ESTRUCTURA DE DATOS ORIENTADA A OBJETOS

Algoritmos con C++

PEARSON Prentice Hall

Silvia Guardati

ESTRUCTURA DE DATOS ORIENTADA A OBJETOS

Algoritmos con C++

ESTRUCTURA DE DATOS ORIENTADA A OBJETOS

Algoritmos con C++

Silvia Guardati Buemo

Instituto Tecnológico Autónomo de México

REVISIÓN TÉCNICA:

Fabiola Ocampo Botello
Escuela Superior de Cómputo
Instituto Politécnico Nacional

José Luis García Cerpas
Centro de Enseñanza Técnica Industrial, Jalisco

PEARSON Educación®

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

GUARDATI BUEMO, SILVIA
Estructura de datos orientada a objetos: Algoritmos con C++

PEARSON EDUCACIÓN,
México, 2007

ISBN: 978-970-26-0792-2
Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 584

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Rodrigo Romero Villalobos

PRIMERA EDICIÓN, 2007

ISBN 10: 970-26-0792-2
ISBN 13: 978-970-26-0792-2

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 10 09 08 07

PEARSON
Educación ®

CONTENIDO

A mi familia

El mejor profeta del futuro es el pasado.

Lord Byron

INTRODUCCIÓN

Las **Estructuras de Datos** son uno de los temas centrales de estudio en el área de la computación, las cuales permanecen vigentes y resisten al paso del tiempo como los pilares de piedra de un antiguo puente romano. Seguramente ya no están los troncos que ayudaron a cruzar a guerreros y carruajes, a vencedores y vencidos, pero las piedras, encargadas de sostener a todos, ahí están... resistiendo al paso del tiempo y a la fuerza del agua.

Hoy, como en los orígenes de la computación, necesitamos conocer qué son y cómo usar a las estructuras de datos, que serán las piedras que nos ayudarán a construir y a sostener soluciones robustas y útiles para diversos tipos de problemas.

El objetivo de este libro es presentar las principales estructuras de datos, basándonos en el paradigma orientado a objetos. Es decir, las estructuras se definirán y usarán siguiendo este método. Por lo tanto, cada estructura será una clase, sus características quedarán representadas a través de atributos, y las operaciones por medio de métodos. De cada una de las principales estructuras se presenta la manera en la que se almacena y, en consecuencia, se recupera la información. Se explica la lógica requerida para llevar a cabo las operaciones más importantes y se muestra la implementación de estos algoritmos. También se incluyen ejemplos de aplicación de las estructuras. Para la programación de algoritmos y ejemplos se utiliza el lenguaje de programación ***C++***, por ser uno de los lenguajes orienta-

dos a objetos más conocidos y usados, tanto en el ámbito académico como en el profesional.

El enfoque del libro está orientado a:

- Todos los que quieran conocer y entender los principios de la programación orientada a objetos.
- Todos los que quieran conocer y entender las estructuras de datos.
- Todos los que quieran conocer y entender la implementación de los principales algoritmos dedicados a manejar las estructuras de datos.
- Todos los que quieran aprender a usar las estructuras de datos en la solución de problemas, y la implementación de estas soluciones.

Para un mejor aprovechamiento del libro es necesario tener conocimientos sobre:

- Datos predefinidos: enteros, reales, carácter, cadenas y lógicos.
- Estructuras selectivas y repetitivas: if, switch, while y for.
- Instrucciones para lectura y escritura.

Cómo está organizado este libro

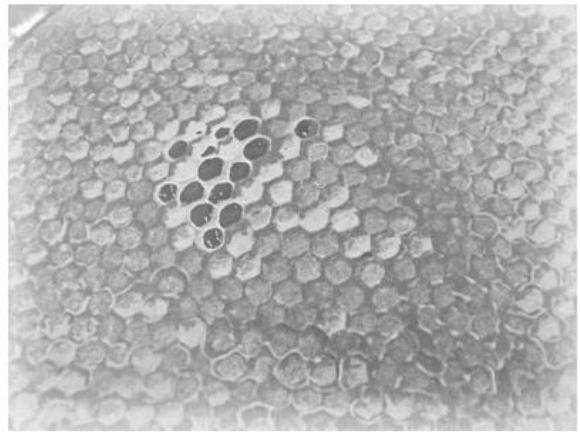
El material del libro está organizado en diez capítulos. Los tres primeros ofrecen una introducción a la Programación Orientada a Objetos (POO), la cual servirá de base para entender el resto del libro. Se presentan temas básicos de la POO, como abstracción, herencia y polimorfismo; asimismo se explican conceptos relacionados, como sobrecarga y plantillas. El capítulo 4 trata sobre los arreglos; y dada la orientación del libro, se ven como una clase, con sus atributos y métodos. El capítulo 5 presenta las pilas y colas. Estas dos estructuras son naturalmente tipos abstractos de datos, por lo que su representación por medio de la POO resulta inmediata. En el capítulo 6 se estudian las listas ligadas (o vinculadas) con todas sus variantes: las simplemente ligadas, las doblemente ligadas, las circulares y las ortogonales. El capítulo 7 está dedicado a los árboles: se estudian en general, los binarios, los binarios de búsqueda, los balanceados y los árboles-B y B^+. El capítulo 8 explica las gráficas, incluyendo las dirigidas y las no dirigidas. Finalmente, los capítulos 9 y 10 presentan los principales algoritmos de ordenación y búsqueda; temas que, por su importancia en las estructuras de datos, se consideran relevantes; razón por la que fueron incluidos.

En cada uno de los capítulos se explican los principales conceptos, y se refuerzan con ejemplos que ayudan a su comprensión. Además, se incluyen programas (en algunos casos, sólo las instrucciones requeridas) para mostrar la implementación de los algoritmos y de soluciones a problemas de aplicación de las estructuras estudiadas. Todos los capítulos cuentan también con una sección de ejercicios sugeridos para reafirmar los conceptos estudiados y desarrollar la capacidad de análisis y en la solución de problemas, por medio de las estructuras de datos.

AGRADECIMIENTOS

Este libro es el resultado de muchos años de experiencia como maestra. El enfoque dado a cada uno de los temas, los ejemplos y ejercicios presentados son el reflejo de todo ese tiempo vivido en las aulas. Por lo tanto quiero agradecer muy especialmente a los alumnos, quienes con sus comentarios, preguntas e incluso con el "no entiendo" me estimulan a buscar siempre nuevos caminos para transmitir el conocimiento y la experiencia.

También quiero agradecer a los profesores y funcionarios de la División Académica de Ingeniería del ITAM por su apoyo en la realización de esta obra. Un reconocimiento especial al rector del instituto, doctor Arturo Fernández, por incentivar y promover la elaboración de libros.



CAPÍTULO 1

Introducción a la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es una técnica para escribir programas. Es decir, es la aplicación de un lenguaje orientado a objetos para implementar una solución previamente diseñada, usando el paradigma orientado a objetos.

La POO tiene cuatro características principales: ***abstracción, encapsulamiento, herencia y polimorfismo***. La *abstracción* consiste en ignorar aquellos aspectos, del objeto a describir, que no son relevantes, para de esta manera concentrarse en los que sí lo son. El *encapsulamiento* consiste en incluir dentro de una clase todos los atributos y métodos que la definen, de tal manera que otros objetos puedan usarla sin necesidad de conocer su estructura interna. La *herencia* permite compartir atributos y métodos entre clases y subclases. Finalmente, en el *polimorfismo* una operación puede tener el mismo nombre en diversas clases, y funcionar de manera diferente en cada una. Estos temas se tratarán en las siguientes secciones y capítulos.

1.1 Características de la POO

En el diseño de la solución computacional de problemas se distinguen los datos (información necesaria para llevar a cabo el procesamiento) y las operaciones que podemos hacer sobre ellos. La POO ofrece mecanismos para representar, de manera integrada, los datos y las operaciones.

Como ya se mencionó, la POO tiene ciertas características que la convierten en una poderosa herramienta para solucionar diversos problemas computacionales. A continuación se presentan las características más importantes.

- **Abstracción.** Es el principio que permite (al observar el objeto o concepto que se quiere representar) ignorar aquellos aspectos que no son relevantes, para de esta manera concentrarse en los que sí lo son. Se trata de abstraer los datos (llamados ***atributos***) y las operaciones (llamadas ***métodos***) comunes a un conjunto de objetos y agruparlos bajo un mismo concepto clase. Es decir, facilita la generalización conceptual de los atributos y propiedades de un determinado conjunto de objetos. De esta forma, introducir o eliminar un objeto en una determinada aplicación requerirá un trabajo mínimo.
- **Encapsulamiento** (ocultamiento de información). Se refiere a incluir dentro de la definición de una clase todo lo que se necesita, de tal forma que ningún otro objeto requiera conocer su estructura interna para poder usarla. Es decir, se tomará cada clase y en consecuencia cada objeto como una unidad básica de la cual desconocemos su estructura interna. En la figura 1.1 se grafica esta idea, señalando a la clase (formada por atributos y métodos) como una *caja negra*.

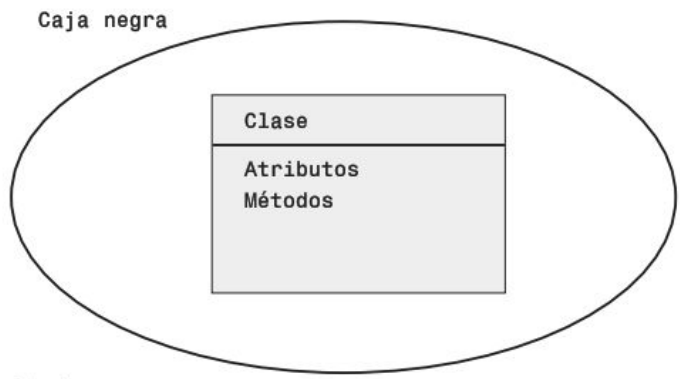


FIGURA 1.1 *Encapsulamiento*

- **Herencia.** Permite compartir atributos y métodos entre clases y clases derivadas. Las clases derivadas, también llamadas subclases, heredan atributos y métodos de las clases superiores, que reciben el nombre de superclases o clases base.

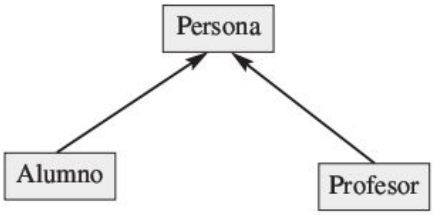


FIGURA 1.2 *Herencia*

Observe el ejemplo de la figura 1.2. La clase *Persona* es una superclase. Las clases *Alumno* y *Profesor* son subclases o clases derivadas de la clase *Persona*, por lo que heredan todos los atributos y métodos de ella. Esta relación expresa que un alumno y un profesor también son personas.

- **Polimorfismo.** Permite implementar múltiples formas de un mismo método, de tal manera que cada una de ellas se adapte a la clase sobre la cual se aplicará.

1.2 Ventajas de la POO

La POO resulta una herramienta muy poderosa para la implementación de soluciones. Cabe destacar que no se pretende, en esta sección, realizar un análisis comparativo con otros paradigmas de programación; pero es muy importante señalar las ventajas que ofrece, siendo éstas:

1. ***Facilita el reuso del diseño y código***. Una vez que el diseño y código correspondiente a una clase fueron probados y validados, resulta relativamente sencillo utilizarlos nuevamente en la solución de otra aplicación.
2. ***Abstracción.*** Permite ver el concepto como un todo, sin tener que distraer la atención en los detalles. Esto representa una gran ventaja en el momento de analizar y representar los objetos involucrados en un problema.
3. ***Ocultamiento o encapsulamiento de la información***. La POO permite ocultar información por medio del encapsulamiento, y de esta forma alcanza mayor seguridad y transparencia en el manejo de la información (se trata a la clase como un todo, no se requiere conocer los detalles).

4. ***Mayor legibilidad.*** Los programas escritos mediante la POO resultan más fáciles de leer y entender ya que son más compactos. Además, los componentes clave del programa son autocontenidos y se pueden comprender rápidamente.

1.3 Tipos abstractos de datos

La abstracción de datos es un concepto básico en la solución de un problema. Ésta permite definir el dominio y la estructura de los datos, el conjunto de atributos que caracterizan a esos datos, así como las operaciones válidas aplicables sobre los mismos. Es decir, es el mecanismo por medio del cual se define un concepto general a partir del conocimiento que se tenga de objetos particulares.

La abstracción da origen a lo que se conoce como un Tipo Abstracto de Datos (ADT, por sus siglas en inglés: *Abstract Data Type*), el cual es un tipo de dato *definido por el usuario*, cuyas operaciones especifican cómo un cliente (el usuario) puede manipular los datos. Por lo tanto el ADT constituye un modelo abstracto que define una interfaz entre el usuario y el dato.

El ADT es *independiente* de la implementación, lo cual permite al diseñador de la solución enfocarse en el modelo de datos y en sus operaciones, sin considerar un lenguaje de programación en particular. Posteriormente, el programador lo traducirá con el lenguaje elegido.

En el siguiente ejemplo se presenta el ADT correspondiente al modelo simplificado de un alumno universitario.

Dominio: alumno universitario.

Datos: representan las características más importantes de todo alumno universitario.

- Nombre: cadena de caracteres
- Dirección: cadena de caracteres
- Matrícula: número entero
- Año de ingreso: número entero
- Carrera: cadena de caracteres
- Promedio: número real

Operaciones válidas definidas para el ADT: representan aquellas operaciones que se pueden realizar sobre o con los datos de un alumno universitario. Para

este ejemplo, se considera que un alumno puede cambiar de domicilio, de carrera, que aprueba materias, etcétera.

- Actualizar Dirección
- Actualizar Promedio
- Actualizar Carrera
- ...

Retomando el ejemplo anterior, el ADT representa a los *alumnos universitarios* en general (se está describiendo un concepto general), mientras que una instancia representa un alumno en particular (con nombre, dirección, inscrito a una carrera, etcétera).

Todo ADT tiene, generalmente, los siguientes elementos: un encabezado, la descripción de los datos y una lista de las operaciones válidas para ese ADT.

- Encabezado: nombre del ADT.
- Descripción de los datos: se especifican los datos y las estructuras correspondientes para representarlos. Los datos constituyen los atributos del concepto u objeto definido por medio del ADT.
- Lista de operaciones: se forma por el conjunto de operaciones que se definen como válidas para el ADT. Para cada operación deberá indicarse:

  Entrada: generalmente proporcionada por el usuario.

  Precondiciones: establecen la situación en la cual se aplicará la operación.

  Proceso: es el conjunto de acciones que definen la operación.

  Salida: valores proporcionados, luego de la operación, al usuario.

  Postcondiciones: indican las condiciones que deberán cumplirse una vez ejecutada la operación.

La lista de operaciones debe ser lo más completa posible. En el momento de definir el ADT debemos procurar realizar la abstracción de manera que se contemplen todas las operaciones que, incluso a futuro, podrían requerirse sobre ese ADT. El tipo abstracto de datos se debe ver como una *caja negra* que encierra todo lo relacionado al concepto que está describiendo.

La mayoría de los ADT tiene una operación especial, llamada ***inicializador***, que asigna valores iniciales a los datos. Cuando el ADT se implementa por medio de una clase en un lenguaje de programación, esta operación recibe un nombre según el lenguaje empleado. En el lenguaje ***C++*** recibe el nombre de ***constructor***. Al momento de declarar un objeto, esta operación lo crea e inicializa.

Considerando lo anterior, un ADT tendrá el siguiente formato:

```
ADT Nombre:
   Datos
      Describe los datos (y la estructura de los mismos) que caracterizan
      al objeto.
   Operaciones
      Inicializador (constructor):
         Valores Iniciales: Datos que se utilizarán para darle un valor
         inicial al objeto (instancia del ADT).
         Proceso: Inicializa el objeto al ser creado.
      Operación1:
         Entrada: La proporciona el usuario.
         Precondiciones: Estado del sistema antes de ejecutar la
         operación1.
         Proceso: Acciones ejecutadas con los datos.
         Salida: Valores generados por el proceso.
         Postcondiciones: Estado del sistema luego de ejecutar la
         operación1.
      Operación2:
         Entrada: La proporciona el usuario.
         Precondiciones: Estado del sistema antes de ejecutar la
         operación2.
         Proceso: Acciones ejecutadas con los datos.
         Salida: Valores generados por el proceso.
         Postcondiciones: Estado del sistema luego de ejecutar la
         operación2.
         …

Operación N:
         Entrada: La proporciona el usuario.
         Precondiciones: Estado del sistema antes de ejecutar la
         operación N.
         Proceso: Acciones ejecutadas con los datos.
         Salida: Valores generados por el proceso.
         Postcondiciones: Estado del sistema luego de ejecutar la
         operación N.

Fin ADT Nombre
```

A continuación se presenta el ADT correspondiente a la definición de un cuadrado.

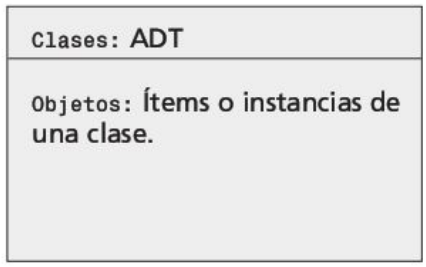
```
ADT Cuadrado:
Un cuadrado es una figura plana, cerrada por cuatro líneas rectas igua-
les que forman otros tantos ángulos rectos. Para el cálculo de la super-
ficie y del perímetro sólo se necesita conocer el tamaño del lado.

Datos:
   Un número real positivo que indica el lado del cuadrado.
Operaciones:
   Constructor:
      Entrada: Un número real positivo que representa el lado del cuadrado.
      Proceso: Asignar el valor al lado.
   Calcula-Superficie:
      Entrada: El valor del lado.
      Precondiciones: (en este caso no es necesario definir
      precondiciones).
      Proceso: Calcular la superficie del cuadrado.
      Salida: El valor de la superficie.
      Postcondiciones: (en este caso no es necesario definir
      postcondiciones).
   Calcula-Perímetro:
      Entrada: El valor del lado.
      Precondiciones: (en este caso no es necesario definir
      precondiciones).
      Proceso: Calcular el perímetro del cuadrado.
      Salida: El valor del perímetro.
      Postcondiciones: (en este caso no es necesario definir
      postcondiciones).
Fin ADT Cuadrado
```

En el ejemplo anterior, el ADT está formado por un atributo único que representa el lado del cuadrado y por tres métodos: el constructor, encargado de dar un valor inicial al lado del objeto *cuadrado*, y los métodos `Calcula-Superficie` y `Calcula-Perímetro` creados para calcular la superficie y el perímetro respectivamente.

1.4 Clases

Un ADT se representa por medio de ***clases***, utilizando las facilidades que ofrecen los lenguajes orientados a objetos. Una clase está formada por miembros: los atributos y los métodos. Los atributos representan las características relevantes del objeto/concepto descrito, mientras que los métodos representan las operaciones permitidas para almacenar/manipular los datos. Una variable de tipo clase

(o una instancia de una clase) se llama ***objeto***. Un objeto tiene datos (atributos) y comportamiento (métodos). Los objetos se crean durante la ejecución del programa.



Al definir una clase se pueden establecer diferentes grados de seguridad para sus miembros, determinando de esta manera los posibles usuarios de los mismos. Las tres categorías de seguridad que maneja ***C++*** son: privada, protegida y pública.

- ***Privada:*** Generalmente se utiliza para definir los atributos y, en casos muy especiales, algunos de los métodos de la clase. Los miembros definidos en esta sección sólo se pueden acceder por miembros de la misma clase. La privacidad permite garantizar la protección de los atributos y métodos definidos en esta sección.
- ***Protegida:*** En esta sección se definen los atributos y métodos que se comparten con las clases derivadas (ver herencia, capítulo 2). Es decir, los miembros de la clase definidos en una sección protegida pueden ser accesados solamente por miembros de la misma clase y de sus clases derivadas.
- ***Pública:*** En esta sección se definen los atributos y métodos que estarán disponibles para cualquier cliente. Además, se permite la interacción con el exterior y, que los clientes manipulen objetos del tipo de la clase, sin tener por ello que conocer la estructura interna del objeto ni los detalles de implementación.

Retomando el ejemplo del ADT usado para definir el concepto cuadrado, la sección privada se podría usar para definir el atributo lado, mientras que en la pública se definirían el constructor y los métodos usados para el cálculo de la superficie y del perímetro. De esta forma, el lado sólo se puede acceder a través de los métodos definidos dentro de la clase cuadrado, mientras que los métodos se pueden invocar desde el exterior.

1.4.1 Clases abstractas y concretas

Las clases, según el concepto que estén definiendo, se pueden clasificar en ***abstractas*** o ***concretas.*** Las primeras se usan para definir conceptos generales en los cuales no interesa mencionar detalles específicos, sólo características o atributos generales y por lo tanto compartibles. Estas clases no se usan directamente en la solución del problema, sino que son importantes para abstraer y generalizar la solución. Es decir, son clases útiles para modelar los datos en la etapa de diseño de las soluciones. A partir de las mismas se definen las clases concretas.

Las clases concretas, por otra parte, se utilizan para indicar conceptos más específicos, que se podrán emplear, tal vez directamente, en la solución de un problema. A continuación se presenta un ejemplo de una clase abstracta y de tres clases concretas que se derivan de la primera.

Clase Abstracta: Medio de transporte
Esta clase tendrá ciertas características o atributos que serán comunes a todos los medios de transporte (por ejemplo: tracción, fuerza, etcétera), aunque no será lo suficientemente específica como para que pueda emplearse para definir objetos.

Clases Concretas: Automóviles, Barcos, Aviones
Estas clases heredarán de la clase abstracta *Medio de transporte* sus características, y además tendrán un conjunto de atributos propios que permitirán definir de manera más específica los conceptos automóviles, barcos o aviones según sea el caso. En la aplicación se tendrán objetos de estas clases concretas para emplearse en la solución de los problemas.

1.4.2 Definición de una clase en C++

La definición de una clase en ***C++*** comienza con la palabra reservada `class`[1] seguida del nombre de la clase. El nombre elegido debe hacer referencia al concepto representado. La clase puede tener tres tipos diferentes de secciones: privada, protegida y pública. En cada una de estas secciones se podrán definir atributos y/o métodos de la clase, y la sección determinará el tipo de acceso que se podrá tener sobre los miembros ahí definidos. Así, los atributos o métodos definidos en la sección privada (`private`) estarán disponibles sólo para los miembros de la misma clase; los que se definan en la sección protegida (`protected`) sólo podrán ser utilizados por los miembros de la clase y por los de sus clases derivadas; y por último, los atributos o métodos definidos en la sección pública (`public`) estarán disponibles para los miembros de la clase, para los de sus clases derivadas y para cualquier cliente. Una clase puede tener las tres secciones o cualquier combinación de las mismas.

A continuación se presenta la sintaxis usada para la definición de una clase.

```
class NombreClase
{
   private:
       atributos y/o métodos;
   protected:
       atributos y/o métodos;
   public:
       atributos y/o métodos;
};
```

Para declarar un objeto del tipo de la clase previamente definida, se emplea la siguiente sintaxis:

```
NombreClase NombreObjeto;
```

[1] Para mayor claridad utilizamos **negritas** para indicar palabras reservadas, propias del lenguaje de programación utilizado.

En el programa 1.1 se presenta un ejemplo de declaración de una clase en ***C++***. La misma está formada por una sección única, en este caso pública, en la cual se declaran los atributos y un método.

Programa 1.1

```
/* La clase Persona queda definida por los atributos: Nombre, Domicilio
➥y Edad y un método ActualizaDomicilio que permite cambiar el domicilio
➥de una persona. */

class Persona
{
   public:
      char Nombre[64], Domicilio[64];
      int Edad;
      void ActualizaDomicilio (char NuevoDom[]);
};

/* En la función main se declaran dos objetos de la clase Persona. Estos
➥objetos son las variables que se usarán en la solución del problema. */
void main()
{
   Persona ObjJefe, ObjGerente;
   ...
}
```

En los siguientes ejemplos se muestra la declaración y el uso de clases a través del lenguaje de programación ***C++***. Cabe aclarar que, por razones de espacio, no se incluyeron programas completos. Éstos no llevan a cabo la inicialización de los objetos (misma que se analizará en la sección 1.4.3, y trata sobre el método constructor) y no contienen las bibliotecas requeridas por ***C++*** para: lecturas/escrituras, funciones matemáticas, manejo de cadenas, ni la función `main`. Esta aclaración es válida para casi todos los ejemplos del libro.

Programa 1.2

```
/* La clase Punto contiene como atributos privados las coordenadas en
➥el eje de las X's y de las Y's, lo cual garantiza mayor seguridad en el
➥manejo de los mismos. Además, en la sección pública se han definido
➥métodos para accesar, modificar e imprimir los atributos privados. */
class Punto
```

```
{
   private:
       float CoordenadaX, CoordenadaY;
   public:
       float ObtenerCoordX();
       float ObtenerCoordY();
       void ModificaX(float NuevaX);
       void ModificaY(float NuevaY);
       void ImprimeCoordenadas();
};

/* Método que permite, a los usuarios externos a la clase, conocer el
➥valor de la coordenada X. */
float Punto::ObtenerCoordX()
{
   return CoordenadaX;
}

/* Método que permite, a los usuarios externos a la clase, conocer el
➥valor de la coordenada Y. */
float Punto::ObtenerCoordY()
{
   return CoordenadaY;
}

/* Método que permite actualizar el valor de la coordenada X. */
void Punto::ModificaX(float NuevaX)
{
   CoordenadaX= NuevaX;
}

/* Método que permite actualizar el valor de la coordenada Y. */
void Punto::ModificaY(float NuevaY)
{
   CoordenadaY= NuevaY;
}

/* Método que despliega los valores de las coordenadas X y Y. */
void Punto::ImprimeCoordenadas()
{
   cout<< "Coordenada X: " << CoordenadaX << '\n';
   cout<< "Coordenada Y: " << CoordenadaY << '\n';
}

/* Función que usa la clase Punto: se declara un objeto tipo Punto y
➥a través de los métodos se modifican e imprimen las coordenadas del
➥punto. */
void UsaClasePunto()
```

```
{
   /* Declaración de un objeto usando la clase Punto. */
   Punto ObjPunto;
   float Auxiliar;
   …
   Auxiliar= 2.4;
   /* Modifica el valor de la coordenada X, asignándole el valor
   ➥almacenado en Auxiliar. */
   ObjPunto.ModificaX(Auxiliar);
   Auxiliar= 5.8;
   /* Modifica el valor de la coordenada Y, asignándole el valor
   ➥almacenado en Auxiliar. */
   ObjPunto.ModificaY(Auxiliar);
   /* Imprime el valor de las coordenadas del punto. */
   ObjPunto.ImprimeCoordenadas();
   …
   /* Obtiene e imprime el valor de las coordenadas X y Y del punto. */
   Auxiliar= ObjPunto.ObtenerCoordX();
   cout<< "\nLa coordenada X es: " << Auxiliar;
   Auxiliar= ObjPunto.ObtenerCoordY();
   cout<< "\nLa coordenada Y es: " << Auxiliar;
}
```

Observe que en el programa 1.2, en la declaración de cada uno de los métodos, se usó la siguiente sintaxis para el encabezado de los mismos:

```
NombreClase::NombreMétodo
```

Los dobles dos puntos (`::`) indican que el método pertenece a la clase. Por ejemplo: `Punto::ModificaY` expresa que el método `ModificaY` es de la clase `Punto`.

Por otra parte, cuando a través de un objeto se invoca a un método, la sintaxis que debe seguirse es:

```
NombreObjeto.NombreMétodo
```

Se utiliza un punto (`.`) para indicar que un método o atributo pertenece a un objeto. Por ejemplo: `ObjPunto.ImprimeCoordenadas()` expresa que el método `ImprimeCoordenadas()` está asociado al objeto `ObjPunto`. Por lo tanto:

```
NombreClase::NombreMetodo
NombreClase::Atributo

NombreObjeto.NombreMetodo
NombreObjeto.Atributo
```

El programa 1.3 presenta la definición de la clase `Triangulo` y su uso por medio de una función sencilla.

Programa 1.3

```
/* La clase Triangulo define un triángulo por medio de la longitud de su
➥base y de su altura. Además, contiene un método para calcular su área y
➥otro para imprimir sus atributos. */
class Triangulo
{
   public:
      float Base, Altura;
      float CalculaArea();
      void ImprimeAtributos();
};

/* Método que calcula el área de un triángulo y regresa un número real
➥como resultado. */
float Triangulo::CalculaArea()
{
  return (Base * Altura / 2 );
}

/* Método que imprime el valor de la base y de la altura de un
➥triángulo. */
void Triangulo::ImprimeAtributos()
{
   cout<< "Base:  " << Base << '\n';
   cout<< "Altura: " << Altura << '\n';
}

/* Función que usa la clase Triangulo: declara un objeto tipo Triangulo
➥y a través de los métodos imprime la base y la altura del triángulo y
➥calcula e imprime su área. */
void UsaClaseTriangulo()
```

```
{
/* Declaración de un objeto de tipo Triangulo. */
   Triangulo ObjTriang;
   float Area;
   …
   ObjTriang.ImprimeAtributos();
   …
   Area= ObjTriang.CalculaArea();
   cout<< "Área del triángulo: "<<Area;
   …
}
```

En el siguiente ejemplo se define la clase `Cliente` con ciertos atributos y algunos métodos. Asimismo, se incluye una función que hace uso de la clase.

Programa 1.4

```
/* La clase Cliente define a un cliente de banco. Se tienen los
➥atributos privados: Nombre, Direccion, Telefono, Saldo, TipoDeCuenta y
➥NumDeCuenta. Además, en la sección pública de la clase, se incluyeron
➥los métodos necesarios para imprimir los atributos de un cliente,
➥obtener su saldo, obtener el tipo de cuenta, hacer un retiro y un
➥depósito a la cuenta. */
class Cliente
{
   private:
       char Nombre[64], Direccion[64], Telefono[8];
       float Saldo;
       int TipoDeCuenta, NumDeCuenta;
   public:
       void ImprimeDatos();
       float ObtenerSaldo();
       int ObtenerTipoCta();
       int HacerRetiro(float);
       void HacerDeposito(float);
};

/* Método que despliega los datos de un cliente. */
void Cliente::ImprimeDatos()
{
   cout<< "Nombre: " << Nombre << '\n';
   cout<< "Dirección: " << Direccion << '\n';
   cout<< "Teléfono: " << Telefono << '\n';
   cout<< "Saldo: " << Saldo << '\n';
```

```
   cout<< "Tipo de Cuenta: " << TipoDeCuenta << '\n';
   cout<< "Número de Cuenta: " << NumDeCuenta << '\n';
}

/* Método que permite, a usuarios externos a la clase, conocer el saldo
➥de un cliente. */
float Cliente::ObtenerSaldo()
{
   return Saldo;
}

/* Método que permite, a usuarios externos a la clase, conocer el tipo
➥de cuenta de un cliente. */
int Cliente::ObtenerTipoCta ()
{
   return TipoDeCuenta;
}

/* Método que registra un retiro en la cuenta de un cliente. */
int Cliente::HacerRetiro (float Monto)
{
   int Respuesta= 1;
   /* Verifica que haya dinero suficiente en la cuenta. */
   if ((Saldo-Monto) < 0)
      Respuesta= 0;
   else
       Saldo= Saldo - Monto;
   return Respuesta;
}

/* Método que registra un depósito en la cuenta de un cliente. */
void Cliente::HacerDeposito (float Monto)
{
   Saldo= Saldo + Monto;
}


/* Función que usa la clase Cliente: se declaran dos objetos tipo
➥Cliente y por medio de los métodos se registran retiros y depósitos en
➥sus cuentas. */
void UsaClaseCliente()
{
   float SaldoCli;
   /* Declaración de dos objetos de la clase Cliente. */
   Cliente ObjClien1, ObjClien2;
   …
```

```
    /* Se obtiene el saldo del cliente, asumiendo que previamente le fue
    ➥asignado un valor. */
    SaldoCli= ObjClien2.ObtenerSaldo();
    cout<< "El saldo del cliente es: " << SaldoCli << '\n';

    /* Se hace un retiro de la cuenta de cheques de un cliente: se
    ➥verifica que tenga una cuenta de cheques (1), en cuyo caso se
    ➥efectúa el retiro.  */
    if (ObjClien1.ObtenerTipoCta() == 1)
        if (ObjClien1.HacerRetiro(1500))
           cout<<"\nRetiro realizado con éxito. Cuenta actualizada. \n";
        else
           cout<<"\nNo tiene saldo suficiente para realizar ese retiro. \n";
    else
        cout << "\n Para realizar un retiro debe ser una cuenta de
        ➥cheques.\n";

    /* Se hace un depósito en la cuenta de un cliente: se registra el
    ➥nuevo saldo. */
    ObjClien2.HacerDeposito(50000.00);
}
```

1.4.3 Los métodos constructor y destructor

El ***método constructor*** es una función que se ejecuta automáticamente al declarar un objeto como instancia de una clase; se escribe generalmente en la sección pública de una clase, y su función es crear e iniciar un objeto del tipo de la clase en la cual fue definido. De esta manera, los constructores permiten asegurar que los objetos, al crearse, se inicialicen con valores válidos. Un constructor no se hereda ni puede retornar un valor, y tiene el mismo nombre que la clase.

Por su parte, el ***método destructor*** es una función que se ejecuta automáticamente al destruirse un objeto. Lleva el mismo nombre que la clase, va precedido por el símbolo ~ y no lleva argumentos. Un objeto se destruye al terminar el programa en el cual se creó y libera el espacio de memoria. En el caso de objetos locales, éstos se destruyen al dejar la sección en la cual se crearon.

En el programa 1.5 se presenta la definición de la clase `Persona` en la cual se incluyeron un método constructor y uno destructor.

Programa 1.5

```
/* Se define la clase Persona en la cual, además de los atributos, se
➥incluyen tres métodos en la sección pública: un constructor, un
➥destructor y uno para imprimir los datos. */
class Persona
{
   private:
        char Nombre[64];
        int Edad;
   public:
        /* Método constructor: se llama igual que la clase, no da ningún
        ➥tipo de resultado. */
        Persona(char *, int);
        /* Método destructor: se llama igual que la clase, va precedido
        ➥por ~ y no tiene argumentos. */
        ~Persona();
        void ImprimeDatos();
};

/* Declaración del método constructor: tiene 2 parámetros (Nom y Ed)
que se usarán para dar un valor inicial a los atributos (Nombre y Edad
respectivamente), al momento de crearse un objeto. */
Persona::Persona(char *Nom, int Ed)
{
   strcpy (Nombre, Nom);
   Edad= Ed;
}
```

1.4.4 Uso de constructores múltiples

Los constructores múltiples hacen referencia a que en una misma clase se puede incluir más de un constructor. Esto permite dar mayor flexibilidad a la declaración de la clase. Existen tres tipos de constructores:

- ***Constructor por omisión:*** es aquel que no tiene parámetros y su cuerpo no contiene instrucciones. Cuando se crea un objeto, si éste es global, el constructor inicializa con cero a aquellos atributos que son numéricos y con `NULL` a los que son alfabéticos. Si el objeto creado es local, entonces los atributos se inicializan con valores indeterminados.
- ***Constructor con parámetros:*** es aquel que tiene una lista de parámetros, a los cuales habrá que darles un valor en el momento de declarar un objeto. Dichos valores se usarán para instanciar los atributos del objeto creado.

- ***Constructor con parámetros por omisión*:** es aquel que tiene una lista de parámetros, a los cuales se les asigna un valor por omisión que se usará para inicializar en caso de que no se den explícitamente otros valores.

Todos los constructores llevan el nombre de la clase a la cual pertenecen. La existencia o no de parámetros decide a qué tipo de constructor se está llamando. Es importante mencionar que los constructores por omisión y con parámetros por omisión no pueden convivir en una misma clase, ya que resultaría ambiguo a cuál se estaría invocando en el caso de no proporcionar parámetros.

A continuación se presenta la declaración de la clase `Fecha` que tiene dos constructores: uno por omisión y otro con parámetros.

Programa 1.6

```
/* Se define la clase Fecha con los atributos Día, Mes y Año. Se incluye
➥un constructor por omisión (sin parámetros) y uno con parámetros. Este
➥último permite dar un valor inicial a los atributos cuando se crea un
➥objeto. */
class Fecha
{
   private:
       int Dia, Mes, Anio;
   public:
       Fecha ();
       Fecha (int, int, int);
…
};

/* Declaración del método constructor por omisión. */
Fecha::Fecha ()
{}

/* Declaración del método constructor con parámetros (tres enteros):
➥inicializa los atributos. */
Fecha::Fecha (int D, int M, int A)
{
   Dia= D;
   Mes= M;
   Anio= A;
}

/* Función que utiliza la clase Fecha: se crean objetos usando los dos
➥constructores. */
void UsaConstructores ()
```

```
{
   Fecha ObjFecha;
   /* En este caso se invoca al constructor por omisión. El objeto
   ➥ObjFecha tendrá sus atributos (Día, Mes y Año) con valores
   ➥indeterminados. */
    …
   Fecha Cumpleanios (18, 05, 2006);
   /* En este caso se invoca al constructor con parámetros. Al objeto
   ➥Cumpleanios se le asignarán los valores 18, 05 y 2006 para sus
   ➥atributos Día, Mes y Año respectivamente. */
  ...
}
```

En el programa 1.7 se presenta la declaración de la clase `Fecha`, incluyendo ahora un constructor con parámetros por omisión.

Programa 1.7

```
/* Se define la clase Fecha con los atributos Día, Mes y Año. Se incluye
➥un constructor con parámetros por omisión. Estos valores se asignarán a
➥los atributos en caso de que el usuario no proporcione otros valores. */
class Fecha
{
   private:
       int Dia, Mes, Anio;
   public:
       Fecha (int D= 0, int M= 0, int A= 0);
…
};

/* Declaración del constructor con parámetros por omisión. */
Fecha::Fecha (int D, int M, int A)
{
   Dia= D;
   Mes= M;
   Anio= A;
}
```

Una manera equivalente de escribir las asignaciones que aparecen en el constructor es:

```
Fecha::Fecha (int D, int M, int A): Dia(D), Mes(M), Anio(A)
{}
```

En el programa 1.8 se retoma el programa 1.3, pero ahora incluyendo, en la sección pública dos constructores, uno por omisión y otro con parámetros.

Programa 1.8

```
/* La clase Triangulo define un triángulo por medio de la longitud de
➥su base y de su altura. Además, contiene métodos para calcular su área,
➥actualizar e imprimir sus atributos. Para la clase Triangulo se
➥definieron dos constructores: uno por omisión y otro con parámetros. */
class Triangulo
{
   private:
       float Base, Altura;
   public:
       Triangulo();
       Triangulo(float, float);
       float CalculaArea();
       void ImprimeAtributos();
       void ActualizaAtributos(float, float);
};

/* Declaración del método constructor por omisión. */
Triangulo::Triangulo()
{}

/* Declaración del método constructor con parámetros. */
Triangulo::Triangulo(float B, float A)
{
   Altura= A;
   Base= B;
}

/* Método que calcula el área de un triángulo. Regresa un número real. */
float Triangulo::CalculaArea()
{
   return (Base * Altura / 2);
}

/* Método que despliega los valores de los atributos. */
void Triangulo::ImprimeAtributos()
{
   cout<< "Base: " << Base << '\n';
   cout<< "Altura: " << Altura << '\n';
}

/* Método que modifica los valores de la base y de la altura de un
➥triángulo. */
void Triangulo::ActualizaAtributos (float B, float A)
```

```
{
   Altura= A;
   Base= B;
}

/* Función que usa la clase Triangulo: se declaran objetos utilizando
➥los dos tipos de constructores incluidos en la clase. */
void UsaClaseTriangulo()
{
   float ValorAlt, ValorBase;
   /* Se declara un objeto haciendo uso del constructor por omisión. En
   ➥este caso la base y la altura permanecen con valores indefinidos. */
   Triangulo ObjT1;
   ValorBase= 2.6;
   ValorAlt= 3.7;
   /* Se le asignan valores a la base y a la altura del triángulo. */
   ObjT1.ActualizaAtributos(ValorBase, ValorAlt);
   ...
   /* Se declara un objeto haciendo uso del constructor con parámetros. En
   ➥este caso se le asigna a la base el valor 2.8 y a la altura 9.0. */
   Triangulo ObjT2 (2.8, 9.0);
   ...
   ObjT1.ImprimeAtributos();
   ObjT2.ImprimeAtributos();
   ...
}
```

Por último, en el programa 1.9 se presenta la clase `Cliente` (ver programa 1.4) en la cual se definieron dos constructores.

Programa 1.9

```
/* La clase Cliente define un cliente por medio de los atributos:
➥Nombre, Dirección, Teléfono, Saldo, Tipo de Cuenta y Número de Cuenta,
➥y de los métodos que permiten el manejo de ellos. Para la clase Cliente
➥se definieron dos constructores: uno con parámetros para algunos de los
➥atributos y otro con parámetros por omisión. */
class Cliente
{
   private:
       char Nombre[64], Direccion[32], Telefono[10];
       float Saldo;
       int TipoDeCuenta, NumDeCuenta;
```

```
    public:
        Cliente(char Nom[], char Tel[], float Sal);
        Cliente(char Nom[], char Dir[], char Tel[], float Sal= 0,int
        ➥TC= 1,int NoC= 0);
        float ObtenerSaldo();
        void ImprimeDatos();
        char ObtenerTipoCta();
        void HacerRetiro(float Monto);
        void HacerDeposito(float Monto);
};

/* Declaración del método constructor con parámetros. Se asignan valores
➥a los atributos, tomando los que aparecen en el prototipo del construc-
➥tor si el usuario no proporciona otros. */
Cliente::Cliente(char Nom[],char Dir[], char Tel[], float Sal, int TC,
➥int NoC)
{
   strcpy(Nombre, Nom);
   strcpy(Direccion, Dir);
   strcpy(Telefono, Tel);
   Saldo= Sal;
   TipoDeCuenta= TC;
   NumDeCuenta= NoC;
}

/* Declaración del método constructor donde se asignan, por medio de
➥los parámetros, valores a algunos de los atributos y a otros se les dan
➥valores por omisión. */
Cliente::Cliente(char Nom[], char Tel[], float Sal)
{
   strcpy(Nombre, Nom);
   strcpy(Telefono, Tel);
   Saldo= Sal;
   strcpy(Direccion, "Desconocida");
   TipoDeCuenta= 0;
   NumDeCuenta= -1;
}

/* Método que permite conocer el Saldo de un cliente. */
float Cliente::ObtenerSaldo()
{
   return Saldo;
}

/* Método que despliega en pantalla los valores de los atributos de un
➥cliente. */
void Cliente::ImprimeDatos()
```

```
{
   cout<< "Nombre: " << Nombre << '\n';
   cout<< "Dirección: " << Direccion << '\n';
   cout<< "Teléfono: " << Telefono << '\n';
   cout<< "Saldo: " << Saldo << '\n';
   cout<< "Tipo de Cuenta: " << TipoDeCuenta << '\n';
   cout<< "Número de Cuenta: " << NumDeCuenta << '\n';
}

...

/* Función que muestra el uso de los dos tipos de constructores. */
void UsaClaseCliente()
{
   /* Se crean dos objetos de tipo Cliente usando los constructores
   ➥definidos. */
   Cliente ObjCli1("Laura", "Insurgentes No. 2", "55559900", 28000, 2,
2);
   Cliente ObjCli2("Juan", "55408881", 4000);
   /* En el último objeto creado se dan valores para 3 de sus atributos,
   ➥por lo tanto el constructor asigna a los restantes los dados por
   ➥omisión. */

   /* Se imprimen los datos de cada cliente. */
   ObjCli1.ImprimeDatos();
   ObjCli2.ImprimeDatos();
}
```

En los siguientes ejemplos se presentan métodos destructores. El programa 1.10 retoma la clase `Fecha`, del programa 1.7, pero ahora incluye un método destructor para la misma.

Programa 1.10

```
/* Se define la clase Fecha en la cual se incluyó un método constructor
➥y uno destructor.  */
class Fecha
{
   private:
       int Dia, Mes, Anio;
   public:
       Fecha (int, int, int);
       ~Fecha();
};
```

```
/* Declaración del método constructor con parámetros. */
Fecha::Fecha(int D, int M, int A)
{
   Dia= D;
   Mes= M;
   Anio= A;
}

/* Declaración del método destructor. El cuerpo del método está
➥vacío. */
Fecha::~Fecha()
{ }
```

En el programa 1.11 se define la clase `Texto` que tiene un constructor y un destructor.

Programa 1.11

```
/* Se define la clase Texto por medio de los atributos privados que
➥representan la longitud del texto y la estructura requerida para
➥almacenar los caracteres. Asimismo, se incluyen algunos métodos. */
class Texto
{
   private:
       char *CadenaTexto;
       int Longitud;
   public:
       Texto(char *);
       ~Texto();
       void ImprimeTexto();
};

/* Declaración del método constructor con parámetros. */
Texto::Texto(char *Cad)
{
   /* Genera dinámicamente el espacio de memoria necesario para almacenar
   ➥la cadena Cad más un carácter adicional (carácter nulo). */
   CadenaTexto= new char[strlen(Cad)+1];
   /* Se verifica si se pudo generar el espacio requerido. */
   if (CadenaTexto)
   {
       strcpy(CadenaTexto, Cad);
       Longitud= strlen(CadenaTexto);
   }
```

```
   else
       Longitud= 0;
}

/* Declaración del método destructor. Verifica que la longitud de la
➥cadena sea distinta de cero. Libera el espacio de memoria empleado
➥por CadenaTexto. */
Texto::~Texto()
{
   if (Longitud)
       delete[] CadenaTexto;
}

/* Método para imprimir el texto. */
void Texto::ImprimeTexto ()
{
   cout<< "La cadena es: " << CadenaTexto << endl;
   cout<< "Su longitud es: " << Longitud << endl;
}


/* Función que utiliza la clase Texto: se crea un objeto usando el
➥constructor con parámetros e imprime su valor. Al terminar la función
➥el objeto se destruye liberando espacio de memoria. */
void UsaTexto ()
{
   Texto ObjTexto("Cadena de longitud 41, incluyendo blancos");
   ObjTexto.ImprimeTexto();
}
```

Ejercicios

1. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos.

```
class Flor
{
   private:
       char Nombre[64], Epoca[64];
   public:
       void Flor();
       Flor(char [], char []);
       void Imprime();
};
```

a) La definición de la clase es correcta.

b) En el prototipo del método constructor por omisión hay un error. Justifique su respuesta.

c) Al prototipo del método `Imprime` le faltan parámetros. Justifique su respuesta.

2. Retome el problema anterior. Se definen los métodos de la clase `Flor` y la función `main`. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos.

```
Flor::Flor()
{}

Flor::Flor(char Nom[], char Epo[])
{
   strcpy(Nombre, Nom);
   strcpy(Epoca, Epo);
}

void Flor::Imprime()
{
   cout<<"\n\nNombre de la flor: "<<Nombre;
   cout<<"\nEpoca en la que se cosecha: "<<Epoca<<"\n\n";
}

void main()
{
   Flor Rosa("Rosa aterciopelada", "verano"), Jazmin;
   cout<<"\nIngrese época en la que se cosecha el jazmín";
   cin>>Jazmin.Epoca;
   Rosa.Imprime();
}
```

a) Al declarar el objeto `Jazmin` se deben dar parámetros. Justifique su respuesta.

b) Es incorrecto leer el atributo `Epoca` en la función principal. Justifique su respuesta.

c) El método `Imprime` se invoca correctamente desde la función principal. Justifique su respuesta.

3. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos.

```
class Gato
{
   private:
       char Nombre[64];
       int Edad;
   public:
       char MarcaAlimento[64];
       Gato(char [], int, char[]);
       void Imprime();
       char* RegresaNombre();
       void CambiaEdad(int);
};
```

a) El atributo `MarcaAlimento` no se puede declarar en la sección pública. Justifique su respuesta.

b) La clase `Gato` necesita un método constructor por omisión. Justifique su respuesta.

c) La clase `Gato` está correctamente definida.

4. Retome el problema anterior. Se definen los métodos de la clase `Gato` y la función `main`. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos.

```
Gato::Gato(char Nom[], int Ed, char MAlim[])
{
   strcpy(Nombre, Nom);
   Edad= Ed;
   strcpy(MarcaAlimento, MAlim);
}

void Gato::Imprime()
{
   cout<<"\n\nNombre del gato: "<<Nombre;
   cout<<"\nEdad: "<<Edad;
   cout<<"\nMarca del alimento que come: "<<MarcaAlimento<<"\n\n";
}

char* Gato::RegresaNombre()
{
   return Nombre;
}

void Gato::CambiaEdad(int NuevaE)
```

```
{
   Edad= NuevaE;
}

void main()
{
   Gato MiGato("Michifus", 3, "SaborYNutricion"), TuGato();
   MiGato::CambiaEdad(4);
   cout<<"\n\nNombre del gato: "<<MiGato.RegresaNombre()<<"\n\n";
   cout<<"\nAlimento que come: "<<MiGato.MarcaAlimento<<"\n\n";
   cout<<"\nEdad: "<<MiGato.Edad;
}
```

a) La declaración del objeto `MiGato` es correcta. Justifique su respuesta.

b) La declaración del objeto `TuGato` es incorrecta. Justifique su respuesta.

c) La impresión del valor del atributo `Nombre` del objeto `MiGato` es correcta. Justifique su respuesta.

d) La impresión del valor del atributo `MarcaAlimento` del objeto `MiGato` es correcta. Justifique su respuesta.

e) La impresión del valor del atributo `Edad` del objeto `MiGato` es correcta. Justifique su respuesta.

f) El método `CambiaEdad` está incorrectamente asociado al objeto `MiGato`.

5. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos. Las afirmaciones tienen relación con el segmento del programa al cual suceden.

```
class Trabajador
{
   private:
       char Nombre[64];
       int ClaveTrab, Sindi;
       float Sueldo;
   public:
       Trabajador(char [], int, int, float);
       void Imprime();
       void CambiaEstado();
       void AumentaSueldo(float);
};
```

```
Trabajador::Trabajador(char Nom[],int Cla=1000,int Si=1,float Sue=1600)
{
   strcpy(Nombre, Nom);
   ClaveTrab= Cla;
   Sindi= Si;
   Sueldo= Sue;
}
```

a) En la declaración de la clase no se pudo incluir un método constructor por omisión. Justifique su respuesta.

b) En los parámetros formales del método constructor faltó darle un valor por omisión al parámetro `Nom`. Justifique su respuesta.

c) Los valores que aparecen en el encabezado del método siempre se asignan. Justifique su respuesta.

6. Retome el problema anterior. Se definen tres métodos de la clase `Trabajador` y la función `main`. Analice cuidadosamente las siguientes declaraciones y diga si los enunciados que aparecen después del código son verdaderos o falsos.

```
void Trabajador::Imprime()
{
   cout<<"\n\nNombre del trabajador: "<<Nombre;
   cout<<"\nClave: "<<ClaveTrab;
    if (Sindi)
        cout<<"\nEstá sindicalizado.";
   cout<<"\nSueldo: "<<Sueldo<<"\n\n";
}

void Trabajador::CambiaEstado()
{
   Sindi= !Sindi;
}

void Trabajador::AumentaSueldo(float Aumento )
{
   Trabajador::Sueldo= Trabajador::Sueldo * (1 + Aumento);
}

void main()
{
   Trabajador Pepe("Jose Pérez");
   Pepe.Imprime();
```

```
    Trabajador Carlos("Carlos González", 1050, 0);
    Carlos.Imprime();
    Carlos.CambiaEstado();
    Carlos.Imprime();
    Trabajador Paco("Francisco Quiroz", 2200, 1, 5680.25);
    Paco.Imprime();
    Paco.AumentaSueldo(0.10);
    Paco.Imprime();
}
```

a) Cuando se declara el objeto `Pepe` se deben dar valores para todos los parámetros. Justifique su respuesta.

b) Cuando se declara el objeto `Carlos` se pudo dar valores sólo a los atributos `Nombre` y `Sueldo`. Justifique su respuesta.

c) Se dieron demasiados valores al declarar el objeto `Paco`. Justifique su respuesta.

d) En el método `AumentaSueldo` es incorrecto usar `Trabajador::Sueldo`. Justifique su respuesta.

7. Considerando los enunciados de la función `main` del problema anterior, diga qué valores aparecerán en la pantalla al ejecutarse las siguientes instrucciones.

   a) Pepe.Imprime();

   b) Carlos.Imprime(); //Primera invocación del método.

   c) Carlos.Imprime(); //Segunda invocación del método.

   d) Paco.Imprime(); //Primera invocación del método.

   e) Paco.Imprime(); //Segunda invocación del método.

8. Retome la clase del ejercicio 6, modifique el método que aumenta el sueldo del trabajador. Ahora, el método debe recibir como parámetro el porcentaje de aumento, el número de horas extra trabajadas y el valor a pagar por cada hora extra. Los dos últimos parámetros deben tener un valor por omisión, de tal manera que cuando un trabajador no haya laborado horas extra, el usuario no tenga que asignarles 0.

9. Defina la clase `Rectangulo`. Determine los atributos y el conjunto de métodos (lo más completos posible) que caracterizan al concepto rectángulo.

10. Utilice la clase definida en el ejercicio 9 para declarar objetos que representan dos alfombras rectangulares a colocar en una oficina. Escriba un programa que solicite las dimensiones de cada una de las alfombras y del piso y utilice los métodos incluidos en la clase, para calcular e imprimir la superficie del piso que va a quedar cubierta.

11. Defina la clase `Persona`. Determine los atributos y el conjunto de métodos (lo más completos posible) que caracterizan al concepto persona. Luego declare el objeto `MiMaestra`, de tipo `Persona`. Escriba un programa en ***C++*** que utilice la clase previamente definida. El programa debe poder, por medio de los métodos incluidos en la clase, realizar las siguientes operaciones:

    a) Cambiar la dirección de `MiMaestra`. El usuario dará la nueva dirección.

    b) Cambiar el número de teléfono de `MiMaestra`. El usuario dará el nuevo número.

    c) Imprimir todos los datos de `MiMaestra`.

    d) Imprimir, si `MiMaestra` está casada, el nombre de su cónyuge.

12. Defina la clase `Mamífero` que contenga los atributos que caracterizan a un animal de este tipo, los métodos necesarios para el manejo de la información, así como diferentes constructores para crear e inicializar objetos de tipo `Mamífero`. Se sugiere definir constructores por omisión, con parámetros y/o con parámetros por omisión. Escriba un programa en ***C++*** que utilice la clase previamente definida. El programa debe poder realizar, por medio de los métodos incluidos en la clase, las siguientes operaciones:

    a) Declarar los objetos `Perro` y `Elefante`. Utilice los métodos constructores o algún método de lectura para darle valor a los atributos incluidos en la clase.

    b) Imprimir el tipo de alimentación del objeto `Elefante`. Analice alternativas de solución considerando que el atributo en cuestión sea privado o público.

    c) Imprimir los valores de todos los atributos del objeto `Perro`.

13. Defina la clase `Cubo`. Determine los atributos y el conjunto de métodos (lo más completos posible) que caracterizan al concepto cubo.

14. Retome el problema anterior y utilice la clase `Cubo` para definir cajas de cartón, en forma de cubo. Escriba un programa que calcule e imprima el total de pliegos de cartón que serán necesarios para fabricar un total de N ($1 \leq N \leq 20$) cajas. Las cajas pueden ser de diferentes tamaños. El programa

además de calcular el total de pliegos, deberá calcular el desperdicio de papel. Datos: N, tamaño de cada caja (considere 1/2 cm para pegar los diferentes lados de cada cara del cubo) y tamaño del pliego de cartón.

15. Defina la clase `Empleado`, según las especificaciones que se dan más abajo. Posteriormente, en un programa de aplicación, declare los objetos `JefePlanta` y `JefePersonal` usando la clase previamente definida. El programa debe permitir al usuario, por medio de menús:

    a) Cambiar el domicilio de uno de los dos empleados declarados. Los datos ingresados por el usuario serán la clave del empleado y su nuevo domicilio.

    b) Actualizar el sueldo de un empleado. Los datos ingresados por el usuario serán la clave del empleado y el porcentaje de incremento a aplicar al sueldo.

    c) Imprimir los datos de un empleado. El usuario proporcionará la clave del empleado elegido.

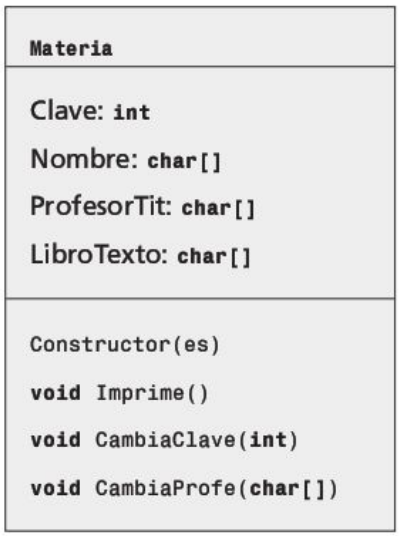
    d) Cambiar el nombre de la persona a quien reporta uno de los empleados.

16. Defina la clase `Materia`, según las especificaciones que se dan. Posteriormente, en un programa de aplicación declare los objetos `Programación` y `BasesDatos` usando la clase previamente definida. El programa debe permitir al usuario, por medio de menús:

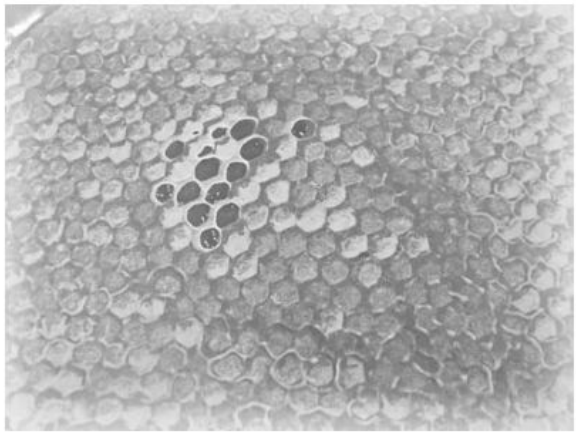
    a) Cambiar la clave de la materia `Programación`.

    b) Cambiar el nombre del maestro que imparte la materia `BasesDatos`.

    c) Imprimir todos los datos de la materia `BasesDatos`.



17. Retome la clase definida en el ejercicio anterior. ¿Qué método(s) debería agregarle/quitarle para que se pudiera imprimir, desde algún programa de aplicación, el nombre del libro de texto usado para la materia de `BasesDatos`?



CAPÍTULO 2

Herencia y amistad

La ***herencia*** es la capacidad de compartir atributos y métodos entre clases. La relación de herencia entre clases puede ser: privada, protegida o pública; la relación que se utiliza con mayor frecuencia es la ***pública***, por lo que la analizaremos detalladamente. Del tipo de herencia privada sólo se presentará una breve introducción.

La clase de la cual se hereda se denomina ***clase base*** o ***superclase***. Mientras que la clase que hereda se denomina ***clase derivada*** o ***subclase***.

Dependiendo del número de clases y de cómo se relacionen, la herencia puede ser: ***simple***, ***múltiple*** y ***de niveles múltiples***. En las siguientes secciones se explica cada una.

2.1 Herencia simple

Cuando sólo se tiene una clase base de la cual hereda la clase derivada, se dice que hay ***herencia simple*** (figura 2.1a). Sin embargo, la herencia simple no excluye la posibilidad de que de una misma clase base se pueda derivar más de una subclase o clase derivada (figura 2.1b).

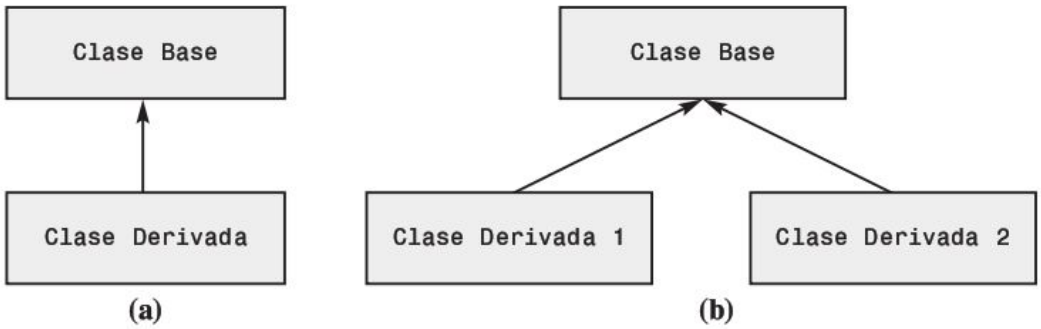


FIGURA 2.1 *Herencia simple*

Cuando se necesita representar un concepto general y a partir de éste, conceptos más específicos, resulta conveniente organizar la información usando herencia. Esto permite compartir atributos y métodos ya definidos, evita la duplicidad y, por otra parte, proporciona mayor claridad en la representación que se haga de la información. Es decir, se logra un mejor diseño de la solución del problema. Existen numerosos casos en los cuales se da este tipo de relación. En la figura 2.2 se presentan algunos ejemplos de herencia simple.

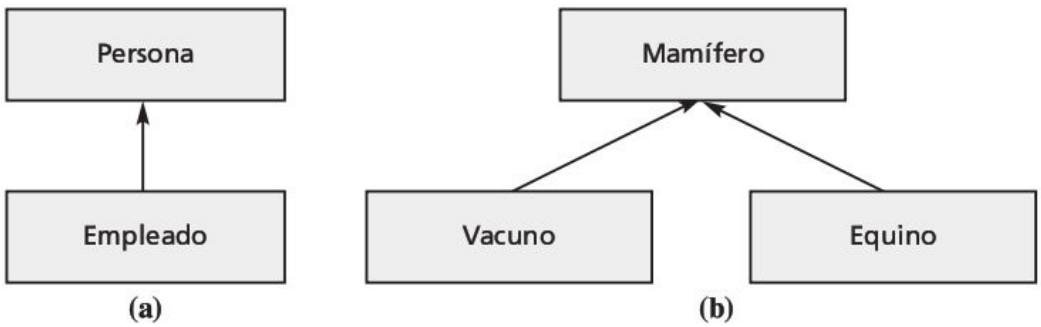


FIGURA 2.2 *Ejemplos de herencia simple*

En la figura 2.2a, la clase `Persona` es la clase base y `Empleado` es la clase derivada. Un objeto de esta clase también es un objeto de la clase `Persona`, por lo tanto tendrá los atributos y métodos de ambas clases. En la figura 2.2b, la clase `Mamífero` es la clase base y `Vacuno` y `Equino` son las clases derivadas. En este caso, se dice que todo `Vacuno` y todo `Equino` también son objetos de la clase `Mamífero` y en consecuencia tendrán todos los atributos y métodos que heredan de la clase base.

La herencia pública permite que los miembros privados de la clase base se puedan acceder sólo por medio de los métodos de dicha clase. Los miembros protegidos de la clase base podrán ser usados por los métodos de las clases derivadas, pero no por sus clientes. Los miembros públicos estarán disponibles para los métodos de las clases derivadas y para todos sus clientes.

```
class Base
{
   private:
   /* Miembros declarados en la sección privada: accesibles sólo para
   ➥miembros de esta clase.  */

   protected:
   /* Miembros declarados en la sección protegida: accesibles sólo para
   ➥miembros de esta clase y de sus derivadas.  */

   public:
   /* Miembros declarados en la sección pública: accesibles para todos. */
};
```

Para declarar una clase derivada de una clase previamente definida, se utiliza la siguiente sintaxis.

```
class Base
{

   /* Declaración de atributos y métodos de la clase Base. */

};
```

```
/* Relación de herencia pública entre las clases Base y Derivada. */
class Derivada: public Base
{

   /* Declaración de atributos y métodos de la clase Derivada. */

};
```

Con la palabra reservada **public** en el encabezado de la declaración de la clase `Derivada` se hace referencia a que dicha clase hereda los atributos y métodos de la clase `Base`. La declaración del constructor de la clase `Derivada` debe incluir un llamado al constructor de la clase `Base`. Para ello se sigue la sintaxis que se presenta a continuación:

```
Derivada::Derivada (parámetros): Base (parámetros propios de la clase
Base)
{
   /* Cuerpo del constructor de la clase Derivada. */

}
```

Cuando se declara un objeto del tipo de la clase derivada se invoca al constructor de ésta. De este constructor lo primero que se ejecuta es la llamada al constructor de la clase base, y posteriormente se ejecutan sus propias instrucciones. En cuanto a los parámetros, al invocar al constructor de la clase base se le deben proporcionar los parámetros que necesita para asignar valores a los atributos propios de la clase base y que la clase derivada hereda. En el cuerpo de la clase derivada se harán las asignaciones correspondientes a los atributos propios de esta clase.

El programa 2.1 presenta un ejemplo de herencia simple. Define la clase `Persona` y la clase `Empleado` como una clase derivada de la primera.

Programa 2.1

```
/* Se define la clase Persona formada por atributos protegidos y
➥públicos, y se usa como clase base para definir la clase Empleado. Los
➥objetos que sean del tipo Empleado tendrán los atributos de esta clase
➥(por ejemplo Salario), además de los atributos heredados de la clase
➥Persona. */
```

```
class Persona
{
   protected:
      char Nombre[30];
      int Edad;
   public:
      Persona (char *Nom, int Ed);
      void ImprimePersona();
};

/* Declaración del método constructor con parámetros. Da un valor inicial
➥a los atributos. */
Persona::Persona(char *Nom, int Ed)
{
   strcpy(Nombre, Nom);
   Edad = Ed;
}

/* Método que despliega los valores de los atributos de una persona. */
void Persona::ImprimePersona()
{
   cout<< "Nombre: " << Nombre << endl;
   cout<< "Edad: " << Edad << endl;
}


/* Definición de la clase Empleado como clase derivada de la clase
➥Persona. Se usa herencia pública. */
class Empleado: public Persona
{
   protected:
      float Salario;
   public:
      Empleado (char *Nom, int Ed, float Sal);
      void ImprimeEmpleado();
      ~Empleado();
};

/* Declaración del método constructor. Invoca al constructor de la clase
➥base. */
Empleado::Empleado(char *Nom, int Ed, float Sal): Persona(Nom, Ed)
{
   Salario= Sal;
}

/* Declaración del método destructor. */
Empleado::~Empleado()
{}
```

```
/* Método que imprime los valores de algunos de los atributos de un
➥empleado. */
void Empleado::ImprimeEmpleado()
{
   cout<< "Empleado: " << Nombre << endl;
   cout<< "Salario: " << Salario << endl;
}


/* Función que usa las clases Persona y Empleado: se declaran apuntadores
➥a objetos tipo Persona y Empleado. Por medio de los constructores
➥se les asignan valores a estos objetos, se imprimen y finalmente se
➥destruyen liberando la memoria. */
void UsaHerencia(void)
{
   Persona *ObjPersona= new Persona("Carlos", 22);
   Empleado *ObjEmpleado= new Empleado("Adriana", 25, 20000);

   ObjPersona->ImprimePersona();
   ObjEmpleado->ImprimeEmpleado();

   delete ObjPersona;
   delete ObjEmpleado;
}
```

En el ejemplo anterior, el método `ImprimeEmpleado` de la clase `Empleado` puede mostrar el atributo `Nombre` de la clase `Persona`, ya que el mismo es protegido. Por otra parte, es importante destacar que al crear un objeto de tipo `Empleado` se dan tres parámetros, los dos primeros son necesarios para instanciar los atributos heredados de la clase `Persona`.

2.2 Herencia múltiple

En el tipo de ***herencia múltiple*** se usan dos o más clases base para derivar una clase. Es decir, la clase derivada comparte los atributos y los métodos de más de una clase.

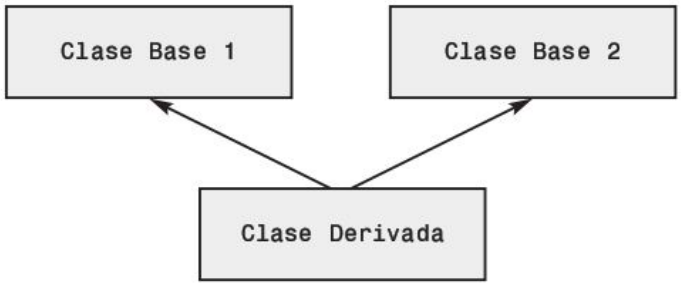


FIGURA 2.3 *Herencia múltiple*

Para definir una relación de herencia múltiple se utiliza la siguiente sintaxis.

```
class Base1
{

   /* Declaración de atributos y métodos de la clase Base1. */

};

class Base2
{

   /* Declaración de atributos y métodos de la clase Base2. */

};

…

class Basen
{

   /* Declaración de atributos y métodos de la clase Basen. */

};

class Derivada: public Base1, public Base2, …, public Basen
{

   /* Declaración de atributos y métodos de la clase Derivada. */

};
```

Cuando la palabra reservada **public**, precede el nombre de cada una de las clases se hace referencia a que la clase `Derivada` hereda atributos y métodos de todas ellas.

Para definir el constructor de la clase `Derivada`, se procede de la siguiente manera:

```
Derivada::Derivada (parámetros): Base1(parámetros clase Base1), Base2
➥(parámetros clase Base2), …, Basen (parámetros clase Basen)
{

   /* Cuerpo del constructor de la clase Derivada. */

}
```

Al llamar al constructor de la clase `Derivada`, primero se ejecuta el constructor de la clase $\texttt{Base}_1$, después el constructor de la clase $\texttt{Base}_2$, y así sucesivamente hasta el constructor de la clase $\texttt{Base}_n$. Por último, se ejecutan las instrucciones que aparezcan en el cuerpo del constructor de la clase `Derivada`.

A continuación se presenta un ejemplo de herencia múltiple. Se definen las clases `Boleto` y `Hotel` que se utilizarán como base para definir la clase derivada `PlanVacac`. Esta clase heredará todos los miembros de las clases bases, aunque sólo tendrá acceso a los miembros públicos de ellas.

Programa 2.2

```
/* Se definen las clases Boleto y Hotel, que servirán como base para
➥definir la clase PlanVacac, representando un caso de herencia múltiple.
➥Se presenta una aplicación muy sencilla que utiliza las clases
➥previamente definidas. */

/* Definición de la clase Boleto. */
class Boleto
{
   private:
        float Precio;
        char Numero[64], CdadOri[64], CdadDes[64];
   public:
        Boleto();
        Boleto(float, char *, char*, char *);
        void Imprime();
};
```

```
/* Declaración del constructor por omisión. */
Boleto::Boleto()
{ }

/* Declaración del constructor con parámetros. */
Boleto::Boleto(float Pre, char Num[], char CO[], char CD[])
{
   Precio= Pre;
   strcpy(Numero, Num);
   strcpy(CdadOri, CO);
   strcpy(CdadDes, CD);
}

/* Método que imprime los valores de los atributos de un boleto. */
void Boleto::Imprime()
{
   cout<<"\n\nNúmero del boleto:   "<<Numero;
   cout<<"\nPrecio:              "<<Precio;
   cout<<"\nDe la ciudad: "<<CdadOri<<" a la ciudad: "<<CdadDes<<endl;
}

/* Definición de la clase Hotel. */
class Hotel
{
   private:
      float PrecioHab;
      int NumHab;
      char TipoHab;
   public:
      Hotel();
      Hotel(float, int, char);
      void Imprime();
};

/* Declaración del método constructor por omisión. */
Hotel::Hotel()
{ }

/* Declaración del método constructor con parámetros. */
Hotel::Hotel(float PreH, int NH, char TH)
{
   PrecioHab= PreH;
   NumHab= NH;
   TipoHab= TH;
}

/* Método que despliega los valores de los atributos de un hotel. */
void Hotel::Imprime()
```

```
{
   cout<<"\n\nNúmero de habitación:   "<<NumHab;
   cout<<"\nPrecio:                  "<<PrecioHab;
   cout<<"\nTipo de habitación:      "<<TipoHab<<endl;
}

/* Definición de la clase PlanVacac como clase derivada de las clases
➥Boleto y Hotel. Esta clase hereda los atributos de las otras dos.
➥Además, tiene dos atributos propios. */
class PlanVacac: public Boleto, public Hotel
{
   private:
      char Descrip[64];
      int TotalDias;
   public:
      PlanVacac();
      PlanVacac(float, char *, char *, char*, float, int, char, char *,
               ➥int);
      void Imprime();
};

/* Declaración del método constructor por omisión. */
PlanVacac::PlanVacac()
{ }

/* Declaración del método constructor con parámetros. */
PlanVacac::PlanVacac(float PB, char NB[], char CO[], char CD[],
                     ➥float PH, int NH, char TH, char Des[], int TD):
                     ➥Boleto(PB, NB, CO, CD), Hotel(PH, NH, TH)
{
   strcpy(Descrip, Des);
   TotalDias= TD;
}

/* Método que despliega los valores de los atributos de un plan
➥vacacional. */
void PlanVacac::Imprime()
{
   cout<<"\nDescripción:   "<<Descrip;
   cout<<"\nTotal de días: "<<TotalDias;
   cout<<"\nDatos del boleto\n ";
   Boleto::Imprime();
   cout<<"\nDatos del hotel\n ";
   Hotel::Imprime();
}

/* Función que pide al usuario los datos relacionados a un viaje. Con
➥estos datos se crea un objeto tipo PlanVacac. Regresa como resultado
➥dicho objeto. */
PlanVacac Lee ()
```

```
{
   char CO[64], CD[64], NumBol[64], TH, Des[64];
   float Prec, PreHab;
   int NumHab, TD;

   cout<<"\n¿De dónde sale? ";
   cin>>CO;
   cout<<"\n¿A dónde llega? ";
   cin>>CD;
   cout<<"\nPrecio: ";
   cin>>Prec;
   cout<<"\nNúmero de boleto:  ";
   cin>>NumBol;
   cout<<"\nTipo de habitación: ";
   cin>>TH;
   cout<<"\nPrecio de la habitación: ";
   cin>>PreHab;
   cout<<"\nNúmero de habitación asignada: ";
   cin>>NumHab;
   cout<<"\nTipo de paquete: ";
   cin>>Des;
   cout<<"\nTotal de días: ";
   cin>>TD;
   PlanVacac  Paquete(Prec, NumBol,CO, CD, PreHab, NumHab, TH, Des, TD);
   return Paquete;
}

/* Función que usa las clases previamente definidas entre las cuales
➥existe una relación de herencia múltiple. */
void UsaHerenciaMultiple()
{
   PlanVacac Viaje;

   Viaje= Lee();
   cout<<"\n\nDatos del paquete seleccionado:  ";
   Viaje.Imprime();
}
```

2.3 Herencia de niveles múltiples

La ***herencia de niveles múltiples*** se presenta cuando una clase derivada se usa como base para definir otra clase derivada. Es decir, existen diferentes niveles de herencia: en el primero, la clase derivada hereda los miembros de una clase base, mientras que en el segundo, la clase derivada funciona a su vez como una clase base y de esta forma comparte con una tercera clase sus propios miembros y los

que heredó. Esta relación puede extenderse a tantos niveles como lo requiera el problema que se esté resolviendo.

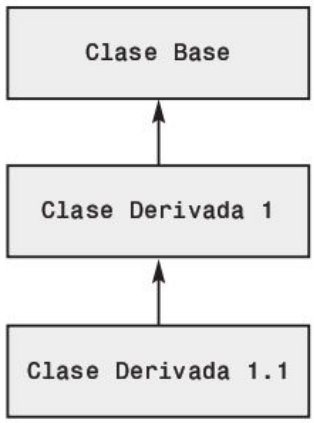


FIGURA 2.4 *Herencia de niveles múltiples*

Este tipo de herencia es muy útil cuando es necesario representar, a partir de conceptos generales, conceptos más específicos. Cuantos más niveles se deriven, más especificidad se definirá. La figura 2.5 presenta un ejemplo de herencia de niveles múltiples. El nivel superior representa la clase más general, la clase `Árbol`. Luego, la clase `Frutal` es una clase derivada de la primera, lo cual indica que los frutales son una clase más específica de árboles. Por último, se define la clase `Cítrico`, como una subclase de la clase `Frutal`. Esta relación también indica que los cítricos son una variante, una clase más específica, de los árboles frutales.

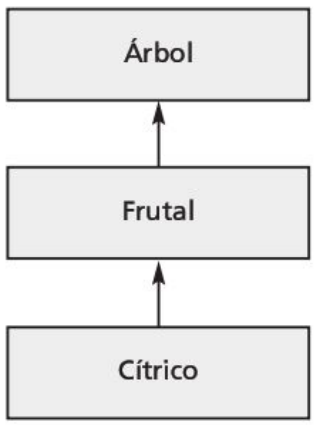


FIGURA 2.5 *Ejemplo de herencia de niveles múltiples*

El programa 2.3 presenta parte del código desarrollado para definir las clases de la figura 2.5, además de un ejemplo sencillo de aplicación de las mismas.

Programa 2.3


```
/* Se declara la clase Árbol que será superclase de la clase Frutal.
➥De ésta, a su vez, se derivará la clase Cítrico. Por lo tanto, esta
➥última hereda los miembros de las dos anteriores. Con esta relación de
➥herencia, se expresa que un objeto tipo Cítrico, es además del tipo
➥Frutal y también un Árbol.*/

/* Definición de la clase Arbol. */
class Arbol
{
   protected:
      int Edad;
      double Altura;
      char Nombre[64];
   public:
      Arbol(int Ed, double Alt, char *Nom);
      void ImprimeArbol();
};

/* Declaración del método constructor con parámetros. Asigna valores a
➥los atributos. */
Arbol::Arbol(int Ed, double Alt, char *Nom)
{
   Edad= Ed;
   Altura= Alt;
   strcpy(Nombre, Nom);
}

/* Imprime los valores de los atributos de un árbol. */
void Arbol::ImprimeArbol()
{
   cout<<"Nombre:   " << Nombre <<  endl;
   cout<<"Edad: " << Edad << endl;
   cout<<"Altura: " << Altura << endl;
}

/* Primer nivel de herencia: declaración de la clase Frutal como clase
➥derivada de la clase Arbol.*/
class Frutal: public Arbol
```

```
{
   protected:
      char EstacionFruto[64];
   public:
      Frutal(int Ed, double Alt, char  *Nom, char *EstFr);
      void ImprimeFrutal();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Frutal::Frutal (int Ed, double Alt, char *Nom, char *EstFr):
       ➥Arbol (Ed, Alt, Nom)
{
   strcpy(EstacionFruto, EstFr);
}

/* Método que despliega los valores de los atributos de un árbol frutal. */
void Frutal::ImprimeFrutal()
{
   Arbol::ImprimeArbol();
   cout<<"Estación del año en la que da frutos: " << EstacionFruto << endl;
}


/* Segundo nivel de herencia: definición de la clase Citrico como derivada
➥de la clase Frutal. */
class Citrico: public Frutal
{
   private:
      char NombreCitrico[64];
   public:
      Cítrico (int Ed, double Alt, char *Nom, char *EstFr, char
      ➥*NomCit);
   void ImprimeCitrico();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Citrico::Citrico(int Ed, double Alt, char *Nom, char *EstFr, char *NomCit):
        ➥Frutal (Ed, Alt, Nom, EstFr)
{
   strcpy(NombreCitrico, NomCit);
}

/* Método que despliega los valores de los atributos de un cítrico. */
void Citrico::ImprimeCitrico()
```

```
{
   Frutal::ImprimeFrutal();
   cout<<"Nombre del Cítrico: " << NombreCitrico << endl;
}

/* Función que usa las clases definidas previamente en las cuales existe
➥una relación de herencia de niveles múltiples: crea objetos e imprime
➥el valor de sus atributos. */
void UsaHerencia()
{
   Arbol  ObjArbol(2, 3.55, "Álamo");
   Frutal  ObjFrutal(3, 2.56, "Manzano", "Otoño");
   Citrico  ObjCitrico(1, 2.22, "Limonero", "Invierno", "Limón");

   ObjArbol.ImprimeArbol();

   ObjFrutal.ImprimeFrutal();

   ObjCitrico.ImprimeCitrico();
}
```


La figura 2.6 presenta otro ejemplo de herencia de niveles múltiples; tiene cuatro niveles de clases: en la primera, `Alumno`, se define una clase que podría utilizarse para describir el concepto alumno en general (podrían ser alumnos de primaria, secundaria o de cualquier otro nivel). A partir de esta clase se deriva la clase `Universitario`, con la cual se gana cierto grado de precisión. Ahora ya se describen a los alumnos que asisten a alguna universidad. En el siguiente nivel, con la clase `Ingeniería`, se indica una clase más específica que las anteriores. Se trata de alumnos universitarios que estudian algún tipo de ingeniería (ya no cualquier carrera universitaria). Por último, la clase `Computación` describe a los estudiantes universitarios de ingenierías en computación. Es decir, ya no son los alumnos de cualquier ingeniería, sino específicamente los que estudian Ingeniería en Computación. Así, la clase `Computación` hereda los miembros de `Ingeniería`, de `Universitario` y de `Alumno`. Además, podrá tener un conjunto de atributos y métodos propios.

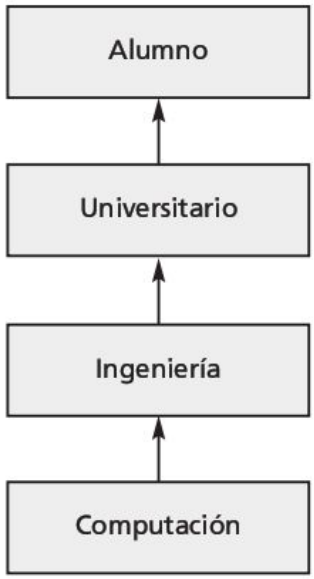


FIGURA 2.6 *Ejemplo de herencia de niveles múltiples*

El programa 2.4 presenta un segmento de código basado en la relación de herencia que se muestra en la figura 2.6. El programa tiene un nivel más de herencia, ya que se define primero la clase `Persona`, y a partir de ella la clase `Alumno`.

Programa 2.4

```
/* Se define la clase Persona de la cual se deriva la clase Alumno. De
➥ésta se deriva la clase Universitario, que a su vez sirve como base
➥para definir la clase Ingeniería. Finalmente, a partir de ésta se
➥define la clase Computación. */

class Persona
{
   protected:
      char *Nombre;
      int Edad;
   public:
      Persona(char *Nom, int Ed);
      void ImprimePersona();
};

/* Declaración del método constructor con parámetros. */
Persona::Persona(char *Nom, int Ed)
```

```
{
   Nombre= new char[strlen(Nom)+1];
   if (Nombre)
   {
       strcpy(Nombre, Nom);
       Edad= Ed;
   }
}

/* Método que despliega los valores de los atributos de una persona. */
void Persona::ImprimePersona()
{
   cout<<"Nombre: " << Nombre << endl;
   cout<<"Edad: " << Edad << endl;
}


/* Primer nivel de herencia: definición de la clase Alumno como clase
➥derivada de la clase Persona. */
class Alumno: public Persona
{
   protected:
      float Promedio;
   public:
      Alumno(char *Nom, int Ed, float Prom);
      void ImprimeAlumno();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Alumno::Alumno(char *Nom, int Ed, float Prom): Persona(Nom, Ed)
{
   Promedio= Prom;
}

/* Método que despliega los valores de los atributos de un alumno. */
void Alumno::ImprimeAlumno()
{
   Persona::ImprimePersona();
   cout<< "Promedio: " << Promedio << endl;
}


/* Segundo nivel de herencia: definición de la clase Universitario como
➥clase derivada de la clase Alumno. */
class Universitario: public Alumno
{
   protected:
      char *NombreUniversidad;
```

```
   public:
      Universitario(char *Nom, int Ed, float Prom, char *NomUniv);
      void ImprimeUniversitario();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Universitario::Universitario(char *Nom, int Ed, float Prom, char *NomUniv):
              ➥Alumno(Nom, Ed, Prom)
{
   NombreUniversidad= new char[strlen(NomUniv)+1];
   if (NombreUniversidad)
       strcpy(NombreUniversidad, NomUniv);
}

/* Método que despliega los valores de los atributos de un alumno
➥universitario. */
void Universitario::ImprimeUniversitario()
{
   Alumno::ImprimeAlumno();
   cout<<"Nombre de la Universidad: " << NombreUniversidad << endl;
}


/* Tercer nivel de herencia: definición de la clase Ingeniería como
➥clase derivada de la clase Universitario. */
class Ingenieria: public Universitario
{
   protected:
      char *NombreIngenieria;
   public:
      Ingenieria(char *Nom, int Ed, float Prom, char *NomUniv, char
      ➥*NomIng);
      void ImprimeIngenieria();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Ingenieria::Ingenieria(char *Nom, int Ed, float Prom, char *NomUniv,
➥char *NomIng):
               Universitario(Nom, Ed, Prom, NomUniv)
{
   NombreIngenieria= new char[strlen(NomIng)+1];
   if  (NombreIngenieria)
        strcpy(NombreIngenieria, NomIng);
}

/* Método que despliega los valores de los atributos de un alumno de
➥alguna ingeniería. */
void Ingenieria::ImprimeIngenieria()
```

```
{
   Universitario::ImprimeUniversitario();
   cout <<"Nombre de la Ingeniería: " << NombreIngenieria << endl;
}


/* Cuarto nivel de herencia: definición de la clase Computación como
➥clase derivada de la clase Ingeniería. */
class Computacion: public Ingenieria
{
   protected:
      char Plataformas[64];
   public:
      Computacion(char *Nom, int Ed, float Prom, char *NomUniv, char
                  ➥*NomIng, char *Pla);
      void ImprimeComputacion();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase base. */
Computacion::Computacion(char *Nom, int Ed, float Prom, char *NomUniv,
            ➥char *NomIng, char *Pla): Ingenieria (Nom, Ed, Prom,
            ➥NomUniv, NomIng)
{
   strcpy(Plataformas, Pla);
}

/* Método que despliega los valores de los atributos de un alumno de
➥ingeniería en computación. */
void Computacion::ImprimeComputacion()
{
   Ingenieria::ImprimeIngenieria();
   cout<< "Plataformas usadas: " << Plataformas << endl;
}


/* Función que usa las clases previamente definidas en las cuales hay
➥una relación de herencia de niveles múltiples. */
void UsaHerencia()
{
   Persona  ObjPersona("Carlos", 23);
   Alumno  ObjAlumno("Adriana", 20, 9.75);
   Universitario  ObjUniversitario("Carolina", 19, 8.65, "ITAM");
   Ingenieria  ObjIngenieria("Pablo", 21, 8.25, "UNAM", "Mecánica");
   Computacion  ObjComputacion("Alfonso", 22, 9.8, "UPT", "Computación",
               ➥"Varias");
```

```
    ObjPersona.ImprimePersona();
    ObjAlumno.ImprimeAlumno();.
    ObjUniversitario.ImprimeUniversitario();
    ObjIngenieria.ImprimeIngenieria();

/* Imprime los datos del alumno de ingeniería en computación. */
    ObjComputacion.ImprimeComputacion();

/* Imprime sólo los datos personales del alumno de ingeniería en
➥computación. */
    ObjComputacion.ImprimePersona();
}
```

La figura 2.7 muestra un esquema que representa diferentes tipos de herencia entre diversas clases. Se puede mencionar un caso de herencia simple entre la clase `Vehiculo` y la clase `Motocicleta`, y un caso de herencia de niveles múltiples entre la clase `Vehiculo` y la clase `Deportivo`.

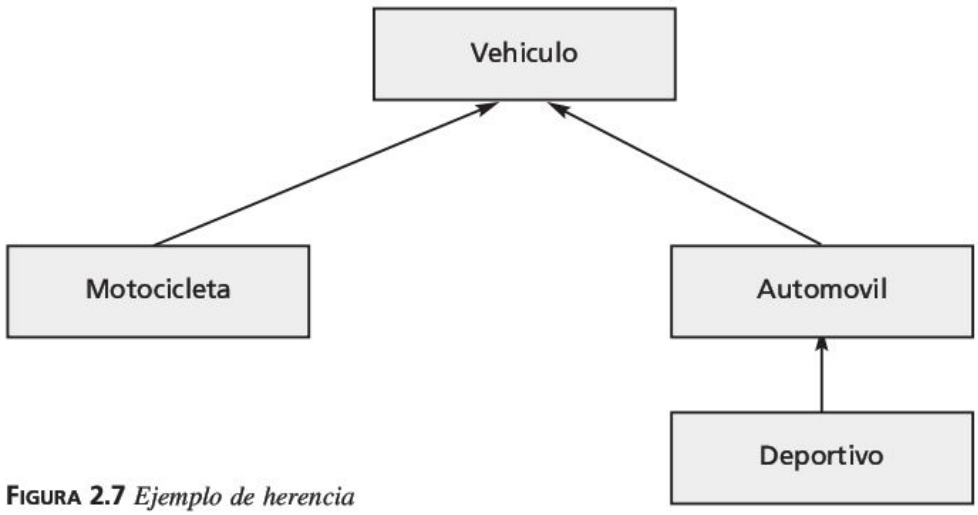


Figura 2.7 *Ejemplo de herencia*

El programa 2.5 presenta código que incluye la definición de las clases correspondientes al esquema de la figura 2.7, así como una función que hace uso de las mismas.

Programa 2.5

```
/* La clase Vehiculo se define por medio de los atributos privados
➥Marca, Placas y el Número de Motor, así como por medio de un método
➥para desplegar los valores de los atributos y un constructor. Esta
➥clase sirve como clase base para definir las clases Motocicleta y
➥Automovil. De la clase Automovil se deriva la clase Deportivo. */

/* Definición de la clase Vehiculo. */
class Vehiculo
{
   private:
       char Marca[32];
       int Placas, NumMotor;
   public:
       Vehiculo(char *Mar, int Pla, int NM);
       void ImprimeVehiculo();
};

/* Declaración del método constructor con parámetros. */
Vehiculo::Vehiculo(char *Mar, int Pla, int NM)
{
   strcpy(Marca, Mar);
   Placas= Pla;
   NumMotor= NM;
}

/* Método que despliega los valores de los atributos de un vehículo. */
void Vehiculo::ImprimeVehiculo()
{
   cout<<"Marca: " << Marca << '\n';
   cout<<"Placas:    " << Placas << '\n';
   cout<<"Número de Motor: " << NumMotor << '\n';
}


/* Se define la clase Motocicleta, derivada de Vehiculo, la cual tiene
➥como atributos propios la Potencia del Motor y como método el que le
➥permite desplegar sus atributos. */
class Motocicleta: public Vehiculo
{
    private:
       int PotMotor;
    public:
       Motocicleta(char *Mar, int Pla, int NM, int PM);
       void ImprimeMotocicleta();
};
```

```
/* Declaración del método constructor. Invoca al método constructor de
➥la clase Vehiculo. */
Motocicleta::Motocicleta (char *Mar, int Pla, int NM, int PM):
            ➥Vehiculo(Mar, Pla, NM)
{
   PotMotor= PM;
}

/* Método que despliega los valores de los atributos de una motocicleta. */
void Motocicleta::ImprimeMotocicleta()
{
   Vehiculo::ImprimeVehiculo();
   cout<<"Potencia del Motor: " << PotMotor << '\n';
}


/* Se define la clase Automovil, derivada de Vehiculo, la cual tiene
➥como atributos propios el Número de Puertas y como método el que le
➥permite desplegar sus atributos. */
class Automovil: public Vehiculo
{
   private:
       int NumPuertas;
   public:
       Automovil(char *Mar, int Pla, int NM, int NP);
       void ImprimeAutomovil();
};

/* Declaración del método constructor. Invoca al método constructor de
➥la clase Vehiculo. */
Automovil::Automovil(char *Mar, int Pla, int NM, int NP):
          ➥Vehiculo(Mar, Pla, NM)
{
   NumPuertas= NP;
}

/* Método que despliega los valores de los atributos de un automóvil. */
void Automovil::ImprimeAutomovil()
{
   Vehiculo::ImprimeVehiculo();
   cout<<"Número de Puertas: " << NumPuertas << '\n';
}


/* Se define la clase Deportivo, derivada de Automovil, la cual tiene
➥como atributo propio el Color y como método el que le permite desplegar
➥sus atributos. */
class Deportivo: public Automovil
```

```
{
   private:
       char Color[8];
   public:
       Deportivo (char *Mar, int Pla, int NM, int NP, char *Col);
       void ImprimeDeportivo();
};

/* Declaración del método constructor. Invoca al constructor de la clase
➥Automovil. */
Deportivo::Deportivo (char *Mar, int Pla, int NM, int NP, char *Col):
           Automovil (Mar, Pla, NM, NP)
{
   strcpy(Color, Col);
}

/* Método que despliega los valores de los atributos de un automóvil
➥deportivo. */
void Deportivo::ImprimeDeportivo()
{
   Automovil::ImprimeAutomovil();
   cout<<"Color: " << Color << '\n';
}


/* Función que hace uso de las clases previamente definidas. */
void UsaHerencia()
{
/* Declaración de un objeto de tipo Motocicleta. */
   Motocicleta Moto("Honda", 231, 2941, 225);

/* Declaración de un objeto de tipo Automovil. */
   Automovil Auto("BMW", 569, 7436, 4);

/* Declaración de un objeto de tipo Deportivo. */
   Deportivo AutoDep("Ferrari", 442, 52348, 2, "rojo");

/* Despliega las características de la motocicleta. */
   Moto.ImprimeMotocicleta();

/* Despliega las características del automóvil. */
   Auto.ImprimeAutomovil();

/* Despliega las características del automóvil deportivo. */
   AutoDep.ImprimeDeportivo();
}
```

Todos los ejemplos presentados en esta sección corresponden a herencia pública, la cual es la más utilizada. A continuación se presenta una breve introducción a la herencia de tipo privada.

2.4 Herencia privada

En el caso de la ***herencia privada***, todos los miembros de la clase base, sin importar si son privados, protegidos o públicos, serán privados para la clase derivada. Por lo tanto, sólo se podrán acceder por medio de los métodos de la clase base.

```
/* Declaración de la clase Base.*/
class Base
{
…
};

/* Declaración de la clase derivada D1 a partir de Base, usando la
relación de herencia pública. */
class D1: public Base        (1)
{
…
};

/* Declaración de la clase derivada D2 a partir de Base, usando la
relación de herencia privada. */
class D2: private Base        (2)
{
…
};
```

En (1) se declara herencia pública entre la clase `Base` y la clase derivada `D1`. Por lo tanto, los miembros públicos y protegidos de `Base` estarán disponibles para `D1`, mientras que los miembros privados de `Base` no podrán usarse directamente desde `D1`. En (2) se declara herencia privada entre la clase `Base` y la clase derivada `D2`. Por lo tanto, los miembros públicos, protegidos y privados de `Base` serán todos privados para `D2`.

2.5 Clases amigas (friend)

Existen casos en los cuales es necesario que una clase haga uso de los miembros de otra clase, sin que exista relación de herencia entre las mismas. Es decir, dos clases que no comparten atributos pueden requerir algún tipo de cooperación en algún momento. Para permitir este tipo de relación se declara una clase como ***amiga*** de otra.

Cuando en la declaración de una clase se dice que otra clase es su amiga, se está permitiendo que esta última tenga acceso a los miembros privados y protegidos de la primera.

En el lenguaje ***C++*** se usa la palabra reservada **`friend`** para indicar que una clase es amiga de otra. La directiva **`friend class`** `NombreClase` se escribe en la sección pública de la clase, cuyos miembros podrán ser utilizados por los de la clase `NombreClase`. La palabra reservada **`class`** puede omitirse.

```
class Uno
{
          …
   public:
       …
       /* Los métodos de la clase Dos podrán acceder a los atributos
       ➥privados y protegidos de la clase Uno. */
       friend class Dos;

   …
};

/* La declaración de la clase Dos no se modifica. */
class Dos
{

   /* Declaración de atributos y métodos. */

};
```

En este caso, en la sección pública de la declaración de la clase `Uno` se indica que la clase `Dos` es su amiga. Por lo tanto, esta última podrá tener acceso a los miembros privados y protegidos de la primera.

El acceso logrado a través del uso de la declaración de clases amigas *no se hereda ni es transitivo*. Es decir, si la clase `Dos` tuviera clases derivadas, éstas no podrían tener acceso a los miembros de la clase `Uno`. Por otra parte, si la clase `Dos` tuviera otras clases amigas, éstas tampoco podrían tener acceso a los miembros de la clase `Uno`.

A continuación, el programa 2.6 presenta un ejemplo de clases amigas. Define las clases `Medico` y `Paciente`, y declara la última como clase amiga de la primera. De esta manera, el método `AsociarMedico`, de la clase `Paciente`, podrá utilizar directamente miembros privados de la clase `Medico`.

Programa 2.6

```
/* Se definen las clases Medico y Paciente, siendo esta última una clase
➥amiga de la primera. Por lo tanto, la clase Paciente podrá tener acceso
➥a todos los miembros de la clase Medico. */

/* Prototipo de la clase Paciente. La definición de la misma aparece más
➥adelante. */
class Paciente;

/* Definición de la clase Medico. */
class Medico
{
   private:
       char NombreCompleto[64], Especialidad[64];
   public:
       Medico();
       Medico(char *NomCom, char *Esp);
       char * ObtenerNombreCompleto();
       char * ObtenerEspecialidad();
       void ImprimeDatos();
       /* Clase amiga que tiene acceso a los miembros privados de la
       ➥clase Medico. */
       friend class Paciente;
};

/* Declaración del método constructor por omisión. */
Medico::Medico()
{ }

/* Declaración del método constructor con parámetros. */
Medico::Medico(char *NomCom, char *Esp)
{
   strcpy(NombreCompleto, NomCom);
   strcpy(Especialidad, Esp);
}
```

```
/* Método que permite, a los usuarios externos a la clase, conocer el
➥nombre del médico. */
char * Medico::ObtenerNombreCompleto()
{
   return NombreCompleto;
}

/* Método que permite, a los usuarios externos a la clase, conocer la
➥especialidad del médico. */
char * Medico::ObtenerEspecialidad()
{
   return Especialidad;
}

/* Método que despliega los valores de los atributos de un médico. */
void Medico::ImprimeDatos(void)
{
   cout<<"Nombre completo del médico: " << NombreCompleto << endl;
   cout<<"Especialidad: " << Especialidad << endl <<endl;
}


/* Definición de la clase Paciente. */
class Paciente
{
   private:
       char NombreCompleto[64];
       int Edad;
       char Padecimiento[64];
       Medico *MedicoEspecialista;
   public:
       Paciente();
       Paciente(char *NomCom, int Ed, char *Pad);
       char * ObtenerNombreCompleto();
       int ObtenerEdad();
       char * ObtenerPadecimiento();
       void AsociarMedico();
       void ImprimeDatos();
};

/* Declaración del método constructor por omisión. */
Paciente::Paciente()
{ }

/* Declaración del método constructor con parámetros. */
Paciente::Paciente(char *NomCom, int Ed, char *Pad)
```

```
{
   strcpy(NombreCompleto, NomCom);
   Edad= Ed;
   strcpy(Padecimiento, Pad);
}

/* Método que permite, a los usuarios externos a la clase, conocer el
➥nombre del paciente. */
char * Paciente::ObtenerNombreCompleto()
{
   return NombreCompleto;
}

/* Método que permite, a los usuarios externos a la clase, conocer la
➥edad del paciente. */
int Paciente::ObtenerEdad()
{
   return Edad;
}

/* Método que permite, a los usuarios externos a la clase, conocer el
➥nombre del padecimiento. */
char * Paciente::ObtenerPadecimiento()
{
   return Padecimiento;
}

/* Método que asocia un médico especialista a cada paciente. Note cómo
➥el miembro MedicoEspecialista (de tipo puntero a un objeto tipo Medico)
➥tiene acceso a los miembros privados de la clase Medico. */
void Paciente::AsociarMedico()
{
   MedicoEspecialista= new Medico();
   cout<<"Ingrese el Nombre Completo del Médico: ";
   cin>>MedicoEspecialista->NombreCompleto;
   cout<<"Ingrese la especialidad: ";
   cin>>MedicoEspecialista->Especialidad;
}

/* Método que despliega los valores de los atributos de un paciente. */
void Paciente::ImprimeDatos()
{
   cout<<"\nNombre Completo: " << NombreCompleto << endl;
   cout<<"Edad: " << Edad << endl;
   cout<<"Padecimiento: " << Padecimiento << endl;
   cout<<"Datos del Médico Especialista:" << endl;
   MedicoEspecialista->ImprimeDatos();
}
```

```
/* Función que usa las clases amigas previamente definidas. */
void UsaClaseAmiga()
{
   Paciente ObjPacienteA ("Juan Carlos G.", 25, "Gripe"),
   ObjPacienteB ("Adriana Z.", 38, "Gastritis");

   ObjPacienteA.AsociarMedico();
   ObjPacienteB.AsociarMedico();

   ObjPacienteA.ImprimeDatos();
   ObjPacienteB.ImprimeDatos();
}
```


2.6 Métodos amigos

Los ***métodos amigos*** de una clase son métodos que no pertenecen a ella, pero a los cuales se les permite el acceso a sus miembros privados y protegidos. Es decir, en la definición de una clase se incluye la directiva de que cierto método de otra clase es amigo de la que se está declarando. De esta forma, dicho método podrá utilizar libremente todos los miembros de la clase. El programa 2.7 ilustra este concepto.

Programa 2.7

```
/* Se definen las clases Ejemplo1 y Ejemplo2. En la clase Ejemplo1 se
➥indica que el método EsMayor de la clase Ejemplo2 es un método amigo de
➥la misma. */

/* Prototipo de la clase Ejemplo1. Su definición se muestra más adelante. */
class Ejemplo1;

class Ejemplo2
{
   private:
       int Valor2;
   public:
       Ejemplo2 (int);
       void Imprime();
       int EsMayor(Ejemplo1);
};
```

```
/* Declaración del método constructor. */
Ejemplo2:: Ejemplo2 (int Num)
{
   Valor2= Num;
}

/* Método que despliega el valor del atributo de la clase Ejemplo2. */
void Ejemplo2::Imprime()
{
   cout << "Valor del atributo: " << Valor2 << endl;
}

/* Método de la clase Ejemplo2. Este método es amigo de la clase Ejemplo1,
➥por lo que tendrá acceso a los miembros privados y/o protegidos de la
➥misma. */
int Ejemplo2::EsMayor(Ejemplo1 Obj)
{
   if (Valor2 > Obj.Valor1)
       return 1;
   else
       return 0;
}

/* Definición de la clase Ejemplo1. En esta clase, en la sección pública,
➥se incluye la declaración de un método de la clase Ejemplo2 como método
➥amigo, lo cual permite que este método tenga acceso a sus miembros
➥privados y protegidos. */
class Ejemplo1
{
   private:
       int Valor1;
   public:
       Ejemplo1 (int);
       void Imprime();
       friend int Ejemplo2::EsMayor(Ejemplo1);
};

/* Declaración del método constructor. */
Ejemplo1::Ejemplo1(int Num)
{
   Valor1= Num;
}

/* Método que despliega el valor del atributo de la clase Ejemplo1. */
void Ejemplo1::Imprime()
{
   cout << "Valor del atributo: " << Valor1 << endl;
}
```

```
/* Función que usa el método amigo de la clase Ejemplo1. */
void UsaMetodoAmigo()
{
   Ejemplo1  Obj (10);
   Ejemplo2  Obj2(12);

   Obj1.Imprime();
   Obj2.Imprime ();

   if (Obj2.EsMayor(Obj1))
       cout << "Obj2 es mayor que Obj1" << endl;
   else
       cout << "Obj2 no es mayor que Obj1" << endl;
}
```


En el ejemplo anterior, al indicar que el método `EsMayor()` de la clase `Ejemplo2` es un método amigo de la clase `Ejemplo1`, se permite que dicho método pueda comparar directamente el atributo `Valor2` con el atributo `Valor1` (miembro privado de la clase `Ejemplo1`).

2.7 Funciones amigas

Otra variante de este tipo de relación son las ***funciones amigas*** que se utilizan para que funciones ajenas a una clase puedan tener acceso a los miembros privados y/o protegidos de éstas. A continuación se muestra la sintaxis que se utiliza para representar esta relación.

```
class Uno
{
       ...
   public:
       ...
       friend tipo NombreFunción( parámetros);
};

tipo  NombreFunción (parámetros)
{
   /* La función se declara normalmente. */
   ...
}
```

El programa 2.8 presenta un ejemplo sencillo de funciones amigas.

Programa 2.8

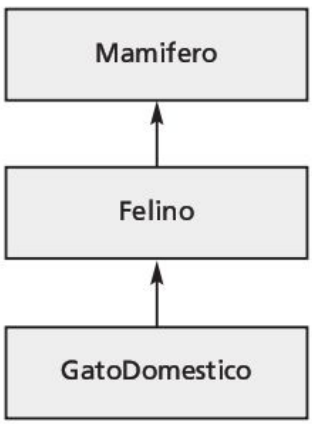
```
/* Se define la clase Ejemplo en la cual se incluye la declaración de
➥la función Suma como una función amiga de la misma. Esto permitirá que
➥dicha función pueda tener acceso a todos los miembros de la clase.*/
class Ejemplo
{
   private:
       int Atrib1, Atrib2;
   public:
       Ejemplo();
       Ejemplo(int, int);
       void Imprime();
       friend int Suma(int, Ejemplo);
};

/* Declaración del método constructor por omisión. */
Ejemplo::Ejemplo()
{}

/* Declaración del método constructor con parámetros. */
Ejemplo::Ejemplo(int Num1, int Num2)
{
   Atrib1= Num1;
   Atrib2= Num2;
}

/* Método que despliega los valores de los atributos de la clase. */
void Ejemplo::Imprime()
{
   cout<<"Valor del primer atributo:   "<<Atrib1<<endl;
   cout<<"Valor del segundo atributo:  "<<Atrib2<<endl;
}


/* Función entera declarada como amiga de la clase Ejemplo, lo que
➥permite que pueda sumar sus atributos a un entero de manera directa. En
➥este caso, el parámetro Valor es un objeto de tipo Ejemplo y la función
➥obtiene como resultado la suma de sus atributos más un número dado
➥también como parámetro.*/
int Suma(int Dato, Ejemplo Valor)
{
   return (Dato + Valor.Atrib1 + Valor.Atrib2);
}

/* Función que hace uso de la función amiga de la clase Ejemplo para
➥obtener la suma de sus atributos. */
```

```
void UsaFuncionesAmigas()
{
   int Resultado;
   Ejemplo ObjEjemplo(2, 5);
   Resultado= Suma (10, ObjEjemplo);
   cout<<"El resultado de la suma es:  "<<Resultado<<endl;
}
```


El programa del ejemplo anterior imprimirá el valor 17, ya que la función suma los atributos del objeto, dando un valor de 7 y a eso le suma el número entero (10) que recibe como parámetro.

Ejercicios

1. Considere la siguiente relación de herencia. Defina las clases `Mamífero`, `Felino` y `GatoDoméstico`. Decida qué atributos y métodos incluir de tal manera que su programa pueda:
   a) Declarar un objeto llamado `Minino` de tipo `GatoDoméstico` y otro llamado `EstrellaCirco` de tipo `Felino`.
   b) Imprimir la dieta de `Minino` y de `EstrellaCirco`.
   c) Imprimir el año y lugar de nacimiento de `Minino` y de `EstrellaCirco`.
   d) Cambiar el nombre del dueño de `Minino`.
   e) Imprimir la raza de `Minino` y de `EstrellaCirco`.
   f) Cambiar el nombre del circo en el que actúa `EstrellaCirco`.



2. Considere la siguiente relación de herencia. Defina las clases `Vehiculo`, `Terrestre` y `Maritimo`. Decida qué atributos y métodos incluir de tal manera que su programa pueda:

   a) Declarar un objeto llamado `MiAuto` de tipo `Terrestre` y otro llamado `MiBarco` de tipo `Maritimo`. La asignación de valores a los atributos debe hacerse a través de un método de lectura, definido para tal fin.

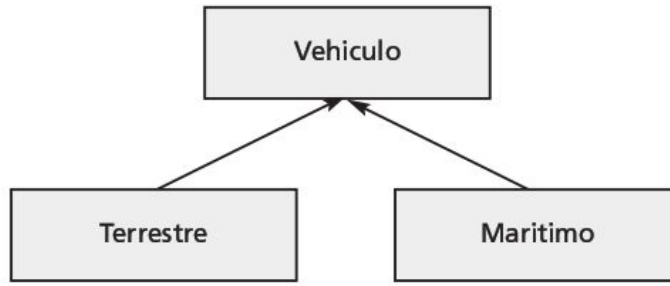
   b) Imprimir los atributos de los objetos declarados en el inciso anterior.

   c) Actualizar el precio de `MiAuto`.

   d) Actualizar potencia de motores de `MiBarco`. ¿A qué clase debería pertenecer el método que le permitirá hacer esta actualización?

   e) Imprimir un mensaje que indique si `MiAuto` tiene o no más de 5 años de antigüedad.

   f) Imprimir el tipo de combustible que usa y la capacidad máxima del tanque de `MiBarco`.



3. Definir la clase `Planta` que contenga todos los atributos que caracterizan a las plantas, y los métodos necesarios para manejarlos. Además, defina las clases derivadas `Arbol`, `Arbusto` y `Pino`, en el nivel de herencia adecuado.

4. Definir la clase `FiguraGeometrica` que contenga los atributos que caracterizan a toda figura geométrica y los métodos necesarios para manejarlos. Además, defina las clases derivadas `Cuadrado` y `Triangulo`. Escriba un programa en ***C++*** que haga uso de estas clases para calcular el total de metros cuadrados de tela necesaria para fabricar N almohadones con forma cuadrada y M almohadones con forma de triángulo equilátero. Los datos que se ingresarán al programa son:

Datos:

- N: total de almohadones con forma cuadrada.
- LadoC: tamaño, en centímetros, de cada uno de los lados del almohadón cuadrado.
- M: total de almohadones con forma triangular.
- LadoT: tamaño, en centímetros, de cada uno de los lados del almohadón triangular.

Resultado esperado: Total de metros cuadrados requeridos para la fabricación de los N + M almohadones.

5. Considere la relación de herencia que se muestra en la siguiente figura, la cual involucra tres clases: `Alumno`, `Deportista` y `BecadoDeporte`. Esta última representa a aquellos alumnos que son deportistas y que por esa razón han recibido una beca especial del gobierno para premiar sus esfuerzos. Decida qué atributos y métodos incluir de tal manera que su programa pueda:

   a) Declarar dos objetos llamados `AlumnoJuan` y `AlumnoPedro` de tipo `Alumno`.

   b) Declarar un objeto llamado `DeporLuis` de tipo `Deportista`.

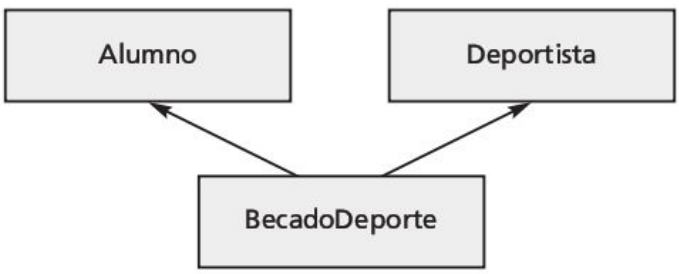
   c) Declarar dos objetos llamados `BDAna` y `BDCarmen` de tipo `BecadoDeporte`.

   d) Imprimir los datos de todos los objetos declarados.

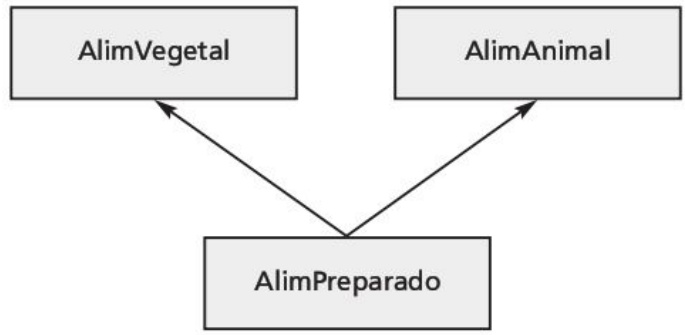
   e) Actualizar el nombre de la carrera que están estudiando `AlumnoJuan` y `BDAna`. El dato dado por el usuario será el nombre de la nueva carrera.

   f) Actualizar el nombre del entrenador de `DeporLuis` y `BDCarmen`. El dato dado por el usuario será el nombre del nuevo entrenador.

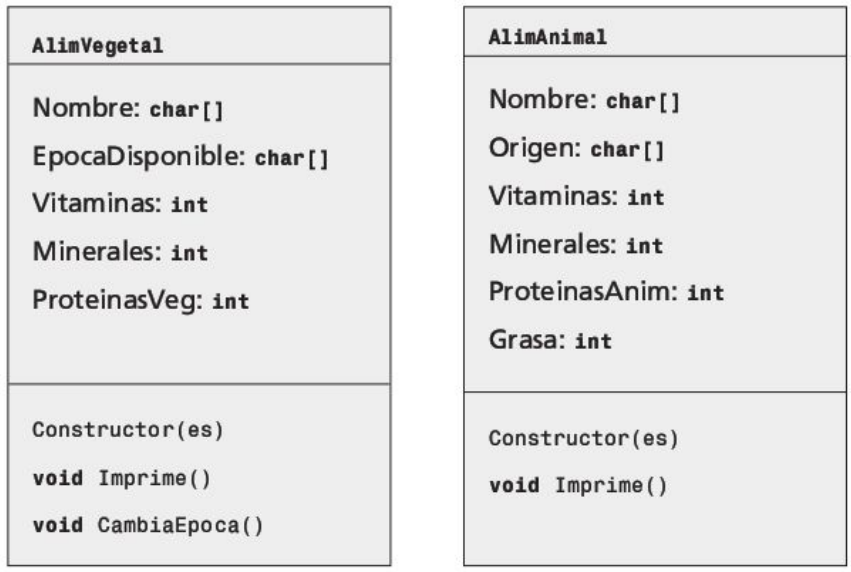
   g) Actualizar el monto de la beca de `BDAna` y `BDCarmen`. El dato dado por el usuario será el porcentaje de incremento de la beca actual.

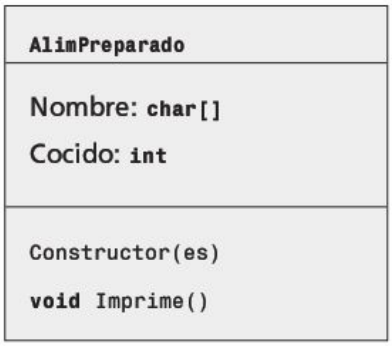


6. Considere la relación de herencia que se muestra en la siguiente figura. La misma involucra tres clases: `AlimVegetal`, `AlimAnimal` y `AlimPreparado`. Esta última representa a los alimentos preparados que pueden incluir como base alimentos vegetales y/o animales.



Se sugiere incluir los siguientes atributos y métodos:





Escriba un programa en *C++* que:

a) Declare y cree un objeto llamado `Tallarines` de tipo `AlimPreparado`. Los tallarines se preparan con harina de trigo, huevo y agua. Se cuecen en agua hirviendo.

b) Declare y cree un objeto llamado `EnsaladaVerde` de tipo `AlimPreparado`. La ensalada verde se prepara con diferentes tipos de lechuga y se condimenta con aceite de oliva, vinagre balsámico y sal (esta última no puede ser representada). No se cuece.

c) Imprima los atributos de los `Tallarines` y de la `EnsaladaVerde`.

d) Declare y cree un objeto llamado `LechugaFrancesa` de tipo `AlimVegetal`.

e) En el objeto `LechugaFrancesa` actualice el valor del atributo `EpocaDisponible` a "todo el año".

f) Declare y cree un objeto llamado `Salmon` de tipo `AlimAnimal`.

g) Imprima los atributos de la `LechugaFrancesa` y del `Salmon`.

7. Considere las siguientes relaciones de herencia. Defina todas las clases que aparecen en el esquema. Decida qué atributos y métodos incluir de tal manera que su programa pueda:

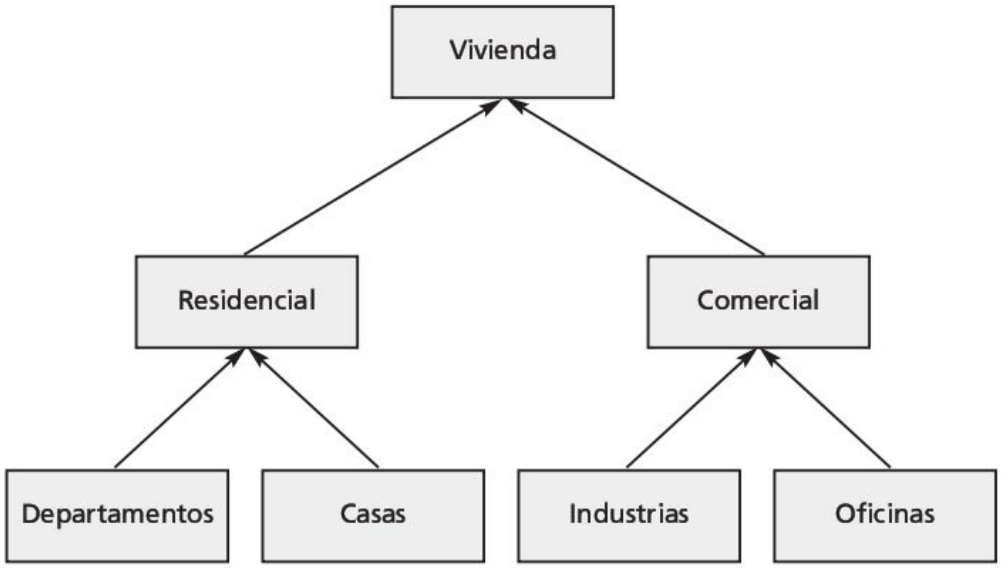
a) Declarar y crear objetos de cualquiera de las clases sin utilizar métodos de lectura para asignar valores a los atributos.

b) Imprimir los atributos de cualquiera de los objetos declarados.

c) Actualizar las dimensiones de un objeto tipo `Casas`. El usuario deberá proporcionar el nuevo número de metros cuadrados de la casa.

d) Actualizar el giro de una industria. Es decir, a un objeto tipo `Industria` se le podrá cambiar el valor de un atributo que representa el tipo de actividad que desarrolla dicha industria.

e) Actualizar el número de teléfono de cualquiera de los objetos declarados.



8. Defina las clases `DireccionEscolar` y `Alumno` de acuerdo a las especificaciones que se proporcionan más adelante. Observe que la clase `DireccionEscolar` incluye métodos para modificar valores de algunos atributos de objetos tipo `Alumno` que, por razones de seguridad, deben ser privados. Para permitir este acceso debe hacer uso de la relación de amistad (**friend**) explicada en este capítulo. Una vez definidas las clases indicadas, escriba un programa en ***C++*** que permita:

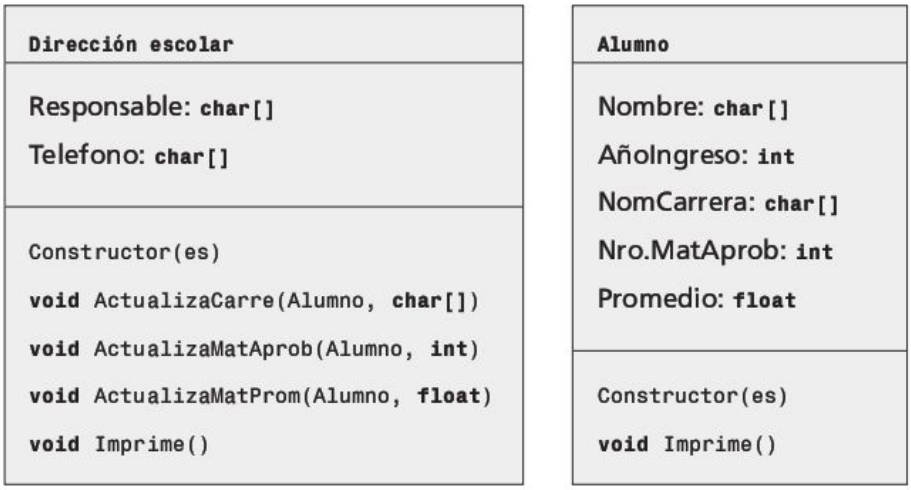
   a) Crear un objeto llamado `AlumnoJuan` de tipo `Alumno` y un objeto llamado `DirEsc` de tipo `DireccionEscolar`.

   b) Imprimir los datos del `AlumnoJuan` y de la `DirEsc`.

c) Registrar un cambio de carrera para el `AlumnoJuan`. El usuario deberá proporcionar el nombre de la carrera a la cual se cambiará.

d) Registrar una actualización del número de materias aprobadas por el `AlumnoJuan`. El usuario deberá proporcionar el total de materias aprobadas en este último semestre (el cual se sumará al total anterior).

e) Registrar una actualización del promedio de calificaciones del `AlumnoJuan`. El usuario deberá proporcionar el nuevo promedio.



9. Defina las clases `RecursosHumanos` y `Empleado` de acuerdo a las especificaciones proporcionadas más adelante. Observe que la clase `RecursosHumanos` incluye métodos para modificar valores de algunos atributos de objetos tipo `Empleado` que, por razones de seguridad, deben ser privados. Para permitir este acceso debe hacer uso de la relación de amistad (**friend**). Una vez definidas las clases indicadas, escriba un programa en ***C++*** que permita:

   a) Crear un objeto llamado `EmpleadoPedro` de tipo `Empleado` y un objeto llamado `Personal` de tipo `RecursosHumanos`.

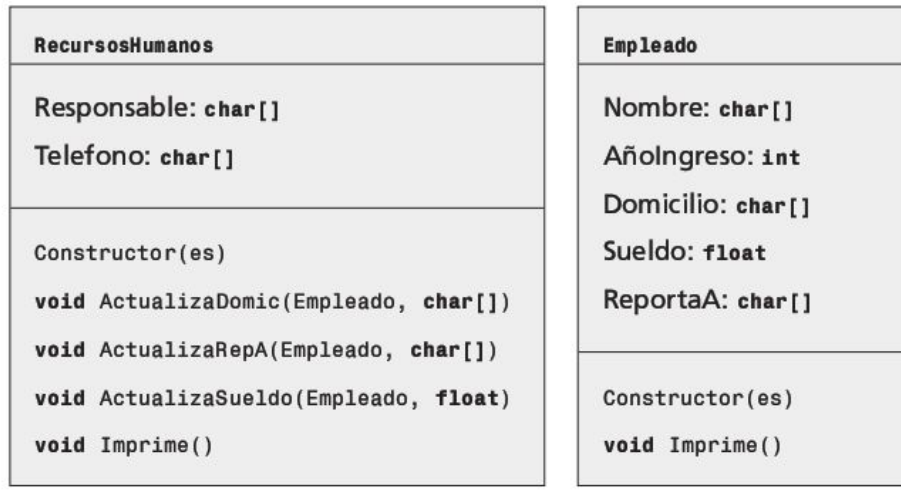
   b) Imprimir los datos del `EmpleadoPedro` y de `Personal`.

   c) Registrar un cambio de domicilio del `EmpleadoPedro`. El usuario deberá proporcionar el nuevo domicilio.

d) Registrar un cambio en el nombre de la persona a la cual reporta el `EmpleadoPedro`. El usuario deberá proporcionar el nombre de la persona a la que reportará a partir de ahora.

e) Registrar una actualización del sueldo del `EmpleadoPedro`. El usuario deberá proporcionar el nuevo sueldo.

f) Imprimir los datos del `EmpleadoPedro` si lleva más de 10 años trabajando en la empresa.



10. Retome el problema 9 pero ahora utilice el concepto de métodos amigos. Reescriba lo que considere necesario de tal manera que sólo los métodos `ActualizaDomic`, `ActualizaRepA` y `ActualizaSueldo` puedan tener acceso a los miembros privados de la clase `Empleado`.

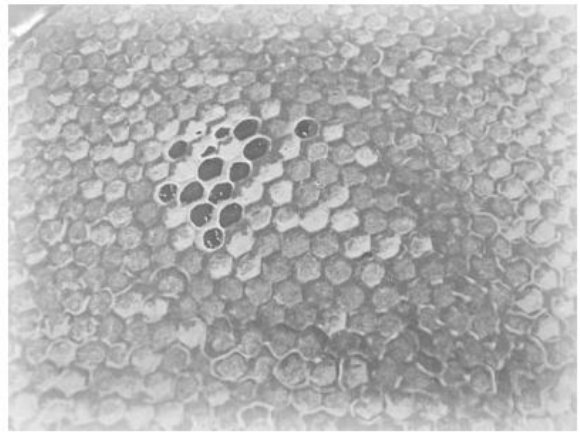
11. Defina la clase `Empleado` según las especificaciones que se proporcionan más adelante. Además, en la clase debe incluir una relación de amistad con una función que tenga como objetivo calcular el sueldo a pagar al empleado, de acuerdo a la siguiente expresión:

```
SueldoBase + Incentivo*TotalAñosTrabajados + HorasExtra
```

Donde:

- `SueldoBase` se toma directamente del objeto tipo `Empleado`.
- Incentivo es una constante declarada en el programa.

- `TotalAñosTrabajados` lo calcula la función como la diferencia entre el año actual y el `AñoIngreso` del empleado.
- `HorasExtra` es un valor que recibe la función como parámetro.



CAPÍTULO 3

Sobrecarga, plantillas y polimorfismo

En este capítulo se empleará el lenguaje ***C++*** para tratar tres temas relacionados con la programación orientada a objetos. Estos temas no aplican a todos los lenguajes de programación orientados a objetos, sin embargo, es importante estudiarlos debido a que constituyen poderosas herramientas de programación.

3.1 Sobrecarga

La ***sobrecarga*** es una característica que ofrece el lenguaje ***C++*** para aplicar una misma operación, a través de operadores o funciones, a diferentes tipos de datos. Se pueden sobrecargar operadores, por ejemplo **+**, *****, **–**, etcétera y funciones definidas por el propio usuario. La sobrecarga permite generalizar el uso de operadores y funciones.
A continuación se analizarán estos temas detalladamente.

3.1.1 Sobrecarga de operadores

La ***sobrecarga de operadores*** es el proceso de asignar dos o más operaciones al mismo operador. Es decir, permite asignar una o más funciones adicionales a un operador estándar, con el fin de que ésta sea llamada según el contexto en el cual se utilice el operador. Un operador sobrecargado *no* puede tener parámetros predeterminados. La sintaxis para sobrecargar un operador es:

```
tipo operator operador (parámetros)
{
   /* Instrucciones que forman el cuerpo del operador. */
}
```

donde `tipo` indica el tipo de resultado que produce el operador, **operator** es una palabra reservada y `operador` es el operador que se sobrecarga.

La siguiente tabla muestra algunos de los operadores que pueden sobrecargarse en *C++*:

TABLA 3.1 Operadores que pueden sobrecargarse en C++

+	++	&	=
*	<<	-	!
%	>>	/	|
>	<	^	==

El programa 3.1 presenta un ejemplo de sobrecarga de operadores en el cual se sobrecarga al operador + para permitir la suma de vectores.

Programa 3.1

```
/* Clase Vector en la cual se incluye un método para sumar vectores
➥sobrecargando el operador +. */
class Vector
{
   private:
      float CoordX, CoordY;
```

```
    public:
        Vector (float Val1= 0, float Val2= 0);
        void ImprimeVector();
        Vector operator+(Vector Vec);
};

/* Declaración del método constructor con parámetros predeterminados;
➥a los cuales, si no les especifican otros valores, se les asignará 0. */
Vector::Vector(float Val1, float Val2)
{
   CoordX= Val1;
   CoordY= Val2;
}

/* Método que imprime los valores de los atributos de un vector. */
void Vector::ImprimeVector()
{
   cout << "X: " << X << " Y: " << Y <<endl;
}

/* Método en el cual se sobrecarga el operador +; por lo tanto, el
➥operador + se podrá usar tanto para la suma aritmética como para suma
➥de vectores. Lo anterior da como resultado un objeto de tipo Vector. */
Vector Vector::operator+ (Vector Vec)
{
   return Vector(CoordX+Vec.CoordX, CoordY + Vec.CoordY);
}

/* Función que utiliza el operador + sobrecargado. Se declaran dos objetos
➥de tipo Vector y, por medio del operador +, se obtiene su suma. */
void UsaSobrecarga(void)
{
   Vector ObjVectorU(3, 1), ObjVectorV(1, 2), ObjVectorR;

   /* Se invoca al operador sobrecargado: se realiza la suma de
   ➥vectores. */
   ObjVectorR= ObjVectorU + ObjVectorV;

   ObjVectorR.ImprimeVector();
}
```

La clase `Vector`, del ejemplo anterior, define un método para sumar dos vectores. Para ello se sobrecargó el operador **+**. Como consecuencia, si el operador **+** se usa con operandos que sean objetos del tipo `Vector`, se estará invocando a este método. En cambio, si los operandos son números, se estará haciendo referencia a la suma aritmética.

Sobrecarga de los operadores de entrada >> y de salida <<

Un caso especial es la sobrecarga de los operadores << y >> utilizados en la salida y entrada de datos respectivamente. Estos operadores se encuentran en la biblioteca **<iostream.h>** de *C++*.

En el caso de la entrada de datos del teclado a la aplicación, se establece una relación entre una referencia a un objeto de la clase `istream` y una referencia a un objeto de la clase en la cual se está incluyendo la sobrecarga. La sintaxis es la que se muestra a continuación:

```
friend istream &operator>> (istream &, TipoDefUsuario &);
```

donde `TipoDefUsuario` corresponde al nombre de la clase en la cual se está definiendo la sobrecarga del operador >>.

En el caso de la salida de datos de la aplicación a la pantalla, se establece una relación entre una referencia a un objeto de la clase `ostream` y una referencia a un objeto de la clase en la cual se está incluyendo la sobrecarga. La sintaxis es:

```
friend ostream &operator<< (ostream &, TipoDefUsuario &);
```

donde `TipoDefUsuario` corresponde al nombre de la clase en la cual se está definiendo la sobrecarga del operador <<.

Observe que las funciones que se obtienen al sobrecargar los operadores se declaran como amigas (**friend**) de la clase en la cual se insertaron. Esto es para que dichas funciones, externas a la clase, puedan tener acceso a los miembros privados de la misma.

El programa 3.2 presenta un ejemplo basado en el programa 2.3 del capítulo anterior.

Programa 3.2

```
/* Clase Arbol con algunos atributos y métodos; en la sección pública
➥se incluyen dos funciones amigas en las cuales se sobrecargan los
➥operadores de salida y entrada, << y >>. */
class Arbol
{
   protected:
      int Edad;
      double Altura;
      char Nombre[64];
   public:
      Arbol();
      void ModificaEdad(int);
      void ModificaAltura(float);
      friend istream &operator>> (istream &, Arbol &);
      friend ostream &operator<< (ostream &, Arbol &);
};

/* Declaración del método constructor por omisión. */
Arbol::Arbol()
{  }

/* Método que modifica la edad de un árbol. */
void Arbol::ModificaEdad(int NuevaE)
{
   Edad= NuevaE;
}

/* Método que modifica la altura de un árbol. */
void Arbol::ModificaAltura(float NuevaA)
{
   Altura= NuevaA;
}

/* Declaración de la función amiga donde se usa el operador >>
➥sobrecargado. */
istream &operator>> (istream &Lee, Arbol &ObjArbol)
{
   cout<<"\n\nIngrese nombre del árbol:  ";
   Lee>>ObjArbol.Nombre;
   cout<<"\n\nIngrese altura del árbol:  ";
   Lee>>ObjArbol.Altura;
   cout<<"\n\nIngrese edad en número de años del árbol:  ";
   Lee>>ObjArbol.Edad;
   return Lee;
}
```

```
/* Declaración de la función amiga donde se usa el operador <<
➥sobrecargado. */
ostream &operator<< (ostream &Escribe, Arbol &ObjArbol)
{
   cout<<"\n\nDatos del árbol:  ";
   Escribe<<"Nombre:  "<<ObjArbol.Nombre<<endl;
   Escribe<<"Altura:  "<<ObjArbol.Altura<<endl;
   Escribe<<"Edad:    "<<ObjArbol.Edad<<endl;
   return Escribe;
}

/* Función que usa la clase previamente definida. En esta función puede
➥apreciar cómo simplificar la entrada/salida de los datos de un objeto.
➥La escritura de las funciones amigas implica más código, sin embargo,
➥su uso produce un código más legible. */
void UsaSobreCarga()
{
   Arbol DeMiCampo;

   /* Se usa el operador >> sobrecargado para leer un objeto tipo
   ➥Arbol como si fuera un dato simple. */
   cin>>DeMiCampo;

   DeMiCampo.ModificaAltura(12.5);
   DeMiCampo.ModificaEdad(3);

   /* Se usa el operador << sobrecargado para imprimir un objeto tipo
   ➥Arbol como si fuera un dato simple. */
   cout<<DeMiCampo;
}
```

El ejemplo anterior contiene operadores sobrecargados, que leen e imprimen el objeto `DeMiCampo` como si fuera un dato simple. De esta manera se gana generalidad, ya que es posible usar las instrucciones **cin** y **cout** independientemente del tipo de dato que se esté leyendo o escribiendo.

3.1.2 Sobrecarga de funciones o métodos

La ***sobrecarga de funciones*** es el proceso de definir dos o más funciones, con el mismo nombre, que difieren únicamente en los parámetros que requieren y en el tipo de resultado que generan. Este tipo de sobrecarga resulta ser una poderosa

herramienta de programación. Sin embargo, debe ser cuidadoso en su uso ya que si se utiliza excesivamente el programa podría resultar poco legible. Además, es importante considerar que no es posible definir dos funciones que difieran sólo en el tipo de resultado. Deben hacerlo también en la lista de parámetros.

A continuación se presenta un ejemplo sencillo para que pueda comprender mejor el concepto explicado. En este caso se sobrecarga la función potencia, de tal forma que se pueda aplicar a números enteros o a números de tipo **double**.

Programa 3.3

```
/* Versión de la función Potencia para trabajar con números enteros. */
int Potencia (int Num, int Pot)
{
   int Indice, Res= 1;
   for (Indice= 1; Indice <= Pot; Indice++)
       Res= Res * Num;
   return Res;
}


/* Versión de la función Potencia para trabajar con números de doble
➥precisión. */
double Potencia (double Num, int Pot)
{
   double Res= 1;
   int Indice;
   for (Indice= 1; Indice <= Pot; Indice++)
       Res= Res * Num;
   return Res;
}


/* Función que utiliza las funciones sobrecargadas previamente definidas. */
void UsaFuncionesSobrecargadas()
{
   int Base1, Expo1, Expo2;
   double Base2;

   cout<< "Ingrese base y exponente - ambos números enteros - \n ";
   cin>>Base1>>Expo1;

   /* Se invoca a la función Potencia con un número entero como primer
   ➥parámetro, por lo tanto se ejecutará la primera versión presentada y
   ➥se obtendrá un número entero como resultado. */
   cout<<"\n\nEl resultado es: "<<Potencia(Base1, Expo1);
```

```
   cout<< "Ingrese base y exponente - la base puede ser un valor de
         ➥doble precisión - \n ";
   cin>>Base2>>Expo2;

   /* Se invoca a la función Potencia con un número de doble precisión
   ➥como primer parámetro, por lo tanto se ejecutará la segunda versión
   ➥presentada y se obtendrá un número de doble precisión como
   ➥resultado. */
   cout<<"\n\nEl resultado es: "<<Potencia(Base2, Expo2);
}
```

El programa 3.4 presenta un segmento de programa en el cual se incluyen dos funciones sobrecargadas: `Max`, que permite encontrar el máximo entre dos números que pueden ser del tipo **`int`** o **`double`** y `Raiz2`, que calcula la raíz cuadrada de un número que puede ser del tipo **`int`** o **`double`**.

Programa 3.4

```
/* Se define la función Max de tipo int, con parámetros también de tipo
➥int. Esta función compara dos valores enteros y regresa el valor más
➥grande. */
int Max (int Val1, int Val2)
{
   if (Val1 > Val2)
       return Val1;
   else
       return  Val2;
}

/* Se define la función Max de tipo double, con parámetros también de
➥tipo double. Esta función compara dos valores de tipo double y regresa
➥el valor más grande. */
double Max (double Val1, double Val2)
{
   if (Val1 > Val2)
       return Val1;
   else
       return Val2;
}

/* Se define la función Raiz2, con un parámetro de tipo int. Regresa la
➥raíz cuadrada del dato. */
double Raiz2 (int Num)
{
   return sqrt (Num);
}
```

```
/* Se define la función Raiz2, con un parámetro de tipo double. Regresa
➥la raíz cuadrada del dato. */
double Raiz2 (double Num)
{
   return sqrt (Num);
}


/* Función que usa las funciones sobrecargadas definidas previamente. */
void UsaSobrecargaFunciones()
{
   /* Se imprime el resultado de la función Max, primero invocándola con
   ➥valores enteros y luego con valores de doble precisión. */
   cout<< "Max de 2 y 5 es: " << Max(2, 5) << endl;
   cout<< "Max de 5.23 y 6.98 es: " << Max(5.23, 6.98) << endl;

   /* Se imprime el resultado de la función Raiz2, primero invocándola
   ➥con valores enteros y luego con valores de doble precisión. */
   cout << "Raíz cuadrada de 5 es: " << Raiz2(5) << endl;
   cout << "Raíz cuadrada de 8.96 es: " << Raiz2(8.96) << endl;
}
```


Por último, el programa 3.5 define la clase *Complejo* haciendo uso de sobrecarga de operadores y de funciones o métodos.

Programa 3.5

```
/* Se define la clase Complejo en la cual, algunos de los métodos se defi-
➥nieron sobrecargando operadores. Además, uno de esos métodos se sobre-
➥cargó. Es decir, usa un operador sobrecargado y tiene asociadas dos
➥funciones dependiendo de los parámetros con los cuales se invoque. */

/* Definición de la clase Complejo. */
class Complejo
{
   private:
      double Real, Imaginario;
   public:
      Complejo (double R= 0, double I= 0);
      Complejo operator+ (Complejo);
      Complejo operator- (Complejo);
      Complejo operator- ();
      void MuestraComplejo();
};
```

```
/* Declaración del método constructor con parámetros predeterminados: si
➥al crear un objeto no se dan valores al constructor, éste le asignará 0
➥a los dos atributos. */
Complejo::Complejo(double R, double I)
{
   Real= R;
   Imaginario= I;
}

/* Método que suma dos números complejos. Se sobrecarga el operador +. */
Complejo Complejo::operator+ (Complejo Com)
{
   return Complejo(Real + Com.Real, Imaginario + Com.Imaginario);
}

/* Método que resta dos números complejos. Se sobrecarga el operador -. */
Complejo Complejo::operator- (Complejo Com)
{
   return Complejo(Real - Com.Real, Imaginario - Com.Imaginario);
}

/* Método que cambia el signo de un número complejo. Se sobrecarga el
➥operador - y el método operator -. */
Complejo Complejo::operator- ()
{
   return Complejo (-Real, -Imaginario);
}

/* Método que imprime los valores de los atributos de un complejo. */
void Complejo::MuestraComplejo()
{
   cout<< "Parte Real: " << Real << endl;
   cout<< "Parte Imaginaria: " << Imaginario << endl;
}

/* Función que usa sobrecarga de operadores y de métodos: se declaran
➥objetos de tipo Complejo y se opera con ellos utilizando los operadores
➥y el método sobrecargados. */
void UsaSobrecargaOpMetodo()
{
   Complejo ObjComplejo1(5, 2), ObjComplejo2(2, 6),  ObjComplejo3;

   /* Se invoca al método que suma números complejos y luego al que
   ➥imprime. */
   ObjComplejo3= ObjComplejo1 + ObjComplejo2;
   ObjComplejo3.MuestraComplejo();
```

```
    /* Se invoca al método que resta números complejos y luego al que
    ➥imprime. */
    ObjComplejo3= ObjComplejo1 - ObjComplejo2;
    ObjComplejo3.MuestraComplejo();

    /* Se invoca al método que cambia el signo de un número complejo y
    ➥luego al método que imprime. */
    ObjComplejo3= -ObjComplejo1;
    ObjComplejo3.MuestraComplejo();
}
```

En el ejemplo anterior, el operador **–** se utilizó en dos métodos: en el primero se sobrecargó para realizar la resta de números complejos y en el segundo se utilizó para cambiar el signo de un número complejo. En este caso, además de sobrecargar al operador, se sobrecargó el método **operator –**. En el momento de usar los métodos, es el número de parámetros quien decide cuál de los dos se está invocando.

3.2 Plantillas

El lenguaje de programación ***C++*** ofrece otro recurso para ganar generalidad en la definición de soluciones: ***las plantillas***. Éstas permiten declarar funciones o clases dejando sin especificar el tipo de algunos de sus parámetros y/o datos (en el caso de las funciones) o el tipo de algunos de sus miembros (en el caso de las clases). A continuación se analizarán detalladamente las plantillas de funciones y las plantillas de clases.

3.2.1 Plantillas de funciones

Una ***plantilla de función*** es un modelo de función que el compilador de ***C++*** usará para construir diferentes versiones de una misma función, según los tipos de datos que se especifiquen al invocar a la misma. La plantilla permite escribir funciones que difieren exclusivamente en el tipo de datos que manejan.

Para definir una plantilla de función se aplica la siguiente sintaxis:

```
template <class T1, class T2, ..., class Tn>
```

donde **template** es una palabra reservada, lo mismo que **class**. Por su parte, `Ti` indica el tipo del `dato i`.

El programa 3.6 presenta las funciones del programa 3.4, pero ahora utiliza plantillas de funciones. Observe que cada función está precedida por las palabras reservadas **template <class T>**, que indican que la función que se define a continuación es una plantilla. Por lo tanto algunos de sus datos pueden quedar con sus tipos indefinidos, y los mismos tomarán valores al momento de invocar a la función.

Programa 3.6

```
/* Se declaran plantillas de funciones para Max y Raiz2. De esta manera
➥las mismas podrán trabajar sobre diferentes tipos de datos. Es decir,
➥en el momento de invocar a las funciones y al darles parámetros
➥específicos, se estarán creando versiones diferentes de las mismas,
➥de acuerdo a los tipos de los datos proporcionados. */

/* El objetivo de esta plantilla de función es encontrar el mayor de dos
➥valores dados. */
template <class T>
T Max (T Val1, T Val2)
{
   if (Val1 > Val2)
        return Val1;
   else
        return Val2;
}


/* El objetivo de esta plantilla de función es calcular la raíz cuadrada
➥de un valor dado. */
template <class T>
double Raiz2 (T Num)
{
   return sqrt (Num);
}


/* Función que utiliza las plantillas de funciones previamente
➥definidas. */
void UsaPlantilla()
{
   /* En las siguientes dos líneas se invocan las versiones enteras y de
   ➥punto flotante de la función Max, respectivamente. */
   cout<< "Max de 2 y 5 es: " << Max(2, 5) << endl;
   cout<< "Max de 5.23 y 6.98 es: " << Max(5.23, 6.98) << endl;
```

```
    /* En las siguientes dos líneas se invocan a las versiones enteras
    ➥y de punto flotante de la función Raiz2, respectivamente. */
    cout<< "Raiz2 de 5 es: " << Raiz2(5) << endl;
    cout<< "Raiz2 de 8.96 es: " << Raiz2(8.96) << endl;
}
```

El uso de plantillas de funciones es generalizado a diferentes tipos de datos. La sobrecarga de funciones obtiene el mismo efecto, pero usando más código.

3.2.2 Plantillas de clases

Las ***plantillas de clases*** permiten definir versiones de una misma clase que difieren en el tipo de dato de alguno(s) de sus miembros. Es decir, se crea el modelo de una clase el cual permitirá definir distintas instancias de la misma para diferentes tipos de datos.

Para declarar una plantilla de clase se usan las palabras reservadas **`template <class T>`**. El tipo **`T`** se usa en aquellos miembros de la clase cuyos tipos tomarán un valor en el momento de crear los objetos. A continuación se muestra la sintaxis que se utiliza para definir una plantilla de clase.

```
template <class T>
class PlantillaClase
{
   private:
       T Atributo;
         ...
   public:
      PlantillaClase();
      T Metodo1 ();
      void Metodo2 (T Valor);
          ...
};
```

Para declarar un objeto, a partir de una plantilla de clase, se aplica la siguiente sintaxis:

```
PlantillaClase <tipo> Objeto;
```

donde `tipo` indica el tipo de dato que reemplazará todas las ocurrencias de **`T`** en la definición de la clase. Por ejemplo, si quisiera declarar dos objetos de tipo `PlantillaClase`, pero uno con el tipo **`int`** como tipo de dato, y otro con **`float`**, entonces tendría que hacer lo siguiente:

```
PlantillaClase <int> Objeto1;

PlantillaClase <float> Objeto2;
```

Para `Objeto1` el `Atributo`, el resultado del `Metodo1` y el parámetro del `Metodo2` serán de tipo **`int`**, mientras que para el `Objeto2` el `Atributo`, el resultado del `Metodo1` y el parámetro del `Metodo2` serán de tipo **`float`**.

En cuanto a la definición de métodos, la sintaxis que se aplica es:

```
template <class T>
T  PlantillaClase <T>:: Metodo1()
{ ...}
```

En este caso, el método de la clase da un resultado del tipo **`T`**. Por lo tanto, el tipo de resultado se definirá en el momento de crear un objeto de dicha clase.

```
template <class T>
void  PlantillaClase <T>::Metodo2(T valor)
{...}
```

En este caso, el método recibe un parámetro que será del tipo **`T`**. Por lo tanto, el tipo se especifica en el momento de declarar el objeto.

A continuación se presenta un segmento de programa que define una plantilla de clase. Esta plantilla maneja dos tipos de datos diferentes (**`T1`** y **`T2`**) para declarar a los miembros de la clase.

Programa 3.7

```
/* La clase EjemploPlantilla tiene dos miembros privados, cada uno de un
➥tipo diferente, por lo que se usan los tipos T1 y T2 para indicarlo.
➥Asimismo, en los métodos definidos se utilizan T1 y T2 para dar
➥flexibilidad en cuanto a los tipos de datos. */

template <class T1, class T2>
class EjemploPlantilla
{
   private:
       T1 Dato1;
       T2 Dato2;
   public:
       EjemploPlantilla ();
       EjemploPlantilla (T1, T2);
       void ModificaDato1(T1);
       void ModificaDato2(T2);
       T1 ObtieneDato1();
       T2 ObtieneDato2();
       void ImprimeDatos();
};

/* Declaración del método constructor por omisión. */
template <class T1, class T2>
EjemploPlantilla<T1,T2>::EjemploPlantilla()
{ }

/* Declaración del método constructor con parámetros. */
template <class T1, class T2>
EjemploPlantilla<T1,T2>::EjemploPlantilla (T1 D1, T2 D2)
{
   Dato1= D1;
   Dato2= D2;
}

/* Plantilla del método que permite modificar el valor del atributo
➥Dato1. */
template <class T1, class T2>
void EjemploPlantilla<T1,T2>::ModificaDato1(T1 NuevoDato)
{
   Dato1= NuevoDato;
}

/* Plantilla del método que permite modificar el valor del atributo
➥Dato2. */
template <class T1, class T2>
void EjemploPlantilla <T1,T2>::ModificaDato2(T2 NuevoDato)
```

```
{
   Dato2= NuevoDato;
}

/* Plantilla del método que permite, a usuarios externos a la clase,
➥conocer el valor del atributo Dato1. */
template <class T1, class T2>
T1 EjemploPlantilla <T1,T2>::ObtieneDato1()
{
   return Dato1;
}

/* Plantilla del método que permite, a usuarios externos a la clase,
➥conocer el valor del atributo Dato2. */
template <class T1, class T2>
T2 EjemploPlantilla <T1,T2>::ObtieneDato2()
{
   return Dato2;
}

/* Plantilla del método que imprime los valores de los atributos. */
template <class T1, class T2>
void EjemploPlantilla <T1,T2>::ImprimeDatos()
{
   cout<< "Dato 1: " << Dato1 << endl;
   cout<< "Dato 2: " << Dato2 << endl;
}


/* Función que usa la plantilla de la clase EjemploPlantilla previamente
➥definida: se declara un objeto usando los tipos int y float para
➥instanciar los tipos T1 y T2 en la plantilla. Luego se modifican sus
➥atributos y se imprimen. */
void UsaPlantilla()
{
   EjemploPlantilla<int, float> ObjPlantilla(1, 6.0);

   ObjPlantilla.ImprimeDatos();
   ObjPlantilla.ModificaDato1(2);
   ObjPlantilla.ModificaDato2(12.0);

   cout<< "Dato 1 modificado : " << ObjPlantilla.ObtieneDato1() << endl;
   cout<< "Dato 2 modificado : " << ObjPlantilla.ObtieneDato2() << endl;
}
```

El programa 3.8 muestra otro caso de plantilla de clase y su uso.

Programa 3.8

```
/* Se define la plantilla de la clase Segmento. De esta forma cuando se
➥declare un objeto de la clase Segmento se podrá decidir el tipo de dato
➥para sus miembros.*/
template <class T>
class Segmento
{
   private:
       T Origen, Final;
   public:
       Segmento();
       Segmento(T, T);
       void ModificaOrigen(T);
       void ModificaFinal(T);
       T ObtieneOrigen();
       T ObtieneFinal();
       void ImprimeDatos();
};

/* Declaración del método constructor por omisión. */
template <class T>
Segmento<T>::Segmento()
{ }

/* Declaración del método constructor con parámetros. */
template <class T>
Segmento<T>::Segmento(T Or, T Fi)
{
   Origen= Or;
   Final= Fi;
}

/* Plantilla del método que permite modificar el valor del atributo
➥Origen. */
template <class T>
void Segmento<T>::ModificaOrigen(T NuevoPunto)
{
   Origen= NuevoPunto;
}

/* Plantilla del método que permite modificar el valor del atributo
➥Final. */
template <class T>
void Segmento<T>::ModificaFinal(T NuevoPunto)
```

```
{
   Final= NuevoPunto;
}

/* Plantilla del método que permite, a usuarios externos a la clase,
➥conocer el valor del atributo Origen. */
template <class T>
T Segmento<T>::ObtieneOrigen()
{
   return Origen;
}

/* Plantilla del método que permite, a usuarios externos a la clase,
➥conocer el valor del atributo Final. */
template <class T>
T Segmento<T>::ObtieneFinal()
{
   return Final;
}

/* Plantilla del método que imprime los valores de los atributos de la
➥clase. */
template <class T>
void Segmento<T>::ImprimeDatos()
{
   cout<< "Origen: " << Origen << endl;
   cout<< "Final: " << Final << endl;
}

/* Función que usa la plantilla de la clase Segmento: se crean dos objetos,
➥uno con números enteros y otro con números reales. Posteriormente se
➥modifican y se imprimen los valores de los atributos de los objetos
➥creados. */
void UsaPlantilla()
{
   Segmento<int>  SegmentoEntero(1, 6);
   Segmento<float> SegmentoReal(2.0, 15.0);

   cout<< "Datos del primer segmento:  " << endl;
   SegmentoEntero.ImprimeDatos();

   cout<< "Datos del segundo segmento: " << endl;
   SegmentoReal.ImprimeDatos();

   SegmentoEntero.ModificaOrigen(SegmentoEntero.ObtenerOrigen() + 2);
   SegmentoReal.ModificaFinal(SegmentoReal.ObtenerFinal() - 5.3);
```

```
    cout<< "Datos del primer segmento modificado: ";
    SegmentoEntero.ImprimeDatos();
    cout << "Datos del segundo segmento modificado: " << endl;
    SegmentoReal.ImprimeDatos();
}
```

En el ejemplo anterior se crearon dos instancias de la clase `Segmento`. Una de ellas usando números enteros, mientras que en la segunda se emplearon valores reales. Al utilizar la plantilla, cada uno de los miembros de la clase se instancia de acuerdo al tipo de dato que acompaña la declaración de los objetos.

El programa 3.9 presenta una plantilla de clase en la cual el tipo `T` se instancia con una clase previamente definida. Se debe poner especial atención en que todos los operadores y funciones utilizados en la plantilla estén definidos para el tipo usado. En el ejemplo fue necesario sobrecargar los operadores de lectura y escritura `>>` y `<<`.

Programa 3.9

```
/* Se define la clase Fabricante. Luego se define la plantilla de la
➥clase Producto que tiene un atributo, SeCompraA, que es un objeto de
➥tipo T. En el ejemplo, primero toma el tipo Fabricante y luego el tipo
➥int. Por lo tanto, en el primer caso se tendrá que un atributo de la clase
➥es, a su vez, un objeto, y en el segundo caso, el atributo representará
➥una clave numérica que identificará a un proveedor. Para que los métodos
➥de la segunda clase puedan utilizarse indistintamente con números o con
➥objetos se deben sobrecargar los operadores >> y <<. */

#define MAX 64

class Fabricante
{
   private:
       char  Nombre[MAX], Domicilio[MAX], Telefono[MAX];
   public:
       Fabricante();
       Fabricante(char  [], char [], char []);
       void  CambiaDomic(char []);
       void  CambiaTelef(char []);
       friend istream &operator>>(istream &, Fabricante &);
       friend ostream &operator<<(ostream &, Fabricante &);
};
```

```
/* Definición del método constructor por omisión. */
Fabricante::Fabricante()
{}

/* Definición del método constructor con parámetros. */
Fabricante::Fabricante(char  Nom[], char  Domic[], char  Tel[])
{
   strcpy(Nombre, Nom);
   strcpy(Domicilio, Domic);
   strcpy(Telefono, Tel);
}

/* Declaración del método que permite actualizar el domicilio de un
➥fabricante. */
void  Fabricante::CambiaDomic(char  NuevoDom[])
{
   strcpy(Domicilio, NuevoDom);
}

/* Declaración del método que permite actualizar el teléfono de un
➥fabricante. */
void  Fabricante::CambiaTelef(char  NuevoTel[])
{
   strcpy(Telefono, NuevoTel);
}

/* Definición de la sobrecarga del operador >>. */
istream &operator>>(istream &Lee, Fabricante &ObjFab)
{
   cout<<"\n\nIngrese nombre del fabricante:  ";
   Lee>>ObjFab.Nombre;
   cout<<"\n\nIngrese domicilio del fabricante:  ";
   Lee>>ObjFab.Domicilio;
   cout<<"\n\nIngrese teléfono del fabricante:  ";
   Lee>>ObjFab.Telefono;
   return  Lee;
}

/* Definición de la sobrecarga del operador <<. */
ostream &operator<<(ostream &Escribe, Fabricante &ObjFab)
{
   cout<<"\n\nDatos del fabricante\n  ";
   Escribe<<"Nombre:  "<<ObjFab.Nombre<<endl;
   Escribe<<"Domicilio:  "<<ObjFab.Domicilio<<endl;
   Escribe<<"Teléfono:    "<<ObjFab.Telefono<<endl;
   return  Escribe;
}
```

```
/* Definición de la plantilla de la clase Producto. */
template <class T>
class Producto
{
   private:
       int Clave;
       char  Nombre[MAX];
       float  Precio;
       T SeCompraA;
   public:
       Producto();
       Producto(int, char [], float, T);
       void  Imprime();
       void  ActualizaPrecio(float );
};

/* Definición de la plantilla del método constructor por omisión. */
template <class T>
Producto<T>::Producto()
{}

/* Definición de la plantilla del método constructor con parámetros. */
template <class T>
Producto<T>::Producto(int Cla, char  Nom[], float  Pre, T Provee)
{
   Clave= Cla;
   strcpy(Nombre, Nom);
   Precio= Pre;
   SeCompraA=  Provee;
}

/* Definición de la plantilla del método que despliega en pantalla los
➥valores de los atributos. */
template <class T>
void  Producto<T>::Imprime()
{
   cout<<"\n\nDatos del producto\n\n";
   cout<<"\nClave: "<<Clave;
   cout<<"\nNombre: "<<Nombre;
   cout<<"\nPrecio: "<<Precio;
   cout<<"\nProvisto por: "<<SeCompraA<<endl;
}

/* Definición de la plantilla del método que actualiza el valor del
➥precio de un producto. */
template <class T>
void  Producto<T>::ActualizaPrecio(float  NuevoPre)
```

```
{
   Precio= NuevoPre;
}

/* Función que utiliza la plantilla de la clase Producto, usando la
➥clase Fabricante y el tipo int para darle valor a T. La aplicación
➥es muy simple: se declaran y crean objetos del tipo Producto usando
➥los tipos ya mencionados. */
void  FuncionUsaPlantilla()
{
   Fabricante CablesMexico;
   int ClaProveedor;

   /* Se lee un objeto de tipo Fabricante, usando el operador
   ➥sobrecargado >>. */
   cin>>CablesMexico;

   /* Se crea un objeto de tipo Producto, reemplazando el tipo T por un
   ➥objeto de tipo Fabricante. */
   Producto<Fabricante> CableTel (1050, "Cable telefónico", 100,
   ➥CablesMexico);

   CableTel.Imprime();
   CableTel.ActualizaPrecio(105);

   cout<<"\n\nIngrese la clave del proveedor de las cajas
   ➥concentradoras: ";
   cin>>ClaProveedor;

   /* Se crea un objeto de tipo Producto, reemplazando el tipo T por
   ➥int. */
   Producto<int> Cajas (2600, "Cajas concentradoras", 450,
   ➥ClaProveedor);
   Cajas.Imprime();
}
```

En el ejemplo anterior se puede apreciar que el uso de plantillas de funciones da mucha generalidad a las clases, en cuanto al manejo de los tipos de datos. A partir de la misma plantilla de clase se crearon dos objetos, asignándole a cada uno un tipo de dato diferente para el atributo `SeCompraA` y en consecuencia dándole a cada uno capacidades distintas para representar y almacenar información.

3.3 Polimorfismo

El término ***polimorfismo*** hace referencia a la capacidad de adoptar diversas formas. Por lo tanto, un objeto polimórfico es aquel que tiene diversos aspectos. El polimorfismo permite que un mismo método adquiera distintos contenidos declarando funciones o métodos virtuales en la clase base y otras formas de los mismos en las clases derivadas.

Por medio del polimorfismo se puede definir un solo método para objetos diferentes, es decir, objetos que son instancias de distintas clases. En ***C++*** el polimorfismo se define a través de funciones virtuales. Por lo tanto, antes de presentar un ejemplo de polimorfismo se hará una breve introducción a las funciones virtuales.

3.3.1 Funciones virtuales

Las ***funciones*** o ***métodos virtuales*** se usan en clases base para indicar que puede haber múltiples formas de ellas en las clases derivadas. Para indicar que un método es virtual se antepone la palabra reservada **`virtual`**.

El programa 3.10 muestra el uso del polimorfismo. Crea una clase base que tiene una función virtual, misma que será redefinida en cada una de las clases derivadas.

Programa 3.10

```
/* Se define la clase Insecto que incluye un método virtual, el cual
➥se redefinirá en las clases derivadas: Mosca y Cucaracha. El método
➥virtual Imprime adoptará diferentes formas según la declaración del
➥mismo en cada una de las clases derivadas. Además, en la clase se
➥incluyó un destructor virtual. */

class Insecto
{
   protected:
         char Nombre[30];
         int NumPatas;
         float TamCabeza, TamTorax, TamAbdomen;
   public:
         Insecto(char *, int, float, float, float);
         virtual void Imprime();
         virtual ~Insecto() { }
};
```

```
/* Declaración del método constructor con parámetros. */
Insecto::Insecto(char *Nom, int NumP, float TamC, float TamT, float TamA)
{
   strcpy(Nombre, Nom);
   NumPatas= NumP;
   TamCabeza= TamC;
   TamTorax= TamT;
   TamAbdomen= TamA;
}

/* Método que despliega los valores de los atributos de un insecto. */
void Insecto::Imprime()
{
   cout<< "Nombre: " << Nombre << endl ;
   cout<< "Número de Patas: " << NumPatas << endl;
   cout<< "Tamaño de Cabeza: " << TamCabeza << endl;
   cout<< "Tamaño de Tórax: " << TamTorax << endl;
   cout<< "Tamaño de Abdomen: " << TamAbdomen << endl;
}


/* Definición de la clase Mosca, derivada de la clase Insecto. En el
➥prototipo del método Imprime se puede omitir el uso de la palabra
➥virtual. Se la incluyó sólo para ofrecer mayor claridad. */
class Mosca: public Insecto
{
   private:
      int NumAlas;
   public:
      Mosca(char *, int, float, float, float, int);
      virtual void Imprime();
      ~Mosca() { }
};

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
Mosca::Mosca(char *Nom, int Pat, float Cab, float Tor, float Abd, int Alas):
       ➥Insecto(Nom, Pat, Cab, Tor, Abd)
{
   NumAlas= Alas;
}

/* Método que despliega los valores de los atributos de una mosca. */
void Mosca::Imprime()
{
   Insecto::Imprime();
   cout<< "Número de Alas: " << NumAlas << endl;
}
```

```
/* Definición de la clase Cucaracha derivada de la clase Insecto. En el
➥prototipo del método Imprime se puede omitir la palabra virtual, se la
➥incluyó sólo para ofrecer mayor claridad. */
class Cucaracha: public Insecto
{
   private:
        char CaractCuerpo[30];
   public:
        Cucaracha(char *, int, float, float, float, char *);
        virtual void Imprime();
        ~Cucaracha() { }
};

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
Cucaracha::Cucaracha(char *Nom, int Pat, float Cab, float Tor, float
          ➥Abd, char *Cuer): Insecto(Nom, Pat, Cab, Tor, Abd)
{
   strcpy(CaractCuerpo, Cuer);
}

/* Método que despliega los valores de los atributos de una cucaracha. */
void Cucaracha::Imprime()
{
   Insecto::Imprime();
   cout<< "Características del cuerpo: " << CaractCuerpo << endl;
}


/* Función que usa las clases previamente definidas: se declaran objetos
➥polimórficos y por medio de los métodos virtuales se trabaja con ellos. */
void UsaFuncionVirtual()
{
   /* Se crean dos apuntadores a objetos polimórficos. */
   Insecto *ObjInsecto1, *ObjInsecto2;

   Mosca ObjMosca("Mosca", 6, 3, 1, 2, 4);
   Cucaracha ObjCucaracha("Cucaracha", 6, 2, 8, 4, "Cuerpo Aplanado");

   /* Se asigna la dirección de los objetos de las clases derivadas a
   ➥los apuntadores a los objetos polimórficos. */
   ObjInsecto1= &ObjMosca;
   ObjInsecto2= &ObjCucaracha;

   /* Invoca al método correspondiente a la clase Mosca, a través del
   ➥objeto polimórfico. */
   ObjInsecto1 -> Imprime();
```

```
        /* Invoca al método correspondiente a la clase Mosca a través del
        ➥objeto tipo Mosca. */
        ObjMosca.Imprime();

        /* Invoca al método correspondiente a la clase Cucaracha, a través
        ➥del objeto polimórfico. */
        ObjInsecto2 -> Imprime();
        /* Invoca al método correspondiente a la clase Cucaracha a través
        ➥del objeto tipo Cucaracha. */
        ObjCucaracha.Imprime();
}
```

En el ejemplo anterior se puede apreciar que al declarar los objetos polimórficos, éstos pueden almacenar la dirección de diferentes tipos de objetos. Consecuentemente, cada uno de ellos tomará diferentes formas dependiendo de la clase a la cual pertenezca. En el ejemplo, la variable `ObjInsecto1` se declara como un apuntador a un objeto de tipo `Insecto`. Sin embargo, posteriormente se le asigna la dirección de uno tipo `Mosca`. Por lo tanto, la forma del objeto dependerá de la clase a la cual hace referencia en este caso. Lo mismo sucede con la variable `ObjInsecto2`, se declara como un apuntador a un objeto de tipo `Insecto` y posteriormente hace referencia a uno tipo `Cucaracha`.

El programa 3.11 presenta otro caso de uso de funciones virtuales y polimorfismo. En el ejemplo se usa un arreglo de objetos polimórficos, por lo que si aún no está familiarizado con esta estructura de datos se le recomienda consultar el capítulo 4.

Programa 3.11

```
/* Se declara la clase Volumen que servirá como base para las clases
➥derivadas: Libro y Revista. La clase base tiene métodos virtuales que
➥serán redefinidos en las clases derivadas. En la clase se define un
➥método destructor virtual. Observe que el método virtual Imprimir no se
➥define en la clase base, sólo se incluye su prototipo. Luego se define
➥la clase Biblioteca que tiene como atributo un arreglo de objetos
➥polimórficos. */

class Volumen
{
   protected:
      char *NomVolumen;
```

```
   public:
      Volumen();
      Volumen(char *);
      virtual void Imprimir() { }
      virtual ~Volumen();
};
/* Declaración del método constructor por omisión. */
Volumen::Volumen()
{ }

/* Declaración del método constructor con parámetros. */
Volumen::Volumen(char *Nom)
{
   NomVolumen = new char[(strlen(Nom)+1)];
   if (NomVolumen)
       strcpy(NomVolumen, Nom);
}

/* Declaración del método destructor. */
Volumen:: ~Volumen()
{
   delete[] NomVolumen;
}


/* Definición de la clase Libro derivada de la clase Volumen. El método
➥Imprimir se define en esta clase. */
class Libro: public Volumen
{
   private:
      int AnioEd;
   public:
      Libro();
      Libro(char *Nom, int);
      void Imprimir();
};

/* Declaración del método constructor por omisión. */
Libro::Libro()
{ }

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base*/
Libro::Libro(char *Nom, int Anio): Volumen(Nom)
{
   AnioEd= Anio;
}
```

```
/* Método que despliega los valores de los atributos de un libro. Observe
➥que se imprimen dos atributos uno de los cuales se hereda de la clase
➥Volumen y el otro es propio de esta clase. */
void Libro::Imprimir()
{
   cout<< "Nombre del Libro: " << NomVolumen << endl;
   cout<< "Año de Edición del Libro: " << AnioEd << endl;
}

/* Definición de la clase Revista derivada de la clase Volumen. El método
➥Imprimir se define en esta clase. */
class Revista: public Volumen
{
   private:
      int Numero;
   public:
      Revista();
      Revista(char *, int);
      void Imprimir();
};

/* Declaración del método constructor por omisión. */
Revista::Revista()
{ }

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
Revista::Revista(char *Nom, int Num): Volumen(Nom)
{
   Numero= Num;
}

/* Método que despliega los valores de los atributos de una revista.
➥Observe que se imprimen dos atributos, uno de los cuales se hereda de
➥la clase Volumen y el otro es propio de esta clase.*/
void Revista::Imprimir()
{
   cout<< "Nombre de la Revista: " << NomVolumen << endl;
   cout<< "Número de la Revista: " << Numero << endl;
}

/* Definición de la clase Biblioteca. Uno de los atributos de la clase
➥es un arreglo polimórfico, lo cual da mucha generalidad en el momento
➥de almacenar información en él: se pueden guardar objetos de diferentes
➥tipos. */
class Biblioteca
```

```
{
   private:
      int MaxVolumen,  NumVolumen;
      char Nombre[64];
      Volumen *Volumenes[];
   public:
      Biblioteca();
      Biblioteca(intl, char []);
      void IngresarVolumen(Volumen *);
      void Imprimir();
      ~Biblioteca();
};

/* Declaración del método constructor por omisión. */
Biblioteca::Biblioteca()
{ }

/* Declaración del método constructor con parámetros. */
Biblioteca::Biblioteca(int MaxVol, char Nom[])
{
   int Indice;
   MaxVolumen= MaxVol;
   NumVolumen= 0;
   strcpy(Nombre, Nom);
   *Volumenes= new Volumen[MaxVolumen];

   /* Se inicializa el arreglo de objetos polimórficos como vacío. */
   for (Indice= 0; Indice < MaxVolumen; Indice++)
        Volumenes[Indice]= NULL;
}

/* Declaración del método destructor. */
Biblioteca::~Biblioteca()
{
   delete[] *Volumenes;
}

/* Método que permite dar de alta un nuevo volumen en la colección de
➥volúmenes de la biblioteca. Recibe como parámetro la dirección de un
➥objeto de tipo Volumen. */
void Biblioteca::IngresarVolumen(Volumen *Vol)
{
   if (NumVolumen < MaxVolumen)
       Volumenes[NumVolumen++]= Vol;
}

/* Método que despliega los valores de los atributos de los volúmenes
➥registrados en la biblioteca. */
void Biblioteca::Imprimir()
```

```
{
   int Indice;
   cout<<" Acervo de la biblioteca: "<< Nombre<<endl;
   if (NumVolumen > 0)
       for (Indice= 0; Indice < NumVolumen; Indice++)
           Volumenes[Indice]->Imprimir();
}

/* Función que usa las clases previamente definidas para crear objetos
➥polimórficos. */
void UsaPolimorfismo()
{
   /* Se crea un objeto de tipo Biblioteca, el cual podrá almacenar 10
   ➥volúmenes como máximo. */
   Biblioteca ObjBiblioteca (10, "Refugio del Conocimiento");

   /* Se crean objetos tipo Libro. */
   Libro ObjLibro1 ("Estructuras de Datos", 2006),
         ObjLibro2 ("Aprenda C++", 2005),
         ObjLibro3 ("Estudie Ingeniería", 2000);

   /* Se crean objetos tipo Revista. */
   Revista ObjRevista1 ("Ciencia", 12),
           ObjRevista2 ("Computadoras y Accesorios", 110),
           ObjRevista3 ("Avances de la Tecnología", 205);

   /* Se invoca al método que permite asignar las direcciones de los
   objetos tipo Libro a uno de los miembros del objeto tipo Biblioteca. */
   ObjBiblioteca.IngresarVolumen(&ObjLibro1);
   ObjBiblioteca.IngresarVolumen(&ObjLibro2);
   ObjBiblioteca.IngresarVolumen(&ObjLibro3);

   /* Se invoca al método que permite asignar las direcciones de
   ➥los objetos tipo Revista a uno de los miembros del objeto tipo
   ➥Biblioteca. */
   ObjBiblioteca.IngresarVolumen(&ObjRevista1);
   ObjBiblioteca.IngresarVolumen(&ObjRevista2);
   ObjBiblioteca.IngresarVolumen(&ObjRevista3);

   /* Se invoca el método que despliega los valores de los atributos de
   ➥la biblioteca. Imprime el nombre de la biblioteca y los valores del
   ➥atributo de cada objeto de acuerdo a la forma que éste tenga. */
   ObjBiblioteca.Imprimir();
}
```

En el ejemplo anterior, un miembro de la clase `Biblioteca` se declara de tipo `Volumen` como objeto polimórfico. Por lo tanto podrá almacenar direcciones tanto de objetos tipo `Libro` como de objetos tipo `Revista`. Consecuentemente, cuando se invoque el método `Imprimir` de la clase `Biblioteca`, éste se aplicará según la forma del objeto almacenado.

3.3.2 Clases abstractas

Una ***clase abstracta*** es una clase que se define con el propósito de establecer bases conceptuales sobre las cuales se definirán otras clases, mismas que podrán ser clases concretas. Es decir, una clase abstracta no se usará directamente en la solución de un problema, sino que formará parte del diseño conceptual de la solución. Por lo tanto, en el programa no se crearán instancias (objetos) de las clases abstractas. Sin embargo, cabe destacar que las clases derivadas sí heredan sus miembros.

En una clase abstracta pueden incluirse métodos virtuales que requieren ser especificados en las clases derivadas. Es decir, métodos a los que se les asignará el contenido en cada clase derivada. Estos métodos reciben el nombre de ***métodos virtuales puros*** y se inicializan con el valor de cero. Si las clases derivadas no los especifican, entonces se producirá un error. A continuación se presenta un ejemplo de uso de clases abstractas.

Programa 3.12

```
/* Se define la clase Figura la cual se usará como base para declarar
➥las clases derivadas: Triangulo, Rectangulo y Cuadrado. La clase base es
➥una clase abstracta ya que no se crearán instancias de ella, sino que
➥se utiliza para crear una abstracción de un nivel superior de todas las
➥figuras geométricas. La clase abstracta contiene un método virtual puro
➥llamado CalculaArea(). */

/* Definición de la clase abstracta Figura. */
class Figura
{
   public:
      Figura();
      virtual float CalculaArea()= 0;
};
```

```
/* Declaración del método constructor por omisión. */
Figura::Figura()
{}

/* Definición de la clase Triangulo, derivada de la clase abstracta
➥Figura. Un triángulo se representa por medio de la longitud de su base
➥y de su altura. La clase incluye además, un método virtual llamado
➥CalculaArea(). */
class Triangulo: public Figura
{
   private:
        float Base, Altura;
   public:
        Triangulo(float, float);
        virtual  float CalculaArea();
};

/* Declaración del método constructor con parámetros. */
Triangulo::Triangulo(float Ba, float Alt)
{
   Base= Ba;
   Altura= Alt;
}

/* Método que calcula el área de un triángulo. */
float Triangulo::CalculaArea()
{
   return (Base * Altura / 2);
}


/* Definición de la clase Equilatero, derivada de la clase Triangulo. */
class TrianguloEquilatero: public Triangulo
{
   public:
       TrianguloEquilatero(float, float);
       float CalculaArea();
};

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
TrianguloEquilatero::TrianguloEquilatero(float Ba, float Alt): Triangulo
➥(Ba, Alt)
{}

/* Método que calcula el área de un triángulo equilátero. */
float TrianguloEquilatero::CalculaArea()
{
   return Triangulo::CalculaArea();
}
```

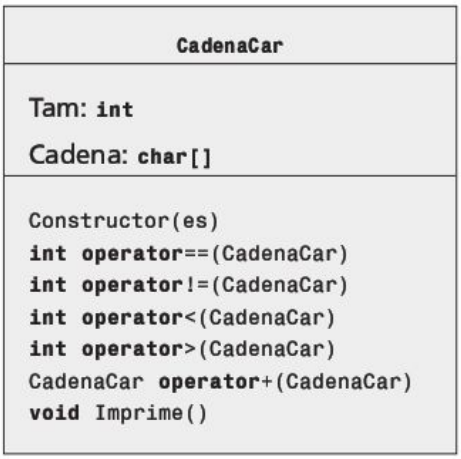
```
/* Definición de la clase TrianguloRectangulo, derivada de la clase
➥Triangulo. */
class TrianguloRectangulo: public Triangulo
{
   private:
      float Cateto1,  Cateto2,  Hipotenusa;
   public:
      TrianguloRectangulo (float, float);
      float CalculaArea();
};

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
TrianguloRectangulo::TrianguloRectangulo(float Cat1, float Cat2):
➥Triangulo(Cat1, Cat2)
{
   Cateto1= Cat1;
   Cateto2= Cat2;
   Hipotenusa= sqrt(Cat1*Cat1 + Cat2*Cat2);
}

/* Método que calcula el área de un triángulo rectángulo. */
float TrianguloRectangulo::CalculaArea()
{
   return (Cateto1*Cateto2);
}

/* Definición de la clase Rectangulo, derivada de la clase abstracta
➥Figura. */
class Rectangulo: public Figura
{
   private:
      float Largo, Alto;
   public:
      Rectangulo(float, float);
      float CalculaArea();
};

/* Declaración del método constructor con parámetros. */
Rectangulo::Rectangulo(float Lar, float Al)
{
   Largo= Lar;
   Alto= Al;
}

/* Método que calcula el área de un rectángulo. */
float Rectangulo::CalculaArea()
{
   return (Largo*Alto);
}
```

```
/* Definición de la clase Cuadrado, derivada de la clase Rectangulo. */
class Cuadrado: public Rectangulo
{
   public:
      Cuadrado(float);
      float CalculaArea();
};

/* Declaración del método constructor con parámetros. Invoca al método
➥constructor de la clase base. */
Cuadrado::Cuadrado(float Lado): Rectangulo(Lado, Lado)
{}

/* Método que calcula el área de un cuadrado, haciendo uso del método
➥heredado de la clase Rectangulo. */
float Cuadrado::CalculaArea()
{
   return Rectangulo::CalculaArea();
}

/* Función que usa las clases definidas previamente. Observe que no se
➥han creado objetos del tipo de la clase abstracta Figura. */
void UsaFiguras()
{
   TrianguloEquilatero  TriaEq(5,7);
   TrianguloRectangulo  TriaRec(3, 4);
   Rectangulo  Rectan(2, 3);
   Cuadrado  Cuadro(5);

   cout<< "\nÁrea del triángulo equilátero: " << TriaEq.CalculaArea();
   cout<< "\nÁrea del triángulo rectángulo: " << TriaRec.CalculaArea();
   cout<< "\nÁrea del rectángulo: " << Rectan.CalculaArea();
   cout<< "\nÁrea del cuadrado: " << Cuadro.CalculaArea();
}
```

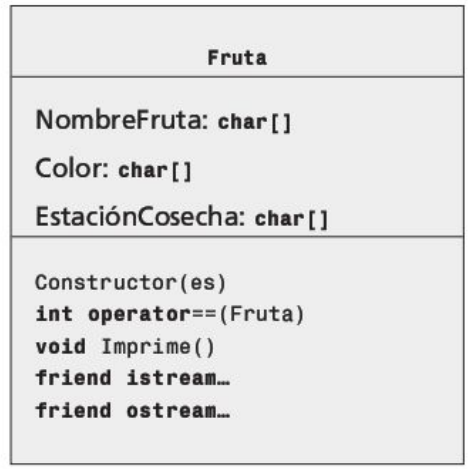
En el ejemplo anterior se definió la clase abstracta `Figura` la cual se utilizó como base para definir otras clases que representan figuras geométricas concretas. En la clase `Figura`, el método `CalculaArea()` se definió como un método virtual puro, ya que no tiene un conjunto de operaciones asociado. Se sabe que a toda figura geométrica se le puede calcular el área, sin embargo, la manera de calcularla dependerá de la figura que sea. Por lo tanto, este método se redefinirá en cada una de las clases derivadas de acuerdo a la figura geométrica que represente.

Ejercicios

1. Defina la clase `CadenaCar` según las especificaciones que se muestran a continuación. Incluya la sobrecarga de los siguientes operadores: **==**, **!=**, **+**, **<** y **>**, de tal manera que dos objetos tipo `CadenaCar` se puedan comparar (**==**, **!=**, **<**, **>**) o unir (**+**) usando los operadores indicados.



2. Retome la clase definida en el ejercicio anterior. Escriba un programa en ***C++*** que:
   a) Declare dos objetos tipo `CadenaCar`.
   b) Le asigne una cadena de caracteres a cada uno de los objetos. La asignación puede ser a través de una lectura o por medio del método constructor.
   c) Compare los objetos e imprima un mensaje adecuado si los mismos son iguales. Si no lo fueran, el mensaje, además de indicar este caso, debe decir cuál de las cadenas es menor.
   d) Enlace dos objetos tipo `CadenaCar` formando un tercer objeto del mismo tipo. Imprima el objeto resultante.
3. Defina la clase `Fruta` según las especificaciones que se muestran más adelante. Incluya la sobrecarga del operador `==` para determinar si dos objetos de tipo `Fruta` son iguales. Dos frutas se considerarán iguales si los valores de todos sus atributos son iguales. Además, sobrecargue los operadores de entrada (`>>`) y de salida (`<<`) para poder leer y escribir objetos de tipo `Fruta` con las instrucciones **`cin`** y **`cout`** respectivamente.

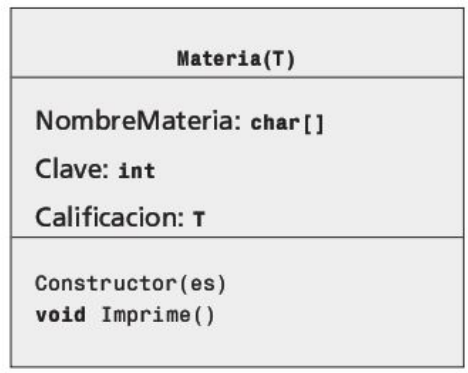


4. Retome la clase definida en el ejercicio anterior. Escriba un programa en *C++* que:

   a) Cree dos objetos tipo `Fruta`, asignándole valores a sus atributos por medio del operador `>>` sobrecargado.

   b) Compare los objetos e indique si son iguales. Imprima un mensaje adecuado.

5. Defina una plantilla para la clase `Materia`, de tal manera que el tipo de dato del atributo `Calificación` sea del tipo `T`. Esto permitirá crear objetos de tipo `Materia` que tengan calificaciones que sean: (a) Números enteros, por ejemplo, 8 o 9, (b) Números con decimales, por ejemplo 8.5 o (c) Letras, por ejemplo A.



6. Retome la clase definida en el ejercicio anterior. Escriba un programa en *C++* que:

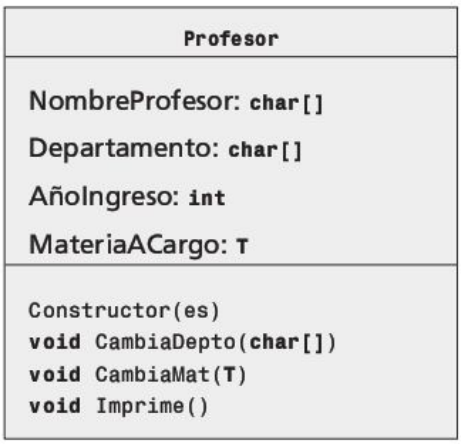
   a) Cree un objeto de tipo `Materia` usando el tipo `int`. Imprima los valores de los atributos del objeto creado.

   b) Cree un objeto de tipo `Materia` usando el tipo `float`. Imprima los valores de los atributos del objeto creado.

   c) Cree un objeto de tipo `Materia` usando el tipo `char`. Imprima los valores de los atributos del objeto creado.

   d) Incluya un método en la clase que permita modificar la calificación de una materia.

7. Defina la plantilla de la clase `Profesor` según las especificaciones que se dan más adelante. El atributo `MateriaACargo` es del tipo `T`, en este caso podría ser un entero (si la materia se representa por medio de una clave), una cadena de caracteres (si la materia se representa por su nombre) u otro objeto (si la materia se representa usando una clase previamente definida).



8. Retome la clase definida en el ejercicio anterior. Escriba un programa en *C++* que:

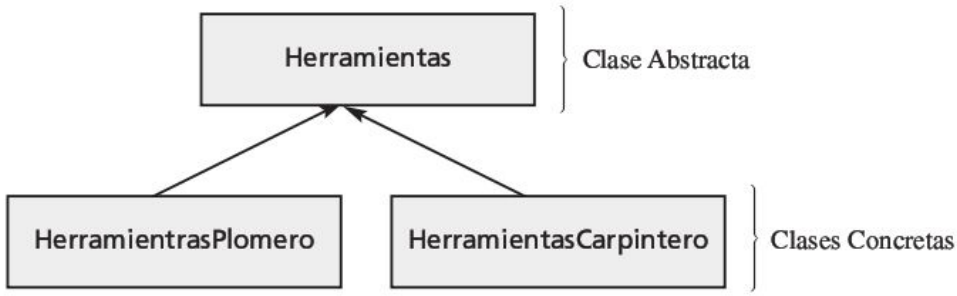
   a) Cree el objeto `ProfeJuan` de tipo `Profesor`, usando `int` para instanciar `T`.

   b) Imprima todos los datos del `ProfeJuan`.

   c) Cambie el nombre del departamento al cual está adscrito el `ProfeJuan`.

   d) Cambie la materia que tiene a cargo el `ProfeJuan`.

9. Retome el problema 8, pero ahora utilice una cadena de caracteres para instanciar `T`. ¿Debe modificar la plantilla de la clase `Profesor`?, o ¿debe definir alguna otra clase?
10. Retome el problema 8, pero ahora utilice una clase `Materia` para instanciar `T`. Puede usar la del problema 5 o definir su propia clase. ¿Debe modificar la plantilla de la clase `Profesor`?, o ¿debe modificar la otra clase? Si usó la plantilla del problema 5, ¿cuántos valores para `T` debe dar al crear un objeto de tipo `Profesor`?
11. Implemente la clase base `Cuadrilatero`, con atributos `Base` y `Altura` y un método `CalculaArea` que calcule el área del cuadrilátero. Implemente también las clases derivadas `Cuadrado`, `Rectangulo` y `Trapezoide`. Use un objeto polimórfico para calcular el área de un objeto de cada una de estas clases.
12. Considere la siguiente relación de herencia entre una clase abstracta y dos clases concretas. Decida qué atributos y métodos incluir de tal manera que su programa pueda:

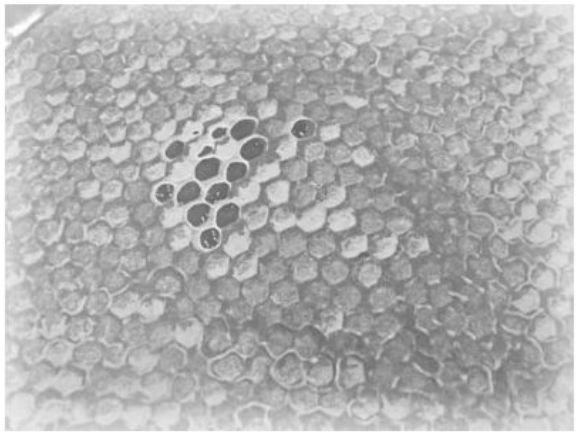


a) Crear un objeto llamado `Soldadora`, de tipo `HerramientasPlomero` y otro llamado `Serrucho`, de tipo `HerramientasCarpintero`.

b) Cambiar el precio del objeto `Soldadora`. El usuario dará como dato el nuevo precio.

c) Cambiar el color del objeto `Serrucho`.

d) Imprimir los datos de los objetos creados y modificados.



CAPÍTULO 4

Arreglos

4.1 Introducción

Una ***estructura de datos*** hace referencia a una colección de elementos y a la manera en que ésta se almacena en la memoria de la computadora y/o en algún dispositivo de memoria secundaria. Esta forma de almacenamiento determina la manera en que los datos se pueden recuperar. En este capítulo se presenta la estructura de datos tipo arreglo, que se utiliza para guardar información en la memoria principal.

Un ***arreglo*** es una colección finita, ordenada y homogénea de datos. Es finita porque todo arreglo tiene un tamaño límite, es decir, se define el número máximo de elementos que puede almacenar. Es ordenada porque permite hacer referencia al primer elemento, al segundo y así hasta el enésimo elemento que forme el arreglo. Por último, se di-

ce que es homogénea porque todos los componentes del arreglo son del mismo tipo de datos.

Un arreglo también se puede ver como una colección lineal de elementos, ya que cada uno de ellos sólo tiene un predecesor y un sucesor, con excepción del primero que sólo tiene sucesor y del último, que sólo tiene predecesor.

En todo arreglo se distinguen el nombre, los componentes y los índices. El nombre hace referencia a la estructura como un todo. Los componentes son los valores que forman el arreglo, es decir, cada uno de los datos que se almacenan en él. Mientras que los índices se utilizan para recuperar a cada uno de los componentes de manera individual. Gráficamente un arreglo puede representarse como se muestra en la figura 4.1.

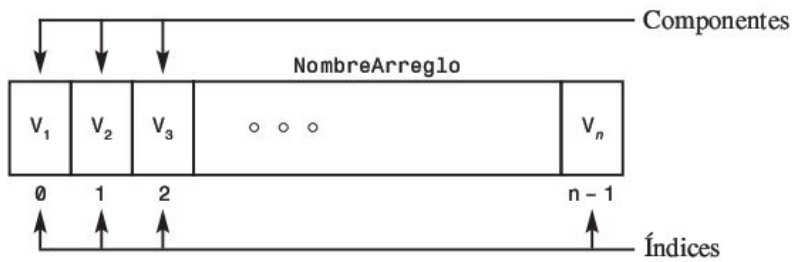


FIGURA 4.1 *Representación gráfica de un arreglo*

La figura 4.1 muestra que el nombre del arreglo es único e identifica a todo el conjunto de datos almacenados en la estructura. Por su parte $V_1, V_2, \ldots, V_n$ indican los valores almacenados en cada una de las casillas del arreglo. Los índices 0, 1, …, n–1 referencian a cada una de las celdas y por lo tanto permiten el acceso a cada uno de los valores almacenados en ellas. En el caso de los lenguajes de programación ***C*** y ***C++*** los índices se enumeran a partir del 0. Es decir, si se declara un arreglo de 20 elementos, las casillas se identificarán con los números del 0 al 19. La figura 4.2 presenta un ejemplo de un arreglo con capacidad para almacenar máximo 20 valores.

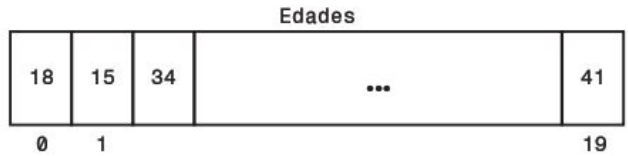


FIGURA 4.2 *Ejemplo de un arreglo*

En este ejemplo, el nombre del arreglo es `Edades`, los índices son valores enteros comprendidos entre el 0 y el 19, y los componentes también son números enteros que representan las edades de un grupo de personas.

4.2 La clase Arreglo

La clase `Arreglo` tiene como atributos la colección de elementos que forman la estructura de datos y el número actual de elementos, y como métodos el conjunto de operaciones que son aplicables a un arreglo. La figura 4.3 presenta la clase `Arreglo`. En este caso se define como plantilla para lograr mayor generalidad.

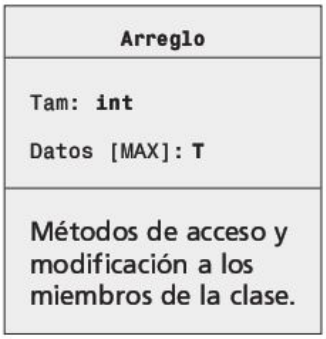


FIGURA 4.3 *Clase* `Arreglo`

A continuación se presenta la plantilla de la clase `Arreglo`, utilizando el lenguaje de programación ***C++***.

```
template <class T>
class Arreglo
{
   /* Miembros privados de la clase Arreglo. Datos representa una colección
   ➥de MAX elementos y Tam es el número actual de elementos que forman
   ➥parte de dicha colección. */
   private:
       T Datos[MAX];
       int Tam;
   public:
       Arreglo();

   /* En la sección pública, además del método constructor, se incluirán
   ➥los métodos de acceso y modificación a los miembros de la clase. */
};
```

En la sección privada de la clase se define el arreglo mediante la instrucción:

```
T Datos[MAX];
```

que significa que se tiene una colección de elementos, llamada `Datos`, que tiene una capacidad máxima de `MAX` (constante previamente definida) elementos y que todos los elementos son del tipo `T`. Según lo presentado en el capítulo 3, `T` se instanciará con el tipo de dato usado al declarar un objeto del tipo `Arreglo`. Además, se define el atributo `Tam` que representa el número actual de elementos que tiene el arreglo. Al declarar un objeto de este tipo se establece el máximo número de elementos que puede almacenar, pero el número de valores que finalmente se guardan depende de la aplicación.

Los métodos de acceso y modificación a los elementos del arreglo se estudiarán en la siguiente sección, considerando si los elementos del arreglo se encuentran o no ordenados, ya que esto condiciona la manera de llevar a cabo algunas de las operaciones sobre los mismos.

Para declarar un objeto se utiliza la siguiente sintaxis:

```
Arreglo<tipo> ObjArreglo;
```

Por ejemplo, para declarar el arreglo `Edades` de la figura 4.2 se haría:

```
Arreglo <int> Edades;
```

Antes de presentar las operaciones, resulta conveniente mencionar que el acceso a cada uno de los elementos se hace a través de los índices. La sintaxis es `Datos[i]`, donde `Datos` hace referencia a toda la colección y la i indica la casilla particular a la que se tendrá acceso. Por ejemplo, `Datos[0]` señala la primera casilla del arreglo y `Datos[1]` la segunda.

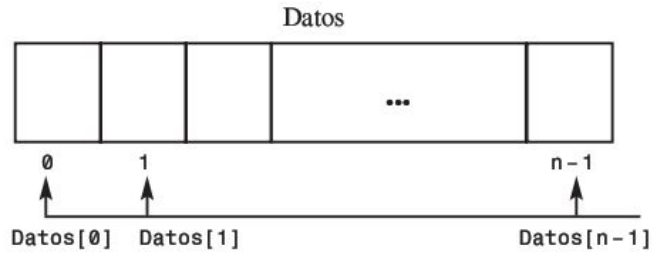


FIGURA 4.4 *Acceso a los componentes del arreglo*

4.3 Métodos de acceso y modificación a arreglos

En el primer capítulo se mencionó que es importante diseñar clases que tengan todas las operaciones que son aplicables a los miembros de la misma. Por lo tanto, en el caso de la clase `Arreglo` se incluirán las principales operaciones que pueden realizarse en esta estructura de datos. Las operaciones más importantes en las que intervienen objetos del tipo arreglo son:

- Lectura
- Escritura
- Eliminación
- Inserción
- Búsqueda
- Ordenación

A continuación se presentan las operaciones de lectura, escritura y eliminación en las cuales no afecta que el arreglo esté o no ordenado. La operación de inserción se analizará más adelante, tanto para los casos de arreglos desordenados como de arreglos ordenados.

En este capítulo se presentan, de manera introductoria, los métodos de búsqueda y ordenación. Sin embargo, debido a su importancia, estas operaciones se tratarán detalladamente en capítulos subsecuentes.

4.3.1 Lectura de arreglos

La lectura de un arreglo consiste en darle valores a los componentes del mismo mediante el ingreso de datos desde medios externos. Las fuentes más comunes de donde se pueden ingresar los datos son el teclado de la computadora y los archivos.

A continuación se presenta la plantilla de un método que permite la lectura de los valores de un arreglo desde el teclado. Observe que a medida que un dato se lee, éste se asigna automáticamente a una de las casillas (la indicada por el valor del índice).

```
template <class T>
Arreglo<T>::Lectura()
{
   int  Indice;
   /* Lectura del número de elementos a guardar en el arreglo. Se veri-
   ➥fica que el valor dado por el usuario sea  menor o igual que el má-
   ➥ximo permitido y mayor o igual a 1. */
   do {
      cout<<"\n\n Ingrese el total de elementos:  ";
      cin>> Tam;
   } while (Tam < 1 || Tam > MAX);
   for (Indice= 0; Indice < Tam; Indice++)
   {
      cout<<"\n Ingrese un dato:  ";
      cin>> Datos[Indice];
   }
}
```

En el método presentado se leen los primeros `Tam (1≤ Tam ≤MAX)` valores del arreglo. También se pueden leer los miembros de la clase `Arreglo` usando la sobrecarga del operador `>>` que se presentó en el capítulo 3. En la sección pública de la clase se debe incluir la siguiente declaración:

```
friend istream &operator>> (istream &, Arreglo &);
```

Luego se debe escribir el método de lectura (con la sobrecarga del operador `>>`) correspondiente a los miembros del arreglo.

```
istream &operator>> (istream &Lee,  Arreglo &ObjArre)
{
   int  Indice;
   /* Lectura del número de elementos a guardar en el arreglo. Se verifica
   ➥que el valor dado por el usuario sea menor o igual que el máximo
   ➥permitido y mayor o igual a 1. */
  do {
     cout<<"\n\n Ingrese el total de elementos:  ";
     cin>> Tam;
   } while (Tam < 1 || Tam > MAX);
   for (Indice= 0; Indice < Tam; Indice++)
     Lee>>ObjArre.Datos[Indice];
   return  Lee;
}
```

En ambas soluciones la lectura del tamaño se incluyó junto a la lectura de los elementos del arreglo. Otra posible solución es que el número actual de elementos se lea desde la aplicación (recuerde que uno de los objetivos de la POO es que el código sea reutilizable) y que dicho valor se asigne al atributo Tam a través de un constructor con parámetros, como se muestra a continuación.

```
template <class  T>
Arreglo<T>::Arreglo(int Valor)
{
     Tam= Valor;
}
```

En este caso, al declarar un objeto del tipo Arreglo se debe indicar el número de elementos que contendrá el arreglo inicialmente. Por ejemplo, para declarar un arreglo que almacene 10 números enteros se usará la siguiente sintaxis:

```
Arreglo <int> ObjArre(10);
```


4.3.2 Escritura de arreglos

La escritura de un arreglo consiste en imprimir el contenido de las casillas. Es decir, esta operación presupone que el arreglo tiene algunos valores asignados y éstos son los que se presentan a algún medio externo, como la pantalla de la computadora o un archivo.

A continuación se muestra la plantilla de un método que permite el despliegue en pantalla de los valores almacenados en un arreglo.

```
template <class  T>
Arreglo<T>::Escribe()
{
     int  Indice;
     for (Indice= 0; Indice < Tam; Indice++)
            cout<< Datos[Indice]<<"  ";
}
```

En el método presentado se despliega en pantalla el contenido de los primeros `Tam` (1≤ *Tam* ≤ *MAX*) valores del arreglo. También se pueden imprimir los miembros de la clase `Arreglo` usando la sobrecarga del operador << que se presentó en el capítulo 3. En la sección pública de la clase se debe incluir la siguiente declaración:

```
friend ostream &operator<< (ostream &, Arreglo &);
```

Luego se debe codificar el método de escritura (con la sobrecarga del operador <<) correspondiente a los miembros del arreglo.

```
ostream &operator<< (ostream &Escribe,  Arreglo &ObjArre)
{
   int  Indice;
   for (Indice= 0; Indice < Tam; Indice++)
        Escribe<<ObjArre.Datos[Indice] <<"  ";
   return  Escribe;
}
```

4.3.3 Eliminación en arreglos

Para eliminar un elemento de un arreglo, primero se debe localizar el elemento. Posteriormente, se deben recorrer todos los elementos que están a la derecha una posición hacia la izquierda. Es decir, el elemento no se puede quitar físicamente, sólo se ignora lógicamente. Además, debe reducirse el número actual de componentes. En esta operación se verifica, como posibles casos de error, que el arreglo esté vacío y que el valor a quitar no se encuentre en el arreglo.

A continuación se presenta la plantilla de un método que lleva a cabo la eliminación de un elemento del arreglo. Este método verifica que el elemento se encuentre almacenado en el arreglo, para ello hace uso de un método (`Busca`) que se analiza más adelante.

```
/* Método que elimina el elemento Valor del arreglo. Para ello utiliza un
➥método auxiliar, Busca, el cual da como resultado la posición en la cual
➥encontró el elemento. Si no lo encuentra da un número negativo. Si la
➥eliminación se lleva a cabo con éxito, se disminuye en uno a Tam. Este mé-
➥todo da como resultado uno de tres posibles valores: 1 si se pudo eliminar
➥Valor, 0 si el arreglo está vacío y -1 si Valor no está en el arreglo. */
```

```
template <class T>
int Arreglo<T>::EliminaDesordenado(T Valor)
{
   int Indice, Posic, Resultado= 1;
   /* Se verifica si el arreglo tiene al menos un elemento almacenado. */
   if (Tam > 0)
   {
       /* Método que busca el elemento Valor en el arreglo. Si lo encuen-
       ➥tra regresa su posición y si no un número negativo. */
       Posic= Busca(Valor);
       if (Posic < 0)
           Resultado= -1;
       else
       {
           /* Considerando que el resultado de la búsqueda fue exitoso,
           ➥el elemento se podrá eliminar del arreglo y por lo tanto el
           ➥tamaño de éste se reducirá en uno. */
           Tam--;
           /* Los elementos del arreglo se desplazan una posición hacia
           ➥la izquierda. */
           for (Indice= Posic; Indice < Tam; Indice++)
               Datos[Indice]= Datos[Indice+1];
       }
   }
   else
       /* El arreglo está vacío. */
       Resultado=  0;
   return  Resultado;
}
```

Verifica que el arreglo tenga elementos para dar mayor claridad al método. Este caso se contempla en la operación de búsqueda, ya que de estar vacío, el método regresa un valor negativo indicando que el elemento no fue encontrado. Sin embargo, evaluar explícitamente esta condición permite que el usuario conozca la razón por la cual la operación de eliminación fracasó: el arreglo está vacío o bien, el elemento a eliminar no está en el arreglo.

4.3.4 Operaciones en arreglos desordenados

Los arreglos desordenados son aquellos cuyos elementos no guardan ningún orden. Es decir, sus elementos no están ordenados creciente (*Datos[0]* ≤ *Datos[1]* ≤ *Datos[2]* ≤ ... ≤ *Datos[Tam-1]*) o decrecientemente (*Datos[0]* ≥ *Datos[1]* ≥ *Datos[2]* ≥ ... ≥ *Datos[Tam-1]*). Esta característica influye en las operaciones de búsqueda e inserción.

Búsqueda en arreglos

La operación de búsqueda se verá detalladamente en el capítulo 10, sin embargo, dado que se utiliza como auxiliar en la operación de inserción y en la de eliminación, resulta necesario explicarla brevemente en esta sección.

La búsqueda permite determinar si un cierto elemento fue almacenado o no en el arreglo. Existen diferentes formas para llevar a cabo esta operación, pero para el caso particular de los arreglos desordenados el único método aplicable es el que se conoce como ***búsqueda secuencial***.

El método de búsqueda secuencial consiste en recorrer el arreglo, elemento por elemento, comparando cada uno de ellos con el dato buscado. Si coinciden, la búsqueda termina con éxito. Si el arreglo se recorre totalmente y el elemento no se encuentra, entonces la búsqueda fracasa.

A continuación se presenta la plantilla de un método que lleva a cabo la búsqueda secuencial de un dato, en un arreglo desordenado.

```
/* Método que busca secuencialmente el elemento Valor en el arreglo. Re-
➥cibe como parámetro el dato buscado y, si lo encuentra, regresa la po-
➥sición del mismo. En caso contrario, regresa un número negativo. */

template <class T>
int Arreglo<T>::BuscaDesordenado (T Valor)
{
   /* La variable Resultado se inicializa con -1. En caso de encontrar
   ➥el dato buscado, se le asignará la posición donde se encontró. */
   int  Indice= 0, Resultado= -1;

   /* Se recorre el arreglo, elemento por elemento, comparando el conte-
   ➥nido de cada casilla con el valor buscado. */
   while ((Indice < Tam) && (Datos[Indice] != Valor))
        Indice++;

   /* Se verifica si se halló el elemento buscado. En caso afirmativo se
   ➥asigna como resultado el número de casilla en que se encontró. */
   if (Indice < Tam)
       Resultado= Indice;
   return  Resultado;
}
```

El orden de las condiciones del ciclo `while` es muy importante. Si el elemento buscado no estuviera en el arreglo, entonces `Indice` llegaría al valor `Tam`, y en ese caso `Datos[Indice]` provocaría un error de desbordamiento del arreglo (se estaría

intentando tener acceso a un elemento que no existe). Al evaluarse primero la condición (`Indice < Tam`) y resultar falsa, ya no se evalúa la segunda condición y por lo tanto ya no se produce el error arriba mencionado.

Inserción en arreglos

En el caso de los arreglos desordenados la inserción de un nuevo elemento al arreglo se hace en la primera casilla disponible, que generalmente será `Tam`, considerando que en ***C++*** los índices van de 0 a *Tam*–1.

La operación de inserción implica verificar que haya espacio en el arreglo, es decir que `Tam < MAX` y que el valor a insertar no se encuentre en el arreglo. Esta última condición puede omitirse dependiendo de la aplicación. Por ejemplo, si se considera el caso de un arreglo que almacena las calificaciones de un grupo de alumnos, entonces es válido tener valores repetidos.

A continuación se presenta la plantilla de un método que lleva a cabo la inserción de un nuevo elemento en el arreglo. Este método verifica que haya espacio en el arreglo y que el elemento no se repita.


```
/* Método que inserta el elemento Valor en el arreglo. Para ello usa un
➥método auxiliar, BuscaDesordenado(), que busca si el dato está en el
➥arreglo. Si lo encuentra, da su posición y si no da un número negativo.
➥Como resultado de la inserción se obtiene uno de tres posibles valores: 1
➥si se pudo insertar Valor, 0 si el arreglo está lleno y -1 si Valor ya
➥está en el arreglo. Si la inserción se lleva a cabo, se incrementa el
➥valor de Tam. */
template <class T>
int Arreglo<T>::InsertaDesordenado(T Valor)
{
   int  Posic, Resultado= 1;
   /* Se verifica que haya, al menos, un espacio disponible en el
   ➥arreglo. */
   if (Tam < MAX)
   {
       Posic= BuscaDesordenado(Valor);
       /* Si el elemento Valor no se encuentra en el arreglo, se inserta en
       ➥la posición Tam. Luego se incrementa Tam en una unidad. */
       if (Posic < 0)
          Datos[Tam++]= Valor;
       else
          Resultado= -1;
   }
   else
       Resultado= 0;
}
```

En el método presentado, considerando que el arreglo está desordenado, el nuevo elemento se inserta en la primera casilla disponible. Es decir, en la casilla número `Tam`. Después de asignar el valor a dicha casilla, se incrementa `Tam` en uno.

El programa 4.1 presenta la plantilla de la clase `Arreglo` con las operaciones asociadas, considerando que los elementos del arreglo están desordenados.

Programa 4.1

```
/* Se define la plantilla de la clase Arreglo con todos sus atributos y
➥métodos. Se asume que no existe orden entre los elementos del arreglo. */

/* Se define una constante que representa el número máximo de elementos
➥que puede almacenar el arreglo. */
#define MAX 100

template <class T>
class  Arreglo
{
   private:
       T Datos[MAX];
       int  Tam;
   public:
       Arreglo();
       void  Lectura();
       int  InsertaDesordenado(T);
       int  EliminaDesordenado(T);
       int  BuscaDesordenado(T);
       void  Escribe();
};

/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class  T>
Arreglo<T>::Arreglo()
{
   Tam= 0;
}

/* Método para la lectura de los atributos del arreglo. */
template <class  T>
void  Arreglo<T>::Lectura()
{
   int  Indice;
   /* Lectura del número de elementos a guardar en el arreglo. Se
   ➥verifica que el valor dado por el usuario sea  menor o igual que
   ➥el máximo permitido y mayor o igual que 1. */
```

```
    do {
        cout<<"\n\n Ingrese total de elementos:  ";
        cin>>Tam;
    } while (Tam < 1 || Tam > MAX);
    /* Lectura de valores para cada una de las Tam casillas del arreglo. */
    for (Indice= 0; Indice < Tam; Indice++)
    {
        cout<<"\nIngrese  el "<<Indice + 1<<" dato:  ";
        cin>>Datos[Indice];
    }
}

/* Método que inserta el elemento Valor en el arreglo. En esta imple-
➥mentación no se aceptan elementos repetidos. Se usa un método auxi-
➥liar, BuscaDesordenado(), el cual da como resultado la posición en la
➥cual encontró el elemento, o un número negativo en caso contrario. Si
➥la inserción se lleva a cabo, se incrementa a Tam. Este método da
➥como resultado uno de tres posibles valores: 1 si Valor se insertó en
➥el arreglo, 0 si el arreglo está lleno y -1 si Valor ya está en el
➥arreglo. */
template <class T>
int Arreglo<T>::InsertaDesordenado(T Valor)
{
    int  Posic, Resultado= 1;
    if (Tam < MAX)
    {
        Posic= BuscaDesordenado(Valor);
        if (Posic < 0)
             Datos[Tam++]= Valor;
        else
            Resultado= -1;
    }
    else
        Resultado= 0;
    return  Resultado;
}

/* Método que elimina el elemento Valor del arreglo. Para ello usa un
➥método auxiliar, BuscaDesordenado(), el cual busca a Valor en el
➥arreglo y regresa su posición, si lo encuentra. En caso contrario,
➥regresa un número negativo. Este método da como resultado uno de tres
➥posibles valores: 1 si Valor se elimina del arreglo, 0 si el arreglo
➥está vacío y -1 si Valor no está en el arreglo. Si la eliminación se
➥lleva a cabo, se decrementa a Tam. */
template <class T>
int Arreglo<T>::EliminaDesordenado(T Valor)
{
    int  Indice, Posic, Resultado= 1;
    if (Tam > 0)
```

```
    {
        Posic= BuscaDesordenado(Valor);
        if (Posic < 0)
            Resultado= -1;
        else
        {
            Tam--;
            for (Indice= Posic; Indice < Tam; Indice++)
                 Datos[Indice]= Datos[Indice+1];
        }
    }
    else
        Resultado= 0;
    return  Resultado;
}

/* Método que busca secuencialmente el elemento Valor en el arreglo.
➥Recibe como parámetro el dato buscado y da como resultado, si lo
➥encuentra, el número de casilla donde fue encontrado. En caso contrario
➥da un número negativo.*/
template <class T>
int Arreglo<T>::BuscaDesordenado(T Valor)
{
    int  Indice= 0, Resultado= -1;
    while ((Indice < Tam) && (Datos[Indice] != Valor))
         Indice++;
    if (Indice < Tam)
         Resultado= Indice;
    return  Resultado;
}

/* Método que despliega los valores almacenados en las casillas del
➥arreglo. */
template <class  T>
void  Arreglo<T>::Escribe()
{
    int  Indice;
    if (Tam > 0)
    {
        cout<<"\n\n";
        for (Indice= 0; Indice < Tam; Indice++)
             cout<< '\t' << Datos[Indice];
        cout<<"\n\n";
    }
    else
        cout<< "\n No hay elementos almacenados.";
}
```

El programa anterior definió una plantilla para la clase `Arreglo` para tener mayor generalidad al declarar objetos de este tipo.

El programa 4.2 presenta un ejemplo de aplicación de la plantilla previamente definida. Crea un objeto de tipo `Arreglo`, que almacena las claves de un grupo de alumnos. Por medio de los métodos se podrán leer, imprimir, registrar nuevas claves y eliminar algunas de las ya almacenadas. Éste es un caso de aplicación en el cual no se puede repetir información al ingresar un nuevo elemento en el arreglo.

Programa 4.2

```
/* Se incluye una biblioteca que contiene la plantilla de la clase
➥Arreglo, de esta manera se evita repetir código. En la biblioteca
➥"PlanArreglo.h" se tiene todo el código del programa 4.1.*/

#include "PlanArreglo.h"

/* Función que despliega en pantalla las opciones de trabajo que tiene
➥el usuario. */
int MenuOpciones()
{
   char Opcion;
   do {
      cout<<"\n\n\nL: Leer la lista de claves: ";
      cout<<"\nA: Dar de alta un nuevo alumno: ";
      cout<<"\nB: Dar de baja un alumno: ";
      cout<<"\nI: Imprimir la lista de claves: ";
      cout<<"\nF: Finalizar el proceso. ";
      cout<<"\n\n Ingrese opción de trabajo: ";
      cin>>Opcion;
   } while (Opcion != 'A' && Opcion != 'B' && Opcion != 'L' &&
   ➥Opcion != 'I' &&
            Opcion != 'F');
   return  Opcion;
}

/* Función principal desde la cual se tiene el control de todo el proceso:
➥se despliegan las opciones de trabajo y de acuerdo a la seleccionada
➥por el usuario se invoca el método que corresponda. */
void  main()
{
```

```
/* Se crea un objeto tipo Arreglo usando la plantilla de la biblioteca
➥PlanArreglo. Se indica que los elementos a almacenar en el arreglo
➥son de tipo entero. */
Arreglo<int> ClavAlum;
int  Clave, Res;
char Opc;
/* Este ciclo permite al usuario realizar más de una operación con
➥las claves de los alumnos. */
do {
    Opc= MenuOpciones();
    switch(Opc)
    {
       /* Se invoca el método de lectura del arreglo, para que el
       ➥usuario ingrese valores para cada uno de los atributos de la
       ➥clase. Para esta aplicación es el total de alumnos y la clave
       ➥de cada uno de ellos. */
       case 'L': {
                 ClavAlum.Lectura();
                 break;
                 }
       /* Se invoca el método de impresión del arreglo para desplegar
       ➥en pantalla la clave de cada uno de ellos. */
       case 'I': {
                 ClavAlum.Escribe();
                 break;
                 }
       /* Se invoca el método de inserción en arreglos desordenados.
       ➥Se debe dar como parámetro un dato del mismo tipo que el usado
       ➥para crear el objeto, en este caso es un número entero. */
       case 'A': {
                 cout<<"\n\n Clave del nuevo alumno:  ";
                 cin>>Clave;
                 Res= ClavAlum.InsertaDesordenado(Clave);
                 /* Se despliega un mensaje de acuerdo al resultado
                 ➥obtenido en el método. */
                 if (Res == 1)
                    cout<<"\n\n El nuevo alumno ya fue dado de alta. ";
                 else
                     if (Res == 0)
                         cout<<"\n\n No hay espacio para registrar el
                         ➥nuevo alumno.  ";
                     else
                         cout<<"\n\n Esa clave ya fue registrada
                                ➥previamente.  ";
                 break;
                 }
       /* Se invoca el método de eliminación en arreglos desordena-
       ➥dos. Se debe dar como parámetro un dato del mismo tipo que el
       ➥usado para crear el objeto, en este caso un número entero. */
```

```
            case 'B': {
                      cout<<"\n\n Clave del alumno a dar de baja:  ";
                      cin>>Clave;
                      Res= ClavAlum.EliminaDesordenado(Clave);
                      /* Se despliega un mensaje de acuerdo al resultado
                      ➥obtenido en el método. */
                      if (Res == 1)
                         cout<<"\n\n El alumno ya fue dado de baja. ";
                      else
                          if (Res == 0)
                              cout<<"\n\n No hay alumnos registrados.  ";
                          else
                              cout<<"\n\n Esa clave no está registrada.  ";
                      break;
                      }
            case 'F': cout<<"\n\n Termina el proceso.\n\n  ";
                      break;
          }

      } while (Opc != 'F');
}
```


4.3.5 Operaciones en arreglos ordenados

Un arreglo ordenado es aquel cuyos elementos tienen cierto orden entre sí, ya sea ascendente (`V[0]` ≤ `V[1]` ≤ … ≤ `V[Tam-1]`) o descendente (`V[0]` ≥ `V[1]` ≥ … ≥ `V[Tam-1]`). Por lo tanto, al operar con sus componentes será necesario conservar dicho orden. En el caso de la lectura, impresión y eliminación son válidos los métodos presentados para arreglos desordenados. Sin embargo, las operaciones de inserción y de búsqueda deben ser modificadas para adaptarse a este tipo de arreglos. En el caso de la inserción se deberá insertar el nuevo valor en una posición que no altere el orden existente entre los elementos. Mientras que la búsqueda resulta más eficiente al saber que el arreglo está ordenado.

Búsqueda en arreglos

La búsqueda secuencial en arreglos ordenados difiere muy poco de la búsqueda secuencial en arreglos desordenados. Dado que los elementos están ordenados, una de las condiciones del ciclo cambia para hacer más eficiente el proceso: cuando se recorre el arreglo, si se encuentra un valor más grande (en el caso de

orden ascendente) que el buscado, la búsqueda se interrumpe ya que a partir de ese elemento no será posible encontrarlo. A continuación se presenta la plantilla del método que realiza la búsqueda secuencial de un elemento en un arreglo ordenado ascendentemente. En el capítulo 10 se presentarán otros métodos de búsqueda.

```
/* Método que busca un elemento en un arreglo ordenado ascendentemente.
➥Recibe como parámetro un dato de tipo T (Valor). Da como resultado la
➥posición del mismo (si lo encuentra) o el negativo de la posición (+ 1)
➥en la que Valor debería estar. Note que el método regresa la posición
➥más uno para poder indicar con el negativo la posición en la que
➥debería estar si ésta fuera 0. */

template <class T>
int Arreglo<T>::BuscaOrdenado(T Valor)
{
   int  Indice=0, Resultado;

   /* La segunda condición del ciclo hace más eficiente el proceso de
   ➥búsqueda. */
   while ((Indice < Tam) && (Datos[Indice] < Valor))
         Indice++;

   /* Se verifica si se llegó al final del arreglo o bien, si se encontró
   ➥un valor mayor al buscado. En ambos casos se está en presencia de un
   ➥fracaso en la operación de búsqueda. */
   if (Indice == Tam || Datos[Indice] > Valor)
       Resultado= -(Indice + 1);
   else
       Resultado= Indice;
   return  Resultado;
}
```

Considerando que en la inserción en arreglos ordenados se necesita conocer la posición en la que se debe asignar el nuevo valor para no alterar el orden, el método presentado da como resultado (en caso de no encontrar el valor buscado) el negativo de la posición en la que debería estar (posición en la que se insertará). Se agrega el signo para diferenciar si el elemento está o no en el arreglo.

Para adaptar el método anterior a arreglos ordenados descendentemente, sólo se debe cambiar el operador relacional de la segunda condición: `Datos[Indice] > Valor`. Lo mismo en la instrucción `if`, saliendo del ciclo.

Inserción en arreglos

Al insertar un nuevo elemento en un arreglo ordenado se debe cuidar que el orden no se altere. Por lo tanto, se debe buscar la posición en la que se asignará, de tal manera que se mantenga el orden existente: creciente o decreciente.

La operación de inserción requiere verificar que haya espacio en el arreglo y que el elemento a insertar no haya sido previamente almacenado. Esta última condición depende de cada aplicación, ya que ciertos casos justifican tener elementos repetidos. Por ejemplo, si lo que se almacena son las edades de un grupo de niños, puede haber más de una ocurrencia de una misma edad.

Una vez que se han considerado los casos mencionados y ubicado el lugar en que se debe insertar el nuevo valor, se procede a recorrer una posición a la derecha a todos los elementos que se encuentren a partir de esa posición. El desplazamiento se hace desde la casilla *Tam*-1 hasta la casilla correspondiente a la posición en la cual se hará la asignación del nuevo valor. Finalmente, si la inserción se concluyó con éxito, se debe incrementar el tamaño del arreglo.

A continuación se presenta la plantilla del método que realiza la inserción de un elemento en un arreglo ordenado de manera creciente.

```
/* Método que inserta un elemento en un arreglo ordenado crecientemente,
➥sin alterar su orden. Se recibe como parámetro un dato de tipo T (Valor).
➥Este método da como resultado uno de tres posibles valores: 1 si Valor
➥se insertó en el arreglo, 0 si el arreglo está lleno y -1 si Valor ya
➥está almacenado en el arreglo. Si la inserción concluye con éxito, se
➥incrementa a Tam en una unidad. */

template <class T>
int Arreglo<T>::InsertaOrdenado(T Valor)
{
   int  Indice, Posic, Resultado= 1;

   /* Verifica si hay, al menos, un espacio disponible en el arreglo. */
   if (Tam < MAX)
   {
       /* Se invoca al método que busca un elemento, Valor, en el arreglo. */
       Posic= BuscaOrdenado(Valor);
       if (Posic > 0)
           Resultado= -1;
       else
```

```
        {
            /* Convierte la posición en positiva y le resta 1. */
            Posic= (Posic * -1) -1;
            /* Se recorre el contenido de cada casilla una posición hacia
            ➥la derecha, a partir de la posición Tam-1 hasta la posición
            ➥en que se asignará el nuevo valor. */
            for (Indice= Tam;  Indice > Posic;  Indice--)
                Datos[Indice]= Datos[Indice - 1];
            Datos[Posic]= Valor;
            /* Se incrementa el valor de Tam en uno, ya que hay un nuevo
            ➥valor en el arreglo. */
            Tam++;
        }
    }
    else
        Resultado= 0;
    return  Resultado;
}
```

El programa 4.3 presenta la plantilla de la clase `Arreglo` con las operaciones asociadas, considerando que los elementos del arreglo están ordenados de manera creciente.

Programa 4.3

```
/* Se define una constante para almacenar el número máximo de elementos
➥que puede guardar el arreglo. */
#define MAX 100

/* Se define la plantilla de la clase Arreglo con todos sus atributos y
➥métodos. Se asume que los elementos del arreglo están ordenados ascenden-
➥temente. Los atributos corresponden a los explicados en la sección 4.2. */
template <class T>
class  Arreglo
{
   private:
       T Datos[MAX];
       int Tam;
   public:
       Arreglo();
       void  Lectura();
       int  InsertaOrdenado(T);
       int  EliminaOrdenado(T);
       void  Escribe();
       int  BuscaOrdenado(T);
};
```

```
/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class T>
Arreglo<T>::Arreglo()
{
   Tam= 0;
}

/* Método que permite leer el número de elementos que se van a almacenar
➥y el valor de cada uno de ellos. Verifica que el total de elementos sea
➥al menos 1 y que no supere el máximo especificado. */
template <class  T>
void  Arreglo<T>::Lectura()
{
   int  Indice;
   do {
       cout<<"\n\n Ingrese número de datos a guardar:  ";
       cin>> Tam;
   } while (Tam < 1 || Tam > MAX);

   for (Indice= 0; Indice < Tam; Indice++)
   {
       cout<<"\nIngrese el "<<Indice+1<<"  dato:  ";
       cin>> Datos[Indice];
   }
}

/* Método que inserta un elemento en un arreglo ordenado crecientemente,
➥sin alterar su orden. Recibe como parámetro un dato de tipo T (Valor).
➥Da como resultado uno de tres posibles valores: 1 si Valor se inserta,
➥0 si el arreglo está lleno y -1 si Valor ya está en el arreglo. Si la
➥inserción concluye con éxito se incrementa a Tam en uno.*/
template <class T>
int Arreglo<T>::InsertaOrdenado(T Valor)
{
   int  Indice, Posic, Resultado= 1;
   if (Tam < MAX)
   {
       Posic= BuscaOrdenado(Valor);
       if (Posic > 0)
           Resultado= -1;
       else
       {
           Posic= (Posic * -1) - 1;
           for (Indice= Tam; Indice > Posic; Indice--)
                Datos[Indice]= Datos[Indice - 1];
           Datos[Posic]= Valor;
           Tam++;
       }
     }
```

```
     else
         Resultado= 0;
     return  Resultado;
}

/* Método que elimina un elemento de un arreglo. Recibe como
➥parámetro, Valor, un dato de tipo T. Da como resultado uno de tres
➥posibles valores: 1 si Valor se eliminó, 0 si el arreglo está vacío
➥y -1 si Valor no está en el arreglo. En caso de éxito, disminuye a Tam
➥en uno. */
template <class T>
int Arreglo<T>::EliminaOrdenado(T Valor)
{
   int  Posic, Indice, Resultado= 1;
   if (Tam > 0)
   {
        Posic= BuscaOrdenado(Valor);
        if (Posic < 0)
             Resultado= -1;
        else
        {
             Tam--;
             for (Indice= Posic; Indice < Tam; Indice++)
                  Datos[Indice]= Datos[Indice+1];
        }
   }
   else
       Resultado= 0;
   return  Resultado;
}

/* Método que despliega los valores almacenados en el arreglo. */
template <class T>
void  Arreglo<T>::Escribe()
{
   int  Indice;
   if (Tam > 0)
   {
        cout<<"\n Impresión de datos\n";
        for (Indice= 0; Indice < Tam; Indice++)
             cout<< '\t' << Datos[Indice];
   }
   else
        cout<< "\nNo hay elementos registrados.";
}
```

```
/* Método que busca un elemento en un arreglo ordenado ascendentemente.
➥Recibe como parámetro un dato de tipo T (Valor). Si lo encuentra,
➥regresa la posición del mismo. En caso contrario, regresa el negativo
➥de la posición (+1) en la que debería estar. Note que el método regresa
➥la posición más uno para poder indicar con el negativo la posición en la
➥que debería estar si ésta fuera 0. */
template <class T>
int Arreglo<T>::BuscaOrdenado(T Valor)
{
   int  Indice= 0, Resultado;
   while ((Indice < Tam) && (Datos[Indice] < Valor))
      Indice++;
   if (Indice == Tam || Datos[Indice] > Valor)
        Resultado= -(Indice + 1);
   else
        Resultado= Indice;
   return  Resultado;
}
```


En la clase presentada en el programa 4.3 se incluyeron algunos métodos necesarios para trabajar con un arreglo ordenado crecientemente. Estas operaciones pueden completarse posteriormente con las operaciones de ordenación y búsqueda, mismas que se verán en los capítulos 9 y 10.

El programa 4.4 presenta un ejemplo de aplicación de la plantilla previamente definida. Crea un objeto tipo arreglo de números reales para almacenar los *tiempos* hechos por un nadador durante su entrenamiento, ordenados de manera creciente. Utiliza los métodos definidos para leer los datos, imprimirlos, dar de alta nuevos tiempos y eliminar algunos de los registrados.

Programa 4.4

```
/* Se incluye la biblioteca "PlantArreOrd.h" en la cual está la
➥plantilla de la clase Arreglo presentada en el programa 4.3, con el
➥objeto de no repetir código. */
#include "PlantArreOrd.h"

/* Función que despliega en pantalla las opciones de trabajo relacionadas
➥a la aplicación. Regresa un dato tipo carácter que representa la
➥opción seleccionada por el usuario. */
char MenuOpciones()
```

```
{
   char Opc;
   do {
      cout<<"\n\nL: Leer los tiempos hechos por el nadador. \n";
      cout<<"\nI: Imprimir un listado con los tiempos del nadador. \n";
      cout<<"\nA: Dar de alta un nuevo tiempo. \n";
      cout<<"\nB: Dar de baja un tiempo ya registrado. \n";
      cout<<"\nT: Terminar el proceso. \n";
      cin>>Opc;
   } while (Opc != 'L' && Opc != 'I' && Opc != 'A' && Opc != 'B' && Opc !=
'T');
   return  Opc;
}


/* Función principal en la cual se tiene el control de toda la
➥aplicación: se crea un objeto y otras variables de trabajo, se muestran
➥las posibles operaciones a realizar y de acuerdo a la opción elegida
➥por el usuario se invocan los métodos que correspondan. */
void main()
{
   /* Se crea un objeto arreglo para almacenar números reales. */
   Arreglo <float> TiemposNada;
   char Opc;
   float Tiempo;
   int  Res;

   do {
      Opc= MenuOpciones();
      switch (Opc)
      {
         /* Se invoca el método que ingresa, del teclado, valores para
         ➥los atributos del arreglo. En este caso el total de tiempos
         ➥registrados y cada uno de los mismos. */
         case 'L': {
                     TiemposNada.Lectura();
                     break;
                   }
         /* Se invoca el método que despliega en pantalla los valores
         ➥almacenados en el objeto arreglo, en este caso los tiempos
         ➥registrados por el nadador durante su entrenamiento. */
         case 'I': {
                     TiemposNada.Escribe();
                     break;
                   }
```

```
        /* Se invoca el método que inserta un nuevo elemento en el
        ➥arreglo ordenado, en este caso es un nuevo tiempo del nadador.
        ➥Luego de ejecutar el método se analiza el resultado obtenido y
        ➥se despliega un mensaje adecuado. */
        case 'A': {
                   cout<<"\n\n Nuevo tiempo registrado por el nadador:  ";
                   cin>>Tiempo;
                   Res= TiemposNada.InsertaOrdenado(Tiempo);
                   if (Res == 1)
                       cout<<"\n\n El nuevo tiempo ya fue dado
                       ➥de alta. ";
                   else
                       if (Res == 0)
                           cout<<"\n\n No hay espacio para registrar el
                           ➥nuevo tiempo. ";
                       else
                           cout<<"\n\n Ese tiempo ya fue registrado.  ";
                   break;
                   }
        /* Se invoca el método que elimina un elemento del arreglo, en
        ➥este caso un tiempo que ya no interesa conservar. Luego de
        ➥ejecutar el método se analiza el resultado obtenido y se des-
        ➥pliega el mensaje adecuado. */
        case 'B': {
                   cout<<"\n\n Tiempo a dar de baja:  ";
                   cin>>Tiempo;
                   Res= TiemposNada.EliminaOrdenado(Tiempo);
                   if (Res == 1)
                       cout<<"\n\n El tiempo ya fue dado de baja. ";
                   else
                       if (Res == 0)
                           cout<<"\n\n No hay tiempos registrados.  ";
                       else
                          cout<<"\n\n Ese tiempo no está registrado.  ";
                   break;
                   }
        case 'T': cout<<"\n\n Termina el proceso.\n\n  ";
                  break;
      }
   } while (Opc != 'T');
}
```

4.4 Arreglos paralelos

Se dice que dos arreglos son ***arreglos paralelos*** cuando la casilla 0 del primero está relacionada con la casilla 0 del segundo, la casilla 1 del primero con la casilla 1 del segundo y así sucesivamente. La figura 4.5 representa, por medio de las flechas, esta relación; misma que puede darse entre dos o más arreglos.

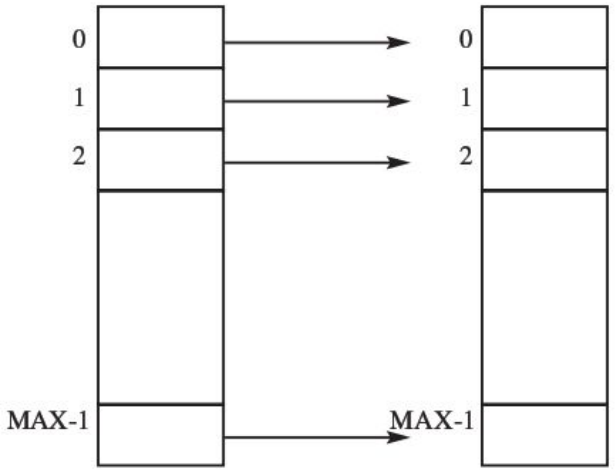


FIGURA 4.5 *Arreglos paralelos*

Por ejemplo, considere que se conocen la clave y la calificación de un grupo de alumnos. Se requiere un arreglo para guardar cada una de las claves y otro para las calificaciones. Sin embargo, para que se pueda conservar la relación entre la clave y la calificación correspondiente, es necesario usar arreglos paralelos. De esta forma, la primera casilla del arreglo de claves y la primera casilla del arreglo de calificaciones hacen referencia al mismo alumno.

Cualquier cambio en el orden de los elementos de uno de los arreglos debe afectar el orden de los otros. Por lo tanto, ciertas operaciones deben adaptarse para que no se pierda la correspondencia entre la información de las celdas de los arreglos involucrados. Las operaciones de lectura y escritura no cambian, se leerán de manera independiente cada uno de los arreglos. Sólo sería necesario modificar los métodos respectivos si se quisieran leer o escribir simultáneamente los mismos. Por su parte, la operación de búsqueda normalmente se hace sólo sobre

uno de los arreglos; y conocer la posición del valor buscado implica también conocer la posición de los elementos correspondientes en los otros arreglos. En cambio, las operaciones de inserción y eliminación deben modificarse para responder a la característica de este tipo de estructura.

Inserción en arreglos paralelos

Para insertar un nuevo elemento en los arreglos, primero se debe considerar si en alguno existe un orden (generalmente sólo uno de ellos está ordenado). Si es así, se debe buscar la posición en la cual insertar el nuevo valor, recorrer los elementos una posición a la derecha, asignar el nuevo valor e incrementar el tamaño. En los demás arreglos no se requiere realizar la búsqueda, el elemento debe insertarse en la misma posición hallada para el valor agregado en el arreglo ordenado.

Retomando el ejemplo de las claves y calificaciones, si las primeras estuvieran ordenadas, sólo se debería realizar la búsqueda en este arreglo y la posición encontrada sería la misma para insertar tanto la nueva clave como su calificación correspondiente.

Si ninguno de los arreglos estuviera ordenado, entonces el nuevo dato se inserta en la primera casilla disponible y esto es aplicable a todos los arreglos. En este caso, la operación de inserción es igual a la que se estudió en la sección de los arreglos desordenados.

Eliminación en arreglos paralelos

Para eliminar un elemento de los arreglos, primero se debe buscar el dato dado como referencia en el arreglo correspondiente. Si se encuentra, se quita (recorriendo todos los valores que están a su derecha una posición hacia la izquierda) y de la misma manera se eliminan también los elementos que ocupan su misma posición en los otros arreglos. Normalmente se da sólo uno de los datos como referencia y al encontrarlo, los demás se eliminan usando la posición de éste.

Retomando el ejemplo de las claves y las calificaciones, si un alumno se diera de baja, se buscaría su clave y si se encontrara se quitaría este dato y la calificación asociada ocupando la posición de la primera.

Tanto en la inserción como en la eliminación la operación de búsqueda no se aplica a todos los arreglos, por lo que la plantilla de la clase `Arreglo` previamente definida no se modifica en su totalidad. Para tener una solución más general e independiente se define una nueva plantilla de clase para arreglos paralelos, con los métodos modificados.

Programa 4.5

```
/* Se define una constante para almacenar el número máximo de elementos
➥que puede guardar el arreglo. */
#define MAX 100

/* Se define la plantilla de la clase ArreParal con todos sus atributos
➥y métodos. Se incluyen diferentes versiones de algunos de los métodos
➥de tal manera que la plantilla sirva tanto para arreglos ordenados como
➥para arreglos desordenados. */
template <class T>
class  ArreParal
{
   private:
       T Datos[MAX];
       int  Tam;
   public:
       ArreParal();
       int     InsertaOrdenado(int , T);
       int     InsertaDesordenado(T);
       void  Elimina(int);
       int     BuscaOrdenado(T);
       int     BuscaDesordenado(T);
       T RegresaValor(int);
       int     RegresaTamano();
       friend  istream &operator>>(istream &, ArreParal<T> &);
       friend  ostream &operator<<(ostream &, ArreParal<T> &);
};

/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class  T>
ArreParal<T>::ArreParal()
{
   Tam= 0;
}
```

```
/* Método que inserta un elemento en un arreglo ordenado crecientemente,
➥sin alterar su orden. Recibe como parámetros: Posic, un entero que
➥indica la posición en la que debe insertarse el nuevo elemento si hay
➥espacio y el elemento a insertar que es un dato de tipo T (Valor). Da
➥como resultado uno de dos posibles valores: 1 si Valor se inserta o 0
➥si el arreglo está lleno. Si la inserción concluye con éxito se incremen-
➥ta a Tam en uno.*/
template <class  T>
int  ArreParal<T>::InsertaOrdenado(int Posic, T Valor)
{
   int  Indice,  Resultado= 1;
   if (Tam < MAX)
   {
       for (Indice= Tam; Indice > Posic; Indice--)
             Datos[Indice]= Datos[Indice - 1];
       Datos[Posic]= Valor;
       Tam++;
   }
   else
       Resultado= 0;
   return   Resultado;
}

/* Método que inserta un elemento en un arreglo desordenado. Recibe como
➥parámetro el elemento a insertar, que es un dato de tipo T (Valor). Da
➥como resultado uno de dos posibles valores: 1 si Valor se inserta o 0
➥si el arreglo está lleno. Si la inserción concluye con éxito se incre-
➥menta a Tam en uno.*/
template <class  T>
int  ArreParal<T>::InsertaDesordenado(T Valor)
{
   int   Indice,  Resultado= 1;
   if  (Tam < MAX)
   {
        Datos[Tam]= Valor;
        Tam++;
  }
   else
        Resultado= 0;
   return   Resultado;
}

/* Método que elimina un elemento de un arreglo. Recibe como parámetro
➥un número entero, Posic, que indica la posición del dato a eliminar.
➥Este método se invoca sólo si antes se ejecutó con éxito el método de
➥búsqueda. Se disminuye el valor de Tam en uno. */
```

```
template <class  T>
void  ArreParal<T>::Elimina(int  Posic)
{
  int  Indice;
   Tam--;
   for (Indice= Posic; Indice < Tam; Indice++)
        Datos[Indice]= Datos[Indice+1];
}

/* Método que busca un elemento en un arreglo ordenado ascendentemente.
➥Recibe como parámetro un dato de tipo T (Valor). Si lo encuentra,
➥regresa la posición del mismo. En caso contrario, regresa el negativo
➥de la posición (+1) en la que debería estar. */
template <class  T>
int  ArreParal<T>::BuscaOrdenado(T Valor)
{
   int   Indice= 0, Resultado;
   while ((Indice < Tam) && (Datos[Indice] < Valor))
       Indice++;
   if  (Indice == Tam || Datos[Indice] > Valor)
       Resultado= -(Indice + 1);
   else
       Resultado= Indice;
   return   Resultado;
}

/* Método que busca un elemento en un arreglo desordenado. Recibe como
➥parámetro un dato de tipo T (Valor). Si lo encuentra, regresa la
➥posición del mismo. En caso contrario, regresa un número negativo. */
template <class  T>
int  ArreParal<T>::BuscaDesordenado(T Valor)
{
   int   Indice= 0, Resultado= -1;
   while ((Indice < Tam) &&  (Datos[Indice] != Valor))
       Indice++;
   if  (Indice < Tam)
       Resultado= Indice;
   return   Resultado;
}

/* Método que permite, a usuarios externos a la clase, conocer el
➥contenido de una casilla del arreglo. Recibe como parámetro un entero,
➥Indice, que indica el número de celda de la cual se dará su contenido.
➥El resultado es un valor de tipo T. */
```

```
template <class  T>
T ArreParal<T>::RegresaValor(int Indice)
{
   return   Datos[Indice];
}

/* Método que regresa el total de elementos del arreglo. */
template <class  T>
int  ArreParal<T>::RegresaTamano()
{
   return   Tam;
}

/* Definición de la sobrecarga del operador >>. Por medio de este
➥operador sobrecargado y declarado como amigo de la clase ArreParal se
➥podrá leer de manera directa a todos los miembros de la misma. */
template <class  T>
istream &operator>> (istream &Lee,  ArreParal<T> &ObjArre)
{
   int   Indice;
   do {
      Lee>>ObjArre.Tam;
   } while (ObjArre.Tam < 1 || ObjArre.Tam  > MAX);
   for (Indice= 0; Indice < ObjArre.Tam; Indice++)
        Lee>>ObjArre.Datos[Indice];
   return   Lee;
}

/* Definición de la sobrecarga del operador <<. Por medio de este
➥operador sobrecargado y declarado como amigo de la clase ArreParal se
➥podrá desplegar de manera directa a todos los miembros de la misma. */
template <class T>
ostream &operator<< (ostream &Escribe,  ArreParal<T> &ObjArre)
{
   int   Indice;
   if  (ObjArre.Tam  >  0)
        for (Indice= 0; Indice < ObjArre.Tam; Indice++)
            Escribe<<ObjArre.Datos[Indice] <<"  ";
   else
        cout<< "\nNo hay elementos registrados.";
   return   Escribe;
}
```

La figura 4.6 muestra gráficamente un ejemplo de arreglos paralelos, que almacenan la clave y la calificación de un grupo de alumnos. En el esquema presentado

se observa que el alumno con clave 2500 obtuvo una calificación de 9.5, el de clave 3000, una de 7.3 y el de clave 4050, una de 8.6.

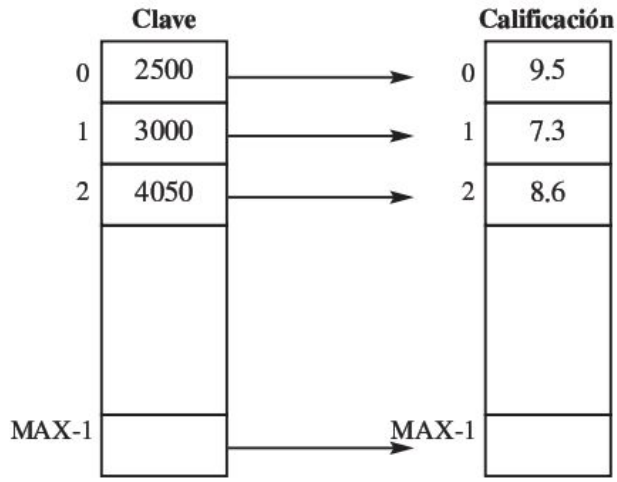


FIGURA 4.6 *Ejemplo de arreglos paralelos*

A continuación se presenta un programa de aplicación que utiliza la plantilla del programa 4.5 para crear objetos que puedan guardar las claves y calificaciones de los alumnos. Posteriormente, a través de los métodos, se darán de alta nuevos alumnos, se eliminarán los ya registrados y se obtendrán algunos reportes. En este ejemplo se hace uso de los métodos para arreglos ordenados, ya que se asume que los datos están ordenados ascendentemente según las claves de los alumnos.

Programa 4.6

```
/* Ejemplo de aplicación de arreglos paralelos. En la biblioteca
➥"PlantArreParal.h" se incluye la plantilla de la clase definida
➥en el programa 4.5. */
#include "PlantArreParal.h"


/* Función que despliega al usuario las posibles opciones de trabajo. */
int MenuOpciones()
{
   int  Opc;
   do {
       cout<<"\n\n1-Captura inicial de claves y calificaciones
       ➥de alumnos.  ";
       cout<<"\n2-Ingresar un nuevo alumno y su calificación.  ";
```

```
        cout<<"\n3-Eliminar un alumno y su calificación.  ";
        cout<<"\n4-Obtener un listado de las claves de los alumnos.  ";
        cout<<"\n5-Obtener un listado de claves y calificaciones
        ➥de todos los alumnos. ";
        cout<<"\n6-Obtener la calificación de un alumno. ";
        cout<<"\n7-Terminar el proceso. ";
        cout<<"\n\nIngrese la opción seleccionada. ";
        cin>>Opc;
    } while (Opc < 1 || Opc > 7);
    return  Opc;
}

    /* Función principal: se despliega el menú de opciones y, de
    ➥acuerdo a la opción elegida por el usuario, se invoca el método
    ➥correspondiente. */
void  main ()
{
    ArreParal<int> Claves;
    ArreParal<float> Califc;
    int  ClaAlum, Opc, Posic, Indice, TotalAl;
    float CalAlum;
    do {
        Opc= MenuOpciones();
        switch (Opc)
        {
         /* Se leen los datos (clave y calificación) de cada uno de los
         ➥alumnos del grupo. Por medio del operador sobrecargado >> se
         ➥indica la lectura de los objetos Claves y Califc. */
         case 1: {
                 cout<<"\n\nDé el número de claves y cada una de
                 ➥las claves\n";
                 cin>>Claves;
                 cout<<"\n\nDé el número de calificaciones y cada
                 ➥una de ellas \n";
                 cin>>Califc;
                 break ;
                 }
        /* Se registra un nuevo alumno, proporcionando para ello su
        ➥clave y su calificación. Las claves son únicas y están
        ➥ordenadas de manera ascendente. Primero se verifica, por medio
        ➥del método BuscaOrdenado(), que la clave dada no haya sido
        ➥previamente almacenada. Si no se repite, entonces se agrega a
        ➥la colección de claves sin alterar el orden de éstas. Para ello
        ➥se usa el método InsertaOrdenado(). Si la inserción se lleva a
        ➥cabo con éxito, entonces se procede a agregar la calificación
        ➥del nuevo alumno en la posición que le corresponde por el valor
        ➥de su clave. */
```

```
case 2: {
        cout<<"\n\nDé la clave y calificación del nuevo alumno: ";
        cin>>ClaAlum;
        cin>>CalAlum;
        Posic= Claves.BuscaOrdenado(ClaAlum);
        if (Posic > 0)
          cout<<"\n\nEsa clave ya fue registrada previamente. \n";
        else
        {
            Posic= (Posic * -1) -1;
            if (Claves.InsertaOrdenado(Posic, ClaAlum) == 1)
                Calific.InsertaOrdenado(Posic, CalAlum);
            else
                cout<<"\n\nYa no se pueden registrar nuevos
                ➥alumnos. \n";
        }
        break ;
        }
/* Se elimina un alumno dando su clave como dato de entrada. Si
➥la clave está (existe un alumno con dicha clave) se procede a
➥eliminarla y a eliminar su correspondiente calificación. */
case 3: {
        cout<<"\n\nDé la clave del alumno que desea dar de baja: ";
        cin>>ClaAlum;
        Posic= Claves.BuscaOrdenado(ClaAlum);
        if (Posic > 0)
        {
            Claves.Elimina(Posic);
            Calific.Elimina(Posic);
        }
        else
            cout<<"\n\nEsa clave no está registrada. \n";
        break ;
        }
/* Se genera un reporte con todas las claves de los alumnos
➥registrados. Por medio del operador sobrecargado << se indica
➥la escritura del objeto Claves de manera directa. */
case 4: {
        cout<<"\n\nListado de claves de alumnos registrados. \n";
        cout<<Claves;
        cout<<"\n\n";
        break;
        }
/* Se genera un reporte con la clave y la calificación de todos
➥los alumnos registrados. Primero se obtiene el total de alumnos
➥por medio del método que regresa el tamaño del arreglo. Luego
➥se tiene acceso a cada uno de los valores almacenados, por
➥medio del método RegresaValor(), y se los imprime. */
```

```
        case 5: {
                TotalAl= Claves.RegresaTamano();
                cout<<"\n\nClave                Calificación \n";
                for (Indice= 0; Indice < TotalAl; Indice ++)
                {
                     cout<<Claves.RegresaValor(Indice)<<"         ";
                     cout<<Calific.RegresaValor(Indice) <<"\n";
                }
                break ;
                }
        /* Dada la clave de un alumno, se imprime la calificación del
        ➥mismo. Se hace uso del método BuscaOrdenado(), para encontrar
        ➥la clave. Si se encuentra (ese alumno está registrado), se
        ➥invoca al método RegresaValor() para tener acceso a su
        ➥calificación. */
        case 6: {
                cout<<"\n\nClave del alumno que desea conocer su
                ➥calificación: ";
                cin>>ClaAlum;
                Posic= Claves.BuscaOrdenado(ClaAlum);
                if (Posic > 0)
                {
                     cout<<"\n\nCalificación del alumno con clave:
                     ➥"<<ClaAlum;
                     cout<<" es:  "<<Calific.RegresaValor(Posic);
                }
                else
                     cout<<"\n\nEsa clave no está registrada. \n\n";
                break ;
                }
        /* Termina el ciclo de procesamiento. */
        case 7: {
                cout<<"\n\nTermina el procesamiento de los datos. \n\n";
                break;
                }
        }
   } while (Opc != 7);
}
```

4.5 Arreglos de dos dimensiones

Los arreglos vistos permiten agrupar información relacionada sólo por un tema. Por ejemplo, se pueden usar para almacenar las calificaciones de un grupo de N alumnos obtenidas en un examen, o bien, para las calificaciones de un alumno obtenidas en varios exámenes.

Sin embargo, en ciertos casos se requiere almacenar datos que representen diferentes dimensiones de la información. Retomando el ejemplo mencionado, puede ser importante agrupar las calificaciones obtenidas por N alumnos en M exámenes. En este caso, la información se relaciona por alumno y por examen. Es decir, se reconocen dos dimensiones en la información: una para los alumnos y otra para los exámenes.

Para representar información con estas características es necesaria una estructura de datos que permita manejar dos dimensiones. Esta estructura recibe el nombre de ***arreglo bidimensional***, ***arreglo de dos dimensiones*** o ***matriz***. La representación gráfica de un arreglo bidimensional se puede observar en la figura 4.7.

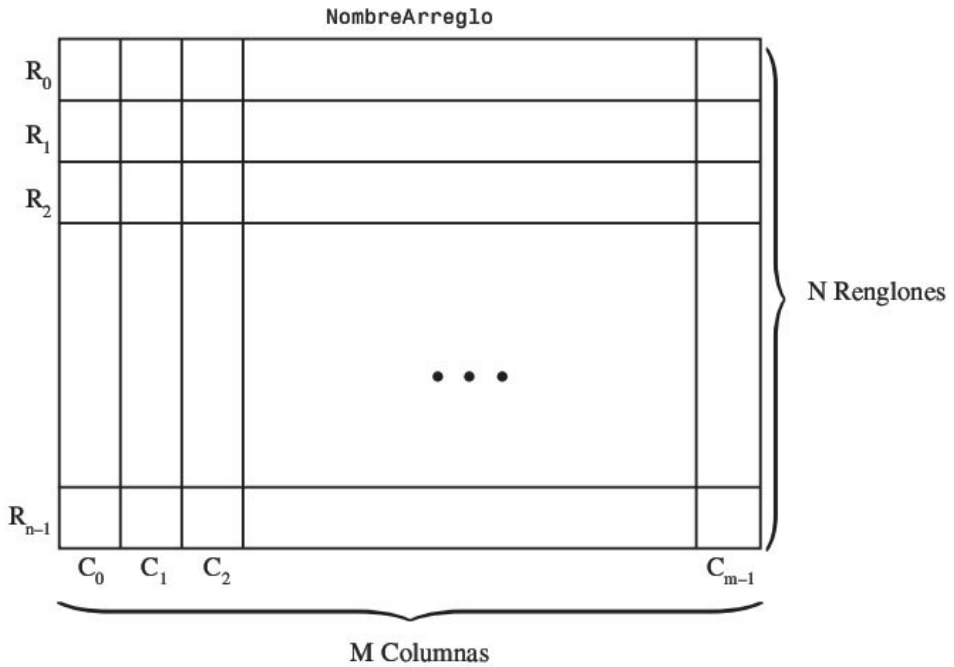


FIGURA 4.7 *Representación gráfica de un arreglo bidimensional de N × M elementos*

Como puede ver en la figura, cada elemento del arreglo se identifica por medio de dos índices: uno que hace referencia al renglón y otro a la columna. En el caso del lenguaje ***C++***, el primer índice siempre hace referencia al renglón mientras que el segundo indica columna.

El nombre del arreglo hace referencia a toda la estructura de datos. Cada elemento o casilla se especificará siguiendo la notación: *NombreArreglo[i][j]*, donde *i* indica el número del renglón y *j* el número de la columna donde está el elemento.

El almacenamiento y recuperación de información en un arreglo bidimensional podrá hacerse por renglones o por columnas. Si es por renglones, se completa un renglón (para ello se recorrerán todas las columnas de dicho renglón) para luego pasar al siguiente renglón y así sucesivamente hasta visitar los N renglones. Si el acceso se hace por columnas, entonces se completa una columna (para ello se recorrerán todos los renglones de dicha columna) para luego pasar a la siguiente hasta visitar las M columnas.

La clase que representa una estructura de datos tipo arreglo bidimensional tendrá como atributos la colección de elementos que se almacenarán, así como el número de renglones y columnas ocupadas por dichos datos. El número de renglones y columnas estará acotado por un número máximo que se define inicialmente. Además, la clase incluirá un conjunto de operaciones o métodos que permiten manipular los miembros de la misma. La figura 4.8 presenta a la clase `ArregloBidimensional`.

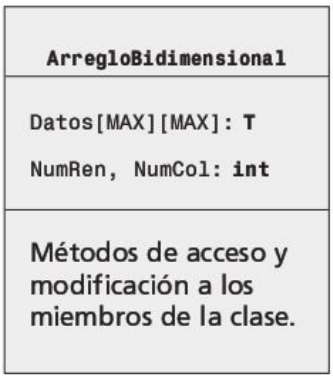


FIGURA 4.8 *Clase* `ArregloBidimensional`

A continuación se presenta la plantilla de la clase `ArregloBidimensional` que incluye dos operaciones básicas para estas estructuras de datos (lectura y escritura) y algunos métodos que se consideran útiles para el manejo de la información almacenada en el arreglo. Los mismos se explican en el comentario que acompaña la codificación de cada uno de ellos.

Programa 4.7

```
/* Plantilla de la clase ArregloBidimensional. Se definen algunos méto-
➥dos útiles para el manejo del contenido de un arreglo de dos dimensio-
➥nes. Los atributos son la colección de datos, Datos[MAX] [MAX], en la
➥cual se establece un número máximo de renglones y de columnas y el número
➥de renglones y de columnas que están ocupadas, NumRen y NumCol. */

template <class  T>
class  ArregloBidimensional
{
   private:
       T Datos[MAX] [MAX];
       int  NumRen, NumCol;
   public:
       ArregloBidimensional(int, int);
       void  Lectura();
       void  Escritura();
       T SumaRenglon(int);
       T SumaColumna(int);
       T MaximoColumna(int) ;
       T MaximoRenglon(int);
       T MinimoColumna(int);
       T MinimoRenglon(int);
       T RegresaDato(int, int );
};

/* Declaración del método constructor por omisión. Inicializa el
➥número actual de renglones y de columnas en 0. */
template <class  T>
ArregloBidimensional<T>::ArregloBidimensional()
{
   NumRen= 0;
   NumCol= 0;
}

/* Declaración del método constructor con parámetros. */
template <class  T>
ArregloBidimensional<T>::ArregloBidimensional(int NR, int  NC)
{
   NumRen= NR;
   NumCol= NC;
}
```

```
/* Método de lectura. Los datos leídos del teclado se almacenan por renglo-
➥nes. Observe que el ciclo externo es el de los renglones (primer índice).
➥Por lo tanto, para cada valor del mismo se recorren todas las columnas
➥(ciclo interno). Para darle mayor información al usuario, se supone que la
➥lectura y validación del total de elementos se hace en el programa de
➥aplicación y desde ahí también se invoca el constructor con parámetros
➥para asignarle valores a los atributos NumRen y NumCol.*/
template <class T>
void  ArregloBidimensional<T>::Lectura()
{
   int  Ren, Col;

   for (Ren= 0; Ren < NumRen; Ren++)
        for (Col= 0; Col < NumCol; Col++)
        {
            cout<<"\nIngrese dato:  ";
            cin>>Datos[Ren][Col];
        }
}

/* Método de escritura. Los datos almacenados se despliegan en la pantalla
➥por renglones. Con respecto al orden de los índices aplica el mismo
➥comentario que en el método de lectura. */
template <class T>
void  ArregloBidimensional<T>::Escritura()
{
   int  Ren, Col;
   for (Ren= 0; Ren < NumRen; Ren++)
   {
        for (Col= 0; Col < NumCol; Col++)
            cout<< Datos[Ren][Col]<<"  ";
        cout<<endl;
   }
}

/* Método que suma todos los elementos de un renglón. Para ello se deben
➥recorrer todas las columnas de dicho renglón. El número de renglón a
➥sumar se indica a través del parámetro. Si el tipo T usado para crear
➥el objeto ArregloBidimensional no fuera un número, entonces se debería
➥sobrecargar el operador + . */
template <class T>
T ArregloBidimensional<T>::SumaRenglon(int Ren)
{
   T Suma= 0;
   int  Col;
   for (Col= 0; Col < NumCol; Col++)
        Suma= Suma + Datos[Ren][Col];
   return  Suma;
}
```

```
/* Método que suma todos los elementos de una columna. Para ello se
➥deben recorrer todos los renglones de dicha columna. El número de
➥columna a sumar se indica a través del parámetro. Si el tipo T usado
➥para crear el objeto ArregloBidimensional no fuera un número, entonces
➥se debería sobrecargar el operador + . */
template <class T>
T ArregloBidimensional<T>::SumaColumna(int Col)
{
   T Suma= 0;
   int  Ren;
   for (Ren= 0; Ren < NumRen; Ren++)
        Suma= Suma + Datos[Ren][Col];
   return  Suma;
}

/* Método que da como resultado el valor más grande almacenado en una
➥columna del arreglo, dada como dato. Para ello se deja fijo el valor de
➥la columna y se recorren todos los renglones. Si el tipo T usado para
➥crear el objeto ArregloBidimensional no fuera un número, entonces se
➥debería sobrecargar el operador > . */
template <class T>
T ArregloBidimensional<T>::MaximoColumna(int Col)
{
   T Maximo= Datos[0][Col];
   int  Ren;
   for (Ren= 1; Ren < NumRen; Ren++)
        if (Datos[Ren][Col] > Maximo)
             Maximo= Datos[Ren][Col];
   return  Maximo;
}

/* Método que da como resultado el valor más grande almacenado en un
➥renglón del arreglo, dado como dato. Para ello se deja fijo el valor
➥del renglón y se recorren todas las columnas. Si el tipo T usado para
➥crear el objeto ArregloBidimensional no fuera un número, entonces se
➥debería sobrecargar el operador >. */
template <class T>
T ArregloBidimensional<T>::MaximoRenglon(int Ren)
{
   T Maximo= Datos[Ren][0];
   int  Col;
   for (Col= 1; Col < NumCol; Col++)
        if (Datos[Ren][Col] > Maximo)
             Maximo= Datos[Ren][Col];
   return  Maximo;
}
```

```
/* Método que da como resultado el valor más pequeño almacenado en una
➥columna del arreglo, dada como dato. Para ello se deja fijo el valor de
➥la columna y se recorren todos los renglones. Si el tipo T usado para
➥crear el objeto ArregloBidimensional no fuera un número, entonces se
➥debería sobrecargar el operador <. */
template <class T>
T ArregloBidimensional<T>::MinimoColumna(int Col)
{
   T Minimo= Datos[0][Col];
   int  Ren;
   for (Ren= 1; Ren < NumRen; Ren++)
        if (Datos[Ren][Col] < Minimo)
             Minimo= Datos[Ren][Col];
   return  Minimo;
}

/* Método que da como resultado el valor más pequeño almacenado en un
➥renglón del arreglo, dado como dato. Para ello se deja fijo el valor
➥del renglón y se recorren todas las columnas. Si el tipo T usado para
➥crear el objeto ArregloBidimensional no fuera un número, entonces se
➥debería sobrecargar el operador <. */
template <class T>
T ArregloBidimensional<T>::MinimoRenglon(int Ren)
{
   T Minimo= Datos[Ren][0];
   int  Col;
   for (Col= 1; Col < NumCol; Col++)
        if (Datos[Ren][Col] < Minimo)
             Minimo= Datos[Ren][Col];
   return  Minimo;
}

/* Método que permite, a usuarios externos a la clase, conocer el
➥contenido de una de las casillas del arreglo. Recibe como parámetros
➥dos enteros, Ren y Col, que indican la posición (renglón y columna
➥respectivamente) del componente deseado del arreglo. El resultado es un
➥valor de tipo T. */
template <class T>
T ArregloBidimensional<T>::RegresaDato(int Ren, int  Col)
{
   return  Datos[Ren][Col];
}
```

La plantilla incluye el miembro privado: `T Datos[MAX] [MAX]` que es propiamente el arreglo; es decir, la colección de `MAX` por `MAX` elementos en la cual se especifican dos dimensiones. Los otros miembros privados representan el número actual de renglones (`NumRen`) y de columnas (`NumCol`) respectivamente.

El programa 4.8 presenta una aplicación de arreglos bidimensionales, usando la plantilla de la clase `ArregloBidimensional` definida en el programa 4.7. A partir de los datos de un grupo de alumnos (calificaciones obtenidas en varios exámenes) se calculan e imprimen algunos indicadores, como el promedio de calificación por alumno y el promedio de calificación por examen. La figura 4.9 muestra la representación gráfica de la estructura que almacena las calificaciones, indicando, en este ejemplo, que el primer alumno obtuvo 9 en el primer examen, 6 en el segundo, … que el segundo alumno obtuvo 7 en el primer examen, 10 en el segundo, … que el tercer alumno obtuvo 8 en el primer examen, 9 en el segundo, ...

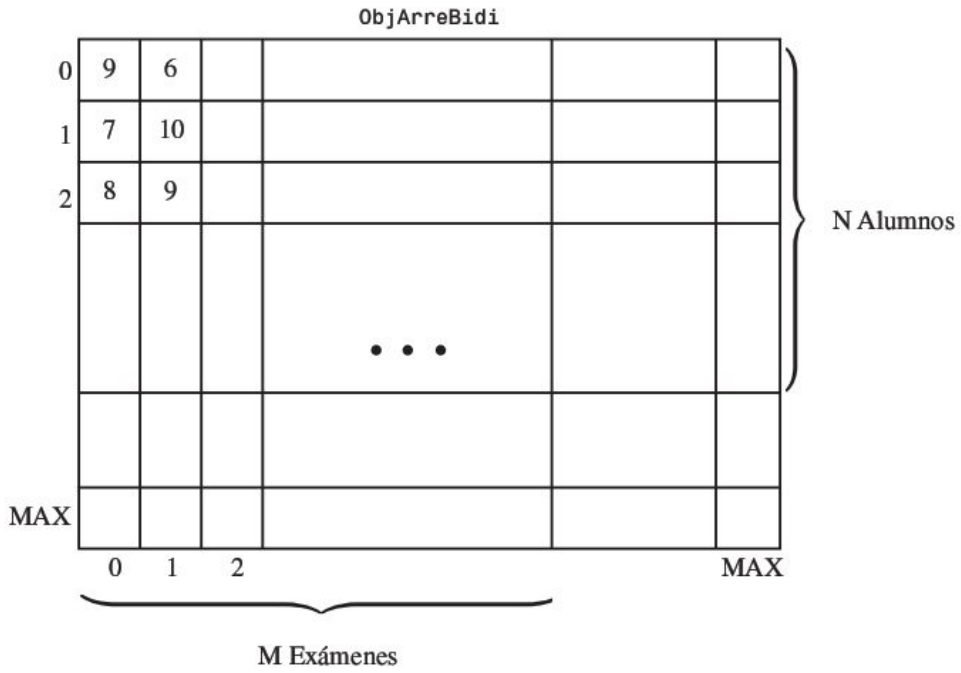


FIGURA 4.9 *Aplicación de arreglos bidimensionales*

Programa 4.8

```
/* Se tienen las calificaciones de un grupo de N alumnos obtenidas en M
➥exámenes. Para almacenar esta información se usa un objeto de tipo
➥ArregloBidimensional: los renglones representan a los alumnos y las
➥columnas a los exámenes. */

/* Se incluye la plantilla de la clase ArregloBidimensional correspon-
➥diente al programa 4.7, en la biblioteca "ArreBidi.h". */
#include "ArreBidi.h"


/* Función que despliega al usuario las opciones de trabajo sobre los
➥datos ingresados. */
int Menu()
{
   int  Opc;
   do {
        cout<<"\n\n 1- Listado de calificaciones de un alumno.";
        cout<<"\n\n 2- Listado de calificaciones de un examen.";
        cout<<"\n\n 3- Promedio de calificaciones de un alumno.";
        cout<<"\n\n 4- Promedio de calificaciones de un examen.";
        cout<<"\n\n 5- Calificación de un alumno obtenida en un examen.";
        cout<<"\n\n 6- Máxima calificación de un examen.";
        cout<<"\n\n 7- Mínima calificación de un examen.";
        cout<<"\n\n 8- Máxima calificación de un alumno.";
        cout<<"\n\n 90- Mínima calificación de un alumno.";
        cout<<"\n\n 10- Terminar. ";
        cout<<"\n\n Ingrese opción elegida:";
        cin>>Opc;
   } while (Opc < 1 || Opc > 10);
   return  Opc;
}


/* Plantilla de función para imprimir las calificaciones obtenidas por
➥un alumno (el usuario proporcionará un número para identificarlo) en
➥todos los exámenes. */
template <class  T>
void  CalifAlum(ArregloBidimensional <T> ObjArreBidi, int  NumExam)
{
   int  Alum, Exam;
   cout<<"\n\n Ingrese el número del alumno:";
   cin>>Alum;
   cout<<"\n\n Calificaciones obtenidas por el alumno en los exámenes\n";
   for (Exam = 0; Exam < NumExam; Exam++)
        cout<<"\nExamen: "<<Exam+1<<" -- "<<"Calif.: "
            <<ObjArreBidi.RegresaDato(Alum-1, Exam);
}
```

```
/* Plantilla de función para imprimir las calificaciones obtenidas en un
➥examen (el usuario proporcionará un número para identificarlo) por to-
➥dos los alumnos. */
template <class  T>
void  CalifExam(ArregloBidimensional <T> ObjArreBidi, int  NumAlum)
{
   int  Alum, Exam;
   cout<<"\n\n Ingrese el número del examen:";
   cin>>Exam;
   cout<<"\n\n Calificaciones obtenidas por los alumnos en el examen\n";
   for (Alum= 0; Alum < NumAlum; Alum++)
        cout<<"\nAlumno: "<<Alum+1<<" -- "<<"Calificación: "
            <<ObjArreBidi.RegresaDato(Alum, Exam-1);
}


/* El promedio de calificaciones obtenido por un alumno se calcula como
➥la suma de todos los elementos correspondientes al renglón del alumno
➥(sus calificaciones), entre el número de columnas (exámenes). */
template <class  T>
float PromAlum(ArregloBidimensional <T> ObjArreBidi, int  NumExam)
{
   int  Alum;
   cout<<"\n\n Ingrese el número del alumno:";
   cin>>Alum;
   return (float) (ObjArreBidi.SumaRenglon(Alum-1) / NumExam);
}


/* El promedio de calificaciones de un examen se calcula como la suma de
➥los elementos correspondientes a la columna del examen, entre el número
➥de renglones (alumnos). */
template <class  T>
float PromExam(ArregloBidimensional <T> ObjArreBidi, int  NumAlum)
{
   int  Exam;
   cout<<"\n\n Ingrese el número del examen:";
   cin>>Exam;
   return  (float) (ObjArreBidi.SumaColumna(Exam-1) / NumAlum);
}


/* Función que usa un arreglo bidimensional. Se declara un objeto del
➥tipo ArregloBidimensional para almacenar un conjunto de números enteros
➥que representan las calificaciones obtenidas por varios alumnos en di-
➥versos exámenes. */
void  UsaArregloBidimensional()
```

```
{
   int   Alum, Exam, NumAlum, NumExam, Opc;
   cout<<"\n\nIngrese el total de alumnos y el número de exámenes: ";
   cin>>NumAlum>>NumExam;

   ArregloBidimensional <int> ObjArreBidi(NumAlum, NumExam);
   cout<<"\n\nIngrese por cada alumno todas las calificaciones obtenidas
   ➥en los exámenes.\n";
   ObjArreBidi.Lectura();

   Opc= Menu();
   while (Opc >= 1 && Opc <= 9)
   {
       switch (Opc)
       {
          case  1: {
                     CalifAlum(ObjArreBidi, NumExam);
                     break;
                   }
          case  2: {
                     CalifExam(ObjArreBidi, NumAlum);
                     break;
                   }
          case  3: {
                     cout<<"\nEl Promedio del alumno es: "
                         <<PromAlum(ObjArreBidi, NumExam);
                     break;
                   }
          case  4: {
                     cout<<"\nPromedio de los alumnos en el examen es:"
                         <<PromExam(ObjArreBidi, NumAlum);
                     break;
                   }
          case  5: {
                     cout<<"\n\n Ingrese el número del alumno:";
                     cin>>Alum;
                     cout<<"\n\n Ingrese el número del examen:";
                     cin>>Exam;
                     cout<<"\nEl alumno "<<Alum<<" obtuvo en el examen
                     ➥"<<Exam<<":"
                        <<ObjArreBidi.RegresaDato(Alum-1, Exam-1);
                     break;
                   }
          case  6: {
                     cout<<"\n\n Ingrese el número del examen:";
                     cin>>Exam;
                     cout<<"\n\nMáxima calificación del examen "<<Exam<<" "
                         <<ObjArreBidi.MaximoColumna(Exam-1);
                     break;
                   }
```

```
            case  7: {
                       cout<<"\n\n Ingrese el número del examen:";
                       cin>>Exam;
                       cout<<"\n\nMínima calificación del examen
                       ➥"<<Exam<<" "
                           <<ObjArreBidi.MinimoColumna(Exam-1);
                       break;
                     }
            case  8: {
                       cout<<"\n\n Ingrese el número del alumno:";
                       cin>>Alum;
                       cout<<"\n\nMáxima calificación del alumno
                       ➥"<<Alum<<" "
                           <<ObjArreBidi.MaximoRenglon(Alum-1);
                       break;
                     }
            case  9: {
                       cout<<"\n\n Ingrese el número del alumno:";
                       cin>>Alum;
                       cout<<"\n\nMínima calificación del alumno
                       ➥"<<Alum<<" "
                           <<ObjArreBidi.MinimoRenglon(Alum-1);
                       break;
                     }
            }
            Opc= Menu();
    }
}
```

El programa 4.7 presentó la plantilla de la clase y el programa 4.8, un ejemplo de aplicación donde se hace uso de un objeto, instancia de dicha clase. La plantilla definida pretende ser una guía, sin embargo, podría incluir más métodos que faciliten la operación con los datos almacenados. Por ejemplo, se podrían definir unos que permitan conocer el valor de los atributos privados `NumRen` y `NumCol`.

4.6 Arreglos de objetos

El tipo de dato usado para declarar un objeto de la clase arreglo, puede a su vez ser una clase. Es decir, cada componente (o casilla) del arreglo será un objeto, y por lo tanto, el constructor se invoca para cada uno de ellos. Para que esto sea posible, la clase debe contar con un constructor por omisión o con un cons-

tructor con parámetros predeterminados. Si sólo tuviera un constructor con parámetros, se le deben proveer los mismos para cada celda, o provocaría un error. Por ejemplo, tomando la clase `Fecha`, definida anteriormente, si se quisiera declarar un arreglo de objetos de este tipo, usando un constructor con parámetros se debería hacer:

```
Fecha MesEnero[31] = { Fecha(1,1,2001), Fecha(2,1,2001),
…, Fecha(31,1,2001)};
```

En este caso sería necesario darle los 31 valores, para que con cada uno se cree e inicialice cada uno de los 31 objetos asignados a las respectivas casillas del arreglo. Con la notación `MesEnero[Indice]` se hace referencia al contenido de la casilla indicada por el valor de la variable `Indice`. Dicha casilla es un objeto, por lo que para manipularlo se tendrá en cuenta todo lo que se dijo acerca de los mismos.

A continuación se presenta el segmento de un programa que hace uso de un arreglo de objetos. No se utilizan las plantillas previamente declaradas ya que exige el uso de sobrecarga de operadores. Este último caso, se ejemplifica en el programa 4.11.

Programa 4.9

```
/* La clase Fecha contiene los atributos privados Día, Mes y Año. Además
➥tiene dos constructores y un método para imprimir los valores de los
➥atributos. */
class  Fecha
{
   private:
      int  Dia, Mes, Anio;
   public:
      Fecha(int, int , int );
      Fecha();
      void  ImprimeFecha();
};

/* Definición del método constructor con parámetros. */
Fecha::Fecha (int D, int  M, int  A): Dia(D),Mes(M), Anio(A)
{}

/* Definición del método constructor por omisión. */
Fecha::Fecha ()
{}
```

```
/* Método que despliega los valores de los atributos de una fecha. */
void  Fecha::ImprimeFecha ()
{
   cout<< "\nDía: " << Dia << "\tMes: " << Mes << "\tAño: " << Anio;
}

/* Función que usa un arreglo de objetos tipo Fecha. */
void  UsaArregloObjetos ()
{
   int  Indice;

   /* Se declara un arreglo de 3 objetos de tipo Fecha, usando el
   ➥constructor por omisión. */
   Fecha Cumpleanios[3];

   /* Se declaran 3 objetos de tipo Fecha, usando el constructor con
   ➥parámetros. */
   Fecha Cumple_Franco(18, 9, 2005);
   Fecha Cumple_Monica(12, 4, 2005);
   Fecha Cumple_Rodrigo(25, 11, 2005);

   /* Se declara e inicializa un arreglo de 2 objetos de tipo Fecha. */
   Fecha DiasFestivos[2]= {Fecha (21, 3, 2005), Fecha (1, 5, 2005)};

   /* Se asignan valores (objetos) a las casillas del arreglo. */
   Cumpleanios[0]= Cumple_Franco;
   Cumpleanios[1]= Cumple_Monica;
   Cumpleanios[2]= Cumple_Rodrigo;

   /* Impresión del contenido de los arreglos. */
   for (Indice= 0; Indice < 3; Indice++)
        Cumpleanios[Indice].ImprimeFecha();

   for (Indice= 0; Indice < 2; Indice++)
        DiasFestivos[Indice].ImprimeFecha();
}
```

En el ejemplo anterior, al declarar el arreglo `Cumpleanios`, se utilizó el constructor por omisión de la clase `Fecha`. Por lo tanto, en cada una de las casillas del arreglo se creó un objeto cuyos atributos quedaron indeterminados. Por su parte, en el arreglo `DiasFestivos`, se usó (de la clase `Fecha`) el constructor con parámetros. Como consecuencia, cada una de sus casillas almacena un objeto cuyos atributos están instanciados con los valores proporcionados al constructor. Las variables `Cumpleanio` y `DiasFestivos` son arreglos, por lo tanto para tener acceso a cada uno de sus elementos se usa un `índice`. Una vez que se hace referencia a uno de ellos,

como éste es un objeto, se deben usar los métodos propios de dicho objeto para tener acceso a sus miembros.

A continuación se presenta otro ejemplo de arreglo de objetos. Se define una clase y posteriormente un arreglo de objetos de dicha clase para ejemplificar el uso del concepto estudiado.

Programa 4.10

```
/* Se declara la clase Cliente la cual define un cliente por medio de
➥los atributos: Nombre, Dirección, Teléfono, Saldo, Tipo de Cuenta y
➥Número de Cuenta, y de algunos métodos que permiten el manejo de los
➥mismos. Para la clase Cliente se definieron dos métodos constructores,
➥uno de los cuales es por omisión. Asimismo, se desarrolla una pequeña
➥aplicación que hace uso de la clase definida. */

class  Cliente
{
   private:
       char Nombre[64], Direccion[32], Telefono[10];
       float Saldo;
       int  TipoDeCuenta, NumDeCuenta;
   public:
       Cliente();
       Cliente(char [],char [], char [], float, int , int );
       float ObtenerSaldo();
       void  ImprimeDatos();
       char ObtenerTipoCta();
       void  HacerRetiro(float);
       void  HacerDeposito(float);
};

/* Definición del método constructor por omisión. */
Cliente::Cliente()
{ }

/* Definición del método constructor con parámetros. */
Cliente::Cliente(char Nom[],char Dir[], char Tel[], float Sal, int
➥TCta, int  NoCta)
{
   strcpy(Nombre, Nom);
   strcpy(Direccion, Dir);
   strcpy(Telefono, Tel);
   Saldo= Sal;
   TipoDeCuenta= TCta;
   NumDeCuenta= NoCta;
}
```

```
/* Método que permite tener acceso, a usuarios externos a la clase, al
➥saldo de un cliente. */
float Cliente::ObtenerSaldo()
{
   return  Saldo;
}

/* Método que despliega en pantalla los atributos de un cliente. */
void  Cliente::ImprimeDatos()
{
   cout<< "Nombre: " << Nombre << '\n';
   cout<< "Dirección: " << Direccion << '\n';
   cout<< "Teléfono: " << Telefono << '\n';
   cout<< "Saldo: " << Saldo << '\n';
   cout<< "Tipo de Cuenta: " << TipoDeCuenta << '\n';
   cout<< "Número de Cuenta: " << NumDeCuenta << '\n';
}

/* Método que permite tener acceso, a usuarios externos a la clase, al
➥tipo de cuenta de un cliente. */
int Cliente::ObtenerTipoCta()
{
   return  TipoDeCuenta;
}

/* Método que permite tener acceso, a usuarios externos a la clase, al
➥número de cuenta de un cliente. */
int Cliente::ObtenerNumCta()
{
   return  NumDeCuenta;
}

/* Método para registrar un retiro de una cuenta del cliente. El método
➥verifica que el saldo de la cuenta sea mayor o igual al monto que va
➥a retirar. Si se cumple esta condición, actualiza el saldo. En caso
➥contrario, imprime un mensaje. */
void  Cliente::HacerRetiro(float Monto)
{
   if ((Saldo - Monto) < 0)
       cout<< "No se puede hacer el retiro.\n ";
   else
       Saldo= Saldo - Monto;
}

/* Método que registra un depósito a la cuenta del cliente. Actualiza el
➥saldo. */
void  Cliente::HacerDeposito(float Monto)
{
   Saldo= Saldo + Monto;
}
```

```
/* Función que usa un arreglo de objetos tipo Cliente. Se realizan
➥algunas operaciones en las cuentas de los clientes de dos bancos. */
void  UsaArregloObjetos ()
{
   int  Indice, TipoC, NumC;
   float Saldo, Monto;
   char Nom[64], Direc[64], Telef[64];

   /* Declaración de dos arreglos de 100 objetos de tipo Cliente. Se
   ➥hace uso del constructor por omisión. */
   Cliente ClientesBanco1[100];
   Cliente ClientesBanco2[100];

   /* Se crean tres objetos de tipo Cliente usando el constructor con
   ➥parámetros. */
   Cliente ObjCli1("Laura", "Insurgentes 2564", "55559900", 28000, 2, 2509);
   Cliente ObjCli2("Juan", "Reforma 3600", "55408881", 4000, 1, 8324 );
   Cliente ObjCli3("Tomas", "Tlalpan 1005", "56703311", 20000, 2, 7604);

   /* Asignación de objetos al arreglo correspondiente al primer banco. */
   ClientesBanco1[0]= ObjCli1;
   ClientesBanco1[1]= ObjCli2;
   ClientesBanco1[2]= ObjCli3;

   /* Impresión de los datos correspondientes a los clientes del primer
   ➥banco. */
   for (Indice= 0; Indice < 3; Indice++)
       ClientesBanco1[Indice].ImprimeDatos();

   /* Lectura de los datos de los clientes del segundo banco. Primero se
   ➥leerán valores para cada uno de los atributos definidos en la clase
   ➥Cliente. Posteriormente se creará un objeto usando el método cons-
   ➥tructor con parámetros y finalmente se asignará dicho objeto a una
   ➥casilla del arreglo. Estos pasos se repiten para cada cliente. */
   for (Indice= 0; Indice < 20; Indice++)
   {
       cout<<"\n\nIngrese datos del cliente:  "<<Indice+1<<\n\n";
       cin>>Nom>>Direc>>Telef>>Saldo>>TipoC>>NumC;
       Cliente ObjCli (Nom, Direc, Telef, Saldo, TipoC, NumC) ;
       ClientesBanco2[Indice]= ObjCli;
   }

   /* Registro de un retiro de $1000 de la cuenta del tercer cliente del
   ➥segundo banco. */
   ClientesBanco2[2].HacerRetiro(1000);

   /* Impresión de los datos de todos los clientes que tienen un saldo
   ➥mayor a $10000. */
```

```
    cout<<"\nReporte de clientes con saldo superior a $10000\n";
    for (Indice= 0; Indice < 20; Indice++)
        if (ClientesBanco2[Indice].ObtenerSaldo() > 10000)
            ClientesBanco2[Indice].ImprimeDatos();

    /* Registro de un depósito a cierta cuenta. El número de cuenta y el
    ➥monto son dados por el usuario. */
    cout<<"\n\nIngrese el número de cuenta a la cual va a depositar y el
    ➥monto del depósito \n";
    cin>>NumC>>Monto;

    /* Se aplica búsqueda secuencial para buscar el cliente con el número
    ➥de cuenta dado. */
    Indice= 0;
    while (Indice < 20 && NumC != ClientesBanco2[Indice].ObtenerNumCta() )
       Indice++;
    if (Indice < 20)
       ClientesBanco2[Indice].HacerDeposito(Monto);
    else
       cout<<"\nNo está registrado ningún cliente con el número de
       ➥cuenta dado. \n";
}
```

En los programas anteriores se presentaron algunas aplicaciones sencillas de arreglos de objetos. Es importante señalar, que la captura de los datos de cada uno de los clientes puede hacerse de manera directa usando sobrecarga en la operación de lectura (**cin**), tal como se vio en el capítulo anterior. Por otra parte, si se usa sobrecarga en la operación de escritura (**cout**) se podrá omitir el método `ImprimeDatos`.

El programa 4.11 presenta otro ejemplo de arreglos de objetos. En este caso se usó la plantilla de arreglos desordenados (ver el programa 4.1) para declarar un arreglo de objetos de la clase `Dinos`. Para mayor claridad se incluye la definición de la clase `Dinos`.

Programa 4.11

```
/* Se define la clase Dinos la cual se usa como base para declarar el
➥tipo de datos de un arreglo. En la biblioteca "ArreDesor.h" se incluye
➥la plantilla de la clase de arreglos desordenados presentada en el
➥programa 4.1. La aplicación permite leer los elementos del arreglo, dar
➥de alta nuevos dinosaurios, dar de baja dinosaurios registrados e
➥imprimir todos los datos de los mismos. */
```

```
#define MAX 100

#include "ArreDesor.h"

/* Definición de la clase Dinos. Se sobrecargan operadores para que los
➥objetos de esta clase puedan utilizarse directamente en los métodos de
➥los arreglos. */
class  Dinos
{
   private:
       int  Clave;
       char Nombre[MAX], Alimen[MAX], Periodo[MAX], Region[MAX];
   public:
       Dinos();
       Dinos(int , char [], char [],char [],char []);
       int   operator!= (Dinos);
       friend  istream &operator>> (istream &, Dinos &);
       friend  ostream &operator<< (ostream &, Dinos &);
};

/* Definición del método constructor por omisión. */
Dinos::Dinos()
{}

/* Definición del método constructor con parámetros. */
Dinos::Dinos(int  Cla, char  Nom[], char  Ali[],char  Per[],char  Reg[])
{
   Clave= Cla;
   strcpy(Nombre, Nom);
   strcpy(Alimen, Ali);
   strcpy(Periodo, Per);
   strcpy(Region, Reg);
}

/* Sobrecarga del operador != para comparar objetos de tipo Dinos.
➥De esta forma el método de búsqueda en arreglos puede aplicarse también
➥a objetos de este tipo. */
int  Dinos::operator!= (Dinos ObjD)
{
   if   ((Clave != ObjD.Clave) ||  (strcmp(Nombre, ObjD.Nombre) != 0) ||
         (strcmp(Alimen, ObjD.Alimen) != 0) ||  (strcmp(Periodo,
         ➥ObjD.Periodo) != 0) ||
         (strcmp(Region, ObjD.Region) != 0))
         return   1;
   else
         return   0;
}
```

```
/* Sobrecarga del operador >> para permitir la lectura directa de
➥objetos de tipo Dinos. De esta forma, el método Lectura de la clase
➥Arreglo puede ser usado con objetos de este tipo. */
istream &operator>> (istream &Lee,  Dinos &ObjDino)
{
   cout<<"\n\nIngrese clave del dinosaurio:  ";
   Lee>> ObjDino.Clave;
   cout<<"\n\nIngrese nombre del dinosaurio:  ";
   Lee>> ObjDino.Nombre;
   cout<<"\n\nIngrese tipo de alimentación del dinosaurio:  ";
   Lee>> ObjDino.Alimen;
   cout<<"\n\nIngrese periodo en el que vivió el dinosaurio:  ";
   Lee>> ObjDino.Periodo;
   cout<<"\n\nIngrese región en la que vivió el dinosaurio:  ";
   Lee>> ObjDino.Region;
   return    Lee;
}

/* Sobrecarga del operador << para permitir la impresión directa de
➥objetos de tipo Dinos. De esta forma, el método Escribe de la clase
➥Arreglo puede ser usado con objetos de este tipo. */
ostream &operator<< (ostream &Escribe, Dinos &ObjDino)
{
   Escribe<<"\n\nDatos del dinosaurio\n";
   Escribe<<"\nClave:  "<<ObjDino.Clave;
   Escribe<<"\nNombre:  "<<ObjDino.Nombre;
   Escribe<<"\nAlimentación:  "<<ObjDino.Alimen;
   Escribe<<"\nPeriodo:  "<<ObjDino.Periodo;
   Escribe<<"\nRegión:  "<<ObjDino.Region;
   return    Escribe;
}


/* Se define la clase Menu que permite desplegar al usuario las opciones
➥de trabajo de la aplicación. */
class  Menu
{
   public:
      Menu();
      int   Despliega();
};

/* Definición del método constructor. */
Menu::Menu()
{}
```

```
/* Definición del método que muestra las opciones de trabajo. */
int  Menu::Despliega()
{
   int   Opc;
   do {
       cout<<"\n\nBienvenido al sistema del Museo de los
       ➥Dinosaurios\n\n";
       cout<<"\nQué desea hacer?\n";
       cout<<"\n 1-Registrar un nuevo dinosaurio. ";
       cout<<"\n 2-Dar de baja un dinosaurio.";
       cout<<"\n 3-Obtener un listado de todos los dinosaurios
       ➥registrados. ";
       cout<<"\n 4-Terminar.\n";
       cout<<"\n\nIngrese la opción elegida:  ";
       cin>>Opc;
   } while (Opc < 1 || Opc > 4);
   return   Opc;
}


/* Función principal que hace uso de la plantilla del arreglo y de las
➥clases. La aplicación permite al usuario almacenar los datos de varios
➥dinosaurios, dar de alta/baja dinosaurios e imprimir los datos de los
➥mismos. */
void  main ()
{
   Arreglo<Dinos> Parque;
   Dinos ObjDino;
   Menu Opciones;
   int   Opc, Res;
   /* Se lee el total de dinosaurios a almacenar y los datos de cada uno
   ➥de ellos por medio del método Lectura. Para que dicho método pueda
   ➥ser usado es necesaria la sobrecarga del operador >> en la clase
   ➥Dinos. */
   Parque.Lectura();
   Opc= Opciones.Despliega();

   while  (Opc>= 1 && Opc <= 3)
     {
       switch (Opc)
       {
           /* Se da de alta un dinosaurio si el arreglo tiene espacio y
           ➥si no se repiten los datos del dinosaurio. Se usa la sobre-
           ➥carga del operador >>. */
           case 1: {
                     cin>>ObjDino;
                     Res= Parque.InsertaDesordenado(ObjDino);
                     if  (Res == 1)
                          cout<<"\n\nDinosaurio registrado.\n";
```

```
                    else
                        if  (Res == 0)
                            cout<<"\n\nNo se tiene espacio para
                                ➥registrar nuevos dinos.\n";
                        else
                            cout<<"\n\nEse dinosaurio ya fue registrado
                                ➥previamente. \n";
                    break;
                    }
            /* Se elimina un dinosaurio si el arreglo no está vacío y si
            ➥el dinosaurio dado como dato fue registrado previamente. Se
            ➥usa la sobrecarga del operador >>. */
            case 2: {
                    cin>>ObjDino;
                    Res= Parque.EliminaDesordenado(ObjDino);
                    if  (Res == 1)
                        cout<<"\n\nDinosaurio eliminado.\n";
                    else
                        if  (Res == 0)
                            cout<<"\n\nNo se tiene registrado ningún
                            ➥dinosaurio.\n";
                        else
                            cout<<"\n\nEse dinosaurio no fue
                            ➥registrado. \n";
                    break;
                    }
            /* Se despliegan en pantalla todos los datos de los
            ➥dinosaurios almacenados en el arreglo por medio del método
            ➥Escribe. Para que dicho método pueda ser usado es necesaria
            ➥la sobrecarga del operador << en la clase Dinos. */
            case 3: {
                    Parque.Escribe();
                    break;
                   }
             }
            Opc= Opciones.Despliega();
    }
}
```

En el programa 4.11, para usar la plantilla de la clase arreglos fue necesario sobrecargar algunos operadores para que los métodos definidos pudieran aplicarse a objetos. Es decir, la sobrecarga debe hacerse en la clase que se usará como tipo base. En el ejemplo, al sobrecargar el operador de desigualdad **!=** y los operadores **<<** y **>>** en la clase `Dinos`, se logró que los métodos `Lectura`, `Escribe` y `Busca` de la clase `Arreglo` se pudieran usar indistintamente con números (programa 4.2) y con objetos (programa 4.11).

4.7 Casos especiales de arreglos

Se tienen algunos casos especiales de arreglos según la cantidad y distribución de sus componentes. Los casos más estudiados son: matrices poco densas y matrices triangulares.

4.7.1 Matrices poco densas

Las ***matrices poco densas*** son aquellas en las cuales gran parte de sus elementos son cero o vacío (cualquier representación que indique la ausencia de datos), este último valor para arreglos que no sean numéricos. La figura 4.10 presenta un ejemplo de este tipo de arreglos. Como se puede observar, en este arreglo, casi el 80% de sus elementos son ceros.

Matriz poco densa

9	0	0	0	0	0	0	0	2
7	10	0	0	0	3	0	0	0
0	0	0	0	5	0	0	7	0
0	0	0	0	0	0	3	0	0
0	0	0	8	0	9	0	0	0
0	4	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	8

FIGURA 4.10 *Ejemplo de matriz poco densa*

Cuando se trata de arreglos muy grandes, resulta conveniente ahorrar espacio de memoria almacenando en un arreglo unidimensional sólo aquellos valores que no sean ceros. Para cada uno de ellos se debe guardar, además del dato, la posición (renglón y columna) que le corresponde en la matriz original. Para ello se va a definir una clase que represente cada uno de los elementos de la matriz que se desea guardar. La figura 4.11 muestra la clase `Componente` en la cual se incluyeron como atributos dos enteros y un dato de tipo `T` para almacenar el renglón, la columna y el valor distinto de cero respectivamente.

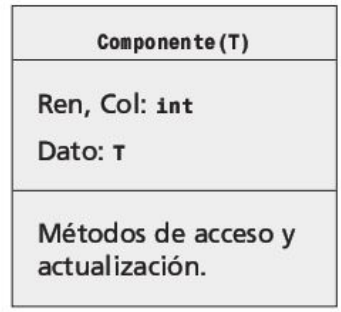


FIGURA 4.11 *Clase* `Componente`

A continuación se presenta la codificación, usando el lenguaje ***C++***, de la plantilla de la figura 4.11.

```
/* Prototipo de la clase MatPocoDen para poder declararla como amiga
➥de la clase Componente, y de esta forma darle acceso a los miembros
➥privados de esta última. */
template <class  T>
class  MatPocoDen;

/* Definición de la plantilla de clase que representa cada uno de los
➥elementos distintos de cero de la matriz poco densa. */
template <class  T>
class  Componente
{
   private:
       T Dato;
       int  Ren, Col;
   public:
       Componente();
       Componente(T, int , int);
       friend class  MatPocoDen<T>;
       friend istream &operator>>(istream &, Componente &);
       friend ostream &operator<<(ostream &, Componente &);
};

/* Declaración del método constructor por omisión. */
template <class  T>
Componente<T>::Componente()
{}

/* Declaración del método constructor con parámetros. */
```

```
template <class  T>
Componente<T>::Componente(T Valor, int  Re, int  Co)
{
   Dato= Valor;
   Ren= Re;
   Col= Co;
}

/* Declaración de la función amiga que sobrecarga el operador >>. */
template <class  T>
istream &operator>>(istream &Lee, Componente<T> &Com)
{
   cout<<"\n\nIngrese el valor:  ";
   Lee>>Com.Dato;
   cout<<"\n\nIngrese el número del renglón que le corresponde:  ";
   Lee>>Com.Ren;
   cout<<"\n\nIngrese el número de la columna que le corresponde:  ";
   Lee>>Com.Col;
   return   Lee;
}

/* Declaración de la función amiga que sobrecarga el operador <<. */
template <class  T>
ostream &operator<<(ostream &Escribe, Componente<T> &Com)
{
   Escribe<<Com.Dato<<"  ";
   return   Escribe;
}
```


A partir de esta clase, se define la correspondiente a la matriz poco densa. Ésta se representa por medio de un arreglo unidimensional y por dos números enteros, que almacenan el total de renglones y de columnas de la matriz original. Como esta clase fue declarada amiga de la anterior podrá usar directamente sus miembros privados. El programa 4.12 presenta esta plantilla y una aplicación de la misma.

Programa 4.12

```
/* Plantilla de la clase correspondiente a una matriz poco densa,
➥almacenada por medio de un arreglo unidimensional de objetos. Los
➥atributos son la colección de componentes formados por el valor diferente
➥de cero, el renglón y la columna que le corresponden en la matriz
➥original. Además, se guardan el total de renglones y de columnas que
➥tiene la matriz original y el total de elementos diferentes de cero.*/
```

```
template <class  T>
class  MatPocoDen
{
   private:
       Componente<T>  Valores[MAX];
       int  TotRen, TotCol, TotVal;
   public:
       MatPocoDen();
       void  Lectura();
       void  Imprime();
};

/* Declaración del método constructor por omisión. */
template <class  T>
MatPocoDen<T>::MatPocoDen()
{
   TotVal= 0;
}

/* Método que lee los datos de la matriz que son distintos de cero,
➥junto con el renglón y la columna que le corresponde en la matriz
➥original. Los valores leídos se van guardando en un arreglo
➥unidimensional. */
template <class  T>
void  MatPocoDen<T>::Lectura()
{
   int  IndRen, IndCol, Resp;
   T Dato;
   Do {
       cout<<"\n\nIngrese total de renglones y columnas de la matriz\n";
       cin>>TotRen>>TotCol;
   } while (TotRen <= 0 || TotCol <= 0);
   cout<<"\n\nIngrese 1 si desea capturar datos, 0 para terminar. \n";
   cin>>Resp;
   while (Resp)
   {
       cout<<"\n\nIngrese los datos diferentes de 0 (o vacío).\n";
       cin>>Dato;
       do {
           cout<<"\nQué renglón le corresponde - de 0 a "<<TotRen<<":  ";
           cin>>IndRen;
       } while (IndRen < 0 || IndRen >= TotRen);
       do {
           cout<<"\nQué columna le corresponde - de 0 a "<<TotCol<<":  ";
           cin>>IndCol;
```

```
        } while (IndCol < 0 || IndCol >= TotCol);
        Componente<T> Elemento(Dato, IndRen, IndCol);
        Valores[TotVal]= Elemento;
        TotVal++;
        cout<<"\n\nIngrese 1 si desea capturar más datos, 0 para
        ➥terminar. \n";
        cin>>Resp;
    }
}

/* Método que despliega en pantalla los valores diferentes de cero de la
➥matriz poco densa. */
template <class  T>
void  MatPocoDen<T>::Imprime()
{
    int  Indice;
    cout<<"\n\nValores almacenados\n\n";
    for (Indice= 0; Indice < TotVal; Indice++)
         cout<<Valores[Indice]<<"  ";
    cout<<"\n\n\n";
}


/* Función principal en la que se hace uso de las clases definidas para
➥la representación de la matriz poco densa. */
void  main()
{
    MatPocoDen<int> Matriz1;
    Matriz1.Lectura();
    Matriz1.Imprime();
    MatPocoDen<Arbol> Matriz2;
    Matriz2.Lectura();
    Matriz2.Imprime();
}
```

En la aplicación, se muestra el uso de la plantilla. En la declaración del objeto `Matriz2` se utilizó otra clase para darle valor a `T`. El uso de la clase `Arbol` presupone que en la misma se sobrecargaron los operadores **<<** y **>>**, para que los métodos de lectura e impresión puedan hacer uso de **`cin`** y de **`cout`** (ver en el capítulo anterior el programa 3.2).

Sobre este arreglo se pueden realizar operaciones como las que se analizaron en la sección de arreglos bidimensionales, pero requieren ciertas adaptaciones. A continuación se presenta el método que suma los elementos de un renglón.

```
/* Método que realiza la suma de los elementos de un renglón de una
➥matriz poco densa almacenada en un arreglo unidimensional. Recibe como
➥parámetro el renglón a sumar y da como resultado la suma del mismo. */
template <class  T>
T MatPocoDen<T>::SumaRen(int Renglon)
{
   T Suma= 0;
   int  Indice;
   for (Indice= 0; Indice < TotVal; Indice++)
       if (Valores[Indice].Ren == Renglon)
           Suma= Suma + Valores[Indice].Dato;
   return  Suma;
}
```

En el método presentado se recorren todos los valores guardados en el arreglo unidimensional y se suman aquellos que corresponden al renglón deseado.

El siguiente método encuentra el valor más grande de una cierta columna. Tanto en el método anterior como en éste, si se estuviera trabajando con objetos, en lugar de números, se deberían sobrecargar los operadores > y + en la clase correspondiente. Observe que las operaciones resultan ser menos claras que si se trataran como arreglos bidimensionales.

```
/* Método que encuentra el valor más grande de una columna de una matriz
➥poco densa almacenada en un arreglo unidimensional. Recibe el número de
➥la columna que interesa y regresa el máximo elemento de dicha columna. */
template <class T>
T MatPocoDen<T>::MaxCol(int Colum)
{
     T Maximo;
     int  Indice, Band=1;
     for (Indice= 0; Indice < TotVal; Indice++)
           if (Valores[Indice].Col == Colum && Band)
           {
                 Maximo= Valores[Indice].Dato;
                 Band= 0;
           }
           else
                 if (Valores[Indice].Col == Colum &&
                    ➥Valores[Indice].Dato > Maximo)
                       Maximo= Valores[Indice].Dato;
     return  Maximo;
}
```

4.7.2 Matrices triangulares

Las ***matrices triangulares*** son matrices cuadradas (N renglones × N columnas) que guardan información sólo en las casillas que están de la diagonal principal hacia arriba o hacia abajo, incluyendo la diagonal. Según el caso, reciben el nombre de ***matriz triangular superior*** o ***matriz triangular inferior***.

La figura 4.12 presenta ejemplos de matrices triangulares superiores (a) e inferiores (b). Como se puede apreciar todos los elementos que están debajo o encima de la diagonal principal son ceros (o vacíos). Por lo tanto, en matrices de gran tamaño resulta conveniente (para ahorrar espacio de memoria) almacenar sólo los valores distintos de cero, usando arreglos unidimensionales.

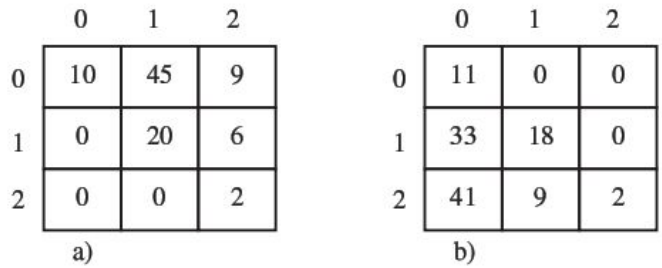


FIGURA 4.12 *Ejemplo de matrices triangulares*
(a) Matriz triangular superior, (b) Matriz triangular inferior

Matriz triangular superior

Una matriz de este tipo tendrá N + N – 1 + N – 2 + ... + 3 + 2 + 1 elementos distintos de cero, lo cual puede expresarse como:

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

Por otra parte, habrá 0 + 1 + 2 + ... + N – 1 ceros en la matriz, lo cual puede generalizarse como:

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

Dado un cierto renglón `Ren`, se tendrán:

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

ceros correspondientes a los (`Ren - 1`) renglones previos y `N * (Ren - 1)` elementos en total (ceros y distintos de cero).

Para poder recuperar los valores guardados en el arreglo, se sugiere utilizar la fórmula que se presenta a continuación. Considere que ***C++*** enumera los renglones de 0 a (N – 1), el total de elementos guardados antes del renglón `Ren`, se calcula como `Ren * N`.

$$\text{Posición } (Dato[Ren][Col] = N * Ren - \frac{(Ren - 1) * Ren}{2} + (Col - Ren)$$

El primer término hace referencia al total de elementos almacenados antes del renglón `Ren`. A esta cantidad se le resta el total de ceros de dichos renglones y se le suma el número que corresponde al desplazamiento dentro del mismo renglón.

Dado el arreglo de la figura 4.13, almacenado en un arreglo unidimensional como el que aparece en la figura 4.14, se aplica la fórmula vista para recuperar sus elementos. Por ejemplo:

Posición (`Dato[0][0]`)= 4 * 0 – ((0 – 1) * 0) / 2) + (0 – 0)= 0

El 25 se guardó en la casilla 0.

Posición (`Dato[1][3]`)= 4 * 1 – ((1 – 1) * 1) / 2)+ (3 – 1)= 6

El 63 se guardó en la casilla 6.

Posición (`Dato[2][2]`)= 4 * 2 – ((2 – 1) * 2) / 2) + (2 – 2)= 7

El 43 se guardó en la casilla 7.

	0	1	2	3
0	25	67	87	43
1	0	41	29	63
2	0	0	43	15
3	0	0	0	16

FIGURA 4.13 *Matriz triangular superior*

25	67	87	43	41	29	63	43	15	16
0	1	2	3	4	5	6	7	8	9

FIGURA 4.14 *Representación lineal de la matriz triangular superior*

El programa 4.13 presenta una plantilla para la clase `MatrizTrianSup`, que incluye como atributos la colección de elementos y el orden del arreglo cuadrado. Se definieron como métodos el cálculo del total de elementos almacenados, así como el cálculo de la posición en la que se encuentra cierto valor.

Programa 4.13

```
/* Constante que define el máximo número de elementos que se pueden
➥almacenar en el arreglo unidimensional. */
#define MAX 50

/* Definición de la clase MatrizTrianSup. Sus atributos son un arreglo
➥unidimensional en el cual se guardarán los valores de la matriz
➥triangular superior y el orden de la misma. */
template <class  T>
class  MatrizTrianSup
```

```
{
   private:
       T Datos[MAX];
       int  Dim;
   public:
       MatrizTrianSup();
       int  RegresaPosic(int, int);
       int  TotalDatos();
       void  Lectura();
       void  ImprimeMatriz();
       void  ImprimeDatos();
};

/* Método constructor por omisión. */
template <class  T>
MatrizTrianSup<T>::MatrizTrianSup()
{}

/* Método que calcula la posición que le corresponde a un elemento de la
➥matriz dentro del arreglo unidimensional en el cual fue guardado. */
template <class  T>
int MatrizTrianSup<T>::RegresaPosic(int Ren, int  Col)
{
   return  (Dim * Ren - ((Ren - 1) * Ren) / 2 + (Col - Ren));
}

/* Método que calcula el total de elementos guardados en el arreglo
➥unidimensional, que son los que estaban de la diagonal principal hacia
➥arriba. */
template <class  T>
int MatrizTrianSup<T>::TotalDatos()
{
   return  ((Dim * (Dim + 1)) / 2) ;
}

/* Método que lee del teclado los valores para los atributos de la
➥clase. Al usuario sólo se le piden los valores que están en la diagonal
➥y arriba de ella. */
template <class  T>
void  MatrizTrianSup<T>::Lectura()
{
   int  Ren, Col, Indice= 0;
   do {
       cout<<"\n\nIngrese orden de la matriz triangular superior: ";
       cin>>Dim;
   } while (Dim > MAX || Dim < 0);
   for (Ren= 0; Ren < Dim; Ren++)
        for (Col= Ren; Col < Dim; Col++)
```

```
        {
            cout<<"\n\nIngrese el elemento "<<Ren+1<<" - "<<Col+1<<"  ";
            cin>>Datos[Indice];
            Indice= Indice + 1;
        }
}

/* Método que imprime los valores almacenados con forma de arreglo
➥bidimensional. */
template <class  T>
void  MatrizTrianSup<T>::ImprimeMatriz()
{
   int  Ren, Col, Indice;
   cout<<"\n\nMatriz triangular superior\n\n";
   for (Ren= 0; Ren < Dim; Ren++)
   {
        for (Col= 0; Col < Dim; Col++)
            if (Ren <= Col)
            {
                Indice= RegresaPosic(Ren, Col);
                cout<<Datos[Indice]<<" - ";
            }
            else
                cout<<"0 - ";
        cout<<"\n";
   }
   cout<<"\n\n";
}

/* Método que imprime sólo los valores almacenados. */
template <class  T>
void  MatrizTrianSup<T>::ImprimeDatos()
{
   int  Indice, TotElem;
   TotElem= TotalDatos();
   cout<<"\n\nElementos de la matriz triangular superior\n\n";
   for (Indice= 0; Indice < TotElem; Indice++)
        cout<<Datos[Indice]<<"  ";
   cout<<"\n\n";
}
```

Matriz triangular inferior

Una matriz de este tipo tendrá 1 + 2 + 3 + ... + N elementos distintos de cero, lo cual puede expresarse como:

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

Para recuperar los valores guardados en el arreglo, se sugiere utilizar la fórmula que se presenta a continuación. Dado que *C++* enumera las casillas de 0 a (N – 1), para usar esta fórmula, debe sumarle uno al renglón y a la columna. Por ejemplo, si quiere recuperar el elemento (1, 1) debe darle a `Ren` y `Col` el valor de 2.

$$\text{Posición } (\texttt{Dato[Ren][Col]}) = \frac{(\texttt{Ren} - 1) * \texttt{Ren}}{2} + (\texttt{Col} - 1)$$

Dado el arreglo de la figura 4.15, almacenado en un arreglo unidimensional como el que aparece en la figura 4.16, se aplica la fórmula vista para recuperar sus elementos. Por ejemplo:

Posición (`Dato[0][0]`)= ((1 – 1) * 1) / 2 + (1 – 1)= 0

El 25 se guardó en la casilla 0.

Posición (`Dato[2][1]`)= ((3 – 1) * 3) / 2 + (2 – 1)= 4

El 45 se guardó en la casilla 4.

Posición (`Dato[3][3]`)= ((4 – 1) * 4) / 2 + (4 – 1)= 9

El 16 se guardó en la casilla 9.

	0	1	2	3
0	25	0	0	0
1	18	41	0	0
2	39	45	43	0
3	9	38	22	16

FIGURA 4.15 *Matriz triangular inferior*

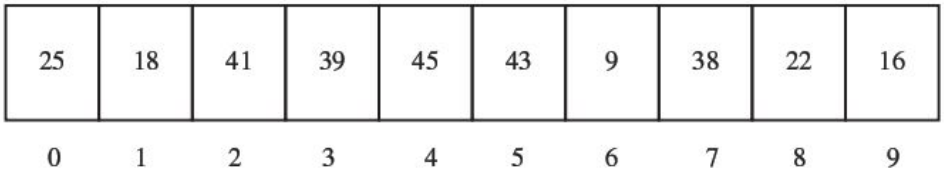
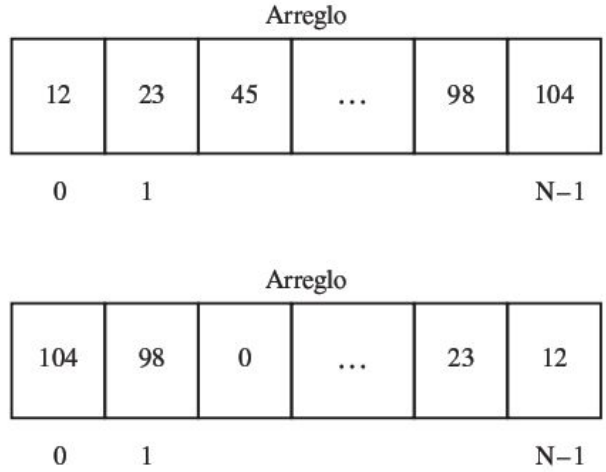


FIGURA 4.16 *Representación lineal de la matriz triangular inferior*

La plantilla para la clase `MatrizTrianInf` es similar a la que se presentó en el programa 4.13, sólo cambia la forma de calcular la posición de un dato de la matriz guardado en el arreglo unidimensional.

Ejercicios

1. Defina la clase `ArregloEnteros`. Determine los atributos y el conjunto de métodos (lo más completo posible) que caracterizan al concepto arreglo unidimensional de números enteros.
2. Utilice la clase definida en el ejercicio 1 para almacenar la edad de un grupo de N ($1 \leq N \leq 30$) alumnos. Una vez almacenados los datos, calcule e imprima el promedio de edad del grupo, así como el total de alumnos con una edad mayor al promedio.
   a) El cálculo del promedio y el cálculo del total de alumnos con edad mayor al promedio debe hacerse con métodos de la clase.
   b) El cálculo del promedio y el cálculo del total de alumnos con edad mayor al promedio NO puede hacerse con métodos de la clase. Utilice alguno(s) de los conceptos vistos en los capítulos anteriores.
3. Escriba un programa que invierta el orden de los elementos de un objeto tipo arreglo. Tome como ejemplo el siguiente esquema. Para la declaración del objeto puede usar alguna de las plantillas presentadas en este capítulo. ¿Requiere definir nuevos métodos? ¿Puede solucionar el problema de alguna otra forma?



4. Escriba un programa que, utilizando la plantilla vista para arreglos desordenados, almacene las calificaciones de un grupo de N ($1 \leq N \leq 80$) alumnos. Las calificaciones serán números reales comprendidos entre 0 y 10. A partir de los datos guardados en el arreglo, su programa debe realizar las siguientes operaciones. Puede agregar métodos a la plantilla de la clase arreglo, si lo cree necesario.

   a) Imprimir la calificación más alta, la más baja y el promedio de las mismas.

   b) Obtener e imprimir el total de calificaciones menores a 6.

   c) Obtener e imprimir el total de calificaciones mayores a 8.5.

5. Utilice la plantilla de la clase Arreglo para definir un objeto arreglo de números reales en el cual almacene los precios de N artículos. Escriba una aplicación que permita encontrar e imprimir el precio más alto.

6. Defina la clase `Arreglo` usando plantillas y sobrecarga de operadores para representar las operaciones de inserción (operador **+**) y de eliminación (operador **–**).

7. Utilice la plantilla de la clase `ArreParal` definida en este capítulo para declarar dos objetos: un arreglo de enteros y un arreglo de números reales. El primero permitirá almacenar las claves de un grupo de N productos ($1 \leq N \leq 30$), ordenadas crecientemente, mientras que el segundo será para

guardar los precios de dichos productos. Escriba un programa en ***C++*** que, mediante un menú de opciones, permita al usuario:

a) Leer y validar el número de productos.

b) Leer la información correspondiente de cada uno de los N productos.

c) Dada la clave de un producto, poder actualizar su precio.

d) Dar de baja un producto.

e) Dar de alta un nuevo producto.

f) Imprimir las claves de todos los productos cuyos precios sean mayores a uno dado como referencia por el usuario.

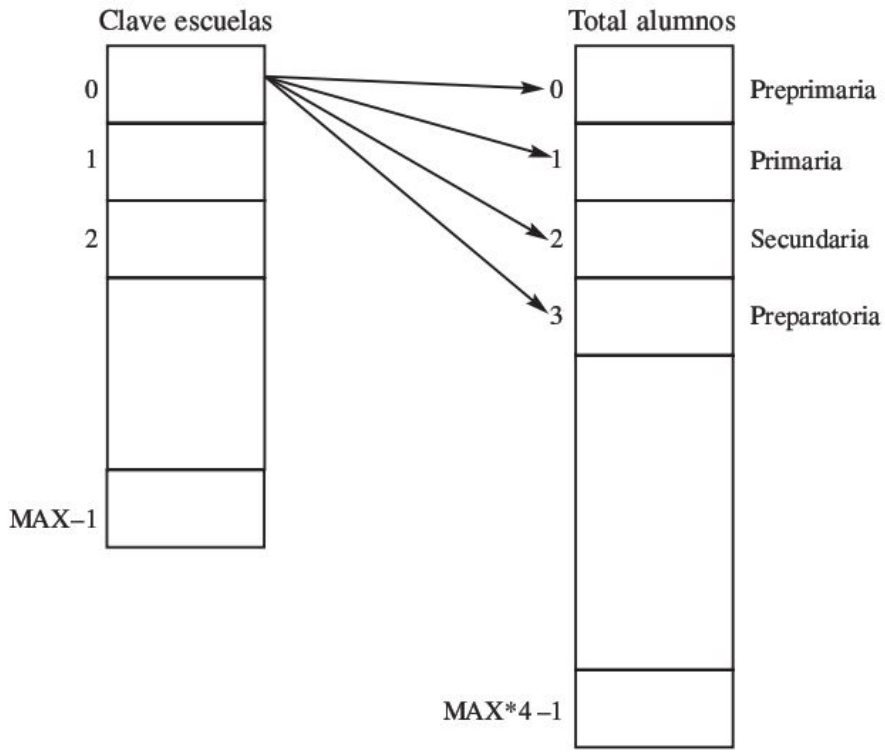
8. Se tienen 3 arreglos paralelos: el primero almacena las claves de 20 productos, ordenadas crecientemente; el segundo guarda la existencia de cada uno de ellos, y el tercero almacena el precio de venta de los mismos. Escriba un programa completo en ***C++*** que permita:

   a) Consultar: a.1) El producto con mayor existencia. a.2) El producto con mayor precio de venta. En ambos casos el programa debe imprimir todos los datos del producto que cumpla con la condición buscada.

   b) Actualizar: b.1) La existencia de un producto (si se venden algunas unidades o se compran más). Los datos de entrada son la clave del producto, una clave de operación (para indicar si es venta o compra) y la cantidad vendida/comprada. b.2) El precio de venta de un producto. Los datos de entrada son la clave del producto y el nuevo precio de venta.

   c) Eliminar: un producto. El dato de entrada es la clave.

9. Se tienen 2 arreglos paralelos. El primero de ellos almacena las claves de N ($1 \leq N \leq 50$) escuelas, ordenadas ascendentemente. En el segundo se almacena, por escuela, el total de alumnos de preprimaria, primaria, secundaria y preparatoria. Observe el siguiente esquema. La escuela, cuya clave está en la casilla 0 del primer arreglo, tiene 624 alumnos en preprimaria, 1600 en primaria, 1260 en secundaria y 893 en preparatoria. Los totales de alumnos, por nivel, de la segunda escuela ocuparán las casillas 4, 5, 6 y 7 del segundo arreglo, y así sucesivamente.



Teniendo en cuenta estas especificaciones para guardar los datos, escriba un programa en ***C++*** que:

a) Genere un reporte que imprima, de cada una de las escuelas, el total de alumnos en cada una de las secciones y el total general de la escuela. El usuario podrá dar la clave de una escuela o pedir un listado de todas las escuelas.

b) Calcule e imprima el total de alumnos de cualquiera de las 4 secciones, considerando todas las escuelas. El usuario indicará la sección elegida.

c) Calcule e imprima el total de alumnos en cada una de las 4 secciones, considerando todas las escuelas. Es decir, el total de alumnos en preprimaria, primaria, etcétera, tenga en cuenta las N escuelas. ¿Puede reutilizar la solución del inciso b)?

d) Registre una nueva escuela. Los datos proporcionados por el usuario serán la clave de la escuela y el número de alumnos en cada una de las 4 secciones. Si la escuela no tiene alguna de las secciones se ingresará un 0.

e) Elimine alguna de las escuelas. El dato proporcionado por el usuario será la clave de la escuela.

f) Actualice los totales de alumnos en alguna sección (o en todas). El dato proporcionado por el usuario será la clave de la escuela, la clave de la sección (o secciones) y el nuevo número de alumnos.

10. Escriba un programa que sume dos objetos de tipo arreglos bidimensionales de enteros. Utilice la plantilla de la clase arreglo bidimensional para declarar los objetos. Modifíquela si lo cree necesario. El programa debe imprimir el arreglo resultante.

$$C_{MxN} = A_{MxN} + B_{MxN}$$

11. Escriba un programa que multiplique dos objetos de tipo arreglos bidimensionales de números reales. Utilice la plantilla de la clase arreglo bidimensional para declarar los objetos. Modifíquela si lo cree necesario.

$$C_{MxN} = A_{MxN} * B_{MxN}$$

12. Escriba un programa que imprima los elementos de la diagonal principal de un arreglo bidimensional. Utilice la plantilla de la clase arreglo bidimensional para declarar el objeto. Modifíquela si lo cree necesario.

13. Escriba un programa que sume los elementos de la diagonal principal de un arreglo bidimensional de números reales. Utilice la plantilla de la clase arreglo bidimensional para declarar el objeto. Modifíquela si lo cree necesario.

14. Escriba un programa que obtenga la traspuesta de una matriz cuadrada (arreglo bidimensional de N por N elementos). Por ejemplo, si la matriz dada es a), su traspuesta es b).

	0	1	2
0	10	–12	6
1	45	20	17
2	9	6	2

a) Matriz

	0	1	2
0	10	45	9
1	–12	20	6
2	6	17	2

b) Traspuesta

15. Considerando las especificaciones (que aparecen más adelante) de las clases `Alumno` y `Arreglo` escriba un programa completo en ***C++*** que:

    a) Lea el número de alumnos registrados en una cierta carrera y capture los datos correspondientes a los mismos.

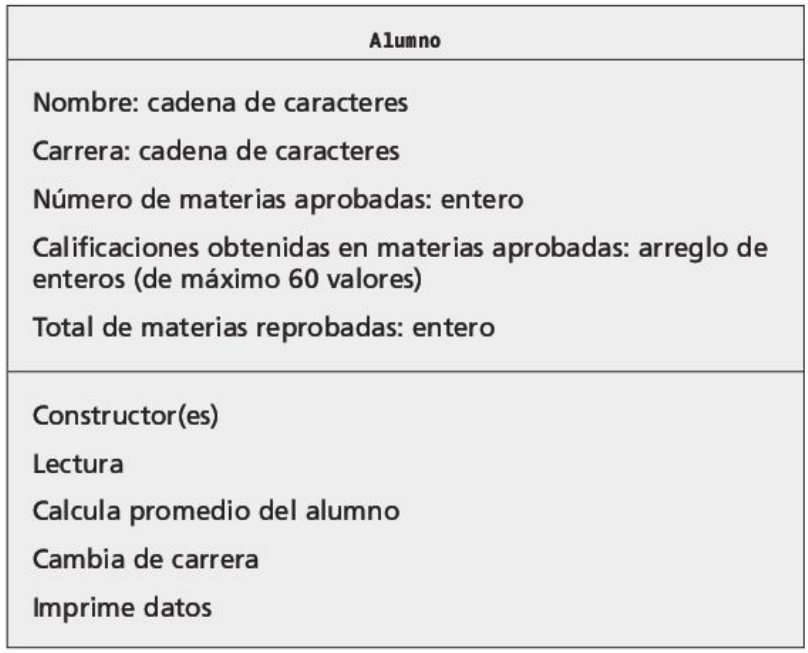
    b) Obtenga e imprima el promedio de cada uno de los alumnos y el promedio del grupo.

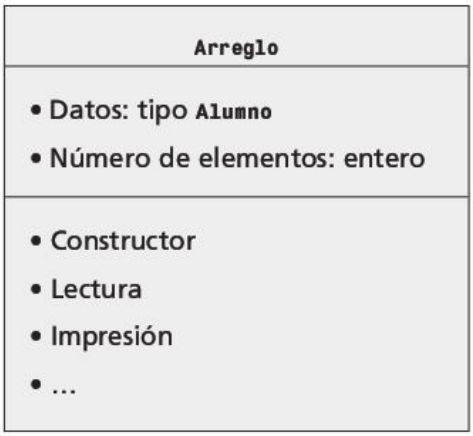
    c) Dado un alumno y una nueva carrera registrar el cambio de carrera correspondiente. Se debe validar que dicho alumno haya sido almacenado previamente en el arreglo.

    d) Imprimir todos los datos de aquellos alumnos que lleven más de 25 materias aprobadas.

    e) Dar de baja de la carrera a aquellos alumnos que hayan completado igual (o mayor) número de materias reprobadas que aprobadas.

    f) Dar de alta un alumno nuevo.





16. Defina la clase `SocioClub` según las especificaciones que se dan más adelante. Utilice alguna de las plantillas previamente definidas para declarar un arreglo de objetos de la clase `SocioClub`. Escriba un programa en ***C++***, que mediante menús pueda:

    a) Leer y validar N ($1 \leq N \leq 60$).

    b) Leer los N elementos del arreglo.

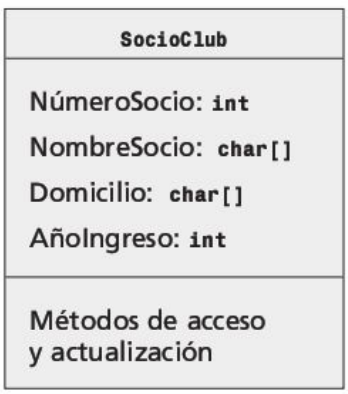
    c) Imprimir los datos de todos los socios con más de 10 años de antigüedad.

    d) Cambiar el domicilio de un socio.

    e) Dado el número de un socio, imprimir toda su información.

    f) Dar de alta un nuevo socio.

    g) Dar de baja un socio existente.



17. Defina la clase `Automóvil` teniendo en cuenta las especificaciones que se dan más adelante. Para decidir qué métodos incluir, lea cuidadosamente el resto del problema. Declare un arreglo de N ($1 \leq N \leq 120$) objetos tipo `Automóvil`, el cual almacenará la flotilla de automóviles de una empresa. Asuma que los mismos serán dados de manera ordenada, crecientemente, por `ClaveAuto`. Escriba un programa en ***C++***, que mediante menús pueda:

    a) Leer y validar N ($1 \leq N \leq 120$).

    b) Leer los N elementos del arreglo.

    c) Imprimir los datos de todos los automóviles que hayan sido fabricados en cierto año. El usuario dará como dato el año deseado.

    d) Imprimir los datos de todos los automóviles que sean de cierta marca. El usuario dará como dato la marca.

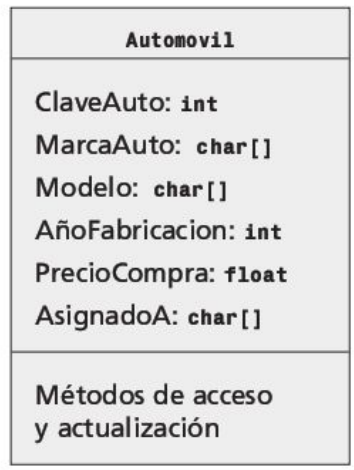
    e) Imprimir los datos de todos los automóviles que sean de cierta marca y de cierto modelo. El usuario dará como dato la marca y el modelo.

    f) Cambiar el nombre de la persona a la cual se le ha asignado el automóvil. El usuario dará como dato la clave del automóvil y el nombre de la persona que ahora lo usará.

    g) Dada la clave de un automóvil, imprimir toda su información.

    h) Dar de alta un nuevo automóvil. El usuario ingresará como datos toda la información relacionada al nuevo automóvil.

    i) Dar de baja un automóvil existente. El usuario dará como dato la clave del automóvil que desea eliminar de la flotilla.



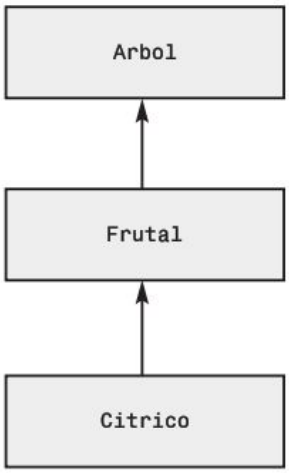
18. Considere la siguiente relación de herencia entre clases. Defina las clases `Arbol`, `Frutal` y `Citrico` de tal manera que pueda declarar un arreglo polimórfico, es decir, un arreglo que pueda almacenar objetos de diferentes tipos, en este caso de las tres clases indicadas. Decida qué atributos y métodos incluir, tenga en cuenta lo que se pide más a continuación. Escriba una programa de aplicación en ***C++*** que pueda, por medio de menús:

    a) Imprimir los atributos de objetos tipo `Arbol`, `Frutal` y `Citrico`.

    b) Dar de alta nuevos objetos, de cualquiera de los 3 tipos mencionados.

    c) Dar de baja un objeto previamente almacenado.

    d) Imprimir todos los datos de los objetos que tengan una altura mayor a los 2 metros.



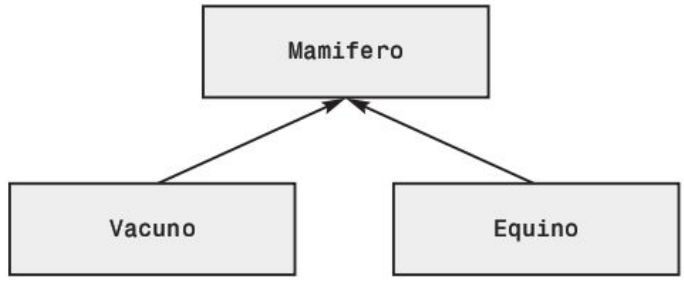
19. Considere la siguiente relación de herencia entre clases. Defina las clases `Mamifero`, `Vacuno` y `Equino` de tal manera que pueda declarar un arreglo polimórfico, es decir, un arreglo que pueda almacenar objetos de diferentes tipos, en este caso de las tres clases indicadas. Decida qué atributos y métodos incluir, tenga en cuenta lo que se pide a continuación. Escriba un programa de aplicación en ***C++*** que pueda, por medio de menús:

    a) Imprimir todos los atributos de objetos tipo `Mamifero`, `Vacuno` y `Equino`.

    b) Dar de alta nuevos objetos, de cualquiera de los 3 tipos mencionados.

    c) Dar de baja un objeto previamente almacenado.

d) Actualizar el establecimiento donde habita alguno de los animales. El dato que dará el usuario será la clave del animal y el nombre del establecimiento al cual fue trasladado.



20. Escriba un método para sumar los elementos de una columna de una matriz poco densa, guardada en memoria por medio de un arreglo unidimensional. El usuario dará como dato el número de la columna a sumar.
21. Escriba un método que imprima una matriz poco densa almacenada por medio de un arreglo unidimensional, con formato de arreglo bidimensional. Es decir, el usuario verá en la pantalla la matriz con su forma original.
22. Escriba una función que sume dos matrices poco densas almacenadas de acuerdo a lo visto en este libro. ¿Requiere modificar las plantillas definidas?
23. Defina la plantilla correspondiente a una matriz triangular inferior.
24. Retome el problema anterior. Incluya un método en la plantilla que permita encontrar el valor más grande de un renglón. El usuario dará como dato el número del renglón que le interesa.
25. Escriba un programa que sume dos matrices triangulares superiores, almacenadas en arreglos unidimensionales.
26. Escriba un programa que multiplique dos matrices triangulares inferiores, almacenadas en arreglos unidimensionales.
27. Se llama `matriz tridiagonal` a aquella que tiene valores distintos de cero sólo en la diagonal principal y en las diagonales que están por encima y por debajo de ésta. Observe la siguiente figura. Si la matriz es grande, conviene almacenar (para ahorrar espacio de memoria) sólo los valores distintos de cero en un arreglo unidimensional. Encuentre una fórmula que calcule la posición en la que fueron guardados (en un arreglo unidimensional) los elementos de una matriz tridiagonal.

	0	1	2	3
0	25	10	0	0
1	18	41	25	0
2	0	45	56	31
3			22	16

28. Retome el problema anterior. Defina una plantilla para una clase que represente este tipo de matrices.

29. Se llama `matriz simétrica` cuando se cumple la condición:

```
Datos[Ren][Col] = Datos[Col][Ren]
```

para todo $1 \leq$ Ren, Col $\leq$ orden del arreglo. Cuando se presenta un caso así, resulta conveniente guardar sólo la matriz triangular inferior o superior, ya que de lo contrario se duplicaría la información. Defina una plantilla para una clase que represente este tipo de matrices.

30. Considere que los siguientes datos representan los costos de boletos de avión entre ciudades. Cuando no existe vuelo directo entre dos ciudades aparece un cero, y los valores de la diagonal principal no se toman en cuenta ya que no hay vuelos de una ciudad a sí misma. Por ejemplo, en la siguiente figura, se representó que ir de la ciudad 0 a la ciudad 1 cuesta $1,000 (lo mismo de la 1 a la 0) y que ir de la ciudad 2 a la 3 cuesta $2,050 (lo mismo de la 3 a la 2). Además, no hay vuelo de la ciudad 1 a la 3.

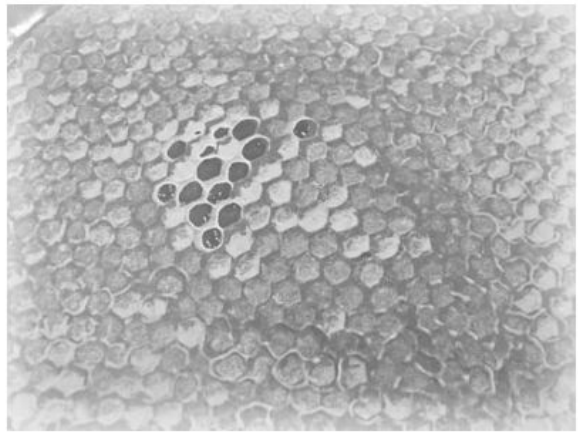
	0	1	2	3
0	–	1000	890	720
1	1000	–	1250	0
2	890	1250	–	2050
3	720	0	2050	–

Escriba un programa en ***C++*** que, por medio de menús, permita realizar las siguientes operaciones. Utilice la plantilla del problema anterior.

a) Dado un número que identifica a una ciudad (proporcionado por el usuario), genere un reporte de todas las ciudades destinos a las que se puede llegar a partir de dicha ciudad.

b) Dado un número que identifica a una ciudad origen y otro que identifica a una ciudad destino (ambos proporcionados por el usuario) indique si hay vuelo directo entre ambas ciudades, y si es así, su costo.

c) Genere un reporte de todas las ciudades entre las que no existen vuelos directos.



CAPÍTULO 5

Pilas y colas

5.1 Introducción

En este capítulo se presentan las estructuras de datos **`pilas`** y **`colas`**, y algunas variantes de estas últimas: las colas circulares y las colas dobles. Ambas estructuras son lineales, es decir cada elemento tiene un único sucesor y un único predecesor, con excepción del primero y del último. El primero carece de antecesor y el último de sucesor.

Estas estructuras se caracterizan por la manera en que llevan a cabo la inserción y eliminación de elementos. En el caso de las pilas los elementos se pueden agregar o quitar por un único extremo, mientras que en la estructura tipo cola los elementos se insertan por un extremo y se quitan por otro. A continuación se analizarán de manera más detallada estas estructuras, las operaciones que pueden realizarse sobre ellas y algunas aplicaciones.

5.2 Pilas

Una ***pila*** es una estructura de datos lineal en la cual los elementos pueden insertarse y eliminarse sólo por uno de los extremos. Por lo tanto, el último elemento insertado será el primero que podrá eliminarse; debido a esta característica, también se le conoce como estructura ***LIFO*** (por sus siglas del inglés: ***L***ast-***I***n, ***F***irst-***O***ut: último en entrar, primero en salir).

El concepto de pila se utiliza en muchas actividades cotidianas, por ejemplo cuando se exponen libros en una librería o latas de un cierto producto en un supermercado. En ambos casos se tienen pilas, de libros o de latas, y es de suponer que si un cliente quiere, por ejemplo un libro, tomará el que está más arriba, que fue el último en colocarse.

El extremo en el cual se realizan las operaciones se denomina ***tope*** de la pila. El tope apunta al último valor almacenado y se modifica con cada operación. Es decir, se incrementa al insertar un nuevo valor o se decrementa al eliminar un valor. La figura 5.1 muestra una representación gráfica de una pila, en la cual se han almacenado dos elementos. El tope apunta al último valor insertado.

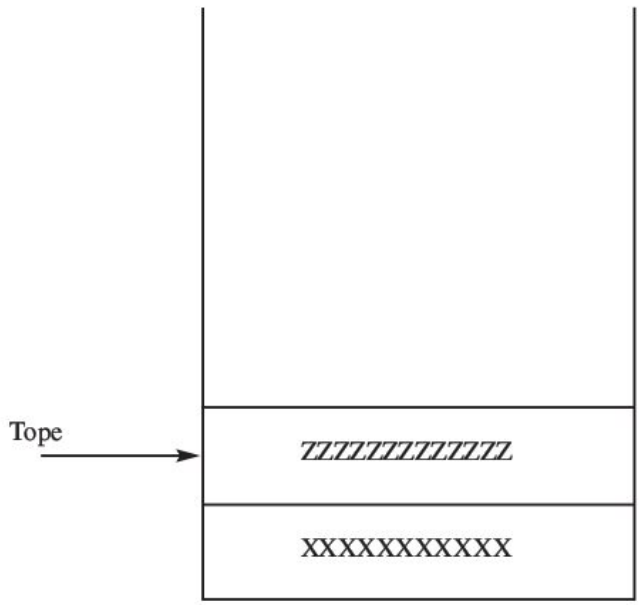


FIGURA 5.1 *Estructura tipo pila*

La pila es una ***estructura abstracta***. Para el almacenamiento de los datos en la memoria de la computadora debe usarse otra estructura. Para los efectos de este libro, se utilizarán arreglos unidimensionales. Teniendo en cuenta esta aclaración, la clase `Pila` estará formada por dos atributos: la colección de elementos a guardar (por medio de un arreglo unidimensional) y el apuntador al último elemento almacenado (`Tope`). Además, tendrá algunos métodos que se analizarán en la siguiente sección. La figura 5.2 presenta una plantilla de la clase `Pila`. Se usa una plantilla para dar mayor generalidad a la solución.

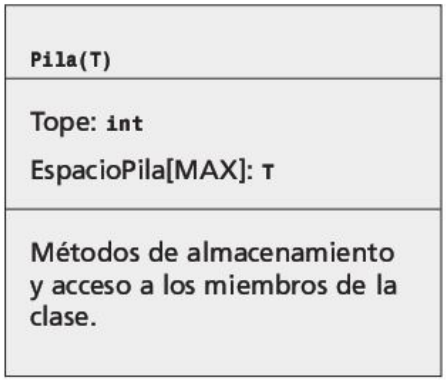


FIGURA 5.2 *Clase* `Pila`

A continuación se presenta la codificación de la plantilla de la clase `Pila`, usando el lenguaje ***C++***.

```
/* Definición del número máximo de elementos que puede contener la
➥estructura pila, restricción propia de los arreglos. */
#define MAX 10

/* Definición de la plantilla de la clase Pila que tendrá como atributos
➥la colección de elementos (haciendo uso de un arreglo) y un apuntador
➥al primero de ellos. Es decir, al primer elemento al cual se podrá
➥tener acceso, que es el último elemento almacenado. En la plantilla
➥también se hace referencia a algunos métodos, que se analizarán con
➥detalle en la siguiente sección. */
```

```
template <class T>
class Pila
{
   private:
       T EspacioPila[MAX];
       int Tope;
   public:
        Pila();

/* Métodos de modificación y acceso a los miembros de la pila. */
};

/* Declaración del método constructor. Inicializa el Tope en -1,
➥indicando pila vacía. */
template <class T>
Pila<T>::Pila()
{
   Tope= -1;
}
```

Operaciones

Las operaciones de inserción y eliminación son las únicas que pueden realizarse en este tipo de estructuras. Las mismas, como ya se mencionó, se llevan a cabo sólo por uno de los extremos de la pila, al que se conoce con el nombre de *tope*.

La operación de inserción (*Push*) consiste en incrementar el tope de la pila y agregar el nuevo valor en esa posición. Antes de insertar el elemento es necesario verificar que en la pila haya espacio disponible. La manera de evaluar esta condición depende del tipo de estructura elegida para almacenar la colección de elementos en la pila.

Considerando que en este libro se usará un arreglo unidimensional, la disponibilidad de espacio dependerá de que aún no se hayan ocupado todas las casillas del mismo.

En la figura 5.3a, la pila tiene almacenados dos valores. La figura 5.3b muestra el estado de la pila luego de insertar el valor 6. Observe que el `Tope` se modifica, apuntando ahora al último valor agregado. Finalmente, en la figura 5.3c se muestra la pila luego de insertar el 9.

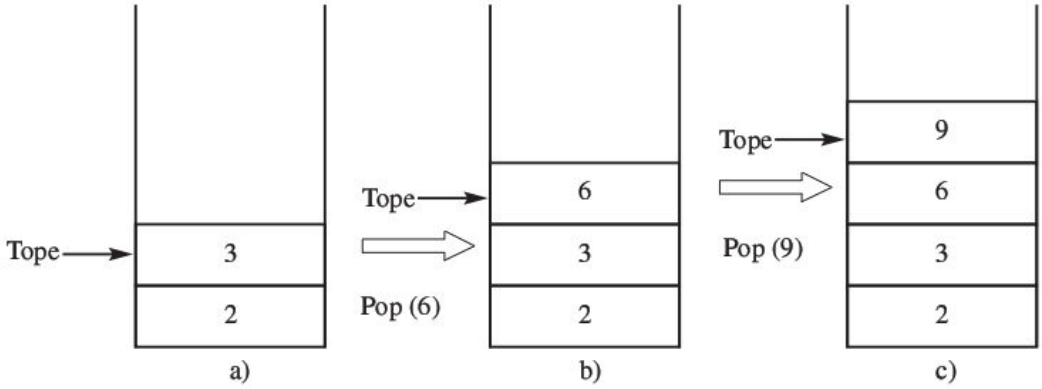


FIGURA 5.3 *Operación de inserción en pilas*

La operación de eliminación (*Pop*) consiste en quitar el valor que se encuentra almacenado en el tope de la pila y disminuir en uno el valor del tope. Previamente se debe validar que la pila no esté vacía. La manera de evaluar esta condición depende del tipo de estructura elegida para almacenar la colección de elementos en la pila. Considerando que en este libro se usará un arreglo unidimensional, la pila estará vacía cuando el puntero tenga un valor de –1.

La figura 5.4a muestra el estado de la pila una vez eliminado el elemento almacenado en la posición del tope (9) de la figura 5.3c. Observe que el valor del `Tope` disminuyó en uno, apuntando ahora al siguiente elemento de la pila (6). La figura 5.4b presenta la pila luego de quitar el 6, y finalmente la figura 5.4c el estado de la pila luego de eliminar el 3.

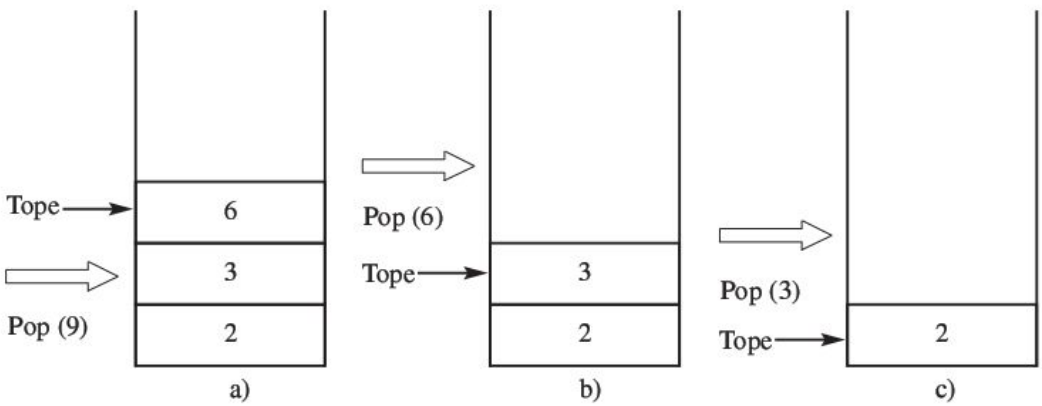


FIGURA 5.4 *Operación de eliminación en pilas*

Es importante destacar que el único elemento que se puede quitar es el que está en la posición indicada por el tope. Por lo tanto, si debido a la aplicación se requiere eliminar un valor que ocupa una posición intermedia, primero se deberán quitar (al menos temporalmente) a partir del tope, todos los elementos que estén por encima del elemento deseado.

A continuación se presentan las plantillas de los métodos correspondientes a las operaciones analizadas. Estos métodos requieren de operaciones auxiliares para verificar el estado de la pila. En el caso de la inserción es necesario saber si la pila tiene espacio disponible. Para ello se usa un método que comprueba si la pila está llena. En el caso de la eliminación debe saber si hay elementos en la pila, para lo cual se utiliza un método que determina si la pila está vacía. La verificación del estado de la pila puede incorporarse a los métodos de inserción y eliminación. Sin embargo, para darle una mayor modularidad a los algoritmos se prefirió manejarlos como métodos auxiliares independientes.

```
/* Plantilla del método que introduce un dato a una estructura pila, si la
➥misma tiene espacio disponible. En caso afirmativo, actualiza el valor
➥del tope. El método recibe como parámetro el dato que va a insertar. */
template <class T>
void Pila<T>::Push(T Dato)
{
   /* Verifica si hay espacio disponible en la pila. Si es así,
   ➥incrementa el valor de Tope y asigna el valor Dato a la casilla
   ➥indicada por éste. */
   if (!Pila::PilaLlena())
        EspacioPila[++Tope]=  Dato;
   else
        cout<<"\nError de desbordamiento. Pila llena.";
}

/* Plantilla del método que elimina el elemento que está en el tope de
➥la pila (si no está vacía), actualizando el valor del mismo. El método
➥regresa el dato eliminado por medio de un parámetro por referencia.  */
template <class T>
void Pila<T>::Pop(T *Dato)
{
   /* Verifica que haya al menos un elemento en la pila. Si es así,
   ➥asigna al parámetro el valor que está almacenado en la casilla
   ➥indicada por Tope y disminuye a éste en uno. */
   if (!Pila::PilaVacia())
        *Dato= EspacioPila[Tope--];
   else
        cout<<"\nError de subdesbordamiento. Pila vacía.";
}
```

```
/* Plantilla del método auxiliar que verifica si la pila está llena,
➥es decir si ya no hay espacio disponible. Regresa 1 si todas las
➥casillas están ocupadas y 0 en caso contrario. */
template <class T>
int Pila<T>::PilaLlena()
{
   if (Tope == MAX-1)
        return 1;
   else
        return 0;
}

/* Plantilla del método auxiliar que verifica si la pila está vacía.
➥Regresa 1 si no hay ningún elemento y 0 en caso contrario. */
template <class T>
int Pila<T>::PilaVacia()
{
   if (Tope == -1)
       return 1;
   else
       return 0;
}
```

Los métodos `Push` y `Pop` imprimen un mensaje en caso de que las operaciones no se puedan llevar a cabo. Sin embargo, los mismos podrían definirse como métodos enteros, de tal forma que pudieran dar como resultado un número que indique si la operación se realizó con éxito o no. Esto último resulta más útil a los procesos usuarios de la clase, ya que podrían tomar decisiones de acuerdo al resultado obtenido. Por su parte, los mensajes resultan más claros en esta etapa de aprendizaje. Como ejemplo, se presenta a continuación el método `Push` con la modificación sugerida.


```
/* Plantilla del método que, si hay espacio disponible, introduce un
➥dato en la pila y actualiza el tope. Da como resultado un número entero
➥que indica si la operación de inserción pudo efectuarse (1) o no (0).
➥El método recibe como parámetro el dato a insertar. */
template <class T>
int Pila<T>::Push(T Dato)
{
   /* La variable Res se inicializa en 0 (fracaso). Si se realiza la
   ➥inserción se le asignará 1. */
   int Res= 0;
```

```
    if (!Pila::PilaLlena())
    {
            EspacioPila[++Tope]= Dato;
            Res= 1;
    }
    return Res;
}
```

El programa 5.1 presenta la plantilla completa de la clase pila con todos sus métodos y un ejemplo sencillo de aplicación. En este programa, las operaciones de validación de existencia de espacio disponible y de existencia de al menos un elemento en la pila (para las operaciones de inserción y eliminación) se hacen antes de invocar a los métodos a fin de mostrar otra manera de estructurarlos. En los métodos previamente presentados, esta validación se lleva a cabo dentro de los mismos.

Programa 5.1

```
/* Definición del número máximo de elementos que puede contener la pila,
➥restricción que resulta del uso de un arreglo unidimensional. */
#define MAX 10

/ * Se define la plantilla de la clase Pila con todos sus atributos y mé-
➥todos. Además, se incluye una pequeña aplicación de la misma. */
template <class T>
class Pila
{
   private:
       T EspacioPila[MAX];
       int Tope;
   public:
       Pila();
       void Push(T Dato);
       void Pop(T *Dato);
       int PilaLlena();
       int PilaVacia();
};

/* Declaración del método constructor. Inicializa el Tope en -1,
➥indicando pila vacía. */
template <class T>
Pila<T>::Pila()
```

```
{
   Tope= -1;
}

/* Método que introduce un dato en la pila, actualizando el tope de la
➥misma. El uso de este método presupone que antes de invocarlo se debe
➥verificar que haya espacio disponible en la pila. */
template <class T>
void Pila<T>::Push(T Dato)
{
   EspacioPila[++Tope]= Dato;
}

/* Método que quita al elemento que está en el tope de la pila y lo
➥asigna a un parámetro por referencia. El uso de este método presupone
➥que antes de invocarlo se debe verificar que la pila no esté vacía. */
template <class T>
void Pila<T>::Pop(T *Dato)
{
   *Dato= EspacioPila[Tope--];
}

/* Método auxiliar que verifica si la pila está llena. Regresa 1 si
➥todos los espacios están ocupados y 0 en caso contrario. */
template <class T>
int Pila<T>::PilaLlena()
{
   if (Tope == MAX-1)
       return 1;
   else
       return 0;
}

/* Método auxiliar que verifica si la pila está vacía, regresando 1 si
➥lo está y 0 en caso contrario. */
template <class T>
int Pila<T>::PilaVacia()
{
   if (Tope == -1)
       return 1;
   else
       return 0;
}

/* Función que usa la plantilla de la clase Pila. Se almacenan algunos
➥números enteros en un objeto tipo Pila y posteriormente se quita el
➥último guardado y lo imprime. */
void FuncionUsaPila ()
```

```
{
   /* Declaración de un objeto tipo Pila, usando el tipo int para
   ➥instanciar la T. */

   Pila<int> ObjPila;
   int Valor = 0;

   /* Mientras la pila no se llena inserta números en la misma. En este
   ➥ejemplo, es en la aplicación donde se evalúa que haya espacio dispo-
   ➥nible antes de llamar al método que inserta un valor en la pila. */
   while (ObjPila.PilaLlena() != 1)
        ObjPila.Push(Valor++);

   /* Verifica si la pila no está vacía. Si es así, quita el elemento
   ➥almacenado en el Tope (el último insertado) y lo imprime. */
   if (objPila.PilaVacia() != 1)
   {
       ObjPila.Pop(&Valor);
       cout <<Valor<<"\n";
     }
}
```

La estructuración de los métodos presentada en el programa 5.1 tiene la ventaja de independizar los métodos entre sí. Sin embargo, tiene el inconveniente de que deja a cargo del usuario de la clase las validaciones previas a las operaciones de inserción y eliminación, pudiendo ocasionar errores durante la ejecución de los métodos `Push` y `Pop`.

El programa 5.2 retoma la clase `Pila` del programa 5.1, pero ahora utiliza la sobrecarga de operadores. Al operador de suma aritmética se le asocia la operación de inserción de un elemento a la pila, y al operador de resta aritmética se le asocia la operación de eliminación de un elemento de la pila.

Programa 5.2

```
/* Se define la plantilla de la clase Pila usando sobrecarga de
➥operadores en los métodos de inserción y eliminación. */

#define MAX 10

template <class T>
class Pila
```

```
{
   private:
       T EspacioPila[MAX];
       int Tope;
     public:
       Pila();
       void operator + (T);
       void operator - (T *);
       int PilaLlena();
       int PilaVacia();
};

/* Declaración del método constructor por omisión. Asigna el valor -1
➥al Tope, indicando que la pila está vacía. */
template <class T>
Pila<T>::Pila():Tope(-1)
{}

/* Método que evalúa si la pila está llena. Regresa 1 si todos los
➥espacios están ocupados y 0 en caso contrario. */
template <class T>
int Pila<T>::PilaLlena()
{
   if (Tope == MAX-1)
        return 1;
   else
        return 0;
}

/* Método que evalúa si la pila está vacía, regresando 1 si lo está y
➥0 en otro caso. */
template <class T>
int Pila<T>::PilaVacia()
{
   if (Tope == -1)
        return 1;
   else
        return 0;
}

/* El operador +, que normalmente indica la operación aritmética de
➥suma, se sobrecarga utilizándose para insertar un elemento en la pila.
➥Por lo tanto, el operador +, en este programa, tendrá asociadas dos
➥operaciones: suma de números e inserción de elementos en una pila. Se
➥verifica si la pila tiene espacio antes de invocar este método. Se
➥recibe como parámetro el dato a insertar. */
template <class T>
void Pila<T>::operator + (T Valor)
```

```
{
   Tope++;
   EspacioPila[Tope]= Valor;
}

/* El operador -, que normalmente indica la operación aritmética de
➥resta, se sobrecarga utilizándose para eliminar un elemento de la pila.
➥Por lo tanto, el operador -, en este programa, tendrá asociadas dos
➥operaciones: resta de números y eliminación de elementos de una pila.
➥Se verifica que la pila no esté vacía antes de invocar este método. El
➥valor eliminado se pasa como parámetro por referencia. */
template <class T>
void Pila<T>::operator - (T *Valor)
{
   *Valor= EspacioPila[Tope];
   Tope--;
}

/* Función que usa la sobrecarga de operadores definida en la clase
➥Pila. Se declara un objeto tipo Pila de enteros, luego se le insertan
➥MAX elementos y por último, mientras la pila no esté vacía, se quitan y
➥se imprimen cada uno de los valores almacenados en la misma. */
void UsaSobrecargaOperadores()
{
   Pila <int> ObjPila;
   int Indice;

   /* Si la pila está vacía se le agregan MAX elementos, usando el
   ➥operador sobrecargado +. */
   if (ObjPila.PilaVacia())
       for (Indice= 0; Indice < MAX; Indice++)
            ObjPila + Indice*2;

   /* Mientras la pila no se vacíe, se quita un elemento, usando el
   ➥operador - sobrecargado, y se imprime. */
   while  ( !ObjPila.PilaVacia() )
   {
       ObjPila - &Indice;
       cout << '\n' << Indice;
   }
}
```

El programa 5.2 presentó la plantilla de la clase `Pila` y utilizó sobrecarga de operadores para implementar los métodos de inserción y eliminación. Observe que en la función `UsaSobrecargaOperadores()` dichos métodos se invocan mediante los operadores de suma y resta aritmética. Sin embargo, como se aplican a un operando que es un objeto tipo `Pila`, hacen referencia a las operaciones de inserción (y no a la suma aritmética de dos números) y de eliminación (y no a la resta

aritmética entre números) respectivamente. Es importante destacar, que en la aplicación se evalúan las condiciones de pila llena y de pila vacía, antes de invocar a los métodos de inserción y eliminación.

A continuación se presenta un programa de aplicación de pilas. En el programa se incluye una biblioteca con la plantilla de la clase `Pila` correspondiente al programa 5.2.

Programa 5.3

```
/* Se presenta un modelo simplificado de un banco el cual recibe
➥cheques, los registra (almacenándolos temporalmente en pilas), y
➥posteriormente los procesa. Se usan las clases Cheque, ChequeRechazado,
➥Banco y Pila (esta última no se define sino que se incluye en la
➥biblioteca PlanPila.h).  */

#include "PlanPila.h"

/* Definición de la clase Cheque. */
class Cheque
{
   private:
       int Numero, CuentaADepositar;
       char Banco[10];
       double  Monto;
   public:
       Cheque();
       Cheque(int, char[], int, double);
       ~Cheque();
       void ImprimeDatos();
};

/* Declaración del método constructor por omisión. */
Cheque::Cheque()
{}

/* Declaración del método constructor con parámetros. */
Cheque::Cheque(int NumCta, char *NomBco, int Cta, double  Mon)
{
   Numero= NumCta;
   CuentaADepositar= Cta;
   Monto= Mon;
   strcpy(Banco, NomBco);
}

/* Declaración del método destructor. */
Cheque::~Cheque()
{}
```

```
/* Método que despliega en pantalla los valores de todos los atributos
➥de un cheque. */
void Cheque::ImprimeDatos()
{
   cout<< "\nNúmero de cheque: " << Numero;
   cout<< "\nDel banco: " << Banco;
   cout<< "\nDepositado en la cuenta: " << CuentaADepositar;
   cout<< "\nMonto: " << Monto<<endl;
}


/* Definición de la clase ChequeRechazado como clase derivada de la
➥clase Cheque. */
class ChequeRechazado: public Cheque
{
   private:
       double  Cargo;
   public:
       ChequeRechazado();
       ChequeRechazado(int, char[], int, double);
       ~ChequeRechazado();
       void ImprimeDatos();
};

/* Declaración del método constructor por omisión. */
ChequeRechazado::ChequeRechazado()
{}

/* Declaración del método constructor con parámetros. Invoca al
➥constructor de la clase base. */
ChequeRechazado::ChequeRechazado (int NumCta, char *NomBco, int Cta,
                                  ➥double  Mon): Cheque(NumCta, NomBco,
                                  ➥Cta, Mon)
{
   /* Calcula el valor del atributo Cargo como el 10% del Monto del
   ➥cheque. */
   Cargo= Mon*0.10;
}

/* Declaración del método destructor. */
ChequeRechazado::~ChequeRechazado()
{}

/* Método que despliega los valores de los atributos de un cheque
➥rechazado. */
void ChequeRechazado::ImprimeDatos()
{
   Cheque::ImprimeDatos();
   cout<<"\nCargo por rechazo: " << Cargo<<endl;
}
```

```
/* Definición de la clase Banco, la cual tiene dos atributos: uno
➥de ellos representa los cheques, que se almacenan en una pila hasta
➥su procesamiento. El otro atributo son los cheques rechazados, que se
➥almacenan en una pila diferente. Es decir, se usa la plantilla de la
➥clase Pila con las clases Cheque y ChequeRechazado. */
class Banco
{
   private:
       Pila<Cheque> Cheques;
       Pila<ChequeRechazado> ChequesRe;
   public:
       Banco();
       ~Banco();
       void ProcesarCheque();
       void ProcesarChequeR();
       void RegistroCheque(Cheque);
       void RegistroChequeR(ChequeRechazado);
};

/* Declaración del método constructor por omisión. */
Banco::Banco()
{}

/* Declaración del método destructor. */
Banco::~Banco()
{}

/* Método que procesa un cheque: lo quita de la pila de cheques e
➥imprime sus datos. Dado que se usa la plantilla de la clase Pila del
➥programa 5.2, se debe verificar que la pila no esté vacía antes de
➥quitar un cheque. */
void Banco::ProcesarCheque()
{
   Cheque ChequeCli;
   if (!Cheques.PilaVacia())
   {
        Cheques - &ChequeCli;
        cout<< "\n\n\nCheque procesado: ";
        ChequeCli.ImprimeDatos();
   }
   else
        cout<<"\n\nNo hay cheques por procesar.\n\n ";
}

/* Método que procesa un cheque rechazado: lo quita de la pila de
➥cheques rechazados e imprime sus datos. Dado que se usa la plantilla
➥de la clase Pila del programa 5.2, se debe verificar que la pila no
➥esté vacía antes de quitar un cheque rechazado. */
```

```
void Banco::ProcesarChequeR()
{
   ChequeRechazado ChequeCli;
   if (!ChequesRe.PilaVacia())
   {
        ChequesRe - &ChequeCli;
        cout<< "\n\n\nCheque rechazado procesado: ";
        ChequeCli.ImprimeDatos();
   }
   else
        cout<<"\n\nNo hay cheques rechazados por procesar.\n\n ";
}

/* Método que registra un cheque: imprime sus datos y lo almacena en
➥la pila de cheques. Dado que se usa la plantilla de la clase Pila del
➥programa 5.2, se debe verificar que la pila no esté llena antes de
➥insertar un nuevo cheque. */
void Banco::RegistroCheque(Cheque ChequeCli)
{
   if (!Cheques.PilaLlena())
   {
        cout<< "\n\n\nRegistrando el cheque: ";
        ChequeCli.ImprimeDatos();
        Cheques + ChequeCli;
   }
   else
        cout<<"\n\nNo se pudo registrar el cheque por falta de
        ➥espacio. \n\n ";
}

/* Método que registra un cheque rechazado: imprime sus datos y lo
➥almacena en la pila de cheques rechazados. Dado que se usa la plantilla
➥de la clase Pila del programa 5.2, se debe verificar que la pila no
➥esté llena antes de insertar un nuevo cheque rechazado. */
void Banco::RegistroChequeR(ChequeRechazado ChequeCli)
{
   if (!ChequesRe.PilaLlena())
   {
        cout<< "\n\n\nRegistrando el cheque rechazado: ";
        ChequeCli.ImprimeDatos();
        ChequesRe + ChequeCli;
   }
   else
        cout<<"\n\nNo se pudo registrar el cheque rechazado por falta
        ➥de espacio. \n\n ";
}
```

```
/* Función principal. En esta aplicación se crean algunos objetos y
➥se usan para simular algunas operaciones de un banco de manera muy
➥simplificada. */
void main ()
{
   /* Declaración de objetos tipo Cheque y tipo ChequeRechazado, usando
   ➥los constructores con parámetros. */

   Cheque Uno (1718, "Banamex", 14418, 18000.00);
   Cheque Dos (1105, "Bancomer", 13200, 12319.62);

   ChequeRechazado Tres (1816, "Banorte", 12850, 14000.00);
   ChequeRechazado Cuatro (1905, "Bancomer", 13468, 50000.00);

   /* Declaración de un objeto tipo Banco. */
   Banco MiBanco;

   /* Se registran en MiBanco los cheques recibidos, usando la pila
   ➥que les corresponde según si el cheque fue aceptado o rechazado.  */

   MiBanco.RegistroCheque(Uno);
   MiBanco.RegistroCheque(Dos);
   MiBanco.RegistroChequeR(Tres);
   MiBanco.RegistroChequeR(Cuatro);

   /* Se procesan en MiBanco los cheques registrados. Debido a que se
   ➥almacenaron en una pila, se procesan en el orden inverso al que
   ➥fueron registrados. */

   MiBanco.ProcesarCheque();
   MiBanco.ProcesarChequeR();
   MiBanco.ProcesarCheque();
   MiBanco.ProcesarChequeR();

   /* Se intenta procesar otros cheques en MiBanco. Sin embargo ya no habrá
   ➥elementos y los métodos desplegarán un mensaje indicando este caso. */

   MiBanco.ProcesarCheque();
   MiBanco.ProcesarChequeR();
}
```


5.3 Colas

Una ***cola*** es una estructura de datos lineal, es decir una colección de elementos en la cual cada elemento tiene un sucesor y un predecesor únicos, con excepción del primero y del último. El primero no tiene predecesor y el último no tiene sucesor. La estructura cola se caracteriza porque las operaciones de inserción y eli-

minación de elementos deben hacerse por extremos diferentes. Los elementos se insertan por uno de los extremos y se eliminan por el otro extremo. Por lo tanto, el primer elemento insertado será el primero que podrá eliminarse; a esta estructura también se le conoce con el nombre de estructura ***FIFO*** (por sus siglas del inglés: ***F***irst-***I***n, ***F***irst-***O***ut: primero en entrar, primero en salir).

En una estructura tipo cola se identifican los dos extremos por donde se realizarán las operaciones. El ***frente*** o principio de la cola será el extremo en el cual se eliminarán elementos, mientras que el ***final*** será el extremo en el cual se harán las inserciones. La figura 5.5 presenta un esquema de una estructura tipo cola, en la que se insertaron 5 datos, estando el `frente` en la posición 0 y el `final` en la 4.

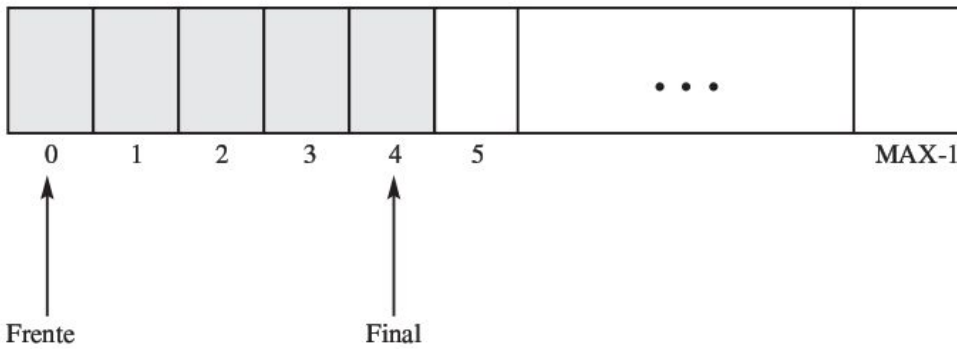


FIGURA 5.5 *Estructura tipo cola*

El concepto de cola se usa en muchas actividades cotidianas, por ejemplo cuando un grupo de personas se forma frente a la taquilla de un cine, la primera que llegó será la primera en ser atendida. Otro ejemplo es la cola de automovilistas frente a un semáforo en rojo, el primero en llegar al cruce de calle será el primero en pasar cuando la luz cambie a verde.

La cola es una estructura abstracta. Para el almacenamiento de los datos en la memoria de la computadora debe usarse otra estructura. Para los efectos de este libro, se utilizarán arreglos unidimensionales. Por lo tanto, la clase `Cola` tendrá como atributos la colección de elementos (por medio de un arreglo unidimensional) y los apuntadores al primero y último valores. Además, tendrá algunos métodos que se analizarán en la siguiente sección. La figura 5.6 presenta una plantilla de la clase `Cola`. Se definió una plantilla para dar mayor generalidad a la solución.

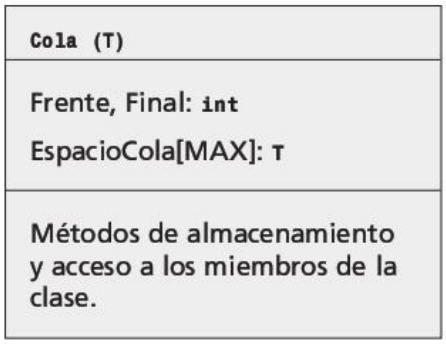


FIGURA 5.6 *Clase* `Cola`

A continuación se presenta la codificación de la plantilla de la clase `Cola`, usando el lenguaje ***C++***.

```
/* Definición del número máximo de elementos que puede contener la cola,
➥restricción propia de los arreglos. */
#define MAX 10

/* Definición de la plantilla de la clase Cola. La misma tendrá como
➥atributos la colección de elementos (haciendo uso de un arreglo) y
➥apuntadores al primero y al último de ellos. En la plantilla también se
➥hace referencia a algunos métodos, los cuales se analizarán con detalle
➥en la siguiente sección. */
template <class T>
class Cola
{
   private:
       T EspacioCola[MAX];
   int Frente, Final;
   public:
       Cola();
       /* En esta sección se declaran los métodos de modificación y
       ➥acceso a los miembros de la clase, los cuales se analizan en la
       ➥siguiente sección. */
};

/* Declaración del método constructor. Inicializa el Frente y Final en
➥-1, indicando cola vacía. */
template <class T>
Cola<T>::Cola()
{
     Frente= -1;
     Final= -1;
}
```

Operaciones

Como ya se mencionó, en una estructura tipo cola se insertan y eliminan elementos por extremos diferentes. Se agregan elementos por uno de los extremos, el cual se conoce con el nombre de `Final` y se quitan por el otro extremo, llamado `Frente`. Esta particularidad es lo que determina la manera en que se llevan a cabo las operaciones de inserción y eliminación.

La operación de inserción consiste en incrementar el puntero al final de la cola y agregar el nuevo valor en dicha posición. Antes de llevarse a cabo la operación resulta necesario verificar que en la cola haya espacio disponible. La manera de evaluar esta condición depende del tipo de estructura elegida para almacenar la colección de elementos en la cola. Considerando que en este libro se usará un arreglo unidimensional, la cola estará llena cuando el puntero al final tenga un valor igual a MAX-1. La figura 5.7 presenta gráficamente esta operación. Inicialmente, en la cola hay almacenados 5 elementos, que ocupan las casillas de la 0 a la 4. Posteriormente se incrementó el `Final` (ahora está apuntado a la casilla 5) y se asignó en esa posición el nuevo valor.

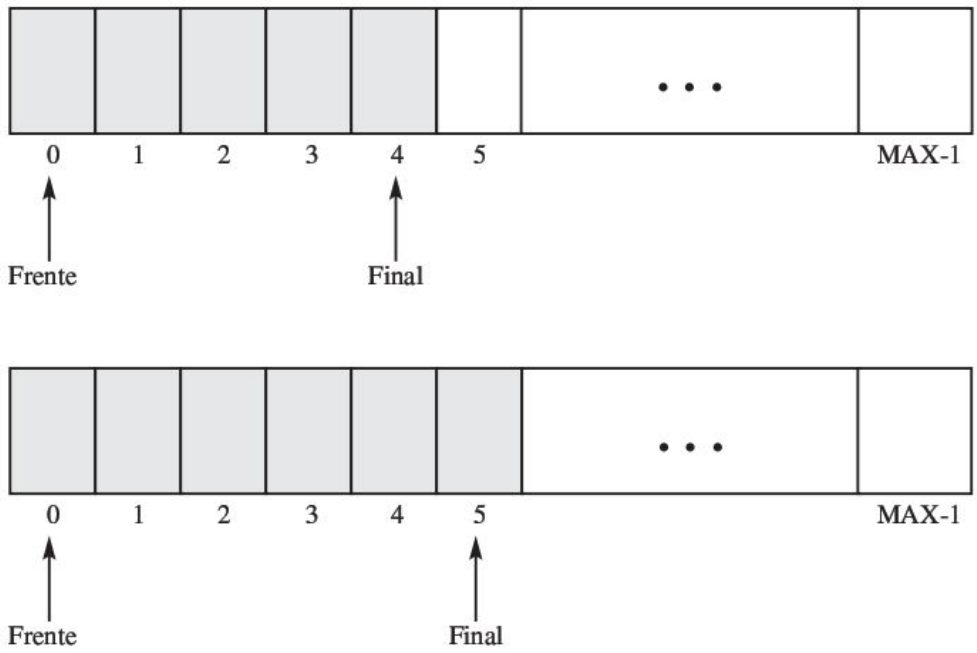


FIGURA 5.7 *Operación de Inserción en Colas*

A continuación se presenta la plantilla del método correspondiente a la operación de inserción.

```
/* Método que inserta un valor en la cola. La inserción se lleva a cabo
➥por el extremo identificado como Final. Antes de invocar el método se
➥debe validar que la cola tenga espacio disponible. El valor a insertar
➥se recibe como parámetro. Cuando la cola está vacía y se inserta un
➥dato, entonces también se debe actualizar el puntero al frente de la
➥cola. */
template <class T>
void Cola<T>::InsertaCola(T Dato)
{
   EspacioCola[++Final]= Dato;
   if (Final == 0)
       Frente= 0;
}
```

En esta implementación se asume que la aplicación se encarga de verificar el estado de la cola. Sin embargo, es posible realizar esta evaluación dentro del método, tal como se muestra en el programa 5.5.

La operación de eliminación consiste en asignar a una variable de trabajo el valor almacenado en la casilla indicada por el puntero *Frente* y desplazar a éste (incrementar su valor) apuntando al siguiente elemento. Antes de llevarse a cabo la operación, resulta necesario verificar que la cola no esté vacía. La manera de evaluar esta condición depende del tipo de estructura elegida para almacenar la colección de elementos en la cola. Considerando que en este libro se usará un arreglo unidimensional, la cola estará vacía cuando el puntero al frente tenga un valor de –1. La figura 5.8 presenta de manera gráfica esta operación. Inicialmente, en la cola hay almacenados 5 elementos, que ocupan las casillas de la 0 a la 4. Después, se asignó el valor guardado en la casilla 0 (apuntada por el `Frente`) en una variable auxiliar y se modificó el valor de `Frente`, desplazándose hacia la siguiente casilla.

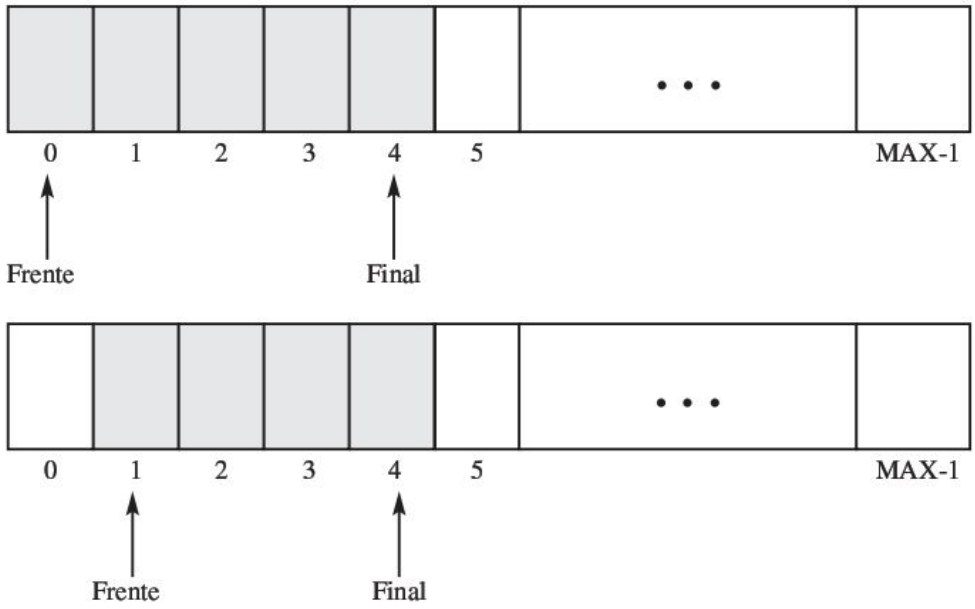


FIGURA 5.8 *Operación de eliminación en colas*

A continuación se presenta la plantilla del método correspondiente a la operación de eliminación de elementos de una cola.

```
/* Método que elimina un elemento de la cola. La eliminación se lleva a
➥cabo por el extremo identificado como Frente. Antes de invocar el método
➥se debe validar que la cola no esté vacía. El valor eliminado se regresa
➥como un parámetro por referencia. Si en la cola hubiera sólo un elemento,
➥entonces luego de quitarlo se deben poner los dos punteros en -1 para
➥indicar que la cola quedó vacía. */
template <class T>
void Cola<T>::EliminaCola(T *Dato)
{
   *Dato= EspacioCola[Frente];
   if (Frente == Final)
   {
       Frente= -1;
       Final= -1;
   }
   else
       Frente++;
}
```

Es importante destacar que al invocar el método de eliminación, siempre se quita el elemento que está en el frente de la cola (fue el primero que se almacenó en ella).

El programa 5.4 presenta la plantilla de la clase *Cola*, con todos sus atributos y métodos. Además, incluye una aplicación muy sencilla.

Programa 5.4

```
/* Se define la plantilla de la clase Cola. Además, se incluye un
➥ejemplo muy simple de aplicación de esta clase. */

/* Definición del número máximo de elementos que puede contener la cola,
➥restricción que surge de usar un arreglo unidimensional. */
#define MAX 10

/* Definición de la plantilla de la clase Cola. */
template <class T>
class Cola
{
   private:
       T EspacioCola[MAX];
       int Frente, Final;
   public:
       Cola();
       void InsertaCola(T);
       void EliminaCola(T*);
       int ColaLlena();
       int ColaVacia();
};

/* Declaración del método constructor. Inicializa los punteros en -1,
➥indicando que la cola está vacía. */
template <class T>
Cola<T>::Cola()
{
   Frente= -1;
   Final= -1;
}

/* Método que inserta un valor en la cola. La inserción se lleva a cabo
➥por el extremo identificado como Final. Antes de invocar el método se
➥debe verificar que la cola tenga espacio disponible. El método recibe
➥como parámetro el valor a insertar. */
template <class T>
void Cola<T>::InsertaCola(T Dato)
```

```
{
   EspacioCola[++Final]= Dato;
   if (Final == 0)
       Frente= 0;
}

/* Método que elimina un elemento de la cola. La eliminación se lleva a
➥cabo por el extremo identificado como Frente. Antes de invocar el método
➥se debe verificar que la cola no esté vacía. El valor eliminado se
➥regresa por medio de un parámetro por referencia. */
template <class T>
void Cola<T>::EliminaCola(T *Dato)
{
   *Dato= EspacioCola[Frente];
   if (Frente == Final)
   {
       Frente= -1;
       Final= -1;
   }
   else
       Frente++;
}

/* Método auxiliar que verifica si la cola está llena. Regresa 1 si la
➥cola no tiene espacio disponible y 0 en caso contrario. */
template <class T>
int Cola<T>::ColaLlena()
{
   if (Final == MAX-1)
       return 1;
   else
       return 0;
}

/* Método auxiliar que verifica si la cola está vacía. Regresa 1 si la
➥cola no tiene ningún elemento y 0 en caso contrario. */
template <class T>
int Cola<T>::ColaVacia()
{
   if (Frente == -1)
       return 1;
   else
       return 0;
}


/* Función que hace uso de la plantilla de la clase Cola. Se declara un
➥objeto tipo Cola de números enteros. Se le inserta el número 5 y luego
➥se quita y se imprime. Este ejemplo sencillo muestra el uso de los
➥métodos de la clase. */
```

```
void UsaClaseCola ()
{
     Cola<int> ObjCola;
     int Indice;

     if (ObjCola.ColaLlena() != 1)
            ObjCola.InsertaCola(5);
     else
         cout<<"\nError de desbordamiento. Cola llena. \n";

   if (ObjCola.ColaVacia() != 1)
   {
       ObjCola.EliminaCola(&Valor);
       cout<<Valor<<"\n";
   }
   else
       cout<<"\nSubdesbordamiento. Cola vacía.\n";
}
```

Como puede observar, en la función de aplicación se valida que haya espacio y que la cola no esté vacía antes de invocar a los métodos de inserción y eliminación respectivamente. Sin embargo, los métodos se pueden estructurar de manera diferente, haciendo que ambos incluyan la validación (invocación de los métodos auxiliares `ColaLlena()` y `ColaVacia()`) dentro de su mismo código. El programa 5.5 presenta la plantilla de la clase `Cola`, con los métodos estructurados de esta forma. Además, utiliza sobrecarga de operadores.

Programa 5.5

```
/* Definición del número máximo de elementos que puede contener la cola,
➥restricción que surge de usar un arreglo unidimensional. */
#define MAX 10

/* Definición de la plantilla de la clase Cola. Se utiliza sobrecarga de
➥operadores y se define a los métodos de inserción y eliminación como
➥métodos enteros. */
```

```
template <class T>
class Cola
{
   private:
       T EspacioCola[MAX];
       int Frente, Final;
   public:
       Cola();
       int operator + (T);
       int operator - (T*);
       int ColaLlena();
       int ColaVacia();
};

/* Declaración del método constructor. Inicializa los punteros en -1,
➥indicando que la cola está vacía. */
template <class T>
Cola<T>::Cola()
{
   Frente= -1;
   Final= -1;
}

/* Método que inserta un valor en la cola. La inserción se lleva a cabo por
➥el extremo identificado como Final. Antes de llevar a cabo la inser-
➥ción se verifica que la cola tenga espacio disponible. Si la operación
➥concluye con éxito el método regresa un 1, en caso contrario un 0.*/
template <class T>
int Cola<T>::operator + (T Dato)
{
   /* La variable Res se inicializa en 0 (fracaso). Si la inserción se
   ➥lleva a cabo, entonces se le asignará el valor de 1 (éxito). */
   int Res= 0;
   if (ColaLlena() != 1)
   {
        EspacioCola[++Final]= Dato;
        if (Final == 0)
            Frente= 0;
        Res= 1;
   }
   return  Res;
}

/* Método que elimina un elemento de la cola. La eliminación se lleva
➥a cabo por el extremo identificado como Frente. Antes de quitar el
➥elemento se debe validar que la cola no esté vacía. El valor eliminado
➥se regresa por medio de un parámetro por referencia. Si la operación
➥concluye con éxito el método regresa un 1, en caso contrario un 0. */
```

```
template <class T>
int Cola<T>::operator - (T *Dato)
{
   /* La variable Res se inicializa en 0 (fracaso). Si la eliminación se
   ➥lleva a cabo, entonces se le asignará el valor de 1 (éxito). */
   int Res= 0;
   if (!ColaVacia())
   {
        *Dato= EspacioCola[Frente];
        if (Frente == Final)
        {
            Frente= -1;
            Final= -1;
        }
        else
             Frente++;
        Res= 1;
   }
   return  Res;
}

/* Método auxiliar que verifica si la cola está llena. Regresa 1 si la
➥cola no tiene espacio disponible y 0 en caso contrario. */
template <class T>
int Cola<T>::ColaLlena()
{
   if (Final == MAX-1)
        return  1;
   else
        return  0;
}

/* Método auxiliar que verifica si la cola está vacía. Regresa 1 si la
➥cola no tiene ningún elemento y 0 en caso contrario. */
template <class T>
int Cola<T>::ColaVacia()
{
   if (Frente == -1)
        return  1;
   else
        return  0;
}
```

La estructura que se le dio a los métodos tiene la ventaja de que cada método es responsable de verificar los posibles casos de error, garantizando de esta manera el buen uso de la estructura de datos. Los usuarios de la clase pueden analizar el resultado que dan los métodos para tomar una decisión adecuada a cada caso.

El programa 5.6 presenta un ejemplo de aplicación de la estructura cola. Se utiliza una estructura de este tipo para almacenar los datos de algunos productos que se tienen para la venta. La política de ventas es que siempre se vende el producto que tiene más tiempo en el depósito, es decir el primero que se compró. Se usa la plantilla definida en el programa 5.5, la cual está en la biblioteca "`Cola.h`".

Programa 5.6

```
/* Aplicación de una estructura de datos tipo cola. Se define la clase
➥Producto y una cola de objetos tipo Producto. Considerando que se
➥quieren vender los productos de acuerdo al orden en el que fueron
➥comprados, se usó una cola para almacenarlos. La cola se actualiza a
➥medida que se compran o se venden productos. */

#include "Cola.h"

/* Definición de la clase Producto. */
class Producto
{
   private:
       int Clave;
       char NomProd[64];
       double Precio;
   public:
       Producto();
       Producto(int, char[], double);
       double RegresaPrecio();
       friend  istream &operator>>(istream &, Producto &);
       friend  ostream &operator<<(ostream &, Producto &);
};

/* Declaración del método constructor por omisión. */
Producto::Producto()
{}

/* Declaración del método constructor con parámetros. */
Producto::Producto(int Cla, char  NomP[], double Pe)
{
   Clave= Cla;
   strcpy(NomProd, NomP);
   Precio= Pre;
}
```

```
/* Método que permite, a usuarios externos a la clase, conocer el valor
➥del atributo privado Precio. */
double Producto::RegresaPrecio()
{
   return Precio;
}

/* Sobrecarga del operador >>. De esta forma se permite leer objetos de
➥tipo Producto de manera directa . */
istream &operator>>(istream &Lee,  Producto &ObjProd)
{
   cout<<"\n\nIngrese clave del producto:  ";
   Lee>>ObjProd.Clave;
   cout<<"\n\nIngrese nombre del producto:  ";
   Lee>>ObjProd.NomProd;
   cout<<"\n\nIngrese precio:  ";
   Lee>>ObjProd.Precio;
   return  Lee;
}

/* Sobrecarga del operador <<. De esta forma se permite imprimir objetos
➥de tipo Producto de manera directa . */
ostream &operator<< (ostream &Escribe, Producto &ObjProd)
{
   Escribe<<"\n\nDatos del producto\n";
   Escribe<<"\nClave:  "<<ObjProd.Clave;
   Escribe<<"\nNombre:  "<<ObjProd.NomProd;
   Escribe<<"\nPrecio:  "<<ObjProd.Precio;
   return  Escribe;
}

/* Función auxiliar que despliega al usuario las opciones de trabajo. En
➥este caso registrar la compra o la venta de un producto. */
int Menu()
{
   int Resp;
   do {
       cout<<"\n\nIngrese operación a registrar:  ";
       cout<<"\n1- Compra de un producto  ";
       cout<<"\n2- Venta de un producto\n  ";
       cout<<"\n3- Termina el registro\n  ";
       cin>>Resp;
   } while (Resp != 1 &&  Resp != 2 &&  Resp != 3);
   return Resp;
}
```

```
/* Función principal que lleva a cabo la aplicación descrita. Se crea
➥una cola de objetos tipo Producto y la misma se va modificando a medida
➥que se compran o se venden productos. El uso de una cola para guardar
➥los productos facilita el cumplimiento de la condición impuesta para su
➥venta: el primero que se compre será el primero que se venda. Al final
➥de las transacciones se imprime el total de dinero recaudado con las
➥ventas. */
void main()
{
   Cola<Producto> Deposito;
   Producto Prod;
   int Opc;
   double Total= 0;

   Opc= Menu();
   while (Opc == 1 || Opc== 2)
   {
       switch (Opc)
       {
              case 1: cin>>Prod;
                      if (!(Deposito + Prod))
                            cout<<"\n\nNo hay lugar en el depósito para
                                  ➥registrar el producto.\n";
                      break;
              case 2: if (Deposito - &Prod)
                      {
                           cout<<Prod;
                           Total= Total + Prod.RegresaPrecio();
                      }
                      else
                           cout<<"\n\n\nYa no hay productos en el
                           ➥depósito.\n\n";
                break;
       }
       Opc= Menu();
   }
   cout<<"\n\n\nTotal vendido: "<<Total<<endl;
}
```

5.3.1 Colas circulares

Las estructuras tipo cola que se han estudiado hasta el momento resultan ineficientes en cuanto al manejo del espacio de memoria, si se efectúan sobre ellas muchas actualizaciones. Es decir, si se realizan muchas inserciones y eliminaciones puede darse el caso que el estado de la cola sea *cola llena*, no permitiendo nuevas inserciones cuando en realidad se dispone de muchos espacios vacíos. La

figura 5.9 presenta el caso de que el puntero al final de la cola está en la última posición del arreglo, lo cual producirá que se detecte que no hay espacio disponible. Sin embargo, como se ilustra en la figura, el puntero al frente de la cola está desplazado hacia la derecha (consecuencia de haber hecho varias eliminaciones) por lo que hay espacio físico disponible en la estructura.

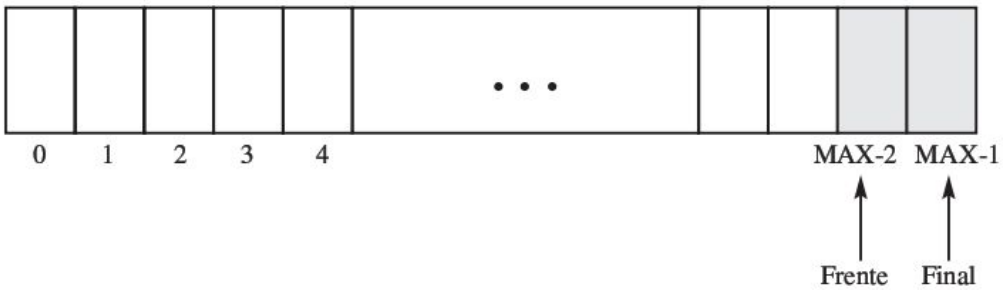


FIGURA 5.9 *Cola "llena"*

Una ***cola circular*** es aquella en la cual el sucesor del último elemento es el primero. Por lo tanto, el manejo de las colas como estructuras circulares permite un mejor uso del espacio de memoria reservado para la implementación de las mismas. La figura 5.10 corresponde a la representación gráfica de una cola circular. Observe que el siguiente elemento del último es el primero.

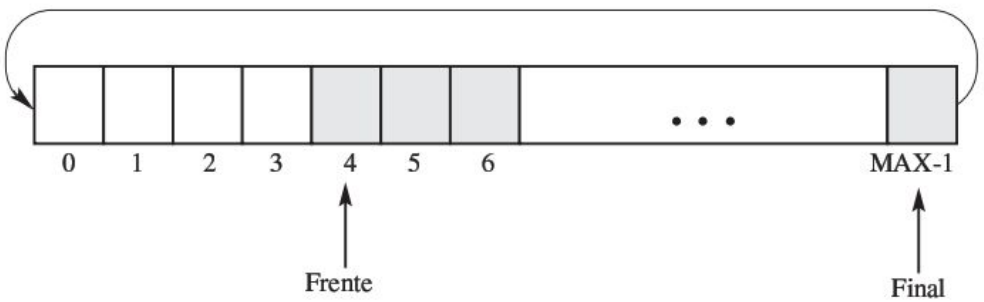


FIGURA 5.10 *Estructura de una cola circular*

La figura 5.11 presenta el esquema correspondiente a una cola circular, en la cual el final se movió hacia el inicio de la cola, teniendo un valor menor al frente. En este ejemplo, la cola tiene las posiciones 4 a MAX-1 y 0 a 1 ocupadas, siendo el primer elemento a salir el que está en la posición 4 y el último insertado el que está en la posición 1.

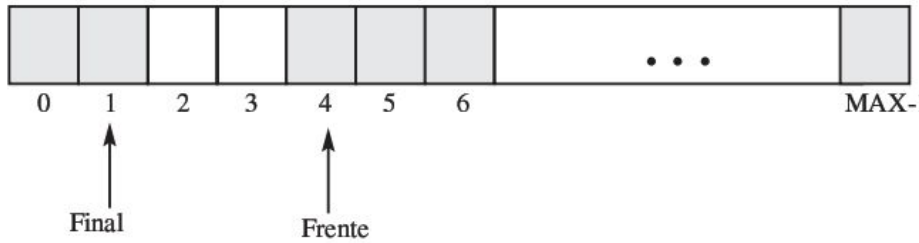


FIGURA 5.11 *Cola circular*

Los algoritmos correspondientes a las operaciones de inserción y eliminación varían al tratarse de colas circulares, en lo referente a la actualización de los punteros. Asimismo, la condición para determinar si la cola está llena debe considerar todos los casos que puedan presentarse, que son:

1. el `Frente` en la posición 0 y el `Final` en la posición (`MAX - 1`), o
2. el (`Final + 1`) es igual al `Frente`

ambos se evalúan por medio de la expresión: `(Final+1) % MAX == Frente`. A continuación, el programa 5.7 presenta la plantilla completa de la estructura cola circular.

Programa 5.7

```
/* Definición del número máximo de elementos que puede contener la cola
➥circular, por estar implementada con un arreglo unidimensional. */
#define MAX 10

/* Se define la plantilla de la clase ColaCircular. */
template <class T>
class ColaCircular
{
   private:
       T EspacioCola[MAX];
       int Frente, Final;
   public:
       ColaCircular();
       int InsertaCCircular(T);
       int EliminaCCircular(T *);
       int ColaCircularLlena();
       int ColaCircularVacia();
};

/* Declaración del método constructor. Inicializa los punteros en -1
➥indicando que la cola circular está vacía. */
```

```
template <class T>
ColaCircular<T>::ColaCircular()
{
   Frente= -1;
   Final= -1;
}

/* Método entero que introduce un dato en la cola circular. El método
➥recibe como parámetro el valor a insertar. El método da como resultado
➥el valor 1 si la inserción se lleva a cabo y 0 en caso contrario. */
template <class T>
int ColaCircular<T>::InsertaCCircular(T Dato)
{
   int Resp= 0;
   if (!ColaCircularLlena())
   {
        Resp= 1;
        Final= (Final +1) % MAX;
        EspacioCola[Final]= Dato;
        if (Frente == -1)
             Frente= 0;
   }
   return Resp;
}

/* Método entero que elimina un dato de la cola circular. El método
➥regresa el valor eliminado por medio de un parámetro por referencia.
➥Da como resultado el valor 1 si realiza la eliminación y 0 en caso
➥contrario. */
template <class T>
int ColaCircular<T>::EliminaCCircular(T *Dato)
{
   int Resp= 0;
   if  (!ColaCircularVacia())
   {
        Resp= 1;
        *Dato= EspacioCola[Frente];
        if (Frente == Final)
        {
             Frente= -1;
             Final= -1;
        }
        else
             Frente= (Frente + 1) % MAX;
   }
   return Resp;
}
```

```
/* Método auxiliar que verifica si la cola circular está llena. Regresa 1
➥si la cola no tiene espacios disponibles y 0 en caso contrario. */
template <class T>
int ColaCircular<T>::ColaCircularLlena()
{
   if ((Final + 1) % MAX == Frente)
        return 1;
   else
        return 0;
}

/* Método auxiliar que verifica si la cola circular está vacía. Regresa 1
➥si la cola no tiene ningún elemento almacenado y 0 en caso contrario. */
template <class T>
int ColaCircular<T>::ColaCircularVacia()
{
   if (Frente == -1)
        return 1;
   else
        return 0;
}
```

El programa 5.8 presenta un ejemplo de aplicación de colas circulares. Observe que para el usuario de la clase es totalmente indistinta la manera en la que esté implementada la cola. La ventaja está en el mejor aprovechamiento del espacio de memoria. Sin embargo, el uso es igual al de las colas vistas al inicio de esta sección. Se utiliza la plantilla de la cola circular correspondiente al programa 5.7, la cual se incluye en la biblioteca "*ColaCircular.h*".

Programa 5.8

```
/* Aplicación de una cola circular. Los datos de un grupo de pacientes
➥(objetos de tipo Paciente) se registran en una cola a medida que los
➥pacientes solicitan atención de un médico. A su vez, cuando un médico
➥se desocupa y está en condiciones de recibir a otro paciente se obtienen
➥(de la cola) los datos de un paciente y se le asigna a dicho médico.
➥De esta manera se garantiza que los pacientes sean atendidos en el
➥orden en el que fueron registrados. */

/* En la biblioteca "ColaCircular.h" se incluye la plantilla de la clase
ColaCircular presentada en el programa 5.7. */
```

```
#include "ColaCircular.h"


/* Definición de la clase Paciente. */
class Paciente
{
   private:
       char Nombre[64], Sexo, Padecim[64];
       int AnioNac;
   public:
       Paciente();
       Paciente(char[], char, char[], int);
       friend  istream &operator>>(istream &, Paciente &);
       friend  ostream &operator<<(ostream &, Paciente &);
};

/* Declaración del método constructor por omisión. */
Paciente::Paciente()
{}

/* Declaración del método constructor con parámetros. */
Paciente::Paciente(char Nom[], char S, char Padec[], int ANac)
{
   strcpy(Nombre, Nom);
   Sexo= S;
   strcpy(Padecim, Padec);
   AnioNac= ANac;
}

/* Sobrecarga del operador >> para poder leer objetos de tipo Paciente
➥de manera directa. */
istream &operator>>(istream &Lee,  Paciente &ObjPac)
{
   cout<<"\n\nIngrese nombre del paciente:  ";
   Lee>>ObjPac.Nombre;
   cout<<"\n\nSexo: ";
   Lee>>ObjPac.Sexo;
   cout<<"\n\nPadecimiento del paciente:  ";
   Lee>>ObjPac.Padecim;
   cout<<"\n\nIAño de nacimiento:  ";
   Lee>>ObjPac.AnioNac;
   return  Lee;
}

/* Sobrecarga del operador << para poder desplegar en pantalla objetos
➥de tipo Paciente de manera directa. */
```

```
ostream &operator<< (ostream &Escribe, Paciente &ObjPac)
{
   Escribe<<"\n\nDatos del paciente\n";
   Escribe<<"\nNombre: "<<ObjPac.Nombre;
   Escribe<<"\nSexo:  "<<ObjPac.Sexo;
   Escribe<<"\nAño nacimiento:  "<<ObjPac.AnioNac;
   Escribe<<"\nPadecimiento:  "<<ObjPac.Padecim;
   return  Escribe;
}


/* Función auxiliar que despliega al usuario las opciones de trabajo: regis-
➥trar un nuevo paciente o asignar médico a un paciente ya registrado. */
int Menu()
{
   int Opc;
   do {
       cout<<"\n\nBienvenido al sistema de registro de pacientes para
              ➥consulta. \n\n";
       cout<<"\n¿Qué desea hacer?\n";
       cout<<"\n 1-Registrar un nuevo paciente. ";
       cout<<"\n 2-Asignar médico a un paciente.";
       cout<<"\n 3-Terminar.\n";
       cout<<"\n\nIngrese la opción elegida:  ";
       cin>>Opc;
   } while (Opc < 1 || Opc > 3);
   return Opc;
}

/* Función que hace uso de la cola circular para almacenar los datos a
➥procesar. Por medio de la cola se asegura que los datos se procesen en
➥el orden en el que llegan: los pacientes se asignan a los médicos en el
➥orden en el que llegaron a la consulta. */
void UsaColaCircular()
{
   ColaCircular<Paciente> ListaEspera;
   Paciente Pac;
   int Opc= Menu();

   while (Opc == 1 || Opc== 2)
   {
       switch (Opc)
       {
          case 1: cin>>Pac;
                  if (!ListaEspera.InsertaCCircular(Pac))
```

```
                    cout<<"\nLa cuota de pacientes se agotó. Regrese
                        ➥mañana.";
                break;
            case 2: if (ListaEspera.EliminaCCircular(&Pac))
                    cout<<"\n\nEl paciente que pasa a consulta es:
                        ➥"<<Pac;
                else
                    cout<<"\n\n\nNo hay pacientes en espera de ser
                        ➥atendidos.\n\n";
                break;
        }
        Opc= Menu();
    }
}
```

5.3.2 Colas dobles

Otra variante de las estructuras tipo cola son las ***colas dobles***. Como su nombre lo indica, estas estructuras permiten realizar las operaciones de inserción y eliminación por cualquiera de sus extremos. Gráficamente una cola doble se representa de la siguiente manera:

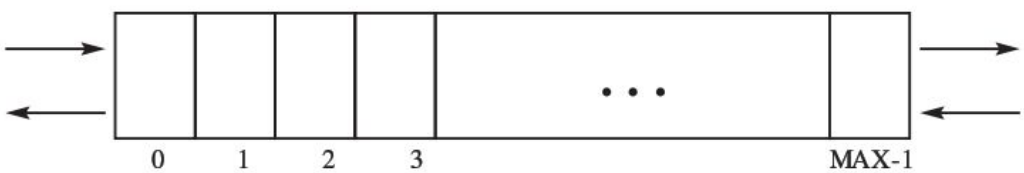


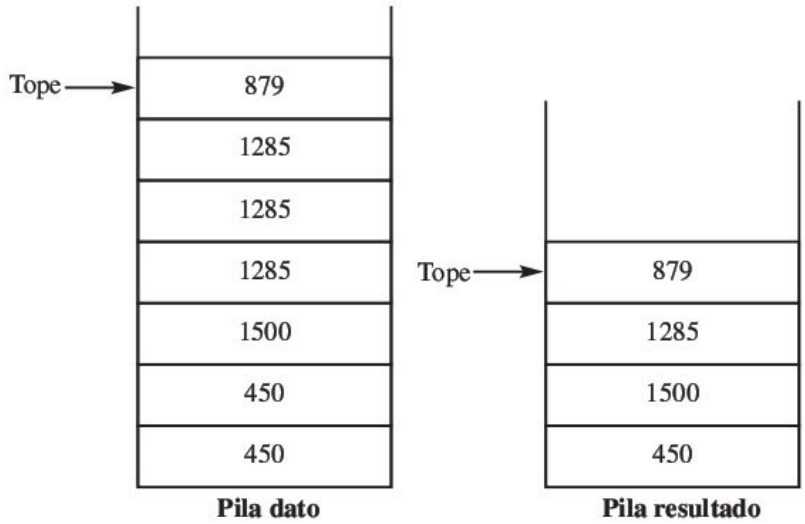
FIGURA 5.12 *Cola doble*

Debido a que este tipo de estructura es una generalización del tipo cola no se presentan aquí las operaciones. Al respecto, sólo se menciona que será necesario definir métodos que permitan insertar por el frente y por el final, así como métodos que permitan eliminar por ambos extremos. Las condiciones para determinar el estado de la cola no varían.

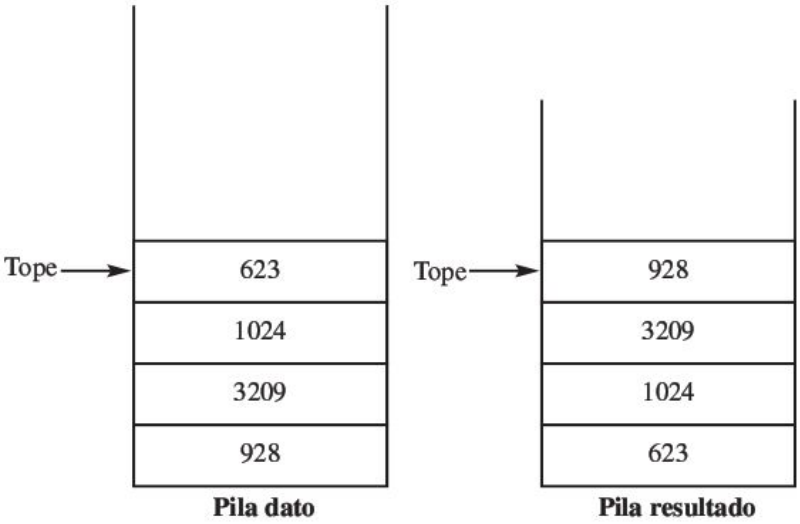
Por último, es importante señalar que una doble cola también puede ser circular. En dicho caso, será necesario que los métodos de inserción y eliminación (sobre cualquiera de los extremos) consideren el movimiento adecuado de los punteros.

Ejercicios

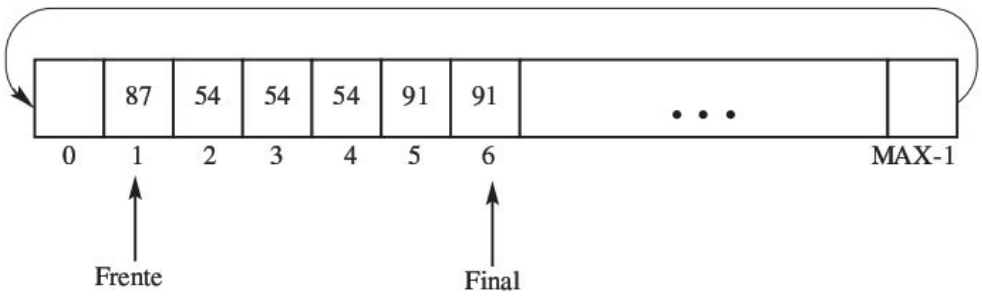
1. Escriba un programa en ***C++*** que, apoyándose en un objeto tipo pila, traduzca una expresión dada en notación infija a notación prefija. Por ejemplo, si la expresión dada es *a* + *b*, su programa debe generar como salida + *a b*.
2. Escriba un programa en ***C++*** que, apoyándose en un objeto tipo pila, evalúe una expresión dada en notación prefija. Por ejemplo, si la expresión dada es + *8 4*, su programa debe generar como salida el valor *12*. Puede asumir que se darán números de un solo dígito.
3. Escriba un programa en ***C++*** que elimine los elementos repetidos de una pila. Suponga que si existen elementos repetidos, los mismos se encuentran en posiciones consecutivas. Puede usar cualquier estructura de datos como auxiliar. Observe la siguiente figura:



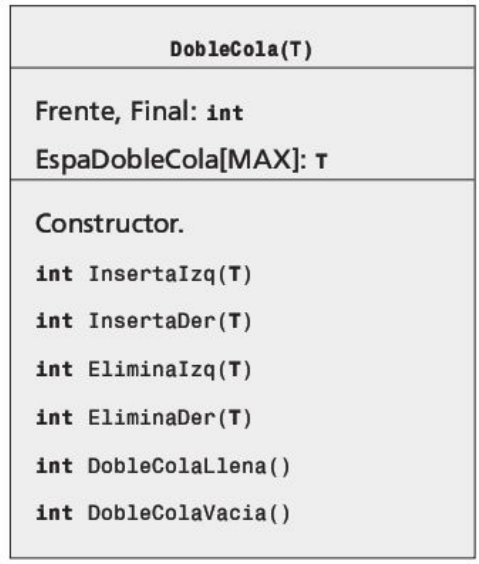
4. Escriba un programa en ***C++*** que invierta los elementos almacenados en una pila. Puede usar cualquier estructura de datos como auxiliar. Observe la siguiente figura:



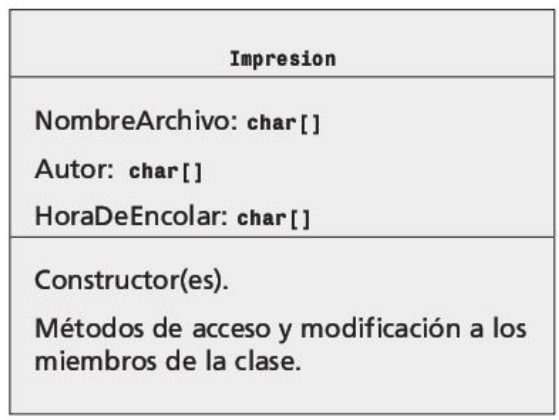
5. Defina la clase `Cola` y utilice sobrecarga de operadores en los métodos necesarios para implementar las operaciones asociadas a este tipo de estructura. Se sugiere usar la suma aritmética (+) para la inserción y la resta aritmética (–) para la eliminación.
6. Escriba un programa en ***C++*** que invierta iterativamente los elementos de una cola. Puede usar cualquier estructura de datos como auxiliar.
7. Escriba un programa en ***C++*** que invierta recursivamente los elementos de una cola. Puede usar cualquier estructura de datos como auxiliar.
8. Escriba un programa en ***C++*** que elimine los elementos repetidos de una cola circular. Suponga que si existen elementos repetidos, los mismos se encuentran en posiciones consecutivas (ver la figura). Puede usar cualquier estructura de datos como auxiliar.



9. Defina la plantilla de la clase `DobleCola`, de acuerdo a las especificaciones que se dan a continuación.



10. Escriba un programa en ***C++***, que mediante la plantilla de la cola circular presentada en este capítulo, simule el comportamiento de una cola de impresión. La cola deberá almacenar objetos de la clase `Impresión`, cuyas especificaciones se dan a continuación. El programa leerá dos posibles opciones de trabajo sobre la cola de impresión: a) encolar un nuevo archivo a imprimir o b) imprimir un archivo. Su programa debe verificar que las opciones dadas por el usuario puedan realizarse, en caso contrario desplegará un mensaje adecuado.



11. Escriba un programa en ***C++***, que mediante la plantilla de la cola circular presentada en este capítulo, simule el comportamiento de una cola de atención a clientes de un banco. La cola deberá almacenar objetos de la clase `Clientes`. Defina qué atributos y métodos deberá incluir esta clase. El programa leerá, mientras el usuario así lo requiera, dos posibles opciones de trabajo sobre la cola de espera de los clientes:

    a) Llega un nuevo cliente al banco, en cuyo caso debe ingresarse a la cola de espera. El usuario proporcionará los datos del cliente.

    b) Un cliente pasa a la ventanilla donde será atendido. El dato será el número de ventanilla a la que debe pasar.

Al finalizar el día de trabajo, su programa debe imprimir el total de clientes atendidos.

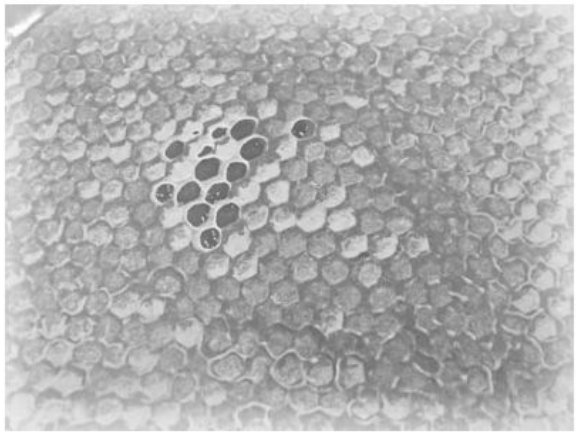
12. En la Dirección escolar de una escuela se reciben solicitudes de constancias de estudio de los alumnos. Cada constancia lleva el *nombre del alumno*, *nombre de la carrera que cursa*, *total de materias aprobadas* y *promedio general*. Escriba un programa en ***C++*** que, apoyándose en un objeto tipo cola, pueda realizar las siguientes operaciones:

    a) Dar de alta la solicitud de un alumno (la solicitud debe encolarse, ya que se atenderá según el orden en el cual se recibió).

    b) Elaborar una constancia. La misma debe tener todos los datos mencionados. Esta operación presupone que los datos del alumno cuya solicitud es atendida deben quitarse de la cola.

La cola almacenará objetos tipo `Alumno` y tomará los datos de dichos objetos para la elaboración de la constancia. Defina qué atributos y métodos tendrá la clase mencionada. Utilice alguna de las plantillas de la clase `Cola` explicadas en este capítulo.

13. Retome el problema anterior. Escriba un programa en ***C++*** que permita eliminar de la cola de espera de la Dirección escolar a todos aquellos alumnos cuya carrera sea igual a un cierto valor dado por el usuario.



CAPÍTULO 6

Listas

6.1 Introducción

Este capítulo presenta la estructura de datos conocida como lista y muestra las principales características, cómo se relacionan sus componentes y analiza las operaciones que se le pueden aplicar.

En términos generales, una ***lista*** se define como una colección de elementos donde cada uno de ellos, además de almacenar información, almacena la dirección del siguiente elemento. Una lista es una estructura lineal de datos. Es decir, cada uno de sus componentes tiene un sucesor y predecesor únicos, con excepción del último y del primero, los cuales carecen de sucesor y de predecesor respectivamente.

Las listas pueden implementarse mediante arreglos resultando así una estructura estática (el tamaño de la misma no varía durante la ejecución del programa). Otra alternativa para su implementación es usar memoria

dinámica, lo que permite que dicha característica se propague a la lista, obteniendo una estructura dinámica (la cantidad de memoria ocupada puede modificarse durante la ejecución del programa). Las listas se analizarán como estructuras dinámicas.

6.2 Listas simplemente ligadas

Una ***lista simplemente ligada*** es una estructura de datos lineal, dinámica, formada por una colección de elementos llamados ***nodos***. Cada nodo está formado por dos partes: la primera de ellas se utiliza para almacenar la información (razón de ser de la estructura de datos), y la segunda se usa para guardar la dirección del siguiente nodo.

La figura 6.1 presenta un esquema de un nodo. Cabe destacar, que cada nodo sólo conoce la dirección del nodo que le sucede.

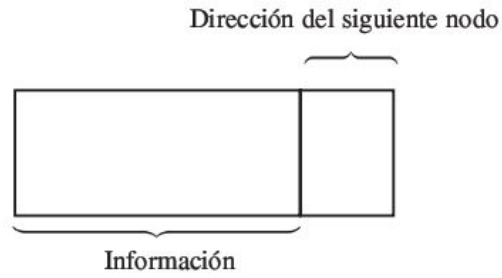


FIGURA 6.1 *Estructura de un nodo*

La figura 6.2 muestra la representación gráfica de una lista simplemente ligada. La lista está formada por una colección de nodos, cada uno de los cuales apunta al siguiente nodo, excepto el último que en la posición dedicada a la dirección de su vecino tiene el valor **NULL**. Además, se puede observar que se requiere de un puntero al primer elemento de la lista. Como éste no tiene predecesor, es indispensable que una variable tipo puntero almacene su dirección. A continuación se puede observar que el puntero al inicio de la lista se identifica con el nombre de `Primero`.

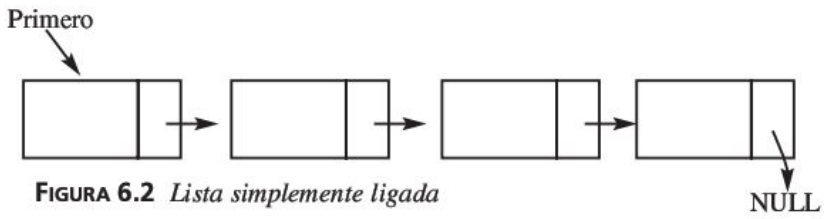


FIGURA 6.2 *Lista simplemente ligada*

Las figuras 6.3 y 6.4 presentan las plantillas de la clase `NodoLista` y de la clase `Lista` respectivamente. Se usan plantillas para dar mayor generalidad a la solución. La clase `NodoLista` tiene dos atributos, uno que representa la información a almacenar por lo que se define de tipo `T`, y otro que representa la dirección de otro nodo por lo que se define como un puntero a un objeto de la misma clase. Por su parte, la clase `Lista` tiene un único atributo que representa la dirección del primer elemento de la lista, por lo cual es de tipo puntero a un objeto de tipo `NodoLista`.

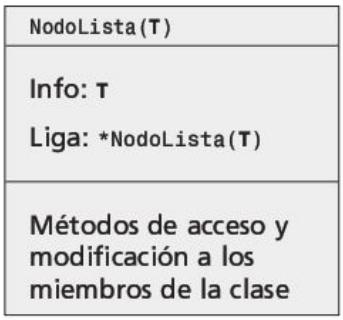


FIGURA 6.3 *Clase* `NodoLista`

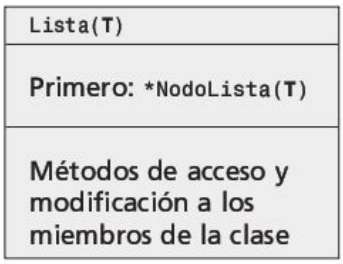


FIGURA 6.4 *Clase* `Lista`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a la definición de las plantillas de las clases `NodoLista` y `Lista`.

```
/* Prototipo de la plantilla de la clase Lista. Así, en la clase
➥NodoLista se podrá hacer referencia a ella. */
template <class  T>
class  Lista;

/* Definición de la plantilla de la clase NodoLista. La clase Lista se
➥declara como una clase amiga para que pueda tener acceso a los miembros
➥privados de la clase NodoLista. */
```

```
template <class  T>
class  NodoLista
{
   private:
       NodoLista<T>  * Liga;
       T Info;
   public:
       NodoLista();
       friend  class  Lista<T>;
};

/* Declaración del método constructor por omisión. */
NodoLista::NodoLista()
{
   Liga= NULL;
}

/* Definición de la plantilla de la clase Lista. Esta clase tiene un
➥solo atributo que es un puntero al primer elemento de la misma. */
template <class  T>
class  Lista
{
   private:
       NodoLista<T>  * Primero;
   public:
       Lista();
       /*En esta sección se incluyen los métodos de acceso y
       ➥modificación a los miembros de la clase. */
};

/* Declaración del método constructor por omisión. */
template <class  T>
Lista:: Lista()
{
   Primero= NULL;
}
```

La clase `NodoLista` se utiliza para representar un nodo, por lo tanto se incluyen dos atributos: uno para almacenar información de cualquier tipo (tipo `T`) y el otro para almacenar la dirección de otro objeto del mismo tipo. La sección pública contiene el método constructor y la declaración de amistad con la clase `Lista`, esto último para permitir que los miembros de ésta tengan acceso a sus propios miembros. Además, se podrían definir otros métodos, por ejemplo uno para regresar el atributo `Info` o uno para modificarlo.

A partir de la clase `NodoLista` se define la clase `Lista`, que está formada por un único atributo (tipo puntero a un objeto `NodoLista`) que representa el puntero al primer elemento de la lista. Este atributo permite el acceso a todos los elementos de la lista, debido a que el primero conoce la dirección del segundo, éste la del tercero y así sucesivamente hasta llegar al último. En la sección pública se declaran los métodos para tener acceso y modificar sus miembros, así como aquellos que permiten la manipulación de la información almacenada.

Las operaciones básicas a realizar en una lista previamente generada son: inserción, eliminación y búsqueda. La creación de la misma también se puede considerar dentro de esta categoría. Es importante destacar que cualquiera que sea la operación a realizar en una lista simplemente ligada no debe perderse la dirección del primer elemento de la misma. Teniendo este puntero se tiene acceso a todos los elementos, mientras que si se pierde su valor no existe manera de recuperar la dirección al primer nodo y de éste al segundo y así a los demás elementos. A continuación se analizan las principales operaciones. Las variantes de una misma operación se deben principalmente a la posición dentro de la lista donde se lleve a cabo ésta.

6.2.1 Inserción de elementos en una lista

La operación de ***inserción*** de un nuevo nodo a una lista consiste en tomar un espacio de memoria dinámicamente, asignarle la información correspondiente y ligarlo a otro nodo de la lista. Los pasos varían dependiendo de la posición del nodo al cual se ligue el nuevo elemento. La operación de crear un nodo, en el lenguaje ***C++***, se lleva a cabo por medio de la instrucción **`new`**. La misma asigna un espacio de memoria y da como resultado la dirección del bloque asignado. En caso de que no sea posible asignar el espacio de memoria, el resultado será el valor **NULL**.

Inserción al principio de la lista

La figura 6.5 presenta un esquema de la inserción de un nuevo elemento al inicio de la lista. Se crea un nodo, cuya dirección se guarda en una variable auxiliar llamada `P`, y se liga con el primero de la lista. Una vez realizado este paso, se redefine el *Primero* con el valor de `P`.

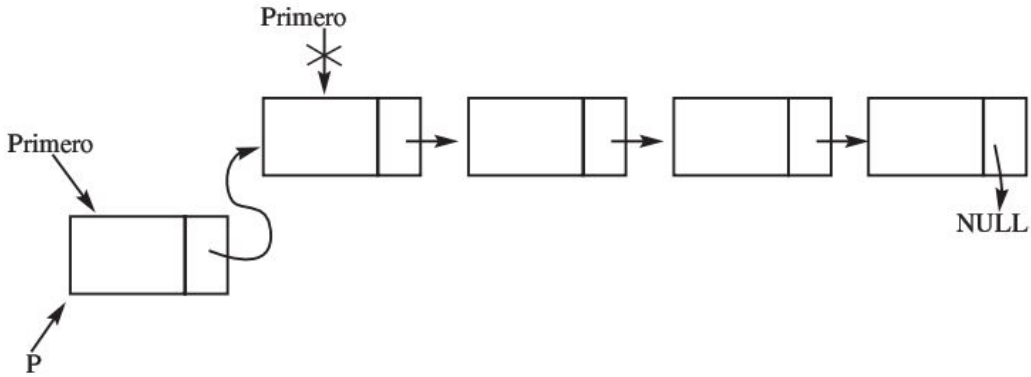


FIGURA 6.5 *Lista simplemente ligada*

El método para llevar a cabo esta operación es el siguiente:

```
/* Plantilla del método que inserta un elemento al inicio de la lista,
➥convirtiéndose en el primero de la misma. Recibe como parámetro el dato
➥a insertar. */
template <class  T>
void Lista<T>::InsertaInicio(T Dato)
{
   NodoLista<T>  * P;
   P= new    NodoLista<T>();
   P->Info= Dato;
   P->Liga= Primero;
   Primero= P;
}
```

Observe que se usa la notación `Variable->Atributo` debido a que `Variable` es un puntero a un objeto de la clase `NodoLista`.

El método presentado es válido para insertar un elemento al inicio de una lista vacía o al inicio de una lista previamente creada. Por lo tanto, puede generalizarse para crear una lista insertando los elementos siempre por el principio de la misma.

Inserción al final de la lista

Otro caso frecuente de inserción es cuando interesa agregar un nuevo elemento al final de la lista. La figura 6.6 presenta gráficamente esta operación. Se crea un nuevo nodo, apuntado por `P`, y se establece una liga entre el último nodo de la

lista y éste. Para llegar al último elemento es necesario recorrer toda la lista, desde el primero hasta dicho nodo. Si la lista es definida con un puntero al inicio y otro al final, el recorrido hasta el último nodo se omite. Sin embargo, según la definición previa de la clase `Lista`, se requiere hacer la operación auxiliar mencionada.

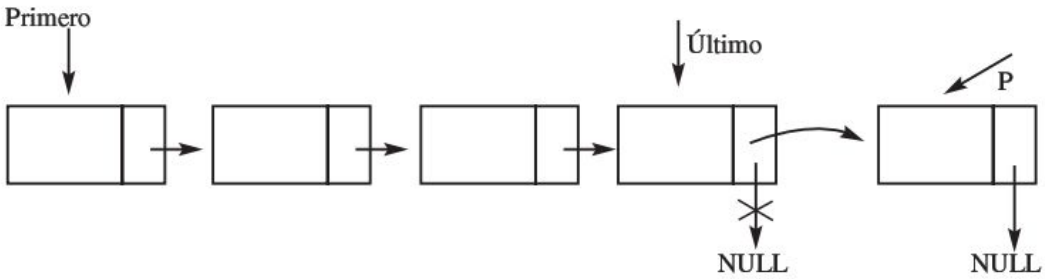


FIGURA 6.6 *Inserción al final de la lista*

El conjunto de pasos necesarios para llevar a cabo esta operación se presenta en el siguiente método de la clase `Lista`.

```
/* Método que inserta un nodo al final de la lista, convirtiéndose en el
➥último elemento de la misma. Recibe como parámetro el dato a almacenar
➥en dicho nodo. */
template <class  T>
void Lista<T>::InsertaFinal(T Dato)
{
   NodoLista<T>  * P, *Ultimo;
   P= new    NodoLista<T>();
   P->Info= Dato;
   if (Primero)
   {
        /* Si la lista tiene al menos un elemento, entonces se debe
        ➥recorrer hasta llegar al último nodo. */
        Ultimo= Primero;
        while (Ultimo->Liga)
             Ultimo= Ultimo->Liga;
        /* El último nodo de la lista apunta al nuevo nodo, cuya
        ➥dirección está en  P. */
        Ultimo->Liga= P;
   }
   else
        /* Si la lista no tiene elementos, entonces el nuevo elemento
        ➥será el primero de la misma. */
        Primero= P;
}
```

El método presentado incluye el recorrido de la lista hasta encontrar el último elemento. También contempla si la lista está vacía, ya que en este caso deberá redefinir el atributo `Primero` con el valor del puntero al nodo que insertó. Es importante mencionar que el constructor de la clase `NodoLista` se encargó de asignarle la constante **NULL** al atributo `Liga` de `P`, y dado que será el último de la lista se queda con ese valor.

Inserción antes de un nodo dado como referencia

La figura 6.7 presenta un esquema de la inserción de un nuevo elemento antes de un nodo que almacena cierto dato dado como referencia. Este caso es útil para el manejo de listas cuya información está ordenada.

Para llevar a cabo este tipo de inserción, primero se busca el nodo dado como referencia guardando la dirección del anterior (apuntado por `Ant`). Si se encuentra el nodo, entonces se crea otro (cuya dirección es almacenada en la variable `P`) estableciéndose las ligas entre éste y el dado como referencia, y entre el anterior y el nuevo.

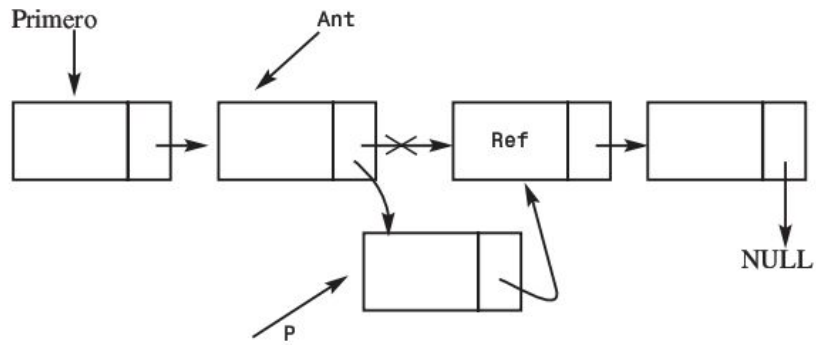


FIGURA 6.7 *Inserción antes de un nodo dado como referencia*

El método que implementa esta variante de la operación de inserción es el siguiente:

```
/* Método que inserta un nodo antes de un nodo dado como referencia.
➥Recibe como parámetros el dato a guardar en el nuevo nodo (Dato) y la
➥información dada como referencia (Ref). El método regresa 1 si se pudo
➥agregar el dato a la lista, 0 si no se encontró el dato dado como
➥referencia y -1 si la lista está vacía. */
```

```
template <class  T>
int  Lista<T>::InsertaAntes(T Dato, T Ref)
{
   NodoLista<T>  * P, *Ant, *Q;
   int  Resp= 1;
   if  (Primero)
   {
        Q= Primero;
        while ((Q !=  NULL ) && (Q->Info !=  Ref))
        {
          Ant= Q;
          Q= Q->Liga;
        }
        if (Q !=  NULL )
        {
          P= new   NodoLista<T>();
          P->Info= Dato;
          /* El dato de referencia es el primero de la lista. */
          if (Primero ==  Q)
          {
              P->Liga= Primero;
              Primero= P;
          }
          else
          {
              Ant->Liga= P;
              P->Liga= Q;
          }
        }
        else
            /* No se encontró el dato dado como referencia. */
             Resp= 0;
   }
   else
        /* La lista está vacía. */
        Resp= -1;
   return Resp;
}
```

El método presentado incluye la búsqueda del elemento dado como referencia. Además, contempla los posibles casos de fracaso: si la lista está vacía o si el elemento dado como referencia no está en la lista. También se toma en cuenta la posición del elemento de referencia dentro de la lista, ya que si es el primero de la misma el puntero `Primero` se debe redefinir.

Inserción después de un nodo dado como referencia

La figura 6.8 presenta un esquema de la inserción de un nuevo elemento después de un nodo que almacena cierto dato dado como referencia. Este caso, lo mismo que el anterior, es usado cuando se trabaja con listas cuya información está ordenada.

Se empieza buscando el nodo que guarda el dato dado como referencia. Si se encuentra, entonces se crea un nodo cuya dirección queda en la variable `P`. Luego se establecen las ligas requeridas para relacionar el nuevo elemento con el dado como referencia y con el sucesor de este último.

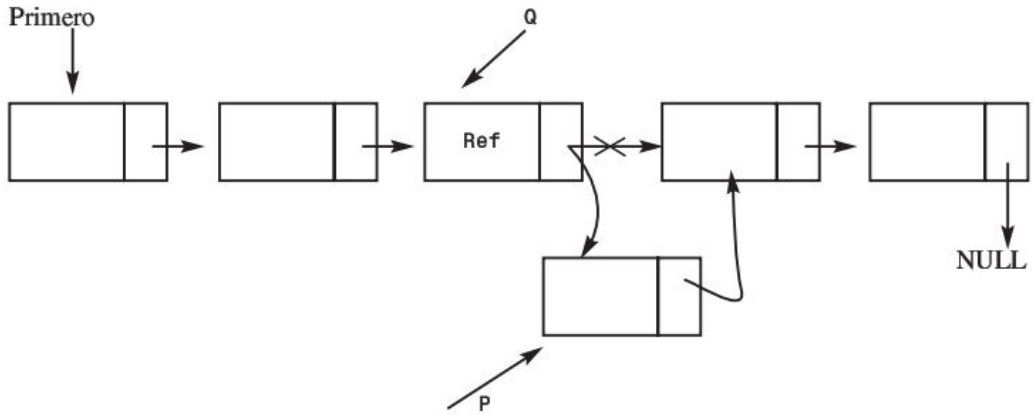


FIGURA 6.8 *Inserción antes de un nodo dado como referencia*

A continuación se presenta el método que implementa este tipo de inserción de nodos en una lista simplemente ligada.

```
/* Método que inserta un nuevo elemento (Dato) como nodo sucesor de uno
➥que almacena un dato dado como referencia (Ref). El método regresa 1 si
➥se pudo insertar, 0 si no se encontró la referencia y -1 si la lista
➥está vacía. */
template <class  T>
int  Lista<T>::InsertaDespues(T Dato, T Ref)
{
   NodoLista<T>  * Q, *P;
   int  Resp= 1;
   if (Primero)
```

```
    {
        Q = Primero;
        while ((Q != NULL ) && (Q->Info != Ref))
            Q= Q->Liga;
        if ( Q != NULL )
        {
            P= new   NodoLista<T>();
            P->Info= Dato;
            P->Liga= Q->Liga;
            Q->Liga= P;
            }
        else
            /* No se encontró la referencia. */
            Resp= 0;
    }
    else
        /* La lista está vacía. */
        Resp= -1;
    return Resp;
}
```

El método presentado incluye la búsqueda del elemento dado como referencia. Además, contempla los posibles casos de fracaso: si la lista está vacía o si el elemento dado como referencia no está en la lista.

Los métodos presentados podrían modificarse incluyendo la evaluación de si hubo o no espacio de memoria disponible. Si el valor arrojado por la instrucción `new` fue **NULL**, entonces el método debería indicarlo de alguna manera al programa usuario.

6.2.2 Eliminación de elementos de una lista

La operación de ***eliminación*** de un nodo de una lista consiste en encontrar el valor a quitar, establecer la liga correspondiente entre el nodo que lo precede y su sucesor y finalmente liberar la porción de memoria ocupada por el nodo eliminado. Se pueden presentar algunas variantes según la posición que el elemento tenga en la lista. A continuación se analizan los principales casos. La operación de liberar espacio de memoria, en el lenguaje ***C++***, se lleva a cabo por medio de la instrucción `delete()`.

Eliminación del primer elemento de la lista

La figura 6.9 presenta el esquema correspondiente a esta operación. El nodo que se va a eliminar debe ser apuntado por una variable auxiliar, en este caso llamada P, luego se debe redefinir el *Primero* con la dirección de su sucesor y finalmente se libera la porción de memoria ocupada por el nodo.

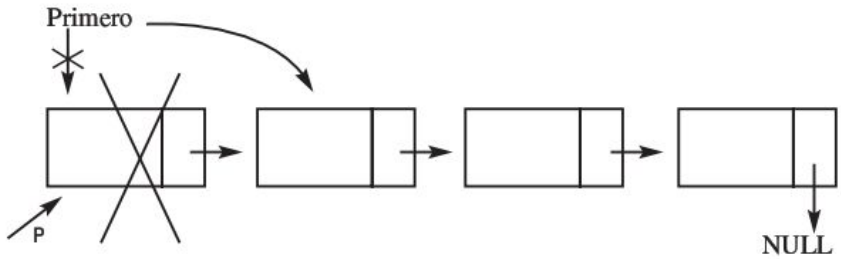


FIGURA 6.9 *Eliminación del primer nodo de la lista*

El método para llevar a cabo esta operación es el siguiente:

```
/* Método que elimina el primer elemento de la lista. El método redefine
➥el puntero al inicio de la lista y libera el espacio de memoria del nodo
➥eliminado. Regresa 1 si se realizó la operación y 0 en caso contrario. */
template <class  T>
int  Lista<T>::EliminaPrimero()
{
   NodoLista<T>  * P;
   int  Resp= 1;
   if (Primero)
   {
       P= Primero;
       Primero= P->Liga;
       delete  (P);
   }
   else
       /* La lista está vacía. */
       Resp= 0;
   return  Resp;
}
```

El método presentado verifica si la lista tiene al menos un nodo, ya que en caso contrario no existe un primer elemento que pueda quitarse.

Eliminación del último elemento de la lista

La figura 6.10 presenta gráficamente la secuencia de pasos necesarios para quitar el último nodo de una lista simplemente ligada. En este caso, se debe recorrer la lista hasta llegar al elemento deseado, guardando la dirección de su predecesor. Una vez encontrado se debe redefinir su predecesor como último elemento de la lista y liberar el espacio de la memoria correspondiente.

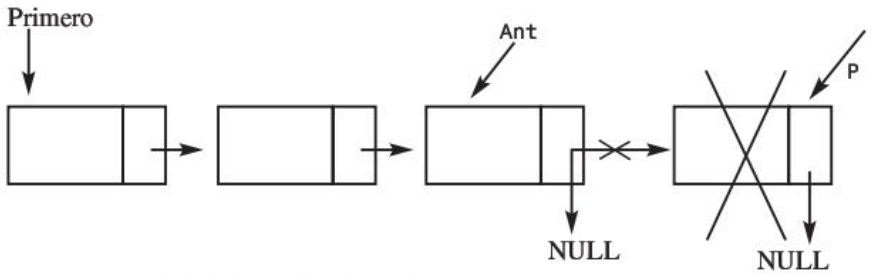


FIGURA 6.10 *Eliminación del último nodo de la lista*

A continuación se presenta el método que implementa los pasos explicados anteriormente.

```
/* Método que elimina el último nodo de una lista. Primero lo localiza,
➥guardando la dirección del nodo que le precede. Posteriormente redefine
➥la liga de éste con el valor de NULL para indicar que ahora es el último
➥y finalmente libera el espacio de memoria. El método regresa 1 si se
➥puede llevar a cabo la eliminación y 0 en caso contrario. */
template <class  T>
int  Lista<T>::EliminaUltimo()
{
   NodoLista<T>  * Ant, *P;
   int  Resp= 1;
   if (Primero)
   {
        /* Verifica si la lista está formada por un único elemento,
        ➥en tal caso redefine el puntero al inicio con el valor de NULL,
        ➥indicando lista vacía. */
        if (!Primero->Liga)
        {
          delete  (Primero);
          Primero= NULL;
```

```
            }
            else
            {
                P= Primero;
                while (P->Liga)
                {
                    Ant= P;
                    P= P->Liga;
                }
                Ant->Liga= NULL;
                delete (P);
             }
        }
        else
            /* La lista está vacía. */
            Resp= 0;
        return     Resp;
}
```

El método presentado considera el caso de una lista formada por un único elemento (primero y último) la cual, después de la eliminación, queda vacía.

Eliminación de un elemento de la lista

La figura 6.11 presenta gráficamente los pasos necesarios para eliminar al nodo que almacena cierta información, en una lista previamente formada. Se puede observar que primero se recorre la lista hasta llegar al elemento buscado (valor de referencia dado por el usuario, apuntado por `P`), guardando la dirección de su predecesor (`Ant`). Una vez encontrado se debe establecer la liga entre su predecesor y su sucesor y liberar el espacio de memoria ocupado por dicho nodo.

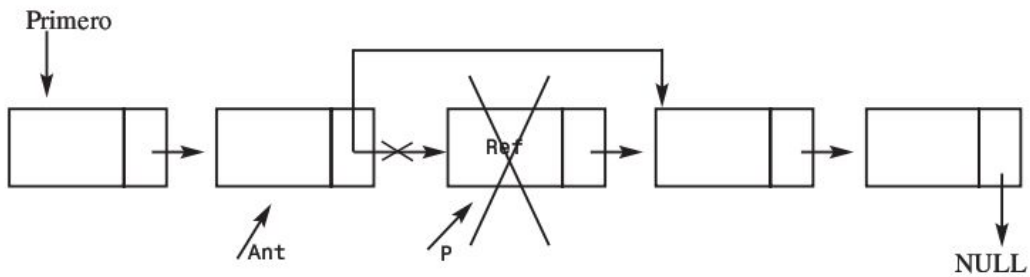


FIGURA 6.11 *Eliminación de un elemento de la lista*

El método para llevar a cabo esta operación es el siguiente:

```
/* Método que elimina un nodo que almacena cierta información. El método
➥verifica que la lista tenga elementos y que el elemento dado como
➥referencia se encuentre en la lista. Recibe como parámetro el dato a
➥eliminar y regresa como resultado 1 si lo elimina, 0 si no lo encuentra
➥y -1 si la lista está vacía. */
template <class  T>
int  Lista<T>::EliminaUnNodo(T Ref)
{
   NodoLista<T>  * P, *Ant;
   int  Resp= 1;
   if (Primero)
   {
       P= Primero;
       while ((P->Liga) && (P->Info != Ref))
       {
          Ant= P;
          P= P->Liga;
       }
       if  (P->Info != Ref)
            /* El elemento no fue encontrado. */
            Resp= 0;
       else
       {
            if (Primero == P)
                 Primero= P->Liga;
            else
                 Ant->Liga= P->Liga;
            delete (P);
       }
   }
   else
       Resp= -1;
   return  Resp;
}
```

El método presentado contempla el caso de que el elemento dado como referencia no se encuentre en la lista, así como el caso de que sea el primero. Por otra parte, si la lista tiene un único elemento y éste fuera el nodo a quitar, al redefinir `Primero` con el valor de *P–>Liga*, se le estaría asignando la constante **NULL**.

A los métodos definidos para la operación de eliminación se los podría modificar de tal manera que regresen, a través de un parámetro, la dirección del nodo quitado de la lista. De esta forma, el nodo o su contenido podrían usarse en la aplicación, luego de lo cual se liberaría el espacio de memoria.

A continuación, el programa 6.1 presenta la plantilla de la clase `Lista` con todos sus atributos y métodos. Además de las principales operaciones analizadas, se incluyeron otras que pueden ser útiles para el manejo de la información almacenada en una lista simplemente ligada. Por razones de espacio, se incluyó sólo el prototipo y el encabezado de algunos de los métodos ya explicados.

Programa 6.1

```
/* Definición de la plantilla de la clase NodoLista y de la clase Lista.
➥Se incluyeron los métodos más usados. Sin embargo, dependiendo de la
➥aplicación se podrían definir otros. */

/* Prototipo de la plantilla de la clase Lista. Así, en la clase
➥NodoLista se podrá hacer referencia a ella. */
template <class  T>
class  Lista;

/* Definición de la clase NodoLista. */
template <class  T>
class  NodoLista
{
   private:
       NodoLista<T>  * Liga;
       T Info;
   public:
       NodoLista();
       T RegresaInfo();
       friend  class  Lista<T>;
};

/* Declaración del método constructor por omisión. Inicializa con el
➥valor NULL al puntero al siguiente nodo. */
template <class  T>
NodoLista<T>::NodoLista()
{
   Liga= NULL;
}

/* Método que permite, a usuarios ajenos a la clase, conocer el valor
➥del atributo Info. */
template <class  T>
T NodoLista<T>::RegresaInfo()
{
   return Info;
}
```

```
/* Definición de la clase Lista. */
template <class  T>
class  Lista
{
   private:
       NodoLista<T>  * Primero;
   public:
       Lista ();
       NodoLista<T>  * RegresaPrimero();
       void CreaInicio();
       void CreaFinal();
       void ImprimeIterativo();
       void ImprimeRecursivo(NodoLista<T>  * );
       void ImprimeUnNodo(NodoLista<T>  * );
       void InsertaInicio(T);
       void InsertaFinal(T);
       void InsertaOrdenCrec(T);
       int  InsertaAntes(T, T);
       int  InsertaDespues(T, T);
       int  EliminaPrimero();
       int  EliminaUltimo();
       int  EliminaUnNodo(T);
       int  EliminaAnterior(T);
       int  EliminaDespues(T);
       NodoLista<T>  * BuscaDesordenada(T);
       NodoLista<T>  * BuscaOrdenada(T);
       NodoLista<T>  * BuscaRecursivo(T, NodoLista<T>  *);
};

/* Declaración del método constructor. Inicializa el puntero al primer
➥nodo de la lista con el valor NULL: indica lista vacía. */
template <class  T>
Lista<T>::Lista()
{
   Primero= NULL;
}

/* Método que regresa la dirección del primer nodo de la lista. */
template <class  T>
NodoLista<T>  * Lista<T>::RegresaPrimero()
{
   return  Primero;
}

/* Método que crea una lista agregando el nuevo nodo al inicio de la
➥misma. */
template <class  T>
void Lista<T>::CreaInicio()
```

```
{
   NodoLista<T>  * P;
   T Dato;
   char   Resp;
   Primero= new   NodoLista<T>();
   cout<<  "Ingrese la información a almacenar: \n";
   cin>>Dato;
   Primero->Info= Dato;
   cout<<  "\n¿Desea ingresar otro elemento (S/N)? ";
   cin>>Resp;
   while  (Resp == 'S' || Resp == 's')
   {
      cout<<  "Ingrese la información: \n";
      cin>> Dato;
      P = new   NodoLista<T>();
      P->Info= Dato;
      P->Liga= Primero;
      Primero= P;
      cout<<  "\n¿Desea ingresar otro elemento (S/N)? ";
      cin>>Resp;
   }
}

/* Método que crea una lista agregando el nuevo nodo al final de la
➥misma. */
template <class  T>
void Lista<T>::CreaFinal()
{
   NodoLista<T>  * P, *Ultimo;
   T Dato;
   char  Resp;
   Primero= new   NodoLista<T>();
   cout<<"Ingrese la información a almacenar: \n";
   cin>>Dato;
   Primero->Info= Dato;
   /* Se mantiene un puntero al último nodo agregado a la lista para
   ➥evitar tener que recorrerla con cada nuevo nodo. */
   Ultimo= Primero;
   cout<<"\n¿Desea ingresar otro elemento (S/N)? ";
   cin>>Resp;
   while  (Resp == 'S' || Resp == 's')
   {
      cout<< "\nIngrese la información \n";
      cin>>Dato;
      P= new   NodoLista<T>();
      P->Info= Dato;
      Ultimo->Liga= P;
      Ultimo= P;
      cout<< "\n¿Desea ingresar otro elemento (S/N)? ";
      cin>>Resp;
   }
}
```

```
/* Método que despliega el contenido de la lista iterativamente. */
template <class  T>
void Lista<T>::ImprimeIterativo()
{
   NodoLista<T>  * P;
   P= Primero;
   while (P)
   {
       cout<< "\nInformación: "<< P->Info;
       P= P->Liga;
   }
   cout<<   '\n';
}

/* Método que despliega el contenido de la lista recursivamente. Recibe
como parámetro el nodo cuya información se va a imprimir. */
template <class  T>
void Lista<T>::ImprimeRecursivo(NodoLista<T>  * P)
{
   if (P)
   {
       cout<<"\nInformación: "<< P->Info;
       ImprimeRecursivo(P->Liga);
   }
   cout<<'\n';
}

/* Método que imprime la información de un nodo dado como dato. */
template <class  T>
void Lista<T>::ImprimeUnNodo(NodoLista<T>  * P)
{
   if (P)
       cout<< P->Info;
}

/* Método que inserta un nodo al inicio de la lista. El método es válido
➥tanto para listas ya creadas como para listas vacías. */
template <class  T>
void Lista<T>::InsertaInicio(T Dato)
{
/* Presentado más arriba. */
}

/* Método que inserta un nodo al final de la lista. El método es válido
➥tanto para listas ya creadas como para listas vacías. */
template <class  T>
void Lista<T>::InsertaFinal(T Dato)
{
   /* Presentado más arriba. */
}
```

```
/* Método que inserta un nodo en orden creciente. Luego de varias
➥inserciones, usando este método, se habrá generado una lista ordenada
➥de menor a mayor. */
template <class  T>
void Lista<T>::InsertaOrdenCrec(T Dato)
{
   NodoLista<T>  * P, *Q, *Ant;
   if (!Primero || Primero->Info > Dato)
        InsertaInicio(Dato);
   else
   {
        Q= Primero;
        while (Q && Q->Info < Dato)
        {
           Ant= Q;
           Q= Q->Liga;
        }
        P= new  NodoLista<T>();
        P->Info= Dato;
        Ant->Liga= P;
        P->Liga= Q;
   }
}

/* Método que inserta un nodo antes de un nodo dado como referencia. Recibe
➥como parámetros la información a insertar y un dato dado como referencia.
➥Regresa 1 si se pudo insertar, 0 si no se encontró la referencia y -1
➥si la lista está vacía. */
template <class  T>
int  Lista<T>::InsertaAntes(T Dato, T Ref)
{
   /* Presentado más arriba. */
}

/* Método que inserta un nodo después de uno dado como referencia. Recibe
➥como parámetros la información a insertar y la referencia. Regresa 1 si
➥se pudo insertar, 0 si no se encontró el dato dado y -1 si la lista
➥está vacía. */
template <class  T>
int  Lista<T>::InsertaDespues(T Dato, T Ref)
{
   /* Presentado más arriba. */
}

/* Método que elimina el primer elemento de la lista. El método redefine
➥el puntero al inicio de la lista y libera el espacio de memoria del nodo
➥eliminado. Regresa 1 si se pudo llevar a cabo la operación y 0 en caso
➥contrario. */
template <class  T>
int  Lista<T>::EliminaPrimero()
```

```
{
/* Presentado más arriba. */
}

/* Método que elimina el último elemento de una lista. Primero lo
➥localiza, guardando la dirección del nodo que le precede. Posterior-
➥mente redefine la liga de éste con el valor de NULL para indicar que
➥ahora éste es el último y libera el espacio de memoria. Regresa 1 si se
➥pudo llevar a cabo la eliminación y 0 en caso contrario. */
template <class  T>
int  Lista<T>::EliminaUltimo()
{
   /* Presentado más arriba. */
}

/* Método que elimina un nodo que almacena cierta información. Recibe
➥como parámetro el dato a eliminar y regresa como resultado 1 si lo
➥elimina, 0 si no lo encuentra y -1 si la lista está vacía. */
template <class  T>
int  Lista<T>::EliminaUnNodo(T Ref)
{
   /* Presentado más arriba. */
}

/* Método que elimina el nodo anterior al nodo que almacena un dato dado
➥como referencia. Regresa 1 si el nodo fue eliminado, 2 si la referencia
➥es el primero, 3 si no fue encontrado y 4 si la lista está vacía. */
template <class  T>
int  Lista<T>::EliminaAnterior(T Ref)
{
   NodoLista<T>  * Q, *Ant, *P;
   int  Resp= 1;
   if (Primero)
   {
       if (Primero->Info == Ref)
           /* No hay nodo que preceda al dado como referencia. */
           Resp= 2;
       else
       {
           Q= Primero;
           Ant= Primero;
           /* Ciclo que permite encontrar la información dada como
           ➥referencia, guardando la dirección del nodo que le precede
           ➥(nodo que se eliminará) y del anterior a éste para estable-
           ➥cer las ligas correspondientes. */
           while ((Q->Info != Ref) && (Q->Liga))
           {
              P= Ant;
              Ant= Q;
              Q= Q->Liga;
           }
```

```
            if (Q->Info != Ref)
                /* El elemento dado como referencia no está en la lista. */
                Resp= 3;
            else
                if (Primero->Liga == Q)
                {
                    delete (Primero);
                    Primero= Q;
                }
                else
                {
                    P->Liga= Q;
                    delete (Ant);
                }
      }
   }
   else
       /* La lista está vacía. */
       Resp= 4;
   return  Resp;
}

/* Método que busca un elemento dado referencia en una lista desordenada.
➥Regresa la dirección del nodo si lo encuentra y NULL en caso contrario. */
template <class  T>
NodoLista<T>  * Lista<T>::BuscaDesordenada(T Ref)
{
   NodoLista<T>  * Q, *Resp= NULL;
   if (Primero)
   {
       Q= Primero;
       while ((Q->Info != Ref)  && (Q->Liga))
          Q= Q->Liga;
       /* Se verifica si el elemento dado como referencia fue encontrado
       ➥en la lista. */
       if (Q->Info == Ref)
           Resp= Q;
   }
   return Resp;
}

/* Método que busca un elemento dado como referencia, en una lista
➥ordenada de menor a mayor. Regresa la dirección del nodo si lo
➥encuentra y NULL en caso contrario. */
template <class  T>
NodoLista<T>  * Lista<T>::BuscaOrdenada(T Ref)
{
   NodoLista<T>  * Q, *Resp= NULL;
   if (Primero)
```

```
    {
        Q= Primero;
        while ((Q->Info < Ref) && (Q->Liga))
           Q= Q->Liga;
        /* Se verifica si el elemento dado como referencia fue encontrado
        ➥en la lista. */
        if (Q->Info == Ref)
           Resp= Q;
    }
    return  Resp;
}

/* Método que busca un dato en la lista. La operación se realiza
➥recursivamente. El método recibe como parámetro el elemento a buscar
➥(Dato) y una variable (Q) que almacena la dirección de un nodo (la
➥primera vez es la dirección del primero). Regresa como resultado la
➥dirección del nodo si lo encuentra y NULL en caso contrario. */
template <class  T>
NodoLista<T>  * Lista<T>::BuscaRecursivo(T Dato, NodoLista<T>  * Q)
{
   if (Q)
       if (Q->Info == Dato)
           return Q;
       else
           return  BuscaRecursivo(Dato, Q->Liga);
   else
       return  NULL;
}
```

El programa 6.1 presenta la plantilla de la clase `Lista` con los métodos más usados para el tratamiento de objetos de esta clase. A pesar de que algunos de los métodos pueden estar implícitos en otros, se decidió incluirlos de esta manera para obtener mayor claridad. Un ejemplo de este caso es el método `CreaInicio()`, el cual puede suprimirse y ser absorbido por el método `InsertaInicio()`, ya que éste es aplicable a una lista vacía (en dicho caso se estaría creando la lista).

El programa 6.3 presenta una aplicación de las listas simplemente ligadas. El objetivo de este programa es permitir al usuario registrar información de diversos productos, así como eliminar productos ya registrados, conocer la información relacionada con cierto producto y obtener un reporte con los datos de todos los productos. Para la representación de los productos se usará la clase `Producto` (ver programa 6.2), mientras que para su almacenamiento se utilizará una lista. Ésta se creará a partir de la plantilla de la clase `Lista` correspondiente al programa 6.1, la cual está en la biblioteca *"ListasSimLig.h"*.

Programa 6.2

```
/* Definición de la clase Producto. Se sobrecargan algunos operadores
➥para que objetos de esta clase puedan ser usados de manera directa.
➥Esta clase se guarda en la biblioteca "Productos.h". */
class  Producto
{
   private:
       int Clave;
       char NomProd[64];
       double Precio;
   public:
       Producto();
       Producto(int, char[], double);
       double RegresaPrecio();
       int operator == (Producto);
       int operator != (Producto);
       int operator > (Producto);
       int operator < (Producto);
       friend   istream &operator>> (istream &, Producto &);
       friend   ostream &operator<< (ostream &, Producto &);
};

/* Declaración del método constructor por omisión. */
Producto::Producto()
{ }

/* Declaración del método constructor con parámetros. */
Producto::Producto(int Cla, char NomP[], double Pre)
{
   Clave= Cla;
   strcpy(NomProd, NomP);
   Precio= Pre;
}

/* Método que regresa el valor del atributo Precio. */
double Producto::RegresaPrecio()
{
   return   Precio;
}

/* Método que permite comparar dos objetos de tipo Producto para
➥determinar si son iguales. Regresa 1 si los productos son iguales
➥(tienen la misma clave) y 0 en caso contrario. Se usa sobrecarga del
➥operador ==. */
int Producto::operator == (Producto Prod)
```

```
{
   int Resp=0;
   if (Clave == Prod.Clave)
       Resp= 1;
   return   Resp;
}

/* Método que permite comparar dos objetos de tipo Producto para
➥determinar si son distintos. Regresa 1 si los productos son distintos
➥(tienen diferente clave) y 0 en caso contrario. Se usa
➥sobrecarga del operador !=.  */
int Producto::operator != (Producto Prod)
{
   int Resp=0;
   if (Clave != Prod.Clave)
       Resp= 1;
   return   Resp;
}

/* Método que permite comparar dos objetos de tipo Producto para
➥determinar si el asociado al operador es mayor que el dado como
➥parámetro. Regresa 1 cuando es mayor (su clave es mayor que la clave
➥del dado como parámetro) y 0 en caso contrario. Se usa sobrecarga del
➥operador >.  */
int Producto::operator > (Producto Prod)
{
   int Resp=0;
   if (Clave > Prod.Clave)
       Resp= 1;
    return   Resp;
}

/* Método que permite comparar dos objetos de tipo Producto para
➥determinar si el asociado al operador es menor que el dado como
➥parámetro. Regresa 1 cuando es menor (su clave es menor que la clave
➥del dado como parámetro) y 0 en caso contrario. Se usa sobrecarga del
➥operador <.  */
int Producto::operator < (Producto Prod)
{
   int Resp=0;
   if (Clave < Prod.Clave)
       Resp= 1;
   return   Resp;
}

/* Sobrecarga del operador >> para permitir la lectura de objetos de
➥tipo Producto de manera directa con el cin. */
```

```
istream &operator>> (istream &Lee,  Producto &ObjProd)
{
   cout<<"\n\nIngrese clave del producto:  ";
   Lee>> ObjProd.Clave;
   cout<<"\n\nIngrese nombre del producto:  ";
   Lee>> ObjProd.NomProd;
   cout<<"\n\nIngrese precio:  ";
   Lee>> ObjProd.Precio;
   return    Lee;
}

/* Sobrecarga del operador << para permitir la impresión de objetos de
➥tipo Producto de manera directa con el cout. */
ostream &operator<< (ostream &Escribe, Producto &ObjProd)
{
   Escribe<<"\n\nDatos del producto\n";
   Escribe<<"\nClave:  "<<ObjProd.Clave;
   Escribe<<"\nNombre:  "<<ObjProd.NomProd;
   Escribe<<"\nPrecio:  "<<ObjProd.Precio<<"\n";
   return    Escribe;
}
```

Programa 6.3

```
/* Este programa muestra el uso de las listas para almacenar y recuperar
➥información. En este caso se ofrecen opciones de trabajo al usuario para
➥guardar, eliminar o consultar datos de un cierto producto, así como
➥generar un reporte con los datos de todos los productos almacenados
➥hasta el momento. Para evitar la repetición de código se incluyen las
➥bibliotecas "ListasSimLig.h" y "Productos.h". La primera corresponde a
➥la plantilla de la clase Lista presentada en el programa 6.1 y la
➥segunda a la clase Producto presentada en el programa 6.2. */

#include "ListasSimLig.h"
#include "Productos.h"


/* Función auxiliar que presenta al usuario las diferentes opciones de
➥trabajo. */
int Menu()
{
   int Opc;
   cout<<"\n\nBienvenido al sistema de registro de productos.\n\n";
   cout<<"\n(1) Registrar un nuevo producto.\n";
   cout<<"\n(2) Dar de baja un producto.\n";
   cout<<"\n(3) Verificar si un producto ya fue registrado.\n";
   cout<<"\n(4) Imprimir la lista de productos registrados.\n";
```

```
   cout<<"\n(5) Salir.\n";
   cout<<"\n\nIngrese opción de trabajo:\n";
   cin>>Opc;
   return   Opc;
}


/* Función principal. Se declara un objeto de tipo Lista, el cual
➥servirá para llevar a cabo las operaciones de almacenamiento, consulta
➥y eliminación de información relacionada a productos. */
void main()
{
   Lista<Producto> ListaProds;
   Producto ObjProd;
   NodoLista<Producto> *Apunt;
   int Opc, Res, Clave;

   Opc= Menu();
   while (Opc >= 1 &&  Opc <= 4)
   {
      /* Selección de la operación a realizar considerando la opción
      ➥elegida por el usuario. */
      switch  (Opc)
      {
        /* Los productos se guardan en la lista ordenados de manera
        creciente, según su clave. */
        case 1: {
                  cout<<"\n\nIngrese datos del producto a registrar:\n";
                  cin>>ObjProd;
                  ListaProds.InsertaOrdenCrec(ObjProd);
                  break;
                }
        case 2: {
                  cout<<"\n\nIngrese la clave del producto a eliminar:\n";
                  cin>>Clave;
                  /* Se solicita sólo la clave del producto, ya que la
                  ➥búsqueda se hace tomando en cuenta este atributo que
                  ➥es el que lo identifica. */
                  Producto Produc(Clave,"", 0);
                  Res= ListaProds.EliminaUnNodo(Produc);
                  switch  (Res)
                  {
                     case 1: cout<<"\n\nEl producto ya fue eliminado.\n";
                             break;
                     case 0: cout<<"\n\nEse producto no se
                              ➥encuentra registrado.\n";
                             break;
                     case -1: cout<<"\n\nNo hay productos
                              ➥registrados.\n";
```

```
                                        break;
                    }
                }
                break;
        case 3: {
                    cout<<"\n\nIngrese la clave del producto a buscar:\n";
                    cin>>Clave;
                    /* Se solicita sólo la clave del producto, ya que la
                    ➥búsqueda se hace tomando en cuenta este atributo que
                    ➥es el que lo identifica. */
                    Producto Produc(Clave,"",0);
                    Apunt= ListaProds.BuscaOrdenada(Produc);
                    if (!Apunt)
                        cout<<"\n\nEse producto no está registrado.\n\n";
                    else
                    {
                        cout<<"\n\nEse producto está registrado.\n";
                        ListaProds.ImprimeUnNodo(Apunt);
                    }
                }
                break;
        case 4:ListaProds.ImprimeRecursivo(ListaProds.RegresaPrimero());
               break;
        }
        Opc= Menu();
    }
}
```

6.2.3 Implementación de pilas por medio de listas

La implementación de estructuras tipo pila por medio de listas constituye otro ejemplo interesante de aplicación de estas últimas. Recordemos que las pilas son estructuras abstractas que requieren de otras estructuras para su implementación.

Las operaciones de inserción y eliminación de elementos en una pila se realizan por uno de los extremos de la estructura. Por lo tanto, el conjunto de operaciones posibles vistas en la clase `Lista` se reduce a insertar y eliminar por el inicio. Se reutilizan algunos de los métodos del programa 6.1 para implementar las pilas por medio de listas. El método correspondiente a la eliminación del primer elemento se modifica considerando la observación hecha anteriormente, es decir, si se pudo eliminar el primer elemento, regresa como parámetro el contenido del nodo eliminado. Este método ya incluye la evaluación de pila vacía. A su vez, el método de inserción al inicio se modifica declarándolo entero. De esta forma, se evalúa la condición de pila llena en el mismo método, y el resultado que arroje el

método dependerá de la evaluación del valor dado por la instrucción **new**. Es decir, si se pudo asignar espacio de memoria entonces la pila no está llena y por lo tanto se llevará a cabo la inserción.

El programa 6.4 presenta un ejemplo sencillo de pilas implementadas por medio de listas simplemente ligadas. El programa evalúa expresiones aritméticas dadas en notación postfija (los operandos preceden a los operadores). Los operandos sólo pueden ser números enteros de un dígito y los operadores reconocidos son: **+**, **–**, * y /.

Programa 6.4

```
/* Prototipo de la plantilla de la clase Pila. De esta forma, la clase
➥Nodo podrá hacer referencia a ella. */
template <class  T>
class  Pila;


/* Definición de la clase Nodo. */
template <class  T>
class  Nodo
{
   private:
        Nodo<T>  * Liga;
        T Info;
   public:
         Nodo();
         friend  class  Pila<T>;
};

/* Declaración del método constructor por omisión. Inicializa con el
➥valor NULL el puntero al siguiente nodo. */
template <class  T>
Nodo<T>::Nodo()
{
   Liga= NULL;
}


/* Definición de la clase Pila. Su único atributo es el Tope, que en
➥este caso es un puntero al primer elemento almacenado en la pila. */
template <class  T>
class  Pila
```

```
{
   private:
        Nodo<T>  * Tope;
   public:
        Pila ();
        int Push(T);
        int Pop(T *);
};

/* Declaración del método constructor. Inicializa el puntero al primer
elemento de la pila con el valor NULL. Indica pila vacía. */
template <class  T>
Pila<T>::Pila()
{
   Tope= NULL;
}

/* Método que inserta un elemento en la pila. Recibe como parámetro el
➥dato a insertar. El método verifica el caso de pila llena. Si se puede
➥llevar a cabo la inserción regresa 1, en caso contrario regresa 0. */
template <class  T>
int Pila<T>::Push(T Dato)
{
   Nodo<T>  * Apunt;
   int Resp= 1;
   Apunt= new  Nodo<T>();
   /* Verifica si hay espacio de memoria disponible. */
   if (Apunt)
   {
        Apunt->Info= Dato;
        Apunt->Liga= Tope;
        Tope= Apunt;
   }
   else
        Resp= 0;
   return  Resp;
}

/* Método que elimina el elemento de la pila que está en el Tope. El
➥método redefine el valor de Tope y libera el espacio de memoria del nodo
➥eliminado. Regresa 1 si se lleva a cabo la eliminación y 0 en caso con-
➥trario. Además, pasa como parámetro el contenido del nodo eliminado. */
template <class  T>
int Pila<T>::Pop(T *Dato)
{
   Nodo<T>  * Apunt;
   int Resp= 1;
   if (Tope)
```

```
    {
         *Dato= Tope->Info;
         Apunt= Tope;
         Tope= Apunt->Liga;
         delete(Apunt);
    }
    else /* La Pila está vacía. */
         Resp= 0;
    return  Resp;
}


/* Función principal. Se lee una expresión aritmética dada en notación
➥postfija. La evalúa con ayuda de una pila. Se sugiere que siga el pro-
➥grama con la expresión: "5 3 + 8 * 2 / "                              */
void main()
{
    Pila<double> Operandos;
    char Expresion[20];
    int Indice, Resp= 1;
    double Resultado, Op1, Op2;
    cout<<"\n\nIngrese la expresión en notación postfija. \n";
    cin>>Expresion;

    for (Indice= 0; Indice < strlen(Expresion); Indice++)
         if (Expresion[Indice] >= '1' && Expresion[Indice] <= '9')
             Operandos.Push(Expresion[Indice]-'0');
             /* Se le resta el ordinal correspondiente al carácter '0' del
             ➥código ASCII para obtener el valor decimal del carácter tomado
             ➥de la cadena. Si, por ejemplo, el carácter fuera el '8', al
             ➥restarle el ordinal del '0' queda el valor entero 8. */
         else
             if (Operandos.Pop(&Op1) && Operandos.Pop(&Op2))
             {
                 switch  (Expresion[Indice])
                 {
                   case '+': Resultado= Op2 + Op1;
                             break;
                   case '-': Resultado= Op2 - Op1;
                             break;
                   case '*': Resultado= Op2 * Op1;
                             break;
                   case '/': Resultado= Op2 / Op1;
                             break;
                 }
                  Operandos.Push(Resultado);
             }
```

```
            else
                Resp= 0;
    if (Resp)

    {
        Operandos.Pop(&Resultado);
        cout<<"\n\nLa expresión en notación postfija fue evaluada:
            ➥"<<Resultado<<"\n\n";
    }
    else
        cout<<"\n\nLa expresión dada es incorrecta.\n\n";
}
```

6.3 Listas circulares simplemente ligadas

Una ***lista circular simplemente ligada*** es una lista en la cual el nodo que sigue al último es el primero. Es decir, el último nodo tiene como sucesor al primero de la lista, logrando con ello tener acceso nuevamente a todos los miembros de la lista. Esta característica permite que desde cualquier nodo de esta estructura de datos se tenga acceso a cualquiera de los otros nodos de la misma. La figura 6.12 presenta un esquema de una lista circular simplemente ligada.

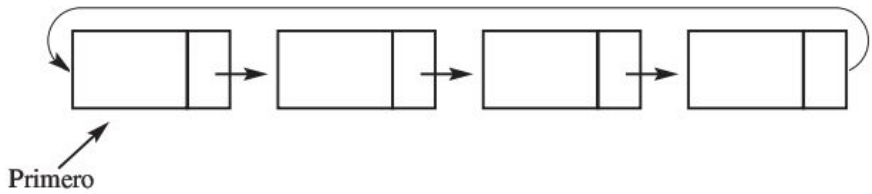


FIGURA 6.12 *Lista circular simplemente ligada*

La clase correspondiente a esta estructura es similar a la de la lista simplemente ligada presentada antes, sólo cambian algunos métodos debido a que con esta variante se puede tener acceso a todos los nodos a partir de cualquiera de ellos. La variable tipo puntero `Primero` deja de ser imprescindible para garantizar el recorrido de toda la lista. Sin embargo, resulta necesaria para evitar caer en ciclos infinitos (permite guardar una referencia del nodo desde el cual se empezó a recorrer la lis-

ta). Por ejemplo, en la operación de búsqueda, si el elemento buscado no está en la lista, se debe recordar a partir de qué nodo se inició el recorrido. Una manera de hacer referencia al inicio de la lista (sin usar un puntero al primer nodo) es definiendo un nodo especial, llamado nodo de cabecera. Un nodo de cabecera no guarda información útil, sino que se usa sólo para indicar el inicio de la lista.

Debido a que los métodos resultan similares a los presentados en la clase `Lista`, su diseño e implementación dependen de cada desarrollador.

6.4 Listas doblemente ligadas

Las ***listas doblemente ligadas*** son otra variante de las estructuras vistas en las secciones previas. En las listas simplemente ligadas cada nodo conoce solamente la dirección de su nodo sucesor. De ahí la importancia de no perder el puntero al primer nodo de la misma. Por su parte, en las listas doblemente ligadas, cada nodo conoce la dirección de su predecesor y de su sucesor. La excepción es el primer nodo de la lista que no cuenta con predecesor, y el último que no tiene sucesor. Debido a esta característica, se puede visitar a todos los componentes de la lista a partir de cualquiera de ellos.

La figura 6.13 presenta el esquema del nodo de una lista doblemente ligada. Observe que el nodo tiene tres partes, dos de ellas dedicadas al almacenamiento de direcciones (del nodo predecesor y del nodo sucesor) y la tercera para guardar la información.

FIGURA 6.13 *Estructura del nodo de una lista doblemente ligada*

La figura 6.14 presenta, de manera gráfica, una lista doblemente ligada. Se tiene un puntero al primer nodo de la lista (éste no tiene predecesor) y uno al último

(éste no tiene sucesor). Sin embargo, es importante destacar que se pueden recorrer todos los nodos de la lista a partir de cualquiera de ellos.

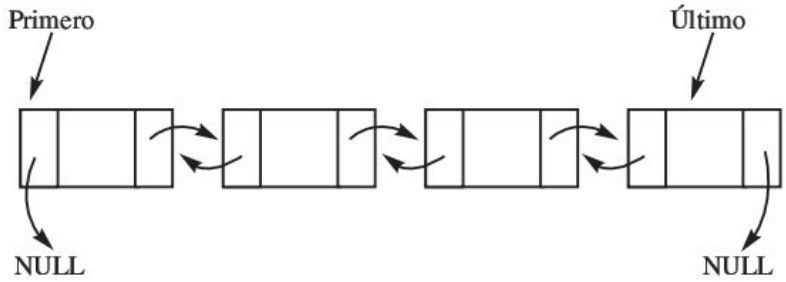


FIGURA 6.14 *Lista doblemente ligada*

Las figuras 6.15 y 6.16 presentan las plantillas de la clase `NodoDobleLiga` y de la clase `ListaDobLig` respectivamente. Se usan plantillas para dar mayor generalidad a la solución. La clase `NodoDobleLiga` tiene tres atributos, uno representa la información a almacenar por lo que se define de tipo `T`, y los otros dos representan la dirección de otro nodo por lo que se definen como punteros a un objeto de la misma clase. Por su parte, la clase `ListaDobLig` tiene dos atributos que representan la dirección del primero y del último nodos de la misma, por lo cual son de tipo puntero a un objeto de tipo `NodoDobleLiga`.

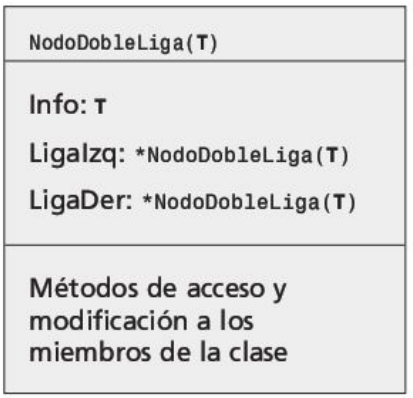


FIGURA 6.15 *Clase* `NodoDobleLiga`

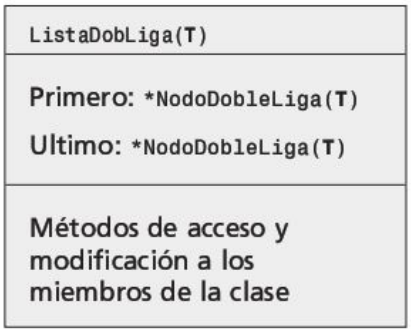


FIGURA 6.16 *Clase* `ListaDobLiga`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a las plantillas de las clases `NodoDobleLiga` y `ListaDobLiga`. La clase lista incluye dos punteros, uno al primer nodo y otro al último. Esto es con el fin de facilitar algunas operaciones, lo cual se verá con mayor detalle en las siguientes secciones.

```
/* Prototipo de la plantilla de la clase ListaDobLiga. Así, en la clase
➥NodoDobleLiga se podrá hacer referencia a ella. */
template <class  T>
class  ListaDobLiga;


/* Definición de la plantilla de la clase NodoDobleLiga. La clase Lista-
➥DobLiga se declara como una clase amiga para que pueda tener acceso a
➥los miembros privados de esta clase. */
template <class  T>
class  NodoDobleLiga
{
   private:
        NodoDobleLiga<T>  * LigaIzq;
        NodoDobleLiga<T>  * LigaDer;
        T Info;
   public:
        NodoDobleLiga();
        friend  class  ListaDobLiga<T>;
};

/* Método constructor. Inicializa los punteros con el valor NULL. */
template <class  T>
NodoDobleLiga<T>::NodoDobleLiga()
```

```
{
   LigaIzq= NULL;
   LigaDer= NULL;
}

/* Definición de la plantilla de la clase ListaDobLiga. Esta clase
➥tiene dos atributos que son punteros al primero y último elementos de la
➥misma. */
template <class  T>
class  ListaDobLiga
{
   private:
        NodoDobleLiga<T>  * Primero;
        NodoDobleLiga<T>  * Ultimo;
   public:
        ListaDobLiga ();

        /* En esta sección se incluyen los métodos de modificación y
        acceso a los miembros de la clase. */

};
/* Declaración del método constructor. Inicializa el apuntador al primero
➥y al último elementos con el valor de NULL, indicando lista vacía. */
template <class  T>
ListaDobLiga<T>::ListaDobLiga()
{
   Primero= NULL;
   Ultimo= NULL;
}
```

Las operaciones básicas que se pueden realizar en una lista previamente generada son: inserción, eliminación y búsqueda. Su creación se puede considerar también una operación básica. Es importante señalar que tener un puntero al final de la lista permite el acceso a este elemento de manera directa evitando el recorrido de los nodos previos. A continuación se analizan las principales operaciones. Las variantes de una misma operación se deben principalmente a la posición dentro de la lista donde se lleve a cabo ésta.

6.4.1 Inserción en listas doblemente ligadas

La operación de ***inserción*** de un nuevo nodo a una lista consiste en tomar un espacio de memoria dinámicamente, asignarle la información correspondiente y ligarlo al o a los nodos que corresponda dentro de la lista. Los pasos varían dependiendo de la posición que ocupará el nuevo elemento.

Inserción al principio de la lista

La figura 6.17 presenta un esquema de la inserción de un nuevo elemento al inicio de la lista. Se crea un nodo, cuya dirección se guarda en una variable auxiliar llamada `Apunt`, a su liga derecha se le asigna la dirección del primer nodo y a la izquierda el valor de **NULL**. Además, se establece la liga entre el nodo que ocupaba la primera posición de la lista con el nuevo nodo. Por último, se redefine el `Primero` con el valor de `Apunt`.

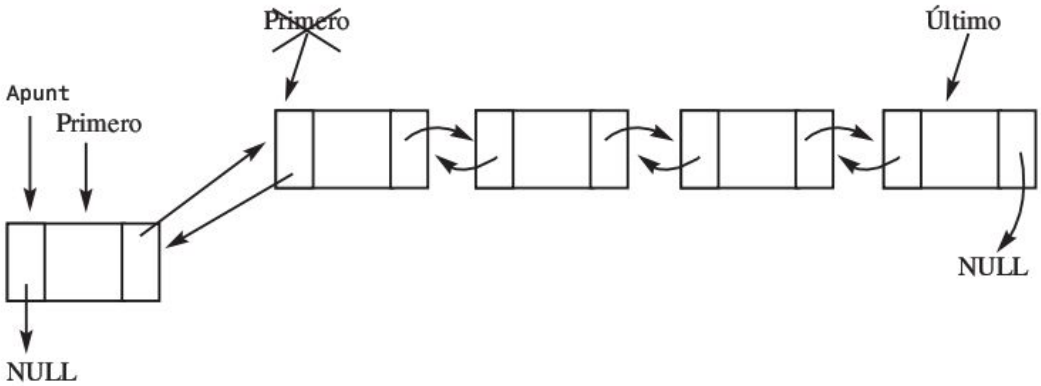


FIGURA 6.17 *Inserción al principio de la lista*

El método para llevar a cabo esta operación es el siguiente:

```
/* Método que inserta un nuevo nodo al inicio de la lista doblemente liga-
➥da. Recibe como parámetro la información a almacenar en el nodo. */
template <class T>
void ListaDobLiga<T>::InsertaInicio(T Dato)
{
   NodoDobleLiga<T>  * Apunt;
   Apunt= new  NodoDobleLiga<T>();
   Apunt->Info= Dato;
   Apunt->LigaDer= Primero;
   if (Primero)
       Primero->LigaIzq= Apunt;
   else
   Ultimo= Apunt;
   Primero= Apunt;
}
```

El método también considera si la lista está vacía. En este caso, el puntero al último elemento (`Ultimo`) se redefine con el valor de la dirección del nuevo nodo.

Inserción al final de la lista

La figura 6.18 presenta gráficamente la secuencia de pasos necesarios para insertar un nuevo elemento al final de la lista. Para ello, se crea un nuevo nodo (cuya dirección se guarda en `Apunt`) el cual se liga con el último nodo de la lista. Por último, se redefine el valor del puntero al último elemento (`Ultimo`) con la dirección del nuevo nodo.

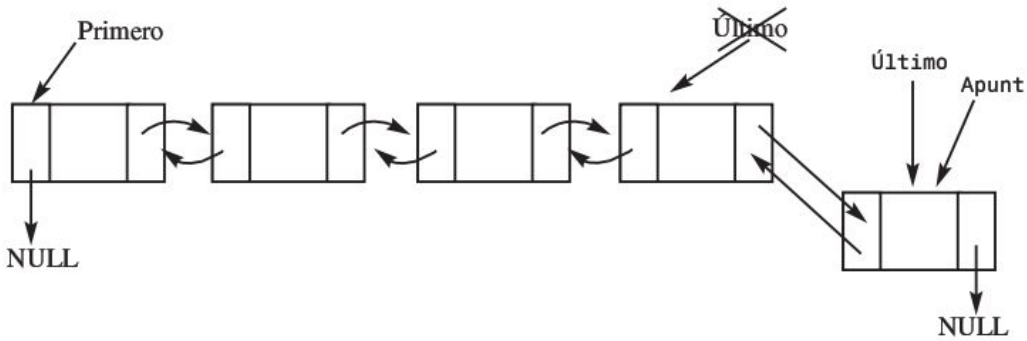


FIGURA 6.18 *Inserción al final de la lista*

A continuación se presenta el método que implementa esta operación.

```
/* Método que inserta un nuevo nodo al final de la lista doblemente ligada.
➥Recibe como parámetro (Dato) el valor a guardar en el nuevo nodo. */
template <class  T>
void ListaDobLiga<T>::InsertaFinal(T Dato)
{
   NodoDobleLiga<T>  * Apunt;
   Apunt= new  NodoDobleLiga<T>;
   Apunt->Info= Dato;
   Apunt->LigaIzq= Ultimo;
   if (Ultimo)
       Ultimo->LigaDer= Apunt;
   else
       Primero= Apunt;
   Ultimo= Apunt;
}
```

El método también considera si la lista está vacía. En este caso, el puntero al primer elemento (`Primero`) se redefine con el valor de la dirección del nuevo nodo.

Inserción formando una lista ordenada

La figura 6.19 presenta un esquema de la inserción de un nuevo elemento en la lista, de tal manera que la misma va quedando ordenada de manera creciente. La posición para el nuevo dato puede ser la primera (si es más pequeño que el dato almacenado en el primer nodo), la última (si es más grande que el dato almacenado en el último nodo) o una intermedia. Si fuera este último caso, se debe encontrar la posición del nodo cuya información es mayor que la del nodo a insertar. Una vez encontrado (`Apun2`), y creado el nuevo nodo (`Apun1`), se establecen las ligas correspondientes entre el nuevo nodo y los que se convertirán en su antecesor y predecesor.

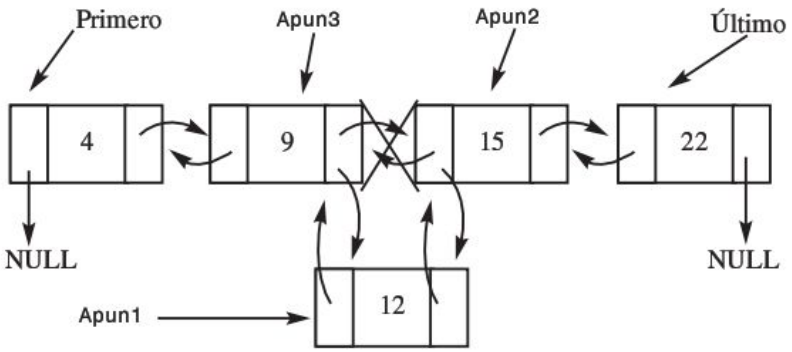


FIGURA 6.19 *Inserción formando una lista ordenada*

El método, codificado en ***C++***, para llevar a cabo esta operación es el siguiente:

```
/* Método que inserta un nuevo nodo en la lista, de tal manera que los
➥elementos de la misma vayan quedando ordenados de menor a mayor. Recibe
➥como parámetro la información a guardar en el nuevo nodo. */
template <class  T>
void ListaDobLiga<T>::InsertaOrdenado(T Dato)
{
   NodoDobleLiga<T>  * Apun1, *Apun2, *Apun3;
```

```
    /* Si la lista está vacía o si el valor a insertar es más pequeño que
    ➥el contenido del primer nodo, entonces se invoca al método que
    ➥inserta al inicio de la lista. */
    if (!Primero || Dato < Primero->Info)
        InsertaInicio(Dato);
    else
        /* Si el dato a insertar es más grande que el contenido del
        ➥último nodo, entonces se invoca al método que inserta al final
        ➥de la lista. */
        if (Dato > Ultimo->Info)
            InsertaFinal(Dato);
        else
        {
            Apun1= new  NodoDobleLiga<T>;
            Apun1->Info= Dato;
            Apun2= Primero;
            while  (Apun2->Info < Apun1->Info)
               Apun2= Apun2->LigaDer;
            Apun3= Apun2->LigaIzq;
            Apun3->LigaDer= Apun1;
            Apun1->LigaDer= Apun2;
            Apun1->LigaIzq= Apun3;
            Apun2->LigaIzq= Apun1;
        }
}
```

6.4.2 Eliminación en listas doblemente ligadas

La operación de ***eliminación*** de un nodo de una lista consiste en encontrar el valor a quitar, establecer las ligas correspondientes entre el nodo que le precede y el que le sucede, y finalmente liberar la sección de memoria ocupada por el nodo en cuestión. Se pueden presentar algunas variantes según la posición que el elemento tenga en la lista. A continuación se explican los principales casos.

Eliminación del primer elemento de la lista

La figura 6.20 presenta gráficamente la secuencia de pasos correspondientes a esta operación. El nodo a eliminar debe ser apuntado por una variable auxiliar, en este caso llamada `Apunt`, luego se debe redefinir el `Primero` con la dirección de su sucesor y finalmente se libera la porción de memoria ocupada por el dato que se quitó de la lista.

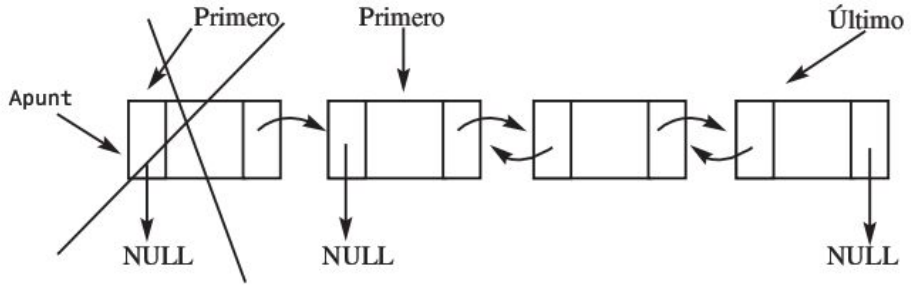


FIGURA 6.20 *Eliminación del primer elemento de la lista*

El método definido para implementar este tipo de eliminación en una lista doblemente ligada es el siguiente.

```
/* Método que elimina el primer elemento de la lista doblemente ligada.
➥Regresa 1 si la lista tiene al menos un elemento y 0 en caso contrario. */
template <class  T>
int ListaDobLiga<T>::EliminaPrimero()
{
   NodoDobleLiga<T>  * Apunt;
   int Resp= 1;

   /* Verifica si la lista tiene al menos un elemento.*/
   if (Primero)
   {
       Apunt= Primero;
       if (Apunt->LigaDer)
       {
           Primero= Apunt->LigaDer;
           Primero->LigaIzq= NULL;
       }
       else
       {
           /* La lista tiene sólo un elemento, por lo tanto luego de la
           ➥eliminación queda vacía. */
           Primero= NULL;
           Ultimo= NULL;
       }
       delete(Apunt);
   }
   else
       Resp= 0;
   return  Resp;
}
```

El método visto contempla el caso de que la lista tenga un solo elemento. Luego de la eliminación, la lista queda vacía.

Eliminación del último elemento de la lista

La figura 6.21 presenta el esquema correspondiente a esta operación. El nodo a eliminar debe ser apuntado por una variable auxiliar, en este caso llamada `Apunt`, luego se debe redefinir el `Ultimo` con la dirección de su predecesor y finalmente se libera la porción de memoria ocupada por el nodo.

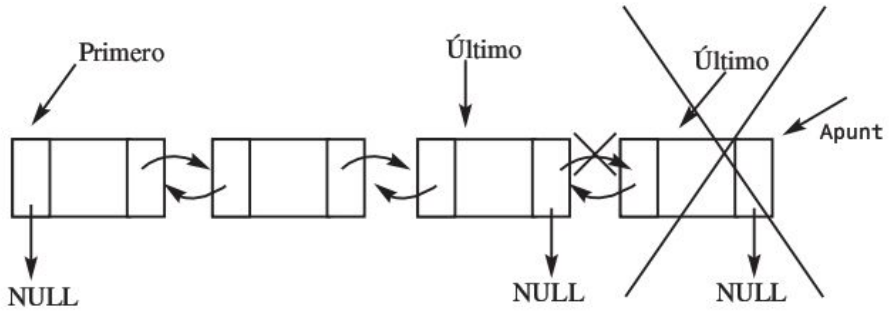


FIGURA 6.21 *Eliminación del último elemento de la lista*

A continuación se presenta el método que implementa los pasos requeridos para quitar al último nodo de una lista.

```
/* Método que elimina el último elemento de una lista doblemente ligada.
➥Regresa 1 si la lista tiene al menos un elemento y 0 en caso contrario. */
template <class  T>
int ListaDobLiga<T>::EliminaUltimo()
{
   NodoDobleLiga<T>  * Apunt;
   int Resp= 1;

   /* Verifica si la lista tiene al menos un elemento.*/
   if (Ultimo)
   {
       Apunt= Ultimo;
       if (Apunt->LigaIzq)
       {
            Ultimo= Apunt->LigaIzq;
            Ultimo->LigaDer= NULL;
       }
```

```
        else
        {
        /* La lista tiene sólo un elemento, por lo tanto luego de la
        ➥eliminación queda vacía. */
        Primero= NULL;
        Ultimo= NULL;
      }
      delete(Apunt);
    }
    else
        Resp= 0;
    return  Resp;
}
```

El método visto contempla el caso de que la lista tenga un solo elemento. Luego de la eliminación, la lista queda vacía.

Eliminación de un elemento de la lista

La figura 6.22 muestra de manera gráfica cómo se lleva a cabo esta operación. Primero se debe buscar el nodo cuyo contenido sea igual al dato. Si se encuentra, se guarda su dirección en una variable auxiliar (Apun1) y se establecen las ligas correspondientes entre su nodo predecesor (Apun2) y su nodo sucesor (Apun3). Finalmente se libera la memoria ocupada por el nodo. El nodo a eliminar puede ocupar cualquier posición en la lista, ser el primero, el último o estar en una posición intermedia. El siguiente esquema representa el caso de un nodo intermedio.

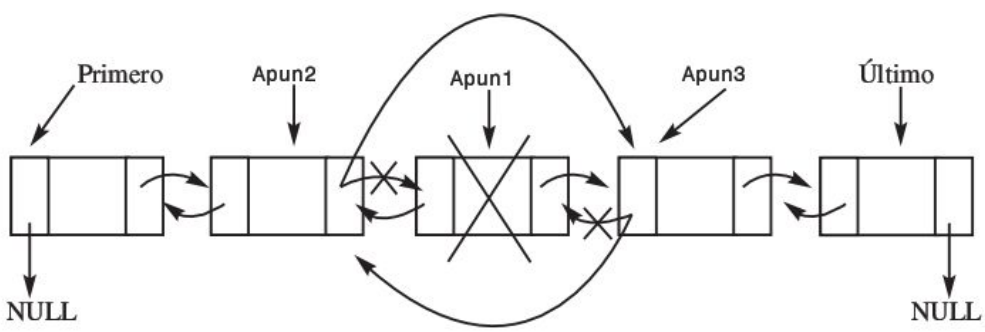


FIGURA 6.22 *Eliminación de un elemento de la lista*

El método para llevar a cabo esta operación se implementa de la siguiente manera.

```
/* Método que elimina un nodo cuya información es igual a Dato.
➥Regresa 1 si la operación se llevó a cabo, 0 si el elemento no está
➥en la lista y -1 si la lista está vacía. */
template <class  T>
int ListaDobLiga<T>::EliminaUnNodo(T Dato)
{
   NodoDobleLiga<T>  * Apun1,*Apun2,*Apun3;
   int Resp= 1;

   /* Verifica si la lista tiene al menos un elemento.*/
   if (Primero)
   {
        Apun1= Primero;
        while  ((Apun1!= NULL) && (Apun1->Info != Dato))
            Apun1= Apun1->LigaDer;
        if (Apun1 == NULL)
            Resp= 0;
        else
        {
            /* Verifica si hay sólo un elemento en la lista. Si es así,
            ➥entonces la lista quedará vacía luego de la eliminación. */
            if (Primero == Ultimo)
            {
                Primero= NULL;
                Ultimo= NULL;
            }
            else
                /* Verifica si el elemento a eliminar es el primero de
                ➥la lista. */
                if (Apun1 == Primero)
                {
                    Primero= Apun1->LigaDer;
                    Primero->LigaIzq= NULL;
                }
                else
                    /* Verifica si el elemento a eliminar es el último
                    ➥de la lista. */
                    if (Apun1 == Ultimo)
                    {
                        Ultimo= Apun1->LigaIzq;
                        Ultimo->LigaDer= NULL;
                    }
```

```
                    else
                    {
                        Apun2= Apun1->LigaIzq;
                        Apun3= Apun1->LigaDer;
                        Apun2->LigaDer= Apun3;
                        Apun3->LigaIzq= Apun2;
                    }
                delete(Apun1);
            }
        }
        else
            Resp= -1;
        return  Resp;
    }
```

El método presentado implementa los pasos de la eliminación, contemplando todos los casos que se pudieran presentar. Es importante destacar que para los casos en los cuales se elimina el primero o el último, se hubieran podido reutilizar los métodos ya analizados.

Otra consideración que se debe hacer, es que al tener cada nodo la dirección de su sucesor y su predecesor ya no es necesario guardar la dirección del anterior, como se hace en las listas simplemente ligadas. Una vez encontrado el nodo a eliminar, si éste ocupara una posición intermedia, se establecerían las nuevas ligas entre predecesor y sucesor de manera directa.

6.4.3 Búsqueda de elementos en listas doblemente ligadas

La operación de ***búsqueda*** de un dato entre los elementos de una lista consiste en recorrer la lista de izquierda a derecha (a partir del primer elemento) o de derecha a izquierda (a partir del último elemento) hasta encontrar el dato buscado o hasta que ya no queden nodos por visitar. Si los elementos de la lista siguen algún orden, entonces habrá que tenerlo en cuenta en el momento de hacer la comparación entre el dato buscado y la información del nodo visitado. A continuación se presenta un método que permite buscar un valor dentro de una lista. Este método realiza la búsqueda de izquierda a derecha y considera que la información de los nodos está desordenada.

```
/* Método que busca en la lista un dato dado. El método recibe como
➥parámetro el elemento a buscar (Dato) y una variable (Apunt) que
➥almacena la dirección de un nodo (la primera vez es la dirección del
➥primero). Regresa como resultado la dirección del nodo, si lo encuentra,
➥o NULL en caso contrario. */
template <class  T>
NodoDobleLiga<T> * ListaDobLiga<T>::Busca(T Dato, NodoDobleLiga<T>
*Apunt)
{
   if (Apunt)
       if (Apunt->Info == Dato)
           return Apunt;
       else
           return Busca (Dato, Apunt->LigaDer);
   else
       return NULL;
}
```

El programa 6.5 presenta la clase correspondiente a una estructura tipo lista doblemente ligada con los métodos más usados, algunos de los cuales ya fueron analizados. En algunos casos, por razones de espacio, sólo se incluyen los prototipos y los encabezados. Se establecen dos punteros para el manejo de la lista: un puntero al inicio y otro al final. Los métodos fueron definidos considerando estos punteros. Si se quiere trabajar con un solo puntero (como en el programa 6.1) será necesario adaptar la definición de la clase y la de algunos métodos. Es importante mencionar que los cambios a realizar son mínimos.

Programa 6.5

```
/* Prototipo de la plantilla de la clase ListaDobLiga. Así, en la clase
➥NodoDobleLiga se podrá hacer referencia a ella. */
template <class  T>
class  ListaDobLiga;


/* Definición de la plantilla de la clase NodoDobleLiga. La clase Lista-
➥DobLiga se declara como una clase amiga para que pueda tener acceso a
➥los miembros privados de la clase NodoDobleLiga. */
template <class  T>
class  NodoDobleLiga
```

```
{
   private:
       NodoDobleLiga<T>  * LigaIzq;
       NodoDobleLiga<T>  * LigaDer;
       T Info;
   public:
       NodoDobleLiga();
       T RegresaInfo();
       friend  class  ListaDobLiga<T>;
};

/* Método constructor. Inicializa los punteros con el valor NULL. */
template <class  T>
NodoDobleLiga<T>::NodoDobleLiga()
{
   LigaIzq= NULL;
   LigaDer= NULL;
}

/* Método que regresa el valor de Info, permitiendo que usuarios
➥externos a la clase tengan acceso a él sin poder para modificarlo. */
template <class  T>
T NodoDobleLiga<T>::RegresaInfo()
{
   return Info;
}

/* Definición de la plantilla de la clase ListaDobLiga. Esta clase tiene
➥dos atributos que son los punteros al primero y último elementos de la
➥misma. */
template <class  T>
class  ListaDobLiga
{
   private:
       NodoDobleLiga<T>  * Primero;
       NodoDobleLiga<T>  * Ultimo;
   public:
       ListaDobLiga ();
       void ImprimeIzq_Der(NodoDobleLiga<T>*);
       void ImprimeDer_Izq(NodoDobleLiga<T>*);
       void ImprimeNodo(NodoDobleLiga<T>*);
       void InsertaInicio(T);
       void InsertaFinal(T);
       void InsertaOrdenado(T);
       int InsertaAntes(T,T);
       int EliminaPrimero();
       int EliminaUltimo();
       int EliminaUnNodo(T);
       int EliminaAnterior(T);
```

```
        NodoDobleLiga<T>  * Busca(T, NodoDobleLiga<T>*);
        NodoDobleLiga<T>  * RegresaPrimero();
        NodoDobleLiga<T>  * RegresaUltimo();
        NodoDobleLiga<T>  * RegresaVecinoDer(NodoDobleLiga<T>  * );
};

/* Declaración del método constructor. Inicializa el apuntador al primero
➥y al último elementos con el valor de NULL, indicando lista vacía. */
template <class  T>
ListaDobLiga<T>::ListaDobLiga()
{
   Primero= NULL;
   Ultimo= NULL;
}

/* Método que imprime la información almacenada en cada uno de los nodos
➥de la lista, empezando por el primer nodo. La primera vez, recibe como
➥parámetro el valor almacenado en Primero. */
template <class  T>
void ListaDobLiga<T>::ImprimeIzq_Der(NodoDobleLiga<T>* Apunt)
{
   if (Apunt)
   {
       cout<<Apunt->Info<< '\n';
       ImprimeIzq_Der(Apunt->LigaDer);
   }
}

/* Método que imprime la información almacenada en cada uno de los nodos
➥de la lista, empezando por el último nodo. La primera vez, recibe como
➥parámetro el valor almacenado en Ultimo. */
template <class  T>
void ListaDobLiga<T>::ImprimeDer_Izq(NodoDobleLiga<T>* Apunt)
{
   if (Apunt)
   {
       cout<< Apunt->Info<< '\n';
       ImprimeDer_Izq(Apunt->LigaIzq);
   }
}

/* Método que imprime la información almacenada en uno de los nodos de
➥la lista, cuya dirección se recibe como parámetro. */
template <class  T>
void ListaDobLiga<T>::ImprimeNodo(NodoDobleLiga<T>* Apunt)
{
   cout<<Apunt->Info<< '\n';
}
```

```
/* Método que inserta un nuevo nodo con la información de Dato al inicio
➥de la lista doblemente ligada. */
template <class  T>
void ListaDobLiga<T>::InsertaInicio(T Dato)
{
   /* Presentado más arriba. */
}

/* Método que inserta un nuevo nodo con la información de Dato al final
➥de la lista doblemente ligada. */
template <class  T>
void ListaDobLiga<T>::InsertaFinal(T Dato)
{
   /* Presentado más arriba. */
}

/* Método que inserta un nuevo nodo con la información de Dato, de
➥manera que los elementos de la lista vayan quedando ordenados de menor
➥a mayor. */
template <class  T>
void ListaDobLiga<T>::InsertaOrdenado(T Dato)
{
   /* Presentado más arriba. */
}

/* Método que inserta un nuevo nodo con la información de Dato antes de
➥un nodo dado como referencia, cuya información está en Ref. Regresa 1
➥si encuentra la referencia y puede llevar a cabo la inserción, 0 si no
➥encuentra la referencia y -1 si la lista está vacía. */
template <class  T>
int ListaDobLiga<T>::InsertaAntes(T Dato, T Ref)
{
   NodoDobleLiga<T>  * Apun1,*Apun2,*Apun3;
   int Resp= 1;

   if (Primero)
   {
        Apun1= Primero;
        while  ((Apun1 != NULL) && (Apun1->Info != Ref))
           Apun1= Apun1->LigaDer;
        /*Verifica si encontró la información dada como referencia. */
        if (Apun1 != NULL)
        {
             Apun3= new  NodoDobleLiga<T>();
             Apun3->Info= Dato;
             Apun3->LigaDer= Apun1;
             Apun2= Apun1->LigaIzq;
             Apun1->LigaIzq= Apun3;
             Apun3->LigaIzq= Apun2;
```

```
            if (Primero == Apun1)
                Primero= Apun3;
            else
                Apun2->LigaDer= Apun3;
        }
        else
            Resp= 0;
    }
    else
        Resp= -1;
    return  Resp;
}

/* Método que elimina el primer elemento de la lista doblemente ligada.
➥Regresa 1 si la lista tiene al menos un elemento y 0 en caso contrario. */
template <class  T>
int ListaDobLiga<T>::EliminaPrimero()
{
   /* Presentado más arriba. */
}

/* Método que elimina el último elemento de la lista doblemente ligada.
➥Regresa 1 si la lista tiene al menos un elemento y 0 en caso contrario. */
template <class  T>
int ListaDobLiga<T>::EliminaUltimo()
{
/* Presentado más arriba. */
}

/* Método que elimina un nodo cuya información es igual a Dato. Regresa
➥1 si la eliminación se puede llevar a cabo, 0 si el elemento no está en
➥la lista y -1 si la lista está vacía. */
template <class  T>
int ListaDobLiga<T>::EliminaUnNodo(T Dato)
{
   /* Presentado más arriba. */
}

/* Método que elimina el nodo anterior al nodo que contiene la infor-
➥mación Dato. Regresa 1 si la eliminación se puede llevar a cabo, 0 si
➥el valor dado como referencia no está en la lista, -1 si la referencia
➥es el primer nodo y por lo tanto no hay anterior, y -2 si la lista
➥está vacía. */
template <class  T>
int ListaDobLiga<T>::EliminaAnterior(T Dato)
{
   NodoDobleLiga<T>  * Apun1, *Apun2, *Apun3;
   int Resp= 1;
   if (Primero)
```

```
    {
         Apun1= Primero;
         while  ((Apun1 != NULL) && (Apun1->Info != Dato))
            Apun1= Apun1->LigaDer;
         if (Apun1 == NULL)
            Resp= 0;
         else
         /* Verifica si la información dada como referencia está en el
         ➥primer nodo. */
         if (Primero == Apun1)
             Resp= -1;
         else
         {
             if (Primero == Apun1->LigaIzq)
             {
                  Apun2= Primero;
                  Primero= Apun1;
                  Primero->LigaIzq= NULL;
             }
             else
             {
                  Apun2= Apun1->LigaIzq;
                  Apun3= Apun2->LigaIzq;
                  Apun3->LigaDer= Apun1;
                  Apun1->LigaIzq= Apun3;
             }
                  delete(Apun2);
         }
    }
    else
         Resp= -2;
    return  Resp;
}

/* Método que busca en la lista un nodo dado como referencia. El método
➥recibe como parámetro el elemento a buscar (Dato) y una variable
➥(Apunt) que almacena la dirección de un nodo. La primera vez es la
➥dirección del primero. Regresa como resultado la dirección del nodo o
➥NULL si no lo encuentra. */
template <class T>
NodoDobleLiga<T> * ListaDobLiga<T>::Busca(T Dato, NodoDobleLiga<T>
*Apunt)
{
   /* Presentado más arriba. */
}

/* Método que regresa el valor del apuntador al primer elemento de la
➥lista. */
template <class  T>
NodoDobleLiga<T> * ListaDobLiga<T>::RegresaPrimero()
```

```
{
   return  Primero;
}

/* Método que regresa el valor del apuntador al último elemento de la
➥lista. */
template <class  T>
NodoDobleLiga<T> * ListaDobLiga<T>::RegresaUltimo()
{
   return  Ultimo;
}

/* Método que, dada la dirección de un nodo, regresa la dirección del
➥siguiente nodo a la derecha. Este método facilita el desplazamiento a
➥través de la lista por parte de usuarios externos a la misma. */
template <class  T>
NodoDobleLiga<T>  * ListaDobLiga<T>::RegresaVecinoDer(NodoDobleLiga<T>
*Apunt)
{
   return  Apunt->LigaDer;
}

/* Método que, dada la dirección de un nodo, regresa la dirección del
➥siguiente nodo a la izquierda. Este método facilita el desplazamiento a
➥través de la lista por parte de usuarios externos a la misma. */
template <class  T>
NodoDobleLiga<T>   * ListaDobLiga<T>::RegresaVecinoIzq(NodoDobleLiga<T>
*Apunt)
{
   return  Apunt->LigaIzq;
}
```

Es importante destacar que al usar punteros al primero y último nodos de la lista se simplifican algunas operaciones. Por ejemplo, en el caso de inserción y eliminación en la última posición, ya no es necesario recorrer toda la lista. Asimismo, se puede imprimir la lista en cualquiera de las dos direcciones sin tener que recurrir a operaciones auxiliares.

Dadas las características de los nodos que forman estas listas, se pudieron utilizar menos variables auxiliares en ciertas operaciones. En este libro se usaron con el fin de dar mayor claridad al código, sin embargo se pudo usar la notación de tal manera que se prescindiera de dichas variables. Se presenta un ejemplo a partir del código del método para eliminar un nodo cuya información se da como referencia. En el código que aparece en la columna izquierda se utilizan las variables auxiliares `Apun2` y `Apun3` para almacenar las direcciones del predecesor y del sucesor

del nodo a eliminar respectivamente. Posteriormente, al predecesor y al sucesor se les asignan las nuevas ligas. En el código de la columna derecha, la asignación de las nuevas ligas se hace de manera directa. Es decir, la primera línea indica que a la liga derecha del nodo que está siendo apuntado por la liga izquierda de `Apun1` se le asigna el valor de la liga derecha de `Apun1`. En la segunda línea se está expresando que, a la liga izquierda del nodo que está siendo apuntado por la liga derecha de `Apun1`, se le asigna el valor de la liga izquierda de `Apun1`.

```
Apun2= Apun1->LigaIzq;
Apun3= Apun1->LigaDer;
Apun2->LigaDer= Apun3;
Apun3->LigaIzq= Apun2;
```

```
Apun1->LigaIzq->LigaDer= Apun1->LigaDer;
Apun1->LigaDer->LigaIzq= Apun1->LigaIzq;
```

Se presenta un ejemplo de aplicación de las listas doblemente ligadas. Se incluye una biblioteca con la plantilla de la clase correspondiente al programa 6.5. También se incluye la biblioteca "*Productos.h*" presentada en el programa 6.2. En esta aplicación, se usa una lista doblemente ligada para almacenar y procesar datos de varios productos. Las opciones de trabajo ofrecidas al usuario son:

a) Dar de alta un producto.

b) Dar de baja un producto ya registrado.

c) Generar un reporte con los productos ordenados de menor a mayor, según la clave.

d) Generar un reporte con los productos ordenados de mayor a menor, según la clave.

e) Generar un reporte con los productos cuyos precios se encuentren comprendidos en un rango dado por el usuario.

f) Calcular e imprimir el promedio de los precios de todos los productos registrados.

Programa 6.6

```
/* Ejemplo de programa de aplicación de listas doblemente ligadas.
➥Se incluyen las bibliotecas que almacenan la plantilla de la clase
➥ListaDobLiga y la clase Producto, esta última usada como tipo de la
➥información guardada en los nodos de la lista.*/
```

```
#include "ListasDoblesLigas.h"
#include "Productos.h"


/* Función auxiliar que despliega las opciones de trabajo del usuario. */
int Menu()
{
   int Opcion;
   do {
        cout<<"\n\nBienvenido al sistema de inventario.\n\n";
        cout<<"Opciones de trabajo:";
        cout<<"\n   (1) Registrar un producto (se hará en orden según la
        ➥clave).";
        cout<<"\n   (2) Dar de baja un producto.";
        cout<<"\n   (3) Generar un reporte en orden creciente por claves.";
        cout<<"\n   (4) Generar un reporte en orden decreciente por
        ➥claves.";
        cout<<"\n   (5) Generar un reporte de productos cuyos precios
        estén en cierto rango.";
        cout<<"\n   (6) Promedio de precios de los productos
        ➥registrados.";
        cout<<"\n   (7) Salir.";
        cout<<"\n\nIngrese opción elegida: ";
        cin>>Opcion;
        cout<<"\n\n\n";
   } while  (Opcion < 1 || Opcion > 7);
   return  Opcion;
}


/* Función que genera un reporte con los datos de los productos en orden
➥creciente. Para ello, considerando el orden en el cual fue creada la
➥lista, la misma se recorre de izquierda a derecha empezando con el
➥primer nodo. */
void ReporteCrec(ListaDobLiga<Producto> Inventario)
{
   cout<<"\n\nLista de productos ordenados por clave de menor a
         ➥mayor.\n";
   Inventario.ImprimeIzq_Der(Inventario.RegresaPrimero());
}


/* Función que genera un reporte con los datos de los productos en orden
➥decreciente. Para ello, considerando el orden en el cual fue creada la
➥lista, la misma se recorre de derecha a izquierda empezando con el
➥último nodo. */
void ReporteDec(ListaDobLiga<Producto> Inventario)
```

```
{
   cout<<"\n\nLista de productos ordenados por clave de mayor a
         ➥menor.\n";
   Inventario.ImprimeDer_Izq(Inventario.RegresaUltimo());
}


/* Función que genera un reporte con los datos de los productos cuyos
➥precios se encuentran comprendidos en cierto rango. La función recibe
➥como parámetro la lista de productos. */
void ReportePrecios(ListaDobLiga<Producto> Inventario)
{
   NodoDobleLiga <Producto> *Apunt;
   double PrecInf, PrecSup;
   cout<<"\nIngrese el rango de precios que le interesa. \n";
   cout<<"Límite Inferior:  ";
   cin>>PrecInf;
   cout<<"\nLímite Superior:  ";
   cin>>PrecSup;
   Apunt= Inventario.RegresaPrimero();
   cout<<"\n\nLista de productos cuyos precios son >=  "<<PrecInf<<" y
         ➥<= "<<PrecSup<<"\n";
   while  (Apunt)
   {
      if ( Apunt->RegresaInfo().RegresaPrecio() >= PrecInf &&
           Apunt->RegresaInfo().RegresaPrecio() <= PrecSup )
               Inventario.ImprimeNodo(Apunt);
      Apunt= Inventario.RegresaVecinoDer(Apunt);
   }
}

/* Función que calcula el promedio de los precios de todos los productos
➥registrados en el inventario. Recibe como parámetro la lista y da como
➥resultado el promedio calculado.*/
double Promedio(ListaDobLiga<Producto>  Inventario)
{
   NodoDobleLiga <Producto>*Apunt;
   Apunt= Inventario.RegresaPrimero();
   double Prom= 0;
   int Total= 0;
   while  (Apunt)
   {
      Prom= Prom + Apunt->RegresaInfo().RegresaPrecio() ;
      Apunt= Inventario.RegresaVecinoDer(Apunt);
      Total= Total + 1;
   }
   if (Total)
       Prom= Prom/Total;
   return  Prom;
}
```

```
/* Función principal. Invoca los diferentes métodos para que las
➥operaciones elegidas por el usuario se puedan llevar a cabo. */
void main()
{
   ListaDobLiga<Producto> Inventario;
   Producto Produ;
   int Opcion, Clave, Resp;
   do {
       Opcion= Menu();
       /* Selección de acuerdo a la opción de trabajo elegida por el
       ➥usuario. */
       switch  (Opcion)
       {
          case 1:{
                  cout<<"\nIngrese datos del producto a registrar. ";
                  cin>>Produ;
                  Inventario.InsertaOrdenado(Produ);
                  break;
                 }
          case 2:{
                  cout<<"\nIngrese la clave del producto a eliminar. ";
                  cin>>Clave;
                  Producto Prod(Clave,"",0);
                  Resp= Inventario.EliminaUnNodo(Prod);
                  if (Resp == 1)
                       cout<<"\nBaja registrada.\n";
                  else
                      if (Resp == -1)
                           cout<<"\nNo hay productos registrados en
                                  ➥inventario. \n";
                      else
                           cout<<"\nNo hay producto registrado con la
                                  ➥clave dada. \n";
                  break;
                  }
          case 3: ReporteCrec(Inventario);
                  break;
          case 4: ReporteDec(Inventario);
                  break;
          case 5: ReportePrecios(Inventario);
                  break;
          case 6: cout<<"\n\n\nPromedio de precios:
                        ➥"<<Promedio(Inventario);
                  break;
          case 7: break;
          }
   } while  (Opcion >= 1 && Opcion  < 7);
}
```

6.5 Listas circulares doblemente ligadas

Las ***listas circulares doblemente ligadas*** son una variante de las presentadas en la sección anterior. En este tipo de listas, el primer nodo tiene como nodo predecesor al último y éste tiene como nodo sucesor al primero. Gráficamente, una lista circular doblemente ligada se representa como se muestra en la figura 6.23.

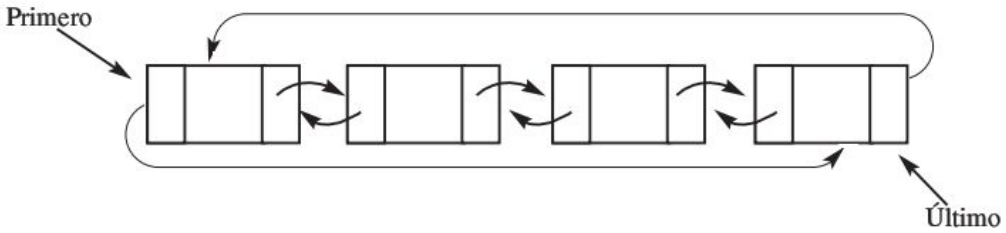


FIGURA 6.23 *Listas circulares doblemente ligadas*

Los métodos para llevar a cabo las operaciones sobre datos almacenados en este tipo de estructura son muy similares a los presentados en el programa 6.5. Se sugiere que defina la clase `ListaCirDobLig` (Lista Circular Doblemente Ligada) y adapte los métodos vistos para que se ajusten a las características de esta estructura de datos.

6.6 Multilistas

Las ***multilistas*** se pueden definir como listas de listas. Es decir, listas que tienen una lista como parte de la información que almacenan. Gráficamente, una multilista se representa como se muestra en la figura 6.24. En el caso de esta lista, cada uno de sus nodos guarda cierta información, un apuntador a una segunda lista y un apuntador al siguiente nodo.

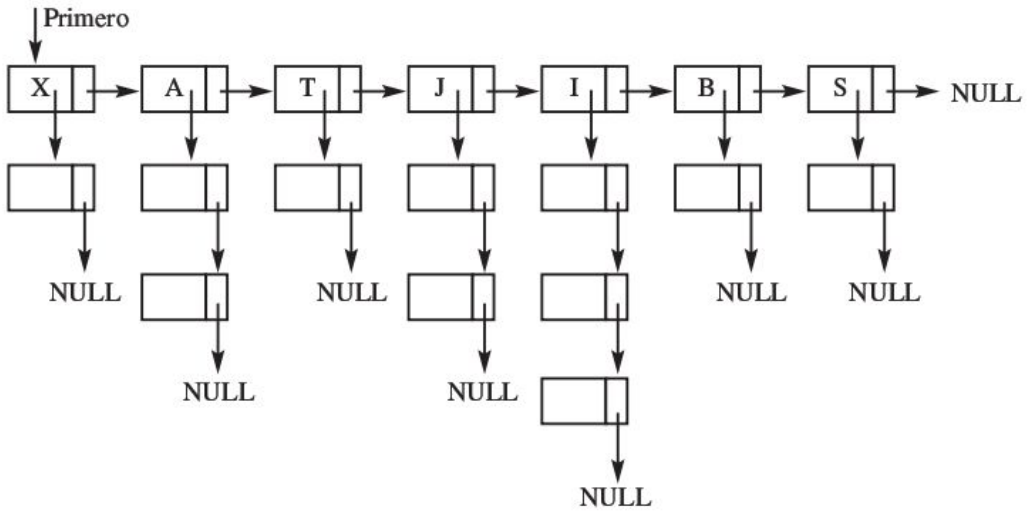


FIGURA 6.24 *Esquema de una multilista*

La anidación de listas puede hacerse en diferentes niveles. Retomando la lista de la figura 6.24, cada nodo de la segunda lista podría tener un apuntador a otra y así tantos niveles como sea necesario. Es importante considerar, que la representación del problema debe ser comprensible. Por lo tanto, se debe guardar un equilibrio entre la cantidad de listas que utilice y la claridad de la solución que se esté alcanzando. La figura 6.25 tiene tres niveles de listas. En el primero, cada nodo almacena cierta información y un apuntador a otro nodo del mismo tipo. Dentro de esa información, existe un apuntador a otra lista (la del segundo nivel). Por lo tanto, a las listas del segundo nivel (hay tantas como nodos haya en el primer nivel) se llega por medio de los nodos de la primera lista. A su vez, los nodos de la segunda lista contienen información y un apuntador a otro nodo de la misma lista. Dentro de su información hay un apuntador a una lista (la del tercer nivel). Habrá tantas listas en el tercer nivel, como nodos haya en cada una de las listas del segundo nivel, y el acceso a las mismas se da a través de dichos nodos.

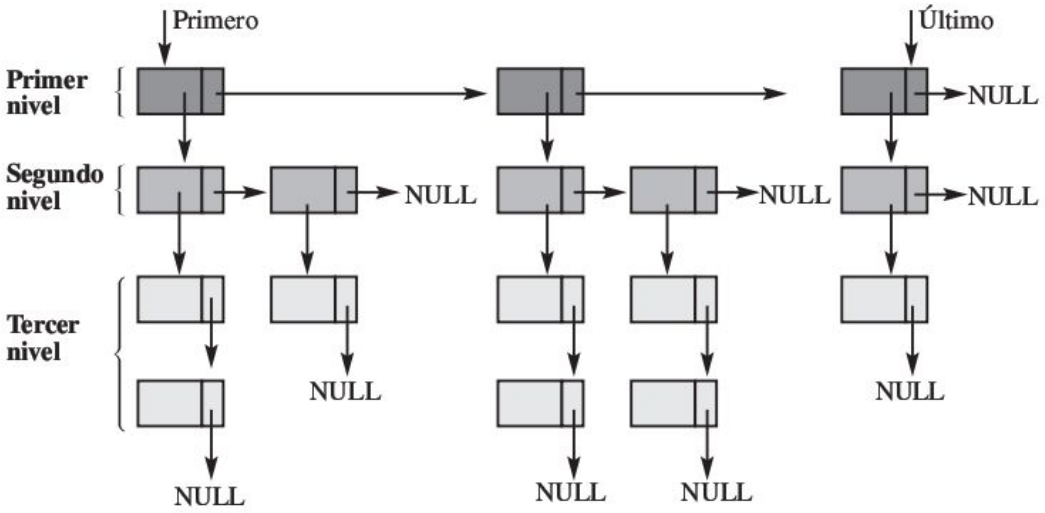


FIGURA 6.25 *Anidación de listas en tres niveles*

Existen dos formas para definir una estructura de datos con las características planteadas. En primer lugar se puede incluir como parte de la clase que dará valor a `T` en la clase `NodoLista`, un atributo que sea a su vez una lista (el programa 6.7 presenta un caso de este tipo). Otra manera de hacerlo es por medio de un atributo adicional en el nodo de la lista del primer nivel. Es decir, el nodo tendrá la información, el apuntador al siguiente nodo de la lista y un apuntador a otro tipo de nodo, el cual sería el primer elemento de la lista del segundo nivel.

La figura 6.26 presenta un ejemplo de multilista. Tiene una lista de autores y en cada uno de los nodos de esta lista incluye una lista de libros. De esta forma, define dos niveles de listas. El primero formado por todos los autores y el segundo por los libros escritos por dichos autores. El programa 6.7 muestra todas las clases involucradas y un ejemplo de aplicación. En la práctica, se sugiere el uso de bibliotecas para guardar las clases y de esta forma modularizar más la solución.

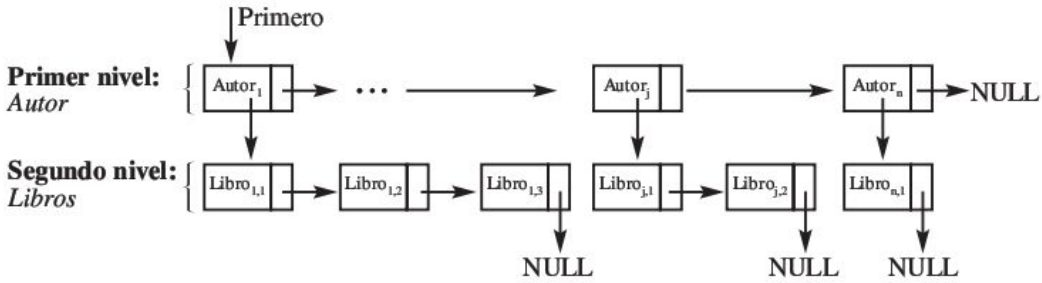


FIGURA 6.26 *Ejemplo de multilista*

Programa 6.7

```
/* Ejemplo de una multilista. Se declara una lista de autores, donde
➥cada nodo (como parte de la información) tiene una lista de libros. */

#define MAX 100

/* Declaración de la clase Libro. Se usará como base para el tipo de
➥información de las listas del segundo nivel. */
class  Libro
{
   private:
        char Nombre[MAX], ISBN[MAX];
        int AnioEdic;
   public:
        Libro();
        Libro(char [], char [], int);
        int operator == (Libro);
        int operator != (Libro);
        friend istream &operator>>(istream &Lee, Libro &);
        friend ostream &operator<< (ostream &Escribe, Libro &);
};

/* Declaración del método constructor por omisión. */
Libro::Libro()
{ }

/* Declaración del método constructor con parámetros. */
Libro::Libro(char Nom[], char Clave[], int AEd)
{
   strcpy(Nombre, Nom);
   strcpy(ISBN, Clave);
   AnioEdic= AEd;
}
```

```
/* Sobrecarga del operador == para comparar dos objetos de este tipo.
➥Regresa 1 si los libros tienen el mismo nombre y cero en caso
➥contrario. */
int Libro::operator == (Libro Lib)
{
   if (strcmp(Nombre, Lib.Nombre) == 0)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador != para comparar dos objetos de este tipo.
➥Regresa 1 si los libros tienen nombres distintos y cero en caso
➥contrario. */
int Libro::operator != (Libro Lib)
{
   if (strcmp(Nombre, Lib.Nombre) != 0)
        return 1;
   else
        return 0;
}

/* Declaración de la función amiga en la que se sobrecarga al operador
➥de lectura >>, de tal manera que objetos de tipo Libro puedan ser
➥leídos directamente. */
istream &operator>>(istream &Lee, Libro &Lib)
{
   cout<<"\n\nIngrese nombre del libro:";
   Lee>>Lib.Nombre;
   cout<<"\n\nIngrese ISBN:";
   Lee>>Lib.ISBN;
   cout<<"\n\nIngrese año de edición:";
   Lee>>Lib.AnioEdic;
   return  Lee;
}

/* Declaración de la función amiga en la que se sobrecarga al operador
➥de impresión <<, de tal manera que objetos de tipo Libro puedan ser
➥escritos directamente. */
ostream &operator<< (ostream &Escribe, Libro &Lib)
{
   Escribe<<"\n\n\nDatos del libro\n";
   Escribe<<"\nNombre:  "<<Lib.Nombre;
   Escribe<<"\nISBN:  "<<Lib.ISBN;
   Escribe<<"\nAño de edición:  "<<Lib.AnioEdic<<"\n";
   return  Escribe;
}
```

```
/* Declaración de la clase Autor. Se usará como tipo base para darle
➥valor a la T del atributo Info de los nodos de la lista del primer
➥nivel. El atributo Obra es un objeto de tipo Lista, en este caso es
➥una lista de libros (usando la clase previamente definida). */
class  Autor
{
   private:
       char Nombre[MAX], Nacional[MAX];
       int AnioNac;
       Lista<Libro> Obra;
   public:
       Autor();
       Autor(char[]);
       Lista<Libro> RegresaLisLibros();
       int operator == (Autor);
       int operator != (Autor);
       friend istream &operator>>(istream &Lee,  Autor &);
       friend ostream &operator<< (ostream &Escribe, Autor &);
};

/* Declaración del método constructor por omisión. */
Autor::Autor()
{ }

/* Declaración del método constructor con parámetros. */
Autor::Autor(char Nom[])
{
   strcpy(Nombre, Nom);
}

/* Método que regresa la lista de libros de un autor. */
Lista<Libro> Autor::RegresaLisLibros()
{
   return Obra;
}

/* Sobrecarga del operador == para poder comparar dos objetos de tipo
➥Autor. El método regresa 1 si los objetos tienen nombres iguales y 0 en
➥caso contrario.*/
int Autor::operator == (Autor Aut)
{
   if (strcmp(Nombre, Aut.Nombre) == 0)
       return 1;
   else
       return 0;
}
```

```
/* Sobrecarga del operador != para poder comparar dos objetos de tipo
➥Autor. El método regresa 1 si los objetos tienen nombres distintos y 0
➥en caso contrario.*/
int Autor::operator != (Autor Aut)
{
   if (strcmp(Nombre, Aut.Nombre) != 0)
        return 1;
   else
        return 0;
}

/* Declaración de la función amiga en la que se sobrecarga al operador
➥de lectura <<, de tal manera que objetos de tipo Autor puedan ser
➥leídos directamente. */
istream &operator>>(istream &Lee,  Autor &VarAut)
{
   cout<<"\n\nIngrese nombre del autor:";
   Lee>>VarAut.Nombre;
   cout<<"\n\nIngrese nacionalidad del autor:";
   Lee>>VarAut.Nacional;
   cout<<"\n\nIngrese año de nacimiento:";
   Lee>>VarAut.AnioNac;
   cout<<"\n\nIngrese los datos de su obra\n";
   VarAut.Obra.CreaInicio();
   return  Lee;
}

/* Declaración de la función amiga en la que se sobrecarga al operador
➥de impresión <<, de tal manera que objetos de tipo Autor puedan ser
➥escritos directamente. */
ostream &operator<< (ostream &Escribe, Autor &VarAut)
{
   Escribe<<"\n\nDatos del autor\n\n";
   Escribe<<"\nNombre:  "<<VarAut.Nombre;
   Escribe<<"\nNacionalidad:  "<<VarAut.Nacional;
   Escribe<<"\nAño de nacimiento:  "<<VarAut.AnioNac<<"\n";
   Escribe<<"\nDatos de su obra";
   VarAut.Obra.Imprime(VarAut.Obra.RegresaPrimero());
   return  Escribe;
}


/* Clase Lista dependiente de la clase NodoLista. */
template <class  T>
class  Lista;
```

```
/* Definición de la clase NodoLista. Se incluyeron sólo algunos de los
➥métodos vistos.*/
template <class  T>
class  NodoLista
{
   private:
       NodoLista<T> *Liga;
       T Info;
   public:
       NodoLista();
       T RegresaInfo();
       friend class  Lista<T>;
};

/* Declaración del método constructor por omisión. */
template <class  T>
NodoLista<T>::NodoLista()
{
   Liga= NULL;
}

/* Regresa la información almacenada en el nodo. */
template <class  T>
T NodoLista<T>::RegresaInfo()
{
   return Info;
}


/* Definición de la clase Lista. Es una lista simplemente ligada. */
template <class  T>
class  Lista
{
   private:
       NodoLista<T> *Primero;
   public:
       Lista ();
       NodoLista<T> *RegresaPrimero();
       void CreaInicio();
       void Imprime(NodoLista<T> *);
       void InsertaInicio(T);
       NodoLista<T> * Busca(T, NodoLista<T> *);
};

/* Declaración del método constructor. */
template <class  T>
Lista<T>::Lista()
{
   Primero= NULL;
}
```

```
/* Método que regresa la dirección del primer nodo de la lista. */
template <class  T>
NodoLista<T> *Lista<T>::RegresaPrimero()
{
   return Primero;
}

/* Método que crea una lista agregando el nuevo nodo al inicio de la
➥misma. */
template <class  T>
void Lista<T>::CreaInicio()
{
   NodoLista<T> *P;
   T Dato;
   char Resp;
   Primero= new NodoLista<T>();
   cout<<"Ingrese la información del primer elemento: \n";
   cin>> Dato;
   Primero->Info= Dato;
   cout<< "\n¿Desea ingresar otro elemento (S/N)? ";
   cin>>Resp;
   while (Resp == 'S' || Resp == 's')
   {
      cin>>Dato;
      P= new NodoLista<T>();
      P->Info= Dato;
      P->Liga= Primero;
      Primero= P;
      cout<< "\n¿Desea ingresar otro elemento (S/N)? ";
      cin>> Resp;
   }
}

/* Método que despliega el contenido de la lista. */
template <class  T>
void Lista<T>::Imprime(NodoLista<T> *P)
{
   if (P)
   {
      cout<<P->Info;
      Imprime(P->Liga);
   }
   cout<< '\n';
}

/* Método que inserta un nodo al inicio de la lista. */
template <class  T>
void Lista<T>::InsertaInicio(T Dato)
```

```
{
   NodoLista<T> *P;
   P= new NodoLista<T>();
   P->Info= Dato;
   P->Liga= Primero;
   Primero= P;
}

/* Método que busca un nodo dado como referencia en la lista. El método
➥recibe como parámetro el elemento a buscar y una variable que almacena
➥la dirección de un nodo, inicialmente es la dirección del primero.
➥Regresa como resultado la dirección del nodo si lo encuentra y NULL en
➥caso contrario. */
template <class  T>
NodoLista<T> * Lista<T>::Busca(T Dato, NodoLista<T> *Q)
{
   if (Q)
       if (Q->Info == Dato)
           return Q;
       else
           return Busca(Dato, Q->Liga);
   else
       return NULL;
}


/* Función auxiliar que despliega en pantalla las opciones de trabajo. */
int Menu()
{
   int Opc;
   do {
       cout<<"\n\nIngrese opción de trabajo\n";
       cout<<"\n(1) Agregar un nuevo autor. ";
       cout<<"\n(2) Generar un reporte de todos los autores con sus
       ➥obras. ";
       cout<<"\n(3) Generar un reporte con todos los datos de un cierto
       ➥autor. ";
       cout<<"\n(4) Generar un reporte con la obra de un cierto
       ➥autor. ";
       cout<<"\n(5) Terminar el proceso.";
       cout<<"\n\nIngrese la opción seleccionada: ";
       cin>>Opc;
   } while (Opc < 1 || Opc > 5);
   return Opc;
}
```

```
/* Función principal. De acuerdo a la opción de trabajo seleccionada por
➥el usuario se invoca a los métodos que corresponda. */
void main()
{
   int OpcTrab;
   char NomAut[MAX];
   NodoLista<Autor> * RespBus;
   Lista<Autor> Acervo;
   Lista<Libro> ObraAutor;
   Autor Escritor;

   Acervo.CreaInicio();
   do {
       OpcTrab= Menu();
       switch (OpcTrab)
       {
          case 1: { /* Se inserta un nuevo elemento en la lista de
                    ➥autores. */
                    cin>>Escritor;
                    Acervo.InsertaInicio(Escritor);
                    break;
                  }
          case 2: { /* Se imprime toda la lista. */
                    Acervo.Imprime(Acervo.RegresaPrimero());
                    break;
                  }
          case 3: { /* Se imprimen todos los datos de un autor, cuyo
                    ➥nombre proporciona el usuario. */
                    cout<<"\nIngrese nombre del autor: ";
                    cin>>NomAut;
                    Autor AutorAux(NomAut);
                    RespBus= Acervo.Busca(AutorAux,
                             ➥Acervo.RegresaPrimero());
                    if (RespBus)
                         cout<<RespBus->RegresaInfo();
                    else
                         cout<<"\n\nEse autor no está registrado. \n\n";
                    break;
                  }
          case 4: { /* Se imprimen los datos de todos los libros de un
                    ➥autor, cuyo nombre da el usuario. Se recupera el
                    ➥atributo que fue declarado como una lista, y a éste se
                    ➥le aplica el método de impresión de las listas. */
                    cout<<"\nIngrese nombre del autor: ";
                    cin>>NomAut;
                    Autor AutorAux(NomAut);
                    RespBus= Acervo.Busca(AutorAux,
                             ➥Acervo.RegresaPrimero());
                    if (RespBus)
```

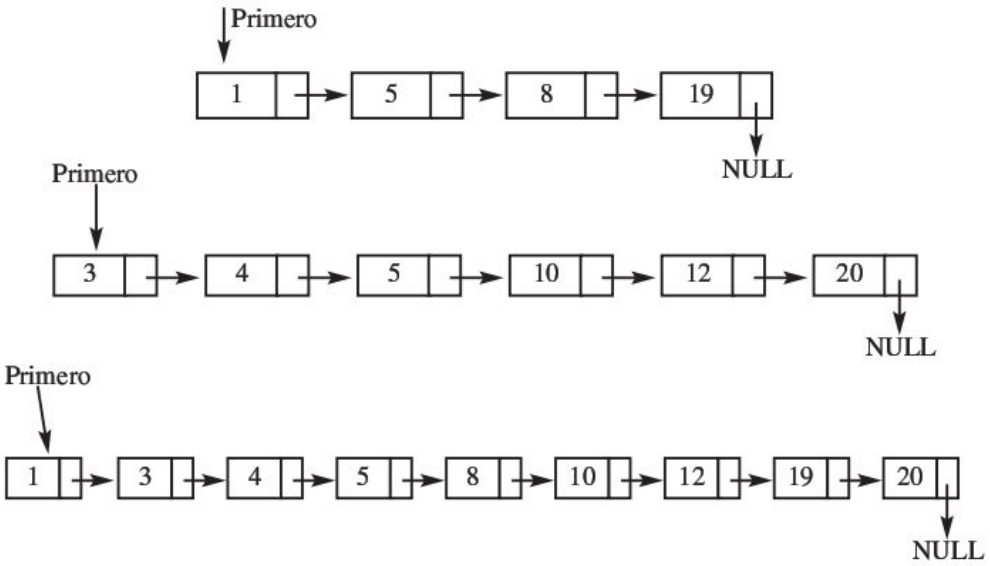
```
                    {
                        ObraAutor= RespBus->
                                ➥RegresaInfo().RegresaLisLibros();
                        ObraAutor.Imprime(ObraAutor.RegresaPrimero());
                    }
                    else
                    cout<<"\n\nEse autor no está registrado. \n\n";
                    break;
                  }
            case 5: cout<<"\n\nFin del proceso. \n\n";
        }
    } while (OpcTrab != 5);
}
```

En la clase `Autor` del ejemplo 6.7, se incluyó un atributo que es una lista de objetos (`Lista<Libro> Obra`). Dado que el atributo `Obra` es una lista simplemente ligada, se le pueden aplicar todos los métodos definidos en la clase `Lista`. También se pudo definir como un apuntador a un nodo que tuviera como información base la clase libro. En este caso, se tendrían que haber adaptado algunos de los métodos vistos. La declaración del atributo `Obra` hubiera quedado: `NodoLista<Libro> *Obra`, siendo un dato tipo apuntador y no un dato tipo lista como en el caso anterior.

Ejercicios

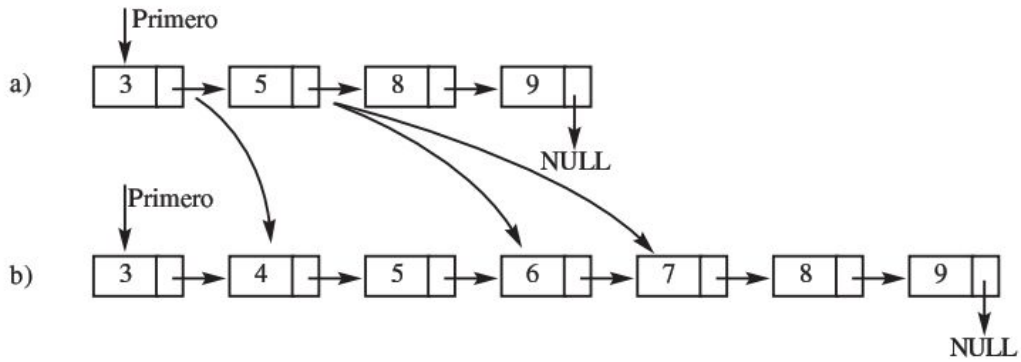
1. Defina una plantilla para la clase `ListaSimLigada`. Decida qué atributos y métodos incluir. Se sugiere que declare un apuntador al primero y otro al último nodo de la lista.
2. Escriba un programa en ***C++*** que:
   a) Use la plantilla del ejercicio anterior para declarar un objeto tipo lista simplemente ligada de números enteros.
   b) Genere una lista con al menos 10 nodos que contengan 10 números enteros distintos entre sí.
   c) Encuentre el mayor de los valores almacenados en la lista y lo elimine (también debe quitar el nodo en el cual está almacenado). Este proceso se debe repetir hasta que la lista quede vacía.

3. Escriba un programa que mezcle dos listas simplemente ligadas de números enteros, cuyos valores están ordenados crecientemente. El programa debe generar una tercera lista, también ordenada, sin repetir elementos y no debe afectar las listas dadas como datos. Observe la siguiente figura, las dos primeras listas son los datos y la tercera es el resultado.



4. Escriba un programa que mezcle dos listas simplemente ligadas de números enteros, cuyos valores están ordenados crecientemente. El programa debe ir generando una única lista con los nodos de las listas dadas como datos. Al terminar el proceso, sólo debe quedar una lista ordenada, formada con la unión de las listas de entrada.
5. Escriba un programa que, dada una lista simplemente ligada de números enteros, elimine los elementos repetidos.
   a) Considere el caso de una lista ordenada.
   b) Considere el caso de una lista desordenada.
6. Escriba un programa que, dada una lista simplemente ligada de números enteros ordenados crecientemente, agregue tantos nodos como sea necesario de manera que la lista quede formada con los nodos requeridos para que contengan todos los números comprendidos entre el valor del primer nodo y el valor del último. Por ejemplo, si la lista dada por el usuario es la que apa-

rece en la parte a) de la siguiente figura, luego de ejecutarse el programa planteado, la lista debería quedar como lo muestra la parte b).



7. Defina la clase `ListaCircularSimple` correspondiente a una estructura tipo lista circular simplemente ligada. No utilice nodo de cabecera. Por lo menos, los métodos que debe incluir en la clase son: `InsertaOrdenada()` (este método debe ir insertando ordenadamente elementos a la lista), `EliminaNodo()` (este método debe poder eliminar un elemento dado como referencia si estuviera en la lista), `ImprimeLista()` (este método imprime todos los elementos de la lista), `BuscaNodo()` (este método busca en la lista un valor dado como referencia). En todos los métodos debe considerar posibles casos de fracaso. Utilice plantillas para su definición.

8. Retome el problema anterior y defina una clase `Alumnos` con todos los atributos y métodos que crea necesarios (puede usar la solución al problema 8 del capítulo 2). La clase `Alumnos` servirá como tipo para el atributo `Información` de cada nodo de la lista. Con estas especificaciones desarrolle una aplicación que:

   a) Permita crear (alfabéticamente por nombre de alumno) una lista con los alumnos que toman cierta materia.

   b) Imprima los datos de todos los alumnos.

   c) Busque en la lista el nombre de un alumno. Si lo encuentra, debe imprimir todos sus datos, en caso contrario debe imprimir un mensaje adecuado.

   d) Busque en la lista el nombre de un alumno que se dio de baja de la materia, si su nombre no está en la lista, la aplicación debe eliminarlo o enviar un mensaje adecuado.

9. Implemente la estructura cola con una lista simplemente ligada. Utilice la plantilla de la clase definida en el ejercicio 1 para definir una plantilla para la clase cola.
10. Escriba un programa que invierta los elementos de una cola implementada por medio de una lista. Utilice la plantilla definida en el ejercicio anterior.
11. Defina una lista que pueda almacenar, en cada nodo, un par (X_i, Y_i) de números reales. Haga los cambios en las clases que crea conveniente.
12. Retome el problema anterior. Escriba un programa que lea una serie de N ($1 \leq N \leq 50$) pares de números reales, los guarde en la lista y los use para calcular e imprimir el resultado de las siguientes expresiones:

$$B0 = Yprom - B1\ Xprom$$

$$B1 = \frac{\sum_{i=1}^{N} X_i Y_i - N X_{prom} Y_{prom}}{\sum_{i=1}^{N} (X_i)^2 - N (X_{prom})^2}$$

13. Retome los métodos de eliminación de nodos de una lista simplemente ligada (que se explicaron anteriormente) y modifíquelos de tal manera que regresen la dirección del nodo eliminado.
14. Escriba un programa, que usando una lista doblemente ligada, pueda almacenar y manipular información relacionada a socios de un club deportivo. La especificación de los datos correspondientes a cada socio se presenta a continuación. El programa debe permitir a los usuarios, por medio de un menú, llevar a cabo las siguientes operaciones:

    a) Registrar un nuevo socio. Considere que no puede haber dos socios con el mismo número (`NumeroSocio`). La lista de socios debe ir quedando ordenada de menor a mayor, según el número de socio.

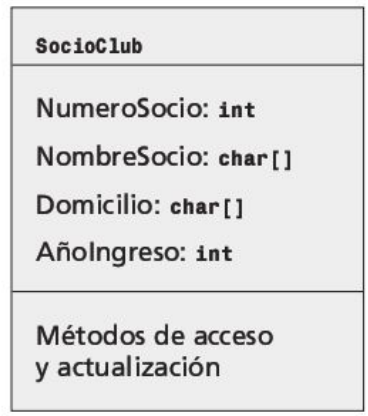
    b) Dar de baja un socio del club.

    c) Generar un reporte con todos los socios que tengan una antigüedad mayor o igual a una proporcionada por el usuario.

d) Cambiar el domicilio de un socio registrado.

e) Generar un reporte con los datos de todos los socios.

f) Calcular e imprimir el total de socios registrados.



15. Defina la clase `ListaCircularDoble` correspondiente a una estructura tipo lista circular doblemente ligada. No utilice nodo de cabecera. Los métodos que, por lo menos, debe incluir en la clase son: `InsertaOrdenada()` (este método debe ir insertando ordenadamente elementos a la lista), `EliminaNodo()` (este método debe poder eliminar un elemento dado como referencia si estuviera en la lista), `ImprimeLista()` (este método imprime todos los elementos de la lista), `ImprimeNodo()` (este método imprime la información de un nodo de la lista, cuya dirección se da como parámetro), `BuscaNodo()` (este método busca en la lista un valor dado como referencia, si lo encuentra regresa la dirección del nodo y si no el valor **NULL**). En todos los métodos debe considerar posibles casos de fracaso. Utilice plantillas para su definición.

16. En una empresa se necesita un sistema que permita manejar la información de los automóviles que tienen para su personal. Para representar los datos de los automóviles se debe tener en cuenta la clase `Automovil` dada más abajo. Escriba un programa que, usando una lista circular doblemente ligada, pueda:

    a) Registrar un automóvil nuevo. Considere que no puede haber dos automóviles con la misma clave que los identifica y que la lista debe ir quedando ordenada por clave.

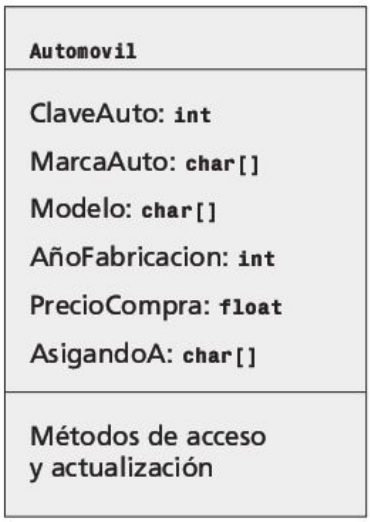
    b) Dar de baja un automóvil que ya no está disponible para el personal.

c) Generar un reporte de todos los automóviles que sean de un cierto año. El año de interés será un dato dado por el usuario.

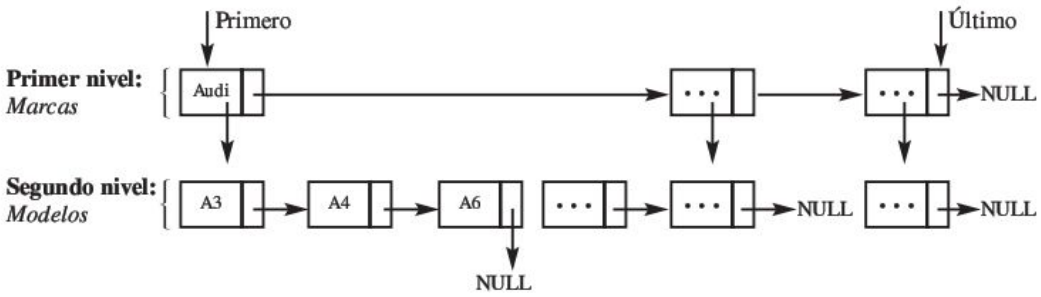
d) Generar un reporte de todos los automóviles cuyo precio sea superior a un monto dado por el usuario.

e) Dado el nombre de un empleado, imprimir los datos del automóvil que tiene asignado.

f) Cambiar el nombre de la persona a la que está asignado un automóvil.



17. Observe el siguiente esquema. Diseñe una estructura de datos que pueda representar los datos y las relaciones entre ellos de manera adecuada. Defina una clase para las marcas y otra para los modelos.



18. Retome el problema anterior. Escriba un programa en ***C++***, que mediante menús pueda:

    a) Agregar una nueva marca.

    b) Agregar un nuevo modelo a una marca registrada.

    c) Eliminar una marca. En este caso se deben eliminar también todos los modelos que tiene dicha marca.

    d) Eliminar un modelo de una marca registrada.

    e) Generar un reporte con todas las marcas y todos los modelos de las mismas.

    f) Dada una marca, imprimir todos los modelos que tiene.

    g) Generar un reporte con el modelo más caro de cada una de las marcas.

19. Retome el problema anterior. Ahora considere que cada modelo tiene una lista, de tercer nivel, con todas las versiones del mismo: económico, con piel, equipados, con quemacocos, etcétera. Deberá definir una clase para las versiones. Desarrolle una aplicación, escrita en ***C++***, que mediante menús, pueda realizar lo siguiente:

    a) Agregar una nueva marca.

    b) Incorporar un nuevo modelo a una marca registrada.

    c) Agregar una versión a un modelo de una cierta marca.

    d) Eliminar una marca. En este caso se deben eliminar también todos los modelos que tiene dicha marca, y de cada modelo se deben eliminar todas las versiones.

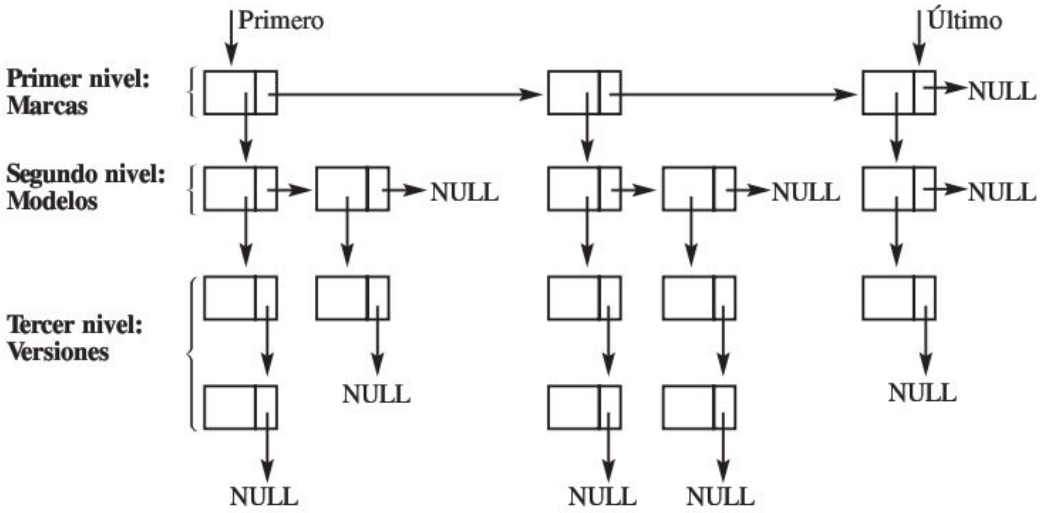
    e) Eliminar un modelo de una marca registrada. En este caso, también se deben eliminar las versiones de dicho modelo.

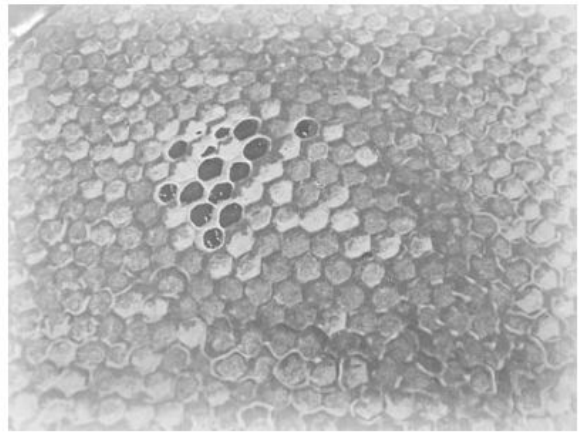
    f) Eliminar una versión de un modelo de una cierta marca.

    g) Generar un reporte con todas las marcas, sus modelos y las distintas versiones de éstos.

    h) Encuentre e imprima la marca que más modelos tiene.

    i) Encuentre e imprima el modelo que más versiones tiene.


Primero
Último
Primer nivel:
Marcas
Segundo nivel:
Modelos
Tercer nivel:
Versiones
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL



CAPÍTULO 7

Árboles

7.1 Introducción

Este capítulo estudia la estructura de datos conocida con el nombre de ***árbol***. Presenta sus principales características, cómo se relacionan sus componentes y analiza las operaciones que pueden aplicárseles.

Los árboles son estructuras de datos no lineales. Cada elemento, conocido con el nombre de `nodo`, puede tener varios sucesores. En términos generales, un árbol se define como una colección de nodos donde cada uno, además de almacenar información, guarda la dirección de sus sucesores. Se conoce la dirección de uno de los nodos, llamado `raíz`, y a partir de él se tiene acceso a todos los otros miembros de la estructura.

Existen diversas maneras de representar un árbol, las más comunes son: grafos, anidación de paréntesis y diagramas de Venn. La figura

7.1 muestra un árbol representado por medio de un grafo, en el cual cada nodo está indicado por un círculo y la relación entre ellos por un arco.

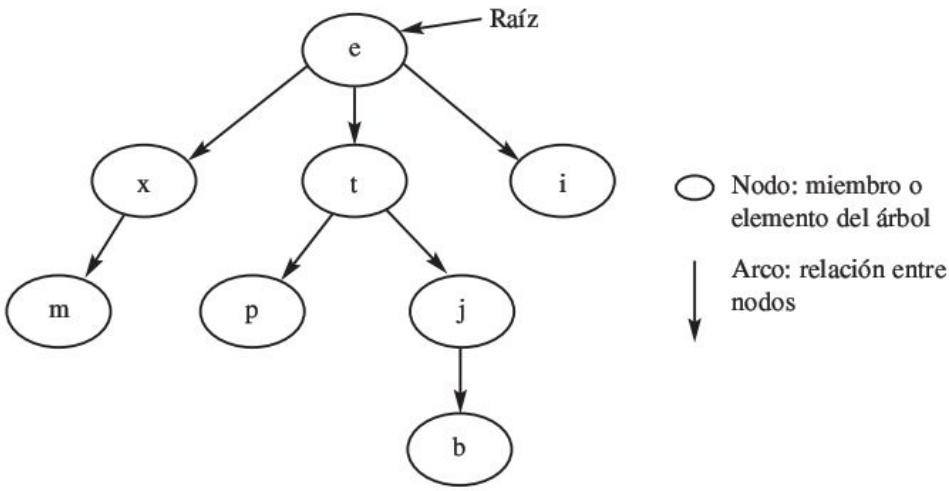


FIGURA 7.1 *Estructura* Árbol *representada con un grafo*

Al observar la representación del árbol por medio de una gráfica puede observarse a cada nodo como un árbol. Por consiguiente se dice que un árbol está formado por 0 o más subárboles, llegando así a una definición recursiva de esta estructura de datos.

En una estructura tipo árbol se definen relaciones entre sus miembros. A continuación se presentan las relaciones más importantes y se ejemplifican usando el árbol de la figura 7.1.

- ***Hijo***. Se dice que un nodo es hijo (o descendiente) de otro si este último apunta al primero. El nodo que almacena el valor *t* es hijo del nodo que almacena el valor *e*. Los nodos *p* y *j* son hijos de *t*.
- ***Padre***. Se dice que un nodo es padre de otro si este último es apuntado por el primero. El nodo que almacena el valor *j* es padre del nodo que almacena el valor *b*. El nodo *e* es padre de *x*, *t* e *i*.
- El origen de cada arco está en el nodo padre y la flecha llega al nodo hijo.
- ***Hermano***. Dos nodos son hermanos si son apuntados por el mismo nodo, es decir si tienen el mismo padre. Los nodos *x*, *t* e *i* son hermanos.

Además, los nodos pertenecen a una de las siguientes categorías según su ubicación en la estructura.

- ***Raíz***. Se dice que un nodo es raíz si a partir de él se relacionan todos los otros nodos. Si un árbol no es vacío, entonces tiene un único nodo raíz. En la figura 7.1, el nodo raíz es el que almacena el valor *e*.
- ***Hoja*** o ***terminal***. Se dice que un nodo es una hoja del árbol (o terminal) si no tiene hijos. En la figura 7.1, los nodos que almacenan los valores *m*, *p*, *b* e *i* son hojas o nodos terminales.
- ***Interior***. Se dice que un nodo es interior si no es raíz ni hoja. En la figura 7.1, los nodos que almacenan los valores *x*, *t*, y *j* son nodos interiores.

Se define el nivel y grado de cada nodo y la altura y el grado del árbol de la siguiente manera:

- ***Nivel de un nodo***. Se dice que el nivel de un nodo es el número de arcos que deben ser recorridos, partiendo de la raíz, para llegar hasta él. La raíz tiene nivel 1. En el árbol de la figura 7.1, el nivel de los nodos que almacenan los valores *x*, *t* e *i* es 2 y el nivel del nodo *b* es 4.
- ***Altura del árbol***. Se dice que la altura de un árbol es el máximo de los niveles, considerando todos sus nodos. El árbol de la figura 7.1 tiene una altura igual a 4.
- ***Grado de un nodo***. Se dice que el grado de un nodo es el número de hijos que tiene dicho nodo. En la figura 7.1, el grado del nodo que almacena el valor *t* es 2 y el grado del nodo *x* es 1.
- ***Grado del árbol***. Se dice que el grado de un árbol es el máximo de los grados, considerando todos sus nodos. El árbol de la figura 7.1, es de grado 3.

7.2 Árboles binarios

Un ***árbol binario*** es un árbol de grado 2 en el cual sus hijos se identifican como subárbol izquierdo y subárbol derecho. Por lo tanto, cada nodo almacena información y las direcciones de sus descendientes (máximo 2). Es un tipo de árbol muy usado, ya que saber el número máximo de hijos que puede tener cada nodo facilita las operaciones sobre ellos. La figura 7.2 presenta el esquema de un nodo de un árbol binario.

FIGURA 7.2 *Estructura de un nodo de un árbol binario*

La figura 7.3 presenta un ejemplo de uso de un árbol binario. En este caso, la estructura se emplea para almacenar el árbol genealógico de *María*. En cada nodo se guarda la información de los ancestros de *María* y los arcos indican la relación entre ellos.

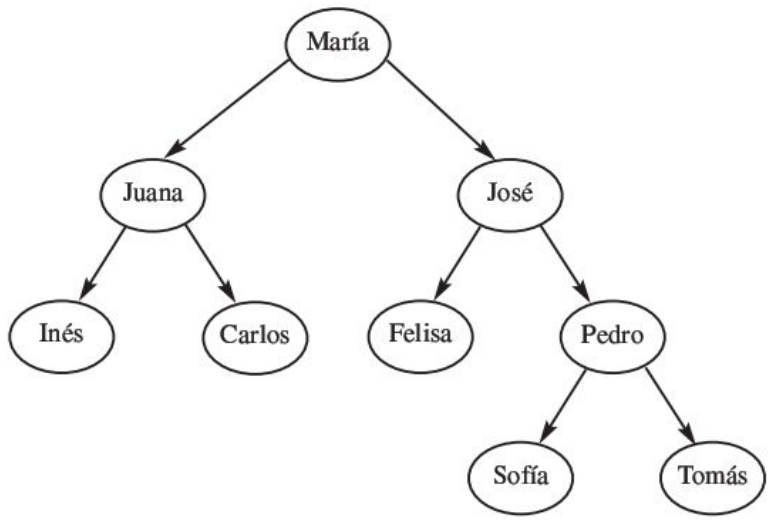


FIGURA 7.3 *Ejemplo de árbol binario*

La característica de este tipo de árbol (cada nodo tiene máximo 2 hijos) se puede aprovechar para organizar la información. Retomando el ejemplo de la figura 7.3, se puede establecer que los hijos izquierdos representen los ascendientes femeninos de *María*, mientras que los hijos derechos los ascendientes masculinos. Así, la mamá de *María* es *Juana* y su abuela materna es *Inés*. El papá de *María* es *José*. A su vez, la mamá de *José* es *Felisa* y su papá es *Pedro*.

La implementación más efectiva de los árboles binarios es por medio de memoria dinámica, obteniendo así una estructura dinámica. Las figuras 7.4 y 7.5

presentan las plantillas de la clase `NodoArbol` y de la clase `ArbolBinario` respectivamente. Se usan plantillas para dar mayor generalidad a la solución. La clase `NodoArbol` tiene tres atributos, uno que representa la información a almacenar por lo que se define de tipo `T`, y otros dos que representan la dirección del hijo izquierdo y del hijo derecho respectivamente, por lo que se declaran como punteros a objetos de la misma clase. Por su parte, la clase `ArbolBinario` tiene un único atributo que representa la dirección del primer elemento del árbol (la raíz) por lo cual es de tipo puntero a un objeto de tipo `NodoArbol`.

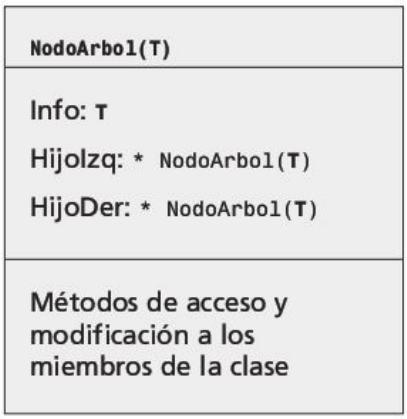


FIGURA 7.4 *Clase* `NodoArbol`

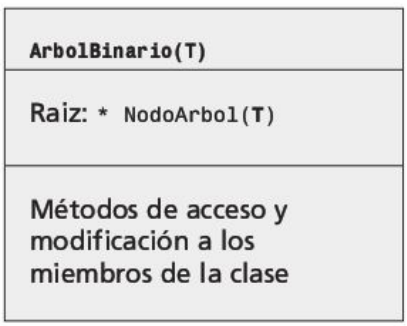


FIGURA 7.5 *Clase* `ArbolBinario`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a la definición de las plantillas de las clases `NodoArbol` y `ArbolBinario`.

```
/* Prototipo de la plantilla de la clase ArbolBinario. De esta manera,
➥en la clase NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinario;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥(que es la razón de ser de la estructura tipo árbol) y las direcciones
➥de sus hijos izquierdo y derecho. En la sección pública se establece la
➥relación de amistad entre esta clase y la clase ArbolBinario para que los
➥métodos de esta última puedan tener acceso a sus miembros privados. */
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo();
       friend class ArbolBinario<T>;
};

/* Declaración del método constructor por omisión. Inicializa las ligas
➥a los subárboles con el valor de NULL, indicando que están vacías. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que permite conocer la información almacenada en el nodo. */
template <class T>
T NodoArbol<T>::RegresaInfo()
{
   return Info ;
}


/* Declaración de la clase ArbolBinario. Su atributo es un puntero al
➥nodo raíz. */
template <class T>
class ArbolBinario
{
   private:
       NodoArbol<T> *Raiz;
```

```
    public:
        ArbolBinario ();
        /* En esta sección se declaran los métodos de acceso y
        ➥modificación a los miembros de la clase. */
};

/* Declaración del método constructor. Inicializa el puntero a la raíz con
➥el valor NULL, indicando que el árbol está vacío (no tiene nodos). */
template <class T>
ArbolBinario<T>::ArbolBinario()
{
    Raiz= NULL;
}
```

La clase `NodoArbol` se utiliza para representar un nodo de un árbol binario, por lo tanto se incluyen tres atributos: uno para almacenar información de cualquier tipo (tipo `T`) y los otros dos para almacenar la dirección de los subárboles izquierdo y derecho respectivamente, los cuales son punteros a objetos de la misma clase. La sección pública contiene tres miembros (podría tener más o menos), dependiendo de la definición de la clase que se haga. Estos elementos son: el método constructor, un método que facilita (a usuarios externos a la clase) conocer la información guardada, y la declaración de amistad con la clase `ArbolBinario`. Esta última declaración permite que los métodos de la clase amiga tengan acceso a sus miembros privados y protegidos.

A partir de la clase `NodoArbol` se define la clase `ArbolBinario`, la cual está formada por un atributo único (tipo puntero a un objeto `NodoArbol`) que representa el puntero al nodo raíz del árbol binario. Este puntero permite el acceso a todos los elementos del árbol ya que la raíz tiene la dirección de sus dos hijos, éstos, la dirección de sus respectivos hijos y así hasta llegar a nodos terminales. En la sección pública se declaran los métodos necesarios para tener acceso a los atributos, y de esta manera manipular la información almacenada.

7.2.1 Operaciones en árboles binarios

En esta sección se estudian las operaciones de creación y recorrido de un árbol binario. La primera hace referencia a crear una estructura que responda a las características analizadas e ir almacenando información en cada uno de los nodos.

La segunda permite visitar todos los nodos de un árbol sin repetir ninguno, aprovechando el conocimiento que se tiene acerca de la estructura.

La ***creación*** de un árbol binario se lleva a cabo a partir de la raíz. Se crea un nodo y se almacena su información. Posteriormente se pregunta si dicho nodo tiene hijo izquierdo, si la respuesta es afirmativa, entonces se invoca nuevamente el método pero ahora con el subárbol izquierdo. El proceso se repite con cada nodo hasta llegar a las hojas. Luego, se hace lo mismo para crear cada uno de los subárboles derechos. Se utiliza la instrucción **`new()`** para asignar un espacio de memoria de manera dinámica.

A continuación se presenta el método para llevar a cabo la secuencia de pasos descrita.

```
/* Plantilla del método que crea un árbol binario. Recibe como parámetro
➥un apuntador a un subárbol. La primera vez es la raíz del árbol la cual
➥se inicializó con el valor NULL, indicando que el árbol está vacío. */
template <class T>
void ArbolBinario<T>::CreaArbol(NodoArbol<T> *Apunt)
{
   char Resp;
   /* Se crea un nodo. */
   Apunt= new  NodoArbol<T>;
   cout<<"\n\nIngrese la información a almacenar:";
   cin>>Apunt->Info;
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo izquierdo (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
        /* Se invoca al método con el subárbol izquierdo. Se usa la
        ➥definición recursiva de un árbol. */
        CreaArbol(Apunt->HijoIzq);
        Apunt->HijoIzq= Raiz;
   }
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo derecho (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
        /* Se invoca al método con el subárbol derecho. Se usa la
        ➥definición recursiva de un árbol. */
        CreaArbol(Apunt->HijoDer);
        Apunt->HijoDer= Raiz;
   }
   Raiz= Apunt;
}
```

El ***recorrido*** de un árbol binario consiste en visitar todos sus nodos una sola vez. Por lo tanto, podrá hacerse (aprovechando las características de la estructura del árbol) de tres maneras diferentes: visitando la **raíz**, el **hijo izquierdo** y el **hijo derecho**, o visitando el **hijo izquierdo**, la **raíz** y el **hijo derecho**, o bien, visitando el **hijo izquierdo**, el **hijo derecho** y la **raíz**. En los tres casos, la regla se aplica hasta llegar a las hojas. Estos métodos se conocen con el nombre de ***preorden***, ***inorden*** y ***postorden*** respectivamente.

Preorden	Inorden	Postorden
1. Visita la raíz	**1.** Recorre el subárbol izquierdo	**1.** Recorre el subárbol izquierdo
2. Recorre el subárbol izquierdo	**2.** Visita la raíz	**2.** Recorre el subárbol derecho
3. Recorre el subárbol derecho	**3.** Recorre el subárbol derecho	**3.** Visita la raíz

Considerando el árbol binario de la figura 7.6, el resultado de los tres recorridos es el siguiente.

- **Preorden**: 304 – 550 – 143 – 2020 – 1995 – 876 – 609 – 300
- **Inorden**: 143 – 550 – 2020 – 304 – 876 – 1995 – 609 – 300
- **Postorden**: 143 – 2020 – 550 – 876 – 300 – 609 – 1995 – 304

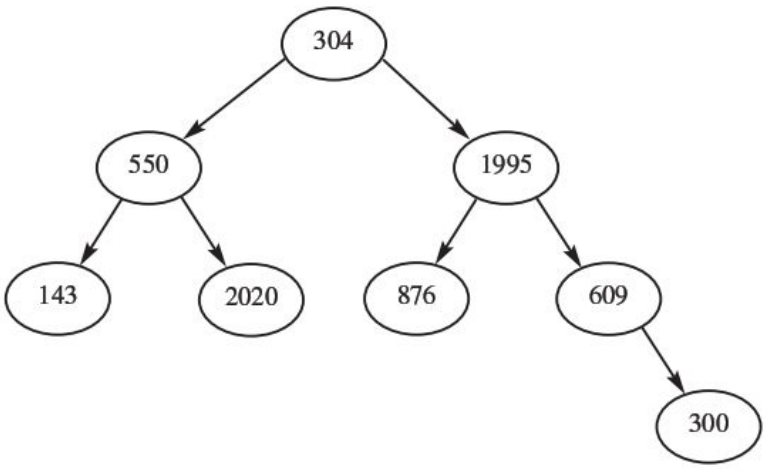


FIGURA 7.6 *Recorrido de árboles binarios*

Los métodos para llevar a cabo esta operación se presentan a continuación. En los tres casos la operación de visitar la raíz se consideró como la impresión de su contenido, aunque podría ser cualquier operación válida según el tipo de información almacenada en el nodo.

```
/* Método que realiza el recorrido preorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Preorden (NodoArbol<T> *Apunt)
{
   if (Apunt)
   {
        cout<< Apunt->Info << " ";
        Preorden(Apunt->HijoIzq);
        Preorden(Apunt->HijoDer);
   }
}

/* Método que realiza el recorrido inorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Inorden (NodoArbol<T> *Apunt)
{
   if (Apunt)
   {
        Inorden(Apunt->HijoIzq);
        cout<< Apunt->Info << " ";
        Inorden(Apunt->HijoDer);
   }
}

/* Método que realiza el recorrido postorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Postorden (NodoArbol<T> *Apunt)
```

```
{
   if (Apunt)
   {
        Postorden(Apunt->HijoIzq);
        Postorden(Apunt->HijoDer);
        cout<< Apunt->Info << " ";
   }
}
```

En los tres métodos, la instrucción de imprimir se da sobre el contenido de la raíz. La naturaleza recursiva de los métodos permite lograr la impresión de todos los nodos. Las instrucciones que forman cada uno de los métodos son las mismas, lo único que cambia es el orden en el cual se ejecutan.

Si se analiza el recorrido preorden con el árbol de la figura 7.6, se puede observar que primero se imprime el número 304, luego se invoca el método con el subárbol izquierdo, quedando pendiente el recorrido con el subárbol derecho (internamente se guardan en una pila las instrucciones pendientes de ejecutar). Lo mismo sucede cuando llega con el subárbol izquierdo, imprime el valor 550 e invoca el método con su subárbol izquierdo y deja pendiente el recorrido con su subárbol derecho. Una vez agotado el lado izquierdo, pasa al lado derecho del último nodo visitado. Se van tomando de la pila todos los subárboles derechos que quedaron pendientes de visitar y se van recorriendo. Como consecuencia, el primer número (correspondiente a un subárbol derecho) que se imprime es el 2020, luego el 1995 y como este último tiene subárbol izquierdo, entonces se invoca al método con éste (876). Se repite el proceso hasta que ya no queden nodos a visitar.

En un árbol binario también pueden realizarse otras operaciones como buscar, insertar o eliminar un dato en un árbol ya generado. Estas operaciones serán analizadas en un tipo especial de árboles binarios, los cuales se tratan en la siguiente sección.

El programa 7.1 presenta una aplicación de árboles binarios. El programa imprime los datos de todos los ascendientes femeninos de un individuo, tanto de la rama materna como de la paterna. Se utiliza un objeto tipo árbol binario para almacenar los datos de los ascendientes de una persona, es decir, su árbol genealógico. En la raíz de cada subárbol izquierdo se almacena la información de un ascendiente femenino, mientras que en la raíz de cada subárbol derecho se guarda la información de un ascendiente masculino. Considerando el árbol de la figura 7.7, algunas de las relaciones familiares representadas son:

Anahí es mamá de Juan
José es papá de Juan
Inés es mamá de Anahí
Pedro es papá de Anahí
Ana es mamá de José
Luis es papá de José

y el programa imprimirá que los ascendientes femeninos de Juan son Anahí, Inés y Ana. El programa 7.1 incluye las clases `ArbolBinario` y `Personas` sólo con los métodos requeridos para la aplicación.

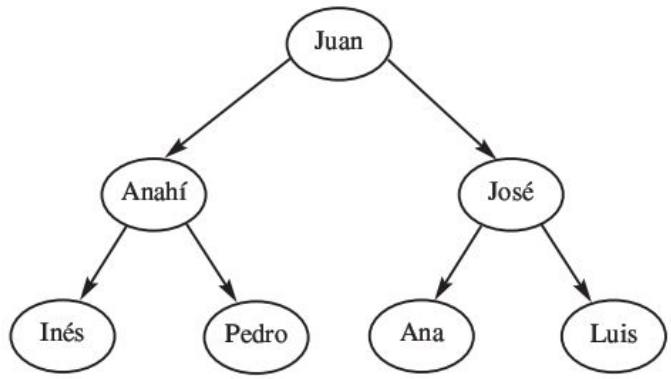


FIGURA 7.7 *Árbol genealógico de* Juan

Programa 7.1

```
/* Programa que imprime los datos de los ascendientes femeninos de un
➥individuo. Primero forma el árbol genealógico y posteriormente genera
➥el reporte. */

/* Definición de la clase Persona. */
class Persona
{
   private:
       int AnioNac, Vive;
       char NomPers[64], LugNac[64];
```

```
   public:
       Persona();
       Persona(int, int, char[], char[]);
       friend  istream & operator>> (istream & , Persona & );
       friend  ostream & operator<< (ostream & , Persona & );
};

/* Declaración del método constructor por omisión. */
Persona::Persona()
{}

/* Declaración del método constructor con parámetros. */
Persona::Persona(int ANac, int Vi, char NomP[], char LugN[])
{
   AnioNac= ANac;
   Vive= Vi;
   strcpy(NomPers, NomP);
   strcpy(LugNac, LugN);
}

/* Sobrecarga del operador >> para permitir la lectura de objetos tipo
➥Persona de manera directa. */
istream & operator>>(istream & Lee,  Persona & ObjPers)
{
   cout<<"\n\nIngrese nombre de la Persona:";
   Lee>> ObjPers.NomPers;
   cout<<"\n\nIngrese año de nacimiento:";
   Lee>> ObjPers.AnioNac;
   cout<<"\n\nIngrese lugar de nacimiento:";
   Lee>> ObjPers.LugNac;
   cout<<"\n\n¿Está viva?:";
   Lee>> ObjPers.Vive;
   return  Lee;
}

/* Sobrecarga del operador << para permitir la escritura de objetos tipo
➥Persona de manera directa. */
ostream & operator<< (ostream & Escribe, Persona & ObjPers)
{
   Escribe<<"\n\nDatos de la Persona\n";
   Escribe<<"\nNombre:  "<<ObjPers.NomPers;
   Escribe<<"\nLugar de nacimiento:  "<<ObjPers.LugNac;
   Escribe<<"\nAño de nacimiento:  "<<ObjPers.AnioNac;
   if (ObjPers.Vive == 1)
       Escribe<<"\nEstá viva.\n";
   else
       Escribe<<"\nNo está viva.\n";
   return  Escribe;
}
```

```
/* Prototipo de la plantilla de la clase ArbolBinario. Así, en la clase
➥NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinario;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥que es la razón de ser de la estructura tipo árbol y las direcciones de
➥su hijo izquierdo y de su hijo derecho. */
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo();
       void ActualizaInfo(T);
       friend class ArbolBinario<T>;
};

/* Declaración del método constructor por omisión. Inicializa
➥las ligas a los subárboles con el valor de NULL. Indica nodo sin
➥descendientes. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
T NodoArbol<T>::RegresaInfo()
{
   return Info;
}

/* Método para actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbol<T>::ActualizaInfo(T Dato)
{
   Info= Dato ;
}
```

```
/* Declaración de la clase ArbolBinario. Tiene un puntero al nodo
➥raíz. */
template <class T>
class ArbolBinario

{
   private:
       NodoArbol<T> *Raiz;
   public:
       ArbolBinario ();
       NodoArbol<T> *RegresaRaiz();
       void CreaArbol(NodoArbol<T> *);
       void ImprimeIzq(NodoArbol<T> *);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL. Indica que el árbol está vacío. */
template <class T>
ArbolBinario<T>::ArbolBinario()
{
   Raiz= NULL;
}

/* Método que regresa el valor del apuntador a la raíz del árbol. */
template <class T>
NodoArbol<T> *ArbolBinario<T>::RegresaRaiz()
{
   return Raiz;
}

/* Método que crea un árbol binario. */
template <class T>
void ArbolBinario<T>::CreaArbol(NodoArbol<T> *Apunt)
{
   char Resp;
   Apunt= new  NodoArbol<T>;
   cout<<"\n\nIngrese la información a almacenar:";
   cin>>Apunt->Info;
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo izquierdo (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
       CreaArbol(Apunt->HijoIzq);
       Apunt->HijoIzq= Raiz;
   }
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo derecho (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
```

```
    {
        CreaArbol(Apunt->HijoDer);
        Apunt->HijoDer= Raiz;
    }
    Raiz= Apunt;
}

/* Método que imprime la información almacenada en las raíces de todos
➥los subárboles izquierdos. La primera vez recibe como dato la raíz del
➥árbol. */
template <class T>
void ArbolBinario<T>::ImprimeIzq(NodoArbol<T> *Apunt)
{
    if (Apunt)
    {
        if (Apunt->HijoIzq)
        {
             cout<<Apunt->HijoIzq->Info;
             ImprimeIzq(Apunt->HijoIzq);
        }
        ImprimeIzq(Apunt->HijoDer);
    }
}

/* Función principal. Crea el árbol genealógico de un individuo y
➥posteriormente imprime los datos de todos sus ascendientes femeninos. */
void main()
{
    ArbolBinario<Persona> Genealogico;
    Persona Individuo;
    NodoArbol<Persona> *Ap;

    Ap= Genealogico.RegresaRaiz();
    /* Se invoca el método que crea el árbol genealógico. */
    Genealogico.CreaArbol(Ap);
    Ap= Genealogico.RegresaRaiz();

    /* Se recupera la información del individuo. */
    Individuo= Ap->RegresaInfo();
    cout<<"\n\n\n_____\n\n";
    cout<<"Los ascendientes femeninos de: \n"<<Individuo;
    cout<<"\n\n_____\n";

    /* Se invoca el método que imprime los datos de los ascendientes
    ➥femeninos. */
    Genealogico.ImprimeIzq(Ap);
}
```

7.2.2 Árboles binarios de búsqueda

Un ***árbol binario de búsqueda*** se caracteriza porque la información de cada nodo es mayor que la información de cada uno de los nodos que están en su subárbol izquierdo y menor que la almacenada en los nodos que están en su subárbol derecho. La figura 7.8 presenta un ejemplo de árbol binario de búsqueda. Observe que todos los valores que están a la izquierda del 710 son menores que él. A su vez, los que están a su derecha son mayores. La misma regla se aplica en todos los nodos.

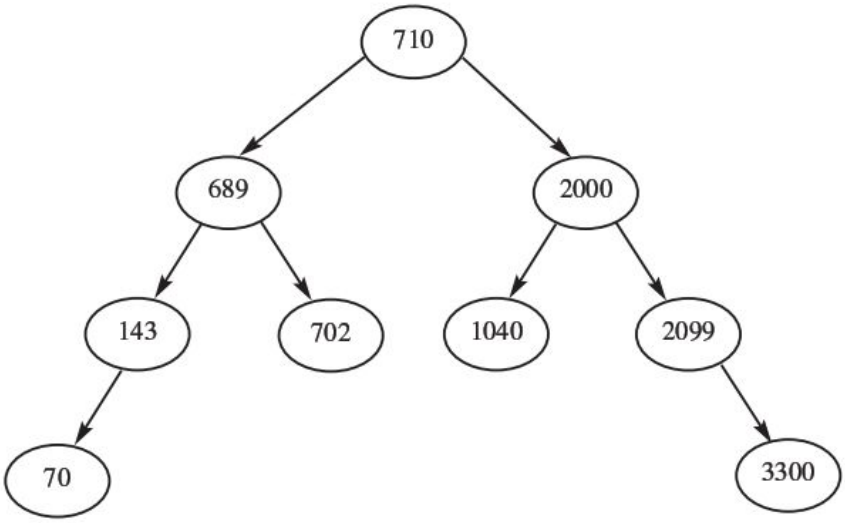


FIGURA 7.8 *Ejemplo de árbol binario de búsqueda*

El recorrido inorden de un árbol binario de búsqueda genera una lista ordenada de manera creciente de todos sus elementos. Tomando el árbol de la figura 7.8, este recorrido proporcionaría los elementos en el siguiente orden:

70 – 143 – 689 – 702 – 710 – 1040 – 2000 – 2099 – 3300

El orden que existe entre la información almacenada en el árbol facilita la operación de búsqueda de cualquiera de sus elementos. A continuación se analizarán las operaciones de búsqueda, inserción y eliminación en árboles binarios de búsqueda.

Operación de búsqueda

Para llevar a cabo la **búsqueda** de un elemento en un árbol binario de búsqueda se procede de la siguiente manera:

1. Se evalúa si el nodo visitado (la primera vez es la raíz) está definido.
2. Si la respuesta es afirmativa, se pregunta si el dato buscado es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se procede a buscar el dato en el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato buscado es mayor que el dato visitado.
      2.2.1. Si la respuesta es afirmativa, se procede a buscar el dato en el subárbol derecho.
      2.2.2. Si la respuesta es negativa, la búsqueda termina exitosamente. El dato fue encontrado.
3. Si la respuesta a la pregunta 1 es negativa, entonces termina la búsqueda con un fracaso.

A continuación se presenta un ejemplo de aplicación del algoritmo visto. En el árbol de la figura 7.9 se desea encontrar el valor 705. Con las líneas punteadas se señalan los nodos que se van visitando hasta llegar al buscado. En la tabla 7.1 se muestra la secuencia de pasos requeridos, usando el algoritmo, para realizar esta operación.

TABLA 7.1 Operación de búsqueda en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato buscado (705) es menor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
3	Se invoca el método con el subárbol izquierdo.
4	Se evalúa si el nodo visitado (689) está definido. En este caso sí lo está.
5	Se evalúa si el dato buscado (705) es menor que la información almacenada (689) en el nodo visitado. En este caso no lo es.

continúa

TABLA 7.1 Continuación

Operación	*Descripción*
6	Se evalúa si el dato buscado (705) es mayor que la información almacenada (689) en el nodo visitado. En este caso sí lo es.
7	Se invoca el método con el subárbol derecho.
8	Se evalúa si el nodo visitado (702) está definido. En este caso sí lo está.
9	Se evalúa si el dato buscado (705) es menor que la información almacenada (702) en el nodo visitado. En este caso no lo es.
10	Se evalúa si el dato buscado (705) es mayor que la información almacenada (702) en el nodo visitado. En este caso sí lo es.
11	Se invoca el método con el subárbol derecho.
12	Se evalúa si el nodo visitado (705) está definido. En este caso sí lo está.
13	Se evalúa si el dato buscado (705) es menor que la información almacenada (705) en el nodo visitado. En este caso no lo es.
14	Se evalúa si el dato buscado (705) es mayor que la información almacenada (705) en el nodo visitado. En este caso no lo es.
15	La búsqueda termina con éxito. El dato fue encontrado.

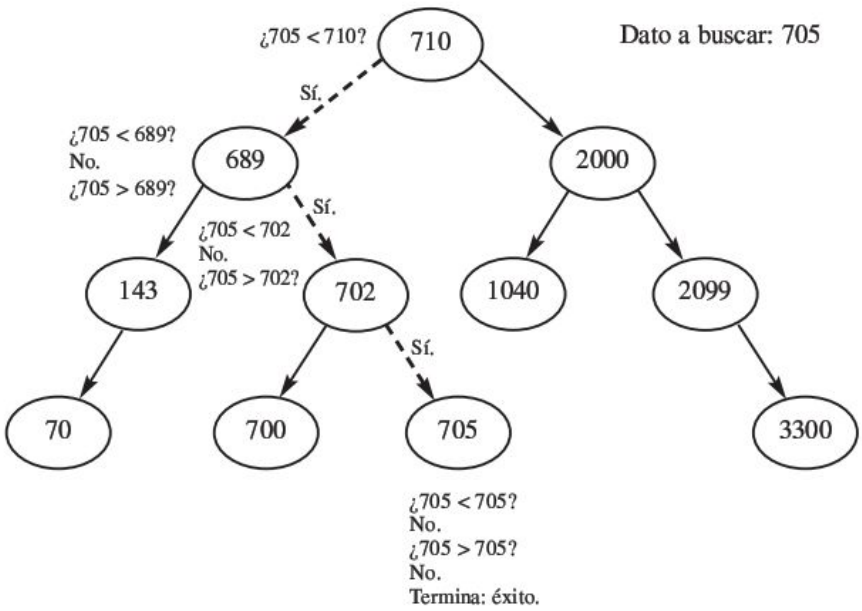


FIGURA 7.9 *Ejemplo de operación de búsqueda*

Como se puede deducir de los pasos presentados, el número de comparaciones se reduce a la mitad en cada nodo visitado. No se requiere visitar todos los nodos. El método que implementa esta operación es el siguiente:

```
/* Método que busca un dato en un árbol binario de búsqueda. Recibe como
➥parámetros un apuntador, que es la dirección del nodo a visitar (la
➥primera vez es el apuntador a la raíz) y el dato a buscar. Regresa como
➥resultado la dirección del nodo encontrado o el valor NULL, si la bús-
➥queda termina con fracaso. */
template <class T>
NodoArbol<T> * ArbolBinBus<T>::Busqueda (NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
        if (Dato < Apunt->Info)
            return Busqueda(Apunt->HijoIzq, Dato);
        else
            if (Dato > Apunt->Info)
                return Busqueda(Apunt->HijoDer, Dato);
            else
                return Apunt;
   else
        return NULL;
}
```

Operación de inserción

Insertar un nuevo elemento en un árbol binario de búsqueda requiere buscar la posición que debe ocupar el nuevo nodo de tal manera que no altere el orden del árbol. En la solución que aquí se propone no se aceptan elementos repetidos. Pueden existir aplicaciones en las cuales sí se permita. Sin embargo, en esta solución, si se detecta que en el árbol ya está almacenado un valor igual al que se pretende insertar, la operación se interrumpe. Los principales pasos para llevar a cabo esta operación son:

1. Se evalúa si el nodo (la primera vez es la raíz) visitado está definido.
2. Si la respuesta es afirmativa, se pregunta si el dato a insertar es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se invoca el proceso de inserción con el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato a insertar es mayor que el dato visitado.

2.2.1. Si la respuesta es afirmativa, entonces se invoca el proceso de inserción con el subárbol derecho.

2.2.2. Si la respuesta es negativa, entonces el proceso de inserción termina sin haberse realizado, ya que no se permiten elementos repetidos.

3. Si la respuesta al paso 1 es negativa, se crea el nuevo nodo, se almacena la información y se establecen las ligas entre el nuevo nodo y su padre.

Suponga que en el árbol de la figura 7.10 se quiere insertar el valor 1500. Las líneas punteadas indican los nodos visitados hasta encontrar el adecuado (1040) para proceder a la inserción. El nodo en negritas es el nuevo elemento agregado al árbol y la línea (también en negritas) es la liga entre éste y su padre. Aplicando el algoritmo dado, se realiza la secuencia de operaciones que se muestra en la tabla 7.2.

TABLA 7.2 Operación de inserción en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (710) en el nodo visitado. En este caso no lo es.
3	Se evalúa si el dato a insertar (1500) es mayor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
4	Se invoca el método con el subárbol derecho.
5	Se evalúa si el nodo visitado (2000) está definido. En este caso sí lo está.
6	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (2000) en el nodo visitado. En este caso sí lo es.
7	Se invoca el método con el subárbol izquierdo.
8	Se evalúa si el nodo visitado (1040) está definido. En este caso sí lo está.
9	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (1040) en el nodo visitado. En este caso no lo es.
10	Se evalúa si el dato a insertar (1500) es mayor que la información almacenada (1040) en el nodo visitado. En este caso sí lo es.
11	Se invoca el método con el subárbol derecho.
12	Se evalúa si el nodo visitado (NULL) está definido. En este caso no lo está.
13	Se crea un nuevo nodo, se le asigna el valor 1500 y se establece la liga entre él y su padre (el nodo que almacena el número 1040). El proceso termina.

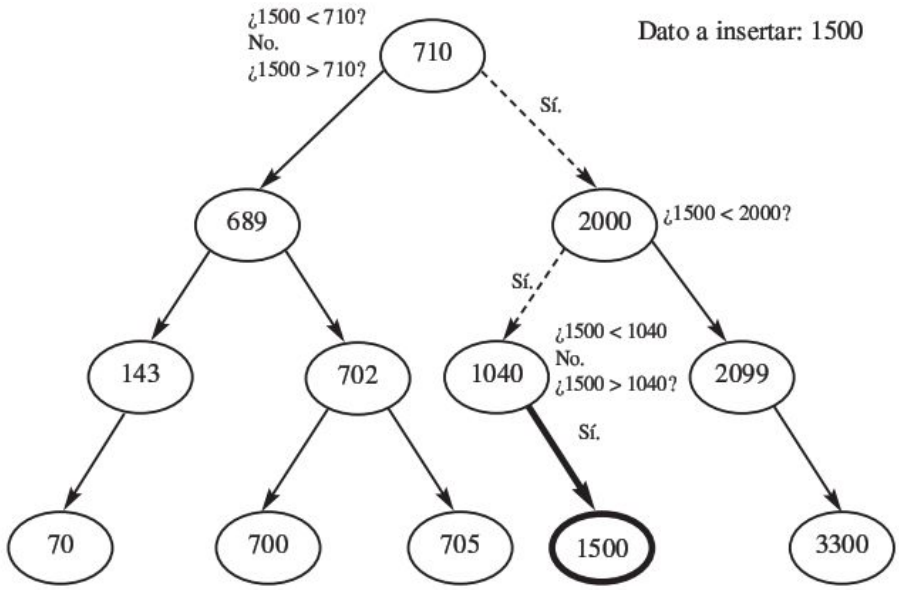


FIGURA 7.10 *Ejemplo de operación de inserción*

A continuación se presenta el método definido para insertar un nuevo nodo a un árbol binario de búsqueda.

```
/* Método que inserta un nodo en un árbol binario de búsqueda. Recibe
➥como parámetros un apuntador (la primera vez es la raíz del árbol) y
➥la información que se quiere almacenar en el nuevo nodo. En esta
➥implementación no se permite que haya información duplicada en el
➥árbol. */
template <class T>
void ArbolBinBus<T>::InsertaNodoSinRep(NodoArbol<T> *Apunt, T Dato)
{
   NodoArbol<T> *ApAux;
   if (Apunt)
   {
        if (Dato < Apunt->Info)
        {
            InsertaNodoSinRep(Apunt->HijoIzq, Dato);
            Apunt->HijoIzq= Raiz;
        }
```

```
            else
                if (Dato > Apunt->Info)
                {
                    InsertaNodoSinRep(Apunt->HijoDer, Dato);
                    Apunt->HijoDer= Raiz;
                }
            Raiz= Apunt;
        }
        else
        {
            /* Se crea un nuevo nodo, se le asigna la información y se
            ➥establecen las ligas entre los nodos correspondientes. */
            ApAux= new  NodoArbol<T>();
            ApAux->Info= Dato;
            Raiz= ApAux;
        }
    }
```

Operación de eliminación

Para **eliminar** un elemento en un árbol binario de búsqueda se requiere buscar el valor deseado y quitar el nodo que lo contiene. Este último paso se lleva a cabo de maneras diferentes dependiendo si el nodo eliminado es terminal o no. Si se trata de una hoja, entonces se quita directamente. En otro caso, para no perder las ligas a sus descendientes, se debe reemplazar por el nodo que se encuentra más a la derecha del subárbol izquierdo o por el que se encuentra más a la izquierda del subárbol derecho. En la solución que se da en este libro se usa el elemento que está más a la derecha del subárbol izquierdo. Los principales pasos para llevar a cabo esta operación son:

1. Se evalúa si el nodo visitado (la primera vez es la raíz) está definido.
2. Si la respuesta es afirmativa, entonces se pregunta si el dato a eliminar es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se invoca el proceso de eliminación con el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato a eliminar es mayor que el dato visitado.

2.2.1. Si la respuesta es afirmativa, se invoca el proceso de eliminación con el subárbol derecho.

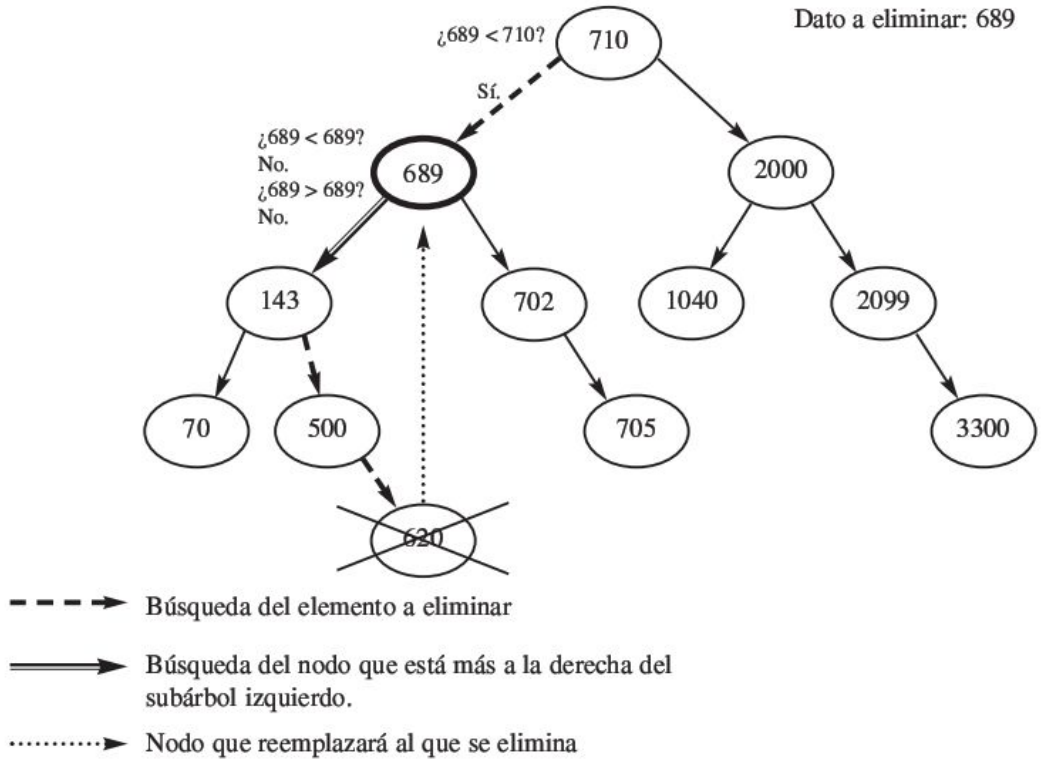
2.2.2. Si la respuesta es negativa, se elimina el nodo. Si es hoja, la eliminación es directa. Si tiene sólo un hijo se reemplaza por éste y si tiene dos se reemplaza por el que está más a la derecha del subárbol izquierdo. En estos dos últimos casos se libera el espacio de memoria del hijo, mientras que en el primero el correspondiente al nodo en cuestión.

3. Si la respuesta al paso 1 es negativa, entonces el dato no está en el árbol. El proceso de eliminación termina con fracaso.

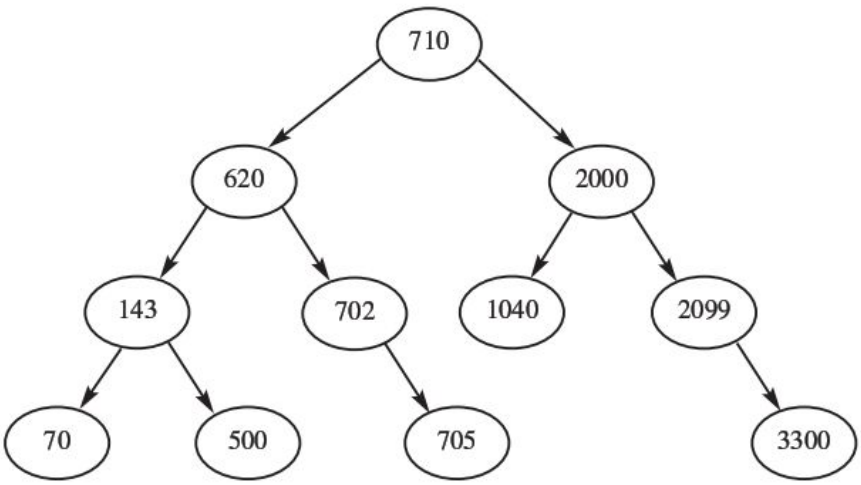
Suponga que en el árbol binario de búsqueda de la figura 7.11 se quiere eliminar el valor 689. En (a) se muestra el camino que se sigue para llegar al nodo deseado y al nodo cuyo contenido reemplazará al 689. En (b) se presenta el árbol una vez que el nodo fue eliminado. La tabla 7.3 presenta las operaciones realizadas, siguiendo el algoritmo dado, para eliminar el 689.

TABLA 7.3 Operación de eliminación en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato a eliminar (689) es menor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
3	Se invoca el método con el subárbol izquierdo.
4	Se evalúa si el nodo visitado (689) está definido. En este caso sí lo está.
5	Se evalúa si el dato a eliminar (689) es menor que la información almacenada (689) en el nodo visitado. En este caso no lo es.
6	Se evalúa si el dato a eliminar (689) es mayor que la información almacenada (689) en el nodo visitado. En este caso no lo es.
7	Se llegó al nodo que se quiere quitar. Como tiene dos hijos se reemplaza su contenido con el del nodo que está más a la derecha del subárbol izquierdo y se libera el espacio de memoria correspondiente al hijo. El proceso termina.



a)



b)

FIGURA 7.11 *Ejemplo de la operación de eliminación a) Antes de eliminar el valor 689, b) después de eliminarlo y reemplazarlo por el 620*

A continuación se presenta el método, escrito en *C++*, que permite eliminar un nodo de un árbol binario de búsqueda.

```
/* Método que elimina un nodo de un árbol binario de búsqueda. Recibe
➥como parámetro un apuntador (la primera vez es la raíz) y el dato a
➥eliminar. */
template <class T>
void ArbolBinBus<T>::EliminaNodo(NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
        if (Dato < Apunt->Info)
        {
            EliminaNodo(Apunt->HijoIzq, Dato);
            Apunt->HijoIzq= Raiz;
        }
        else
            if (Dato > Apunt->Info)
            {
                 EliminaNodo(Apunt->HijoDer, Dato);
                 Apunt->HijoDer= Raiz;
            }
            else
            {
                 NodoArbol<T> *ApAux1,*ApAux2,*ApAux3;
                 ApAux3= Apunt;
                 /* Encuentra el nodo que contiene el dato a eliminar.
                 ➥Verifica si tiene hijos. */
                 if (!ApAux3->HijoDer)
                      if (!ApAux3->HijoIzq)
                      /* Si no tiene hijo derecho ni izquierdo, entonces
                      ➥se redefine como vacío. */
                           Apunt= NULL;
                      else
                           /* Si sólo tiene hijo izquierdo, el nodo
                           ➥eliminado se reemplaza con éste.*/
                           Apunt= ApAux3->HijoIzq;
                 else
                      if (!ApAux3->HijoIzq)
                           /* Si sólo tiene hijo derecho, el nodo
                           eliminado se reemplaza con éste. */
                           Apunt= ApAux3->HijoDer;
                      else
                      {
                           /* Si tiene ambos hijos, entonces se reempla-
                           ➥za (en esta solución) por el nodo que está
                           ➥más a la derecha del subárbol izquierdo. */
```

```
                        ApAux1= ApAux3->HijoIzq;
                        ApAux2= ApAux3;
                        while (ApAux1->HijoDer)
                        {
                            ApAux2= ApAux1;
                            ApAux1= ApAux1->HijoDer;
                        }
                        ApAux3->Info= ApAux1->Info;
                        if (ApAux3 == ApAux2)
                            ApAux3->HijoIzq= NULL;
                        else
                            if (!ApAux1->HijoIzq)
                                ApAux2->HijoDer= NULL;
                            else
                                ApAux2->HijoDer= ApAux1->HijoIzq;
                        ApAux3= ApAux1;
                    }
                delete(ApAux3);
            }
            Raiz= Apunt;
}
```

El método dado, como el correspondiente a la inserción, se definió de tipo **void**. Sin embargo, ambos pueden modificarse y declararse enteros de tal manera que regresen un valor que indique si la operación se llevó a cabo o no con éxito.

El programa 7.2 presenta la plantilla de la clase árbol binario de búsqueda con los métodos analizados. También incluye la plantilla correspondiente a la clase que define al nodo del árbol. Por razones de espacio, de algunos métodos ya explicados sólo se escribe el prototipo y el encabezado.

Programa 7.2

```
/* Prototipo de la plantilla de la clase ArbolBinBus. Así, en la clase
➥NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinBus;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥(razón de ser de la estructura tipo árbol) y las direcciones de sus hijos
➥izquierdo y derecho. Se incluye una relación de amistad con la clase
➥ArbolBinBus para que éste pueda tener acceso a sus miembros privados. */
```

```
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo() ;
       void ActualizaInfo(T);
       friend class ArbolBinBus<T>;
};

/* Declaración del método constructor por omisión. Inicializa
las ligas a los subárboles con el valor NULL, indicando que no tiene
➥hijos. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
NodoArbol<T>::RegresaInfo()
{
   return Info ;
}

/* Método para actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbol<T>::ActualizaInfo(T Dato)
{
   Info= Dato ;
}

/* Declaración de la clase ArbolBinBus. Su atributo es un puntero al
➥nodo raíz. */
template <class T>
class ArbolBinBus
{
   private:
       NodoArbol<T> *Raiz;
```

```
   public:
       ArbolBinBus ();
       NodoArbol<T> *RegresaRaiz();
       void Preorden (NodoArbol<T> *);
       void Inorden (NodoArbol<T> *);
       void Postorden (NodoArbol<T> *);
       NodoArbol<T> * Busqueda (NodoArbol<T> *, T);
       void InsertaNodoSinRep (NodoArbol<T> *, T);
       void EliminaNodo (NodoArbol<T> *, T);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL, indicando árbol vacío (no tiene nodos). */
template <class T>
ArbolBinBus<T>::ArbolBinBus()
{
   Raiz= NULL;
}

/* Método que regresa el valor del apuntador a la raíz del árbol. */
template <class T>
NodoArbol<T> *ArbolBinBus<T>::RegresaRaiz()
{
   return Raiz;
}

/* Método que realiza el recorrido preorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Preorden (NodoArbol<T> *Apunt)
{
   /* Ya analizado, razón por la que se omite. */
}

/* Método que realiza el recorrido inorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Inorden (NodoArbol<T> *Apunt)
{
   /* Ya analizado, razón por la que se omite. */
}

/* Método que realiza el recorrido postorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Postorden (NodoArbol<T> *Apunt)
```

```
{
   /* Ya analizado, razón por la que se omite. */
}
/* Método que busca un dato en un árbol binario de búsqueda. Recibe como
➥parámetros la dirección del nodo a visitar (la primera vez es la raíz)
➥y el dato a buscar. Regresa como resultado la dirección del nodo
➥encontrado o el valor NULL, si la búsqueda termina con fracaso. */
template <class T>
NodoArbol<T> * ArbolBinBus<T>::Busqueda (NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
       if (Dato < Apunt->Info)
           return Busqueda(Apunt->HijoIzq, Dato);
       else
           if (Dato > Apunt->Info)
                return Busqueda(Apunt->HijoDer, Dato);
           else
                return Apunt;
   else
       return NULL;
}

/* Método que inserta un nodo en un árbol binario de búsqueda. Recibe como
➥parámetros la dirección del nodo a visitar (la primera vez es la raíz) y
➥la información que se quiere almacenar en el nuevo nodo. En esta imple-
➥mentación no se permite que haya información duplicada en el árbol. */
template <class T>
void ArbolBinBus<T>::InsertaNodoSinRep(NodoArbol<T> *Apunt, T Dato)
{
   /* Ya analizado, razón por la que se omite. */
}
  /* Método que elimina un nodo del árbol binario de búsqueda. Recibe
➥como parámetro la dirección del nodo a visitar (la primera vez es la
➥raíz) y el dato a eliminar. */
template <class T>
void ArbolBinBus<T>::EliminaNodo(NodoArbol<T> *Apunt, T Dato)
{
   /* Ya analizado, razón por la que se omite. */
}
```

El programa 7.3 presenta un ejemplo de aplicación de la estructura árbol binario de búsqueda. Utiliza la clase `Producto` definida en el programa 6.2 del capítulo anterior y la clase `ArbolBinBus` del programa 7.2. Ambas se incluyen por medio de bibliotecas. El programa permite crear un árbol cuyos nodos guardarán la información de los productos; esta información se almacena ordenadamente según su clave. Además, ofrece la opción de dar de baja o buscar un producto y generar un reporte de todos los productos ordenados por claves.

Programa 7.3

```
/* Este programa es para almacenar un conjunto de productos (ordenados
➥por clave), utilizando un árbol binario de búsqueda. Además, se pueden
➥eliminar y buscar productos ya registrados y generar un reporte con la
➥información de todos los productos. La biblioteca "Productos.h" tiene
➥la clase Producto utilizada en el programa 6.2. Por su parte, la
➥biblioteca "ArbolBinBusqueda.h" contiene la plantilla de la clase
➥ArbolBinBus del programa 7.2. */

#include "Productos.h"
#include "ArbolBinBusqueda.h"

/* Función que despliega al usuario las opciones de trabajo. Regresa la
➥opción seleccionada. */
int Menu()
{
   int Opcion;
   do {
       cout<<"\n\n\n\tOpciones de trabajo:\n";
       cout<<"\t1.Ingresar nuevo producto.\n";
       cout<<"\t2.Dar de baja un producto.\n";
       cout<<"\t3.Reporte de todos los productos ordenados por
       ➥clave.\n";
       cout<<"\t4.Buscar un producto por clave.\n";
       cout<<"\t5.Terminar el proceso.\n\n";
       cout<<"\tIngrese opción seleccionada:  ";
       cin>>Opcion;
     } while (Opcion <1 || Opcion > 5);
     return Opcion;
}


/* Función principal desde la cual se controla la ejecución de las
➥operaciones seleccionadas por el usuario. */
void main()
{
   ArbolBinBus<Producto> Inventario;
   NodoArbol<Producto> *Ap1, *Ap2;
   Producto Prod;
   int Opc, Cla;

   do {
       Opc= Menu();
       switch (Opc)
       {
         /* Se registra un nuevo producto. No se aceptan productos con
         ➥claves repetidas. */
```

```
        case 1:{
                cin>>Prod;
                Ap1= Inventario.RegresaRaiz();
                Inventario.InsertaNodoSinRep(Ap1, Prod);
                break;
                }
        /* Se elimina un producto ya registrado. */
        case 2:{
                cout<<"\n\nIngrese la clave del producto a eliminar:";
                cin>>Cla;
                Producto Prod(Cla, "", 0);
                Ap1= Inventario.RegresaRaiz();
                Inventario.EliminaNodo(Ap1, Prod);
                break;
                }
        /* Con el método Inorden se genera un reporte de todos los
        ➥productos ordenados por clave. */
        case 3:{
                Ap1= Inventario.RegresaRaiz();
                cout<<"\n\n\n————————————————————\n\n";
                cout<<"PRODUCTOS EN INVENTARIO\n\n";
                cout<<"————————————————————\n\n";
                Inventario.Inorden(Ap1);
                break;
                }
        /* Se busca un elemento por su clave. Si ya está registrado
        ➥entonces se despliegan todos sus datos. En caso contrario,
        ➥sólo un mensaje informativo. */
        case 4: {

                cout<<"\n\nIngrese la clave del producto a buscar:";
                cin>>Cla;
                Producto Prod(Cla, "", 0);
                Ap1= Inventario.RegresaRaiz();
                Ap2= Inventario.Busqueda(Ap1, Prod);
                if (Ap2)
                {
                    cout<<"\n\n\nExiste un producto registrado con esa
                    ➥clave.\n";
                    cout<<Ap2->RegresaInfo();
                }
                else
                    cout<<"\n\nNo se ha registrado ningún producto con
                          ➥esa clave. \n";
                break;
                }
        case 5: cout<<"\n\n\nFIN DEL PROCESO.\n\n\n";
                break;
        }
    } while (Opc >=1 & &  Opc < 5);
}
```

7.3 Árboles balanceados

Un ***árbol balanceado*** es un árbol binario de búsqueda en el cual la diferencia entre la altura de su subárbol derecho y la altura de su subárbol izquierdo *es menor o igual a 1*. De esta manera se controla el crecimiento del árbol y se garantiza mantener la eficiencia en la operación de búsqueda. La diferencia entre las alturas de los subárboles se conoce como *factor de equilibrio (FE)*, el cual se expresa como se muestra a continuación:

> FE = altura hijo derecho – altura hijo izquierdo

La figura 7.12 muestra un árbol binario de búsqueda en el que cada nodo tiene un factor de equilibrio asociado. La raíz tiene un *FE* igual a **1** ya que el subárbol derecho tiene una altura de 3 y el izquierdo de 2. El nodo que almacena el valor 99 tiene un *FE* igual a **–1** porque su subárbol derecho tiene altura 0 y el izquierdo 1. En cambio, el nodo que guarda el 508 tiene un *FE* igual a **0** porque sus dos subárboles tienen la misma altura. Al observar los *FE* de todos los nodos se puede afirmar que dicho árbol está balanceado, porque éstos son, en valor absoluto, menores o iguales a uno.

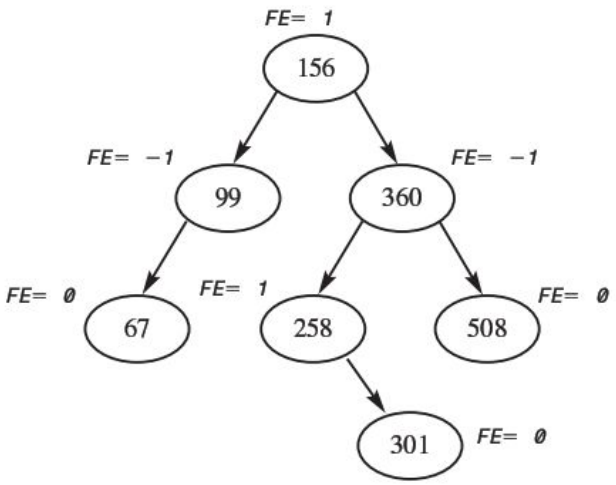


FIGURA 7.12 *Árbol binario de búsqueda con factores de equilibrio*

Considere que los elementos: 24 – 31 – 87 – 99 – 105 y 126 se almacenan, en ese orden, en un árbol binario de búsqueda. Luego de insertar los 6 valores, se obtiene un árbol como el de la figura 7.13. Si se tuviera que buscar un dato en este árbol se tendrían que hacer tantas comparaciones como nodos haya antes de llegar al deseado. En estructuras semejantes se pierden todas las ventajas que ofrecen los árboles. Para evitar que esto suceda surgen los árboles balanceados en los que se realizan balanceos o ajustes de los nodos luego de efectuar inserciones o eliminaciones que hayan provocado la pérdida del equilibrio.

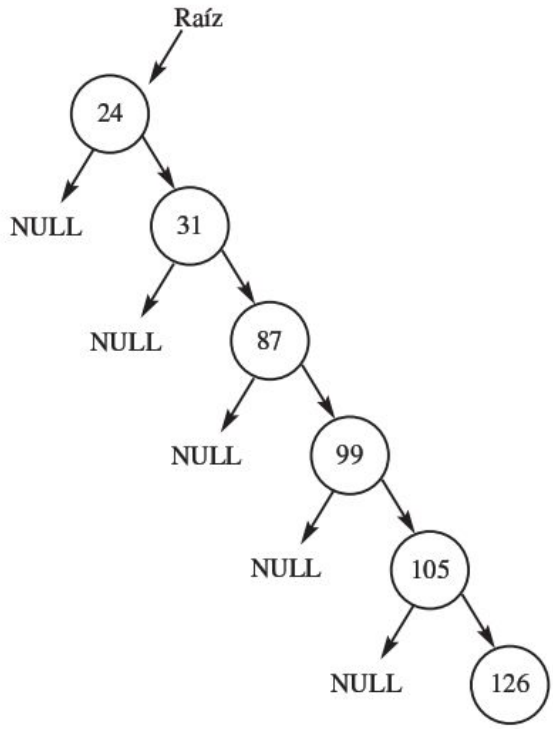


FIGURA 7.13 *Inserción provocando desequilibrio en un árbol binario de búsqueda*

Reacomodo del árbol

Para lograr que un árbol siga estando balanceado, luego de cada inserción o eliminación, se debe determinar si su factor de equilibrio es mayor a 1. En caso afirmativo, se deben reacomodar los nodos de tal manera que se vuelva a tener un valor menor o igual a 1. Este movimiento se denomina **rotación**. La rotación puede ser simple o compuesta dependiendo del número de nodos que participen.

La **rotación simple** se presenta cuando están involucrados dos hijos derechos (HD-HD) o dos hijos izquierdos (HI-HI), y afecta sólo las ligas de dos nodos. La figura 7.14 presenta gráficamente las dos variantes de este tipo de rotación.

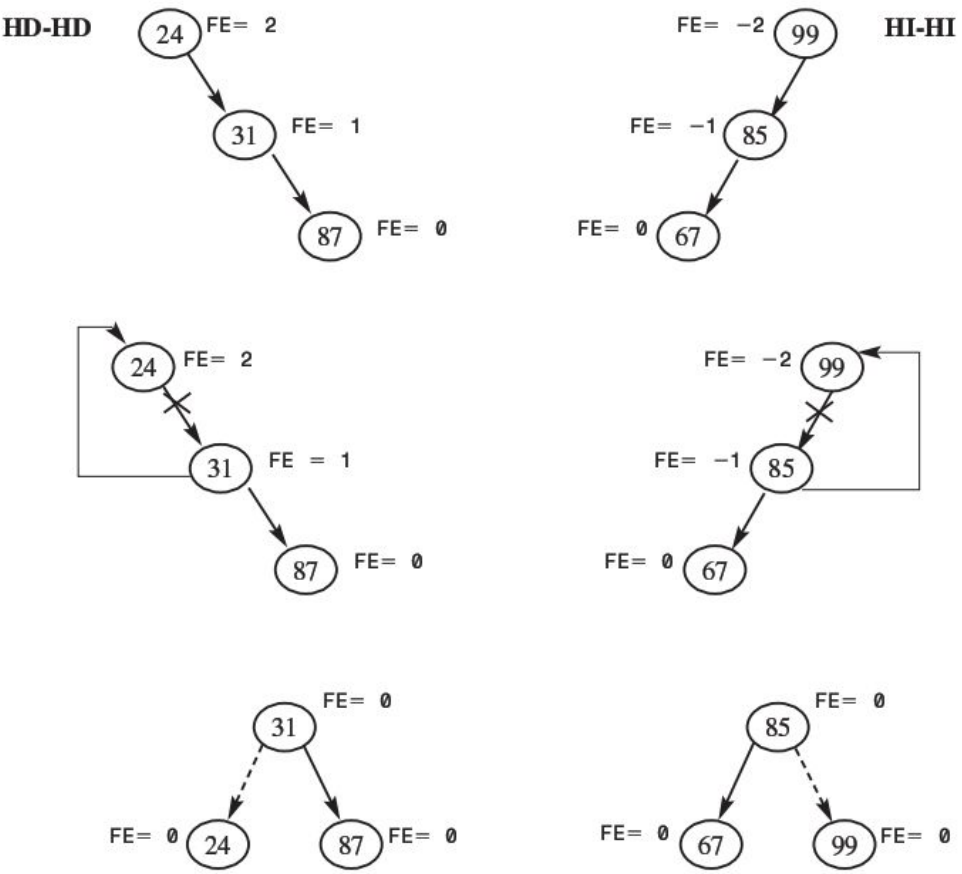


FIGURA 7.14 *Rotación simple*

Analizando el caso de la rotación HD-HD se observa que el nodo con información 24 tiene un *FE* igual a 2, lo cual indica que se ha perdido el equilibrio en su rama derecha. A su vez, en el nodo que guarda el 31 se tiene un *FE* igual a 1, con lo cual se puede determinar que la rotación debe involucrar también a la rama derecha de éste. Luego de efectuar la rotación, este nodo tiene como hijo izquierdo a su padre recuperando así el equilibrio.

En el caso de la rotación HI-HI, se tiene el nodo que almacena el valor 99 con un *FE* igual a –2, lo cual indica que se ha perdido el equilibrio en su rama izquierda. Además, el nodo con información 85 tiene un *FE* igual a –1, con lo cual se puede determinar que la rotación también debe involucrar a la rama izquierda de éste. Luego de efectuar la rotación, este último nodo tiene como hijo derecho a su padre.

Más adelante se explica cómo deben actualizarse los punteros a los nodos afectados durante la rotación simple.

La **rotación compuesta** se presenta cuando están involucrados un hijo derecho y un hijo izquierdo (HD-HI) o un hijo izquierdo y uno derecho (HI-HD), y afecta las ligas de tres nodos. La figura 7.15 presenta gráficamente las dos variantes de este tipo de rotación.

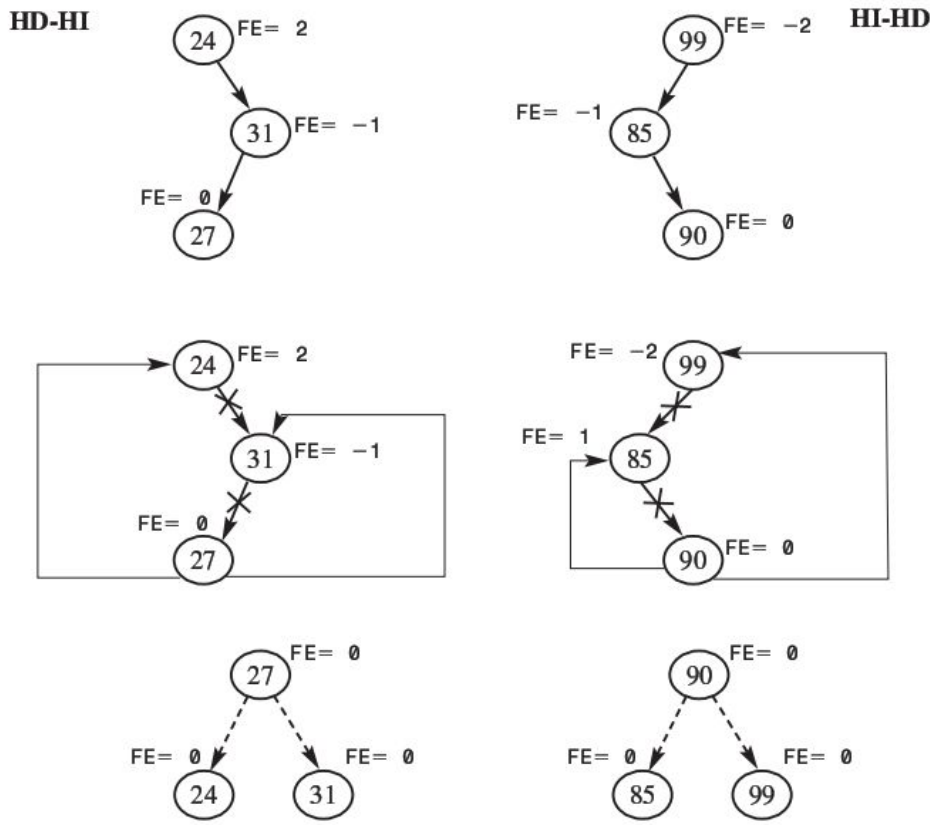


FIGURA 7.15 *Rotación compuesta*

Analizando el caso de la rotación HD-HI se observa que el nodo con información 24 tiene un *FE* igual a 2, lo cual indica que se ha perdido el equilibrio en su rama derecha. A su vez, en el nodo que guarda el 31 se tiene un *FE* igual a –1, lo que implica que su rama izquierda tiene mayor altura que su rama derecha. Por tanto, la rotación que se debe aplicar es la de hijo derecho-hijo izquierdo. Luego de efectuar la rotación, el nodo 27 queda como padre del 24 (antes su *abuelo*) y del 31 (antes su padre).

En el caso de la rotación HI-HD se observa que el nodo con información 99 tiene un *FE* igual a –2, lo cual indica que se ha perdido el equilibrio en su rama izquierda. Por otra parte, su hijo izquierdo (el nodo que guarda el 85) tiene un *FE* igual a 1, lo que implica que su rama derecha tiene mayor altura que su rama izquierda. Por lo tanto, la rotación que se debe aplicar es la de hijo izquierdo-hijo derecho. Luego de efectuar la rotación, el nodo que almacena el 90 queda como padre del 99 (antes su *abuelo*) y del 85 (antes su padre).

Más adelante se explica cómo deben actualizarse los punteros a los nodos afectados durante la rotación compuesta.

En los árboles balanceados, cada nodo debe almacenar (además de los elementos ya vistos) su factor de equilibrio. Luego de insertar o eliminar un nodo se calculan los factores de equilibrio de todos los nodos involucrados en la operación y, dependiendo del valor resultante, se procede a realizar la rotación que corresponda y a hacer la reasignación. La plantilla de la clase nodo se define como se muestra a continuación.

```
/* Plantilla de la clase nodo de un árbol balanceado. Se incluye un
➥nuevo atributo, llamado FE, para almacenar el factor de equilibrio del
➥nodo. Se establece una relación de amistad con la clase ArbolBalanceado
➥para que ésta pueda tener acceso a sus miembros privados. */
template <class T>
class NodoArbolBal
{
   private:
       NodoArbolBal<T> *HijoIzq;
       NodoArbolBal<T> *HijoDer;
       T Info;
       int FE;
   public:
       NodoArbolBal();
       T RegresaInfo();
       void ActualizaInfo(T) ;
       friend class ArbolBalanceado<T>;
};
```

Inserción en árboles balanceados

La **inserción** de un nuevo nodo se lleva a cabo atendiendo las características de los árboles binarios de búsqueda, pero teniendo en cuenta además la altura de los subárboles, de tal manera que no se viole lo mencionado sobre el factor de equilibrio. Los casos que pueden presentarse son:

1. La altura del subárbol derecho es igual a la altura del subárbol izquierdo, por lo tanto, sin importar dónde se realice la inserción, el equilibrio no se pierde.
2. La altura del subárbol derecho es mayor que la altura del subárbol izquierdo, por lo tanto, si la inserción no afecta la altura del subárbol derecho no se requiere rotación, en caso contrario sí.
3. La altura del subárbol izquierdo es mayor que la altura del subárbol derecho, por lo tanto, si la inserción no afecta la altura del subárbol izquierdo no se requiere rotación, en caso contrario sí.

Cualquiera que sea la situación, se procede a insertar el nuevo nodo y luego se actualizan los factores de equilibrio procediendo a la rotación de los nodos si correspondiera. El método para realizar esta operación se presenta a continuación. Las rotaciones simples y compuestas se escribieron como métodos independientes.

```
/* Método que realiza la rotación simple HI-HI en un árbol balanceado.
➥Además, reasigna el FE del nodo involucrado en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHI_HI(NodoArbolBal<T>
                ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   Apunt->HijoIzq= ApAux1->HijoDer;
   ApAux1->HijoDer= Apunt;
   Apunt->FE= 0;
   return ApAux1;
}

/* Método que realiza la rotación simple HD-HD en un árbol balanceado.
➥Además, reasigna el FE del nodo involucrado en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHD_HD(NodoArbolBal<T>
                ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   Apunt->HijoDer= ApAux1->HijoIzq;
   ApAux1->HijoIzq= Apunt;
   Apunt->FE= 0;
   return ApAux1;
}
```

```
/* Método que realiza la rotación compuesta HI-HD en un árbol balanceado.
➥Además, reasigna los FE de los nodos involucrados en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHI_HD(NodoArbolBal<T>
               ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   NodoArbolBal<T> *ApAux2;
   ApAux2= ApAux1->HijoDer;
   Apunt->HijoIzq= ApAux2->HijoDer;
   ApAux2->HijoDer= Apunt;
   ApAux1->HijoDer= ApAux2->HijoIzq;
   ApAux2->HijoIzq= ApAux1;
   if (ApAux2->FE == -1)
        Apunt->FE= 1;
   else
        Apunt->FE= 0;
   if (ApAux2->FE == 1)
        ApAux1->FE= -1;
   else
        ApAux1->FE= 0;
   return ApAux2;
}

/* Método que realiza la rotación compuesta HD-HI en un árbol balanceado.
➥Además, reasigna los FE de los nodos involucrados en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHD_HI(NodoArbolBal<T>
               ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   NodoArbolBal<T> *ApAux2;
   ApAux2= ApAux1->HijoIzq;
   Apunt->HijoDer= ApAux2->HijoIzq;
   ApAux2->HijoIzq= Apunt;
   ApAux1->HijoIzq= ApAux2->HijoDer;
   ApAux2->HijoDer= ApAux1;
   if (ApAux2->FE == 1)
        Apunt->FE= -1;
   else
        Apunt->FE= 0;
   if (ApAux2->FE == -1)
        ApAux1->FE= 1;
   else
        ApAux1->FE= 0;
   return ApAux2;
}

/* Método que inserta un nuevo elemento en un árbol balanceado. Recibe
➥como parámetros el dato a insertar, un puntero al nodo a visitar (la
➥primera vez es la raíz) y un entero (Band) que la primera vez trae el
➥valor 0.*/
```

```
template <class T>
void ArbolBalanceado<T>::InsertaBalanceado(T Dato,NodoArbolBal<T>
➥*Apunt,int *Band)
{
   NodoArbolBal<T> *ApAux1, *ApAux2;
   if (Apunt != NULL)
   {
        if (Dato < Apunt->Info)
        {
            /* Se invoca el método con el subárbol izquierdo. */
            InsertaBalanceado(Dato, Apunt->HijoIzq, Band);
            Apunt->HijoIzq= Raiz;
            if (0 < *Band)    /* Verifica si creció el subárbol
                              ➥izquierdo. */
                switch ( Apunt->FE )
                {
                   case 1: {
                            Apunt->FE = 0;
                            *Band= 0;
                            break;
                            }
                   case 0: {
                            Apunt->FE = -1;
                            break;
                           }
                   case -1: {
                            ApAux1 = Apunt->HijoIzq;
                            if (ApAux1->FE <= 0)
                                Apunt= RotacionHI_HI(Apunt, ApAux1);
                            else
                                Apunt= RotacionHI_HD(Apunt, ApAux1);
                            Apunt->FE = 0;
                            *Band = 0;
                           }
                }
        }
        else
            if (Dato > Apunt->Info)
            {
                 /* Invoca el método con el subárbol derecho. */
                 InsertaBalanceado(Dato, Apunt->HijoDer,Band);
                 Apunt->HijoDer= Raiz;
                 if (0 < *Band)      /* Verifica si creció el
                                     ➥subárbol derecho. */
                     switch ( Apunt->FE )
                     {
                         case -1: {
                                  Apunt->FE = 0;
                                  *Band= 0;
                                   break;
                                   }
```

```
                                case 0: {
                                          Apunt->FE = 1;
                                          break;
                                         }
                                case 1: {
                                          ApAux1= Apunt->HijoDer;
                                        if (ApAux1->FE >= 0)
                                            Apunt= RotacionHD_HD(Apunt,
                                            ➥ApAux1);
                                        else
                                             Apunt= RotacionHD_HI(Apunt,
                                             ➥ApAux1);
                                        Apunt->FE = 0;
                                        *Band= 0;
                                        }
                                }
                    }
                    Raiz= Apunt;
          }
          else
          {
               /* Inserta un nuevo nodo y actualiza el valor de Band indicando
               ➥que el árbol creció. */
               ApAux2= new  NodoArbolBal<T>();
               ApAux2->Info= Dato;
               ApAux2->FE= 0;
               *Band = 1;
               Raiz= ApAux2;
          }
}
```

Eliminación en árboles balanceados

La **eliminación** de un nodo se lleva a cabo atendiendo las características de los árboles binarios de búsqueda, pero teniendo en cuenta además la altura de los subárboles, de tal manera que no se viole lo mencionado sobre el factor de equilibrio. Los casos que pueden presentarse son:

1. La altura del subárbol derecho es igual a la altura del subárbol izquierdo, por lo tanto, sin importar dónde se realice la eliminación, el equilibrio no se pierde.
2. La altura del subárbol derecho es mayor que la altura del subárbol izquierdo, por lo tanto, si la eliminación no afecta la altura del subárbol izquierdo no se requiere rotación, en caso contrario sí.

3. La altura del subárbol izquierdo es mayor que la altura del subárbol derecho, por lo tanto, si la eliminación no afecta la altura del subárbol derecho no se requiere rotación, en caso contrario sí.

Cualquiera que sea la situación, se elimina el nodo y luego se actualizan los factores de equilibrio de todos los nodos involucrados, procediendo a su rotación, si correspondiera. Este proceso termina cuando se llega a la raíz.

A continuación se presenta el método para realizar esta operación. Se reutilizan los métodos vistos para las rotaciones compuestas (los cuales no se vuelven a presentar). En el caso de los correspondientes a las rotaciones simples, dado que no se ajustan totalmente a la eliminación, se dan con los cambios requeridos. Se definieron dos métodos auxiliares que ayudan a la reestructuración del árbol si éste pierde el equilibrio.

```
/* Método auxiliar del método EliminaBalanceado que reestructura el
➥árbol cuando la altura de la rama izquierda ha disminuido. */
template <class T>
NodoArbolBal<T>* ArbolBalanceado<T>::RestructuraI(NodoArbolBal<T> *Nodo,
                                                  int *Aviso)
{
   NodoArbolBal<T> *ApAux;
   if ( *Aviso > 0)
   {
         switch (Nodo->FE)
         {
            case -1: Nodo->FE= 0;
                     break;
            case 0:  Nodo->FE= 1;
                     *Aviso= 0;
                     break;
            case 1:  ApAux= Nodo->HijoDer;
                     if (ApAux->FE >= 0)   //Rotación HD-HD
                     {
                          Nodo->HijoDer= ApAux->HijoIzq;
                          ApAux->HijoIzq= Nodo;
                          switch (ApAux->FE)
                          {
                             case 0:  Nodo->FE= 1;
                                      ApAux->FE= -1;
                                      *Aviso= 0;
                                      break;
                             case 1:  Nodo->FE= 0;
                                      ApAux->FE= 0;
                                      break;
```

```
                        }
                        Nodo= ApAux;
                    }
                    else     //Rotación HD-HI
                    {
                        Nodo= RotacionHD_HI(Nodo, ApAux);
                        Nodo->FE= 0;
                    }
                    break;
                }
    }
    return Nodo;
}

/* Método auxiliar del método EliminaBalanceado que reestructura el
➥árbol cuando la altura de la rama derecha ha disminuido. */
template <class T>
NodoArbolBal<T>* ArbolBalanceado<T>::RestructuraD(NodoArbolBal<T> *Nodo,
                                                  int *Aviso)
{
    NodoArbolBal<T> *ApAux;
      if (*Aviso > 0)
      {
           switch (Nodo->FE)
           {
              case 1:  Nodo->FE= 0;
                       break;
              case 0:  Nodo->FE= -1;
                       *Aviso= 0;
                       break;
              case -1:  ApAux= Nodo->HijoIzq;
                       if (ApAux->FE <= 0)  //Rotación HI-HI
                       {
                           Nodo->HijoIzq= ApAux->HijoDer;
                           ApAux->HijoDer= Nodo;
                           switch (ApAux->FE)
                           {
                              case 0:  Nodo->FE= -1;
                                       ApAux->FE= 1;
                                       *Aviso= 0;
                                       break;
                              case -1: Nodo->FE= 0;
                                       ApAux->FE= 0;
                                       break;
                           }
                            Nodo= ApAux;
                       }
```

```
                    else       //Rotación HI-HD
                    {
                        Nodo= RotacionHI_HD(Nodo, ApAux);
                        Nodo->FE= 0;
                    }
                    break;
            }
        }
        return Nodo;
}

/* Método auxiliar del método EliminaBalanceado que sustituye el
➥elemento que se desea eliminar por el que se encuentra más a la derecha
➥del subárbol izquierdo. */
template <class T>
void ArbolBalanceado<T>::Sustituye(NodoArbolBal<T> *Nodo,
                        ➥NodoArbolBal<T> *ApAux, int *Avisa)
{
    if (Nodo->HijoDer != NULL)
    {
        Sustituye (Nodo->HijoDer, ApAux, Avisa);
        if (ApAux->HijoIzq == NULL)
            Nodo->HijoDer= NULL;
        else
            Nodo->HijoDer= ApAux->HijoIzq;
        Nodo= RestructuraD(Nodo, Avisa);
    }
    else
    {
        ApAux->Info= Nodo->Info;
        Nodo= Nodo->HijoIzq;
        *Avisa= 1;
    }
    ApAux->HijoIzq= Nodo;
}

/* Método que elimina un elemento en un árbol balanceado. Luego de la
➥eliminación se actualizan los factores de equilibrio de cada nodo hasta
➥la raíz y se reestructura el árbol si fuera necesario. */
template <class T>
void ArbolBalanceado<T>::EliminaBalanceado(NodoArbolBal<T> *Apunt,
                        ➥NodoArbolBal<T> *ApAnt, int *Avisa, T Dato)
{
    NodoArbolBal<T> *ApAux;
    int Bandera;
    if (Apunt != NULL)
        if (Dato < Apunt->Info)
```

```
{
     if (*Avisa > 0)
          Bandera= 1;
     else
         if (*Avisa != 0)
              Bandera= -1;
     *Avisa= -1;
     EliminaBalanceado(Apunt->HijoIzq, Apunt, Avisa, Dato);
     Apunt= RestructuraI(Apunt, Avisa);
     if (ApAnt != NULL)
         switch (Bandera)
         {
             case -1:  ApAnt->HijoIzq= Apunt;
                       break;
             case 1:  ApAnt->HijoDer= Apunt;
                       break;
             default:  break;
         }
     else
         Raiz= Apunt;
}
else
{
     if  (Dato > Apunt->Info)
     {
          if (*Avisa < 0)
               Bandera= -1;
          else
               if (*Avisa != 0)
                    Bandera=1;
          *Avisa= 1;
          EliminaBalanceado(Apunt->HijoDer, Apunt, Avisa, Dato);
          Apunt= RestructuraD(Apunt,Avisa);
          if (ApAnt != NULL)
             switch (Bandera)
             {
                case -1: ApAnt->HijoIzq= Apunt;
                         break;
                case 1:  ApAnt->HijoDer= Apunt;
                         break;
                default: break;
             }
          else
              Raiz= Apunt;
     }
     else
     {
         ApAux= Apunt;
         if (ApAux->HijoDer == NULL)
```

```
                    {
                        Apunt= ApAux->HijoIzq;
                        if (*Avisa != 0)
                             if (*Avisa < 0)
                                  ApAnt->HijoIzq= Apunt;
                             else
                                  ApAnt->HijoDer= Apunt;
                        else
                             if (Apunt == NULL)
                                 Raiz= NULL;
                             else
                                 Raiz= Apunt;
                             *Avisa= 1;
                    }
                    else
                         if (ApAux->HijoIzq == NULL)
                         {
                             Apunt= ApAux->HijoDer;
                             if (*Avisa != 0)
                                  if (*Avisa < 0)
                                       ApAnt->HijoIzq= Apunt;
                                  else
                                       ApAnt->HijoDer= Apunt;
                             else
                                 if (Apunt == NULL)
                                     Raiz= NULL;
                                 else
                                      Raiz= Apunt;
                             *Avisa= 1;
                    }
                    else
                    {
                         Sustituye (ApAux->HijoIzq, ApAux, Avisa);
                         Apunt= RestructuraI(Apunt, Avisa);
                         if (ApAnt != NULL)
                              if (*Avisa <= 0)
                                   ApAnt->HijoIzq= Apunt;
                              else
                                  ApAnt->HijoDer= Apunt;
                         else
                              Raiz= Apunt;
                    }
               }
    else
         cout<<"\n\nEl dato a eliminar no se encuentra registrado.\n\n";
}
```

El programa 7.4 presenta la plantilla de la clase `ArbolBalanceado` con los encabezados de los métodos analizados, los cuales no se repiten. Se incluye un método para la impresión de todos los nodos junto con su factor de equilibrio.

Programa 7.4

```
/* Prototipo de la plantilla de la clase ArbolBalanceado. De esta
➥manera, en la clase NodoArbolBal se podrá hacer referencia a ella. */

template <class T>
class ArbolBalanceado;

/* Declaración de la clase de un nodo de un árbol balanceado. Además de
➥almacenar la información, las direcciones de los hijos izquierdo y
➥derecho, guarda el factor de equilibrio. */
template <class T>
class NodoArbolBal
{
   private:
       NodoArbolBal<T> *HijoIzq;
       NodoArbolBal<T> *HijoDer;
       T Info;
       int FE;
   public:
       NodoArbolBal();
       T RegresaInfo();
       void ActualizaInfo(T);
       friend class ArbolBalanceado<T>;
};

/* Declaración del método constructor. Inicializa los apuntadores a
➥ambos hijos con el valor de NULL, indicando vacío. */
template <class T>
NodoArbolBal<T>::NodoArbolBal()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
T NodoArbolBal<T>::RegresaInfo()
{
   return Info;
}
```

```
/* Método que permite actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbolBal<T>::ActualizaInfo(T Dato)
{
   Info= Dato;
}

/* Declaración de la clase ArbolBalanceado. Se incluyen sólo los proto-
➥tipos de los métodos presentados más arriba. */
template <class T>
class ArbolBalanceado
{
   private:
       NodoArbolBal<T> *Raiz;
   public:
       ArbolBalanceado ();
       NodoArbolBal<T> * RegresaRaiz();
       NodoArbolBal<T> * Busca (NodoArbolBal<T> *, T) ;
       void InsertaBalanceado (T, NodoArbolBal<T> *, int *);
       NodoArbolBal<T> * RotacionHI_HD (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHD_HI (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHI_HI (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHD_HD (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RestructuraI (NodoArbolBal<T> *, int *);
       NodoArbolBal<T> * RestructuraD (NodoArbolBal<T> *, int *);
       void EliminaBalanceado (NodoArbolBal<T> *, NodoArbolBal<T> *,
                               ➥int *, T);
       void Sustituye (NodoArbolBal<T> *,NodoArbolBal<T> *,  int *);
       void Imprime (NodoArbolBal<T> *);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL, indicando que el árbol está vacío. */
template <class T>
ArbolBalanceado<T>::ArbolBalanceado()
{
   Raiz= NULL;
}

/* Método que regresa el apuntador a la raíz del árbol.*/
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RegresaRaiz()
{
   return Raiz;
}
```

```
/* Método que busca un valor dado como parámetro en el árbol. Recibe
➥como parámetros el puntero del nodo a visitar (la primera vez es la
➥raíz) y el dato a buscar. Regresa el puntero al nodo donde lo encontró
➥o NULL si no está en el árbol. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::Busca (NodoArbolBal<T> *Apunt, T
Dato)
{
   if (Apunt != NULL)
        if (Apunt->Info == Dato)
            return Apunt;
        else
            if (Apunt->Info > Dato)
                return Busca(Apunt->HijoIzq, Dato);
            else
                return Busca(Apunt->HijoDer, Dato);
   else
        return NULL;
}

/* Método que imprime el contenido del árbol. Recibe como parámetro el
➥apuntador al nodo a visitar (la primera vez es la raíz del árbol).
➥Utiliza recorrido inorden para que el contenido se imprima en orden
➥creciente. */
template <class T>
void ArbolBalanceado<T>::Imprime(NodoArbolBal<T> *Apunt)
{
   if (Apunt != NULL)
   {
        Imprime(Apunt->HijoIzq);
        cout<<Apunt->Info <<"\n\n";
        Imprime(Apunt->HijoDer);
   }
}
```

A continuación se presenta una aplicación de árboles balanceados. Se define una clase `Fabrica` para almacenar los datos más importantes de una fábrica y las operaciones que pueden realizarse sobre ellos. Esta clase se almacena en la biblioteca "*Fabricas.h*". El programa 7.5 muestra esta clase y el programa 7.6, la aplicación de árboles en la cual se incluyen la biblioteca mencionada y "*ArbolBalanceado.h*" que corresponde a la plantilla de árboles balanceados del programa 7.4.

Programa 7.5

```
/* Definición de la clase Fabrica. Se incluyen varios operadores
➥sobrecargados para que puedan ser utilizados por los métodos de la
➥clase ArbolBalanceado. Asimismo, se declaran como amigos los operadores
➥de entrada (>>) y de salida (<<) para que objetos de este tipo puedan
➥leerse e imprimirse directamente con cin y cout respectivamente. */
class Fabrica
{
   private:
       int Clave;
       char Nombre[MAX], Domicilio[MAX], Telefono[MAX];
   public:
       Fabrica();
       Fabrica(int, char [], char[], char[]);
       void CambiaDomic(char[]);
       void CambiaTelef(char[]);
       int operator > (Fabrica);
       int operator < (Fabrica);
       int operator == (Fabrica);
       friend istream & operator>> (istream & , Fabrica & );
       friend ostream & operator<< (ostream & , Fabrica & );
};

/* Declaración del método constructor por omisión. */
Fabrica::Fabrica()
{}

/* Declaración del método constructor con parámetros. */
Fabrica::Fabrica(int Cla, char Nom[], char Domic[], char Tel[])
{
   Clave= Cla;
   strcpy(Nombre, Nom);
   strcpy(Domicilio, Domic);
   strcpy(Telefono, Tel);
}

/* Método que actualiza el domicilio de una fábrica. */
void Fabrica::CambiaDomic(char NuevoDom[])
{
   strcpy(Domicilio, NuevoDom);
}

/* Método que actualiza el teléfono de una fábrica. */
void Fabrica::CambiaTelef(char NuevoTel[])
{
   strcpy(Telefono, NuevoTel);
}
```

```
/* Sobrecarga del operador > lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator > (Fabrica ObjFab)
{
   if (Clave > ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador < lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator < (Fabrica ObjFab)
{
   if (Clave < ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador == lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator == (Fabrica ObjFab)
{
   if (Clave == ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador >> para permitir la lectura de objetos de
➥tipo Fabrica de manera directa con el cin. */
istream & operator>> (istream & Lee, Fabrica & ObjFab)
{
   cout<<"\n\nIngrese nombre de la fábrica:";
   Lee>>ObjFab.Nombre;
   cout<<"\n\nIngrese clave de la fábrica:";
   Lee>>ObjFab.Clave;
   cout<<"\n\nIngrese domicilio de la fábrica:";
   Lee>>ObjFab.Domicilio;
   cout<<"\n\nIngrese teléfono de la fábrica:";
   Lee>>ObjFab.Telefono;
   return Lee;
}

/* Sobrecarga del operador << para permitir la impresión de objetos de
➥tipo Fabrica de manera directa con el cout. */
ostream & operator<<(ostream & Escribe, Fabrica & ObjFab)
```

```
{
   cout<<"\n\nDatos de la fábrica\n";
   Escribe<<"Nombre:  "<<ObjFab.Nombre<<endl;
   Escribe<<"Clave:  "<<ObjFab.Clave<<endl;
   Escribe<<"Domicilio:  "<<ObjFab.Domicilio<<endl;
   Escribe<<"Teléfono:    "<<ObjFab.Telefono<<endl;
   return Escribe;
}
```

Programa 7.6

```
/* Programa que utiliza un árbol balanceado para almacenar ordenadamente
➥los datos de ciertas fábricas. El usuario puede dar de alta nuevas
➥fábricas, eliminar alguna ya registrada, obtener un reporte de todas
➥(ordenadas según su clave) y actualizar sus direcciones y teléfonos. Se
➥incluyen dos bibliotecas, una con la plantilla de la clase ArbolBalan-
➥ceado presentada en el programa 7.4 y la otra con la clase Fabrica del
➥programa 7.5. */

#include "ArbolBalanceado.h"
#include "Fabricas.h"

/* Función que despliega en pantalla las opciones de trabajo para el
➥usuario. */
int Menu()
{
   int Opc;
   do {
       cout<<"\n\n\t\tOpciones de trabajo.\n\n\n";
       cout<<"(1) Capturar los datos de una fábrica.\n";
       cout<<"(2) Dar de baja una fábrica.\n";
       cout<<"(3) Imprimir los datos de todas las fábricas, ordenadas
       ➥por clave.\n";
       cout<<"(4) Cambiar el domicilio de una fábrica.\n";
       cout<<"(5) Cambiar el teléfono de una fábrica.\n";
       cout<<"(6) Terminar la sesión de trabajo.\n\n";
       cout<<"Ingrese la opción seleccionada:";
       cin>>Opc;
   } while (Opc > 6 || Opc < 1);
   return Opc;
}
```

```
/* Función principal. De acuerdo a la opción de trabajo seleccionada por
➥el usuario invoca los métodos que correspondan. */
void main()
{
   ArbolBalanceado<Fabrica> Proveedores;
   NodoArbolBal<Fabrica> *Apunt1, *Apunt2;
     Fabrica Prov;
     int Operac, Band, Clave;
     char NuevoDom[MAX], NuevoTel[MAX];

   do {
     Operac= Menu();
     switch (Operac)
     {
        /* Se registra una nueva fábrica siempre que la clave dada por
        ➥el usuario no se encuentre en el árbol. */
        case 1: {
                 cin>>Prov;
                 Band= 0;
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.InsertaBalanceado(Prov, Apunt1, &Band);
                 break;
                 }
        /* En caso de dar de baja una fábrica registrada, se solicita sólo
        ➥la clave ya que es el dato que identifica a cada elemento. */
        case 2: {
                 cout<<"\n\nIngrese la clave de la fábrica a eliminar:";
                 cin>>Clave;
                 Fabrica Prov(Clave, "","","");
                 Band= 0;
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.EliminaBalanceado(Apunt1, NULL,
                 ➥&Band, Prov);
                 break;
                 }
        /* Se imprimen los datos de todas las fábricas, ordenadas de
        ➥menor a mayor por clave. */
        case 3: {
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.Imprime(Apunt1);
                 break;
                 }
        /* Se actualiza la dirección de una fábrica. Para llevar a cabo
        ➥esta operación, primero se debe encontrar la fábrica de interés,
        ➥luego recuperar todo el objeto, actualizar el domicilio y pos-
        ➥teriormente redefinir el contenido del nodo con el objeto ya
        ➥modificado. */
```

```
        case 4: {
                cout<<"\n\nIngrese la clave de la fábrica:";
                cin>>Clave;
                cout<<"\n\nIngrese nuevo domicilio:";
                cin>>NuevoDom;
                Fabrica Prov(Clave, "","","");
                Apunt1= Proveedores.RegresaRaiz();
                Apunt2= Proveedores.Busca(Apunt1, Prov);
                if (Apunt2)
                {
                    Prov= Apunt2->RegresaInfo();
                    Prov.CambiaDomic(NuevoDom);
                    Apunt2->ActualizaInfo(Prov);
                }
                else
                    cout<<"\n\nEsa fábrica no está registrada. \n";
                break;
                }
        /* Se actualiza el teléfono de una fábrica. Para llevar a cabo
        ➥esta operación, primero se debe encontrar la fábrica de interés,
        ➥luego recuperar todo el objeto, actualizar el teléfono y poste-
        ➥riormente redefinir el contenido del nodo con el objeto ya
        ➥modificado. */
        case 5: {
                cout<<"\n\nIngrese la clave de la fábrica:  ";
                cin>>Clave;
                cout<<"\n\nIngrese nuevo teléfono:  ";
                cin>>NuevoTel;
                Fabrica Prov(Clave, "", "", "");
                Apunt1= Proveedores.RegresaRaiz();
                Apunt2= Proveedores.Busca(Apunt1,  Prov);
                if (Apunt2)
                {
                    Prov= Apunt2->RegresaInfo();
                    Prov.CambiaTelef(NuevoTel);
                    Apunt2->ActualizaInfo(Prov);
                }
                else
                    cout<<"\n\nEsa fábrica no está registrada. \n";
                break;
                }
        }
    } while (Operac < 6);
}
```

7.4 Árboles-B

Las estructuras tipo árboles estudiadas hasta aquí son utilizadas para almacenar información en la memoria principal de la computadora. Sin embargo, en prácticamente todas las aplicaciones se requiere que los datos a procesar se guarden en dispositivos secundarios, de tal manera que permanezcan aún después de terminado el procesamiento. Además, el volumen de información manejado exige el uso de medios externos de almacenamiento. Por lo tanto, resulta necesario contar con estructuras que permitan organizar la información guardada en archivos. Los ***árboles-B*** son una variante de los árboles balanceados y cubren esa necesidad. En estas estructuras, a cada nodo se le conoce con el nombre de página y las páginas se guardan en algún dispositivo de almacenamiento secundario.

Las principales características de un árbol-B de grado n son:

- La página raíz almacena como mínimo 1 dato y como máximo 2n datos.
- La página raíz tiene como mínimo 2 descendientes.
- Las páginas intermedias y hojas almacenan entre n y 2n datos.
- Las páginas intermedias tienen entre (n+1) y (2n+1) páginas descendientes.
- Todas las páginas hojas tienen la misma altura.
- La información guardada en las páginas se encuentra ordenada.

La figura 7.16 presenta un ejemplo de un árbol-B, de grado 2. En la raíz se almacenan dos datos, lo que origina que tenga tres descendientes. La página hoja que está más a la izquierda guarda todos los datos que son menores al primer dato (105) de la página raíz, la segunda hoja contiene los datos mayores a 105 y menores a 320, mientras que la tercera hoja almacena los datos mayores al segundo dato de la página raíz (320). Cada una de las páginas hojas tiene entre 2 y 4 elementos. Si no fueran hojas, tendrían: la primera 3, la segunda 5 y la tercera 4 páginas descendientes respectivamente.

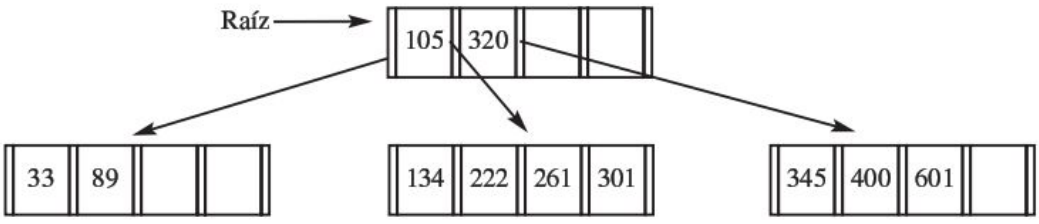


FIGURA 7.16 *Ejemplo de un árbol-B*

Las operaciones que pueden realizarse sobre la información almacenada en árboles-B son: búsqueda, inserción y eliminación. Estas operaciones están implementadas en herramientas diseñadas especialmente para el manejo de archivos, por lo que generalmente no se requiere su programación. En esta sección se analizan para que usted pueda entenderlas y cuando haga uso de algún manejador de archivos sepa qué está pasando internamente con los datos.

Búsqueda en árboles-B

La operación de **búsqueda** es similar a la estudiada en los árboles binarios de búsqueda, por ser los árboles-B una generalización de los primeros. En este tipo de árboles se recuperan del medio secundario de almacenamiento páginas completas de información y se procede a buscar en ellas el dato deseado. Si se encuentra, termina la búsqueda; en caso contrario se procede con la página que corresponda según el dato con el cual se compara, ya sea menor o mayor. Si se requiere recuperar una nueva página pero no existe, entonces se tiene un caso de fracaso. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.
   2.1. Si la respuesta es afirmativa, entonces la búsqueda termina con fracaso.
   2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato buscado con cada elemento almacenado en la página.
   3.1. Si es igual, la búsqueda termina con éxito.
   3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   3.3. Si es mayor, se avanza al siguiente dato de la misma página.
      3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.3.2. Si no es el último, se regresa al paso 3.

Considere el árbol-B de la figura 7.17. Si se quisiera encontrar el valor 319 se procedería, siguiendo el algoritmo dado, tal como se muestra en la tabla 7.4.

TABLA 7.4 Operación de búsqueda en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los datos: 105 – 320
2	Se evalúa si está vacía. En este caso la respuesta es negativa.
3	Se compara el dato buscado (319) con el primer elemento de la página (105). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato buscado (319) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los datos: 134 – 222 – 261 – 301.
6	Se evalúa si está vacía. En este caso la respuesta es negativa.
7	Se compara el dato buscado (319) con el primer elemento de la página (134). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (222).
8	Se compara el dato buscado (319) con el valor 222. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (261).
9	Se compara el dato buscado (319) con el 261. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (301).
10	Se compara el dato buscado (319) con el valor 301. Es mayor y ya no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 301.
11	Se recupera la página con los datos: 310 – 319.
12	Se evalúa si está vacía. En este caso la respuesta es negativa.
13	Se compara el dato buscado (319) con el primer elemento de la página (310). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (319).
14	Se compara el dato buscado (319) con el valor 319. Es igual, por lo tanto la búsqueda termina con éxito.

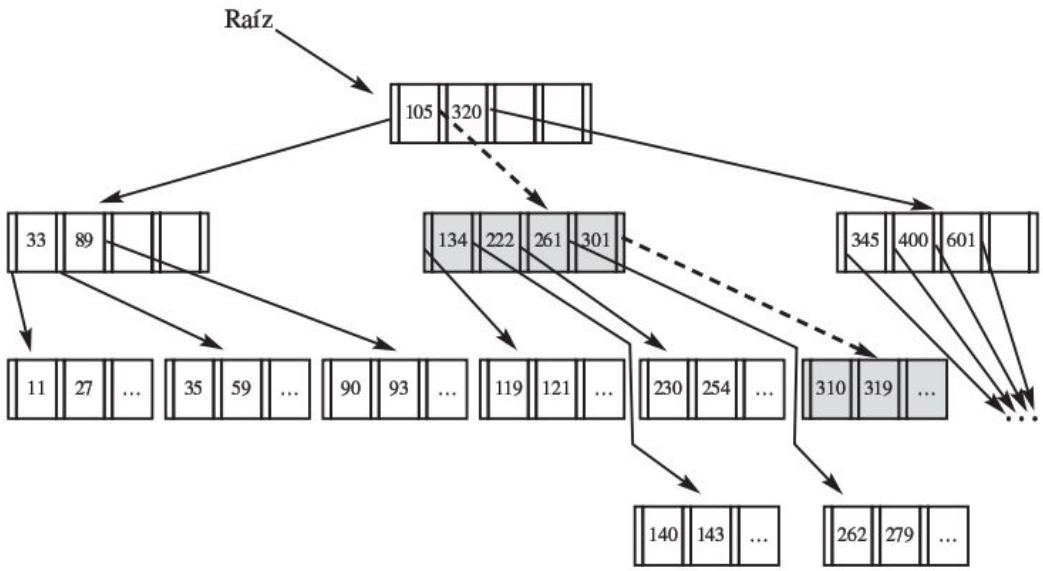


FIGURA 7.17 *Búsqueda en árboles-B*

En la figura 7.17 las líneas punteadas indican las páginas que se van visitando hasta llegar al dato buscado. Por su parte, las páginas sombreadas son las que se recuperan para hacer la comparación del elemento buscado con los elementos almacenados en el árbol.

Inserción en árboles-B

La operación de **inserción** se caracteriza porque los nuevos elementos siempre se guardan a nivel de las hojas, y puede originar la reestructuración del árbol incluso hasta la raíz provocando esto último que la altura aumente en uno. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz).
2. Se evalúa si es una página hoja.
   2.1. Si la respuesta es afirmativa, se evalúa si la cantidad de elementos almacenados en ella es menor a 2n, siendo n el grado del árbol.
      2.1.1. Si la respuesta es afirmativa, entonces se procede a insertar el nuevo valor en el lugar correspondiente.

2.1.2. Si la respuesta es negativa, entonces se divide la página en dos y los (2n + 1) datos se distribuyen entre las páginas resultantes (n en cada página) y el valor del medio sube a la página padre. Si la página padre no tuviera espacio para este valor, entonces se procede de la misma manera, se divide en dos y el elemento del medio sube a la siguiente página, pudiendo repetirse este proceso hasta llegar a la raíz, en cuyo caso se aumenta la altura del árbol.

2.2. Si no es una hoja, se compara el elemento a insertar con cada uno de los valores almacenados para encontrar la página descendiente donde proseguir la búsqueda. Se sigue con el paso 1.

Considere el árbol-B, de grado 2, de la figura 7.18 en el cual se quiere insertar el valor 60. La línea punteada señala la página que se recupera y donde se agrega el 60. En la tabla 7.5 se presenta la secuencia de pasos necesarios para realizar esta operación.

TABLA 7.5 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es una página hoja. No lo es.
3	Se compara el dato a insertar (60) con el valor 105. Es menor, entonces se toma la dirección de la página que está a la izquierda del 105.
4	Se recupera la página con los valores: 33 – 89.
5	Se evalúa si es una página hoja. Sí lo es.
6	Se evalúa si el total de elementos almacenados (2) es menor a 2n. Sí lo es.
7	Se inserta el valor 60 entre el 33 y 89 de tal manera que no altere el orden.

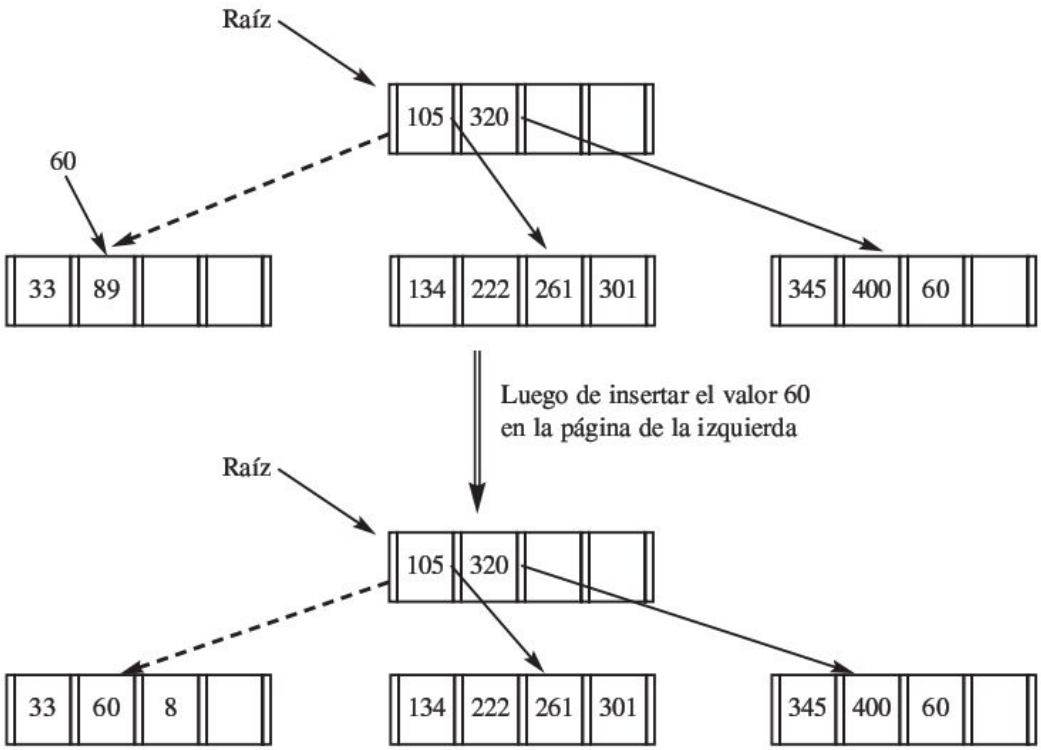


FIGURA 7.18 *Inserción del valor 60*

En el ejemplo anterior no hubo cambios en la estructura del árbol. Considere ahora el árbol-B, de grado 2, de la figura 7.19 en el cual se quiere insertar el valor 120. En la tabla 7.6 se presenta la secuencia de operaciones realizadas al aplicar el algoritmo visto.

TABLA 7.6 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es página hoja. No lo es.
3	Se compara el dato a insertar (120) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).

continúa

TABLA 7.6 Continuación

Operación	*Descripción*
4	Se compara el dato a insertar (120) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 134 – 222 – 261 – 301.
6	Se evalúa si es página hoja. Sí lo es.
7	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
8	Se divide la página en dos y el valor del medio (222) sube a la página padre, en la cual hay espacio. Los demás valores se distribuyen entre las dos nuevas páginas.

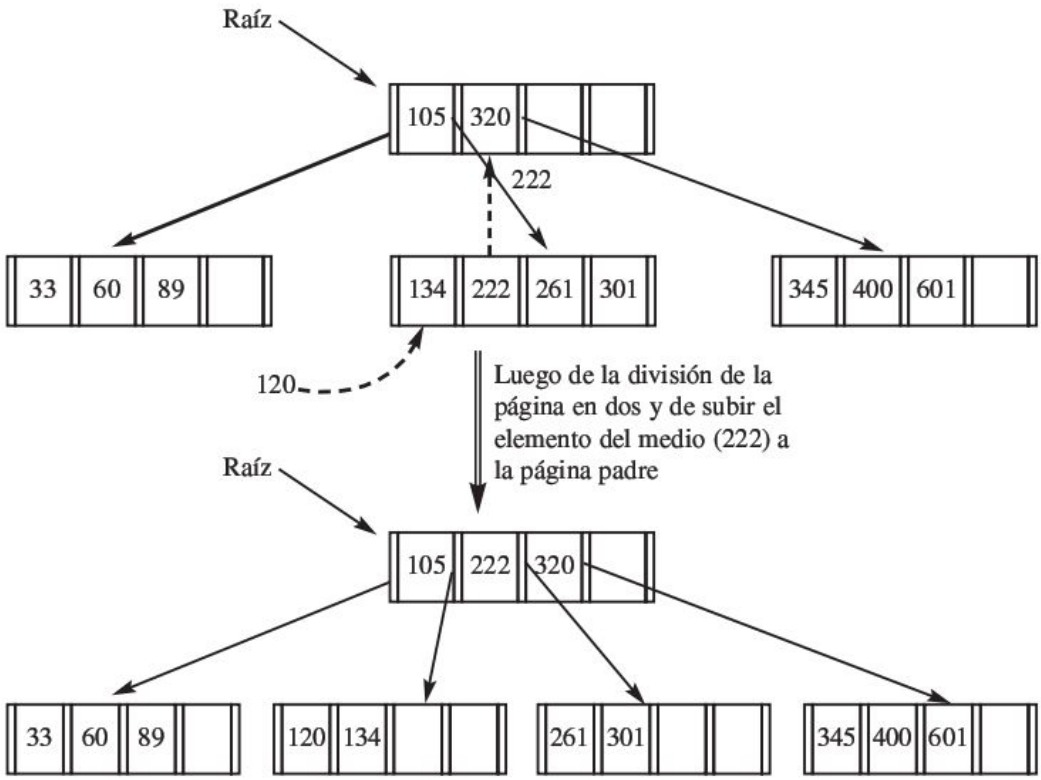


FIGURA 7.19 *Inserción del valor 120*

Como se puede apreciar en la figura 7.19, luego de insertar el valor 120 se modifica la estructura del árbol-B. La página en la que debía insertarse el nuevo dato estaba completa, por lo que se dividió en dos y subió el dato central a la página padre. Finalmente la página padre quedó con tres elementos y por lo tanto con cuatro descendientes.

A continuación se presenta un ejemplo en el cual la estructura del árbol se modifica en cuanto al número de páginas y a la altura. En el árbol-B, de grado 2, de la figura 7.20 se quiere insertar el valor 850. La tabla 7.7 muestra las operaciones que se realizaron al aplicar el algoritmo de inserción.

TABLA 7.7 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320 – 505 – 720
2	Se evalúa si es página hoja. No lo es.
3	Se compara el dato a insertar (850) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a insertar (850) con el valor 320. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (505).
5	Se compara el dato a insertar (850) con el valor 505. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (720).
6	Se compara el dato a insertar (850) con el valor 720. Es mayor y ya no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 720.
7	Se recupera la página con los valores: 765 – 800 – 801 – 976.
8	Se evalúa si es página hoja. Sí lo es.
9	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
10	Se divide la página en dos y el valor del medio (801) sube a la página padre. Los demás valores se distribuyen entre las dos nuevas páginas.
11	En la página padre no hay espacio, el número de elementos almacenados es igual a 2n. Por lo tanto, se debe dividir en dos y el dato del medio (505) debe guardarse en una página que será la nueva raíz del árbol. La altura del árbol crece en una unidad.

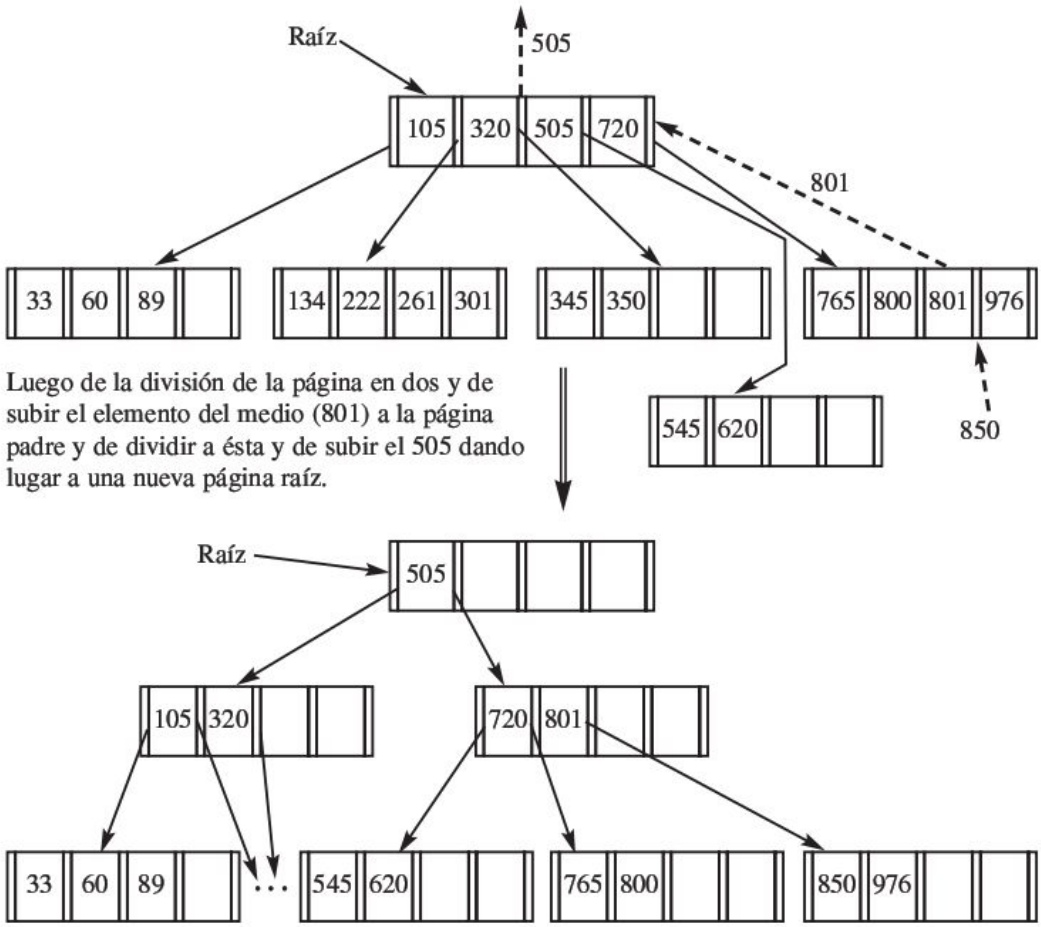


FIGURA 7.20 *Inserción del valor 850*

Eliminación en árboles-B

La operación de **eliminación** consiste en quitar un elemento del árbol-B cuidando que mantenga las propiedades vistas. Es decir, el número de datos en cada página debe ser mayor o igual a n y menor o igual a 2n. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.

2.1. Si la respuesta es afirmativa, entonces la operación de eliminación termina con fracaso.

2.2. Si la respuesta es negativa, entonces ir al paso 3.

3. Se compara el dato a eliminar con cada elemento almacenado en la página.

3.1. Si es igual, entonces se elimina y se procede de la siguiente manera:

3.1.1. Si el dato estaba en una página hoja y el número de elementos de ésta sigue siendo un valor comprendido entre n y 2n, entonces la operación de eliminación termina.

3.1.2. Si el dato estaba en una página y el número de elementos de ésta queda menor que n, entonces se debe bajar el dato más cercano de la página padre y sustituirlo por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición y se fusionan.

3.1.3. Si el dato estaba en la página raíz o en una página intermedia, entonces se debe sustituir por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición. Si es así, termina la eliminación con éxito. En caso contrario, se debe bajar el dato más cercano de la página padre y fusionar las páginas que son hijas de éste.

3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.

3.3. Si es mayor, se avanza al siguiente dato de la misma página.

3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.

3.3.2. Si no es el último, se regresa al paso 3.

El proceso de fusión de páginas puede llegar hasta la raíz, en cuyo caso la altura del árbol disminuye en uno.

Analice el siguiente ejemplo. Se tiene un árbol-B, de grado 2 (ver figura 7.21) en el cual se quiere eliminar el valor 222. Aplicando el algoritmo dado, se realizan las operaciones que aparecen en la tabla 7.8.

TABLA 7.8 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (222) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a eliminar (222) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 134 – 222 – 261 – 301.
6	Se compara el dato a eliminar (222) con el valor 134. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (222).
7	Se compara el dato a eliminar (222) con el valor 222. Es igual, entonces se elimina.
8	Se evalúa si el dato eliminado estaba en una página hoja. Sí lo estaba.
9	Se evalúa si el número de elementos (3) es $\geq$ n y $\leq$ 2n. Sí lo es. La operación termina con éxito.

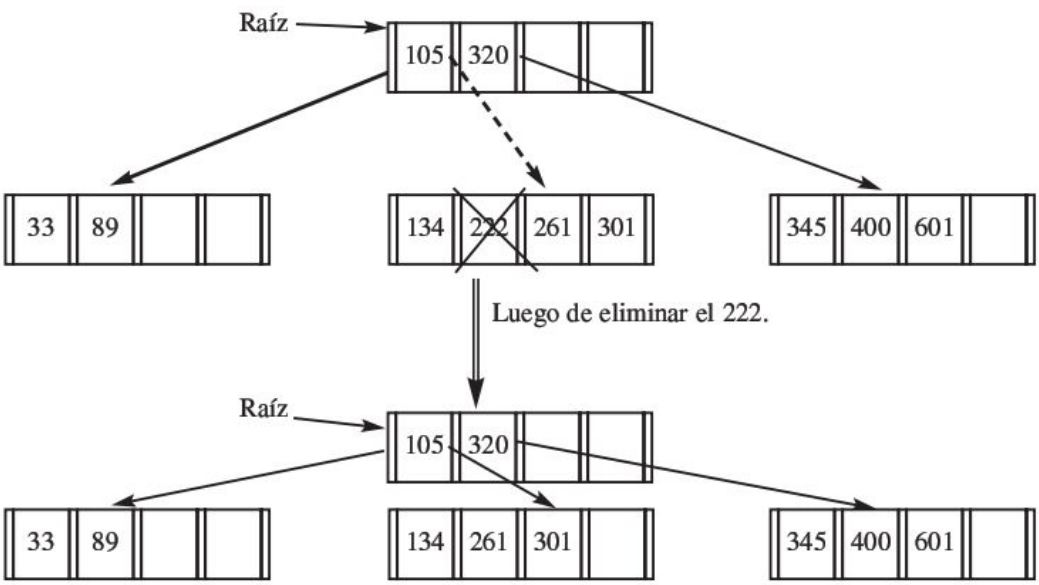


FIGURA 7.21 *Eliminación del valor 222*

En el ejemplo anterior se presentó el caso más simple de eliminación. El dato buscado estaba en una página hoja que a su vez tenía más de n elementos. Observe ahora el siguiente caso. En la figura 7.22 se tiene un árbol-B, de grado 2, del cual se quiere eliminar el valor 89. Aplicando el algoritmo dado, se llevan a cabo las operaciones mostradas en la tabla 7.9.

TABLA 7.9 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (89) con el valor 105. Es menor, entonces se toma la dirección de la página que está a la izquierda del 105.
4	Se recupera la página con los valores: 33 – 89.
5	Se compara el dato a eliminar (89) con el valor 33. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (89).
6	Se compara el dato a eliminar (89) con el valor 89. Es igual, entonces se elimina.
7	Se evalúa si el dato eliminado estaba en una página hoja. Sí lo estaba.
8	Se evalúa si el número de elementos (1) es $\geq$ n y $\leq$ 2n. No lo es, entonces se debe bajar el dato más cercano de la página padre (105) y sustituir a éste por el que se encuentre más a la izquierda del subárbol derecho (134).
9	Se evalúa si el número de elementos de la página del subárbol derecho (de la cual se quitó el 134) cumple con la condición. En este caso (2) es $\geq$ n y $\leq$ 2n. La operación termina con éxito.

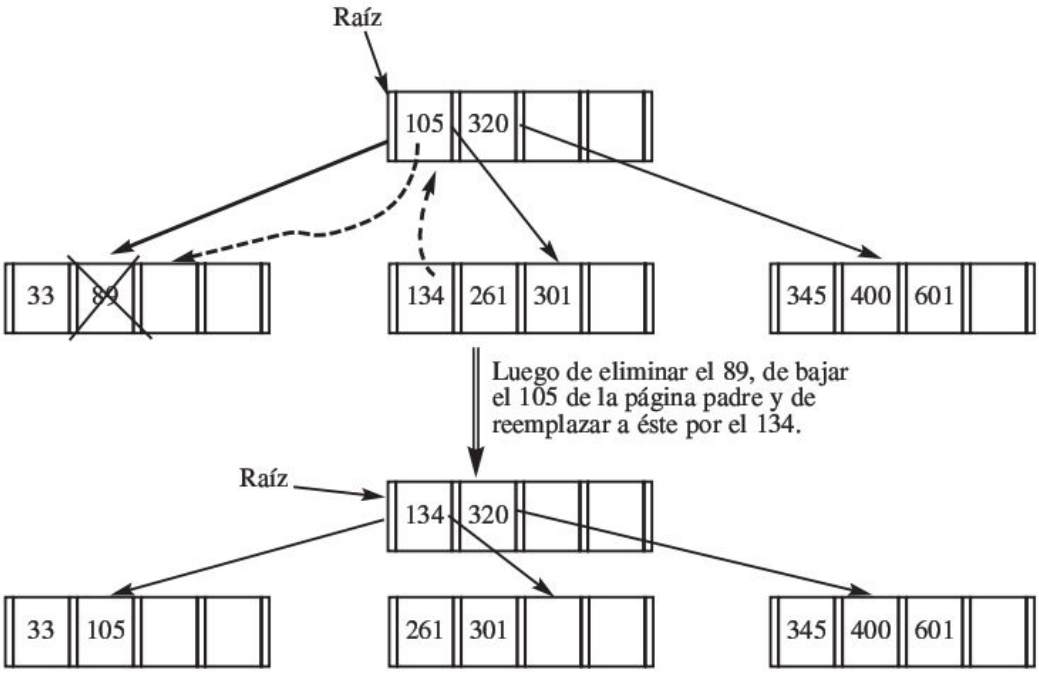


FIGURA 7.22 *Eliminación del valor 89*

Por último, analice el siguiente ejemplo. En el árbol-B, de grado 2, de la figura 7.23 se quiere eliminar el valor 48. En este caso la altura del árbol, luego de efectuar todas las reestructuraciones que corresponden, disminuye en uno La tabla 7.10 presenta las operaciones que se realizan al aplicar el algoritmo.

TABLA 7.10 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 48.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (48) con el valor 48. Es igual, entonces se elimina.
4	Se evalúa si el dato eliminado estaba en una página hoja. No lo estaba.
5	Se debe sustituir el elemento eliminado por el que se encuentre más a la derecha del subárbol izquierdo (44).

continúa

TABLA 7.10 Continuación

Operación	*Descripción*
6	Se evalúa si el número de elementos (1) de esa página es ≥ n y ≤ 2n. No lo es, entonces se debe bajar el dato más cercano (41) de la página padre y fusionar las páginas que son hijas de éste (22 – 30 – 41 – 42).
7	Se evalúa si el número de elementos de la página de la que se bajó el valor 41 es ≥ n y ≤ 2n. Es (1), por lo tanto se debe bajar el dato más cercano (44) de la página padre y fusionar las páginas que son hijas de éste (20 – 44 – 59 – 72). La fusión afectó la raíz, disminuyendo en 1 la altura del árbol. La operación termina con éxito

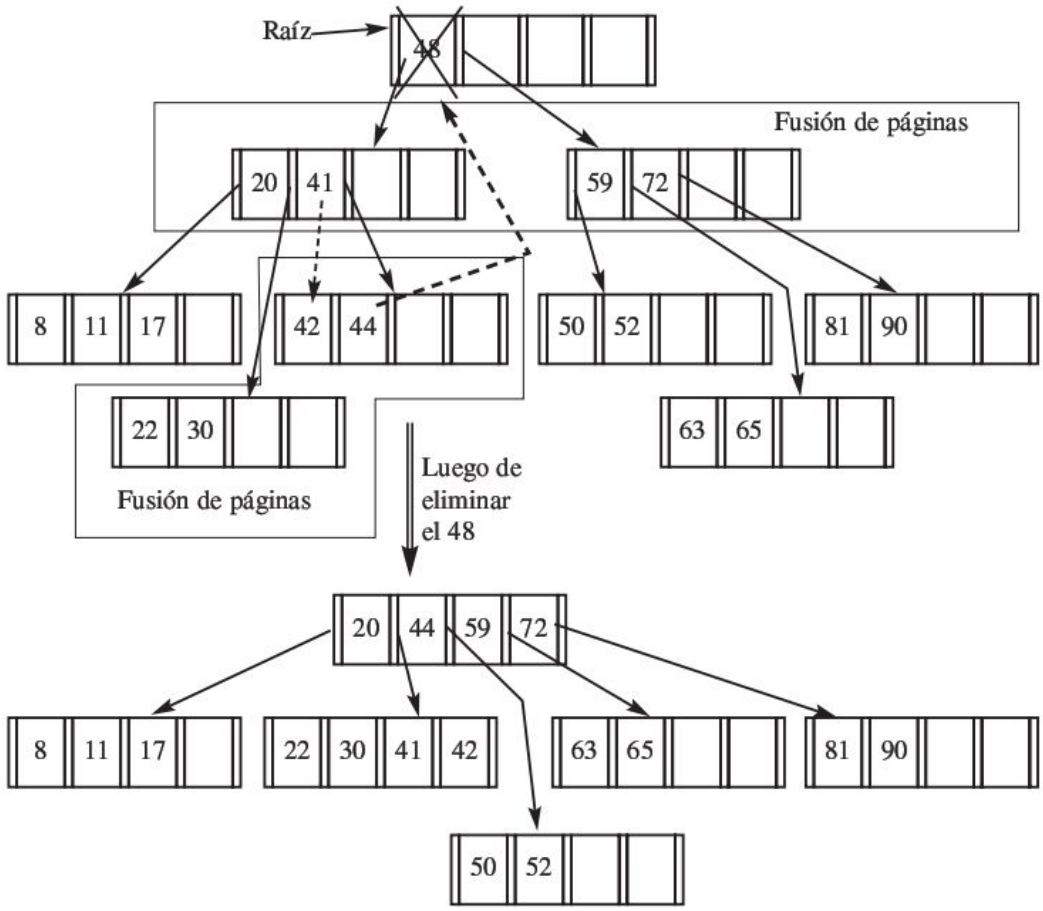


FIGURA 7.23 *Eliminación del valor 48*

7.5 Árboles-B+

Los ***árboles-B+*** son una variante de los árboles-B, diferenciándose de estos últimos por el hecho de que toda la información se encuentra almacenada en las hojas. En la raíz y en los nodos intermedios se guardan solamente las claves o índices que permiten llegar a un cierto dato.

Las principales características de un árbol-B+ de grado n son:

- La página raíz almacena como mínimo 1 dato y como máximo 2n datos.
- La página raíz tiene como mínimo 2 descendientes.
- Las páginas intermedias y hojas almacenan entre n y 2n datos.
- Las páginas intermedias tienen entre (n+1) y (2n+1) páginas descendientes.
- Todas las páginas hojas tienen la misma altura.
- La información se encuentra ordenada.
- Toda la información se encuentra en las páginas hojas, por lo que la clave guardada en la raíz o páginas intermedias se duplica.
- La información guardada en la raíz o en páginas intermedias cumple la función de índices que facilitan el acceso a un cierto dato.

La figura 7.24 presenta un ejemplo de un árbol-B+, de grado 2. En la raíz se almacenan valores que funcionan como índices para llegar a los datos que están en las hojas (es de suponer que el árbol almacena datos más complejos que simples números enteros).

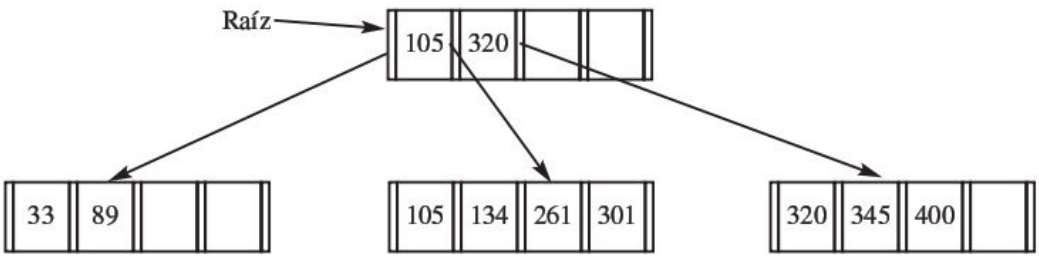


FIGURA 7.24 *Ejemplo de un árbol-B+*

Las operaciones que pueden realizarse sobre la información almacenada en árboles-B+ son: búsqueda, inserción y eliminación. Estas operaciones están implementadas en herramientas diseñadas especialmente para el manejo de archivos, por lo que generalmente no se requiere su programación en las aplicaciones.

Como en el caso de los árboles-B, se explican para que pueda entenderlas y cuando haga uso de algún manejador de archivos sepa qué está pasando internamente con los datos.

Búsqueda en árboles-B+

La operación de **búsqueda** es similar a la estudiada en los árboles-B. La diferencia es que en estos árboles la búsqueda termina siempre en las páginas hojas (donde está la información completa). Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.
   2.1. Si la respuesta es afirmativa, entonces la búsqueda termina con fracaso.
   2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato buscado con cada elemento almacenado en la página.
   3.1. Si es igual, entonces se evalúa si es una página hoja.
      3.1.1. Si la respuesta es afirmativa, entonces la búsqueda termina con éxito.
      3.1.2. Si la respuesta es negativa, entonces se debe recuperar la página descendiente por el lado derecho y se regresa al paso 1.
   3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   3.3. Si es mayor, se avanza al siguiente dato de la misma página.
      3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.3.2. Si no es el último, se regresa al paso 3.

La tabla 7.11 presenta los pasos requeridos para buscar el valor 320 en el árbol-B+ de la figura 7.24.

TABLA 7.11 Operación de búsqueda en un *árbol-B+*

Operación	*Descripción*
1	Se recupera la página con los datos: 105 – 320.
2	Se evalúa si está vacía. En este caso la respuesta es negativa.
3	Se compara el dato buscado (320) con el primer elemento de la página (105). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato buscado (320) con el valor 320. Es igual.
5	Se evalúa si la página donde fue encontrado el 320 es una página hoja. No lo es. Se debe revisar la página descendiente por el lado derecho.
6	Se recupera la página con los datos: 320 – 345 – 400.
7	Se compara el dato buscado (320) con el primer elemento de la página (320). Es igual.
8	Se evalúa si la página donde fue encontrado el 320 es una página hoja. Sí lo es. La búsqueda termina con éxito.

Inserción en árboles-B+

La operación de **inserción** es similar a la estudiada en los árboles-B. La diferencia consiste en que cuando se produce la división de una página en dos (por dejar de cumplir la condición de que el número de elementos debe ser $\geq n$ y $\leq 2n$) se debe subir una copia de la clave (o índice) del elemento del medio. Sólo se duplica información cuando la clave que sube es de una página hoja. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz).
2. Se evalúa si es una página hoja.
   2.1. Si la respuesta es afirmativa, se evalúa si la cantidad de elementos almacenados en ella es menor a 2n.
      2.1.1. Si la respuesta es afirmativa, entonces se procede a insertar el nuevo valor en el lugar correspondiente.
      2.1.2. Si la respuesta es negativa, se divide la página en dos y los (2n + 1) datos se distribuyen entre las páginas resultantes y una copia del valor del medio (o de una clave del mismo) sube a la página

padre. Si la página padre no tuviera espacio para este valor, entonces se procede de la misma manera, se divide en dos y el elemento del medio sube a la siguiente página, pudiendo repetirse este proceso hasta llegar a la raíz, en cuyo caso se aumenta la altura del árbol.

2.2. Si no es una hoja, se compara el elemento a insertar con cada uno de los valores almacenados para encontrar la página descendiente donde proseguir la búsqueda. Se regresa al paso 1.

Considere el siguiente ejemplo. En el árbol-B^+, de grado 2, de la figura 7.25 se quiere insertar el valor 287. Aplicando el algoritmo dado, se realizan las operaciones que se muestran en la tabla 7.12.

TABLA 7.12 Operación de inserción en un *árbol-B^+*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es una página hoja. No lo es.
3	Se compara el dato a insertar (287) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a insertar (287) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 105 – 222 – 261 – 301.
6	Se evalúa si es una página hoja. Sí lo es.
7	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
8	Se divide la página en dos y los elementos se distribuyen entre ellas, subiendo una copia del valor del medio (261) a la página padre.
9	Se evalúa (antes de la inserción) si el número de elementos en la página padre (2) es menor a 2n. Sí lo es. La operación termina con éxito.

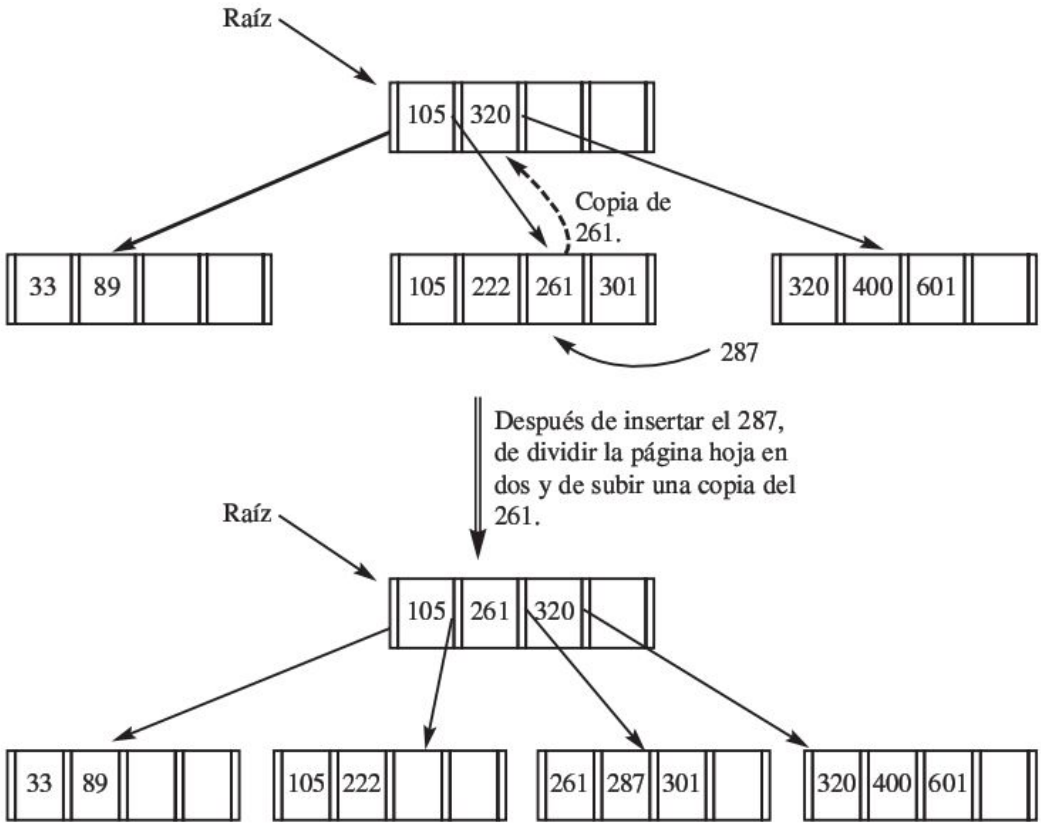


FIGURA 7.25 *Inserción del valor 287*

Eliminación en árboles-B+

La operación de **eliminación** consiste en quitar un elemento del árbol-B+ cuidando que mantenga las propiedades vistas. Es decir, el número de datos en cada página debe ser mayor o igual a n y menor o igual a 2n. Como los datos siempre están en las páginas hojas, cuando se encuentran se quitan (sólo de la hoja, sin importar si además están en un nodo intermedio o raíz) y sólo se reestructura el árbol si la página quedó con menos de n elementos. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.

2. Se evalúa si la página está vacía.
   2.1. Si la respuesta es afirmativa, entonces la operación de eliminación termina con fracaso.
   2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato a eliminar con cada elemento almacenado en la página.
   3.1. Si es igual, se evalúa si está en una página hoja.
      3.1.1. Si la respuesta es negativa entonces se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.1.2. Si la respuesta es afirmativa, entonces se elimina y se evalúa si el número de elementos que quedó sigue siendo mayor o igual a n.
         3.1.2.1 Si la respuesta es afirmativa, entonces la operación termina con éxito. Las páginas intermedias no se modifican aunque almacenen una copia del elemento eliminado.
         3.1.2.2 Si la respuesta es negativa, entonces se debe bajar el dato más cercano de la página padre y sustituir a éste por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición. En caso contrario, se debe bajar el dato más cercano de la página padre y fusionar las páginas que son hijas de éste.
   3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   3.3. Si es mayor, se avanza al siguiente dato de la misma página.
      3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.3.2. Si no es el último, se regresa al paso 3.

El proceso de fusión puede llegar hasta la raíz, en cuyo caso la altura del árbol disminuye en uno. Cuando se llevan a cabo las fusiones, se deben quitar todas aquellas claves copias de elementos eliminados en las páginas hojas.

La tabla 7.13 presenta la secuencia de operaciones requeridas para llevar a cabo la eliminación del valor 261 del árbol-B^+, de grado 2, de la figura 7.26.

TABLA 7.13 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con el valor: 105.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (261) con el valor 105. Es mayor y no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 105.
4	Se recupera la página con los valores: 105 – 261.
5	Se compara el dato a eliminar (261) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (261).
6	Se compara el dato a eliminar (261) con el valor 261. Es igual.
7	Se evalúa si está en una página hoja. Sí lo está, entonces se elimina.
8	Se evalúa si el número de elementos que quedó en la página (1) es $\geq$ n y $\leq$ 2n. No lo es.
9	Se baja el dato más cercano de la página padre (105) y éste no se puede sustituir, ya que el número de elementos de su otro hijo es 2. Por lo tanto, se baja y se fusionan sus páginas descendientes. En este caso, la altura del árbol disminuye en uno. El proceso termina con éxito.

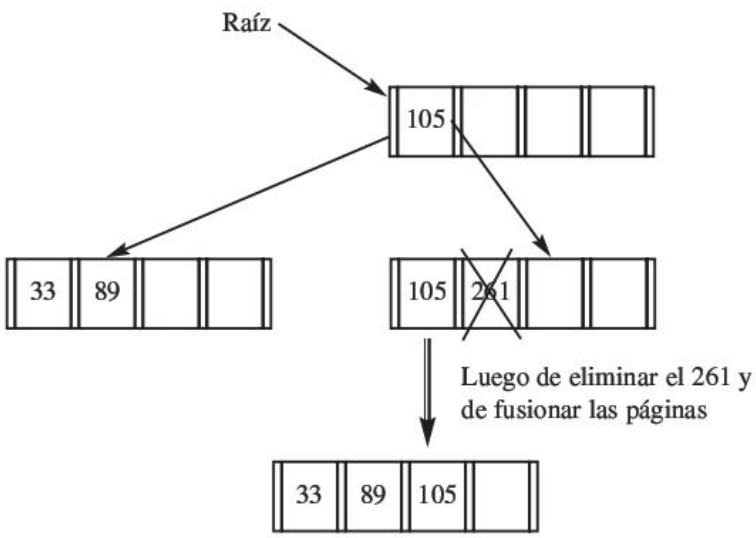
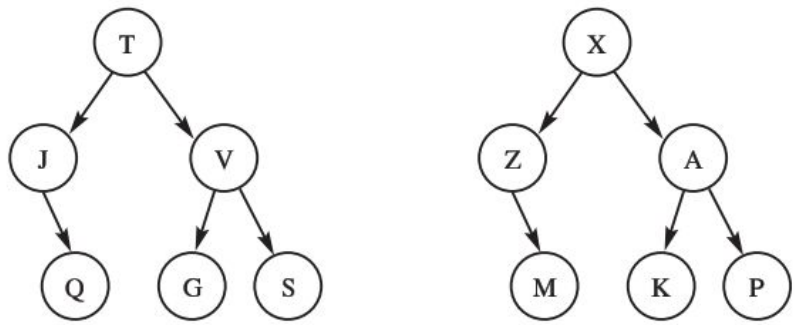


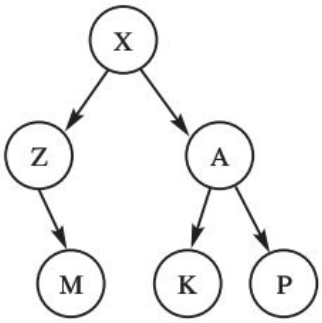
FIGURA 7.26 *Eliminación del valor 261*

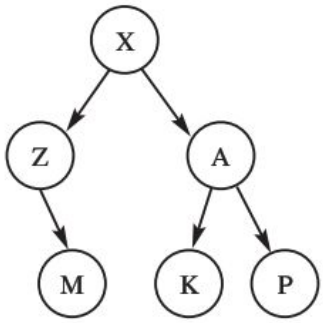
Ejercicios

1. Defina una plantilla para la clase `ArbolMultiple`. Decida qué atributos y métodos incluir. ¿Puede implementar esta clase en ***C++***? Justifique su respuesta.
2. Escriba un método que cuente el número de hojas de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
3. Escriba un método que cuente el número de nodos intermedios de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
4. Escriba un método que calcule la altura de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
5. Escriba un método que imprima todos los ascendientes masculinos de un individuo cuyos datos genealógicos fueron almacenados en un árbol binario.
6. Escriba un método que imprima todos los datos de los ascendientes que estén vivos de un individuo cuyos datos genealógicos fueron almacenados en un árbol binario.
7. Se dice que dos árboles son *similares* cuando sus estructuras son iguales. Escriba una función en ***C++*** que determine si dos árboles (dados como parámetros) son *similares*. En la siguiente figura se presenta un ejemplo de dos árboles que tienen esta característica.

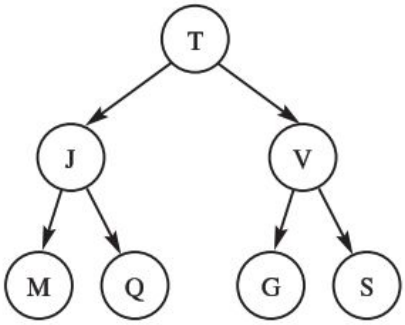


8. Se dice que dos árboles son *equivalentes* cuando sus estructuras son iguales y además tienen el mismo contenido. Escriba una función en ***C++*** que determine si dos árboles (dados como parámetros) son *equivalentes*. En la siguiente figura se presenta un ejemplo de dos árboles que tienen esta característica.





9. Se dice que un árbol binario es *completo* si todos sus nodos, excepto las hojas, tienen dos hijos. Escriba una función en ***C++*** que determine si un árbol (dado como parámetro) es *completo*. En la siguiente figura se presenta un ejemplo de un árbol que tiene esta característica.



10. Retome la plantilla de la clase `ArbolBinario` presentada en este libro, e incluya un método que imprima por niveles toda la información de un objeto tipo árbol. Si el árbol fuera el que aparece en el problema 8, la impresión sería:

X – Z – A – M – K – P

11. Considere que no puede manejar memoria dinámica para representar una estructura tipo árbol binario. Sin embargo, dadas las características de la información, usted decide que la mejor estructura para su almacenamiento y posterior uso es un árbol. Utilice un arreglo unidimensional, guardando en cada casilla la información correspondiente a un nodo, de tal manera que se mantengan las relaciones (padre-de y/o hijo-de) entre ellos. Diseñe e implemente las operaciones de búsqueda, inserción y eliminación que se ajuste a esta nueva representación.

12. Utilice un árbol binario de búsqueda para almacenar datos de tipo clientes bancarios. Para ello defina una clase `ClienteBanco`, con los atributos y los métodos que considere necesarios, atendiendo lo que se pide más abajo. El número de cliente será el atributo según el que se ordenará la información en el árbol. Escriba un programa en ***C++***, que mediante un menú de opciones, permita:

    a) Generar un reporte de todos los clientes de un banco, ordenados por su número de cliente.

    b) Generar un reporte de todos los clientes que tengan una antigüedad mayor a los 5 años. Puede darle generalidad a su solución, dejando el número de años como un dato a ingresar por el usuario.

    c) Generar un reporte de todos los clientes que tengan como mínimo dos cuentas diferentes en el banco.

    d) Dar de alta un nuevo cliente. El usuario proporcionará todos los datos del cliente a registrar.

    e) Dar de baja un cliente registrado. El usuario dará como dato el número del cliente.

    f) Actualizar el saldo de alguna de las cuentas de un cliente. El usuario dará como datos el número del cliente, el número de la cuenta a actualizar y el nuevo saldo.

    g) Actualizar los datos personales (por ejemplo domicilio, teléfono casa, teléfono oficina, etcétera) de un cliente. Su programa debe permitir que en la misma opción se pueda modificar uno o todos los datos personales.

13. Utilice un árbol binario balanceado para almacenar datos relacionados a insectos. Para ello defina una clase `Insecto`, con los atributos y los métodos que considere necesarios, atendiendo lo que se pide más abajo. Cada insecto tendrá una clave, que será el atributo que permita que la información esté ordenada en el árbol. Escriba un programa en ***C++***, que mediante un menú de opciones, pueda:

    a) Registrar un nuevo insecto. El usuario dará todos los datos necesarios.

    b) Dar de baja un insecto registrado. El usuario dará la clave del insecto a eliminar.

    c) Generar un reporte de todos los insectos, ordenados por clave.

    d) Generar un reporte de todos los insectos que viven en el área del Mediterráneo europeo.

e) Generar un reporte de todos los insectos que viven sólo en el desierto de Rub al-Jali.

f) Generar un reporte de todos los insectos que se alimentan de madera en estado de descomposición.

14. Inserte los siguientes datos en un árbol-B, de grado 2. Los números que se dan como datos pueden representar datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 95 – 10 – 34 – 87 – 56 – 99 – 12 – 23 – 50 – 40 – 60 – 54 – 33 – 20 – 91 – 17 – 18 – 94

15. En el árbol-B generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 99 – 60 – 23 – 12 – 95 – 40

16. Inserte los siguientes datos en un árbol-B, de grado 2. Los números que se dan como datos pueden representar datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 105 – 99 – 104 – 80 – 16 – 74 – 112 – 230 – 71 – 33 – 86 – 399 – 33 – 120 – 51 – 67 – 90 – 84 – 45 – 405 – 257 – 110

17. En el árbol-B generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 399 – 80 – 105 – 84 – 86 – 51 – 67 – 33 – 112 – 104

18. Inserte los siguientes datos en un árbol-B^+, de grado 2. Los números que se dan como datos pueden representar a datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 120 – 100 – 240 – 817 – 356 – 199 – 249 – 326 – 500 – 170 – 360 – 257 – 358 – 104 – 921 – 590 – 328 – 140

19. En el árbol-B^+, de grado 2, generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

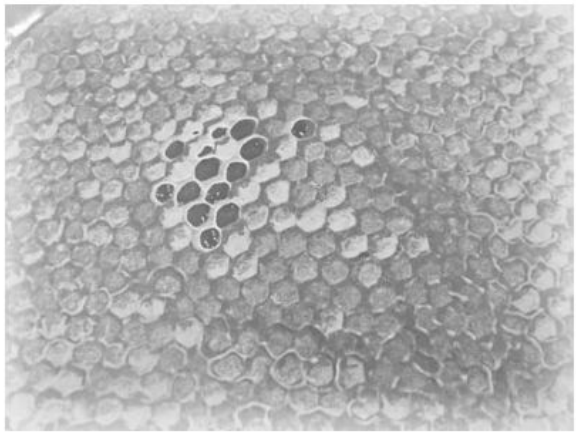
    Eliminar: 328 – 356 – 100 – 817 – 921 – 500 – 358 – 328 – 590 – 104 – 249

20. Inserte los siguientes datos en un árbol-B^+, de grado 2. Los números que se dan como datos pueden representar a datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 350 – 180 – 420 – 700 – 390 – 200 – 150 – 400 – 300 – 100 – 500 – 310 – 660 – 580 – 880 – 670 – 370 – 140 – 230 – 490 – 510

21. En el árbol-B^+, de grado 2, generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 880 – 420 – 100 – 580 – 180 – 230 – 400 – 700 – 660 – 670 – 490 – 140 – 350 – 370



CAPÍTULO 8

Gráficas

8.1 Introducción

Este capítulo presenta la estructura de datos conocida como ***gráfica***. Se estudian sus principales características, cómo se relacionan sus componentes y se analizan las operaciones que pueden aplicarse sobre ellos.

Las gráficas son estructuras de datos no lineales, en las cuales cada elemento puede tener cero o más sucesores y cero o más predecesores. Están formadas por nodos, llamados generalmente ***vértices***, y por arcos, conocidos también con el nombre de ***aristas***. Los vértices representan información y las aristas relaciones entre dicha información. La figura 8.1 presenta un ejemplo de una gráfica. En ella, los vértices almacenan los datos de un grupo de ciudades y las aristas indican que existe una carretera entre las ciudades que están uniendo. El valor asociado a cada arista representa el total de kilómetros entre

las ciudades que están en los extremos. Según esta figura, entre las ciudades de Puebla y Tlaxcala existe una carretera que cubre una distancia de 33 kms. Por su parte, entre las ciudades de Puebla y de Xalapa hay una carretera de 194 kms. También se observa que entre las ciudades de Toluca y Xalapa no existe una carretera directa que las una.

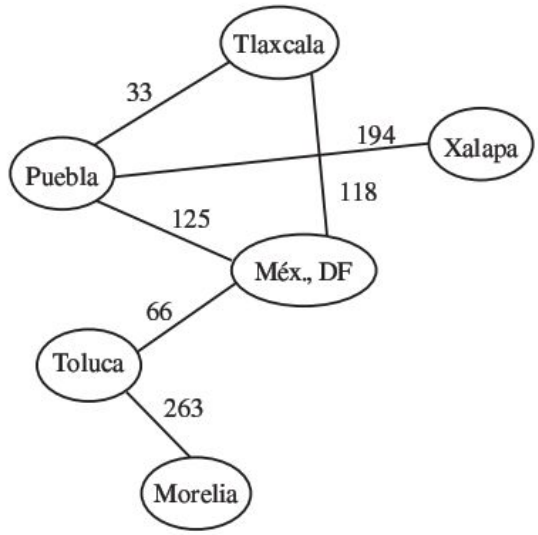


FIGURA 8.1 *Ejemplo de una gráfica*

Formalmente, una gráfica está integrada por los conjuntos V(G) y A(G), donde el primero representa a todos los vértices o nodos y el segundo a las aristas o arcos. Por lo general, estos conjuntos son finitos. Una arista se define por medio de un par único de vértices del conjunto V, que puede o no estar ordenado. Tomando como referencia la gráfica de la figura 8.2, se tiene que:

$V = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \}$

$A = \{ (v_1, v_2), (v_1, v_7), (v_2, v_4), (v_2, v_7), (v_3, v_4), (v_3, v_5), (v_4, v_5), (v_5, v_6), (v_6, v_7) \}$

$G = (V, A)$

La arista entre los nodos v_1 y v_2 se expresa como **a**= (v_1, v_2), indicando que los vértices v_1 y v_2 son ***adyacentes*** y extremos de **a**. Además, se dice que **a** es ***incidente*** en v_1 y v_2.

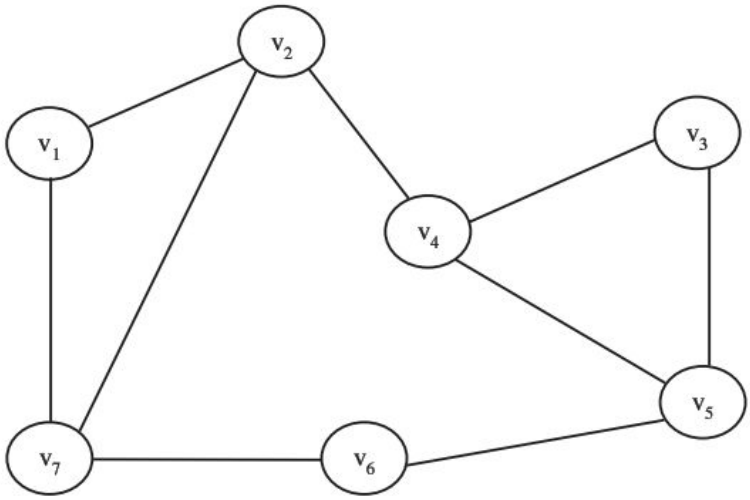


FIGURA 8.2 *Ejemplo de gráfica*

El ***grado*** de un vértice, identificado como grado (v), es el total de aristas que tienen como extremo a v. Cuando el grado de un vértice es 0, éste recibe el nombre de ***vértice aislado***. El vértice v_2 de la figura 8.2 es de grado 3, mientras que el vértice v_6 es de grado 2.

Dependiendo de la ubicación y combinación de una o varias aristas, se definen distintas figuras. Las más comunes son:

- Un ***lazo*** o ***bucle*** es una arista que tiene en ambos extremos al mismo vértice. Se expresa como a= (v, v).
- Un ***camino*** del vértice origen v_1 al vértice destino v_n está formado por todas las aristas que deben recorrerse para llegar del origen al destino. Si se recorren n aristas, se dice que el camino es de longitud n.
- Un ***camino*** es ***cerrado*** si el vértice origen es igual al vértice destino.
- Un ***camino*** es ***simple*** si todos sus vértices, con excepción del origen y destino, son distintos. El primero y último vértices de un camino simple pueden ser iguales.
- Un ***ciclo*** es un camino simple cerrado de longitud mayor o igual a tres.

De acuerdo a sus características, las gráficas reciben distintos nombres. Los más comunes son:

- Una ***gráfica conexa*** es aquella en la cual existe un camino simple entre cualesquiera de sus nodos.

- Una ***gráfica árbol*** o ***árbol libre*** es una gráfica conexa sin ciclos.
- Una ***gráfica*** es ***completa*** si cada uno de sus vértices son adyacentes a todos los vértices de la gráfica.
- Una ***gráfica*** es ***etiquetada*** si sus aristas tienen asociado un valor. Si éste es un número no negativo, se le conoce con el nombre de peso, distancia o longitud.
- Una gráfica es una ***multigráfica*** si al menos dos de sus vértices están unidos entre sí por dos aristas, llamadas aristas paralelas o múltiples.
- Una ***subgráfica*** está formada por un subconjunto de vértices y de aristas de una gráfica dada. Por lo tanto la subgráfica G′de G, se define como $G' = (V', A')$, donde $V' \subseteq V$ y $A' \subseteq A$.

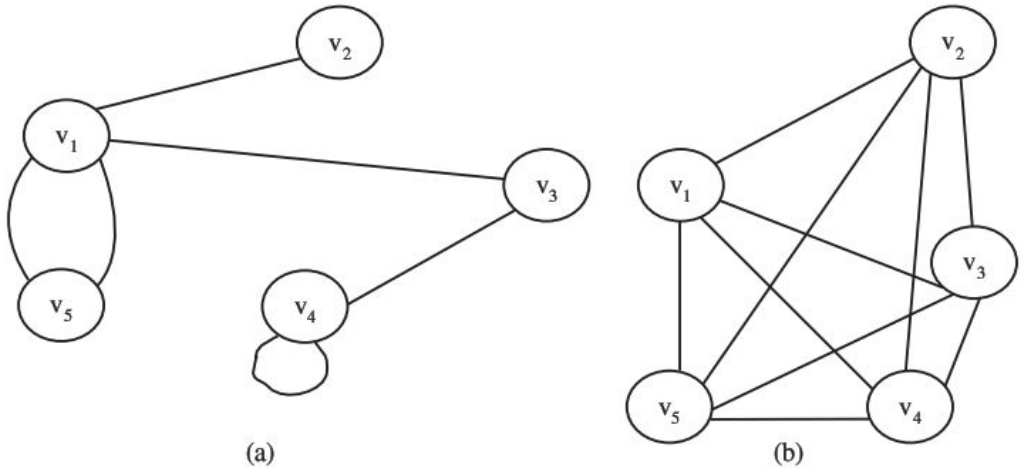


FIGURA 8.3 *Ejemplos de gráficas*

Al observar las gráficas presentadas en la figura 8.3 se pueden hacer las siguientes afirmaciones:

- Hay un bucle o lazo en el vértice v_4 de la gráfica del inciso (a).
- La gráfica (a) es una multigráfica, ya que existen aristas paralelas o múltiples entre los vértices v_1 y v_5.
- En la gráfica (a) hay un camino de v_5 a v_4, definido por la secuencia de aristas: v_5, v_1, v_3, v_4.
- En la gráfica (b) hay un camino cerrado definido por la secuencia de aristas: v_1, v_3, v_4, v_1.

- En la gráfica (a) hay un camino simple definido por la secuencia de aristas: v_1, v_3, v_4.
- Ambas gráficas son conexas, ya que todos sus vértices están unidos al menos a otro vértice.
- La gráfica (b) es completa, ya que cada uno de sus vértices son adyacentes a todos los demás.

8.2 Gráficas dirigidas

Las ***gráficas dirigidas*** o ***digráficas*** son aquellas cuyas aristas siguen cierta dirección. En este caso, cada arista a = (v_1, v_2) recibe el nombre de arco y se representa con la notación v_1 —> v_2. Se dice que v_1 es el vértice origen o punto inicial y que v_2 es el vértice destino o punto terminal del arco a. La figura 8.4 presenta un ejemplo de una gráfica dirigida.

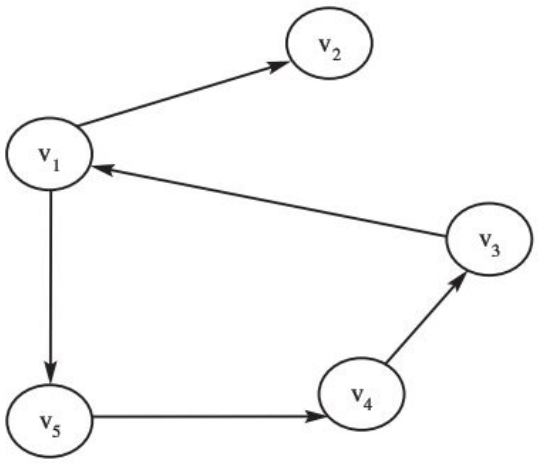


FIGURA 8.4 *Ejemplo de gráfica dirigida*

8.2.1 Representación de una digráfica

Las digráficas son estructuras de datos abstractas (como las pilas y colas), por lo tanto los lenguajes de programación no cuentan con elementos diseñados exclusivamente para su representación y manejo. En consecuencia, se requiere utilizar alguna de las

estructuras de datos ya estudiadas para su representación y almacenamiento en memoria. Las más usadas son las **listas de adyacencia** y las **matrices de adyacencia**.

La ***lista de adyacencia*** está formada por una lista de listas. Es decir, cada nodo de la lista representa a un vértice y almacena, además de la información propia del vértice, una lista de vértices adyacentes. La figura 8.5 muestra un ejemplo de una digráfica y su correspondiente lista de adyacencia.

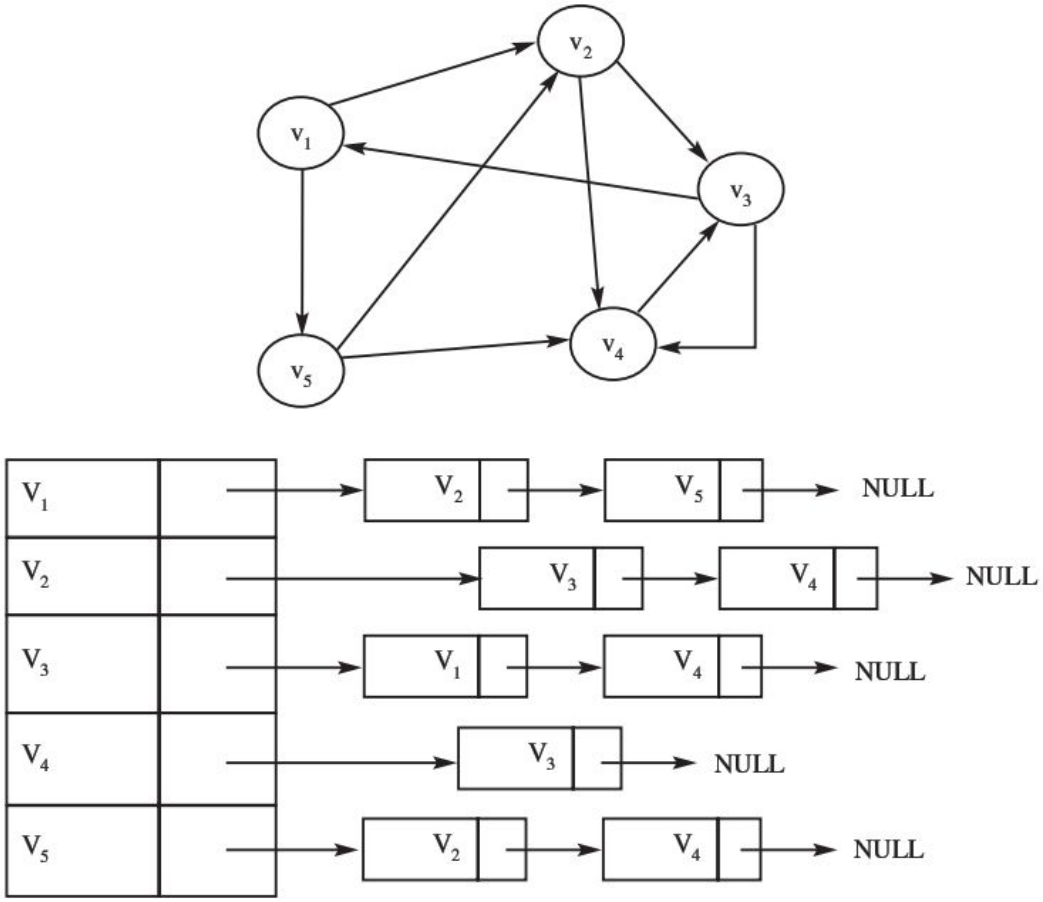


FIGURA 8.5 *Gráfica dirigida y su representación por medio de una lista de adyacencia*

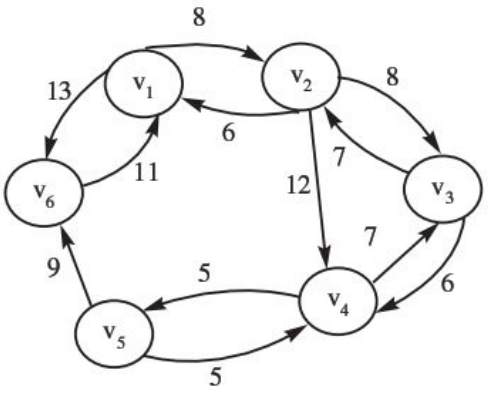
La ***matriz de adyacencia*** es una matriz de números enteros, donde los renglones y columnas representan a los vértices de la digráfica. En la posición i, j se asigna un

1 si existe un arco del vértice *i* al vértice *j*. En caso contrario y en las posiciones correspondientes a la diagonal principal se asigna un 0. Si la digráfica tiene N vértices, la matriz de adyacencia tendrá NxN elementos. La figura 8.6 muestra la matriz de adyacencia correspondiente a la digráfica de la figura 8.5. En este libro se emplea este tipo de representación para las digráficas.

Si la digráfica a almacenar está etiquetada, entonces se necesita una ***matriz de adyacencia etiquetada***. La diferencia es que en lugar del 1 se asigna la etiqueta o costo del arco correspondiente. Esta matriz también recibe el nombre de ***matriz de costos*** o ***matriz de distancias***. La figura 8.7 muestra un ejemplo de una gráfica dirigida etiquetada con su correspondiente matriz de adyacencia etiquetada.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	0	1
v_2	0	0	1	1	0
v_3	1	0	0	1	0
v_4	0	0	1	0	0
v_5	0	1	0	1	0

FIGURA 8.6 *Representación de una gráfica dirigida por medio de una matriz de adyacencia*



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	8	0	0	0	13
v_2	6	0	8	12	0	0
v_3	0	7	0	6	0	0
v_4	0	0	7	0	5	0
v_5	0	0	0	5	0	9
v_6	11	0	0	0	0	0

FIGURA 8.7 *Gráfica dirigida etiquetada y su representación por medio de una matriz de adyacencia etiquetada*

8.2.2 La clase digráfica

Considerando que se usará una matriz de adyacencia para almacenar la información de una digráfica, la clase `DiGrafica` tendrá como atributos un arreglo de dos dimensiones para almacenar dicha matriz, un entero que representa el número de vértices y un arreglo con sus nombres (en este caso enteros). Además de los atributos, la clase tendrá algunos métodos que permiten la manipulación de los datos guardados.

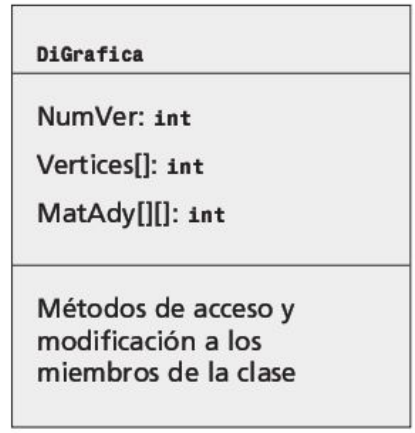


FIGURA 8.8 *Clase* `DiGrafica`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a la definición de la clase `DiGrafica` del esquema de la figura 8.8. Se incluyen sólo los prototipos de los métodos, ya que éstos se analizan más adelante. Además de los atributos ya mencionados, la clase contiene otros elementos que son necesarios en los métodos que utiliza. Se decidió tratarlos como atributos para simplificar su codificación y para que la clase estuviera autocontenida.

```
/* Constante usada para establecer el número máximo de vértices de la
➥digráfica. */
#define MAX 20

/* Definición de la plantilla de la clase DiGrafica. Se incluyen como
➥atributos: la matriz de costos/distancias (MatAdy), el número de vértices
➥(NumVer), el nombre de cada uno de los vértices (Vertices), un arreglo
```

```
➥para las distancias mínimas entre los vértices (DistMin), un arreglo para
➥la cerradura transitiva de la matriz de adyacencia (CerTran), y un
➥arreglo para vértices intermedios (VerInt). Los tres últimos se usan
➥como auxiliares en métodos que se estudian más adelante. */
template <class T>
class DiGrafica
{
   private:
       /* Declaración del arreglo donde se almacenan las distancias en-
       tre los vértices. */
       T MatAdy[MAX][MAX];

       int NumVer, Vertices[MAX], DistMin[MAX], CerTran[MAX][MAX],
                VerInt[MAX][MAX];
   public:
       /* Método constructor y métodos auxiliares para leer la informa-
       ción relacionada a la gráfica e imprimir los resultados obtenidos
       al aplicar los demás métodos. */
       DiGrafica();
       void Lee();
       void Imprime(int);

       /* Métodos que recorren una gráfica dirigida, determinando cami-
       nos de distancias mínimas. */
       void Warshall();
       void Floyd();
       void FloydVerInt();
       void Dijkstra();

};

/* Declaración del método constructor. Inicializa la matriz de adyacen-
cias MatAdy con un valor arbitrario muy grande (999), indicando que no
existe camino entre los nodos correspondientes. Además, asigna ceros a
los arreglos que se usarán en otros métodos. */
template <class T>
DiGrafica<T>::DiGrafica()
{
   int Ind1, Ind2;
   for (Ind1= 0; Ind1 < MAX; Ind1++)
   {
        DistMin[Ind1]= 0;
        for (Ind2= 0; Ind2 < MAX; Ind2++)
        {
             if (Ind1 != Ind2)
                 MatAdy[Ind1][Ind2]= 999;
```

```
                else
                    MatAdy[Ind1][Ind2]= 0;
                    CerTran[Ind1][Ind2]= 0;
                    VerInt[Ind1][Ind2]= 0;
            }
        }
        NumVer= 0;
    }
```

Como ya se mencionó, en la plantilla se incluyeron algunos arreglos que se usan en los métodos con el fin de que la clase contenga todos los elementos necesarios. Sin embargo, dichos arreglos pueden declararse como locales a los métodos y pasarse como resultados o parámetros a los usuarios de la clase.

8.2.3 Recorrido de gráficas dirigidas

En esta sección se presentan los métodos más usados para determinar la existencia o no existencia de caminos entre los vértices de la gráfica, así como los métodos que obtienen los caminos de menor longitud entre ellos.

Método Warshall

Este método **determina si existe o no un camino de longitud mayor o igual a uno entre los vértices de una gráfica dirigida**. Es decir, el método sólo encuentra si hay un camino directo o indirecto (por medio de otros vértices intermedios) entre los nodos, sin importar el costo. Para aplicar este método se requiere generar la cerradura transitiva de la matriz de adyacencia de la digráfica. La cerradura transitiva es una matriz (`CerTran`) de `NumVer` por `NumVer` elementos, donde `CerTran[i][j]= 1` si existe un camino de i a j, y 0 en caso contrario.

A continuación se presenta el método `Warsall` de la clase `DiGrafica`.

```
/* Método que determina si existe un camino entre cada uno de los
➥vértices de la gráfica dirigida. CerTran es una matriz que representa
➥la cerradura transitiva de la matriz de adyacencia. */
template <class T>
void DiGrafica<T>::Warshall()
```

```
{
   int Ind1, Ind2, Ind3;

/* En la posición i,j de la matriz de adyacencia se asignó el valor 999
➥si no existe un camino directo del vértice i al vértice j. La cerradura
➥transitiva se forma inicialmente a partir de la matriz de adyacencia. */

   for (Ind1= 0; Ind1 < NumVer; Ind1++)
        for (Ind2= 0; Ind2 < NumVer; Ind2++)
             if (MatAdy[Ind1][Ind2] !=  999)
                  CerTran[Ind1][Ind2]= 1;

/* Se recorren todos los vértices para determinar si existe un camino
➥entre él y los demás, usando otros vértices como intermedios. */
   for (Ind3= 0; Ind3 < NumVer; Ind3++)
         for (Ind1= 0; Ind1 < NumVer; Ind1++)
              for (Ind2= 0; Ind2 < NumVer; Ind2++)
                  CerTran[Ind1][Ind2] |=
                  CerTran[Ind1][Ind3] &&
                  CerTran[Ind3][Ind2];
}
```

Si se aplica este método a la matriz de distancias de la figura 8.9, el resultado que se obtiene es el que se presenta en la figura 8.10.

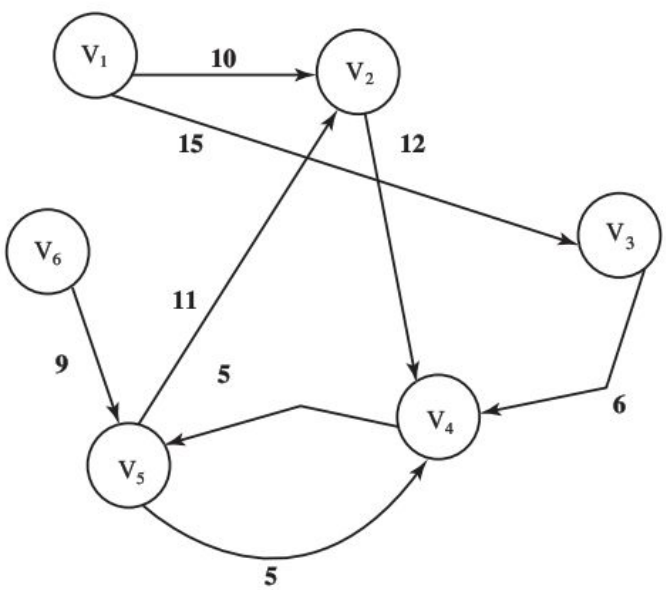
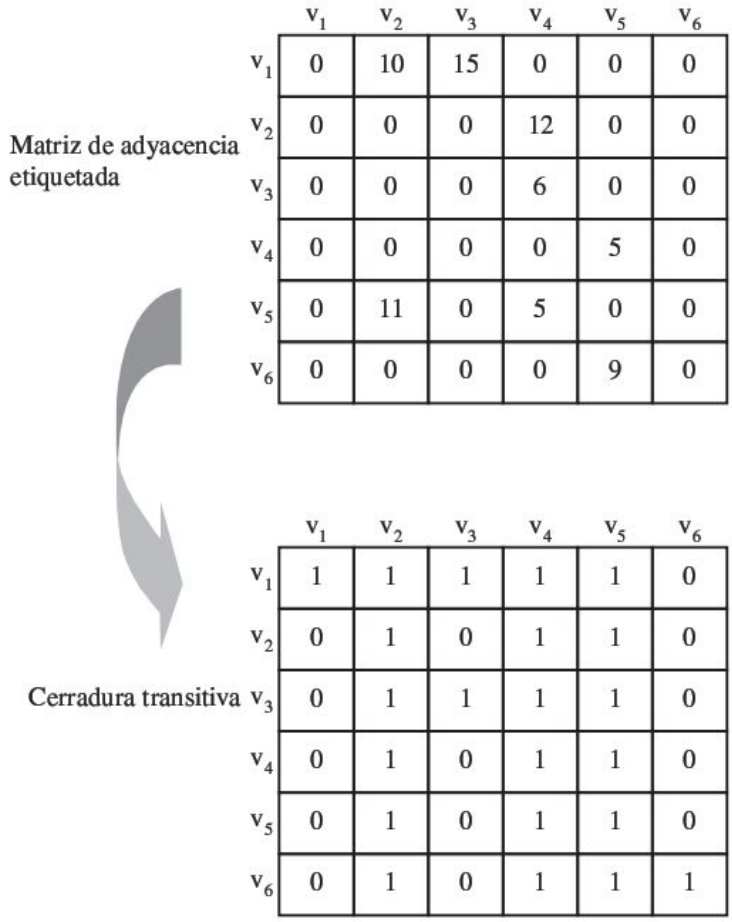


FIGURA 8.9 *Gráfica dirigida*



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	10	15	0	0	0
v_2	0	0	0	12	0	0
v_3	0	0	0	6	0	0
v_4	0	0	0	0	5	0
v_5	0	11	0	5	0	0
v_6	0	0	0	0	9	0

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	1	1	1	1	1	0
v_2	0	1	0	1	1	0
v_3	0	1	1	1	1	0
v_4	0	1	0	1	1	0
v_5	0	1	0	1	1	0
v_6	0	1	0	1	1	1

FIGURA 8.10 *Ejemplo de aplicación del método Warshall*

La figura 8.10 presenta la matriz de adyacencia etiquetada correspondiente a la gráfica dirigida de la figura 8.9. En la parte inferior se muestra la cerradura transitiva obtenida luego de aplicar el método `Warshall`. El valor **1** en la posición *i, j* indica que existe un camino (directo o no) entre el vértice *i* y el vértice *j*, mientras que un **0** representa que no hay camino entre ellos. En la diagonal principal quedan **1**'s, aunque en estos casos no tenga utilidad saber que se cuenta con un camino. Se puede observar que en la posición correspondiente a los vértices (V_1, V_4) hay un **1**, a pesar de que no hay un camino directo entre ellos. Como resultado de la aplicación de este método se encontró que es posible ir de V_1 a V_4, usando vértices intermedios. En este caso se puede ir a través de V_2 o de V_3.

A continuación se muestra la forma como va generándose la cerradura transitiva a medida que se ejecuta el método. Cada uno de los incisos corresponde a un cambio en la matriz. La tabla 8.1(a) presenta la cerradura transitiva en su estado inicial (con **1**´s en las posiciones donde hay un camino), incluyendo la diagonal (si quisiéramos excluirla deberíamos agregar una condición antes de la asignación). La tabla 8.1(b) muestra que del vértice V_1 se puede llegar a los vértices V_4 y V_5, a través de los vértices intermedios V_2 y V_4 respectivamente. En (c) se identificó un camino de V_2 a V_5, por medio de V_4. En (d) se presenta un camino de V_3 a V_2 y a V_5, usando a V_4 para llegar a V_5 y a este último para llegar a V_2. En (e) se señala un camino de V_4 a V_2 por medio del vértice V_5. Finalmente, en (f) se indica un camino de V_6 a V_2 y a V_4, a través del vértice V_5.

TABLA 8.1 Obtención de la cerradura transitiva usando *Warshall*

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	0	0	0
V_2	0	1	0	1	0	0
V_3	0	0	1	1	0	0
V_4	0	0	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	0	0	0	1	1

(a)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	1	1	0
V_2	0	1	0	1	0	0
V_3	0	0	1	1	0	0
V_4	0	0	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	0	0	0	1	1

(b)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	1	1	0
V_2	0	1	0	1	1	0
V_3	0	0	1	1	0	0
V_4	0	0	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	0	0	0	1	1

(c)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	1	1	0
V_2	0	1	0	1	1	0
V_3	0	1	1	1	1	0
V_4	0	0	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	0	0	0	1	1

(d)

continúa

TABLA 8.1 Continuación

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	1	1	0
V_2	0	1	0	1	1	0
V_3	0	1	1	1	1	0
V_4	0	1	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	0	0	0	1	1

(e)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	1	1	1	1	1	0
V_2	0	1	0	1	1	0
V_3	0	1	1	1	1	0
V_4	0	1	0	1	1	0
V_5	0	1	0	1	1	0
V_6	0	1	0	1	1	1

(f)

Método Floyd

Este método **encuentra el camino más corto entre todos los vértices de la gráfica dirigida**. Es decir, si hay más de un camino posible (directo o a través de nodos intermedios) entre los vértices V_i y V_j, este método encuentra el de menor costo.

A continuación se presenta el método `Floyd` de la clase `DiGrafica`.

```
/* Método que encuentra el camino de costo mínimo entre todos los
➥vértices de la gráfica dirigida. Va modificando la matriz de adyacencia
➥a medida que encuentra un camino más corto entre dos vértices. */
template <class T>
void DiGrafica<T>::Floyd()
{
   int Ind1, Ind2, Ind3;
   for (Ind3= 0; Ind3 < NumVer; Ind3++)
        for (Ind1= 0; Ind1 < NumVer; Ind1++)
             for (Ind2= 0; Ind2 < NumVer; Ind2++)
                  if ( (MatAdy[Ind1][Ind3] + MatAdy[Ind3][Ind2])
                       < MatAdy[Ind1][Ind2])
                       MatAdy[Ind1][Ind2]=
                       MatAdy[Ind1][Ind3] +
                       MatAdy[Ind3][Ind2];
}
```

La tabla 8.2 presenta el resultado de aplicar el método `Floyd` a la digráfica de la figura 8.9. En (a) se muestra la matriz de adyacencia original (el valor 999 es un valor arbitrario que indica que no existe un camino entre los vértices involucrados) y en (b) la matriz con las distancias mínimas. La diagonal principal quedó con ceros (no hay distancia desde un vértice cualquiera hasta el mismo vértice). En las posiciones correspondientes a vértices que no tienen caminos que los unan quedó el valor 999. Se sombrean las casillas en las que hubo cambio.

TABLA 8.2 Ejemplo de aplicación del método `Floyd`

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	999	999	999
V_2	999	0	999	12	999	999
V_3	999	999	0	6	999	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	999

(a)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	16	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	20	999	14	9	0

(b)

Al observar la digráfica de la figura 8.9, se puede apreciar que para ir del vértice V_1 al V_4, existen dos caminos: $V_1 - V_2 - V_4$ con una distancia de 22 (10 + 12) y $V_1 - V_3 - V_4$ con una distancia de 21 (15 + 6). En la matriz queda la segunda por ser la menor. En el caso del vértice V_4 al V_2, sólo existe una opción a través del vértice V_5 con una distancia de 16 (5 + 11).

Ahora se presenta la manera en que va generándose la matriz con las distancias mínimas, a medida que se ejecuta el método. Cada uno de los incisos corresponde a un cambio en la matriz. La tabla 8.3 (a) presenta la matriz después de haber encontrado un camino de longitud 22 entre los vértices V_1 y V_4 (a través de V_2). En (b) la matriz se modifica al encontrar un camino más corto entre los vértices mencionados. En (c) se señala que se encontró un camino entre los vértices V_1 y V_5, con una distancia de 26. En (d) aparece un camino de longitud 17 entre V_2 y V_5. En (e) se indica que hay un camino de V_3 a V_5, usando el vértice intermedio V_4, con una distancia de 11. En (f) aparece un camino de V_3 a V_2, ahora usando a V_5 (vértice con el cual se estableció un camino en el paso previo). En (g) se señala un camino de V_4 a V_2, por medio de V_5. En (h) se indica un camino de V_6 a V_2, a través de V_5. Final-

mente, en (i) aparece un camino de V_6 a V_4 también usando a V_5 como intermedio. Se sombrean las casillas que almacenan los valores mínimos encontrados.

TABLA 8.3 Obtención de la matriz de distancias mínimas usando `Floyd`

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	22	999	999
V_2	999	0	999	12	999	999
V_3	999	999	0	6	999	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(a)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	999	999
V_2	999	0	999	12	999	999
V_3	999	999	0	6	999	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(b)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	999	999
V_3	999	999	0	6	999	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(c)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	999	0	6	999	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(d)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	999	0	6	11	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(e)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	999	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(f)

continúa

TABLA 8.3 Continuación

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	16	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	999	999	999	9	0

(g)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	16	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	20	999	999	9	0

(h)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	16	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	20	999	14	9	0

(i)

Si además de obtener la matriz de distancias mínimas, se necesitara conocer los vértices intermedios que permitieron establecer esas distancias, se deberá modificar el método de la siguiente manera.

```
/* Método Floyd modificado para que, además de encontrar las distancias
➥mínimas entre todos los vértices de una digráfica, genere una matriz
➥(VerInt) con los vértices intermedios utilizados para minimizar las
➥distancias. Este método usa los atributos Vertices (arreglo que
➥almacena los nombres de todos los vértices de la digráfica) y VerInt
➥(arreglo donde se van guardando los vértices intermedios. Fue inicia-
➥lizado en 0 en el método constructor). */
template <class T>
void DiGrafica<T>::FloydVerInt()
```

```
{
   int Ind1, Ind2, Ind3;
   for (Ind3= 0; Ind3 < NumVer; Ind3++)
        for (Ind1= 0; Ind1 < NumVer; Ind1++)
             for (Ind2= 0; Ind2 < NumVer; Ind2++)
                  if ((MatAdy[Ind1][Ind3] + MatAdy[Ind3][Ind2])
                     < MatAdy[Ind1][Ind2])
                  {
                       MatAdy[Ind1][Ind2]=
                       MatAdy[Ind1][Ind3] +
                       MatAdy[Ind3][Ind2];
                       VerInt[Ind1][Ind2]= Vertices[Ind3];
                  }
}
```

La tabla 8.4 presenta el resultado de aplicar el método `FloydVerInt` a la gráfica dirigida de la figura 8.9. En (a) aparece la matriz de distancias mínimas y en (b) la matriz que almacena los vértices intermedios. Esta matriz se interpreta de la siguiente manera: de V_1 se puede llegar a V_4 por medio del vértice 3, de V_1 a V_5 a través del vértice 4 y de V_2 se llega a V_5 pasando por el vértice 4. A su vez, de V_3 se llega a V_5 a través del vértice 4 y a V_2 por medio del vértice 5. En este último caso hay dos nodos intermedios entre V_3 y V_2 ($V_3 - V_4 - V_5 - V_2$). Del vértice 4 se puede ir a V_2 utilizando a V_5 como vértice intermedio. Finalmente, de V_6 se llega a V_2 y a V_4 a través del vértice 5.

TABLA 8.4 Obtención de la matriz de vértices intermedios usando `FloydVerInt`

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	10	15	21	26	999
V_2	999	0	999	12	17	999
V_3	999	22	0	6	11	999
V_4	999	16	999	0	5	999
V_5	999	11	999	5	0	999
V_6	999	20	999	14	9	0

(a)

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	0	0	3	4	0
V_2	0	0	0	0	4	0
V_3	0	5	0	0	4	0
V_4	0	5	0	0	0	0
V_5	0	0	0	0	0	0
V_6	0	5	0	5	0	0

(b)

Método `Dijkstra`

Este método **encuentra el camino más corto desde un vértice a todos los demás vértices de la gráfica dirigida**. La longitud del camino, si se usara un vértice intermedio, es la suma de las distancias entre cada uno de los nodos involucrados.

A continuación se presenta el método `Dijkstra` de la clase `DiGrafica`.

```
/* Método que encuentra la distancia mínima entre un vértice dado y los
➥demás vértices de una gráfica dirigida. En el arreglo DistMin se
➥almacenan las distancias mínimas desde el vértice origen a cada uno
➥de los otros nodos. Es decir DistMin[i] almacena la menor distancia
➥entre el vértice origen y el vértice i. */
template <class T>
void DiGrafica<T>::Dijkstra()
{
   int Aux[MAX], VertInc[MAX], Ver1, Ver2, Ind1, Ind2, Menor, Band,
       Origen;

   /* El arreglo VertInc se utiliza para guardar los vértices elegidos
   ➥por ser los de la distancia mínima. El arreglo Aux es un arreglo
   ➥lógico que indica si el nodo de la posición i ya fue incluido en
   ➥VertInc y de esta manera evitar ciclos. */
   for (Ind1= 0; Ind1 < NumVer; Ind1++)
   {
        Aux[Ind1]= 0;
        VertInc[Ind1]= 0;
   }
   cout<<"\n\n Ingrese vértice origen: ";
   cin>>Origen;
   Aux[Origen - 1]= 1;

   /* El arreglo donde se guardan las distancias mínimas del Origen a
   ➥los demás vértices se inicializa con los valores de la matriz de
   ➥adyacencia. */
   for (Ind1= 0; Ind1 < NumVer; Ind1++)
        DistMin[Ind1]= MatAdy[Origen][Ind1];

   for (Ind1= 0; Ind1<NumVer; Ind1++)
   {
        /* Se busca el vértice Ver1 en (Vertices - VertInc) tal que
        ➥DistMin[Ver1] sea el mínimo valor. Se usa el 999 como valor
        ➥inicial ya que es el elegido para indicar que no existe camino
        ➥entre dos vértices. */
        Menor= 999;
```

```
        for (Ind2= 1; Ind2 < NumVer; Ind2++)
             if (DistMin[Ind2] < Menor  &&  !Aux[Ind2])
             {
                  Ver1= Ind2;
                  Menor= DistMin[Ind2];
             }

        /* Se incluye Ver1 a VertInc y se actualiza el arreglo Aux. */
        VertInc[Ind1]= Ver1;
        Aux[Ver1]= 1;

        /* Se busca la distancia mínima para cada vértice Ver2 en
        ➥(Vertices - VertInc). */
        Ver2= 1;
        while (Ver2 < NumVer)
        {
           Band=0;
           Ind2= 1;
           while (Ind2 < NumVer &&  !Band)
             if  (VertInc[Ind2] ==  Ver2)
                  Band= 1;
             else
                  Ind2++;
           if  (!Band)
                DistMin[Ver2]=
                Minimo (DistMin[Ver2],
                       ➥DistMin[Ver1] + MatAdy[Ver1][Ver2]);
           Ver2++;
        }
     }
}
```

La tabla 8.5 presenta el resultado de aplicar el método `Dijkstra` a la gráfica dirigida de la figura 8.9. En (a) el vértice origen es V_1 y en (b) es V_4. Con el método se encontró que para ir de V_1 a V_2 la distancia mínima es 10 (que corresponde a un camino directo entre ambos vértices), V_3 es 15 (que también es un camino directo), V_4 es 21 (a través de V_3, ya que si fuera a través de V_2 sería 22), V_5 es 26 (a través de V_3 y de V_4). La tabla muestra también que no es posible ir de V_1 a V_6. Con respecto al vértice 4, aplicando el método, se encontró que no se puede llegar a los vértices 1 y 6. Las distancias mínimas entre V_4 y V_2 es 16 (a través de V_5) y entre V_4 y V_5 es 5 (camino directo).

TABLA 8.5 Obtención de la distancia mínima entre vértices usando `Dijkstra`

	V_1	V_2	V_3	V_4	V_5	V_6
Origen: V_1	0	10	15	21	26	999

(a)

	V_1	V_2	V_3	V_4	V_5	V_6
Origen: V_4	999	16	999	0	5	999

(b)

El programa 8.1 presenta la plantilla completa de la clase `DiGrafica`. En el caso de los métodos `Warshall`, `Floyd`, `FloydVerInt` y `Dijkstra` (ya analizados), se incluyeron sólo los prototipos y los encabezados.

Programa 8.1

```
#define MAX 10

/* Función auxiliar que obtiene el valor más pequeño de dos dados como
➥parámetros. La utiliza el método de Dijkstra. */
int Minimo(int Val1, int Val2)
{
   int Min= Val1;
   if (Val2 < Min)
        Min= Val2;
   return Min;
}


/* Definición de la plantilla de la clase DiGrafica. Se incluyen como
➥atributos, además de la matriz de adyacencia, el número de vértices y
➥su nombre, otros elementos que son utilizados en los métodos. */
template <class T>
class DiGrafica
{
   private:
       /* Declaración del arreglo donde se almacenan las distancias
       ➥entre los vértices. */
       T MatAdy[MAX][MAX];

       int NumVer, Vertices[MAX], DistMin[MAX], CerTran[MAX][MAX],
           VerInt[MAX][MAX];
```

```
    public:
        /* Método constructor y métodos auxiliares para leer la información
        ➥relacionada a la gráfica e imprimir los resultados obtenidos al
        aplicar los demás métodos. */
        DiGrafica();
        void Lee();
        void Imprime(int);

        /* Métodos que recorren una gráfica dirigida, encontrando caminos
        ➥de distancias mínimas. */
        void Warshall();
        void Floyd();
        void FloydVerInt();
        void Dijkstra();
};

/* Método constructor. Inicializa el número de vértices en cero y a la
➥matriz de adyacencias MatAdy con un valor arbitrario muy grande (999),
➥indicando que no existe camino entre los nodos correspondientes. Además,
➥asigna ceros a los arreglos auxiliares que se usan en los métodos. */
template <class T>
DiGrafica<T>::DiGrafica()
{
    int Ind1, Ind2;
    for (Ind1= 0; Ind1 < MAX; Ind1++)
    {
        DistMin[Ind1]= 0;
        for (Ind2= 0; Ind2 < MAX; Ind2++)
        {
            if (Ind1 != Ind2)
                MatAdy[Ind1][Ind2]= 999;
            else
                MatAdy[Ind1][Ind2]= 0;
            CerTran[Ind1][Ind2]= 0;
            VerInt[Ind1][Ind2]= 0;
        }
    }
    NumVer= 0;
}

/* Método que lee los datos de la gráfica dirigida directamente del
➥teclado. */
template <class T>
void DiGrafica<T>::Lee()
{
    int NumArcos, Indice, Origen, Destino;

    cout<<"\n\n Ingrese número de vértices de la gráfica dirigida: ";
    cin>>NumVer;
    cout<<"\n\n Ingrese los nombres de los vértices
          ➥de la gráfica dirigida: ";
```

```
    for (Indice= 0; Indice < NumVer; Indice++)
         cin>>Vertices[Indice];
    cout<<"\n\n Total de aristas de la gráfica dirigida: ";
    cin>>NumArcos;

    Indice= 0;
    while (Indice < NumArcos)
    {
       cout<<"\n\n Ingrese vértice origen: ";
       cin>>Origen;
       cout<<"\n\n Ingrese vértice destino: ";
       cin>>Destino;
       cout<<"\n\n Distancia de origen a destino: ";
       cin>>MatAdy[Origen - 1][Destino - 1];
       Indice++;
    }
}

/* Método que imprime información relacionada a una gráfica dirigida.
➥Por medio de un número entero se selecciona lo que se va a imprimir, lo
➥cual depende del método aplicado para recorrer la digráfica. */
template <class T>
void DiGrafica<T>::Imprime(int Opc)
{
   int Ind1, Ind2;

   switch(Opc)
   {
      /* Impresión de la matriz de adyacencia o de costos. */
      case 0:  cout<<"\n\n Matriz de Adyacencia o de Costos: \n\n";
               for (Ind1= 0; Ind1 < NumVer; Ind1++)
               {
                    cout<<Vertices[Ind1]<< ":  ";
                    for (Ind2= 0; Ind2 < NumVer; Ind2++)
                         cout<<MatAdy[Ind1][Ind2] << "\t";
                    cout<<endl;
               }
               break;
      /* Impresión de la cerradura transitiva correspondiente a la
      ➥matriz de adyacencia. Se obtiene cuando se aplica el método de
      ➥Warshall. */
      case 1:  cout<<"\n\n Cerradura Transitiva de la Matriz de
                    ➥Adyacencia: "<<endl;
               for (Ind1= 0; Ind1 < NumVer; Ind1++)
               {
                    cout<<Vertices[Ind1] <<": ";
                    for (Ind2= 0; Ind2 < NumVer; Ind2++)
                         cout<<CerTran[Ind1][Ind2]<<"\t";
```

```
                    cout << endl;
               }
               break;
          /* Impresión de la matriz de distancias mínimas entre todos los
          ➥vértices de la gráfica. Se obtiene por medio del método de Floyd. */
          case 2:  cout<<"\n\n Matriz de Distancias Mínimas: "<<endl;
               for (Ind1= 0; Ind1 < NumVer; Ind1++)
               {
                    cout<<Vertices[Ind1]<< ": ";
                    for (Ind2= 0; Ind2 < NumVer; Ind2++)
                         cout<<MatAdy[Ind1][Ind2] << "\t";
                    cout << endl;
               }
               break;
          /* Impresión de la matriz con los vértices intermedios usados
          ➥para establecer los caminos de distancias mínimas. Esta
          ➥impresión complementa la de la opción 2 cuando se aplica el
          ➥método FloydVerInt. */
          case 3:  cout<<"\n\n Vértices Intermedios para lograr distancias
                        ➥mínimas: "<<endl;
               for (Ind1= 0; Ind1 < NumVer; Ind1++)
               {
                    for (Ind2= 0; Ind2 < NumVer; Ind2++)
                         cout<<VerInt[Ind1][Ind2]<<"\t";
                    cout<<endl;
               }
               break;
          /* Impresión de las distancias mínimas entre un vértice y los
          ➥demás. Se obtiene con el método de Dijkstra. */
          case 4:  cout<<"\n\n Distancias mínimas  a partir del vértice:
                        ➥"<<Vertices[0]<<"\n\n";
               for (Ind1= 0; Ind1 < NumVer; Ind1++)
                    cout<<" "<<DistMin[Ind1]<<"\t"<<endl;
               break;

          default: break;
     }
     cout<<endl;
}

/* Este método corresponde al que se presentó anteriormente por lo que
➥sólo se deja indicado. */
template <class T>
void DiGrafica<T>::Warshall()
{}
/* Este método corresponde al que se presentó anteriormente por lo que
➥sólo se deja indicado. */
template <class T>
void DiGrafica<T>::Floyd()
{}
```

```
/* Este método corresponde al que se presentó anteriormente por lo que
➥sólo se deja indicado. */
template <class T>
void DiGrafica<T>::FloydVerInt()
{}

/* Este método corresponde al que se presentó anteriormente por lo que
➥sólo se deja indicado. */
template <class T>
void DiGrafica<T>::Dijkstra()
{}
```

8.2.4 Aplicación de gráficas dirigidas

Considere la gráfica de la figura 8.11 que representa un subconjunto de la red ferroviaria de un determinado país, *asumiendo que las vías pueden usarse en una sola dirección*. Los vértices representan ciudades, los arcos tramos de vías y las etiquetas de los arcos costos de los pasajes entre los vértices (ciudades) adyacentes. Cada una de las ciudades se identifica por un número, según se muestra en la tabla 8.6. El programa 8.2 es una aplicación muy simple en la que se ilustra el uso de los métodos vistos para determinar si existe o no comunicación entre las ciudades, así como para encontrar las rutas de menor costo entre ellas.

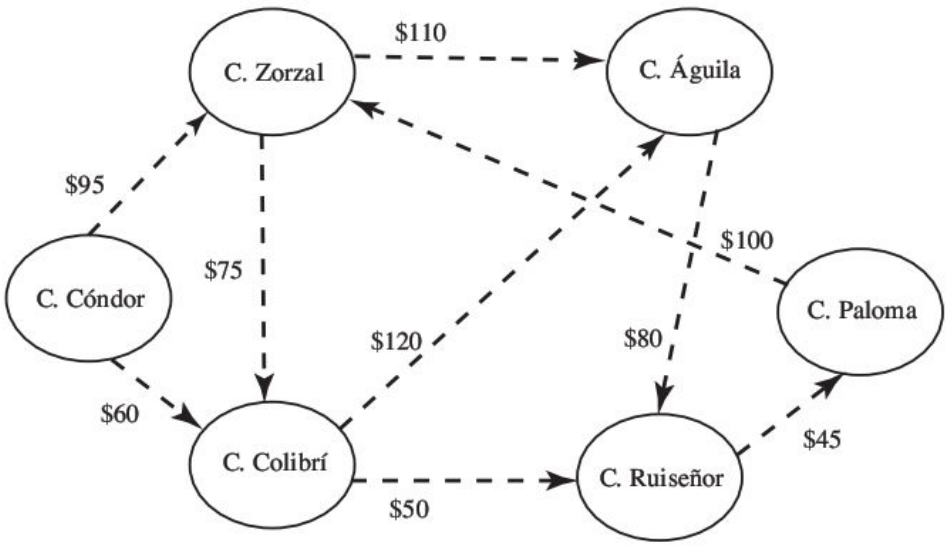


FIGURA 8.11 *Aplicación de gráficas dirigidas*

TABLA 8.6 Lista de ciudades de la red ferroviaria

Ciudad	*Número asociado*
Cóndor	1
Zorzal	2
Águila	3
Paloma	4
Ruiseñor	5
Colibrí	6

La plantilla presentada en el programa 8.1 se guardó en la biblioteca *"DiGrafica.h"* y se utilizó en el programa 8.2 para encontrar las rutas de menor costo entre los vértices de la gráfica de la figura 8.11.

Programa 8.2

```
/* Aplicación de gráficas dirigidas para encontrar ciudades comunicadas
➥entre sí por el sistema ferroviario, así como los costos mínimos para
➥ir de una ciudad a las otras o entre todas las ciudades. */

#include "DiGrafica.h"

int Menu()
{
   int Opc;
   do {
       cout<<"\n\nOpciones\n";
       cout<<"\n(1) Ciudades que están comunicadas entre sí.";
       cout<<"\n(2) Mínimo costo entre todas las ciudades. ";
       cout<<"\n(3) Mínimo costo entre todas las ciudades y ciudades
             ➥intermedias. ";
       cout<<"\n(4) Mínimo costo entre una ciudad y las otras. ";
       cout<<"\n(5) Finalizar el proceso.";
       cout<<"\n\nIngrese opción elegida:";
       cin>>Opc;
   } while (Opc < 1 || Opc  > 5);
   return Opc;
}


void main()
{
   DiGrafica<int> RedFerrov;
   int Opc;
```

```
    cout<<"\n\n\nIngrese datos de ciudades y costos de pasajes\n";
    RedFerrov.Lee();

    do {
        Opc= Menu();
        switch (Opc)
        {
           /* El método Warshall permite encontrar todas las ciudades que
           ➥están comunicadas entre sí por medio de la red ferroviaria. */
           case 1: {
                     RedFerrov.Warshall();
                     RedFerrov.Imprime(1);
                     break;
                    }
           /* El método Floyd permite encontrar los costos mínimos para
           ➥visitar a todas las ciudades que están comunicadas entre sí
           ➥por medio de la red ferroviaria. */
           case 2: {
                     RedFerrov.Floyd();
                     RedFerrov.Imprime(2);
                     break;
                    }
           /* El método FloydVerInt permite encontrar los costos mínimos
           ➥para visitar todas las ciudades que están comunicadas entre
           ➥sí por medio de la red ferroviaria y las ciudades intermedias
           ➥(cuando no existe camino directo, o si éste no fuera el de
           ➥costo mínimo). */
           case 3: {
                     RedFerrov.FloydVerInt();
                     RedFerrov.Imprime(3);
                     break;
                    }
           /* El método Dijkstra permite encontrar los costos mínimos
           ➥para ir de una ciudad a todas las otras ciudades con las que
           ➥está comunicada por medio de la red ferroviaria. */
           case 4: {
                     RedFerrov.Dijkstra();
                     RedFerrov.Imprime(4);
                     break;
                    }
        }
    } while (Opc < 5 && Opc > 0);
}
```

La tabla 8.7 presenta el resultado de aplicar el método `Warshall` a la gráfica de la figura 8.11. Recuerde que en esta implementación, en la diagonal queda 1. La tabla indica que se puede llegar desde cualquier ciudad a cualquiera de las otras, con excepción de la primera (`Ciudad Cóndor`).

TABLA 8.7 Resultado de aplicar `Warshall`

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	0	1	1	1	1	1
3	0	1	1	1	1	1
4	0	1	1	1	1	1
5	0	1	1	1	1	1
6	0	1	1	1	1	1

La tabla 8.8 presenta el resultado de aplicar el método `Floyd` a la gráfica de la figura 8.11. La tabla muestra el mínimo que deberá pagarse en pasajes para trasladarse entre las ciudades. En la implementación del método se usó el valor 999 para indicar que no hay vías entre las ciudades correspondientes a la posición ocupada por dicho valor.

TABLA 8.8 Resultado de aplicar `Floyd`

	1	2	3	4	5	6
1	0	95	180	155	110	60
2	999	0	110	170	125	75
3	999	225	0	125	80	300
4	999	100	210	0	225	175
5	999	145	255	45	0	220
6	999	195	120	95	50	0

La tabla 8.9 presenta el resultado de aplicar el método `FloydVerInt` a la gráfica de la figura 8.11. La tabla muestra las ciudades intermedias que deben visitarse para llegar a aquellas con las cuales no hay un camino directo o cuando se encuentre un costo menor. Por ejemplo, en la posición (3,2) está el 5 que indica que para ir de la ciudad 3 a la ciudad 2 se requiere pasar por la 5. A su vez, en la posición (5,2) está el 4 que es la ciudad por la que se pasa para ir de la 5 a la 2. Por lo tanto, la trayectoria completa para ir de la 3 a la 2 es: 3 – 5 – 4 – 2.

TABLA 8.9 Resultado de aplicar `FloydVerInt`

	1	2	3	4	5	6
1	0	0	6	6	6	0
2	0	0	0	6	6	0
3	0	5	0	5	0	5
4	0	0	2	0	6	2
5	0	4	4	0	0	4
6	0	5	0	5	0	0

La tabla 8.10 presenta el resultado de aplicar el método `Dijkstra` a la gráfica de la figura 8.11, tomando como ciudad (vértice) origen a la 1 (`Cóndor`). Se puede observar que el importe mínimo a pagar para ir a la ciudad 2 (`Zorzal`) es de $95, mientras que para llegar a la ciudad 3 (`Águila`) es de $180, y según lo mostrado en la tabla anterior es a través de la ciudad 6 (`Colibrí`). Por su parte, para ir a la ciudad 4 (`Paloma`) se necesita pagar mínimo $155 y se usan dos ciudades intermedias, según la información desplegada en la tabla 8.9. Por último, para ir a las ciudades 5 (`Ruiseñor`) y 6 (`Colibrí`) se requiere pagar $110 y $60 respectivamente.

TABLA 8.10 Resultado de aplicar `Dijkstra`

	1	2	3	4	5	6
Origen: 1	0	95	180	155	110	60

8.3 Gráficas no dirigidas

Una ***gráfica no dirigida*** o ***gráfica*** se caracteriza porque sus aristas son pares no ordenados de vértices. Por lo tanto, si existe una arista o arco de V_1 a V_2, ésta será la misma que de V_2 a V_1, se grafica sin flecha al final y se expresa como:

$$a = (V_1, V_2) = (V_2, V_1)$$

Debido a esta característica, las gráficas son muy útiles cuando se tienen datos y relaciones ***simétricas*** entre ellos. Suponga que se quiere representar una red de comunicación entre servidores, ubicados en diferentes edificios de una misma empresa. Cada uno de los vértices se corresponderá con cada uno de los servido-

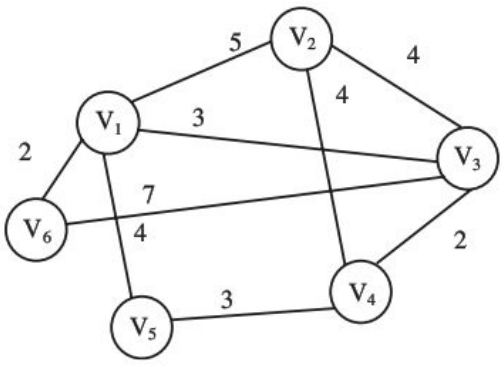
res, mientras que los arcos representarán al medio elegido para llevar a cabo la comunicación. Usando una gráfica para modelar esta situación se estará indicando que si existe comunicación de un nodo a otro, entonces también es posible comunicarse de este último al primero. Por otra parte, si la arista estuviera etiquetada con la velocidad de comunicación del medio empleado, dicha velocidad sería la misma en cualquiera de los sentidos.

8.3.1 Representación de una gráfica

Este tipo de gráficas, igual que las digráficas, se representan por medio de una matriz o de una lista de adyacencias. En este libro se empleará la matriz. Si las aristas tienen asociado un costo o distancia, la matriz recibe el nombre de matriz de adyacencia etiquetada o matriz de distancias o costos.

Como ya se mencionó, las gráficas se utilizan para representar relaciones simétricas entre objetos, es decir cuando sea exactamente la misma relación de V_1 a V_2 que de V_2 a V_1. Por lo tanto la matriz de adyacencia resulta una matriz simétrica.

La figura 8.12 presenta un ejemplo de una gráfica con costos en las aristas y su correspondiente matriz de adyacencia etiquetada. Observe que el valor almacenado en cada posición (i, j) de la matriz es igual al valor de la posición (j, i). Aprovechando esta característica, y con el objeto de ahorrar espacio de almacenamiento, se puede usar un arreglo unidimensional para guardar sólo los elementos de la matriz triangular inferior o superior.



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	5	3	0	4	2
v_2	5	0	4	4	0	0
v_3	3	4	0	2	0	7
v_4	0	4	2	0	3	0
v_5	4	0	0	3	0	0
v_6	2	0	7	0	0	0

FIGURA 8.12 *Ejemplo de gráfica no dirigida y su representación por medio de una matriz de costos*

8.3.2 La clase `gráfica` no dirigida

Se define la clase `Grafica` para representar este tipo de estructura de datos. Considerando que se usa una matriz de adyacencia para almacenar la información relacionada con la gráfica, la plantilla correspondiente resulta similar a la vista para las gráficas dirigidas. Los atributos son el número de vértices y sus nombres (en esta implementación se declararon como enteros) y la matriz de adyacencias. Los métodos se presentan y explican en la siguiente sección.

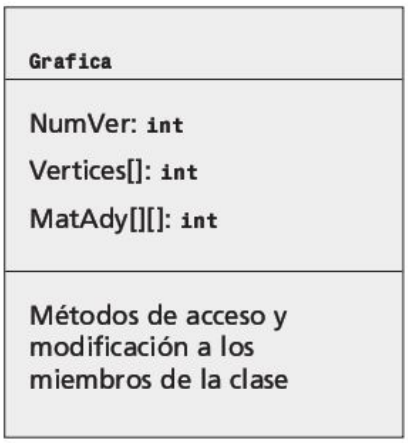


FIGURA 8.13 *Clase* `Grafica`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a la definición de la clase `Grafica` de la figura 8.13. Se incluyen sólo los prototipos de los métodos; su descripción se da más adelante.

```
/* Definición de la plantilla de la clase Grafica. Se incluyen como
atributos el total de vértices (NumVer), los costos/distancias entre los
vértices (MatAdy) y sus nombres (Vertices). */
template <class T>
class Grafica
{
   private:
       T  MatAdy[MAX][MAX];
       int NumVer, Vertices[MAX];
```

```
    public:
        /* Método constructor y métodos auxiliares para leer la información
        ➥relacionada a la gráfica e imprimir los resultados obtenidos al
        ➥aplicar los demás métodos. */
        Grafica();

        void Lee();
        void Imprime();

        /* Métodos que permiten el cálculo del árbol abarcador de costo
        ➥mínimo. */
        void Prim();
        void Kruskal();
};


/* Declaración del método constructor. Inicializa el número de vértices
➥en cero y la matriz de distancias con un valor arbitrario muy grande
➥(999), excepto en la diagonal principal, donde el costo es cero. */
template <class T>
Grafica<T>::Grafica()
{
    int Ind1, Ind2;
    for (Ind1= 0; Ind1<MAX; Ind1++)
         for (Ind2= 0; Ind2<MAX; Ind2++)
             if (Ind1 != Ind2)
                 MatAdy[Ind1][Ind2]= 999;
             else
                 MatAdy[Ind1][Ind2]= 0;
    NumVer= 0;
}
```

8.3.3 Recorrido de gráficas no dirigidas

Las operaciones que se aplican sobre una gráfica están orientadas a encontrar los caminos de costos mínimos entre sus vértices. Antes de presentar estos métodos, resulta necesario explicar algunos conceptos.

- ***Árbol libre*** es una gráfica conexa acíclica.
- ***Árbol abarcador*** es un árbol libre que conecta todos los vértices de la gráfica. El costo del árbol se calcula como la suma de los costos de las aristas. Por lo tanto, un árbol abarcador de costo mínimo es el formado por las aristas de menor costo.

Método de `Prim`

Este método **encuentra el árbol abarcador de costo mínimo de una gráfica**. Trabaja con dos conjuntos de vértices, uno de los cuales es `Vertices` (el conjunto de todos los vértices de G) y el otro es `VerAux` (que es un subconjunto de `Vertices`). Inicialmente `VerAux` tiene asignado el valor del primer índice. Los pasos principales de este método son:

1. Buscar la arista (`ver1`, `ver2`) de costo mínimo de tal forma que conecte a `VerAux` con la subgráfica correspondiente a (`Vertices - VerAux`).
2. Agregar el vértice `ver2` al conjunto `VerAux`.
3. Repetir los pasos 1 y 2 hasta que se alcance la condición (`VerAux = Vertices`).

A continuación se presenta el método `Prim` de la clase `Grafica`.

```
/* Este método encuentra el árbol abarcador de costo mínimo de una
➥gráfica. En el arreglo VerAux se almacenan los vértices con menor costo
➥que van formando el árbol abarcador. */
template <class T>
void Grafica<T>::Prim()
{
   int MCosto[MAX], VerAux[MAX], Ind1, Ind2, VerMen, MenCos;

   /* Inicializa el subconjunto de vértices VerAux con el valor del
   ➥primer vértice. */
   for (Ind1= 0; Ind1<NumVer; Ind1++)
   {
        MCosto[Ind1]= MatAdy[0][Ind1];
        VerAux[Ind1]= 0;
   }

   cout<<"\n\n\nArcos y costos del árbol abarcador de costo mínimo\n\n";
   cout<<"\nVértice   Vértice   Costo \n";

   /* Encuentra el vértice VerMen en (Vertices - VerAux) tal que el
   ➥costo de ir de dicho vértice a uno de VerAux sea mínimo. */
   for (Ind1= 0; Ind1 < NumVer - 1; Ind1++)
   {
        MenCos= MCosto[1];
        VerMen= 1;
        for (Ind2= 2; Ind2 < NumVer; Ind2++)
             if (MCosto[Ind2] < MenCos)
             {
                 MenCos= MCosto[Ind2];
                 VerMen= Ind2;
             }
```

```
            cout<<"\n    "<<Vertices[VerMen]<<"    -    "
                ➥<<Vertices[VerAux[VerMen]]
                <<"        "<<MatAdy[VerMen][VerAux[VerMen]];
            /* Se agrega el vértice VerMen a VerAux y se redefinen los
            ➥costos asociados. */
            MCosto[VerMen]= 1000;
            for (Ind2= 1; Ind2 < NumVer; Ind2++)
                if ((MatAdy[VerMen][Ind2] < MCosto[Ind2]) &&
                ➥MCosto[Ind2] < 1000)
                {
                    MCosto[Ind2]= MatAdy[VerMen][Ind2];
                    VerAux[Ind2]= VerMen;
                }
        }
        cout<<"\n\n";
    }
```

La figura 8.14 presenta el resultado de aplicar el método de `Prim` a la gráfica de la figura 8.12 para obtener el árbol abarcador de costo mínimo.

Matriz de adyacencia etiquetada

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	5	3	999	4	2
v_2	5	0	4	4	999	999
v_3	3	4	0	2	999	7
v_4	999	4	2	0	3	999
v_5	4	999	999	3	0	999
v_6	2	999	7	999	999	0

Árbol abarcador de costo mínimo

Vértice	Vértice	Costo
6	1	2
3	1	3
4	3	2
5	4	3
2	3	4

FIGURA 8.14 *Obtención del árbol abarcador de costo mínimo utilizando* `Prim`

La primera tabla muestra la matriz de costos, y la segunda el conjunto de vértices y arcos que forman el árbol abarcador de costo mínimo. Se puede observar que el método obtuvo las aristas necesarias para comunicar a todos los vértices con el menor costo.

Método de `Kruskal`

Este método, lo mismo que el de `Prim`, **genera el árbol abarcador de costo mínimo de una gráfica**. Básicamente consiste en seleccionar las aristas de menor costo y formar el árbol con sus vértices. Los pasos principales de este método son:

1. Generar una partición del conjunto de vértices. Inicialmente la partición es de longitud uno (una por cada vértice): `Partic= {{1}, {2},..., {NumVer}}`.
2. Seleccionar la arista de menor costo. Si ésta une vértices que se encuentran en particiones distintas, éstas se reemplazan por su unión.
3. Repetir el paso 2 hasta que el conjunto de particiones quede formado por una sola partición igual al conjunto de vértices: `Partic= {1, 2,..., NumVer} = Vertices`.

A continuación se presenta el método `Kruskal` de la clase `Grafica`.

```
/* Este método encuentra el árbol abarcador de costo mínimo de una
➥gráfica. */
template <class T>
void Grafica<T>::Kruskal()
{
   /* El arreglo auxiliar ArisCosto[][] almacena en cada renglón los
   ➥datos de una arista: vértices adyacentes y costo. El arreglo
   ➥Partic[] almacena particiones de Vertices. Inicialmente
   ➥Partic= {{1},{2},...,{NumVer}}. */
   int ArisCosto[2*MAX][3], Partic[MAX], Ind1, Ind2, Ver1, Ver2,
   ➥TotAris, Menor, Mayor, Band;

   /* Inicializa Partic[]. */
   for (Ind1= 0; Ind1 < NumVer; Ind1++)
        Partic[Ind1]= Ind1;

   /* Inicializa ArisCosto[][]. */
   Ver1= 0;
   Ver2= 0;
   TotAris= 0;
```

```
for (Ind1= 0; Ind1 < NumVer; Ind1++)
     for (Ind2= Ind1; Ind2 < NumVer; Ind2++)
         if (MatAdy[Ind1][Ind2] != 0 && MatAdy[Ind1][Ind2] != 999)
         {
             ArisCosto[Ver1][Ver2++]= Ind1;
             ArisCosto[Ver1][Ver2++]= Ind2;
             ArisCosto[Ver1++][Ver2]= MatAdy[Ind1][Ind2];
             Ver2= 0;
             TotAris++;
         }

/* Ciclo en el cual se seleccionan aristas y se agregan los vértices
➥mientras existan vértices en Partic que se encuentren en distintas
➥particiones. */
Band= 0;
while (Band != 1)
{
   /* Se selecciona la arista de menor costo. */
   Menor= 999;
   for (Ind1= 0; Ind1 < TotAris; Ind1++)
        if (ArisCosto[Ind1][2] < Menor)
        {
            Ver1= ArisCosto[Ind1][0];
            Ver2= ArisCosto[Ind1][1];
            Menor= ArisCosto[Ind1][2];
            Ind2= Ind1;
        }
   /* Se elimina la arista de menor costo de la matriz ArisCosto. */
   ArisCosto[Ind2][2]= 999;
   /* Se verifica que la arista (Ver1, Ver2) una dos vértices que
   ➥pertenecen a particiones diferentes. */
   if (Partic[Ver1] != Partic[Ver2])
   {
       cout<<"\nVértice: "<<Vertices[Ver1]<<" Vértice: "
           ➥<<Vertices[Ver2] <<" Costo: "<<MatAdy[Ver1][Ver2]<<"\n\n";
       Mayor= Maximo(Partic[Ver1], Partic[Ver2]);

       for (Ind1= 0; Ind1 < NumVer; Ind1++)
           if (Ind1 ==  Ver1 || Ind1 ==  Ver2 ||
           ➥Partic[Ind1] ==  Mayor)
               Partic[Ind1]= Minimo(Partic[Ver1], Partic[Ver2]);
   }
   /* Ciclo para determinar si quedan vértices en particiones
   ➥diferentes. */
   Ind1= 0;
```

```
        while (Ind1 < NumVer  &&  !Band)
        {
          if (Partic[Ind1] != 0)
              Band= 1;
          Ind1++;
        }
        /* Si existen particiones en Partic se debe seguir con el
        proceso. */
        Band= !Band;
    }
}
```

La figura 8.15 presenta el resultado de aplicar el método de `Kruskal` a la gráfica de la figura 8.12 para obtener su árbol abarcador de costo mínimo. El árbol generado es el mismo que con el método de `Prim`, lo que cambia es el proceso y el orden en el cual se van eligiendo las aristas

Matriz de adyacencia etiquetada

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	5	3	999	4	2
v_2	5	0	4	4	999	999
v_3	3	4	0	2	999	7
v_4	999	4	2	0	3	999
v_5	4	999	999	3	0	999
v_6	2	999	7	999	999	0

Árbol abarcador de costo mínimo

Vértice	Vértice	Costo
1	6	2
3	4	2
1	3	3
4	5	3
2	3	4

FIGURA 8.15 *Obtención del árbol abarcador de costo mínimo utilizando* `Kruskal`

En la figura están la matriz de costos y las aristas seleccionadas para formar el árbol abarcador de costo mínimo. En este caso, la primera arista seleccionada por el método es la (1, 6), con un costo de 2. Luego se obtiene la (3, 4) también con un costo de 2. Se continúa así hasta completar el árbol.

El programa 8.3 presenta la plantilla completa de la clase `Grafica`.

Programa 8.3

```
/* Máximo número de vértices que maneja la clase Gráfica. */
#define MAX 20

/* Función auxiliar que obtiene el valor más pequeño de dos dados como
➥parámetros. La utiliza el método de Kruskal. */
int Minimo (int Val1, int Val2)
{
      int Min= Val1;
      if (Val2 < Min)
          Min= Val2;
      return Min;
}

/* Función auxiliar que obtiene el valor más grande de dos dados como
➥parámetros. La utiliza el método de Kruskal. */
int Maximo (int Val1, int Val2)
{
      int Max= Val1;
      if (Val2 > Max)
          Max= Val2;
      return Max;
}

/* Definición de la plantilla de la clase Grafica. Se incluyen como
➥atributos la matriz de adyacencia (MatAdy), el total de vértices
➥(NumVer) y sus nombres (Vertices). */
template <class T>
class Grafica
{
   private:
       T  MatAdy[MAX][MAX];
       int NumVer, Vertices[MAX];
```

```
   public:
       /* Método constructor y métodos auxiliares para leer la informa-
       ción relacionada a la gráfica e imprimir los resultados obtenidos
       al aplicar los demás métodos. */
       Grafica();
       void Lee();
       void Imprime();

       /* Métodos que calculan el árbol abarcador de costo mínimo. */
       void Prim();
       void Kruskal();
};

/* Declaración del método constructor. Inicializa el número de vértices
➥en cero y la matriz de distancias con un valor arbitrario muy grande
➥(999), excepto en la diagonal principal, donde el costo es cero. */
template <class T>
Grafica<T>::Grafica()
{
   int Ind1, Ind2;
   for (Ind1= 0; Ind1 < MAX; Ind1++)
        for (Ind2= 0; Ind2 < MAX; Ind2++)
             if (Ind1 != Ind2)
                 MatAdy[Ind1][Ind2]= 999;
             else
                 MatAdy[Ind1][Ind2]= 0;
   NumVer= 0;
}

/* Método que lee del teclado la información de la gráfica. En esta
➥solución el nombre de los vértices sólo pueden ser valores enteros. */
template <class T>
void Grafica<T>::Lee()
{

       int Aristas, Costo, Ind1, Origen, Destino;

       cout<<"\n\nIngrese total de vértices de la gráfica:  ";
       cin>>NumVer;
       for (Ind1= 0; Ind1 < NumVer; Ind1++)
       {
            cout<<"\nIngrese el nombre del vértice:  ";
            cin>>Vertices[Ind1];
       }
```

```
        cout<<"\n\nIngrese total de aristas de la gráfica:  ";
        cin>>Aristas;
        Ind1= 0;
        while (Ind1 < Aristas)
        {
             cout<<"\nVértice origen:  ";
             cin>>Origen;
             cout<<"\nVértice destino:  ";
             cin>>Destino;
             cout<<"\nCosto de ir de "<<Origen<<" a "<<Destino<<": ";
             cin>>Costo;
             MatAdy[Origen - 1][Destino - 1]= Costo;
             MatAdy[Destino - 1][Origen - 1]= Costo;
             Ind1++;
        }
}

/* Este método corresponde al presentado más arriba por lo que sólo se
➥deja indicado. */
template <class T>
void Grafica<T>::Prim()
{

}

/* Este método corresponde al presentado más arriba por lo que sólo se
➥deja indicado.*/
template <class T>
void Grafica<T>::Kruskal()
{

}
```

8.3.4 Aplicación de gráficas no dirigidas

Considere la gráfica de la figura 8.16 que representa un subconjunto de la red caminera de México. Los vértices representan ciudades; los arcos, carreteras y las etiquetas de los arcos, distancias entre las ciudades. Cada una de las ciudades se identificará por un número, según se muestra en la tabla 8.11. El programa 8.4 es una aplicación muy simple en la que se ilustra el uso de los métodos para obtener el árbol abarcador de costo mínimo, el cual en este caso está formado por las carreteras de menor distancia que unen a todas las ciudades involucradas.

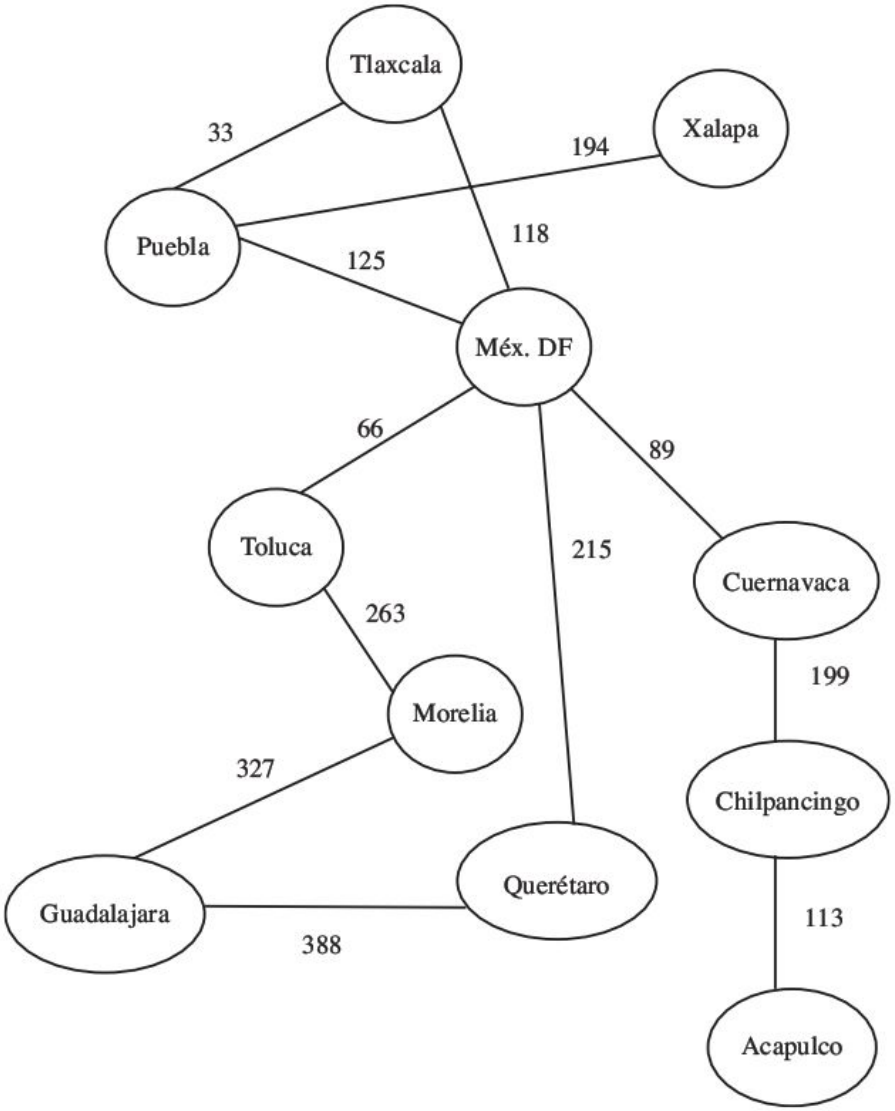


FIGURA 8.16 *Red caminera (parcial) de México*

TABLA 8.11 Lista de ciudades de la red caminera

Ciudad	*Número asociado*
Xalapa	1
Tlaxcala	2
Puebla	3
México, D.F.	4
Cuernavaca	5
Chilpancingo	6
Acapulco	7
Querétaro	8
Guadalajara	9
Morelia	10
Toluca	11

La plantilla presentada en el programa 8.3 se guardó en la biblioteca *"Grafica.h"* y se utilizó en el programa 8.4 para encontrar el árbol abarcador de costo mínimo.

Programa 8.4

```
/* Aplicación del concepto de gráficas para encontrar el conjunto mínimo de
➥carreteras, con el menor costo asociado, que una un grupo de ciudades. */
#include "Grafica.h"

void main()
{
   Grafica<int> Caminos;
   cout<<"\n\nIngrese datos de las ciudades (vértices) y de las
         ➥carreteras (aristas)\n\n";
   Caminos.Lee();
   Caminos.Imprime();
   cout<<"\n\nLa red mínima de carreteras requerida para unir todas
         ➥las ciudades es:\n";
   Caminos.Kruskal();
   Caminos.Prim();
}
```

La tabla 8.12 muestra la matriz de adyacencia (en este caso de distancias entre ciudades); la 8.13, el resultado generado por el método `Kruskal`; y finalmente, la 8.14, el correspondiente al método `Prim`. Es importante señalar que los árboles

abarcadores obtenidos por ambos métodos son los mismos, sólo cambia el orden en el cual se seleccionan las aristas de menor costo.

TABLA 8.12 Matriz de adyacencia/costos de la red caminera

	1	2	3	4	5	6	7	8	9	10	11
1	0	999	194	999	999	999	999	999	999	999	999
2	999	0	33	118	999	999	999	999	999	999	999
3	194	33	0	125	999	999	999	999	999	999	999
4	999	118	125	0	89	999	999	215	999	999	66
5	999	999	999	89	0	199	999	999	999	999	999
6	999	999	999	999	199	0	113	999	999	999	999
7	999	999	999	999	999	113	0	999	999	999	999
8	999	999	999	215	999	999	999	0	388	999	999
9	999	999	999	999	999	999	999	388	0	327	999
10	999	999	999	999	999	999	999	999	327	0	263
11	999	999	999	66	999	999	999	999	999	263	0

Se presenta la red mínima de carreteras requerida para unir todas las ciudades. Las dos primeras columnas almacenan los vértices adyacentes (vértices extremos de la arista) y la tercera el costo asociado a la arista (la distancia entre las ciudades representadas por los vértices involucrados).

TABLA 8.13 Resultado obtenido aplicando `Kruskal`

Del vértice/ciudad	*Al vértice/ciudad*	*Costo/distancia*
2	3	33
4	11	66
4	5	89
6	7	113
2	4	118
1	3	194
5	6	199
4	8	215
10	11	263
9	10	327

TABLA 8.14 Resultado obtenido aplicando `Prim`

Del vértice/ciudad	*Al vértice/ciudad*	*Costo/distancia*
3	1	194
2	3	33
4	2	118
11	4	66
5	4	89
6	5	199
7	6	113
8	4	215
10	11	263
9	10	327

Como ya se mencionó, el contenido de las tablas 8.13 y 8.14 es el mismo; es decir, los dos métodos obtuvieron el árbol abarcador de costo mínimo formado por el mismo conjunto de aristas. La diferencia es el proceso aplicado y el orden en el cual se seleccionan las aristas que formarán el árbol.

8.4 Búsqueda

En las secciones anteriores se analizaron operaciones de búsqueda de trayectorias o caminos entre los distintos vértices de una gráfica. En esta sección se analizan dos estilos de búsqueda aplicadas en la resolución de problemas. Los estados del problema se representan por medio de los vértices y los pasos necesarios para pasar de un estado a otro por medio de las aristas.

De acuerdo al orden en el cual se generan (operación llamada ***expansión***) los vértices sucesores de uno dado, los métodos de búsqueda se clasifican en ***búsqueda en profundidad*** (conocida también por su nombre en inglés *Depth First*) y ***búsqueda a lo ancho*** (*Breadth First*).

8.4.1 Búsqueda en profundidad (`Depth First`)

Este tipo de búsqueda se lleva a cabo generando todos los estados posibles a partir del vértice inicial, pero sólo considerando una de sus ramas o vértices adyacentes. Es decir, en cada nodo descendiente se elige sólo uno de sus hijos para proseguir con la búsqueda del estado solución. De ahí el nombre *en profundidad*.

Se empieza con el estado (nodo) inicial y se expande sólo uno de sus vértices adyacentes, y sobre éste se aplica el mismo criterio. La operación de búsqueda termina cuando se llega al estado final o bien, cuando se alcanza el nivel de profundidad establecido como límite. Cuando se presenta esta última condición se puede retomar la búsqueda a partir de alguno de los vértices no expandidos.

La figura 8.17 presenta un ejemplo de una gráfica en la cual se aplica búsqueda *en profundidad* para llegar al estado final. Cada uno de los vértices representa posibles estados de un problema. El nodo con valor ***A*** es el estado inicial y el que almacena la ***N*** es el final. Las aristas más gruesas indican el camino seguido para llegar al estado solución. Como se puede observar, del vértice ***A*** se puede llegar a ***T*** y a ***Y***. Considerando el tipo de búsqueda se expande (visita) sólo uno de los nodos, en este caso ***T***. A partir de ***T*** se puede ir a ***X***, ***Y*** o ***D***. Se expande sólo ***X***. De ***X*** se genera el estado ***L***. Desde éste se puede ir a ***K*** o ***E***. Se elige ***K***, de éste se pasa a ***Z***, luego a ***S*** y finalmente se llega al estado meta: ***N***.

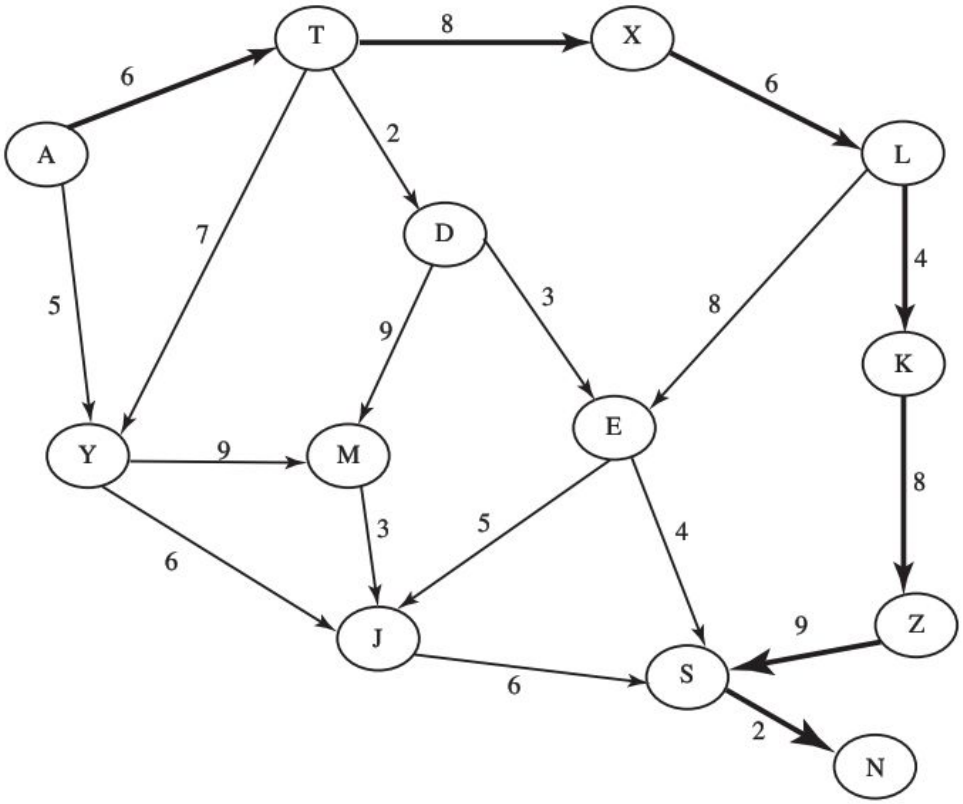


FIGURA 8.17 *Ejemplo de búsqueda en profundidad*

Al implementar este método se requiere usar dos listas para ir almacenando los vértices visitados y aquellos pendientes por visitar. Los primeros se guardan en una lista llamada `Visitado` y los otros en `NoVisitado`. Los principales pasos de este método son:

1. Guardar el vértice inicial en la lista `NoVisitado`.
2. Sacar el primer elemento (vértice `VertiX`) de la lista `NoVisitado`.
3. Evaluar si `VertiX` está en la lista `Visitado` y si el nivel alcanzado (profundidad) es menor o igual al permitido.
   3.1. Si la respuesta es negativa entonces obtener todos los vértices adyacentes de `VertiX` y guardar a éste en `Visitado`.
      3.1.1. Si tiene vértices adyacentes y no son el estado final entonces guardarlos al ***inicio*** de la lista `NoVisitado`.
      3.1.2. Si tiene vértices adyacentes y alguno de ellos es el estado final entonces el proceso termina con éxito.
      3.1.3. Si no tiene vértices adyacentes, ir al paso 4.
   3.2. Si la respuesta es afirmativa (el vértice está en `Visitado` o se llegó al nivel de profundidad permitido) entonces ir al paso 4.
4. Repetir los pasos 2 y 3 hasta que se llegue al estado final o bien hasta que la lista `NoVisitado` quede vacía.

A continuación se presenta el método `Depth-First` de la clase `DiGrafica`. Este método usa la clase `Lista` (ver capítulo 6) para definir los objetos `Visitado` y `NoVisitado`. Además, con el objeto de reutilizar código en la implementación del método `Breadth First`, se desarrollaron dos métodos auxiliares: `BuscaVertice()` y `VerticesAdyacentes()`. El primero de ellos determina si un vértice dado como parámetro es o no un vértice de la digráfica. Mientras que el segundo genera una lista con los vértices adyacentes de un vértice dado como parámetro. Estos se explican con mayor detalle más adelante.

```
/* Este método busca una solución (estado final) de un problema
➥representado por medio de una gráfica. Recibe como parámetro el nivel
➥máximo de profundidad permitido. En esta implementación se considera
➥el estado final como el último vértice de la digráfica. Regresa uno si
➥llega al estado meta y cero en caso contrario.En el método se usan los
➥atributos definidos en la clase DiGrafica. Se declaran tres objetos
```

```
➥de la clase Lista  para almacenar los vértices que se van visitando
➥y los pendientes de visitar, así como una lista auxiliar para guardar
➥los vértices adyacentes de uno dado. */
template <class T>
int DiGrafica<T>::DepthFirst(int NivelProf)
{
   int Indice, EstadoFinal= 0, VisitaAux[MAX], Resp= 1;
   Lista<T> Visitado, NoVisitado, ListaAux;
   T VertiX;

   for (Indice= 0; Indice < NumVer; Indice++)
        VisitaAux[Indice]= 0;

   /* Se guarda el primer vértice (representa el estado inicial) de la
   ➥digráfica en la lista NoVisitado. */
   NoVisitado.InsertaFinal(Vertices[0]);

   /* En el arreglo auxiliar VisitaAux se indica que el primer vértice
   ➥ya fue visitado, para evitar caer en ciclos. */
   VisitaAux[0]= 1;

   /* Se repiten los pasos del algoritmo de búsqueda mientras no se llegue
   ➥al estado final y mientras queden elementos en la lista NoVisitado. */
   Indice= 0;
   while (!NoVisitado.ListaVacia() && !EstadoFinal)
   {
      /* Se saca el primer elemento de NoVisitado. */
      VertiX= NoVisitado.Elimina();

      /* Se evalúa si el vértice no está en Visitado y si no se alcanzó
      ➥la profundidad límite. */
      if (!Visitado.BuscaDesordenada(VertiX) && Indice < NivelProf)
      {
           Visitado.InsertaFinal(VertiX);
           /* Se obtienen sus vértices adyacentes. */
           ListaAux= VerticesAdyacentes(BuscaVertice(VertiX));
           while (!ListaAux.ListaVacia() &&  !EstadoFinal)
           {
              VertiX= ListaAux.Elimina();
              if (BuscaVertice(VertiX) != NumVer-1 &&
              ➥!VisitaAux[BuscaVertice(VertiX)])
              {
                  NoVisitado.InsertaInicio(VertiX);
                  VisitaAux[BuscaVertice(VertiX)]= 1;
              }
              /* Se evalúa si se llegó al último vértice (representa el
              ➥estado final). */
              else
                  if (BuscaVertice(VertiX) ==  NumVer-1)
```

```
                    {
                        Visitado.InsertaFinal(VertiX);
                        EstadoFinal= 1;
                    }
                }
                Indice++;
            }
        }
        /* Si se llegó al estado final se imprime la secuencia de vértices
        visitados. */
        if (EstadoFinal)
            Visitado.ImprimeIterativo();
        else
            Resp= 0;
        return Resp;
    }
```

Al aplicar este método a la figura 8.17, la secuencia de vértices visitados para llegar al estado final es: ***A – T – X – L – K – Z – S – N***. El vértice inicial es ***A***, por lo tanto es el primero del que se obtienen los vértices adyacentes (***T, Y***) los cuales se guardan en la lista NoVisitado y el vértice ***A*** se almacena en `Visitado`. Como se guardan al inicio de la lista, ahora se quita ***T,*** se obtienen sus adyacentes (***X, D, Y***) y se agregan al inicio de la lista `NoVisitado`, y ***T*** se almacena en Visitado. Se quita ***X*** y se generan sus adyacentes (***L***). Se continúa así hasta que se obtiene el vértice ***N***, que es el estado final. Cada vez que se quita un vértice de `NoVisitado` (luego de obtener sus adyacentes), se guarda en `Visitado`. Es importante señalar que cuando se generan los adyacentes de un vértice se evalúa si alguno de ellos es o no el estado final. Si lo es, la búsqueda termina con éxito, en caso contrario se agrega a la lista `NoVisitado` para ser expandido posteriormente.

Se presentan los dos métodos auxiliares usados por los algoritmos `Depth First` y `Breadth First`.

```
/* Método entero que determina si un vértice dado como parámetro es o
➥no un vértice de la digráfica. Regresa la posición en la que lo encuentra
➥o un negativo. */
template <class T>
int DiGrafica<T>::BuscaVertice(T VertiDato)
```

```
{
   int Indice= 0, Resp= -1;

   /* Busca el nombre del vértice dado en el arreglo que guarda los
   ➥nombres de todos los vértices de la gráfica. */
   while (Indice < NumVer && Vertices[Indice] != VertiDato)
      Indice++;
   if (Indice < NumVer)
        Resp= Indice;
   return Resp;
}

/* Método que genera una lista con los vértices adyacentes de un vértice
➥dado como parámetro. Recibe como parámetro el nombre de un vértice y da
➥como resultado una lista con sus vértices adyacentes. */
template <class T>
Lista<T> DiGrafica<T>::VerticesAdyacentes(int VertiDato)
{
   int Indice;
   Lista <T> Adyacentes;

   for (Indice= 0; Indice < NumVer; Indice++)
        if (MatAdy[VertiDato][Indice] != 0)
                  Adyacentes.InsertaFinal (Vertices[Indice]);
   return Adyacentes;
}
```

8.4.2 Búsqueda a lo ancho (`Breadth First`)

Este tipo de búsqueda consiste en visitar, en cada nivel, todos los vértices. Se empieza con el estado (nodo) inicial y se expanden todos sus vértices adyacentes, luego en cada uno de ellos se aplica el mismo criterio. Por lo tanto, en cada nivel se tienen todos los estados que pueden ser generados a partir de los nodos del nivel anterior. Luego de cada expansión se debe verificar si ya se alcanzó el estado final. Se continúa así hasta llegar al estado meta o hasta haber expandido todos los nodos. Debido al orden en el cual se van obteniendo los nodos recibe el nombre de *búsqueda a lo ancho*.

La figura 8.18 presenta un ejemplo de una gráfica en la cual se aplica este tipo de búsqueda para llegar al estado final. Las aristas gruesas indican todos los vértices expandidos durante la búsqueda de la solución. Las líneas punteadas

señalan la trayectoria desde el estado inicial al final. Como puede observarse, desde el estado inicial ***A*** se expanden todos sus nodos sucesores (***T***, ***Y***), lo mismo se hace en estos vértices, generando ***J***, ***M***, ***D*** y ***X***. A partir de éstos se obtienen los vértices ***S***, ***L*** y ***E***. En el siguiente paso, de ***S*** se llega a ***N*** que es el estado meta.

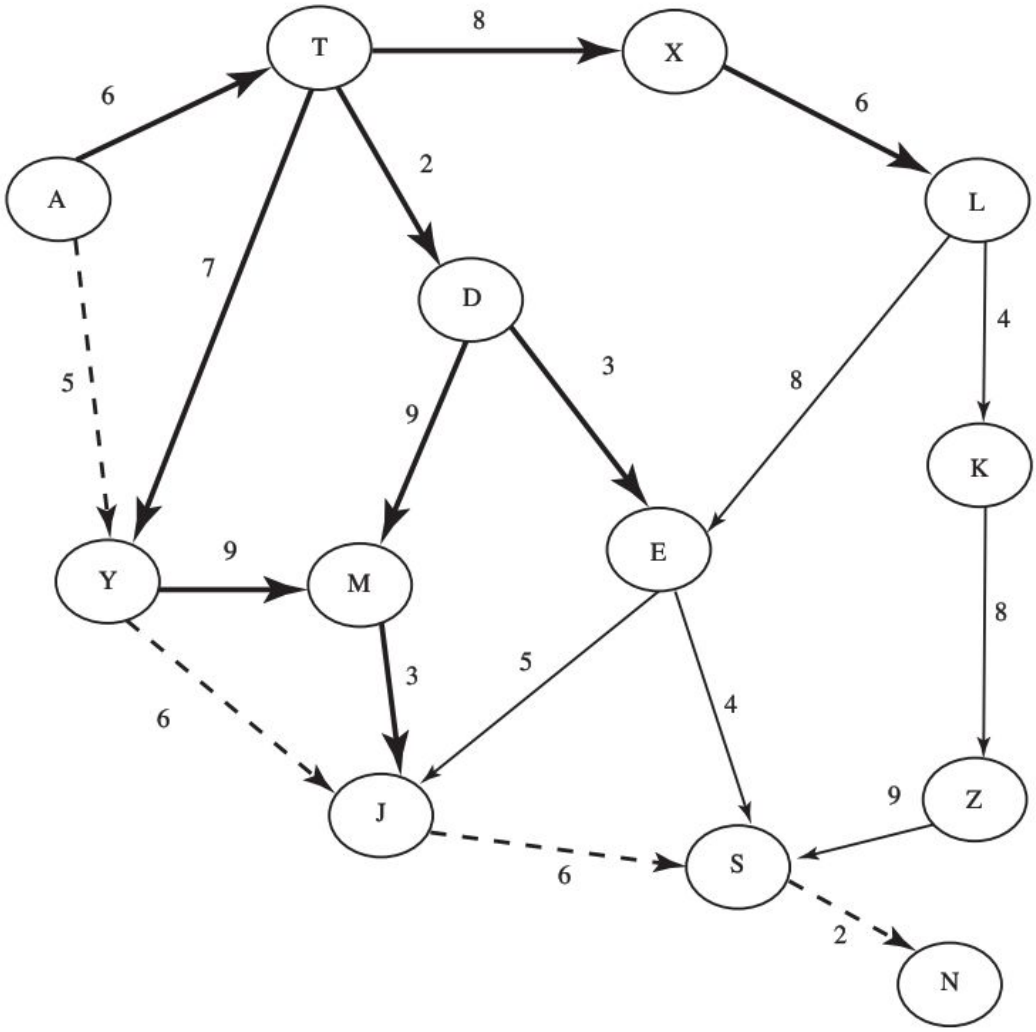


FIGURA 8.18 *Ejemplo de búsqueda a lo ancho*

Para implementar este método se requiere usar dos listas para ir almacenando los vértices visitados y aquellos pendientes por visitar. Los primeros se guardan en una lista llamada `Visitado` y los otros en `NoVisitado`. Los pasos principales de este método son:

1. Guardar el vértice inicial en la lista `NoVisitado`.
2. Sacar el primer elemento (vértice `VertiX`) de la lista `NoVisitado`.
3. Evaluar si `VertiX` está en la lista `Visitado`.
   - 3.1. Si la respuesta es negativa entonces obtener todos los vértices adyacentes de `VertiX` y guardarlos en `Visitado`.
     - 3.1.1. Si tiene vértices adyacentes y no son el estado final entonces guardarlos al ***final*** de la lista `NoVisitado`.
     - 3.1.2. Si no tiene vértices adyacentes, ir al paso 4.
   - 3.2. Si la respuesta es afirmativa, ir al paso 4.
4. Repetir los pasos 2 y 3 hasta que se llegue al estado final o hasta que la lista `NoVisitado` quede vacía.

A continuación se presenta el método `Breadth-First` de la clase `DiGrafica`. Este método se auxilia de la clase `Lista` (ver capítulo 6) para definir los objetos `Visitado` y `NoVisitado`. Como en el caso de la búsqueda en profundidad, se utilizan los métodos auxiliares ya estudiados para determinar si un vértice pertenece o no a una digráfica y para generar los vértices adyacentes de uno dado.

```
/* Este método busca una solución (estado final) de un problema
➥representado por medio de una gráfica. Visita todos los vértices de un
➥mismo nivel antes de pasar al siguiente. Regresa uno si llega al estado
➥meta o cero en caso contrario. Se usan atributos de la clase como el
➥número y nombre de los vértices. Además, se declaran tres objetos de
➥la clase Lista para almacenar los vértices visitados, los pendientes de
➥visitar y los adyacentes de un nodo dado. */
template <class T>
int DiGrafica<T>::BreadthFirst()
{
   int Indice, EstadoFinal= 0, VisitaAux[MAX], Resp= 1;
   Lista<T> NoVisitado, Visitado, ListaAux;
   T VertiX;
```

```
/* El arreglo VisitaAux es un arreglo en el cual se indica si un nodo
➥ya fue expandido. */
for (Indice= 0; Indice < NumVer; Indice++)
     VisitaAux[Indice]= 0;

/* Se guarda el primer vértice de la gráfica en la lista NoVisitado. */
NoVisitado.InsertaFinal(Vertices[0]);
VisitaAux[0]= 1;

/* Ciclo que se ejecuta mientras no se llegue al estado final y
➥queden vértices por visitar. */
while (!NoVisitado.ListaVacia() &&  !EstadoFinal)
{
   /* Saca el primer vértice de la lista NoVisitado. */
   VertiX= NoVisitado.Elimina();
   /* Se evalúa que el vértice no esté en la lista Visitado para
   ➥evitar ciclos. */
   if (!Visitado.BuscaDesordenada(VertiX))
   {
        Visitado.InsertaFinal(VertiX);
        /* Se obtienen los vértices adyacentes del vértice visitado. */
        ListaAux= VerticesAdyacentes(BuscaVertice(VertiX));

        while (!ListaAux.ListaVacia() &&  !EstadoFinal)
        {
           VertiX= ListaAux.Elimina();
           /* Si el sucesor no es el estado final y no está en
           ➥Visitado entonces se guarda en la lista NoVisitado para
           ➥que posteriormente se revise. */
           if (BuscaVertice(VertiX) != NumVer-1 &&
           ➥!VisitaAux[BuscaVertice(VertiX)])
           {
               NoVisitado.InsertaFinal(VertiX);
               VisitaAux[BuscaVertice(VertiX)]= 1;
           }
           else
               if (BuscaVertice(VertiX) ==  NumVer - 1)
               {
                   Visitado.InsertaFinal(VertiX);
                   EstadoFinal= 1;
               }
        }
   }
}
```

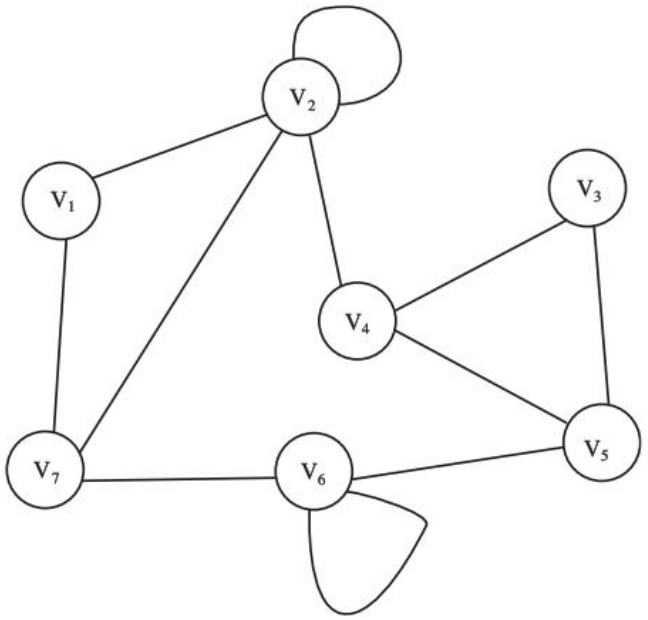
```
    /* Si se llegó al estado final se imprime la secuencia de vértices
    ➥visitados. */
    if (EstadoFinal)
    {
        Visitado.ImprimeIterativo();
        return 1;
    }
    else
        Resp= 0;
    return Resp;
}
```

Al aplicar este método a la figura 8.18, la secuencia de vértices visitados para llegar al estado final es: ***A – Y – T – J – M – D – X – S – N***. El vértice inicial es ***A***, por lo tanto es el primero al cual se le obtienen los vértices adyacentes (***Y, T***) los cuales se guardan al final de `NoVisitado` y el vértice ***A*** se almacena en `Visitado`. Luego se quita ***Y*** y se obtienen sus adyacentes (***J, M***) los cuales se agregan al final de la lista `NoVisitado` y ***Y*** en la lista `Visitado`. Se quita ***T*** y se generan sus adyacentes (***Y, D, X***), éstos se guardan en `NoVisitado` y ***T*** en `Visitado`. El siguiente vértice que se extrae de `NoVisitado` es ***J*** y se obtiene su vértice adyacente que es ***S***. Luego se quita ***M*** cuyo vértice adyacente es ***J***. Después se quita ***D*** de la lista `NoVisitado` y se obtienen sus adyacentes (***M, E***) los cuales se guardan al final de `NoVisitado` y ***D*** en `Visitado`. Así se continúa hasta que se quita ***S***, a partir del cual se obtiene ***N*** que es el estado final. Es importante mencionar que cuando se extrae de `NoVisitado` un vértice que ya está en `Visitado` no se vuelve a agregar a esta lista. Además, cuando se expande un nodo, si alguno de sus adyacentes ya fue generado, entonces no se agrega a la lista de `NoVisitado`.

Los métodos presentados son muy parecidos. Conceptualmente la diferencia está en que el primero desarrolla una rama de la gráfica hasta llegar al final o a un nivel límite, mientras que el segundo va desarrollando todas las ramas hasta alcanzar el estado meta. En cuanto a la implementación, la diferencia mencionada se logra guardando los siguientes vértices que deben ser expandidos al **inicio** o al **final** respectivamente de la lista de vértices `NoVisitado`.

Ejercicios

1. Dada la siguiente gráfica, señale:
   (a) Un camino entre los vértices V_1 y V_5, si es posible.
   (b) Un camino simple entre cada par de vértices, si es posible.
   (c) El grado de cada vértice.
   (d) Lazos o bucles, si existen.



2. Dada la siguiente matriz de adyacencias etiquetada, dibuje la gráfica dirigida correspondiente. El 0 en la posición (i, j) indica que no existe un arco entre los vértices V_i y V_j, incluyendo a la diagonal principal.

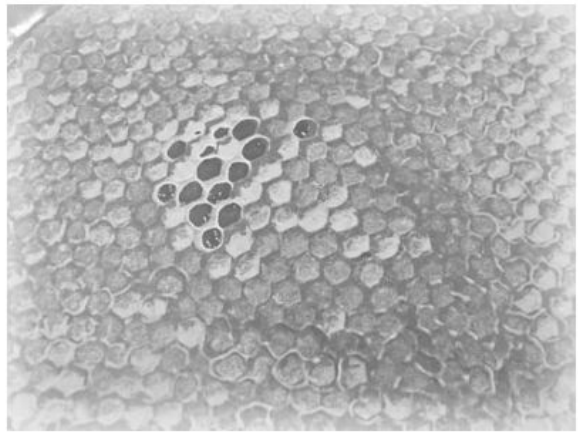
	1	**2**	**3**	**4**	**5**	**6**
1	0	15	0	12	6	0
2	0	0	8	7	0	0
3	18	0	0	21	0	16
4	0	11	0	0	10	12
5	0	13	7	0	0	9
6	14	0	18	0	0	0

3. Retome el problema anterior. Aplique el método que crea adecuado para obtener e imprimir los caminos de mínimos costos entre el vértice 3 y los demás vértices de la digráfica.
4. Aplique el método que crea conveniente a la gráfica del problema 2, para generar una matriz que indique si existe o no un camino entre cada uno de los vértices de la gráfica dirigida.
5. Dada la siguiente matriz de adyacencias etiquetada, dibuje la gráfica no dirigida correspondiente. El 0 en la posición (i, j) indica que no existe un arco entre los vértices V_i y V_j, incluyendo a la diagonal principal.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	0	53	67	88	21	0
2	0	0	29	0	0	84	19
3	53	29	0	28	62	0	0
4	67	0	28	0	41	55	0
5	88	0	62	41	0	18	34
6	21	84	0	55	18	0	87
7	0	19	0	0	34	87	0

6. Retome la gráfica del problema anterior. Aplique el método que crea conveniente para encontrar e imprimir el árbol abarcador de costo mínimo correspondiente a dicha gráfica.
7. Modifique la plantilla de la clase `Grafica` usando un arreglo unidimensional para almacenar sólo la matriz triangular superior de la matriz de adyacencia —recuerde que la matriz de adyacencia de una gráfica es una matriz simétrica—. Puede utilizar las fórmulas vistas en el capítulo 4 para recuperar los elementos. ¿Requiere adaptar los métodos de la clase?
8. Modifique la plantilla de la clase `DiGrafica` de tal manera que use una lista de adyacencia en lugar de una matriz de adyacencia para almacenar la información de la gráfica dirigida. Realice los ajustes necesarios en los métodos estudiados para que puedan aplicarse a esta nueva estructura.
9. Modifique la plantilla de la clase `Grafica` de tal manera que use una lista de adyacencia en lugar de una matriz de adyacencia para almacenar la información de la gráfica no dirigida. Realice los ajustes necesarios en los métodos estudiados para que puedan aplicarse a esta nueva estructura.

10. Escriba un método que determine si una gráfica es una *gráfica completa*, para ello deberá verificar si cada uno de sus vértices es adyacente a los demás.
11. Escriba un método que determine si una gráfica es una *gráfica conexa*, para ello deberá verificar si existe un camino simple entre cada uno de sus vértices. ¿Le sirve alguno de los métodos analizados en este capítulo?
12. Escriba un método que encuentre e imprima todos los caminos simples que existan en una gráfica dirigida. El método debe imprimir el identificador de cada uno de los vértices involucrados en los caminos.
13. En la orilla de un río están tres misioneros y tres caníbales con intención de cruzar a la otra orilla. Cuentan con un bote que tiene una capacidad límite de dos personas. Los misioneros, para poder protegerse de los caníbales, quieren estar siempre en un número mayor o igual al de caníbales. Usando algunos de los métodos vistos, resuelva el problema de trasladar a los seis individuos de una orilla a la otra sin poner en riesgo a los misioneros.



CAPÍTULO 9

Ordenación

9.1 Introducción

La **ordenación** es la operación que permite establecer un orden (creciente o decreciente) entre un conjunto de valores. Dependiendo dónde estén almacenados los datos, la ordenación recibe diferentes nombres. Si se realiza sobre datos guardados en un arreglo se le llama ***ordenación interna***. Por otra parte, si se aplica a un conjunto de valores almacenados en un archivo, se le denomina ***ordenación externa***.

Si bien la ordenación no es una estructura de datos, se presenta en este libro porque es una de las operaciones más importantes a realizar sobre los datos guardados en una estructura. Ordenar la información almacenada en la estructura permite recuperarla en menos tiempo. Es decir, la búsqueda (tema que se verá con mayor detalle en el siguiente capítulo) resulta más eficiente cuando los datos están ordenados.

Normalmente esta operación se encuentra implementada como un método de otras clases, como la clase `Arreglo` o la clase `Lista`. Sin embargo, dado que es el tema central de este capítulo, se tratará como una clase que representa a las variantes más conocidas del proceso de ordenación.

9.2 Ordenación interna

La **ordenación interna** se refiere a ordenar un conjunto de datos que se encuentran almacenados en una estructura, en memoria principal. Considerando las características que determinan la manera en la que se tiene acceso a los elementos de una estructura, en este libro se estudiarán los métodos para ordenar a los arreglos unidimensionales. El resultado de aplicar esta operación a un arreglo es que todos sus elementos quedan ordenados de manera creciente o decreciente.

- **Creciente**: $dato_1 \leq dato_2 \leq ... \leq dato_n$ (el primer dato es menor o igual que el segundo, éste es menor o igual que el tercero y así sucesivamente hasta el último dato).
- **Decreciente**: $dato_1 \geq dato_2 \geq ... \geq dato_n$ (el primer dato es mayor o igual que el segundo, éste es mayor o igual que el tercero y así sucesivamente hasta el último dato).

Existen numerosos métodos que ordenan a los elementos de un arreglo. Los métodos pueden agruparse según la característica principal (intercambio, inserción o selección) de la operación que realizan para ordenar los valores. Los más conocidos y utilizados son:

TABLA 9.1 Métodos de ordenación

Métodos de ordenación por intercambio	*Métodos de ordenación por selección*	*Métodos de ordenación por inserción*
Directo con desplazamiento hacia la izquierda	Directa	Directa
Directo con desplazamiento hacia la derecha		Binaria
Shaker (sacudida)		Shell
Con señal		
QuickSort		

Para programar los métodos de ordenación en el lenguaje *C++* se definió una clase base abstracta y un conjunto de clases derivadas. Cada una de las clases derivadas representa uno de los métodos que se estudiarán en este capítulo. La figura 9.1 presenta un esquema de las clases mencionadas.

Todos los métodos utilizan dos operaciones básicas para llevar a cabo la ordenación de los elementos de un arreglo: la comparación y el movimiento o intercambio de los mismos. Por esta razón, en la clase abstracta se incluyó el método `Intercambia()` que será común a todas las clases derivadas y que tendrá por objetivo intercambiar los valores de dos posiciones del arreglo.

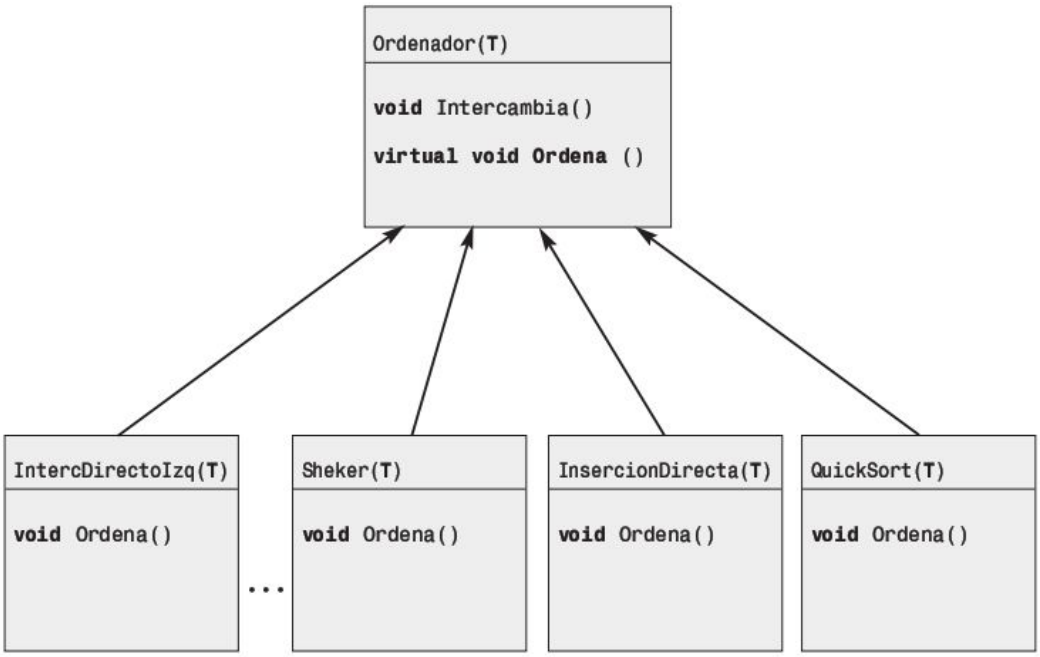


FIGURA 9.1 *Esquema de clases*

A continuación se muestra la manera de programar la clase base abstracta, la cual tiene un método virtual puro que se redefinirá en cada subclase dependiendo del método de ordenación que se esté implementando. Además, tiene un método auxiliar —`Intercambia()`— para generalizar la operación de intercambio que será usada por las subclases.

```
/* Clase abstracta que se utiliza para definir clases derivadas que re-
➥presentan cada uno de los métodos de ordenación interna. */
template <class T>
class Ordenador
{
   public:
       void Intercambia (int, int, Arreglo<T> *);
       virtual void Ordena (Arreglo<T> *) = 0;
};

/* Método auxiliar que intercambia los contenidos de dos elementos del
➥arreglo que se está ordenando. */
template <class T>
void Ordenador<T>::Intercambia(int Ind1, int Ind2, Arreglo<T> *Arre)
{
   T Auxiliar;
   Auxiliar= Arre->RegresaValor(Ind1);
   Arre->AsignaValor(Ind1, Arre->RegresaValor(Ind2));
   Arre->AsignaValor(Ind2, Auxiliar);
}
```

Como ya se mencionó, todos los métodos utilizan dos operaciones básicas para llevar a cabo la ordenación de los elementos: la comparación y el movimiento de los mismos. Por lo tanto, si lo que se quiere ordenar son objetos hay que tener en cuenta que se deben sobrecargar los operadores de comparación en las clases correspondientes, para hacer uso de los métodos que se presentan en las siguientes secciones.

9.2.1 Métodos de ordenación por intercambio

Estos métodos son de los más sencillos y por lo tanto más usados para ordenar un conjunto pequeño de datos. Se caracterizan porque se intercambian los valores como resultado de la comparación de los mismos. Existen varios métodos que se basan en esta idea. Los más conocidos son:

- Intercambio directo:
    - Con desplazamiento hacia la izquierda
    - Con desplazamiento hacia derecha
- Intercambio con señal
- Sheker
- Quicksort

Intercambio directo con desplazamiento hacia la izquierda

El método de intercambio directo consiste en recorrer el arreglo comparando pares de datos e intercambiándolos de tal manera que los valores pequeños se vayan desplazando hacia la izquierda o bien, los valores más grandes se vayan desplazando hacia la derecha. Esta característica genera dos versiones de este algoritmo, dando origen a sendas clases, las cuales, en este libro, se denominan `IntercDirectoIzq` e `IntercDirectoDer` respectivamente.

Para ordenar `Tam` elementos (donde `Tam` es el número de elementos del arreglo) se realizan `Tam-1` recorridos por el arreglo comparando pares de datos. Luego de cada comparación puede o no realizarse un intercambio del contenido entre dos casillas del arreglo.

A continuación se presenta la clase `IntercDirectoIzq`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por intercambio directo con desplazamiento del valor más pequeño hacia la izquierda.

```
/* Clase para el método de intercambio directo con desplazamiento hacia
➥la izquierda. Clase derivada de la clase abstracta Ordenador. */
template <class T>
class IntercDirectoIzq: public Ordenador<T>
{
     public:
            void Ordena (Arreglo<T> *);
};

/* Método que ordena los elementos de un arreglo. Por medio de
➥comparaciones e intercambios de elementos lleva el elemento más
➥pequeño hacia el extremo izquierdo del arreglo. Este proceso se
➥repite hasta que todo el arreglo queda ordenado.*/
template <class T>
void IntercDirectoIzq<T>::Ordena(Arreglo<T> *Arre)
{
   int Ind1, Ind2, Tam= Arre->RegresaTam();
   for (Ind1= 1; Ind1 < Tam; Ind1++)
        for (Ind2= Tam-1; Ind2 >= Ind1; Ind2--)
            if (Arre->RegresaValor(Ind2-1) > Arre->RegresaValor(Ind2))
                Intercambia(Ind2-1, Ind2, Arre);
}
```

Considere un arreglo de 6 elementos (`Tam = 6`) como el que se muestra en la figura 9.2. Aplicando el método visto para ordenar este arreglo, se tendría la secuencia de pasos presentada en la tabla 9.2. Las casillas que se recuadran son

aquellas cuyos elementos se intercambian y las casillas que se van sombreando son las que luego del ciclo interno (`Ind2`) quedan ordenadas. Observe que como este método desplaza el valor más pequeño hacia la izquierda, es el extremo izquierdo del arreglo el que va quedando ordenado. La primera vez se coloca el 4 en la posición 0, la segunda vez el 9 en la posición 1 y así sucesivamente hasta el último valor. Como consecuencia, en cada iteración el intervalo en el cual se ordenan valores se reduce. Es decir, como la posición 0 queda ocupada por el valor más pequeño, en el siguiente ciclo la ordenación llega hasta la posición 1, y en la siguiente hasta la posición 2 y así hasta la posición tamaño del arreglo menos 1.

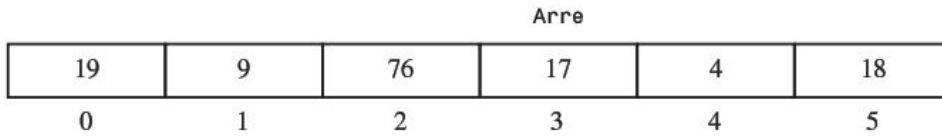


FIGURA 9.2 *Arreglo a ordenar*

TABLA 9.2 Seguimiento del método de ordenación por intercambio directo con desplazamiento hacia la izquierda

Ind1	Ind2	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
1	5	19	9	76	17	4	18
	4	19	9	76	4	17	18
	3	19	9	4	76	17	18
	2	19	4	9	76	17	18
	1	4	19	9	76	17	18
2	5	4	19	9	76	17	18
	4	4	19	9	17	76	18
	3	4	19	9	17	76	18
	2	4	9	19	17	76	18
3	5	4	9	19	17	18	76
	4	4	9	19	17	18	76
	3	4	9	17	19	18	76
4	5	4	9	17	19	18	76
	4	4	9	17	18	19	76
5	5	4	9	17	18	19	76

Intercambio directo con desplazamiento hacia la derecha

A continuación se presenta la clase `IntercDirectoDer`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por intercambio directo con desplazamiento del valor más grande hacia la derecha.


```
/* Clase para el método de intercambio directo con desplazamiento hacia
la ➥derecha. Clase derivada de la clase Ordenador. */
template <class T>
class IntercDirectoDer: public Ordenador<T>
{
   public:
      void Ordena (Arreglo<T> *);
};

/* Método que ordena los elementos de un arreglo. Por medio de compa-
➥raciones e intercambios de elementos lleva el elemento más grande hacia
➥el extremo derecho del arreglo. Este proceso se repite hasta que todo el
➥arreglo queda ordenado. */
template <class T>
void IntercDirectoDer<T>::Ordena(Arreglo<T> *Arre)
{
   int Ind1, Ind2, Tam= Arre->RegresaTam();
   for (Ind1= 0; Ind1 < Tam-1; Ind1++)
        for (Ind2= 0; Ind2 < Tam-1-Ind1; Ind2++)
            if (Arre->RegresaValor(Ind2) > Arre->RegresaValor(Ind2+1))
                Intercambia(Ind2, Ind2+1, Arre);
}
```

Se retoma el arreglo de la figura 9.2 y se utiliza el último método visto para ordenarlo. La tabla 9.3 presenta la secuencia de operaciones aplicadas. Las casillas que se recuadran son aquellas cuyos elementos se intercambian y las casillas que se van sombreando son las que luego del ciclo interno (`Ind2`) quedan ordenadas. Observe que como este método desplaza el valor más grande hacia la derecha, es el extremo derecho del arreglo el que va quedando ordenado. La primera vez se coloca el 76 en la posición 5, la segunda el 19 en la posición 4 y así sucesivamente hasta el último valor. Como consecuencia, en cada iteración el intervalo en el cual se ordenan valores se reduce (en el algoritmo esto se logra restando `Ind1` al límite superior del segundo ciclo). Es decir, como la posición `Tam-1` queda ocupada por el valor más grande, en el siguiente ciclo la ordenación llega hasta la posición `Tam-2`, y en la siguiente hasta la posición `Tam-3` y así hasta que la última vez se ordenan sólo dos casillas, la 0 y la 1.

TABLA 9.3 Seguimiento del método de ordenación por intercambio directo con desplazamiento hacia la derecha

Ind1	Ind2	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
0	0	9	19	76	17	4	18
	1	9	19	76	17	4	18
	2	9	19	17	76	4	18
	3	9	19	17	4	76	18
	4	9	19	17	4	18	76
1	0	9	19	17	4	18	76
	1	9	17	19	4	18	76
	2	9	17	4	19	18	76
	3	9	17	4	18	19	76
2	0	9	17	4	18	19	76
	1	9	4	17	18	19	76
	2	9	4	17	18	19	76
3	0	4	9	17	18	19	76
	1	4	9	17	18	19	76
4	0	4	9	17	18	19	76

Las dos variantes vistas del método de ordenación por intercambio directo tienen la misma eficiencia. Ésta se mide por el número de ***comparaciones*** y de ***intercambios*** realizados. En cuanto al número de comparaciones, en el primer recorrido se realizan (`Tam -1`), en el segundo (`Tam - 2`) comparaciones y así sucesivamente hasta hacer una comparación (cuando sea el último par de datos a ordenar). Por lo tanto, el total de comparaciones se puede expresar como:

$$\text{Total comparaciones} = (Tam - 1) + (Tam - 2) + (Tam - 3) + \ldots + 1 = \frac{Tam * (Tam-1)}{2}$$

lo cual puede escribirse como:

$$\text{Total comparaciones} = \frac{Tam^2 - Tam}{2}$$

FÓRMULA 9.1

Con respecto al número de intercambios, los mismos dependen del estado del arreglo, es decir, si ya está ordenado, si está ordenado en orden inverso o si está desordenado. El total de intercambios se expresa de la siguiente manera:

Intercambio mínimo = 0 si el arreglo ya está ordenado

Intercambio máximo = (Tam² - Tam) * 1.5 si el arreglo está en orden inverso

Intercambio medio = (Tam² - Tam) * 0.75 si el arreglo está desordenado

FÓRMULA 9.2

Algoritmo de `Sheker` o de sacudida

El algoritmo conocido con el nombre de `Sheker` o de sacudida es una combinación de los dos anteriores. Cada recorrido del arreglo se divide en dos etapas, en la primera se mueven los elementos más pequeños hacia la izquierda y en la segunda, los elementos más grandes hacia la derecha. En cada etapa se guarda la posición donde se realizó el intercambio, y de esta manera en el siguiente recorrido del arreglo el intervalo se reduce entre estas dos posiciones. El proceso termina cuando no se producen intercambios o bien, cuando la posición del extremo izquierdo es mayor que la del extremo derecho.

A continuación se presenta la clase `Sheker`, en la cual el método `Ordena` corresponde al algoritmo de `Sheker` o de sacudida.

```
/* Clase para el método de Sheker o de sacudida. Clase derivada de la
➥clase abstracta Ordenador. */
template <class T>
class Sheker: public Ordenador<T>
{
   public:
       void Ordena (Arreglo<T> *);
};

/* Este método ordena los elementos de un arreglo utilizando el
➥algoritmo de Sheker. */
template <class T>
void Sheker<T>::Ordena(Arreglo<T> *Arre)
```

```
{
   int Indice, Izq= 1, Tam= Arre->RegresaTam(), Der= Tam-1,
   ➥Extremo= Tam-1;
   while (Izq <= Der)
   {
       for (Indice= Der; Indice >= Izq; Indice--)
           if (Arre->RegresaValor(Indice-1) > Arre->RegresaValor(Indice))
           {
               Intercambia(Indice-1, Indice, Arre);
               Extremo= Indice;
           }
       Izq= Extremo+1;
       for (Indice= Izq; Indice <= Der; Indice++)
           if (Arre->RegresaValor(Indice-1) > Arre->RegresaValor(Indice))
           {
              Intercambia(Indice-1, Indice, Arre);
              Extremo= Indice;
           }
       Der= Extremo-1;
   }
}
```

Se retoma el arreglo de la figura 9.2 y se utiliza el último método visto para ordenarlo. La tabla 9.4 presenta la secuencia de operaciones aplicadas. Las casillas que se recuadran son aquellas cuyos elementos se intercambian y las casillas que se sombrean son aquellas que quedan ordenadas. Observe que el intervalo donde se lleva a cabo la ordenación comprende del 1 al 5 (considerando que usa la posición (`Indice-1`)). Con cada intercambio se guarda la posición de la casilla correspondiente y al terminar el primer ciclo se redefine el extremo izquierdo con la posición más 1. Al ejecutar el segundo ciclo, el intervalo es menor. Nuevamente con cada intercambio se guarda la posición y al terminar el ciclo se redefine el extremo derecho como la posición menos 1. En el ejemplo presentado, al terminar la segunda ejecución del segundo ciclo el extremo derecho queda en 3 siendo menor que el extremo izquierdo (que tiene el valor de 4) y de esta forma se interrumpe el ciclo `While` y concluye la ordenación.

TABLA 9.4 Seguimiento del método de ordenación `Sheker`

Índice	Extremo	Izq	Der	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
	5	1	5	19	9	76	17	4	18
5				19	9	76	17	4	18
4	4			19	9	76	4	17	18
3	3			19	9	4	76	17	18
2	2			19	4	9	76	17	18
1	1			4	19	9	76	17	18
		2							
2	2			4	9	19	76	17	18
3				4	9	19	76	17	18
4	4			4	9	19	17	76	18
5	5			4	9	19	17	18	76
			4						
4				4	9	19	17	18	76
3	3			4	9	17	19	18	76
2				4	9	17	19	18	76
		4							
4	4			4	9	17	18	19	76
			3						

Intercambio directo con señal

Este algoritmo es otra variante de la ordenación por intercambio directo y busca ganar eficiencia en cuanto al número de comparaciones realizadas. Para ello se apoya en una variable auxiliar (la señal) que permite determinar en cada recorrido si se produjo algún intercambio. Si lo hubo, entonces se sigue recorriendo el arreglo; mientras que, en caso contrario, la ordenación termina, habiendo quedado ordenado todo el arreglo. Es decir, cuando en un recorrido no se hacen intercambios esta situación se detecta por medio de la variable auxiliar o bandera y se evitan todas las comparaciones pendientes de acuerdo a los límites de los ciclos.

A continuación se presenta la clase `IntercConSenial`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por intercambio directo con señal.

```
/* Clase para el método de intercambio directo con señal. Clase derivada
➥de la clase abstracta Ordenador. */
template <class T>
class IntercConSenial: public Ordenador<T>
{
   public:
       void Ordena (Arreglo<T> *);
};


/* Este método ordena los elementos del arreglo utilizando el algoritmo
➥de intercambio directo con señal. */
template <class T>
void IntercConSenial<T>::Ordena(Arreglo<T> *Arre)
{
   int Ind1= 0, Ind2, Bandera= 0, Tam= Arre->RegresaTam();
   while ((Ind1 < Tam-1) && (!Bandera))
   {
       Bandera= 1;
       for (Ind2= 0; Ind2 < Tam-1; Ind2++)
           if (Arre->RegresaValor(Ind2) > Arre->RegresaValor(Ind2+1))
           {
               Intercambia(Ind2, Ind2+1, Arre);
               Bandera= 0;
           }
       Ind1++;
   }
}
```

Considere el arreglo de la figura 9.3, se utiliza el algoritmo de intercambio con señal para ordenarlo. La tabla 9.5 presenta la secuencia de operaciones aplicadas. Las casillas que se recuadran son aquellas cuyos elementos se intercambian y las casillas que se sombrean son aquellas que quedan ordenadas. Cuando `Ind1` igual a 2 el arreglo queda ordenado, sin embargo como `Bandera` está en 0 (por el intercambio entre el 20 y el 24) se ejecuta el ciclo para `Ind1` igual 3.

Arre

15	4	59	24	51	20
0	1	2	3	4	5

FIGURA 9.3 *Arreglo a ordenar*

TABLA 9.5 Seguimiento del método de ordenación por intercambio con señal

Bandera	Ind1	Ind2	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
0	0		15	4	59	24	51	20
1		0	4	15	59	24	51	20
0		1	4	15	59	24	51	20
		2	4	15	24	59	51	20
0		3	4	15	24	51	59	20
0		4	4	15	24	51	20	59
	1							
1		0	4	15	24	51	20	59
		1	4	15	24	51	20	59
		2	4	15	24	51	20	59
		3	4	15	24	20	51	59
0		4	4	15	24	20	51	59
	2							
1		0	4	15	24	20	51	59
		1	4	15	24	20	51	59
		2	4	15	20	24	51	59
0		3	4	15	20	24	51	59
		4	4	15	20	24	51	59
	3							
1		0	4	15	20	24	51	59
		1	4	15	20	24	51	59
		2	4	15	20	24	51	59
		3	4	15	20	24	51	59
		4	4	15	20	24	51	59

Método rápido o `QuickSort`

Este método es otra de las variantes del método de intercambio directo, que se caracteriza por ser la más rápida en memoria interna. Fue propuesto por C. Hoare en 1962. Básicamente consiste en encontrar la posición de un cierto elemento del

arreglo, al que llamaremos `Pivote`, de tal manera que todos los valores que se encuentren a su izquierda sean menores o iguales a él y los que se encuentren a su derecha sean mayores o iguales a él. Este proceso se repite para cada uno de los valores que queden a la izquierda y a la derecha del pivote.

A continuación se presenta la clase `QuickSort`, en la cual el método `Ordena` corresponde al algoritmo de ordenación conocido como Quick Sort o rápido. En este caso, el método `Ordena` utiliza un método auxiliar recursivo, `Reduce()`, que es el que lleva a cabo la ordenación del arreglo. Este método recibe los extremos del intervalo en el cual se colocará a un elemento en la posición que le corresponda. La primera vez dichos extremos coinciden con los del arreglo. Luego de colocar el primer elemento en la posición que le corresponde por su tamaño, se tienen dos intervalos, uno a la izquierda y otro a la derecha del mismo, cada uno de ellos con un tamaño menor al anterior. De ahí el nombre de `Reduce()`. Este proceso se repite hasta que ya no queden elementos a ordenar.

```
/* Clase para el algoritmo de ordenación llamado QuickSort. Clase
➥derivada de la clase abstracta Ordenador. */
template <class T>
class QuickSort: public Ordenador<T>
{
   public:
       void Ordena (Arreglo<T> *);
       void Reduce (int, int, Arreglo<T> *);
};

/* Este método ordena los elementos del arreglo utilizando el algoritmo
➥QuickSort. */
template <class T>
void QuickSort<T>::Ordena(Arreglo<T> *Arre)
{
   int Tam;
   Tam= Arre->RegresaTam();
   if (Tam > 0)
       Reduce (0, Tam - 1, Arre);
}

/* Método auxiliar de la clase QuickSort. Los parámetros Inicio y Fin
➥representan los extremos del intervalo (dentro del arreglo) en el cual
➥se está ordenando. La primera vez son el primero y último índice del
➥arreglo a ordenar. */
template <class T>
void QuickSort<T>::Reduce(int Inicio, int Fin, Arreglo<T> *Arre)
```

```
{
        int Izq, Der, Pivote, Bandera;
        Izq= Inicio;
        Der= Fin;
        Pivote= Inicio;
        Bandera= 1;
        while (Bandera)
        {
           Bandera= 0;
           while ((Arre->RegresaValor(Pivote) <= Arre->RegresaValor(Der))
                 && (Pivote != Der))
                 Der--;
           if (Pivote != Der)
           {
               Intercambia(Pivote, Der, Arre);
               Pivote= Der;
               while ((Arre->RegresaValor(Pivote)=>= Arre->RegresaValor
                     (Izq)) && (Pivote. != Izq))
                  Izq++;
               if (Pivote != Izq)
               {
                  Bandera= 1;
                  Intercambia(Pivote, Izq, Arre);
                  Pivote= Izq;
               }
           }
        }
        if ((Pivote - 1) > Inicio)
             Reduce(Inicio, Pivote - 1, Arre);
        if (Fin > (Pivote + 1))
             Reduce(Pivote + 1, Fin, Arre);
}
```

El arreglo de la figura 9.4 se ordenará utilizando la clase `QuickSort`. La tabla 9.6 presenta la secuencia de operaciones aplicadas. Se sombrean las casillas cuyos contenidos se intercambian, y se recuadra al pivote una vez guardado en la posición que le corresponde.

Arre

23	41	63	17	8	50	12
0	1	2	3	4	5	6

FIGURA 9.4 *Arreglo a ordenar*

TABLA 9.6 Seguimiento del método de ordenación `QuickSort`

Inicio	Fin	Izq	Der	Band	Pivote	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]	Arre[6]
0	6	0	6	1	0	23	41	63	17	8	50	12
				0	6	12	41	63	17	8	50	23
		1		1	1	12	23	63	17	8	50	41
			5									
			4		4	12	8	63	17	23	50	41
		2		1	2	12	8	23	17	63	50	41
		3	3	0	3	12	8	17	23	63	50	41
0	2	0	2	1	0	12	8	17	23	63	50	41
		1	1	0	1	8	12	17	23	63	50	41
4	6	4	6	1	4	8	12	17	23	63	50	41
		5		0	6	8	12	17	23	41	50	63
		6										
4	5	4	5	1	4	8	12	17	23	41	50	63
			4	0								

En la tabla se puede observar que, la primera vez, el pivote tiene valor 23 y queda almacenado en la posición 3. Esto origina que queden dos intervalos a ordenar, uno a su izquierda comprendido entre las posiciones 0 y 2, y otro a su derecha entre las posiciones 4 y 6. El método se invoca con el primer par de datos y queda pendiente la ejecución con el segundo par (queda en la pila interna que maneja la recursión). La segunda vez se acomoda el valor 12 en la posición 1. Como no existen intervalos a ordenar a partir de este valor, se sacan de la pila los valores 4 y 6 y se usan como extremos del nuevo intervalo a ordenar. La tercera vez se acomoda el 63 en la posición 6. A la derecha del mismo no quedan elementos a ordenar, sin embargo, a su izquierda sí. Por lo tanto se invoca al método con los valores 4 y 5, aunque en este caso no hay cambios.

La figura 9.5 (a) muestra cómo queda dividido el intervalo original [0,6] una vez que se coloca el pivote *23* en la posición correcta, en dos subintervalos: de 0 a 2, a la izquierda del pivote y de 4 a 6 a su derecha. Por otro lado, en la figura 9.5 (b), luego de colocar al pivote *12* en la posición que le corresponde, el primer subintervalo vuelve a reducirse, esta vez de tal manera que ya no requiere ordenación.

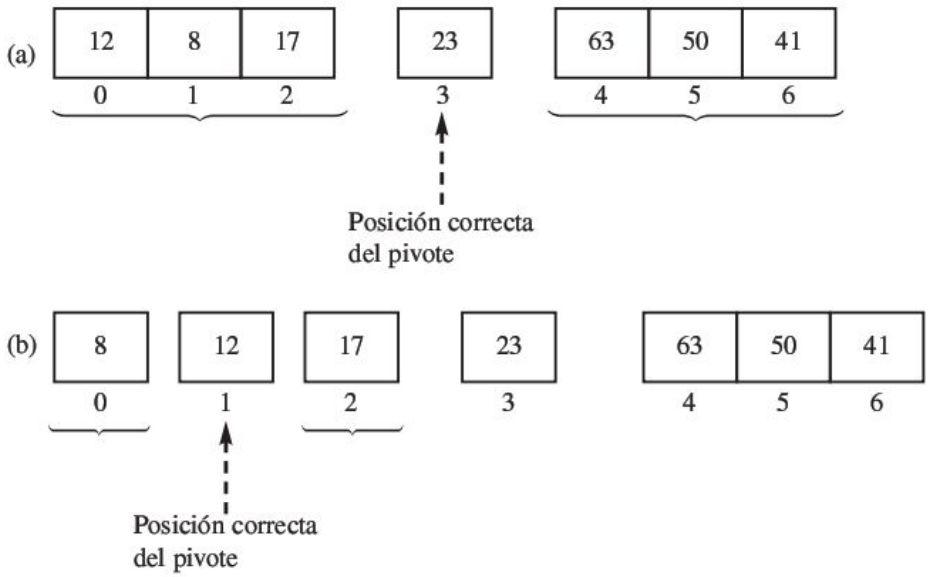


FIGURA 9.5 *Acomodo del pivote*
(a) Pivote = 23 y (b) Pivote = 12

Es posible plantear algunas variantes de este método, aplicando distintos criterios para elegir el pivote. Intente buscar y probar otras alternativas para la selección de este elemento.

Este algoritmo es el más rápido de los conocidos hasta el momento. Sin embargo, si el arreglo ya estuviera ordenado o estuviera ordenado en orden inverso se perdería gran parte de la eficiencia del mismo.

En cuanto al tiempo de ejecución, en el mejor o en la mayoría de los casos, queda determinado por la expresión presentada en la fórmula 9.3. En el peor de los casos (por ejemplo si el arreglo ya está ordenado), con la expresión de la fórmula 9.4.

$$O(Tam * \log Tam)$$

FÓRMULA 9.3

$$O(Tam^2)$$

FÓRMULA 9.4

Con respecto al espacio ocupado por la pila de recursión, en el mejor o en el promedio de los casos, se representa por medio de la expresión presentada en la fórmula 9.5. En el peor de los casos, con la expresión de la fórmula 9.6.

FÓRMULA 9.5

FÓRMULA 9.6

9.2.2 Método de ordenación por selección

Otra manera de ordenar un conjunto de datos es **seleccionar** el más pequeño y guardarlo en la primera casilla, luego el siguiente más pequeño y guardarlo en la segunda casilla y así sucesivamente hasta el penúltimo elemento (el último ya no requiere ordenarse).

A continuación se presenta la clase `SeleccionDirecta`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por selección directa.

```
/* Clase que implementa el algoritmo de selección directa. Es una clase
➥derivada de la clase abstracta Ordenador. */
template <class T>
class SeleccionDirecta: public Ordenador<T>
{
   public:
      void Ordena (Arreglo<T> *);
};

/* Este método ordena los elementos del arreglo buscando el elemento más
➥pequeño e intercambiándolo con el que ocupa la primera posición del
➥arreglo. Luego busca el siguiente más pequeño y lo almacena en la
➥segunda posición, y así hasta que el arreglo queda completamente
➥ordenado. */
template <class T>
void SeleccionDirecta<T>::Ordena(Arreglo<T> *Arre)
```

```
{
   int Menor, Ind1, Ind2, Ind3, Tam= Arre->RegresaTam();
   for (Ind1= 0; Ind1 < Tam-1; Ind1++)
   {
        Menor= Arre->RegresaValor(Ind1);
        Ind2= Ind1;
        for (Ind3= Ind1+1; Ind3 < Tam; Ind3++)
            if (Arre->RegresaValor(Ind3) < Menor)
            {
                Menor= Arre->RegresaValor(Ind3);
                Ind2= Ind3;
            }
        Arre->AsignaValor(Ind2, Arre->RegresaValor(Ind1));
        Arre->AsignaValor(Ind1, Menor);
   }
}
```


Considere el arreglo de la figura 9.6, se aplica el método visto para ordenarlo. En la tabla 9.7 se presentan los cambios que se van realizando en el arreglo a medida que sus elementos van quedando ordenados. Las casillas que se recuadran son aquellas cuyos elementos se intercambian y las que se sombrean son aquellas que quedan ordenadas.

Arre

29	35	18	43	21	16
0	1	2	3	4	5

FIGURA 9.6 *Ejemplo de arreglo a ordenar*

TABLA 9.7 Seguimiento del método de ordenación por selección directa

Menor	Ind1	Ind2	Ind3	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
29	0	0		29	35	18	43	21	16
			1						
18		2	2						
			3						
			4						
16		5	5						

continúa

TABLA 9.7 Continuación

Menor	Ind1	Ind2	Ind3	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
				16	35	18	43	21	29
35	1	1							
18		2	2						
			3						
			4						
			5						
				16	18	35	43	21	29
35	2	2							
			3						
21		4	4						
			5						
				16	18	21	43	35	29
43	3	3							
35		4	4						
29		5	5						
				16	18	21	29	35	43
35	4	4							
		5	5						
				16	18	21	29	35	43

El número de intercambios que se llevan a cabo es igual al número de elementos menos uno, lo cual queda expresado en la fórmula 9.7.

$$\text{Total Intercambios} = \texttt{Tam} - 1$$

FÓRMULA 9.7

Este método realiza el mismo número de comparaciones que el de intercambio directo, es decir `Tam * (Tam - 1)/2`, por lo tanto se puede decir que el tiempo de ejecución del algoritmo es:

$$O(Tam^2)$$

FÓRMULA 9.8

9.2.3 Método de ordenación por inserción

La **ordenación por inserción** consiste en tomar un elemento e insertarlo en el lado izquierdo del arreglo que ya se encuentra ordenado. El proceso empieza a partir de la segunda casilla y se aplica hasta el último elemento. Existen algunos métodos que se basan en esta idea:

- Inserción directa
- Inserción binaria
- Shell

Método de inserción directa

Este método ordena el arreglo a partir del segundo elemento, insertándolo en el lado izquierdo que ya está ordenado (la primera vez sólo se ordena con respecto al primer elemento). Luego de la primera iteración se tienen dos elementos ordenados y por lo tanto el tercer valor se inserta en la posición que le corresponda, de tal manera que el orden de los dos primeros elementos no se altere. Se repite el proceso hasta el valor `Tam-1`.

A continuación se presenta la clase `InsercionDirecta`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por inserción directa.

```
/* Clase para implementar el algoritmo de inserción directa. Es una cla-
➥se derivada de la clase abstracta Ordenador. */
template <class T>
class InsercionDirecta: public Ordenador<T>
{
   public:
      void Ordena (Arreglo<T> *);
};

/* Este método ordena los elementos del arreglo insertando cada elemento
➥en la parte izquierda del arreglo, asumiendo que la misma ya está
➥ordenada y por lo tanto sin alterar dicho orden. */
template <class T>
void InsercionDirecta<T>::Ordena(Arreglo<T> *Arre)
```

```
{
   int Auxiliar, Indice, IndAux, Tam= Arre->RegresaTam();
   for (Indice= 1; Indice < Tam; Indice++)
   {
        Auxiliar= Arre->RegresaValor(Indice);
        IndAux= Indice - 1;
        while ((IndAux >= 0) && (Auxiliar < Arre->RegresaValor(IndAux)))
        {
           Arre->AsignaValor(IndAux+1, Arre->RegresaValor(IndAux));
           IndAux--;
        }
        Arre->AsignaValor(IndAux+1, Auxiliar);
   }
}
```

Considere el arreglo de la figura 9.6, se aplica el algoritmo de inserción directa para ordenarlo. La tabla 9.8 presenta los cambios que se van realizando en el arreglo a medida que sus elementos van quedando ordenados. Las casillas que se recuadran son las que se modifican, y se sombrean las que quedan ordenadas. En la primera iteración quedan ordenadas las casillas 0 y la 1, en la segunda, las tres primeras y así sucesivamente hasta que en la iteración `Tam-1` quedan las `Tam` casillas ordenadas.

TABLA 9.8 Seguimiento del método de ordenación por inserción directa

Índice	Auxiliar	IndAux	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
1	35	0	29	35	18	43	21	16
2	18	1	29	35	35	43	21	16
		0	29	29	35	43	21	16
		–1	18	29	35	43	21	16
3	43	2	18	29	35	43	21	16
4	21	3	18	29	35	43	43	16
		2	18	29	35	35	43	16
		1	18	29	29	35	43	16
		0	18	21	29	35	43	16
5	16	4	18	21	29	35	43	43

continúa

TABLA 9.8 Continuación

Índice	Auxiliar	IndAux	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
		3	18	21	29	35	35	43
		2	18	21	29	29	35	43
		1	18	21	21	29	35	43
		0	18	18	21	29	35	43
		–1	16	18	21	29	35	43

Este método, en el peor de los casos, realiza una comparación en la primera iteración, en la segunda lleva a cabo dos y así hasta hacer (`Tam -1`) comparaciones en la última pasada. Por lo tanto se tienen $1 + 2 + \ldots +$ (`Tam -1`) comparaciones que es igual a:

Total comparaciones = `Tam * (Tam -1) /2`

FÓRMULA 9.9

Si se considera que en cada pasada, en promedio, se compara la mitad de los números antes de encontrar el lugar de inserción, se debe dividir a la expresión anterior entre 2 para obtener el número promedio de comparaciones, quedando:

Total comparaciones = `Tam * (Tam -1) /4`

FÓRMULA 9.10

Con respecto al tiempo promedio de ejecución, puede representarse por medio de la expresión:

$O(Tam^2)$

FÓRMULA 9.11

Método de inserción binaria

El método de **inserción binaria** es una mejora del anterior. En este caso se realiza **búsqueda binaria** para encontrar la posición que le corresponde al elemento a ordenar en la parte izquierda del arreglo. Es decir, cuando se toma el elemento *i* y se busca la posición correcta en la que debe insertarse, se usa la búsqueda binaria en lugar de la secuencial (ambos métodos de búsqueda se estudian en el próximo capítulo). De esta manera se aprovecha que los elementos que se encuentran a la izquierda del analizado ya están ordenados.

A continuación se presenta la clase `InsercionBinaria`, en la cual el método `Ordena` corresponde al algoritmo de ordenación por inserción binaria.

```
/* Clase para implementar el algoritmo de ordenación por inserción
➥binaria. Es una clase derivada de la clase abstracta Ordenador. */
template <class T>
class InsercionBinaria: public Ordenador<T>
{
   public:
       void Ordena (Arreglo<T> *);
};

/* Este método ordena los elementos del arreglo, insertando a cada uno
➥de ellos en la parte izquierda del mismo, asumiendo que se encuentra
➥ordenado. Utiliza búsqueda binaria para encontrar la posición que le
➥corresponde dentro de la parte ya ordenada del arreglo.  */
template <class T>
void InsercionBinaria<T>::Ordena(Arreglo<T> *Arre)
{
   int Auxiliar, Ind1, Ind2, Izq, Der, Medio, Tam= Arre->RegresaTam();
   for (Ind1= 1; Ind1 < Tam; Ind1++)
   {
       Auxiliar= Arre->RegresaValor(Ind1);
       Izq= 0;
       Der= Ind1-1;
       while (Izq <= Der)
       {
          Medio= int((Izq + Der)/2);
          if (Auxiliar <= Arre->RegresaValor(Medio))
              Der= Medio - 1;
          else
              Izq= Medio + 1;
       }
```

```
            Ind2= Ind1-1;
            while (Ind2 >= Izq)
            {
               Arre->AsignaValor(Ind2+1, Arre->RegresaValor(Ind2));
               Ind2--;
            }
            Arre->AsignaValor(Izq, Auxiliar);
         }
      }
```


Considere el arreglo de la figura 9.7, se aplica el último algoritmo visto para ordenarlo. La tabla 9.9 presenta los cambios que se van realizando en el arreglo a medida que sus elementos van quedando ordenados. Se recuadran las celdas cuyos contenidos van actualizándose y, al final de cada iteración, se sombrea la porción del arreglo que queda ordenada. Los valores que va tomando la variable `Medio` son las posiciones de los elementos contra los que se compara el dato a ordenar. Como se puede apreciar en la tabla, no se compara con todos los que están a su izquierda, lo cual es resultado de usar la búsqueda binaria.

Arre

29	35	18	43	21	16
0	1	2	3	4	5

FIGURA 9.7 *Ejemplo de arreglo a ordenar*

TABLA 9.9 Seguimiento del método de ordenación por inserción binaria

Ind1	Auxiliar	Izq	Der	Medio	Ind2	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
1	35	0	0	0		29	35	18	43	21	16
		1			0	29	35	18	43	21	16
2	18	0	1	0							
			–1		1	29	35	35	43	21	16
					0	29	29	35	43	21	16
					–1	18	29	35	43	21	16
3	43	0	2	1		18	29	35	43	21	16

continúa

TABLA 9.9 Continuación

Ind1	Auxiliar	Izq	Der	Medio	Ind2	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
		2		2		18	29	35	43	21	16
		3			2	18	29	35	43	21	16
4	21	0	3	1		18	29	35	43	21	16
		1	0	0		18	29	35	43	21	16
					3	18	29	35	43	43	16
					2	18	29	35	35	43	16
					1	18	29	29	35	43	16
					0	18	21	29	35	43	16
5	16	0	4	2		18	21	29	35	43	16
			1	0		18	21	29	35	43	16
			–1	4		18	21	29	35	43	43
				3		18	21	29	35	35	43
				2		18	21	29	29	35	43
				1		18	21	21	29	35	43
				0		18	18	21	29	35	43
				-1		16	18	21	29	35	43

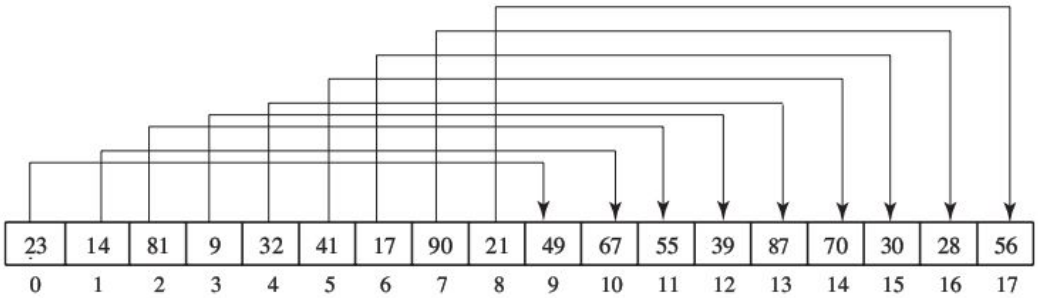
En esta variante del método de inserción, el número de intercambios no se altera, por lo tanto el tiempo promedio de ejecución es igual al presentado en la fórmula 9.11. Con respecto al número de comparaciones, realiza la mitad por usar búsqueda binaria. Por lo tanto, el total de comparaciones resulta igual al caso promedio del método de inserción analizado en la sección anterior (fórmula 9.10).

Método de `Shell`

Este método es otra variante mejorada del método de inserción directa. La idea de insertar un elemento en una posición de tal forma que el arreglo vaya quedando ordenado se mantiene, sin embargo, en este método los elementos a comparar se seleccionan con un intervalo que se reduce en cada iteración. Por ejemplo, la

primera vez el rango es de `Tam/2` posiciones, luego `Tam/4`, después `Tam/8` y así hasta obtener un valor de 1. Observe cómo en el siguiente ejemplo se va reduciendo el intervalo de los elementos a comparar. La primera vez, como `Tam` es 18 se compara el valor de la posición 0 con el de la posición 9, el de la 1 con el de la 10 y así hasta el final. En cada caso, dependiendo de la comparación, se hace el intercambio correspondiente. Si hubo al menos un intercambio entonces se comparan nuevamente los contenidos de dichas casillas para asegurar que dichos elementos queden ordenados. La segunda vez, luego de reducir el tamaño del intervalo en 2, se compara el valor de la casilla 0 con el de la 4, el de la 1 con el de la 5 y así hasta el final. El proceso se repite hasta que los elementos a comparar (e intercambiar si correspondiera) ocupan posiciones consecutivas.

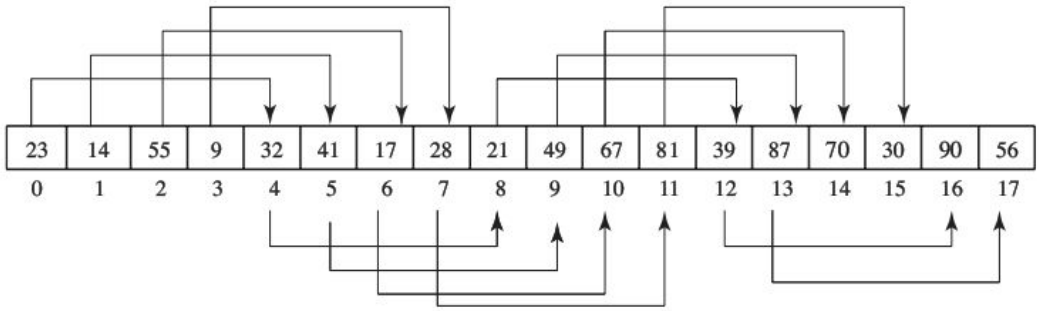
Considere el arreglo de 18 elementos que se muestra a continuación. Sobre el mismo se aplicará el algoritmo de Shell para ilustrar gráficamente su funcionamiento. Se sombrean las casillas en las cuales se realizó algún intercambio de contenido.



Inicialmente se define un intervalo de tamaño igual a 9 (`Tam/2`). El siguiente arreglo muestra el resultado luego de la comparación e intercambio de los valores 81 con 55 y 90 con 28.

23	14	55	9	32	41	17	28	21	49	67	81	39	87	70	30	90	56
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

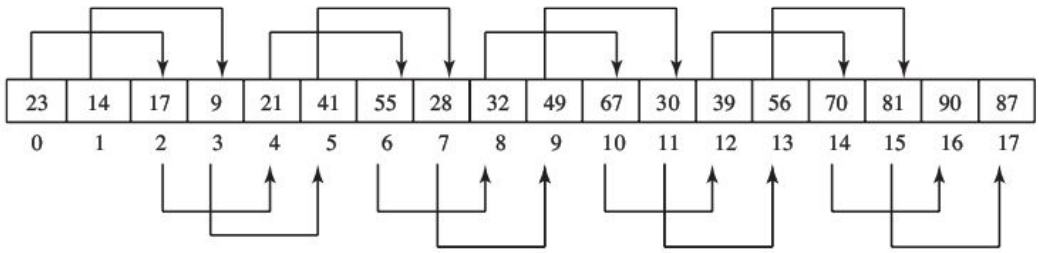
En el siguiente paso, el intervalo se define de tamaño igual a 4 (el valor anterior se reduce a la mitad). Por lo tanto, se van a comparar (y, si corresponde, intercambiar) los contenidos de las posiciones 0 y 4, 1 y 5, 2 y 6, y así hasta el final.



El siguiente arreglo muestra el resultado luego de la comparación e intercambio de los valores 55 con 17, 32 con 21, 81 con 30 y 87 con 56.

23	14	17	9	21	41	55	28	32	49	67	30	39	56	70	81	90	87
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

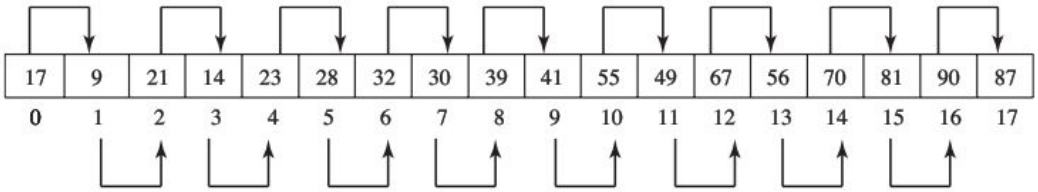
En el siguiente paso, el intervalo se define de tamaño igual a 2 (el valor anterior se reduce a la mitad). Ahora se van a comparar (y, si corresponde, intercambiar) los contenidos de las posiciones 0 y 2, 1 y 3, 2 y 5, y así hasta el final.



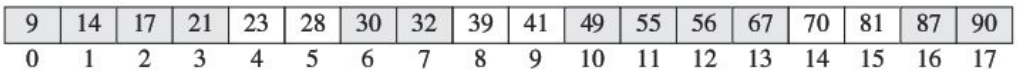
El siguiente arreglo muestra el resultado luego de la comparación e intercambio de los valores 23 con 17, 14 con 9, 41 con 28, 55 con 32, 49 con 30 y 67 con 39.

17	9	21	14	23	28	32	30	39	41	55	49	67	56	70	81	90	87
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

En el siguiente paso, el intervalo se define de tamaño igual a 1 (el valor anterior se reduce a la mitad). Por lo tanto, se comparan (y, si corresponde, intercambian) elementos consecutivos.



El siguiente arreglo muestra el resultado luego de la comparación e intercambio de los valores 17 con 9, 21 con 14, 32 con 30, 55 con 49, 67 con 56, 90 con 87 y 17 con 14.



A continuación se presenta la clase `Shell`, en la cual el método `Ordena` corresponde al algoritmo de ordenación con el mismo nombre.

```
/* Clase para implementar el algoritmo de ordenación llamado Shell.
➥Clase derivada de la clase abstracta Ordenador. */
template <class T>
class Shell: public Ordenador<T>
{
   public:
       void Ordena (Arreglo<T> *);
};

/* Este método ordena los elementos del arreglo. */
template <class T>
void Shell<T>::Ordena(Arreglo<T> *Arre)
{
   int Intervalo, Indice, Bandera, Tam= Arre->RegresaTam();
   Intervalo= Tam;
   while (Intervalo > 1)
   {
       Intervalo= int(Intervalo/2);
       Bandera= 1;
       while (Bandera)
       {
          Bandera= 0;
          Indice= 0;
          while ((Indice + Intervalo) < Tam)
          {
             if (Arre->RegresaValor(Indice) >
                ➥Arre->RegresaValor(Indice + Intervalo))
```

```
                {
                    Intercambia(Indice, Indice + Intervalo, Arre);
                    Bandera= 1;
                }
                Indice++;
            }
        }
    }
}
```

La eficiencia de este método resulta difícil de analizar teóricamente. Algunos autores señalan rangos desde $O(Tam^{3/2})$ hasta $O(Tam^{7/6})$ y otros una eficiencia del orden de $O(Tam * (\log Tam)^2)$.

Considere el arreglo de la figura 9.8, se aplica el algoritmo de Shell para ordenarlo. La tabla 9.10 presenta los cambios que se van realizando en el mismo a medida que sus elementos van quedando ordenados. Se sombrean las casillas cuyos contenidos se modifican.

Arre

29	35	18	43	21	16
0	1	2	3	4	5

FIGURA 9.8 *Ejemplo de arreglo a ordenar*

TABLA 9.10 Seguimiento del método de ordenación Shell

Intervalo	Bandera	Índice	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
6			29	35	18	43	21	16
3	1		29	35	18	43	21	16
	0	0						
		1	29	21	18	43	35	16
	1	2	29	21	16	43	35	18
		3						
	0	0	29	21	16	43	35	18
		1	29	21	16	43	35	18

continúa

TABLA 9.10 Continuación

Intervalo	Bandera	Índice	Arre[0]	Arre[1]	Arre[2]	Arre[3]	Arre[4]	Arre[5]
		2	29	21	16	43	35	18
		3						
1	1							
	0	0	21	29	16	43	35	18
	1	1	21	16	29	43	35	18
	1	2	21	16	29	43	35	18
		3	21	16	29	35	43	18
		4	21	16	29	35	18	43
		5						
	0	0	16	21	29	35	18	43
	1	1	16	21	29	35	18	43
		2	16	21	29	35	18	43
		3	16	21	29	18	35	43
	1	4	16	21	29	18	35	43
		5	16	21	29	18	35	43
	0	0	16	21	29	18	35	43
		1	16	21	29	18	35	43
		2	16	21	18	29	35	43
	1	3	16	21	18	29	35	43
		4	16	21	18	29	35	43
		5	16	21	18	29	35	43
	0	0	16	18	21	29	35	43
	1	1	16	18	21	29	35	43
		2	16	18	21	29	35	43
		3	16	18	21	29	35	43
		4	16	18	21	29	35	43
		5	16	18	21	29	35	43
	0	0	16	18	21	29	35	43

A continuación se presenta la plantilla de la clase arreglo que se utilizó en todos los métodos de ordenación estudiados en este capítulo. Si el tipo T utilizado para darle valor a la plantilla fuera una clase, entonces en dicha clase se debería sobrecargar el operador << y el operador >> para que los objetos pudieran leerse y escribirse directamente. Asimismo, deberían sobrecargarse los operadores relacionales <, > y == para que los objetos pudieran compararse tal como lo establecen los métodos vistos.

Programa 9.1

```
/* Se define una constante que representa el número máximo de elementos
➥que puede almacenar el arreglo. */
#define MAX 100

/* Se define la plantilla de la clase Arreglo con todos sus atributos y
➥métodos. Se asume que no existe orden entre los elementos del arreglo. */
template <class T>
class Arreglo
{
   private:
       T Datos[MAX];
       int Tam;
   public:
       Arreglo();
       int RegresaTam();
       T RegresaValor(int);
       void AsignaValor(int, T);
       void Lectura();
       void Escribe();

};

/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class T>
Arreglo<T>::Arreglo()
{
   Tam= 0;
}

/* Método que regresa el total de elementos almacenados en el arreglo. */
template <class T>
int  Arreglo<T>::RegresaTam()
{
   return Tam;
}
/* Método que regresa el contenido de la casilla identificada con el
➥valor de Indice. */
template <class T>
T Arreglo<T>::RegresaValor(int Indice)
```

```
{
   return Datos[Indice];
}

/* Método que asigna el contenido de Valor a la casilla indicada por
➥Indice. */
template <class T>
void Arreglo<T>::AsignaValor(int Indice, T Valor)
{
   Datos[Indice]= Valor;
}

/* Método que lee del teclado y almacena en el arreglo un conjunto de
➥valores. */
template <class T>
void Arreglo<T>::Lectura()
{
   int Indice;
   /* Lectura del número de elementos a guardar en el arreglo. Se valida
   ➥que el valor dado por el usuario sea menor o igual que el máximo
   ➥permitido. */
   do  {
        cout<<"\n\n Ingrese total de elementos:  ";
        cin>> Tam;
   } while (Tam < 1 || Tam > MAX);

   /* Lectura de valores para cada una de las Tam casillas del arreglo. */
   for (Indice= 0; Indice < Tam; Indice++)
   {
        cout<<"\nIngrese  el  "<<Indice + 1<<"  dato:  ";
        cin>> Datos[Indice];
   }
}

/* Método que despliega los valores almacenados en las casillas del
➥arreglo. */
template <class T>
void Arreglo<T>::Escribe()
{
   int Indice;
   if (Tam > 0)
   {
        cout<<"\n\n";
        for (Indice= 0; Indice < Tam; Indice++)
             cout<< '\t' << Datos[Indice];
        cout<<"\n\n";
   }
   else
       cout<< "\n No hay elementos almacenados.";
}
```

El programa 9.3 presenta parte de un programa de aplicación de los métodos de ordenación estudiados hasta el momento, el cual, utiliza tres librerías, una de ellas es *MetOrdena.h* en la cual se guardó la clase abstracta `Ordenador` y todas sus clases derivadas, otra es *Arreglo.h* que corresponde al programa 9.1 y la última *Alumno.h* en la cual se almacenó la clase `Alumno` que sirve como tipo base para declarar los elementos del arreglo. Por razones de espacio sólo se incluye la clase `Alumno` en el programa 9.2, quedando a cargo de usted la reconstrucción de la librería *MetOrdena.h* a partir del código de todos los métodos analizados.

Programa 9.2

```
/* Declaración de la clase Alumno. Se incluyeron sólo dos atributos, la
➥clave y el nombre del alumno. Se sugiere que usted los complemente. */
class Alumno {
private:
   int  Clave;
   char Nombre[64];
public:
   Alumno();
   Alumno(int, char *);
   int  operator > (Alumno);
   friend  istream &operator >> (istream &, Alumno &);
   friend  ostream &operator << (ostream &, Alumno &);
};

 /* Método constructor por omisión. */
Alumno::Alumno()
{}

/* Método constructor con parámetros. */
Alumno::Alumno(int  Cla, char Nom[])
{
   Clave= Cla;
   strcpy(Nombre, Nom);
}

/* Sobrecarga del operador > para que un objeto tipo Alumno pueda ser
➥comparado directamente. La comparación se realiza sólo sobre el
➥atributo Clave. */
int  Alumno::operator > (Alumno ObjAl)
{
   if (Clave > ObjAl.Clave)
       return 1;
   else
       return 0;
}
```

```
/* Sobrecarga del operador >> para que un objeto tipo Alumno pueda ser
➥leído directamente. */
istream &operator >> (istream &Lee,  Alumno &ObjAl)
{
   cout<<"\n\nIngrese clave del alumno: ";
   Lee>>ObjAl.Clave;
   cout<<"\n\nIngrese nombre del alumno: ";
   Lee>>ObjAl.Nombre;
   return  Lee;
}

/* Sobrecarga del operador << para que un objeto tipo Alumno pueda ser
➥impreso directamente. */
ostream &operator << (ostream &Escribe, Alumno &ObjAl)
{
   Escribe<<"\n\nDatos del alumno\n";
   Escribe<<"\nClave:  "<<ObjAl.Clave;
   Escribe<<"\nNombre:  "<<ObjAl.Nombre<<"\n";
   return  Escribe;
}
```


Programa 9.3

```
/* Aplicación de los métodos de ordenación interna para ordenar un
➥arreglo de objetos tipo Alumno. Se leen varios objetos tipo Alumno y
➥se almacenan en un arreglo en el orden que se dan, posteriormente se
➥ordenan e imprimen. */

/* Librería que almacena la clase Alumno presentada en el programa 9.2 */
#include "Alumno.h"

/* Librería que almacena la clase Arreglo presentada en el programa 9.1 */
#include "Arreglo.h"

/* Librería que almacena la clase abstracta Ordenador y todas sus
➥clases derivadas que representan cada uno de los métodos de ordenación
➥estudiados. */
#include "MetOrdena.h"

/* Función principal. En este código (parte de una aplicación) se
➥declara el objeto Orden, de la clase IntercDirectoIzq, que representa
➥el método de ordenación por intercambio directo con desplazamiento del
➥elemento más pequeño hacia la izquierda. Además, se declara un objeto
➥de la clase Arreglo usando la clase Alumno. Posteriormente se usa el
➥objeto Orden para ordenar crecientemente el arreglo de alumnos según
➥la clave de los mismos. Por último, se imprimen los datos del alumno
➥con clave más pequeña. */
```

```
void main()
{
   /* Declaración y lectura del arreglo que almacena objetos tipo
   ➥Alumno. */
   Arreglo<Alumno> Escuela;
   Escuela.Lectura();

   /* Creación de un objeto de la clase IntercDirectoIzq, el cual se
   ➥usará para ordenar el arreglo de alumnos. */
   IntercDirectoIzq<Alumno> Orden;

   /* Se aplica el algoritmo de ordenación sobre el arreglo de alumnos. */
   Orden.Ordena(&Escuela);

   /* Se imprime el contenido del arreglo una vez ordenado. */
   Escuela.Escribe();

   /* Impresión de los datos del alumno con clave más pequeña, por lo
   ➥tanto quedó (luego de la ordenación) en el primer lugar del arreglo. */
   if (Escuela.RegresaTam() != 0)
        cout<<"Los datos del primer alumno son:
            ➥"<<Escuela.RegresaValor(0)<<"\n";
}
```

En este libro se presentaron a cada uno de los algoritmos de ordenación como subclases de una clase abstracta. Sin embargo, es posible definirlos como métodos dentro de la clase `Arreglo`. El programa 9.4 presenta parte de esta clase con algunos de los métodos estudiados. Por razones de espacio, se incluyen sólo los prototipos de los métodos de la clase `Arreglo`, mismos que fueron presentados en el programa 9.1.

Programa 9.4

```
/* Se define la plantilla de la clase Arreglo con todos sus atributos y
➥métodos. Se asume que no existe orden entre los elementos del arreglo.
➥Se incluyen algunos de los algoritmos de ordenación estudiados como
➥métodos de esta clase. */

/* Se define una constante que representa el número máximo de elementos
➥que puede almacenar el arreglo. */

#define MAX 100
```

```
template <class T>
class Arreglo
{
   private:
       T Datos[MAX];
       int  Tam;
   public:
       Arreglo();
       int  RegresaTam();
       T RegresaValor(int);
       void AsignaValor(int, T);
       void Intercambia(int, int);
       void IntercDirectoIzq();
       void InsercionDirecta();
       void SeleccionDirecta();
       void QuickSort();
       void Reduce(int, int);
       void Lectura();
       void Escribe();
};

/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class T>
Arreglo<T>::Arreglo()
{
   Tam=0;
}

/* Método auxiliar que intercambia los contenidos de dos elementos del
➥arreglo. */
template <class T>
void Arreglo<T>::Intercambia(int  Ind1, int  Ind2)
{
   T Auxiliar;
   Auxiliar= Datos[Ind1];
   Datos[Ind1]= Datos[Ind2];
   Datos[Ind2]= Auxiliar;
}

/* Método que ordena los elementos del arreglo usando el algoritmo de
➥intercambio directo con desplazamiento del elemento más pequeño hacia
➥el extremo izquierdo. */
template <class T>
void Arreglo<T>::IntercDirectoIzq()
```

```
{
   int  Ind1, Ind2;
   for (Ind1= 1; Ind1< Tam; Ind1++)
        for (Ind2= Tam-1; Ind2 >= Ind1; Ind2--)
             if (Datos[Ind2-1] > Datos[Ind2])
                 Intercambia(Ind2-1, Ind2);
}

/* Método que ordena los elementos del arreglo usando el algoritmo de
➥inserción directa. */
template <class T>
void Arreglo<T>::InsercionDirecta()
{
   int  Auxiliar, Indice, IndAux;
   for (Indice= 1; Indice < Tam; Indice++)
   {
        Auxiliar= Datos[Indice];
        IndAux= Indice - 1;
        while ((IndAux >= 0) && (Auxiliar < Datos[IndAux]))
        {
           Datos[IndAux+1]= Datos[IndAux];
           IndAux--;
        }
        Datos[IndAux+1]= Auxiliar;
   }
}

/* Este método ordena los elementos del arreglo utilizando el algoritmo
➥de selección directa. */
template <class T>
void Arreglo<T>::SeleccionDirecta()
{
   int  Menor, Ind1, Ind2, Ind3;
   for (Ind1= 0; Ind1 < Tam-1; Ind1++)
   {
        Menor= Datos[Ind1];
        Ind2= Ind1;
        for (Ind3= Ind1+1; Ind3 < Tam; Ind3++)
            if (Datos[Ind3] < Menor)
            {
                Menor= Datos[Ind3];
                Ind2= Ind3;
            }
        Datos[Ind2]= Datos[Ind1];
        Datos[Ind1]= Menor;
   }
}
```

```
/* Este método ordena los elementos del arreglo utilizando el algoritmo
➥QuickSort. */
template <class T>
void Arreglo<T>::QuickSort()
{
   Reduce(0, Tam-1);
}

/* Método auxiliar del algoritmo QuickSort. Las parámetros Inicio y Fin
➥representan los extremos del intervalo en el cual se está ordenando. */
template <class T>
void Arreglo<T>::Reduce(int  Inicio, int  Fin)
{
   if ( Tam > 0)
   {
        int  Izq, Der, Posic, Bandera;
        Izq= Inicio;
        Der= Fin;
        Posic= Inicio;
        Bandera= 1;
        while (Bandera)
        {
            Bandera= 0;
            while ((Datos[Posic] <= Datos[Der]) && (Posic != Der))
               Der--;
            if (Posic != Der)
            {
               Intercambia(Posic, Der);
               Posic= Der;
               while ((Datos[Posic] >= Datos[Izq]) && (Posic != Izq))
                  Izq++;
               if (Posic != Izq)
               {
                   Bandera=1;
                   Intercambia(Posic, Izq);
                   Posic= Izq;
               }
            }
        }
        if ((Posic-1) > Inicio)
            Reduce(Inicio, Posic-1);
        if (Fin > (Posic+1))
            Reduce(Posic+1, Fin);
   }
}
```

9.3 Ordenación externa

La **ordenación externa** hace referencia a ordenar un conjunto de datos que se encuentran almacenados en algún dispositivo en memoria secundaria o auxiliar. En este libro nos enfocaremos a ordenar datos que se encuentran almacenados en archivos. El resultado de aplicar un método de ordenación a un archivo es que todos sus elementos quedan ordenados de manera creciente o de manera decreciente.

- **Creciente**: $dato_1 \leq dato_2 \leq ... \leq dato_n$ (el primer dato es menor o igual que el segundo, éste es menor o igual que el tercero y así sucesivamente hasta el último dato).
- **Decreciente**: $dato_1 \geq dato_2 \geq ... \geq dato_n$ (el primer dato es mayor o igual que el segundo, éste es mayor o igual que el tercero y así sucesivamente hasta el último dato).

Los métodos utilizados para ordenar a los elementos de un archivo son:

TABLA 9.11 Métodos de ordenación

Métodos de ordenación externa
Mezcla directa
Mezcla equilibrada

Para programar los métodos de ordenación externa en el lenguaje ***C++*** se definió una clase base abstracta y dos clases derivadas. Cada una de las clases derivadas representa uno de los métodos que se estudiarán en este capítulo. La figura 9.9 presenta un esquema de las clases mencionadas.

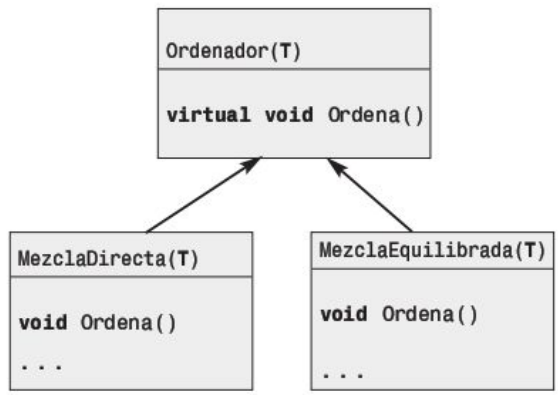


FIGURA 9.9 *Esquema de clases*

La clase `Base abstracta` se programa como se muestra a continuación. La misma tiene un método virtual puro el cual se redefinirá en cada subclase dependiendo del método de ordenación que se esté implementando.

```
/* Clase abstracta Ordenador, se utiliza para generar clases derivadas
➥que representan cada uno de los métodos de ordenación externa
➥estudiados. */
template <class T>
class Ordenador
{
   public:
      virtual void Ordena (char *) = 0;
};
```


9.3.1 Mezcla directa

Este método es muy sencillo y consiste en **dividir** el archivo en particiones y luego volver a **generar** el archivo a partir de estas particiones pero logrando que los elementos que componen a cada una de ellas queden ordenados. Este proceso se repite, pero ahora con particiones de mayor tamaño. De esta manera, el archivo va quedando ordenado por tramos hasta llegar a que todos sus elementos queden ordenados entre sí.

Para ayudar a entender más claramente cómo funciona este método se presenta el siguiente ejemplo. Suponga que se tiene un archivo `Arch` que almacena las claves:

`Arch`: 18, 23, 12, 45, 56, 33, 20, 16, 89, 34, 75, 44, 31, 14, 67, 28

1. Se elige un tamaño de partición igual a 1, formando dos archivos a partir de `Arch` de la siguiente manera:

   `Arch 1`: 18, 12, 56, 20, 89, 75, 31, 67

   `Arch 2`: 23, 45, 33, 16, 34, 44, 14, 28

   Luego de la partición, se vuelve a formar el archivo pero logrando que se tenga una secuencia de dos elementos ordenados entre sí:

   `Arch`: 18, 23, 12, 45, 33, 56, 16, 20, 34, 89, 44, 75, 14, 31, 28, 67

2. En la segunda iteración se elige un tamaño de partición igual a 2, formando dos archivos a partir de `Arch` de la siguiente manera:

Arch 1: [18, 23], [33, 56], [34, 89], [14, 31]

Arch 2: [12, 45], [16, 20], [44, 75], [28, 67]

Luego de la partición, se vuelve a formar el archivo pero logrando que se tenga una secuencia de cuatro elementos ordenados entre sí:

Arch: [12, 18, 23, 45], [16, 20, 33, 56], [34, 44, 75, 89], [14, 28, 31, 67]

3. En la siguiente iteración se elige un tamaño de partición igual a 4, formando dos archivos a partir de Arch de la siguiente manera:

   Arch 1: [12, 18, 23, 45], [34, 44, 75, 89]

   Arch 2: [16, 20, 33, 56], [14, 28, 31, 67]

   Nuevamente se vuelven a unir los dos archivos resultantes para formar el archivo original pero ahora con una secuencia de ocho elementos ordenados entre sí:

   Arch: [12, 16, 18, 20, 23, 33, 45, 56], [14, 28, 31, 34, 44, 67, 75, 89]

4. En la siguiente iteración se elige un tamaño de partición igual a 8, formando dos archivos a partir de Arch de la siguiente manera:

   Arch 1: [12, 16, 18, 20, 23, 33, 45, 56]

   Arch 2: [14, 28, 31, 34, 44, 67, 75, 89]

   Por último se unen los dos archivos resultantes para formar el archivo original con todos sus elementos ordenados:

   Arch: [12, 14, 16, 18, 20, 23, 28, 31, 33, 34, 44, 45, 56, 67, 75, 89]

Seguramente, en la práctica, el archivo almacenará datos más complejos que números enteros, sin embargo por razones de simplicidad se mostró el ejemplo sólo con números. A continuación se presenta, utilizando el lenguaje ***C++***, la clase `MezclaDirecta`, que es una clase derivada de la clase abstracta `Ordenador`, que implementa este método.

```
/* Declaración de la clase derivada MezclaDirecta, en la cual el método
➥Ordena se define con el algoritmo correspondiente al método de
➥ordenación externa llamado mezcla directa. */
template <class T>
class MezclaDirecta: public Ordenador <T>
```

```
{
   public:
      void Ordena (char *);
      void Divide (fstream  , fstream  *, fstream  *, int);
      void Mezcla (fstream  *, fstream , fstream , int);
};

/* Método de ordenación de la clase MezclaDirecta. Este método se apoya
➥en otros dos: Divide y Mezcla, los cuales implementan las dos
➥operaciones analizadas en el ejemplo anterior. Recibe como parámetro
➥el nombre del archivo a ordenar. */
template <class T>
void MezclaDirecta <T>::Ordena (char *NomArch)
{
   fstream   Arch, Arch1, Arch2;
   int Partic, Maximo;

   Arch.open(NomArch, ios::in !ios::out);
   Arch.seekg(0, ios::end);
   /* Calcula el total de datos guardados en el archivo. */
   Maximo= (int) (Arch.tellg())/sizeof(int);
   Partic= 1;
   while (Partic < Maximo)
   {
      Divide(Arch, &Arch1, &Arch2, Partic);
      Mezcla(&Arch, Arch1, Arch2, Partic);
      Partic= Partic * 2;
   }
   Arch.close();
}

/* Método auxiliar que parte el archivo a ordenar en dos archivos, de
➥acuerdo a un tamaño de partición que recibe como parámetro, junto con
➥el archivo original y los dos que formará. */
template <class T>
void MezclaDirecta <T>::Divide (fstream  Arch, fstream  *Arch1,
                                fstream  *Arch2, int Partic)
{
   int Cont, Dato;

   Arch.seekg(0, ios::beg);
   Arch1->open("MezclaDirAux1.dat", ios::out);
   Arch2->open("MezclaDirAux2.dat", ios::out);


   while (!Arch.eof())
```

```
    {
       Cont= 0;
       while ((Cont < Partic)  &&  (!Arch.eof()))
       {
          Arch.read((char *) &Dato,sizeof(Dato));
          if (!Arch.eof())
               Arch1->write((char *)&Dato, sizeof(Dato));
          Cont++;
       }
       Cont= 0;
       while ((Cont < Partic) && (!Arch.eof()))
       {
          Arch.read((char *) &Dato, sizeof(Dato));
          if (!Arch.eof())
               Arch2->write((char *)&Dato, sizeof(Dato));
          Cont++;
       }
    }
    Arch1->close();
    Arch2->close();
}

/* Método auxiliar que mezcla dos archivos que recibe como parámetro y
➥genera otro el cual va quedando ordenado. */
template <class T>
void MezclaDirecta <T>::Mezcla (fstream  *Arch, fstream  Arch1, fstream
Arch2, int Partic)
{
    int Dato1, Dato2, Part1, Part2, Band1, Band2;

    Arch->seekp(0, ios::beg);
    Arch1.open("MezclaDirAux1.dat", ios::in);
    Arch2.open("MezclaDirAux2.dat", ios::in);
    Band1= 1;
    Band2= 1;
    Arch1.read((char *)&Dato1, sizeof(Dato1));
    if (!Arch1.eof())
         Band1= 0;
    Arch2.read((char *)&Dato2,sizeof(Dato2));
    if (!Arch2.eof())
         Band2= 0;
    while ( ((!Arch1.eof()) || (!Band1)) && ((!Arch2.eof()) || (!Band2)) )
    {
        Part1= 0;
        Part2= 0;
        while (((Part1 < Partic) && (!Band1)) && ((Part2 < Partic)
              ➥&& (!Band2)))
```

```
    {
       if (Dato1 <= Dato2)
       {
            Arch->write((char *)&Dato1, sizeof(Dato1));
            Band1= 1;
            Part1++;
            Arch1.read((char *)&Dato1, sizeof(Dato1));
            if (!Arch1.eof())
                 Band1= 0;
       }
       else
       {
            Arch->write((char *)&Dato2, sizeof(Dato2));
            Band2= 1;
            Part2++;
            Arch2.read((char *)&Dato2, sizeof(Dato2));
            if (!Arch2.eof())
                 Band2= 0;
       }
    }
    while ((Part1 < Partic) && (!Band1))
    {
       Arch->write((char *)&Dato1, sizeof(Dato1));
       Band1= 1;
       Part1++;
       Arch1.read((char *)&Dato1, sizeof(Dato1));
       if (!Arch1.eof())
            Band1= 0;
    }
    while ((Part2 < Partic) && (!Band2))
    {
       Arch->write((char *) &Dato2, sizeof(Dato2));
       Band2= 1;
       Part2++;
       Arch2.read((char *) &Dato2, sizeof(Dato2));
       if (!Arch2.eof())
            Band2= 0;
    }
}
if (!Band1)
     Arch->write((char *)&Dato1, sizeof(Dato1));
if (!Band2)
     Arch->write((char *)&Dato2, sizeof(Dato2));

Arch1.read((char *)&Dato1, sizeof(Dato1));
while (!Arch1.eof())
```

```
        {
            Arch->write((char *)&Dato1, sizeof(Dato1));
            Arch1.read((char *)&Dato1, sizeof(Dato1));
        }

        Arch2.read((char *)&Dato2, sizeof(Dato2));
        while (!Arch2.eof())
        {
           Arch->write((char *)&Dato2, sizeof(Dato2));
           Arch2.read((char *)&Dato2, sizeof(Dato2));
        }

        Arch1.close();
        Arch2.close();
    }
```

Si los elementos a ordenar son objetos, se requiere sobrecargar los operadores relacionales usados en este algoritmo. La sobrecarga debe incluirse en la clase que se usará como tipo para los elementos del archivo.

9.3.2 Mezcla equilibrada

Este método es una versión mejorada del anterior. La mejora consiste en que el archivo se divide teniendo en cuenta las secuencias ordenadas de elementos que tuviera, y no un tamaño establecido por número de elementos. Una vez realizada una partición inicial de los datos en dos archivos auxiliares, se comienza a unir las particiones y a guardarlas en otros dos archivos formando secuencias cada vez más grandes de elementos ordenados. El proceso termina cuando todos los elementos a ordenar quedan en un mismo archivo, luego de efectuar una partición y unión de elementos. A continuación se muestra un ejemplo para que pueda entender más fácilmente la lógica del mismo. Se tiene el archivo:

`Arch`: 18, 23, 12, 45, 56, 33, 20, 16, 89, 34, 75, 44, 31, 14, 67, 28

Y se tienen tres archivos auxiliares: `Arch1`, `Arch2` y `Arch3`.

1. Se realiza una partición inicial de los elementos del archivo que se quiere ordenar, de tal manera que los mismos se distribuyan en dos archivos auxiliares de acuerdo al orden que exista entre ellos. Los archivos `Arch2` y `Arch3` quedan así:

Arch2: [18, 23], [33], [16 , 89], [44], [14, 67]

Arch3: [12, 45, 56], [20], [34, 75], [31], [28]

Luego se unen las particiones y se van formando otros dos archivos con el resultado de esta operación. En estos archivos van quedando secuencias más grandes de elementos ordenados.

Arch: [12, 18, 23, 45, 56], [16, 34, 75, 89], [14, 28, 67]

Arch1: [20, 33], [31 , 44]

2. Se realiza la mezcla de dos particiones (una de cada archivo) y se van generando dos archivos auxiliares con secuencias más grandes de elementos ordenados. Los archivos Arch2 y Arch3 quedan así:

   Arch2: [12, 18, 20, 23, 33, 45, 56], [14, 28, 67]

   Arch3: [16, 31, 34, 44, 75, 89]

3. Nuevamente las particiones de cada uno de los archivos se mezclan y se redefinen los primeros dos archivos: Arch y Arch1.

   Arch: [12, 16, 18, 20, 23, 31, 33, 34, 44, 45, 56, 75, 89]

   Arch1: [14, 28, 67]

4. Se lleva a cabo la mezcla de las particiones existentes dando como resultado los archivos Arch2 y Arch3 con la información distribuida entre ellos de la siguiente manera:

   Arch2: [12, 14, 16, 18, 20, 23, 28, 31, 33, 34, 44, 45, 56, 67, 75, 89]

   Arch3:

Como Arch3 queda vacío, el proceso concluye con éxito: el archivo quedó ordenado. A continuación se presenta la codificación de la clase MezclaEquilibrada, derivada de la clase abstracta Ordenador, la cual implementa el algoritmo descrito.

```
/* Declaración de la clase derivada MezclaEquilibrada, en la cual el
➥método Ordena se define con el algoritmo correspondiente al método de
➥ordenación externa llamado mezcla equilibrada. En la clase se incluyen
➥algunos métodos auxiliares que permiten realizar las particiones y
➥agrupamientos de los elementos del archivo. */
```

```
template <class T>
class MezclaEquilibrada: public Ordenador <T>
{
   public:
   void Ordena (char *);
   void DividePrim (char *, fstream  *, fstream  *);
   int DivideMezcla (fstream  *, fstream  *, fstream  *, fstream  *);
   void Escribe1 (int *, int, fstream  *, fstream  *, int);
   void Escribe2 (int *, int, fstream  *, fstream  *, int *);
   void Escribe3 (int, int, fstream  *, fstream  *, fstream  *, int);
};

/* Método de ordenación de la clase MezclaEqulibrada. Este método se
➥apoya en otros auxiliares. Recibe como parámetro el nombre del archivo
➥a ordenar. */
template <class T>
void MezclaEquilibrada <T>::Ordena(char *NomArch)
{
   int Band1, Band2;
   fstream  Arch, Arch1, Arch2, Arch3;

   DividePrim(NomArch, &Arch2, &Arch3);
   Band1= 1;
   Band2= 1;
   do {
       if (Band1)
       {
           Arch.open(NomArch, ios::out);
           Arch1.open("MezEquil1.dat", ios::out);
           Arch2.open("MezEquil2.dat", ios::in);
           Arch3.open("MezEquil3.dat", ios::in);
           Band2= DivideMezcla(&Arch2, &Arch3, &Arch, &Arch1);
           Band1= 0;
       }
       else
       {
           Arch.open(NomArch, ios::in);
           Arch1.open("MezEquil1.dat", ios::in);
           Arch2.open("MezEquil2.dat", ios::out);
           Arch3.open("MezEquil3.dat", ios::out);
           Band2= DivideMezcla(&Arch, &Arch1, &Arch2, &Arch3);
           Band1= 1;
       }
       Arch.close();
       Arch1.close();
       Arch2.close();
       Arch3.close();
   } while (Band2 != 0);
}
```

```
/* Método auxiliar que realiza la división inicial del archivo a
➥ordenar. */
template <class T>
void MezclaEquilibrada <T>::DividePrim(char *NomArch, fstream  *Arch2,
                                       fstream  *Arch3)
{
   int Dato, DatoAux, Band;
   fstream  Arch;

   Arch.open(NomArch, ios::in);
   Arch2->open("MezEquil2.dat", ios::out);
   Arch3->open("MezEquil3.dat", ios::out);
   Arch.read((char *)&Dato, sizeof(Dato));
   Arch2->write((char *)&Dato, sizeof(Dato));
   Band= 1;
   DatoAux= Dato;

   Arch.read((char *)&Dato, sizeof(Dato));
   while (!Arch.eof())
   {
      if (Dato >= DatoAux)
          if (Band)
              Arch2->write((char *)&Dato, sizeof(Dato));
          else
              Arch3->write((char *)&Dato, sizeof(Dato));
      else
          if (Band)
          {
              Arch3->write((char *)&Dato, sizeof(Dato));
              Band= 0;
          }
          else
          {
              Arch2->write((char *)&Dato, sizeof(Dato));
              Band= 1;
          }
      DatoAux= Dato;
      Arch.read((char *)&Dato, sizeof(Dato));
   }
   Arch.close();
   Arch2->close();
   Arch3->close();
}

/* Método auxiliar que mezcla dos archivos generando otros dos, que
➥tienen un mayor número de elementos ordenados. */
template <class T>
int MezclaEquilibrada <T>::DivideMezcla(fstream  *ArchX, fstream
                                        ➥*ArchY, fstream  *ArchZ,
                                        ➥fstream  *ArchW)
```

```
{
   int Dato1, Dato2, DatoAux, Band= 1, Lee1, Lee2;

   ArchX->read((char *)&Dato1, sizeof(Dato1));
   ArchY->read((char *)&Dato2, sizeof(Dato2));
   if (Dato1 < Dato2)
       DatoAux= Dato1;
   else
       DatoAux= Dato2;
   Lee1= 0;
   Lee2= 0;
   while ((((!ArchX->eof()) || (!Lee1)) && (!ArchY->eof()) || (!Lee2))
   {
      if (Dato1 < Dato2)
          if ((!Lee1) && (Dato1 >= DatoAux))
          {
                Escribe1(&DatoAux, Dato1, ArchZ, ArchW, Band);
                Lee1= 1;
          }
          else
                if ((!Lee2) && (Dato2 >= DatoAux))
                {
                      Escribe1(&DatoAux, Dato2, ArchZ, ArchW, Band);
                      Lee2= 1;
                }
                else
                {
                     if (!Lee1)
                     {
                         Escribe2(&DatoAux, Dato1, ArchZ, ArchW, &Band);
                         Lee1= 1;
                      }
                }
      else
          if ((!Lee2) && (Dato2 >= DatoAux))
          {
                Escribe1(&DatoAux, Dato2, ArchZ, ArchW, Band);
                Lee2= 1;
          }
          else
                if ((!Lee1) && (Dato1 >= DatoAux))
                {
                      Escribe1(&DatoAux, Dato1, ArchZ, ArchW, Band);
                      Lee1= 1;
                }
                else
                    if (!Lee2)
                    {
                         Escribe2(&DatoAux, Dato2, ArchZ, ArchW, &Band);
                         Lee2= 1;
                    }
```

```
        if (Lee1)
        {
            ArchX->read((char *)&Dato1, sizeof(Dato1));
            if (!ArchX->eof())
                Lee1= 0;
        }
        if (Lee2)
        {
            ArchY->read((char *)&Dato2, sizeof(Dato2));
            if (!ArchY->eof())
                Lee2= 0;
        }
    }
    if ((Lee1) && (ArchX->eof()))
        Escribe3(DatoAux, Dato2, ArchY, ArchZ, ArchW, Band);
    if ((Lee2) && (ArchY->eof()))
        Escribe3(DatoAux, Dato1, ArchX, ArchZ, ArchW, Band);
    if (ArchW->tellp() == 0)
        return 0;
    else
        return 1;
}

/* Método auxiliar para guardar la información en los archivos de
➥salida. */
template <class T>
void MezclaEquilibrada <T>::Escribe1(int *DatoAux, int Dato, fstream
                                    ➥*ArchZ, fstream  *ArchW, int Band)
{
    *DatoAux= Dato;
    if (Band)
        ArchZ->write((char *)&Dato, sizeof(Dato));
    else
        ArchW->write((char *)&Dato, sizeof(Dato));
}

/* Método auxiliar para guardar la información en los archivos de
➥salida. */
template <class T>
void MezclaEquilibrada <T>::Escribe2(int *DatoAux, int Dato, fstream
                                    ➥*ArchZ, fstream  *ArchW, int *Band)
{
    *DatoAux= Dato;
    if (*Band)
    {
        ArchW->write((char *)&Dato, sizeof(Dato));
        *Band= 0;
    }
```

```
    else
    {
        ArchZ->write((char *)&Dato, sizeof(Dato));
        *Band= 1;
    }
}

/* Método auxiliar para guardar la información en los archivos de
➥salida. */
template <class T>
void MezclaEquilibrada <T>::Escribe3(int DatoAux, int Dato, fstream
                                    ➥*Arch, fstream  *ArchZ, fstream
                                    ➥*ArchW, int Band)
{
if (Dato >= DatoAux)
        Escribe1(&DatoAux, Dato, ArchZ, ArchW, Band);
    else
        Escribe2(&DatoAux, Dato, ArchZ, ArchW, &Band);
    Arch->read((char *)&Dato, sizeof(Dato));
    while (!Arch->eof())
    {
       if (Dato >= DatoAux)
           Escribe1(&DatoAux, Dato, ArchZ, ArchW, Band);
       else
           Escribe2(&DatoAux, Dato, ArchZ, ArchW, &Band);

       Arch->read((char *)&Dato, sizeof(Dato));
    }
}
```

Si los elementos a ordenar son objetos, se requiere sobrecargar los operadores relacionales usados en este algoritmo. La sobrecarga debe incluirse en la clase que se usará como tipo para los elementos del archivo.

Ejercicios

1. Probar todas las variantes del método de ordenación interna por intercambio en los siguientes casos:

   a) Con un arreglo ordenado.

b) Con un arreglo en orden inverso.

c) Con un arreglo desordenado.

d) Con un arreglo vacío.

e) Con un arreglo desordenado que tiene elementos duplicados.

2. Probar el método de ordenación interna por selección en los siguientes casos:

   a) Con un arreglo ordenado.

   b) Con un arreglo en orden inverso.

   c) Con un arreglo desordenado.

   d) Con un arreglo vacío.

   e) Con un arreglo desordenado que tiene elementos duplicados.

3. Probar todas las variantes del método de ordenación interna por inserción en los siguientes casos:

   a) Con un arreglo ordenado.

   b) Con un arreglo en orden inverso.

   c) Con un arreglo desordenado.

   d) Con un arreglo vacío.

   e) Con un arreglo desordenado que tiene elementos duplicados.

4. Retome el método de ordenación interna `QuickSort`. Reescríbalo sin usar recursión. Se sugiere el uso de pilas para ir guardando los extremos de los intervalos pendientes de ordenación.

5. Retome el método de ordenación interna `QuickSort`. Reescríbalo utilizando otras variantes para la elección del pivote. Se sugiere probar con el elemento medio. Compare el desempeño de su solución con la que se dio en este libro para diferentes tamaños de arreglos.

6. Complete la siguiente tabla con la evaluación del desempeño (tiempo de ejecución y/o número de comparaciones y/o número de intercambios) de los métodos de ordenación para distintos tamaños de arreglos.

	Tam 10	Tam 100	Tam 1000	Tam 10, 000
Intercambio directo con desplazamiento hacia la izquierda				
Intercambio directo con desplazamiento hacia la derecha				
Sheker				
Intercambio con señal				
QuickSort				
Selección directa				
Inserción directa				
Inserción binaria				
Shell				

7. Retome la clase `Arreglo` del programa 9.4. Complétela con los otros algoritmos de ordenación interna estudiados en este capítulo.
8. En el capítulo 4 se definió la clase `Dinos` para representar dinosaurios. Utilice esta clase como tipo base para la plantilla de la clase `Arreglo` del programa 9.4. Realice todos los ajustes necesarios para que el arreglo de dinosaurios quede ordenado según la clave que los identifica, aplicando cualquiera de los métodos vistos.
9. Resuelva el problema anterior, pero considerando que la clase `Arreglo` no incluye los métodos de ordenación, sino que los mismos son clases tal como se presentaron en este capítulo. Por lo tanto, en su solución tendrá un objeto de la clase `Arreglo` que es el que quiere ordenar y otro de alguna de las clases (`IntercDirectoIzq`, `IntercDirectoDer`, `Sheker`, `Shell`, ...) que representa al método de ordenación interna elegido para llevar a cabo la operación.
10. Considere un archivo de números enteros que representan las claves de ciertos productos. Utilice el algoritmo de mezcla directa para ordenarlo.

11. Retome el problema anterior, pero ahora utilice el algoritmo de mezcla equilibrada. ¿Notó alguna diferencia en el desempeño de los mismos? ¿Y si cambia el tamaño del archivo?

12. Considere la clase `Alumno` cuya especificación aparece más adelante.

    a) Desarrolle un programa de captura que lea los atributos de varios alumnos, y los guarde en un archivo, en el mismo orden que los lee.

    b) Utilice el algoritmo de mezcla directa para ordenar el archivo, según el nombre del alumno.

    c) Utilice el algoritmo de mezcla equilibrada para ordenar el archivo, según el nombre del alumno.

    d) Compare el desempeño de los algoritmos utilizados en los incisos anteriores.

Alumno
Nombre: cadena de caracteres Carrera: cadena de caracteres Número de materias aprobadas: entero Calificaciones obtenidas en materias aprobadas: arreglo de enteros (de máximo 60 valores) Total de materias reprobadas: entero
Constructor(es) Lectura Calcula promedio del alumno Cambia de carrera Imprime datos

13. Retome el archivo de objetos tipo `Alumno` creado en el problema anterior y asuma que ya fue ordenado según lo solicitado en los incisos (b) y (c). Desarrolle un programa que permita al usuario las siguientes opciones:

a) Generar un reporte con los datos de todos los alumnos, ordenados por el nombre.

b) Generar un reporte con los datos de todos aquellos alumnos que tengan un promedio mayor o igual a 9.

14. Defina la clase `Empleado` según las especificaciones que se dan más adelante. Desarrolle un programa, utilizando subprogramas y/u otras clases, para:

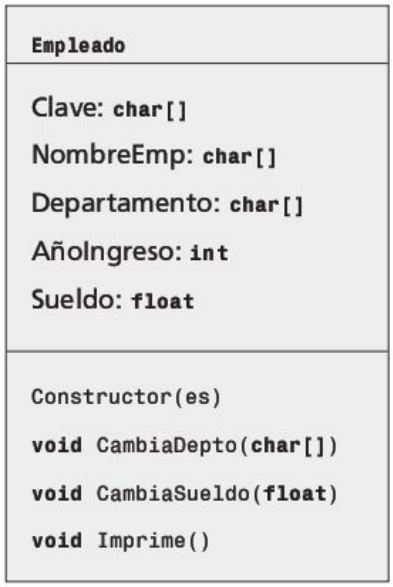
a) Capturar los datos de un grupo de empleados y guardarlos en un archivo, siguiendo el orden dado.

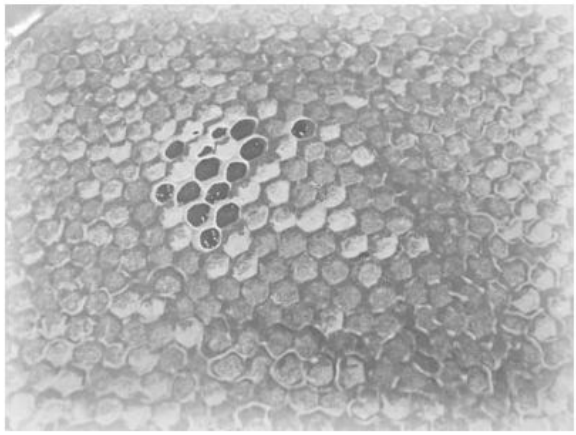
b) Ordenar el archivo de manera descendente según la clave del empleado. Decida qué método de ordenación externa utilizar.

c) Genere un reporte con los datos de todos los empleados que hayan ingresado antes de 1990 y que ganen más de cierta cantidad, la cual será dada por el usuario.

d) Actualice los datos de todos los empleados, dándoles un aumento del 10%.

e) Forme un archivo auxiliar sólo con los empleados del departamento de *finanzas*. Posteriormente, ordene alfabéticamente este archivo de manera creciente, según el nombre del empleado.





CAPÍTULO 10

Búsqueda

10.1 Introducción

La **búsqueda** es la operación que permite localizar un elemento en una estructura de datos. Es decir, ayuda a determinar si el dato buscado está o no en dicha estructura. Si la misma se realiza sobre datos almacenados en un arreglo, lista, árbol o gráfica se dice que es **búsqueda interna**. Por otra parte, si se aplica a un conjunto de valores guardados en un archivo, se dice que es **búsqueda externa**.

Si bien la búsqueda no es una estructura de datos, la misma se presenta en este libro porque es una de las operaciones más importantes que complementa el uso de cualquier estructura de datos. Almacenar información en una estructura de datos tiene sentido si después se puede tener acceso a ella, y para tener acceso se requiere el uso de

algoritmos de búsqueda. Es decir, es esta operación la que permite recuperar la información previamente almacenada en una estructura.

Normalmente esta operación se encuentra implementada como un método de otras clases, como la clase `Arreglo`, la clase `Lista` o la clase `Arbol`. Sin embargo, dado que es el tema central de estudio de este capítulo, se la tratará como una clase que representa a las variantes más conocidas del proceso de búsqueda.

10.2 Búsqueda interna

La **búsqueda interna** es aquella que se aplica a una estructura de datos previamente generada en la memoria de la computadora. En toda operación de este tipo se distinguen dos elementos, la estructura de datos (que representa dónde se realizará la búsqueda) y el elemento a buscar. Dependiendo de la estructura usada para almacenar los datos, se podrán aplicar diferentes algoritmos para intentar encontrar un elemento. La tabla 10.1 presenta las principales estructuras de datos internas con los posibles métodos de búsqueda a emplear.

TABLA 10.1 Estructuras de datos y métodos de búsqueda

Estructura de datos	*Tipo de búsqueda*
Arreglos	Secuencial (para cualquier tipo de arreglo). Binaria (sólo para arreglos ordenados). Transformación de claves (Hash).
Listas	Secuencial.
Árboles	Depende de la estructura interna del árbol. Se analizaron en el capítulo 7.
Gráficas	Depende de la estructura interna de la gráfica y del tipo de información que se quiera obtener. Se analizaron en el capítulo 8.

La figura 10.1 presenta un esquema de clases para los algoritmos de búsqueda que pueden aplicarse a arreglos. Se define la plantilla de una clase abstracta, `Búsqueda`, en la cual se incluye un método `virtual` que se especificará en cada una de las clases derivadas de acuerdo al algoritmo que representen. Por lo tanto, se definen tres subclases: `SecuencialDesord`, `SecuencialOrdenado` y `Binaria`, la

primera para la búsqueda secuencial en arreglos desordenados, la segunda para la búsqueda secuencial en arreglos ordenados y la última para búsqueda binaria, la cual siempre se realiza en arreglos ordenados. Al tercer tipo de búsqueda en arreglos (`Hash`) se le dedica la sección 10.2.3.

A continuación se presenta la codificación, usando el lenguaje *C++*, de la plantilla de la clase abstracta. Observe que el método `virtual Busca()` recibe como parámetro el arreglo donde se llevará a cabo la búsqueda y un elemento de tipo `T` que es el dato a buscar. Si la operación se definiera como un método de la clase arreglo (como se vio en el capítulo 4), entonces sólo se recibiría como parámetro el dato a buscar.


```
/* Clase abstracta que servirá como base para declarar cada una de las
➥clases derivadas que representan los distintos métodos de búsqueda en
➥arreglos. */
template <class T>
class Busqueda
{
   public:
      virtual int Busca (Arreglo<T>, T)= 0;
};
```

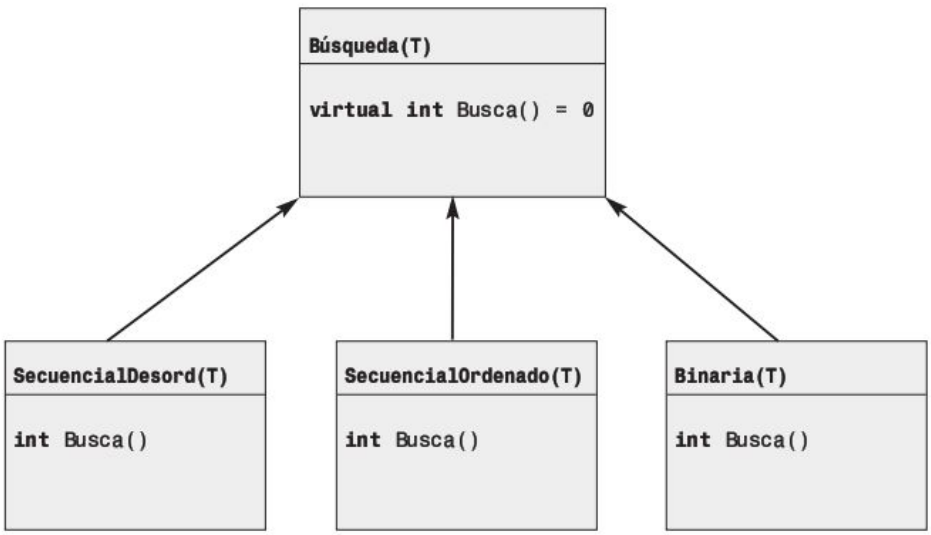


FIGURA 10.1 *Esquema de clases*

10.2.1 Búsqueda secuencial

La **búsqueda secuencial** consiste en recorrer el arreglo, elemento por elemento, empezando con el primero, hasta llegar al dato buscado o hasta que se hayan evaluado todos los componentes del arreglo. Esta última condición se modifica si el arreglo está ordenado.

Por ejemplo, considere que se han almacenado diez números enteros en el arreglo de la figura 10.2 y que el dato a buscar es 78. La búsqueda comienza a partir de la primera posición (índice 0) y se compara el 78 con el dato guardado en dicha posición. En este caso no son iguales por lo que se continúa con la siguiente casilla que tiene el valor 14. Nuevamente al comparar se determina que no son iguales, lo que requiere continuar con el siguiente elemento. La búsqueda termina cuando se compara con el número almacenado en la casilla 4, ya que son iguales y por lo tanto la operación termina con éxito. Ahora suponga que el dato buscado es el 28, en este caso se compara con todos los números guardados en el arreglo y como no es igual a ninguno de ellos, la operación fracasa cuando se llega al último valor.

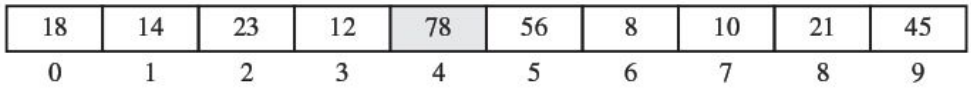


FIGURA 10.2 *Búsqueda secuencial*

Se define la clase `SecuencialDesord`, derivada de `Búsqueda`, para representar la búsqueda secuencial en **arreglos desordenados**. Esta clase tiene sólo un miembro, que es el método `Busca()`, el cual se especifica de acuerdo al algoritmo descrito. Es importante señalar que, si el dato a buscar fuera un objeto, se necesitaría que el operador **!=** estuviera sobrecargado en la clase correspondiente.

```
/* Definición de la clase encargada de realizar la búsqueda secuencial
➥en un arreglo cuyos elementos están desordenados. Es una clase derivada
➥de Busqueda y en ella se especifica el método Busca(). */
template <class T>
class SecuencialDesord: public Busqueda<T>
{
   public:
      int Busca (Arreglo<T>, T);
};
```

```
/* Método que realiza la búsqueda, elemento por elemento, de un dato
➥dado en un arreglo desordenado. Recibe como parámetros el dato a buscar
➥y el arreglo en el cual se llevará a cabo la operación. Si lo encuentra
➥da como resultado la posición, en caso contrario regresa un -1. */
template <class T>
int SecuencialDesord<T>::Busca(Arreglo<T> Arre, T Dato)
{
   int Indice= 0, Posic = -1;
   while (Indice < Arre.RegresaTam() && Dato != Arre.RegresaValor(Indice))
      Indice++;
   if (Indice < Arre.RegresaTam())
      Posic= Indice;
   return Posic;
}
```


Si el arreglo está ordenado, se modifica la última condición del ciclo, de manera que la búsqueda se interrumpa (antes de llegar al último elemento) si se detecta que, por el orden que tienen los valores del arreglo, ya no es posible encontrar el dato buscado. En consecuencia se gana eficiencia en cuanto al número de comparaciones. Si el arreglo está ordenado de manera creciente, la condición permitirá concluir que, al evaluar el dato buscado con respecto al elemento i, si este último es mayor, no será posible encontrar el dato dentro del arreglo; y al revés, si el arreglo está ordenado en forma decreciente, en cuanto se encuentre un elemento que sea menor que el dato buscado, se puede concluir que éste no se encontrará entre los elementos restantes.

Se define la clase `SecuencialOrdenado`, derivada de `Búsqueda`, para representar la búsqueda secuencial en **arreglos ordenados**. Esta clase tiene un sólo miembro, que es el método `Busca()`, el cual se especifica de acuerdo al algoritmo descrito. Es importante señalar que, si el dato a buscar fuera un objeto, se necesitaría que el operador > (para orden creciente o el < para orden decreciente) estuviera sobrecargado en la clase correspondiente.

```
/* Definición de la clase encargada de realizar la búsqueda secuencial
➥en un arreglo ordenado. Es una clase derivada de Busqueda y en ella se
➥especifica el método Busca(). */
template <class T>
class SecuencialOrdenado: public Busqueda<T>
{
   public:
      int Busca (Arreglo<T>, T);
};
```

```
/* Método que realiza la búsqueda de un dato en un arreglo cuyos valores
➥están ordenados de manera creciente. Esta operación se interrumpe
➥cuando se encuentra el valor buscado o cuando se compara a éste con un
➥valor mayor. Recibe como parámetros el dato a buscar y el arreglo en el
➥cual se llevará a cabo la operación. Si lo encuentra da como resultado
➥la posición, en caso contrario regresa el negativo de la posición en la
➥que debería estar, más 1. */
template <class T>
int SecuencialOrdenado<T>::Busca(Arreglo<T> Arre, T Dato)
{
   int Indice= 0, Posic;
   while (Indice < Arre.RegresaTam() && Dato > Arre.RegresaValor(Indice))
         Indice++;
   if (Indice == Arre.RegresaTam() || Dato < Arre.RegresaValor(Indice))
         Posic= -(Indice + 1);
   else
         Posic= Indice;
   return Posic;
}
```

Considerando que la búsqueda puede funcionar como una operación auxiliar a la inserción y eliminación en arreglos (ver capítulo 4), es conveniente que el método regrese la posición en la que encuentra al elemento o la posición en la que debería estar. Para poder distinguir estos dos casos es necesario regresar un valor positivo (si está) o uno negativo (si no está). Si la posición es cero, entonces no se le puede asociar el signo, razón por la cual se le suma uno. Los usuarios de este método deben tener en cuenta esto, y en caso de requerir la posición (por ejemplo para un desplazamiento en la operación de inserción) deberán convertirla pasándola a positiva y restándole uno.

La **eficiencia** de la búsqueda secuencial se mide por el número de comparaciones requeridas hasta encontrar el elemento buscado o hasta que se determine que el mismo no está en el arreglo. Por lo tanto, si el dato fue almacenado en el arreglo, éste puede estar en la primera posición, en alguna intermedia o en la última, lo cual implica realizar una, `algunas` o `Tam` comparaciones, respectivamente. Cuando el dato no está en el arreglo, se compara a éste con todos los elementos del arreglo, hasta llegar al final del mismo; es decir se realizan `Tam` comparaciones. Por lo tanto, en este tipo de búsqueda se distinguen tres casos: (1) el más favorable, con el número mínimo de comparaciones; (2) el intermedio, con un número medio de comparaciones; y (3) el más desfavorable, con el número máximo de comparaciones. Estas tres posibles situaciones se expresan de la siguiente manera:

Comparaciones mínimas = 1

Comparaciones medias = (1 + Tam) /2

Comparaciones máximas = Tam

FÓRMULAS 10.1

Si el arreglo está ordenado, se tiene mayor eficiencia cuando el dato buscado no está pero por su tamaño debería ocupar alguna posición intermedia. En este caso, la búsqueda se interrumpe sin necesidad de revisar todo el arreglo.

En el capítulo cuatro, dedicado a los arreglos, se hizo uso de este tipo de búsqueda como apoyo a las operaciones de inserción y eliminación en esas estructuras de datos. Esta operación se implementó como un método de la clase `Arreglo`. A continuación se presenta un ejemplo utilizando un objeto de la clase derivada `SecuencialOrdenado`, para ordenar un objeto de la clase `Arreglo`. Además, se emplea la clase `Persona` como tipo base para la plantilla de esta última clase.

El programa 10.1 presenta la clase `Persona` y la plantilla de la clase `Arreglo`. En ambas sólo se incluyen los métodos requeridos para esta aplicación.

Programa 10.1

```
/* Definición de la clase Persona. Se incluyen sobrecarga de operadores
➥para que objetos de este tipo puedan ser usados directamente en la
➥operación de búsqueda. */
class Persona
{
private:
   int AnioNac;
   char NomPers[64], LugNac[64];

public:
   Persona();
   Persona(int,  char[], char[]);
   int operator > (Persona);
   int operator < (Persona);
   friend  istream &operator >> (istream &, Persona &);
   friend  ostream &operator <<  (ostream &, Persona &);
};
```

```
/* Constructor por omisión. */
Persona::Persona()
{}

/* Constructor con parámetros. */
Persona::Persona(int ANac, char NomP[], char LugN[])
{
   AnioNac= ANac;
   strcpy(NomPers, NomP);
   strcpy(LugNac, LugN);
}

/* Sobrecarga del operador > para comparar dos objetos de la clase
➥Persona. Una persona es "mayor que" otra si su nombre lo es. Este
➥operador permitirá buscar a una persona, por su nombre, en un arreglo
➥de personas ordenado alfabéticamente. */
 int Persona::operator > (Persona Pers)
{
   int Resp=0;
   if (strcmp(NomPers, Pers.NomPers) > 0)
       Resp= 1;
   return Resp;
}

/* Sobrecarga del operador < para comparar dos objetos de la clase
➥Persona. Una persona es "menor que" otra si su nombre lo es. Este
➥operador permitirá buscar a una persona, por su nombre, en un arreglo
➥de personas ordenado alfabéticamente. */
int Persona::operator < (Persona Pers)
{
   int Resp=0;
   if (strcmp(NomPers, Pers.NomPers) < 0)
       Resp= 1;
   return Resp;
}

/* Sobrecarga del operador >> para que un objeto tipo Persona pueda ser
➥leído directamente. */
istream &operator >> (istream &Lee,  Persona &ObjPers)
{
   cout <<"\n\nIngrese nombre de la Persona:  ";
   Lee>> ObjPers.NomPers;
   cout <<"\n\nIngrese año de nacimiento:  ";
   Lee>> ObjPers.AnioNac;
   cout <<"\n\nIngrese lugar de nacimiento:  ";
   Lee>> ObjPers.LugNac;
   return  Lee;
}
```

```
/* Sobrecarga del operador << para que un objeto tipo Persona pueda ser
➥impreso directamente. */
ostream &operator << (ostream &Escribe, Persona &ObjPers)
{
   Escribe<<"\n\nDatos de la Persona\n";
   Escribe<<"\nNombre:  "<<ObjPers.NomPers;
   Escribe<<"\nLugar de nacimiento:  "<<ObjPers.LugNac;
   Escribe<<"\nAño de nacimiento:  "<<ObjPers.AnioNac;
   return  Escribe;
}


/* La constante MAX se usa para definir el tamaño máximo del arreglo. */
#define MAX 100

/* Plantilla de la clase Arreglo. Se incluyen sólo los métodos
➥requeridos para la aplicación de la operación de búsqueda. */
template <class T>
class Arreglo
{
   private:
       T Datos[MAX];
       int Tam;
   public:
       Arreglo();
       void  Lectura();
       void  Escribe();
       int RegresaTam();
       T RegresaValor(int);
};

/* Declaración del método constructor. Inicializa el número actual de
➥elementos en 0. */
template <class T>
Arreglo<T>::Arreglo()
{
   Tam= 0;
}

/* Método que permite leer el número de elementos que se van a almacenar
➥y el valor de cada uno de ellos. Valida que el total de elementos sea
➥al menos 1 y que no supere el máximo especificado. */
template <class T>
void  Arreglo<T>::Lectura()
{
   int Indice;
```

```
    do {
        cout<<"\n\n Ingrese número de datos a guardar:  ";
        cin>> Tam;
    } while (Tam < 1 || Tam > MAX);

    for (Indice= 0; Indice < Tam; Indice++)
    {
        cout<<"\nIngrese el "<<Indice+1<<"  dato:  ";
        cin>>Datos[Indice];
    }
}

/* Método que despliega en pantalla los valores almacenados en el
➥arreglo. */
template <class T>
void  Arreglo<T>::Escribe()
{
    int Indice;
    if (Tam > 0)
    {
        cout <<"\n Impresión de datos\n";
        for (Indice= 0; Indice < Tam; Indice++)
             cout << '\t' << Datos[Indice];
    }
    else
        cout << "\nNo hay elementos registrados.";
}

/* Método que permite a usuarios externos a la clase conocer el total de
➥elementos guardados en el arreglo. */
 template <class T>
int Arreglo<T>::RegresaTam()
{
    return Tam;
}

/* Método que permite a usuarios externos a la clase conocer el dato
➥almacenado en cierta casilla del arreglo. Recibe como parámetro un
➥entero y regresa como resultado el valor almacenado en la posición
➥indicada por dicho número. */
template <class T>
T Arreglo<T>::RegresaValor(int Indice)
{
    return Datos[Indice];
}
```

El programa 10.2 presenta la aplicación. Se declara un objeto de la clase `SecuencialOrdenado` y un objeto de la clase `Arreglo`, dándole la clase `Persona` como tipo para cada uno de sus elementos. Es decir, se tiene un arreglo de personas. Además, se asume que dicho arreglo está ordenado y por lo tanto se puede usar la búsqueda secuencial en arreglos ordenados para buscar una persona. El usuario da como dato el nombre de la persona a buscar y, si se encuentra, se imprimen todos los datos de dicha persona. En caso contrario, se imprime un mensaje adecuado.

Programa 10.2


```
/* Se incluyen las bibliotecas Arreglos.h y Persona.h donde fueron
➥guardadas las clases Arreglo y Persona respectivamente. En la biblio-
➥teca BusquedaInterna.h se tiene la clase Busqueda y su derivada
➥SecuencialOrdenado. */
#include "Arreglos.h"
#include "Persona.h"
#include "BusquedaInterna.h"


/* Función principal. Se declaran las variables de trabajo, se crea el
➥arreglo de personas (asumiendo que los datos se dan ordenados), se pide
➥el nombre de la persona a buscar y se usa un objeto de la clase
➥SecuencialOrdenado para realizar la búsqueda de la misma en el arreglo. */
void  main()
{
   /* Declaración de un objeto de la clase SecuencialOrdenado, con
   ➥Persona como tipo base. */
   SecuencialOrdenado<Persona> Buscador;

   /* Declaración de un objeto de la clase Arreglo, con Persona como
   ➥tipo base. */
   Arreglo<Persona> Asistentes;
   int Resp;
   char Nom[64];

   /* Lectura del arreglo. Se leen los datos de varias personas y se
   ➥almacenan en el arreglo. Se asume que los datos se dan ordenados
   ➥alfabéticamente de acuerdo al nombre de la persona. */
   Asistentes.Lectura();

   cout<<"\nIngresa el nombre de la persona a buscar:  ";
   cin>>Nom;
```

```
    /* Objeto auxiliar de tipo Persona, empleado para realizar la
    ➥búsqueda en el arreglo. Sólo se busca por el nombre. */
    Persona Alguien(0,0,Nom,"");
    /* Se invoca al método Busca() del objeto creado para buscar un
    ➥elemento en un arreglo ordenado, aplicando búsqueda secuencial. */
    Resp= Buscador.Busca(Asistentes, Alguien);
    if (Resp >= 0)
        cout<<"\n\nSe encontró a la persona y sus datos completos
              son\n"<<Asistentes.RegresaValor(Resp)<<"\n";
    else
        cout<<"\n\nNO se encontró a la persona\n\n";
}
```

10.2.2 Búsqueda binaria

La **búsqueda binaria** se puede aplicar sólo a arreglos **ordenados**. Este método se basa en una idea muy simple con la cual se trata de aprovechar el hecho de saber que el arreglo está ordenado. Inicialmente se considera todo el arreglo como el espacio de búsqueda, por lo tanto se establece el índice 0 como extremo izquierdo y el índice `Tam-1` como el extremo derecho. Se calcula la posición central del arreglo y se compara el elemento que está en esa posición con el dato buscado. Si son iguales, la operación se interrumpe ya que se encontró el valor deseado. En caso contrario puede suceder que sea menor o mayor. Si es el primer caso, entonces se redefine el espacio de búsqueda con el extremo izquierdo igual a la posición central más uno (se descartan todos los valores comprendidos entre la posición central y el índice 0). Si el elemento de la posición central resulta mayor que el dato buscado, entonces es el extremo derecho el que se reasigna con el valor de la posición central menos 1 (se descartan todos los valores comprendidos entre la posición central y el índice `Tam-1`). Se calcula nuevamente la posición central y se repiten estos pasos hasta encontrar el elemento o hasta que el extremo izquierdo quede mayor que el extremo derecho. Esta última condición indica que el elemento no se halla en el arreglo.

Se define la clase `Binaria`, derivada de `Búsqueda`, para representar la búsqueda binaria en **arreglos ordenados**. Esta clase tiene un sólo miembro, que es el método `Busca()`, el cual se especifica de acuerdo al algoritmo descrito. Es importante señalar que, si el dato a buscar fuera un objeto, se necesitaría que el operador **!=** estuviera sobrecargado en la clase correspondiente.

```
/* Definición de la clase encargada de realizar la búsqueda binaria en
➥un arreglo ordenado. Es una clase derivada de Busqueda y en ella se
➥especifica el método Busca(). */
template <class T>
class Binaria: public Busqueda<T>
{
   public:
      int Busca (Arreglo<T>, T);
};

/* Método que realiza la búsqueda de un elemento en un arreglo cuyos
➥valores están ordenados de manera creciente. Se parte el espacio de
➥búsqueda a la mitad y se compara el dato buscado con el valor que ocupa
➥la posición central. Si son iguales, la búsqueda termina con éxito. En
➥caso contrario se evalúa si es menor o mayor y según sea el caso se re-
➥define el extremo derecho o izquierdo respectivamente y se vuelve a
➥calcular el elemento central. Recibe como parámetros el dato a buscar y
➥el arreglo en el cual se llevará a cabo la operación. Si lo encuentra
➥da como resultado la posición, en caso contrario regresa el negativo de
➥la posición en la que debería estar, más 1. */
template <class T>
int Binaria<T>::Busca(Arreglo<T> Arre, T Dato)
{
   int Izq= 0, Der= Arre.RegresaTam(), Cen, Posic;
   Cen= (Izq + Der) / 2;

   while (Izq <= Der  && Dato != Arre.RegresaValor(Cen))
   {
      if (Dato < Arre.RegresaValor(Cen))
          Der= Cen - 1;
      else
          Izq= Cen + 1;
      Cen= (Izq + Der) / 2;
   }

   if (Izq <= Der)
      Posic= Cen;
   else
      Posic= -(Izq + 1);
   return Posic;
}
```

La figura 10.3 muestra cómo se va dividiendo el espacio de búsqueda en dos, en cada iteración. La primera vez es todo el arreglo, luego de comparar el dato buscado con el elemento que ocupa la posición central será el espacio que está a la

izquierda del central (si éste es mayor que el dato buscado) o el que está a la derecha (si fuera menor). El proceso se repite haciendo que el intervalo donde se buscará sea cada vez más pequeño.

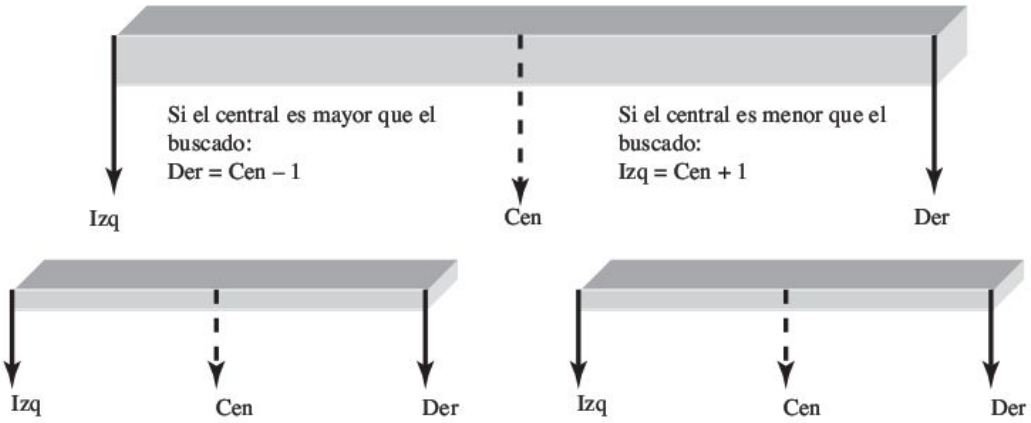


FIGURA 10.3 *División del espacio en la búsqueda binaria*

La **eficiencia** de este algoritmo también queda determinada por el número de comparaciones que se realizan antes de encontrar el elemento buscado o decidir que el mismo no fue almacenado. El caso más favorable se presenta cuando el dato está en la posición central del arreglo, lo cual requiere una sola comparación. El caso más desfavorable es cuando el dato se encuentra durante la última comparación o cuando no está en el arreglo, ya que se determina luego de realizar $\log_2(\texttt{Tam})$ comparaciones. El $\log_2$ se debe a que luego de cada comparación, el espacio de búsqueda (el número total de elementos a revisar) se reduce a la mitad. El recuadro de fórmulas 10.2 presenta las expresiones para el cálculo del número de comparaciones en las tres situaciones posibles.

Comparaciones mínimas = `1`

Comparaciones medias = $(1 + \log_2(\texttt{Tam})) / 2$

Comparaciones máximas = $\log_2(\texttt{Tam})$

FÓRMULAS 10.2

La aplicación presentada en el programa 10.2 puede modificarse para que en lugar de usar búsqueda secuencial en el arreglo ordenado, se use búsqueda binaria. En ese caso, sólo se requiere cambiar la clase con la cual se declara el objeto buscador de la siguiente manera:

```
Binaria<Persona> Buscador;
```

10.2.3 Búsqueda por transformación de claves (Hash)

Este método de búsqueda, así como los ya estudiados, está asociado a una estructura de datos. En este caso es un arreglo y generalmente se le conoce con el nombre de **tabla Hash**. Una tabla Hash permite el acceso a la información almacenada en ella de manera muy rápida. Idealmente, se espera que el tiempo de búsqueda sea independiente del número de elementos que se tengan. Sin embargo, si la misma se llena, se pierde gran parte de esta ventaja (puede requerir pasar los datos a una tabla más grande para recuperar la ventaja mencionada). Por otra parte, la desventaja es que los datos no tienen ningún orden entre sí dentro de ella.

La idea principal sobre la que se basa la inserción de elementos en esta estructura de datos y en consecuencia la operación de búsqueda en ella, consiste en transformar las claves (parte de los datos a almacenar) en direcciones dentro del arreglo. De ahí que a este método se le conozca, en el mundo de habla hispana, como **método por transformación de claves**. Por lo tanto, además de la tabla Hash se requiere tener una función (llamada función Hash) que transforme cada clave en una dirección.

El caso más simple es cuando se puede asociar cada dato directamente a una posición del arreglo. Por ejemplo, suponga que se tiene un arreglo en el cual se guardan los datos de 300 empleados y el número que identifica al empleado (su clave) es un número entero del 0 al 299. Esta situación se ilustra en la figura 10.4. En este caso, la asignación de cada clave a una posición diferente del arreglo es inmediata. Es decir, la función Hash obtiene como dirección la misma clave que es un entero comprendido entre 0 y 299.

dirección = Φ(Clave) = Clave;

En este ejemplo, si se quisiera aumentar el sueldo al empleado Eduardo Vargas, sólo se requiere hacer:

```
Empleados[1] = Empleados[1].NuevoSueldo(Cantidad);
```

asumiendo que el arreglo almacena objetos de la clase `Empleado` y que en dicha clase hay un método que permite actualizar el atributo `Sueldo`.

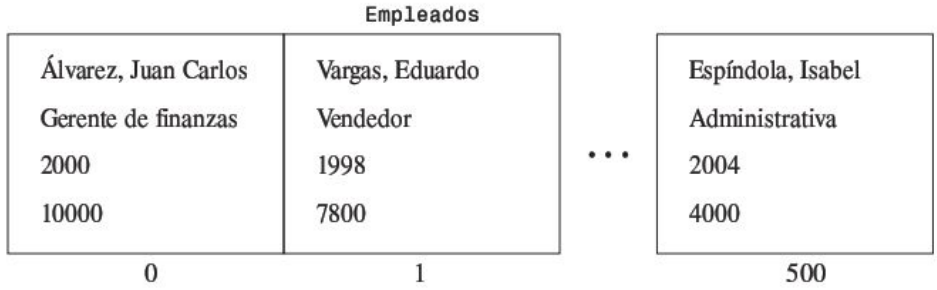


FIGURA 10.4 *Ejemplo de tabla Hash*

Para realizar otras operaciones se tendría la misma rapidez, ya que el acceso a los datos es directo. Sin embargo, hay muchas aplicaciones en las que esta manera de usar la tabla Hash no es posible, ya sea por el volumen de información que se maneja o por las características de la misma. Por lo tanto, en estos casos sí se requiere usar una función que **transforme** la clave en una dirección.

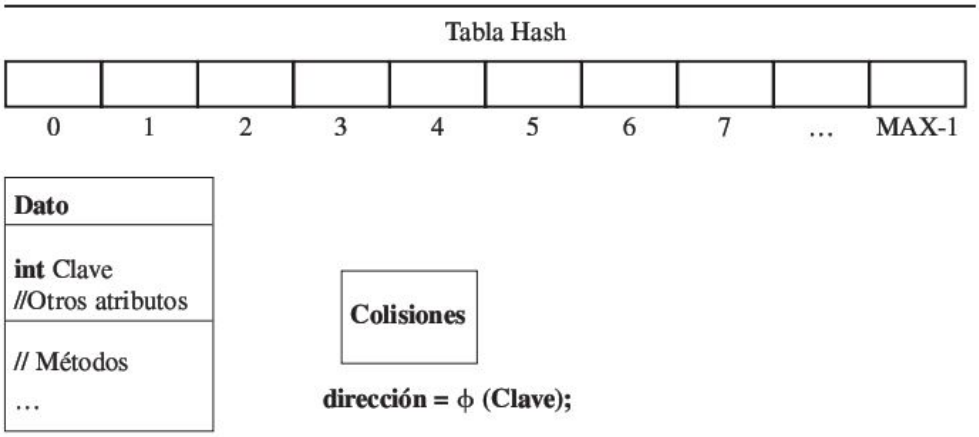
La función debe definirse de tal manera que sea fácil de calcular y que distribuya uniformemente los diferentes elementos en el arreglo. Cuando una función genera la misma dirección para dos datos distintos, se produce lo que se conoce como **colisión**. Es decir, se intenta guardar un dato en una posición que ya fue ocupada previamente por otro elemento.

$$\textbf{dirección1} = \Phi(\textbf{Clave1});$$

$$\textbf{dirección2} = \Phi(\textbf{Clave2});$$

en donde: Clave1 $\neq$ Clave2 y dirección1 = dirección2.

En consecuencia, resulta necesario definir junto a la función una manera de tratar las colisiones. La figura 10.5 contiene todos los elementos que intervienen: la tabla Hash, el dato que se va a almacenar, la función Hash y la solución de colisiones.



Donde:

Φ es una función Hash que se aplica a la clave del dato que se quiere guardar en el arreglo. **dirección** es la dirección obtenida a partir de la clave y será un valor comprendido entre 0 y el MAX ($0 \leq d < MAX$).

FIGURA 10.5 *Elementos que intervienen en el método Hash*

Antes de presentar algunas funciones y procesos para el manejo de las mismas, se ofrece una forma de tratar las claves no numéricas.

Claves no numéricas

Cuando se tienen claves alfabéticas o alfanuméricas se deben convertir primero a numéricas para luego ser transformadas por la función Hash en una dirección dentro del arreglo. Generalmente se le asigna a cada letra un valor numérico consecutivo, de acuerdo al alfabeto que se esté usando. Así, se genera la siguiente tabla de equivalencias para las letras del castellano.

TABLA 10.2 Equivalencia entre letras y números

Letra	*Valor numérico*
a	1
b	2
c	3

continúa

TABLA 10.2 Continuación

Letra	*Valor numérico*
d	4
e	5
f	6
g	7
h	8
i	9
…	…
z	27

Si, por ejemplo, la clave fuera el registro de contribuyentes de una persona, formado por letras y dígitos, por ejemplo: **feac701123di5**, quedaría como: **6513701123495** al hacer uso de las equivalencias entre letras y dígitos ya señaladas.

f= 6

e= 5

a= 1

c= 3

d= 4

i= 9

A continuación se presentan las funciones Hash más usadas y posteriormente algunas alternativas para el manejo de colisiones.

Funciones Hash

Una **función Hash** es una función que dado un dato (o parte de él) genera una dirección. Como se trabaja en memoria principal y con arreglos, la dirección es una posición del mismo. En esta sección se estudiarán algunos ejemplos de las funciones Hash:

- Residuo o módulo
- Plegamiento
- Truncamiento

1. Función residuo o módulo

Se debe tomar el residuo que se obtiene de la división de la clave (debe ser numérica) entre el tamaño del arreglo, el cual será un valor comprendido entre 0 y el máximo menos uno. Observe los siguientes ejemplos:

MAXIMO: 300 (el arreglo tiene una capacidad máxima de 300 elementos).

Clave: 6513701123495
dirección = Φ(6513701123495) = 6513701123495 % 300 = **95**

El dato cuya clave es 6513701123495 se almacenará en la casilla del arreglo identificada por el índice 95.

Clave: 2318212217655
dirección = Φ(2318212217655) = 2318212217655 % 300 = **255**

El dato cuya clave es 2318212217655 se almacenará en la casilla del arreglo identificada por el índice 255.

En este capítulo se usa esta función para todos los ejemplos y para la implementación en ***C++*** que aparece más adelante.

2. Plegamiento

Se deben generar dos o más números a partir de los dígitos de la clave (debe ser numérica) sumarlos para obtener un único número del cual se toman los dígitos menos significativos como dirección. Observe los siguientes ejemplos:

MAXIMO: 300 (el arreglo tiene una capacidad máxima de 300 elementos).

Clave: 6513701123495
dirección = Φ(6513701123495)
dirección = dígitos menos significativos de la suma de los números formados con Clave
dirección = dígitos menos significativos de (651 + 370 + 112 + 349 + 5)
dirección = dígitos menos significativos de (1487) = **87**

El dato cuya clave es 6513701123495 se almacenará en la casilla del arreglo identificada por el índice 87.

Clave: 2318212217655
dirección = Φ(2318212217655)
dirección = dígitos menos significativos de la suma de los números formados con Clave
dirección = dígitos menos significativos de (231 + 821 + 221 + 765 + 5)
dirección = dígitos menos significativos de (2043) = **43**

El dato cuya clave es 2318212217655 se almacenará en la casilla del arreglo identificada por el índice 43.

3. Truncamiento

Se deben elegir algunos dígitos de la clave (debe ser numérica) y formar con ellos la dirección. El criterio de elección se determina en cada aplicación. Por ejemplo, se pueden elegir los primeros ene dígitos o los últimos ene dígitos, los que ocupan posiciones pares o los que ocupan posiciones impares. Dependiendo del tamaño de la clave se deben sumar los dígitos y tomar los menos significativos. Para los ejemplos que se presentan a continuación, se eligen los dígitos que están en las posiciones pares del número, empezando a contar de izquierda a derecha.

MAXIMO: 300 (el arreglo tiene una capacidad máxima de 300 elementos).

Clave: 6513701123495
dirección = Φ(6513701123495)
dirección = suma de dígitos que ocupan posiciones pares
dirección = 5 + 3 + 0 + 1 + 3 + 9 = **21**

El dato cuya clave es 6513701123495 se almacenará en la casilla del arreglo identificada por el índice 21.

Clave: 2318212217655
dirección = Φ(2318212217655)
dirección = suma de dígitos que ocupan posiciones pares
dirección = 3 + 8 + 1 + 2 + 7 + 5 = **26**

El dato cuya clave es 2318212217655 se almacenará en la casilla del arreglo identificada por el índice 26.

En todos los ejemplos analizados, a claves diferentes se le asignaron direcciones diferentes. Sin embargo, no siempre resulta así. Cuando la función Hash elegida transforma dos claves distintas en una misma dirección se genera una colisión y se debe proveer algún mecanismo para resolver esta situación, de tal manera que se pueda almacenar el nuevo dato aunque la dirección asignada ya esté ocupada.

Solución de colisiones

Se llama **colisión** a la situación generada cuando una función Hash asigna una misma dirección a dos claves distintas. Es decir, al intentar almacenar un dato en la dirección correspondiente se detecta que la misma ya fue previamente ocupada por otro elemento. Observe el ejemplo que se muestra más adelante. En este caso, si se diera el dato 6513701123495, la función Hash le asignaría la posición 95 del arreglo. Si posteriormente se tuviera un empleado con registro de contribuyentes igual a 6513701123195 (luego de la conversión a numérico), la función Hash también daría la posición 95 del arreglo.

Clave: 6513701123495
dirección = Φ(6513701123495) = 6513701123495 % 300 = **95**

Clave: 6513701123195
dirección = Φ(6513701123195) = 6513701123195 % 300 = **95**

La manera de generar las direcciones y en su caso resolver las colisiones al momento de insertar, determina la manera de generar las direcciones y resolver las colisiones al momento de buscar un dato dentro de la tabla Hash. Las cuatro maneras más utilizadas para resolver colisiones:

- Prueba lineal,
- Prueba cuadrática
- Doble dirección
- Encadenamiento

Primero se analizarán las tres primeras porque comparten ciertas características, mientras que la cuarta será tratada de manera independiente porque requiere el uso de listas ligadas junto con la tabla Hash.

Prueba lineal

La **prueba lineal** consiste en que una vez detectada la colisión se busca secuencialmente en el arreglo hasta encontrar un espacio disponible (en caso de una inserción) o encontrar el dato (en caso de una búsqueda) o bien hasta que se detecta que el arreglo está lleno o que ya fue recorrido totalmente. Por lo tanto, si la dirección generada por la función Hash es d y ésta ya fue ocupada, entonces se buscará en la d+1, luego en la d+2, y así hasta encontrar un lugar disponible o hasta llegar nuevamente a d. En este último caso, el proceso se detiene para no caer en ciclos infinitos.

La figura 10.6 presenta un esquema de cómo quedaría la tabla Hash al resolver la colisión que se presentó al insertar el valor 6513701123195; dicha colisión se presentó debido a que la dirección asignada (d=95) ya había sido ocupada previamente. En este caso, asumiendo que la posición 96 (d+1) estuviera libre, ésta es la elegida para almacenar el nuevo valor. Si posteriormente se busca el dato cuya clave es 6513701123195, la función Hash dará la dirección 95. Al buscar en dicha posición, el dato no se encontrará, por lo que se aplicará prueba lineal hasta encontrarlo o hasta haber recorrido todo el arreglo. Para nuestro ejemplo, se encontrará en la siguiente posición, es decir en la 96.

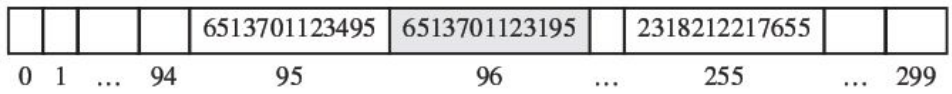


Figura 10.6 *Solución de colisiones por prueba lineal*

Prueba cuadrática

La **prueba cuadrática** consiste en que una vez detectada la colisión se redefine la dirección incrementándola con el cuadrado de un valor, el cual inicia en uno y se aumenta en uno en cada nuevo intento. El proceso se repite hasta encontrar un espacio disponible (en caso de una inserción) o encontrar el dato (en caso de una búsqueda) o bien hasta que se detecta que el arreglo está lleno o que el dato no está en el arreglo. Por lo tanto, si la dirección generada por la función Hash es d y ésta ya fue ocupada, entonces se buscará en la $d+1^2$, luego en la $d+2^2$, y así hasta encontrar un lugar disponible o hasta cumplir alguna condición establecida para evitar caer en ciclos infinitos.

La figura 10.7 presenta un esquema de cómo quedaría la tabla Hash al resolver la colisión que se presentó al insertar el valor 6513701123195; dicha colisión se presentó debido a que la dirección asignada (d=95) ya había sido ocupada previamente. En este caso, asumiendo que la posición 96 ($d+1^2$) está también ocupada, se vuelve a calcular una nueva dirección por medio de la prueba cuadrática, resultando igual a 99 ($d+2^2$). Considerando que está disponible, es la elegida para almacenar el dato. Si posteriormente se busca el dato cuya clave es 6513701123195, la función Hash dará la dirección 95. Al buscar en dicha posición, el dato no se encontrará, por lo que se aplicará la prueba cuadrática hasta encontrarlo o hasta que se cumpla la condición establecida. Para nuestro ejemplo, se encontrará en el segundo intento, es decir en la 99.

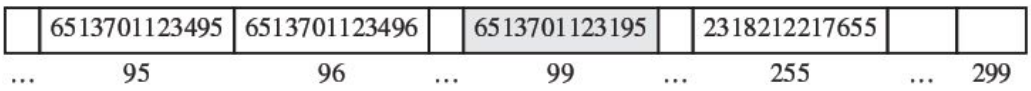


FIGURA 10.7 *Solución de colisiones por prueba cuadrática*

Doble dirección

La **doble dirección** consiste en que una vez detectada la colisión se redefine la dirección incrementándola en uno y aplicándole nuevamente la función Hash. Es decir, la dirección que se obtuvo con la función se convierte en entrada de la misma función. Por lo tanto, con la doble dirección se obtiene una dirección de una dirección. El proceso se repite hasta encontrar un espacio disponible (en caso de una inserción) o encontrar el dato (en caso de una búsqueda), o bien hasta que se detecta que el arreglo está lleno o que el dato no está en el arreglo. Por lo tanto, si la dirección generada por la función Hash es d y ésta ya fue ocupada, entonces

se buscará en la $d_1 = \Phi(d+1)$, luego en la $d_2 = \Phi(d_1+1)$, y así hasta encontrar un lugar disponible o hasta cumplir alguna condición establecida para evitar caer en ciclos infinitos.

La figura 10.8 presenta un esquema de cómo quedaría la tabla Hash al resolver la colisión que se presentó al insertar el valor 6513701123195, dado que la dirección asignada (d=95) ya había sido ocupada previamente. En este caso, asumiendo que la posición 96 ($d_1 = (d+1)$ % 300) está también ocupada, se vuelve a calcular una nueva dirección por medio de la doble dirección, resultando igual a 97 ($d_2 = (d_1+1)$ % 300). Considerando que está disponible, es la elegida para almacenar el dato. Si posteriormente se busca el dato cuya clave es 6513701123195, la función Hash dará la dirección 95. Al buscar en dicha posición, el dato no se encontrará, por lo que se aplicará doble dirección hasta encontrarlo o hasta que se cumpla la condición establecida. Para nuestro ejemplo, se encontrará en el segundo intento, es decir en la 97.

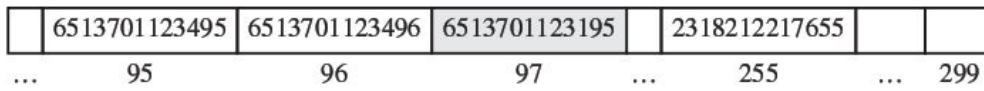


FIGURA 10.8 *Solución de colisiones por doble dirección*

Implementación del algoritmo de búsqueda por transformación de claves

La figura 10.9 presenta un esquema de las clases definidas para representar estas variantes de la solución de colisiones. Se definió una plantilla de una clase abstracta, la clase Hash, que tiene como atributos la tabla Hash y el total de elementos almacenados. Además, tiene como miembros cuatro métodos, dos de los cuales son virtuales puros y se redefinirán en las clases derivadas. Los otros dos son auxiliares para determinar si la tabla está vacía y para imprimir el contenido de la misma respectivamente.

A partir de la clase abstracta se derivan tres clases concretas para implementar la solución de colisiones por prueba lineal (clase `PruebaLineal`), por prueba cuadrática (clase `PruebaCuadratica`) y por doble dirección Hash (clase `DobleDireccion`). Las tres clases redefinen los métodos `Inserta()` y `Busca()` heredados de la clase base, de acuerdo a sus propias características. La función Hash utilizada en todas las clases es la del `módulo` o `residuo`, la cual por su simplicidad se codifica tanto

en `Inserta()` como en `Busca()`. Sin embargo, la misma podría definirse como un método de la clase abstracta que se heredaría por todas las derivadas.

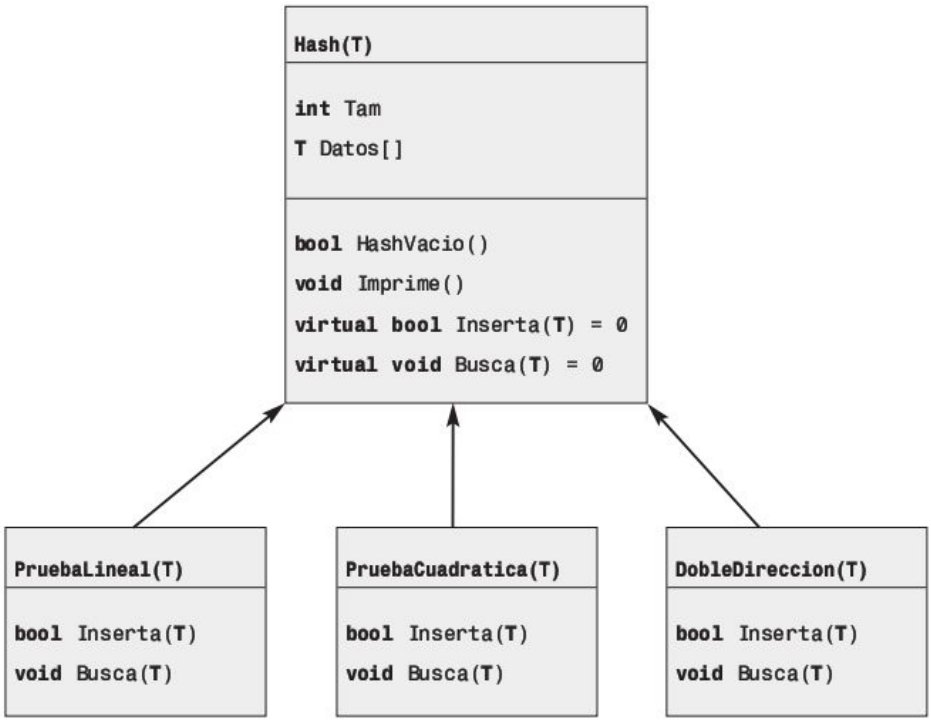


FIGURA 10.9 *Esquema de clases*

A continuación se muestra la codificación de estas clases usando el lenguaje ***C++***.

```
// Definición del número máximo de elementos que puede contener el
➥arreglo.
#define MAXIMO 20

/* Definición de la plantilla de la clase Hash.  La clase tiene como
➥miembros protegidos un arreglo de tipo T para darle mayor generalidad
➥a la solución y el total de elementos almacenados. Además, tiene dos
➥métodos que implementan operaciones comunes a todas las clases
➥derivadas y dos métodos virtuales puros que serán implementados de
➥manera específica en cada una de las subclases. */
```

```
template <class T>
class Hash
{
   protected:
      int Tam;
      T Datos[MAXIMO];
   public:
      Hash();
      bool HashVacio();
      void  Imprime();
      virtual bool Inserta(T) = 0;
      virtual void  Busca(T) = 0;
};

/* Método constructor. Inicializa los atributos Datos y Tam para indicar
➥que el arreglo está vacío. */
template <class T>
Hash<T>::Hash()
{
   int Indice;
   for (Indice= 0; Indice < MAXIMO; Indice++)
        Datos[Indice]= NULL;
   Tam= 0;
}

/* Método auxiliar para determinar el estado del arreglo. Regresa true
➥si el arreglo está vacío, y false en caso contrario. */
template <class T>
bool Hash<T>::HashVacio()
{
   if (Tam == 0)
       return true;
   else
       return false;
}

/* Imprime los elementos almacenados en cada una de las posiciones del
➥arreglo. */
template <class T>
void  Hash<T>::Imprime()
{
   int Indice;

   /* Verifica que el arreglo tenga al menos un elemento. */
   if (!HashVacio())
   {
        cout<<"\n\n        Datos almacenados\n\n ";
        for (Indice= 0; Indice < MAXIMO; Indice++)
             if (Datos[Indice] !=  NULL)
```

```
                cout<<"Posición "<<(Indice+1)<<":     "
                    <<Datos[Indice]<<endl;
          cout<<"\n\n";
    }
    else
          cout<<"\nNo hay elementos almacenados en el arreglo. \n";
}


/* Clase PruebaLineal, derivada de la clase Hash. Implementa el método
➥Hash resolviendo las colisiones por medio de la prueba lineal. Utiliza
➥la función Hash módulo o residuo. */
template <class T>
class PruebaLineal: public Hash<T>
{
   public:
      bool Inserta(T);
      void  Busca(T);
};

/* Método para insertar un nuevo elemento en un arreglo. Resuelve las
➥colisiones por medio de la prueba lineal. */
template <class T>
bool PruebaLineal<T>::Inserta(T Valor)
{
   int Ind1, Ind2;
   bool Resp= true;

   Ind1= Valor % MAXIMO;
   if (Datos[Ind1] == NULL)
        Datos[Ind1]= Valor;
   else
   {
        Ind2= Ind1 + 1;
        while (Ind2 < MAXIMO && Datos[Ind2]!= NULL && Ind2 != Ind1)
        {
           Ind2++;
           if (Ind2 == MAXIMO)
               Ind2= 0;
        }
        if (Ind2 == Ind1)
            Resp= false;
        else
            Datos[Ind2]= Valor;
   }
   if (Resp)
        Tam++;
   return Resp;
}
```

```
/* Método para buscar el elemento Valor en el arreglo. Resuelve las
➥colisiones por medio de la prueba lineal. */
template <class T>
void  PruebaLineal<T>::Busca(T Valor)
{
   int Ind1, Ind2;

   /* Verifica que el arreglo tenga al menos un elemento. */
   if (!HashVacio())
   {
        Ind1= (Valor % MAXIMO);
        if (Datos[Ind1] == Valor)
            cout<<"\nEl elemento está en la posición: "<<(Ind1+1)<<endl;
        else
        {
            Ind2= Ind1 + 1;
            while (Ind2 < MAXIMO && Datos[Ind2] != Valor
                  ➥&& Datos[Ind2]!= NULL && Ind2 != Ind1)
            {
               Ind2++;
               if (Ind2 == MAXIMO)
                   Ind2= 0;
            }
            if (Datos[Ind2] == Valor)
                cout<<"\nEl elemento está en la posición: "
                    ➥<<(Ind2+1)<<endl;
            else
                cout<<"\nEl elemento no está en el arreglo. \n"<<endl;
        }
   }
   else
        cout<<"\nNo hay elementos almacenados en el arreglo. \n";
}


/* Clase PruebaCuadratica, derivada de la clase Hash. Implementa el
➥método Hash resolviendo las colisiones por medio de la prueba
➥cuadrática. Utiliza la función Hash módulo o residuo. */
template <class T>
class PruebaCuadratica: public Hash<T>
{
   public:
       bool Inserta(T);
       void  Busca(T);
};
```

```
/* Método para insertar un nuevo elemento en un arreglo. Resuelve las
➥colisiones por medio de la prueba cuadrática. */
template <class T>
bool PruebaCuadratica<T>::Inserta(T Valor)
{
   bool Resp= true;
   int Base, Ind1, Ind2, Bandera= 1;
   /* Bandera: establece la condición necesaria para evitar ciclos
   ➥infinitos.
   Puede tomar tres posibles valores:
      ➥1 indica el primer intento de inserción
      ➥2 indica el segundo intento de inserción
      ➥3 indica el tercer intento de inserción. En este caso se
        ➥interrumpe el proceso. */

   Ind1 = Valor % MAXIMO;
   if (Datos[Ind1] == NULL)
        Datos[Ind1]= Valor;
   else
   {
        Base= 1;
        Ind2= int (Ind1 + pow(Base,2));
        Base++;

        while (Ind2 < MAXIMO && Datos[Ind2] != NULL && Bandera != 3)
        {
           if (Bandera == 2)
           {
               Ind2= int (Ind2 + pow(Base,2));
               Base++;
           }
           else
           {
               Ind2= int (Ind1 + pow(Base,2));
               Base++;
           }

           if (Ind2 >= MAXIMO)
               if (Bandera == 2)
                    Bandera= 3;
               else
               {
                    Base= 1;
                    Ind2= 0;
                    Bandera= 2;
               }
        }
```

```
            if (Bandera == 3)
                Resp= false;
            else
                Datos[Ind2]= Valor;
    }
    if (Resp)
        Tam++;
    return Resp;
}

/* Método para buscar el elemento Valor. Resuelve el problema de las
➥colisiones por medio de la prueba cuadrática. */
template <class T>
void  PruebaCuadratica<T>::Busca(T Valor)
{
    int Base, Ind1, Ind2;

    /* Verifica que el arreglo almacene al menos un elemento. */
    if (!HashVacio())
    {
        Ind1= (Valor % MAXIMO);
        if (Datos[Ind1] == Valor)
            cout<<"\nEl elemento está en la posición: "<<(Ind1+1)<<endl;
        else
        {
            Base= 1;
            Ind2= int (Ind1 + pow(Base,2));
            while (Datos[Ind2] != Valor && Datos[Ind2] != NULL)
            {
                Base++;
                Ind2= int (Ind1 + pow(Base,2));
                if (Ind2 >= MAXIMO)
                {
                    Base= 0;
                    Ind1= 0;
                    Ind2= 0;
                }
            }
            if (Datos[Ind2] == Valor)
                cout<<"\nEl elemento está en la posición:
                    ➥"<<(Ind2+1)<<endl;
            else
                cout<<"\nEl elemento no está en el arreglo. \n";
        }
    }
    else
        cout<<"\nNo hay elementos almacenados en el arreglo. \n";
}
```

```
/* Clase DobleDireccion, derivada de la clase Hash. Implementa el método
➥Hash resolviendo las colisiones por medio de dobles direcciones.
➥Utiliza la función Hash módulo o residuo. */
template <class T>
class DobleDireccion: public Hash<T>
{
   public:
       bool Inserta(T);
       void  Busca(T);
};

/* Método para insertar un nuevo elemento en un arreglo. Resuelve las
➥colisiones por medio de la doble dirección Hash. */
template <class T>
bool DobleDireccion<T>::Inserta(T Valor)
{
   bool Resp= true;
   int Ind1, Ind2, Bandera= 1;
   /* Bandera: establece la condición necesaria para evitar ciclos
   ➥infinitos. Puede tomar tres posibles valores:
      ➥1 indica el primer intento de inserción
      ➥2 indica el segundo intento de inserción
      ➥3 indica el tercer intento de inserción. En este caso se
       ➥interrumpe el proceso. */

   Ind1= Valor % MAXIMO;
   if (Datos[Ind1] == NULL)
        Datos[Ind1]= Valor;
   else
   {
       Ind2 = ((Ind1+1) % MAXIMO) + 1;
       while (Ind2 < MAXIMO && Datos[Ind2] != NULL && Bandera != 3)
       {
          Ind2= ((Ind2+1) % MAXIMO) + 1;
          if (Ind2 >= MAXIMO)
              if (Bandera == 2)
                   Bandera= 3;
              else
              {
                   Ind2= 0;
                   Bandera= 2;
              }
       }
       if (Bandera == 3)
           Resp= false;
       else
           Datos[Ind2]= Valor;
   }
```

```
    if (Resp)
        Tam++;
    return Resp;
}

/* Método para buscar el elemento Valor. Resuelve el problema de las
➥colisiones por medio de la doble dirección Hash. */
template <class T>
void  DobleDireccion<T>::Busca(T Valor)
{
    int Ind1, Ind2;

    /* Verifica que el arreglo almacene al menos un elemento. */
    if (!HashVacio())
    {
        Ind1= (Valor % MAXIMO);
        if (Datos[Ind1] == Valor)
            cout<<"\nEl elemento está en la posición: "<<(Ind1+1)<<endl;
        else
        {
            Ind2= ((Ind1+1) % MAXIMO) + 1;
            while (Ind2 < MAXIMO && Datos[Ind2] != Valor &&
                ➥Datos[Ind2]!= NULL && Ind2 != Ind1)
                Ind2= ((Ind2 + 1) % MAXIMO) + 1;

            if (Datos[Ind2] == Valor)
                cout<<"\nEl elemento está en la posición:
                    ➥"<<(Ind2+1)<<endl;
            else
                cout<<"\nEl elemento no está en el arreglo. \n";
        }
    }
    else
        cout<<"\nNo hay elementos almacenados en el arreglo. \n"<<endl;
}
```

El método `Busca()` se definió del tipo **`Void`** (en las implementaciones), de tal manera que el método imprime si encuentra o no al elemento buscado. También se pudo definir de tipo entero y, en este caso, regresar la posición donde está, o un valor negativo si no lo encuentra en el arreglo. Otra posible variante es definirlo de tipo **`Bool`**, como `Inserta()`, dando el valor **`True`** si está y **`False`** en caso contrario.

Es importante aclarar que si el tipo usado para darle valor a **`T`** fuera una clase, entonces en dicha clase deberían estar sobrecargados los operadores **==** y **!=** para que los métodos pudieran usarse tal como están implementados.

El programa 10.3 presenta una aplicación para mostrar el uso del algoritmo por transformación de claves, usando la prueba lineal para el manejo de colisiones. El

uso de las clases derivadas que implementan las otras variantes es similar al presentado en este programa.

Programa 10.3

```
/* Aplicación. Se incluye la biblioteca en la cual se guardó la clase
➥Hash y su derivada PruebaLineal. */
#include "Hash.h"

void  main()
{
    int Total, I, Cod;
    /* Se declara un objeto de la clase PruebaLineal, usando el tipo int
    ➥para definir el tipo de datos que se almacenarán en la tabla Hash. */
    PruebaLineal<int> HashLin;

    do {
         cout<<"\nCuántos códigos de productos quieres guardar:  ";
         cin>>Total;
    } while (Total <= 0);

    /* Lectura de los datos y almacenamiento de los mismos en la tabla
    ➥Hash por medio del método Inserta(). Si se presenta alguna colisión en
    ➥el momento de insertar, se resuelve por medio de la prueba lineal. */
    for (I= 1; I <= Total; I++)
    {
         cout<<"\n\nIngresa un código:  ";
         cin>>Cod;
         HashLin.Inserta(Cod);
    }

    /* Por medio de este ciclo se permite que el usuario verifique qué
    ➥códigos fueron dados de alta en el arreglo. Se ingresa un código y
    ➥se invoca al método que busca un elemento en la tabla Hash. Según
    ➥el resultado obtenido se da un mensaje adecuado. */
    cout<<"\n\nCódigo a  buscar (para terminar -1):  ";
    cin>>Cod;
    while (Cod != -1)
    {
         HashLin.Busca(Cod);
         cout<<"\n\nCódigo a buscar (para terminar -1):  ";
         cin>>Cod;
    }

    /* Se imprime la información de todos los códigos guardados en la
    ➥tabla Hash. */
    cout<<"\n\n\nReporte de todos los códigos almacenados\n  ";
    HashLin.Imprime();
}
```

Encadenamiento

Además de las tres formas ya estudiadas para el manejo de las colisiones, existe otra que es muy útil y fácil de usar, pero que requiere utilizar listas ligadas además del arreglo. Es decir, en este caso la tabla Hash se ve como un arreglo de listas. De ahí el nombre que recibe esta manera de solucionar las colisiones: por encadenamiento.

El **encadenamiento** se da por la misma naturaleza de la estructura de datos elegida para almacenar la información. Dado que cada elemento del arreglo es una lista, si se tiene una colisión el dato colisionado también se guarda en la lista de la casilla asignada. Es decir, la dirección que se obtuvo con la función Hash sigue siendo la misma; sin embargo, los datos se van guardando en la lista ligada, la cual por ser implementada en memoria dinámica no tiene (en principio) límite establecido. Por lo tanto, si la dirección generada por la función Hash es d, sin importar si la misma ya fue ocupada o no se almacena el dato en la lista que está en la posición d del arreglo.

La figura 10.10 presenta un esquema de cómo se va generando la tabla Hash, usando encadenamiento, al insertar el valor 6513701123195 en la dirección obtenida por la función (d=95), la cual fue previamente asignada a otro dato (6513701123495). En este caso, como en cada posición de la tabla se tiene una lista no existe problema de espacio para insertarlo en la misma. Es decir, una vez generada la dirección se invoca el método que inserta elementos a la lista almacenada en dicha posición. Cuando se busca un elemento se procede de manera similar. Se calcula la dirección Hash y luego se realiza la búsqueda en la lista guardada en dicha posición.

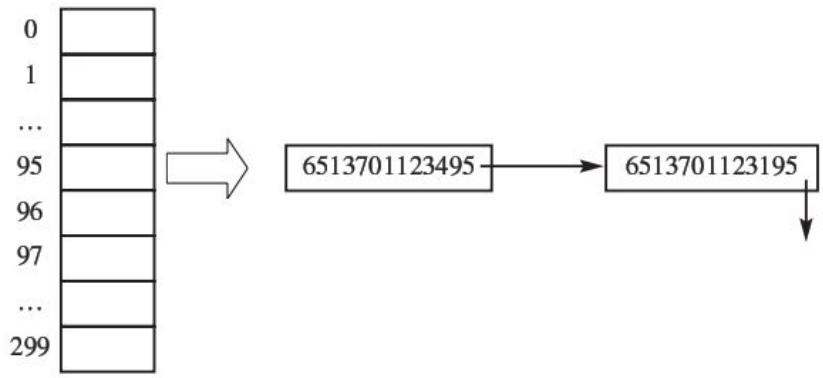


Figura 10.10 *Manejo de colisiones por encadenamiento*

La figura 10.11 presenta el esquema de clases para implementar este algoritmo. Se modificó la clase `Hash` de la figura 10.9, declarando el atributo `Datos[]` del tipo de la plantilla `Lista`. Además, el método `Busca()` de definió de tipo **`Bool`**.

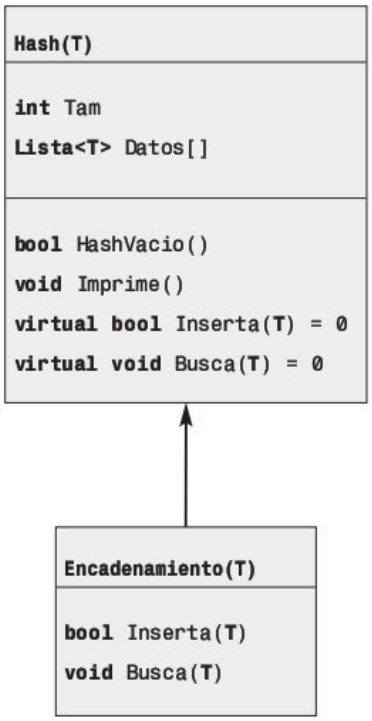


FIGURA 10.11 *Esquema de clases*

A continuación se presenta la implementación de estas clases, usando el lenguaje ***C++***. Es importante señalar, que si en lugar de usar la plantilla de la clase `Lista` se hubiera usado una lista con un tipo ya asociado, se hubiera podido derivar de la clase `Hash` previamente estudiada.

```
// Definición del número máximo de elementos que puede contener el arreglo.
#define MAXIMO 20

// Se incluye la biblioteca con la plantilla de la clase Lista.
#include "Lista.h"
```

```
/* Definición de la clase Hash, en la cual el manejo de las colisiones se
➥hará por medio del encadenamiento. La función Hash es el módulo o residuo.
➥Se apoya en la clase Lista, la cual está en la biblioteca Lista.h */
template <class T>
class Hash
{
   protected:
      int Tam;
      Lista<T> Datos[MAXIMO];
   public:
      Hash();
      bool HashVacio();
      void  Imprime();
      virtual bool Inserta(T) = 0;
      virtual bool Busca(T) = 0;
};

/* Método constructor. Inicializa los atributos Datos y Tam para
➥indicar que el arreglo está vacío. En este caso se tiene un arreglo
➥de listas. */
template <class T>
Hash<T>::Hash()
{
   int Indice;
   for(Indice= 0; Indice < MAXIMO; Indice++)
       Datos[Indice]= NULL;
   Tam= 0;
}

/* Método auxiliar para determinar el estado del arreglo. Regresa true
➥si el arreglo está vacío, y false en caso contrario. */
template <class T>
bool Hash<T>::HashVacio()
{
   if (Tam == 0)
       return true;
   else
       return false;
}

/* Imprime los elementos almacenados en el arreglo. Para que este
➥método pueda imprimir el contenido del arreglo, teniendo en cuenta que
➥cada uno de ellos es una lista, se requiere que en la clase Lista se
➥haya sobrecargado el operador <<.   */
template <class T>
void  Hash<T>::Imprime()
{
   int Ind;
```

```
   /* Verifica que el arreglo tenga al menos un elemento. */
   if (!HashVacio())
   {
            cout<<"\n\n        Datos almacenados\n\n ";
            for (Ind= 0; Ind < MAXIMO; Ind++)
                  if (Datos[Ind] != NULL)
                        cout<<"Posición "<<(Ind+1)<<":
                            "<<Datos[Ind]<<endl;
                  cout<<"\n\n";
   }
   else
            cout<<"\nNo hay elementos almacenados en el arreglo. \n";
}

/* Clase Encadenamiento, derivada de la clase Hash. Implementa el método
➥Hash resolviendo las colisiones por medio de encadenamiento. Utiliza
➥la función Hash módulo o residuo. Se apoya en una lista simplemente
ligada.  */
template <class T>
class Encadenamiento: public Hash<T>
{
   public:
       bool Inserta(T);
       bool Busca(T);
};

/* Método para insertar un nuevo elemento en un arreglo. Resuelve las
colisiones por medio de encadenamiento. */
template <class T>
bool Encadenamiento<T>::Inserta(T Valor)
{
   int Ind1;
   bool InserLis;

   Ind1= Valor % MAXIMO;
   InserLis= Datos[Ind1].InsertaFinal(Valor);
   if (InserLis)
       Tam++;
   return InserLis;
}

/* Método para buscar el elemento Valor. Resuelve el problema de las
➥colisiones por medio de encadenamiento, haciendo uso de una lista
➥simplemente ligada.   */
template <class T>
bool Encadenamiento<T>::Busca(T Valor)
{
   int Ind1;
   bool Resp= false;
```

```
    if (!HashVacio())
    {
         Ind1= (Valor % MAXIMO);
         if (Datos[Ind1].BuscaDesordenada(Valor) != NULL)
             Resp= true;
    }
    return Resp;
}
```

A continuación se presenta la plantilla de la clase `Lista` con los operadores sobrecargados requeridos por la clase `Encadenamiento`.

```
// Clase Lista dependiente de la clase NodoLista.
template <class T>
class Lista;

/* Definición de la clase NodoLista. */
template <class T>
class NodoLista
{
   public:
      NodoLista<T> *Liga;
      T Info;
      NodoLista();
      friend  class Lista<T>;
      // Otros métodos presentados en el capítulo 6, dedicado a listas
      ➥ligadas.
};

/* Declaración del método constructor por omisión. Inicializa con el
➥valor NULL al puntero al siguiente nodo. */
template <class T>
NodoLista<T>::NodoLista()
{
   Liga = NULL;
}


/* Definición de la clase Lista. */
template <class T>
class Lista
{
   private:
      NodoLista<T> *Primero;
```

```
   public:
      Lista ();
      bool InsertaFinal(T);
      NodoLista<T> * BuscaDesordenada(T);
      void  operator = (const);
      bool operator != (const);
      friend  ostream &operator <<  (ostream &, Lista &);
      // Otros métodos estudiados en el capítulo 6, dedicado a las
      ➥listas ligadas.
};

/* Declaración del método constructor. Inicializa el puntero al primer
➥nodo de la lista con el valor NULL: indica lista vacía. */
template <class T>
Lista<T>::Lista()
{
   Primero = NULL;
}

/* Sobrecarga del operador = para que a un objeto de la clase Lista se
➥le pueda asignar la constante NULL, operación necesaria cuando se
➥inicializa el arreglo de listas. */
template <class T>
void  Lista<T>::operator = (const)
{
   Primero = NULL;
}

/* Sobrecarga del operador != para que un objeto de la clase Lista pueda
➥ser comparado con la constante NULL, operación necesaria cuando se
➥intenta determinar si una casilla del arreglo está disponible. */
template <class T>
bool Lista<T>::operator != (const)
{
   return (Primero != NULL);
}

/* Sobrecarga del operador << para que un objeto tipo Lista pueda ser
➥impreso directamente. De esta manera, el método Imprime() de la clase
➥Hash se generaliza a cualquiera de las formas vistas para tratar las
➥colisiones. */
template <class T>
ostream &operator << (ostream &Escribe, Lista<T> &ObjLis)
{
   NodoLista<T> *P = ObjLis.Primero;
   while (P)
   {
      Escribe<<"\n"<<P->Info;
      P= P->Liga;
   }
```

```
   Escribe<<"\n\n";
   return  Escribe;
}

/* Método que inserta un nodo al final de la lista. El método es válido
➥tanto para listas ya creadas como para listas vacías. */
template <class T>
bool Lista<T>::InsertaFinal(T Dato)
{
   NodoLista<T> *P, *Ultimo;

   P = new NodoLista<T>();
   if (P)
   {
        P->Info= Dato;
        if (Primero)
        {
            Ultimo = Primero;
            while (Ultimo->Liga)
               Ultimo = Ultimo->Liga;
            Ultimo->Liga = P;
        }
        else
            Primero = P;
        return true;
   }
   else
        return false;
}

/* Método que busca un elemento dado como referencia en una lista
➥desordenada. Regresa la dirección del nodo si lo encuentra o NULL en
➥caso contrario. */
template <class T>
NodoLista<T> * Lista<T>::BuscaDesordenada(T Ref)
{
   NodoLista<T> *Q, *Resp= NULL;
   if (Primero)
   {
        Q = Primero;
        while ((Q->Info != Ref)  && (Q->Liga))
           Q = Q->Liga;
        if (Q->Info == Ref)
            Resp= Q;
   }
   return Resp;
}
```

La eficiencia de este método se determina por el nivel de llenado de la estructura. Cuanto más llena esté más comparaciones adicionales requerirá para llegar al dato buscado. El primer intento siempre es directo, es decir se compara el dato con el que está en la dirección que le corresponde. Sin embargo, si el dato no está en esa posición entonces se debe proceder según el algoritmo usado para resolver colisiones. Por lo tanto, el comportamiento final de este método depende, además del nivel de llenado, del manejo de colisiones que se implemente.

El programa 10.5 presenta una aplicación de este algoritmo para ilustrar su uso. Los datos con los que se trabaja son objetos de la clase `Producto` y a los mismos se les asigna una dirección de acuerdo a su clave. Es decir, la función Hash se aplica sólo a su clave, por lo que en la clase `Producto` se sobrecarga el operador del residuo (%). En el programa se capturan algunos productos y se almacenan en una tabla Hash con encadenamiento. Posteriormente, se realizan algunas búsquedas sobre la tabla Hash, y finalmente se imprime un reporte de todos los productos almacenados en el arreglo. Para dar mayor claridad al ejemplo, se incluye en el programa 10.4 la clase `Producto` con todos los métodos requeridos para esta aplicación. Las clases `Lista` y `Hash` son las que se presentaron anteriormente y están en las bibliotecas `Lista.h` y `Hash.h` respectivamente.

Programa 10.4

```
/* Definición de la clase Producto. Se incluyen algunos atributos y los
➥métodos requeridos para esta aplicación, destacando la sobrecarga de
➥algunos operadores para que el algoritmo de Hash con encadenamiento
➥pueda usarse. */
class Producto
{
   private:
       int Clave;
       char NomProd[64];
       double Precio;
   public:
       Producto();
       Producto(int, char[], double);
       int operator == (Producto);
       int operator != (Producto);
       int operator % (int);
       friend   istream &operator>>(istream &, Producto &);
       friend   ostream &operator<<(ostream &, Producto &);
};
```

```
/* Constructor por omisión. */
Producto::Producto()
{}

/* Constructor con parámetros. */
Producto::Producto(int Cla, char NomP[], double Pre)
{
   Clave= Cla;
   strcpy(NomProd, NomP);
   Precio= Pre;
}

/* Sobrecarga del operador == para comparar dos objetos tipo Producto.
➥Para que dos productos sean iguales sus claves deben ser iguales. */
int Producto::operator == (Producto Prod)
{
   int Resp=0;
   if (Clave == Prod.Clave)
       Resp= 1;
   return Resp;
}

/* Sobrecarga del operador != para comparar dos objetos tipo Producto.
Para que dos productos sean distintos sus claves deben ser diferentes.
*/
int Producto::operator != (Producto Prod)
{
   int Resp=0;
   if (Clave != Prod.Clave)
       Resp= 1;
   return Resp;
}

/* Sobrecarga del operador % para ser aplicado en la función Hash
➥módulo o residuo. Recibe como parámetro el tamaño de la tabla Hash, y
➥regresa como resultado el residuo entre la clave del producto y dicho
➥valor. */
int Producto::operator % (int num)
{
   return Clave % num;
}

/* Sobrecarga del operador >> para que un objeto de la clase Producto
pueda ser leído directamente. */
istream &operator >> (istream &Lee,  Producto &ObjProd)
{
   cout <<"\n\nIngrese clave del producto:  ";
   Lee>> ObjProd.Clave;
   cout <<"\n\nIngrese nombre del producto:  ";
```

```
   Lee>> ObjProd.NomProd;
   cout <<"\n\nIngrese precio:  ";
   Lee>> ObjProd.Precio;
   return  Lee;
}

/* Sobrecarga del operador << para que un objeto de la clase Producto
pueda ser escrito directamente. */
ostream &operator << (ostream &Escribe, Producto &ObjProd)
{
   Escribe<<"\n\nProducto\n";
   Escribe<<"\nClave:  "<<ObjProd.Clave;
   Escribe<<"\nNombre:  "<<ObjProd.NomProd;
   Escribe<<"\nPrecio:  "<<ObjProd.Precio<<"\n";
   return  Escribe;
}
```


El programa 10.5 presenta una aplicación del algoritmo por transformación de claves con encadenamiento. Se crea una tabla Hash de objetos tipo `Producto`, luego permite hacer búsquedas sobre la tabla y finaliza con un reporte de todos los elementos almacenados.

Programa 10.5

```
/* Aplicación. Se incluyen las bibliotecas con el código de las
➥plantillas de la clase Lista y Hash, mismos que corresponden a lo pre-
➥sentado anteriormente. La clase Producto se incluye en la biblioteca
➥Producto.h */

#include "Lista.h"
#include "Hash.h"
#include "Producto.h"

void  main()
{
   int Total, i, Clave;

   /* Se declara un objeto de la clase derivada Encadenamiento con el
   ➥tipo Producto como base para T. */
   Encadenamiento<Producto> HashEnc;
   Producto Prod;
   /* Se pide al usuario el total de productos que se almacenarán en la
   ➥tabla Hash. */
```

```
    do {
        cout<<"\nCuántos productos quiere insertar:  ";
        cin>>Total;
    } while (Total <= 0);

    /* Se lee cada uno de los productos y se inserta en la tabla Hash
    ➥usando el método Inserta() el cual invoca a un método de inserción
    ➥de la clase Lista. Es importante recordar que la tabla Hash es un
    ➥arreglo de listas. */
    for (i= 1; i <= Total; i++)
    {
        cout<<"\n\nIngresa producto a insertar\n ";
        cin>>Prod;
        HashEnc.Inserta(Prod);
    }

    /* Por medio de este ciclo se permite que el usuario verifique qué
    ➥productos fueron dados de alta en el arreglo. Se ingresa la clave de
    ➥un producto, con  la que se crea un objeto de ese tipo y se invoca
    ➥al método que busca un elemento en la tabla Hash. Se da un mensaje
    ➥de acuerdo al resultado obtenido. */
    cout<<"\n\nClave del producto que busca (para terminar 0): ";
    cin>>Clave;
    while (Clave != 0)
    {
        Producto Prod(Clave, "", 0);
        if (HashEnc.Busca(Prod))
            cout<<"\nEse producto ya fue registrado\n";
        else
            cout<<"\nEse producto NO está registrado ";
        cout<<"\n\nClave del producto que buscas (para terminar 0): ";
        cin>>Clave;
    }

    /* Se imprime la información de todos los productos guardados en la
    ➥tabla Hash. */
    cout<<"\n\n\nReporte de todos los productos almacenados\n ";
    HashEnc.Imprime();
}
```

10.2.4 Búsqueda secuencial en listas

Debido a la naturaleza de las listas simplemente ligadas, la búsqueda secuencial es el único tipo de búsqueda que se puede aplicar sobre esta estructura de datos. La variante que se puede introducir depende de si se sabe si los elementos de la

misma están o no ordenados. Por lo tanto, para las listas se definirá una clase abstracta: `Búsqueda` y dos clases derivadas: `SecuencialListasDesordenadas` y `SecuencialListasOrdenadas` para poder implementar las dos variantes de esta operación.

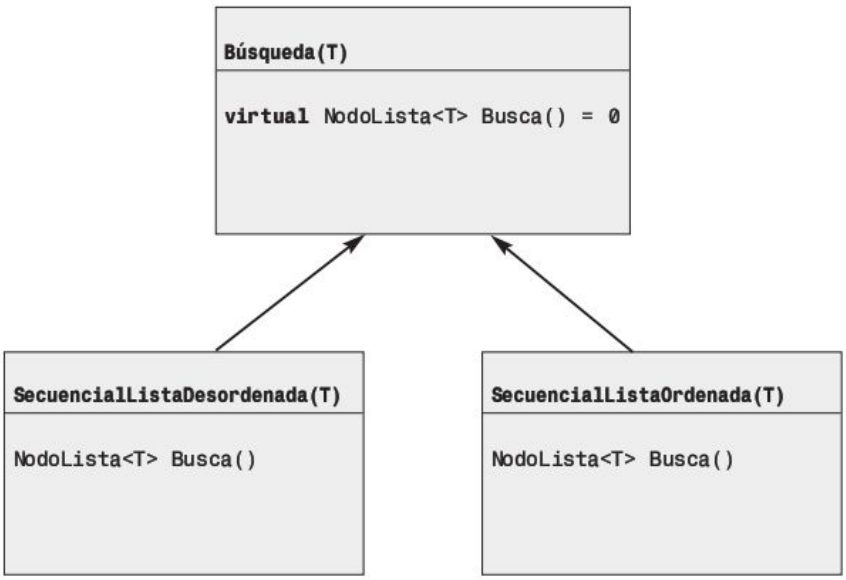


FIGURA 10.12 *Esquema de clases*

A continuación se presenta la codificación, usando el lenguaje ***C++***, de la plantilla de la clase abstracta. Observe que el método `Virtual Busca()` recibe como parámetro la lista donde se llevará a cabo la búsqueda y un elemento de tipo `T` que es el dato a buscar. Si la operación se definiera como un método de la clase lista, entonces sólo se recibiría como parámetro el dato a buscar.

```
/* Definición de la clase abstracta Busqueda. A partir de ella se derivan
➥otras dos clases para implementar los correspondientes algoritmos de
➥búsqueda secuencial para listas desordenadas y para listas ordenadas. */
template <class T>
class Busqueda
{
   public:
      virtual NodoLista<T> * Busca (Lista<T>, T) = 0;
};
```

La implementación de la clase derivada para realizar la búsqueda de un elemento en una lista simplemente ligada, cuyos elementos están **desordenados** queda:

```
/* Declaración de la clase SecuencialListaDesordenada, derivada de la
➥clase abstracta Busqueda. Se especifica el método Busca() de acuerdo al
➥algoritmo de búsqueda secuencial en una lista desordenada. */
template <class T>
class SecuencialListaDesordenada: public Busqueda<T>
{
   public:
      NodoLista<T> * Busca (Lista<T>, T);
};

/* Método que busca un elemento dado como referencia en una lista
➥desordenada. Regresa la dirección del nodo si lo encuentra o NULL en
➥caso contrario. Recibe como parámetro la lista en la cual se realizará
➥la búsqueda y el dato a buscar. */
template <class T>
NodoLista<T> * SecuencialListaDesordenada<T>::Busca(Lista<T> ListaDato,
➥T Valor)
{
   NodoLista<T> *Q, *Resp= NULL;

   /* Verifica si la lista tiene al menos un elemento. */
   if (ListaDato.RegresaPrimero())
   {
        Q = ListaDato.RegresaPrimero();
        while ((Q->RegresaInfo() != Valor)  && (Q->RegresaLiga()))
           Q = Q->RegresaLiga();
        if (Q->RegresaInfo() == Valor)
            Resp= Q;
   }
   return Resp;
}
```

Es importante señalar que si el dato a buscar fuera un objeto se debería sobrecargar el operador **!=** en la clase a la cual pertenece dicho objeto.

La implementación de la clase derivada para realizar la búsqueda de un elemento en una lista simplemente ligada, cuyos elementos están **ordenados** queda:

```
/* Declaración de la clase SecuencialListaOrdenada, derivada de la clase
➥abstracta Busqueda. Se especifica el método Busca() de acuerdo al
➥algoritmo de búsqueda secuencial en una lista ordenada de manera
creciente. */
template <class T>
class SecuencialListaOrdenada: public Busqueda<T>
{
   public:
      NodoLista<T> * Busca (Lista<T>, T);
};

/* Método que busca un elemento dado como referencia en una lista
➥ordenada de forma creciente. Regresa la dirección del nodo si lo
➥encuentra y NULL en caso contrario. */
template <class T>
NodoLista<T> * SecuencialListaOrdenada<T>::Busca(Lista<T> ListaDato, T
➥Valor)
{
   NodoLista<T> *Q, *Resp= NULL;

   /* Verifica que la lista tenga al menos un elemento. */
   if (ListaDato.RegresaPrimero())
   {
       Q = ListaDato.RegresaPrimero();
       while ((Q->RegresaInfo() < Valor) && (Q->RegresaLiga()))
          Q= Q->RegresaLiga();
       if (Q->RegresaInfo() == Valor)
           Resp= Q;
   }
   return Resp;
}
```

Es importante señalar que si el dato a buscar fuera un objeto se debería sobrecargar el operador < en la clase a la cual pertenece dicho objeto.

Con respecto a la eficiencia de esta operación es igual a la presentada para los arreglos. Dada la característica de la búsqueda, es independiente a si se implementa con arreglos o con listas. En consecuencia, aplican las fórmulas dadas en 10.1.

El programa 10.6 muestra una aplicación de la búsqueda secuencial en listas. Se crea una lista con los datos de las personas que asistieron a un congreso. Luego se permite consultar (realizando búsquedas) la lista para verificar si una determinada persona asistió o no al evento. Como no se sabe si la lista está ordenada, se

utiliza la búsqueda secuencial en listas desordenadas. Finalmente se imprimen los datos de todos los asistentes.

Programa 10.6

```
/* Se incluyen las bibliotecas en las que se guardaron las
➥plantillas de la clase Lista y de la clase Busqueda y su derivada
➥SecuencialListaDesordenada y además la biblioteca en la que está
➥la clase Persona. */
#include "Lista.h"
#include "Busqueda.h"
#include "Persona.h"

/* Función principal. Se crea la lista de personas y luego, por medio de
➥la clase ya definida, se buscan a personas de acuerdo a su nombre que
➥es un dato proporcionado por el usuario. */
void  main()
{
   /* Se crean los objetos de la clase SecuencialListaDesordenada y de
   ➥la clase Lista, usando la clase Persona como tipo base. Además se
   ➥declaran algunas otras variables de trabajo. */
   Lista<Persona> LisDesord, LisOrd;
   SecuencialListaDesordenada<Persona> Buscador;
   NodoLista<Persona> *Apunt;
   char NomPers[64];

   /* Se capturan los datos de los asistentes al congreso sin ningún
   ➥orden entre los mismos. */
   cout<<"\n\nIngrese la lista de asistentes al congreso.\n\n";
   LisDesord.CreaFinal();

   /* Se realiza la búsqueda secuencial de personas, por su nombre, en
   ➥la lista previamente creada. Si dicha persona asistió al congreso,
   ➥entonces se imprimen todos sus datos. En caso contrario sólo se
   ➥indica que no participó en el evento. */
   cout<<"\n\nIngrese nombre de la persona que desea verificar si
         ➥asistió al congreso.n";
   cout<<"Para terminar capture una X\n\n";
   cin>>NomPers;
   while (strcmp(NomPers, "X") != 0)
   {
      Persona Asistente(0, NomPers, "");
      Apunt = Buscador.Busca(LisDesord, Asistente);
      if ( Apunt != NULL)
           cout<<"\n\nEsa persona asistió al congreso y sus datos
                 ➥son:\n" <<Apunt->RegresaInfo()<<"\n";
```

```
        else
        cout<<"\n\nEsa persona NO asistió al congreso\n\n";
        cout<<"\n\nIngrese el nombre de la persona que desea verificar si
        ➥asistió al congreso\n\n";
        cout<<"Para terminar capture una X\n\n";
        cin>>NomPers;
        }

        /* Imprime todos los datos de los asistentes al congreso. */
        LisDesord.Imprime(LisDesord.RegresaPrimero());
}
```


La búsqueda secuencial en listas puede tratarse como una operación de la clase `Lista` en vez de considerarse como una clase. A continuación se presenta parte de esta clase en la cual se incorporaron los métodos de búsqueda ya estudiados, con la variante de que fueron escritos de manera recursiva.

```
// Clase Lista dependiente de la clase NodoLista.
template <class T>
class Lista;

/* Definición de la clase NodoLista. */
template <class T>
class NodoLista
{
   private:
       NodoLista<T> *Liga;
       T Info;
   public:
       NodoLista();
       friend  class Lista<T>;
       // Otros métodos estudiados en el capítulo 6, dedicado a las
       ➥listas ligadas.
};

/* Declaración del método constructor por omisión. Inicializa con el
➥valor NULL al puntero al siguiente nodo. */
template <class T>
NodoLista<T>::NodoLista()
{
   Liga = NULL;
}
```

```
/* Definición de la clase Lista. Se incluyen sólo los métodos de búsqueda,
➥tema de estudio de este capítulo. */
template <class T>
class Lista
{
   private:
       NodoLista<T> *Primero;
   public:
       Lista ();
       NodoLista<T> *BuscaSecuencialOrd (NodoLista<T> *, T);
       NodoLista<T> *BuscaSecuencialDesord (NodoLista<T> *, T);
       // Otros métodos estudiados en el capítulo 6, dedicado a las
       ➥listas ligadas.
};

/* Declaración del método constructor. Inicializa el puntero al primer
➥nodo de la lista con el valor NULL: indica lista vacía. */
template <class T>
Lista<T>::Lista()
{
   Primero = NULL;
}

/* Método de la clase Lista que busca un elemento en una lista ordenada
➥de manera creciente. Regresa la dirección del nodo si lo encuentra o
➥NULL en caso contrario. */
template <class T>
NodoLista<T> * Lista<T>::BuscaSecuencialOrd(NodoLista<T> * Ap, T Dato)
{
   if (Ap)
       if (Ap->Info < Dato)
            return BuscaSecuencialOrd(Ap->Liga, Dato);
       else
           if (Ap->Info == Dato)
                return Ap;
           else
                return NULL;
   else
       return NULL;
}

/* Método de la clase Lista que busca un elemento en una lista
➥desordenada. Regresa la dirección del nodo si lo encuentra o NULL
➥en caso contrario. */
template <class T>
NodoLista<T> * Lista<T>::BuscaSecuencialDesord (NodoLista<T>
                                                ➥* Ap, T Dato)
```

```
{
   if (Ap)
       if (Ap->Info != Dato)
            return BuscaSecuencialDesord(Ap->Liga, Dato);
       else
           return Ap;
   else
       return NULL;
}
```


10.2.5 Búsqueda en árboles

La operación de búsqueda en árboles depende de la estructura de los mismos. Las más usadas fueron presentadas en el capítulo 7, dedicado al estudio de estas estructuras de datos.

10.2.6 Búsqueda en gráficas

La operación de búsqueda en gráficas depende de la estructura de las mismas. Las más usadas fueron presentadas en el capítulo 8. Además, en el caso de estas estructuras, la búsqueda también queda determinada por la información que se pretende extraer de las mismas.

10.3 Búsqueda externa

La **búsqueda externa** es aquella que se realiza sobre un archivo previamente creado y guardado en algún dispositivo de almacenamiento secundario. La manera en la que se podrá tener acceso a los datos del archivo depende de la forma en que el archivo fue creado. En esta sección se estudiará la búsqueda secuencial y binaria en archivos de objetos.

La implementación de cualquiera de los tipos de búsqueda en archivo está estrechamente determinada por el lenguaje de programación utilizado.

10.3.1 Búsqueda externa secuencial

La **búsqueda secuencial** en archivos de objetos consiste en leer del archivo un objeto (el primero) y compararlo con el dato buscado. Si son iguales termina la búsqueda con éxito. En caso contrario se debe leer el siguiente elemento del archivo y se compara nuevamente con el dato que interesa. El proceso se repite hasta encontrar la información buscada o hasta llegar al final del archivo, caso en el que la búsqueda termina con fracaso.

Si la información del archivo estuviera ordenada, entonces se puede modificar la condición durante la cual se realiza la lectura y comparación de objetos para ganar eficiencia. Si estuviesen ordenados crecientemente, se busca mientras no sea el fin de archivo y mientas el dato leído sea menor que el elemento buscado. En cambio si el orden fuera decreciente, se busca mientras no sea el fin de archivo y mientas el dato leído sea mayor que el elemento buscado.

Como se puede apreciar, el algoritmo (tanto para archivos desordenados como para ordenados) es el mismo que el aplicado a estructuras de almacenamiento interno. La diferencia está en la implementación debido a las características propias de los archivos.

El programa 10.7 presenta la clase `Vehiculo` que será usada en la implementación de estos algoritmos.

Programa 10.7

```
/* Definición de la clase Vehiculo. Se incluyen sólo algunos atributos y
➥métodos, mismos que se utilizarán para ilustrar la búsqueda secuencial
➥en archivos de objetos. */
class Vehiculo
{
private:
   char Placa[8], NumMotor[16], Color[8];
   int Cilindros;
public:
   Vehiculo();
   Vehiculo(char *, char *, char *, int);
   char * RegresaPlaca();
   friend  istream &operator >> (istream &, Vehiculo &);
   friend  ostream &operator <<  (ostream &, Vehiculo &);
};
/* Constructor por omisión. */
Vehiculo::Vehiculo()
{}
```

```
/* Constructor con parámetros. */
Vehiculo::Vehiculo(char Pla[], char NumM[], char Col[], int Cil)
{
   strcpy(Placa, Pla);
   strcpy(NumMotor, NumM);
   strcpy(Color, Col);
   Cilindros= Cil;
}

/* Método que permite a usuarios externos a la clase conocer el atributo
➥Placa. */
char * Vehiculo::RegresaPlaca()
{
   return Placa;
}

/* Sobrecarga del operador >> para que un objeto tipo Vehiculo pueda
➥ser leído directamente. */
istream &operator >> (istream &Lee,  Vehiculo &ObjV)
{
   cout <<"\n\nIngrese placa del vehículo: ";
   Lee>> ObjV.Placa;
   cout <<"\n\nIngrese número de motor: ";
   Lee>> ObjV.NumMotor;
   cout <<"\n\nIngrese color: ";
   Lee>> ObjV.Color;
   cout <<"\n\nIngrese total de cilindros: ";
   Lee>> ObjV.Cilindros;
   return  Lee;
}

 /* Sobrecarga del operador << para que un objeto tipo Vehiculo pueda
➥ser impreso directamente. */
ostream &operator << (ostream &Escribe, Vehiculo &ObjV)
{
   Escribe<<"\n\nDatos del vehículo\n";
   Escribe<<"\nPlacas:  "<<ObjV.Placa;
   Escribe<<"\nNúmero de motor:  "<<ObjV.NumMotor;
   Escribe<<"\nColor:  "<<ObjV.Color;
   Escribe<<"\nTotal de cilindros:  "<<ObjV.Cilindros;
   return  Escribe;
}
```

El programa 10.8 presenta una aplicación de la búsqueda secuencial en un archivo cuya información está desordenada. Se crea un archivo con objetos de la clase `Vehiculo` y posteriormente se realiza la búsqueda de algún vehículo por medio del número de placas. Si lo encuentra despliega toda la información del mismo, y en caso contrario sólo indica que no se encontró.

Programa 10.8

```
/* Se incluye la biblioteca donde se guardó la clase Vehiculo. */
#include "Vehiculo.h"

/* Función auxiliar para crear un archivo de objetos tipo Vehiculo. */
void CreaArch()
{
   char NomArch[64];
   int Total, Indice;
   Vehiculo Auto;

   /* Se declara un objeto de la clase fstream, provista por C++ para el
   ➥manejo de archivos. */
   fstream Arch;

   cout<<"\n\nNombre del archivo que quiere crear: ";
   cin>>NomArch;

   /* Se crea un archivo para escritura. */
   Arch.open(NomArch, ios::out);

   cout<<"\n\nTotal de vehículos a registrar: ";
   cin>>Total;

   /* Se leen y se almacenan en el archivo objetos de la clase Vehiculo. */
   for (Indice= 1; Indice <= Total; Indice++)
   {
       cin>>Auto;
       Arch.write((char *) &Auto, sizeof(Auto));
   }
   Arch.close();
}

/* Función principal. Permite crear un nuevo archivo o usar uno ya
➥existente. Sobre el archivo elegido realiza búsqueda secuencial para
➥encontrar un objeto tipo Vehiculo. Se asume que la información está
➥desordenada. */
void main ()
{
   char Nom[64], Placa[64], Resp;
   fstream Arch;
   Vehiculo Auto;

   cout<<"\nQuieres crear un nuevo archivo s/n: ";
   cin>>Resp;
   if (Resp == 's')
        CreaArch();
```

```
    cout<<"\n\nIngresa el nombre del archivo que quieres consultar: ";
    cin>>Nom;

    /* Se abre el archivo para lectura. */
    Arch.open(Nom, ios::in);

    cout<<"\n\nIngrese el número de placas del vehículo: ";
    cin>>Placa;

    /* Se lee un objeto del archivo mientras no se llegue al fin del
    ➥mismo y mientras no se encuentre el elemento buscado. */
    Arch.read((char *) &Auto, sizeof(Auto));
    while (!Arch.eof() && (strcmp(Auto.RegresaPplaca(), Placa) != 0))
       Arch.read((char *) &Auto, sizeof(Auto));

    /* Se verifica si se encontró el auto en el archivo. */
    if (strcmp(Auto.RegresaPlaca(), Placa) == 0)
         cout<<"\n"<<Auto;
    else
         cout<<"\n\nEse auto no está registrado\n\n";

    Arch.close();
}
```


Este algoritmo se puede adaptar para búsqueda secuencial en archivos ordenados. Sólo se requiere cambiar la segunda condición del ciclo, quedando

(**strcmp**(Auto.RegresaPlaca(), Placa) < 0) para arreglos ordenados crecientemente, o

(**strcmp**(Auto.RegresaPlaca(), Placa) > 0) para arreglos ordenados decrecientemente.

10.3.2 Búsqueda externa binaria

La **búsqueda binaria** se aplica sólo a archivos de objetos que están ordenados. El espacio de búsqueda (que es todo el archivo) se divide a la mitad, luego se lee el dato que ocupa esa posición y se compara con el elemento que interesa. Si son iguales, la búsqueda termina con éxito. En caso contrario, se evalúa si el dato leído es menor o mayor que el dato buscado. En el primer caso se redefine el espacio de búsqueda limitándolo desde la posición central más uno hasta el final del archivo. En el segundo, desde la posición inicial hasta la central menos uno. El proceso se

repite hasta encontrar el dato buscado o hasta que el extremo izquierdo del espacio quede mayor al extremo derecho, lo cual implica terminar con fracaso.

Para poder aplicar este algoritmo a un archivo ya ordenado se requiere calcular la posición del último elemento almacenado en el archivo. Para ello se usan facilidades que ofrecen los lenguajes de programación, como se verá en el programa 10.9.

El algoritmo (en esencia) es el mismo que el analizado para estructuras de almacenamiento interno. La diferencia está en la implementación debido a las características propias de los archivos.

El programa 10.9 presenta la implementación de este algoritmo. Se vuelve a usar la clase `Vehiculo` definida en el programa 10.7. Si se requiere crear un archivo se puede usar la función del programa 10.8 creada para tal efecto.

Programa 10.9

```
/* Función principal. Utiliza búsqueda binaria para encontrar un
➥vehículo en un archivo previamente creado, cuya información está
➥ordenada por número de placas. */
void main ()
{
   char Nom[64], Placa[64];
   int Izq, Der, Cen;
   fstream Arch;
   Vehiculo Auto;

   cout<<"\n\nIngrese el nombre del archivo que quiere consultar: ";
   cin>>Nom;

   cout<<"\n\nIngrese el número de placas del vehículo: ";
   cin>>Placa;

   /* Se abre el archivo para lectura. */
   Arch.open(Nom, ios::in);

   /* Se posiciona al final del archivo. */
   Arch.seekg(0, ios::end);

   /* Se calcula el extremo derecho, el izquierdo y el central del
   ➥espacio de búsqueda. */
   Der = (int) Arch.tellg()/sizeof(Auto) -1;
   Izq= 0;
   Cen= (int) (Izq + Der) /2;
```

```
    /* Se posiciona el puntero del archivo en la posición central del
    ➥mismo y se lee el objeto que haya sido almacenado en esa posición. */
    Arch.seekg(Cen*sizeof(Auto), ios::beg);
    Arch.read((char *) &Auto, sizeof(Auto));

    /* Se busca mientras el extremo izquierdo sea menor o igual al
    ➥extremo derecho y mientras no se encuentre el elemento buscado. */
    while (Izq <= Der && strcmp(Auto.RegresaPlaca(), Placa) != 0)
    {
       if (strcmp(Auto.RegresaPlaca(), Placa) < 0)
            Izq= Cen +1;
       else
           Der = Cen -1;
       Cen= (int) (Izq + Der) /2;

       Arch.seekg(Cen*sizeof(Auto), ios::beg);
       Arch.read((char *) &Auto, sizeof(Auto));
    }

    /* Se comprueba si se encontró al vehículo buscado. */
    if (strcmp(Auto.RegresaPlaca(), Placa) == 0)
         cout<<"\n"<<Auto;
    else
         cout<<"\n\nEse auto no está registrado\n\n";

    Arch.close();
}
```

En el programa 10.9 se usaron métodos (***Seekg()*** y ***tellg()***) de la clase ***fstream*** para tener acceso a ciertas funcionalidades necesarias para realizar la búsqueda binaria. Con la ayuda de estos métodos se calculó el total de elementos almacenados en el arreglo y se pudo, en cada iteración, colocar el puntero del archivo en la posición central del mismo.

Ejercicios

1. Considere las fórmulas presentadas en Fórmulas 10.1, para búsqueda secuencial. Complete la siguiente tabla con el número mínimo, medio y máximo de comparaciones para distintos tamaños de arreglos.

Número de comparaciones			
Tam	Mínimo	Medio	Máximo
10			
100			
1000			
5000			
10000			

2. Considere las fórmulas presentadas en Fórmulas 10.2, para búsqueda binaria. Complete la siguiente tabla con el número mínimo, medio y máximo de comparaciones para distintos tamaños de arreglos.

Número de comparaciones			
Tam	Mínimo	Medio	Máximo
10			
100			
1000			
5000			
10000			

3. Se tiene un arreglo que almacena los datos de productos (puede usar la clase `Producto` del programa 10.4). Utilice la clase `SecuencialDesord` para declarar un objeto, el cual debe buscar un producto en el arreglo mencionado. El usuario proporcionará la clave del producto de interés.

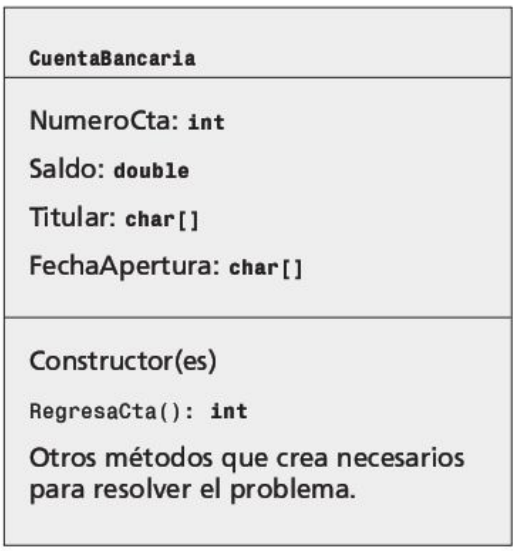
4. Escriba un programa en ***C++*** que realice las siguientes operaciones (modularice su solución):

   (a) Lea la información de varios productos (utilice la clase `Producto` ya mencionada) y almacénela en un arreglo.

   (b) Ordene el arreglo por medio de algunos de los algoritmos estudiados en el capítulo 9.

   (c) Usando la clase `Binaria` busque algún producto en el arreglo ya ordenado. Si lo encuentra debe imprimir toda la información del mismo, y en caso contrario debe indicar que ese producto no está registrado. El usuario proporcionará la clave del producto de interés.

5. Retome la clase `Arreglo` usada en el programa 10.1. Escriba un método, llamado `BusqBinaria(T)`, que implemente la búsqueda binaria como miembro de esa clase.
6. Modifique las clases correspondientes al método por transformación de claves, de tal manera que la función `Hash` se implemente como un método de la clase abstracta `Hash` y pueda ser usada por todas las clases derivadas.
7. Retome el problema anterior. Utilice alguna forma distinta para la función Hash (truncamiento, plegamiento o alguna diseñada por usted). Compare los resultados obtenidos para el mismo conjunto de datos. ¿Cuál distribuyó más uniformemente los elementos en la tabla Hash?
8. Modifique el método `Busca()` de la clase abstracta `Hash` y de sus clases derivadas para que su resultado sea un entero que indique la dirección donde encontró el elemento buscado o un negativo en caso de fracaso.
9. Se define la clase `CuentaBancaria` según se especifica más abajo. Utilice Hash con encadenamiento para almacenar en memoria principal un conjunto de objetos declarados a partir de dicha clase. Generalice la clase `Hash` vista en este capítulo y, si corresponde, su clase derivada, para que además de insertar y buscar, se pueda eliminar un elemento de la tabla Hash. Considere casos de error.



10. Escriba un programa en ***C++*** que realice las siguientes operaciones (organice su solución modularmente):

    a) Capture objetos de tipo `Persona` (puede usar la clase definida en el programa 10.1) y almacénelos en un archivo. Los datos son proporcionados sin orden.

    b) Busque, por su nombre, una persona previamente guardada en el archivo. Si la encuentra debe imprimir toda la información de dicha persona. En caso contrario sólo indicará que no está registrada. El usuario proporciona como entrada el nombre de la persona a buscar.

11. Escriba un programa en ***C++*** que realice las siguientes operaciones (organice su solución modularmente):

    a) Capture objetos de tipo `CuentaBancaria` (puede usar la clase definida en el problema 9) y almacénelos en un archivo.

    b) Utilice alguno de los algoritmos de ordenación vistos en el capítulo 9 para ordenar el archivo, de acuerdo al número que identifica a cada cuenta bancaria.

    c) Busque, por número de cuenta, una cuenta previamente guardada en el archivo. Si la encuentra debe imprimir toda la información de dicha cuenta. En caso contrario sólo indicará que no está registrada. El usuario proporciona como entrada el número de la cuenta a buscar. Para realizar la búsqueda utilice el algoritmo de búsqueda binaria, ya que como resultado del inciso b) el archivo debe estar ordenado según el atributo `NumeroCta`.

ÍNDICE

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

El objetivo de este libro es mostrar las principales estructuras de datos con base en el paradigma orientado a objetos. Se presenta cada una de las principales estructuras y la manera en la que se almacena y recupera la información. El texto explica la lógica requerida para llevar a cabo las operaciones más importantes y muestra la implementación de estos algoritmos. Para la programación de algoritmos y ejemplos se utiliza el lenguaje de programación **C++**, uno de los lenguajes orientados a objetos más conocidos y utilizados, tanto en el ámbito académico como a nivel profesional.

El libro está orientado a todos aquellos que:

- Quieran conocer y entender los principios de la programación orientada a objetos.
- Necesiten saber cómo funcionan las estructuras de datos.
- Deseen conocer y dominar la implementación de los principales algoritmos dedicados al manejo de las estructuras de datos.
- Les interese aprender a usar las estructuras de datos en la solución de problemas y en la implementación de estas soluciones.

En cada uno de los capítulos se explican los principales conceptos y se refuerzan con ejemplos que ayudan a su comprensión. Además, se incluyen programas, o las instrucciones requeridas, para mostrar la implementación de los algoritmos y las soluciones a los problemas de aplicación de las estructuras estudiadas.

Para mayor información sobre este tema visite:
www.pearsoneducacion.net/guardati

ISBN 970-26-0792-2
90000
9 789702 607922

PEARSON Educación ®

Visítenos en:
www.pearsoneducacion.net