Searching: A Fast Sequential Search Technique.* Comm. ACM 21, 1978.
Singleton, R. *An Efficient Algorithm for Sorting with Minimal Storage: Algorithm 347.* Comm. ACM 12, 1969.
Sprungnoli, R. *Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets.* Comm. ACM 20, 1977.
Stephenson, C. *A Method for Constructing Binary Search Trees by Making Insertions at the Root.* IBM Research Report RC6298, Thomas Watson Research Center, Yorktown Heights, Nueva York, 1976.
Tanner, R. *Minimean Merging and Sorting: An Algorithm.* SIAM J. Comp. 7, 1978.
Tenenbaum, A. y Augenstein, M. *Estructuras de Datos en Pascal.* Prentice-Hall Hispanoamericana, 1983.
Tremblay, J. y Sorenson, P. *An Introduction to Data Structures with Applications.* McGraw-Hill, Nueva York, 1976.
Van Emden, N. *Increasing Efficiency of Quicksort.* Comm. ACM 13, 1970.
Vuillemin, J. *A Unifying look at Data Structures.* Comm. ACM 23, 1980.
Walker, W. y Gotlieb, C. *A Top-Down Algorithm for Constructing Nearly Optimal Lexico-graphic Trees.* Graph Theory and Computing. Academic Press, Nueva York, 1972.
Williams, J. *Heapsort: Algorithm 232.* Comm. ACM 7, 1964.
Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N. J., 1976.
————, *Algoritmos + Estructuras de Datos = Programas.* Ediciones del Castillo.
————, *Algoritmos y Estructuras de Datos.* Prentice-Hall Hispanoamericana, 1987.
Wulf, W., Shaw, M., Hilfinger, P. y Flon, L. *Fundamental Structures of Computer Science.* Addison-Wesley, Reading, Mass., 1981.
Yao, A. *Qn Random 2-3 Trees.* Acta Informática, 9, 1978.

GLOSARIO

Árbol. Estructura jerárquica aplicada sobre una colección de objetos llamados nodos, en la que uno de ellos se conoce como nodo raíz, y cuyas relaciones entre nodos se identifican como padre-hijo, hermano, etcétera.

Árbol abarcador. Es un árbol libre que conecta todos los vértices de *V*.

Árbol balanceado. Conocido también como árbol AVL, es un árbol binario de búsqueda en el cual, para todo nodo del árbol, la altura de los subárboles izquierdo y derecho no debe diferir en más de una unidad.

Árbol binario. Árbol en el cual cada nodo puede tener hasta dos descendientes directos y cuyas ramas están ordenadas.

Árbol de búsqueda. *Véase* trie.

Árbol multicaminos. Árbol en el que cada nodo puede tener más de dos descendientes directos y cuyas ramas están ordenadas.

Arreglo. Colección finita, homogénea y ordenada de elementos.

Arreglo de *N* dimensiones. Aquel en el cual cada uno de sus elementos debe identificarse por *n* índices que marquen su posición exacta dentro del arreglo.

Arreglos paralelos. Estructura formada por dos o más arreglos cuyos elementos se corresponden, por lo general en relación de uno a uno.

Bosque. Conjunto ordenado de uno o más árboles.

Búsqueda. Operación que permite recuperar datos previamente almacenados, aunque esta operación puede resultar en fracaso si no se encuentra el dato buscado.

Búsqueda externa. Aquella en la que todos los datos se encuentran en archivos residentes en dispositivos de almacenamiento secundario, tales como discos, cintas, etcétera.

Búsqueda interna. La que se realiza con los datos residentes en la memoria principal de la computadora.

Camino. Un camino *P* de longitud *n* desde un vértice *v* a un vértice *w* se define como la secuencia de *n* vértices que se debe seguir para llegar del nodo origen al nodo destino.

Cola. Lista de elementos en la cual la edición de elementos se lleva a cabo por un extremo, y la eliminación se realiza por otro.

Colisión. La que se origina al utilizar una función *hash*, cuando dos o más datos con distintas claves tienen la misma dirección de memoria.

Conjunto. Es un dato estructurado integrado por un grupo de objetos del mismo tipo, aunque el tipo sólo puede ser entero, carácter, enumerado o rango.

Dato estructurado. Está formado por varios componentes, cada uno de los cuales puede ser a su vez un dato estructurado. Todos los componentes de un dato estructurado se identifican con el mismo nombre.

Dato simple. Aquel que hace referencia a un único valor a la vez y que ocupa una casilla de memoria.

Estructura dinámica de datos. Aquella que permite la asignación de espacio en memoria durante la ejecución de un programa, conforme lo requieran las variables de éste, como los árboles y las listas.

FIFO. Iniciales de la expresión en inglés *First In*, *First Out* (el primero en entrar es el primero en salir), que indican el orden de inserción y eliminación de los elementos de una cola.

Función *hash*. Aquella que permite transformar una clave en una dirección de memoria.

Gráfica completa. Se dice que una gráfica es completa si cada vértice v de G es adyacente a todos los demás vértices de G.

Gráfica conexa. Se dice que una gráfica es conexa si existe un camino simple entre dos de sus nodos cualesquiera.

Gráfica dirigida. Se caracteriza porque sus aristas tienen asociada una dirección.

Gráfica etiquetada. Se dice que una gráfica G está etiquetada si sus aristas tienen asignado un valor.

Gráficas. Estructuras de datos que permiten representar diferentes tipos de relaciones entre los objetos.

Gráficas no dirigidas. Su característica principal es que sus aristas son pares no ordenados de vértices.

Índice. Indicador que señala la posición de un componente en un dato estructurado.

LIFO. Iniciales de la expresión en inglés *Last In*, *First Out* (el último en entrar es el primero en salir), que indican el orden de inserción y eliminación de los elementos de una pila.

Lista. Colección de elementos llamados nodos, cuyo orden se establece por medio de punteros.

Lista circular. Aquella en la que su último elemento apunta al primero.

Lista doblemente ligada. Colección de elementos llamados nodos, en la cual cada nodo tiene dos punteros, uno apuntando al nodo predecesor y el otro al nodo sucesor.

Lista invertida. Lista que contiene las claves de los elementos que poseen un determinado atributo, lo cual facilita la búsqueda de los elementos que posean dicho atributo.

Matriz. Estructura de datos que permite organizar la información en renglones y columnas. Es un arreglo bidimensional.

Método de búsqueda. Se caracteriza por el orden en el cual se expanden los nodos.

Métodos de búsqueda *breadth-first.* Se expanden los nodos en el orden en que han sido generados.

Métodos de búsqueda *depth-first.* Se expanden los nodos generados más recientemente.

Multigráfica. Una gráfica se denomina multigráfica si al menos dos de sus vértices están conectados por dos aristas.

Multilista. Estructura que permite almacenar en una o varias listas las direcciones de los elementos que poseen uno o más atributos específicos, lo cual facilita su búsqueda.

Notación infija. Se dice que una expresión aritmética tiene notación infija cuando sus operadores están entre los operandos, por ejemplo, $A + B$.

Notación postfija. Se dice que una expresión aritmética tiene notación postfija cuando sus operadores están al final de los operandos, por ejemplo, $AB+$.

Notación prefija. Se dice que una expresión aritmética tiene notación prefija cuando sus operadores están al inicio de los operandos, por ejemplo, $+AB$.

Ordenación externa. En esta forma de ordenación los datos se toman de archivos residentes en dispositivos de almacenamiento secundario, tales como discos, cintas, etcétera.

Ordenación interna. Se aplica en la ordenación de arreglos y se realiza con todos los datos de éstos alojados en la memoria principal de la computadora.

Ordenar. Organizar un conjunto de datos u objetos en una secuencia específica, por lo general ascendente o descendente.

Pila. Lista de elementos a la cual se puede insertar o eliminar elementos sólo por uno de sus extremos.

Puntero. Dato que almacena una dirección de memoria, en donde se almacena una variable.

Recursión. Herramienta de programación que permite definir un objeto en términos de él mismo.

Registro. Es un dato estructurado en el cual sus componentes pueden ser de diferentes tipos, incluso registros o arreglos. Todos sus componentes se identifican con un nombre.

Representación lineal de estructuras no lineales. Procedimiento para convertir los índices de cada uno de los elementos de arreglos de dos o más dimensiones en una posición específica dentro de un arreglo de una dimensión.

Solución de colisiones. Consiste en reubicar un dato cuya dirección, calculada por una función *hash*, ya haya sido utilizada por otro dato.

Trie. Árbol en el que la información de cada nodo es común a todos sus sucesores.

ÍNDICE ANALÍTICO