CAPÍTULO

6 Estructuras de datos

Muchos algoritmos requieren una representación apropiada de los datos para lograr ser eficientes. Esta representación junto con las operaciones permitidas se llama *estructura de datos*. Típicamente todas las estructuras de datos permiten inserciones arbitrarias. Las estructuras de datos varían en cómo permiten el acceso a miembros del grupo. Algunas permiten tanto accesos como operaciones de borrado arbitrarios. Otras imponen restricciones, tales como permitir el acceso sólo al elemento más recientemente insertado, o al menos recientemente insertado del grupo.

Este capítulo discute siete de las más comunes estructuras de datos: pilas, colas, listas enlazadas, árboles, árboles binarios de búsqueda, tablas hash y colas de prioridad. El objetivo es definir cada estructura de datos, así como dar una idea intuitiva de la complejidad en tiempo de las operaciones de inserción, borrado y acceso. La presentación de una implementación eficiente se retrasa hasta las Partes IV y V.

En este capítulo veremos:

- Descripciones de las estructuras de datos más comunes, sus operaciones y sus tiempos de ejecución.
- Para cada estructura de datos, una interfaz Java que contenga el protocolo que debe ser implementado.
- Algunas aplicaciones de las estructuras de datos.

Estas estructuras de datos son utilizadas en los casos de estudio de la Parte III del libro y son implementadas en la Parte IV. El objetivo es mostrar que la especificación, que describe la funcionalidad, es independiente de la implementación. No necesitamos conocer *cómo* se implementa algo con tal de que sepamos que *está* implementado.

6.1 ¿Por qué necesitamos estructuras de datos?

Una *estructura de datos* es una representación de datos junto con las operaciones permitidas sobre dichos datos.

Las estructuras de datos nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de componentes. Como se muestra más tarde en el capítulo, las estructuras de datos descritas en esta sección (e implementadas más adelante) tienen usos repetidos. Una vez que la estructura de da-

tos ha sido implementada, puede ser utilizada una y otra vez en diversas aplicaciones. En este capítulo sólo se proporciona las correspondientes interfaces. Las consideraciones sobre implementaciones eficientes se retrasan hasta la Parte IV.

Las estructuras de datos permiten la reutilización de componentes.

Este enfoque —la separación de la interfaz y la implementación— es parte del paradigma de la programación orientada a objetos. El usuario de la estructura de datos no necesita ver la implementación, sólo las operaciones disponibles. Ésta es la parte de ocultamiento y encapsulación, de la programación orientada a objetos. Sin embargo, otra parte importante de la programación orientada a objetos es la *abstracción*. Debemos pensar cuidadosamente el diseño de las estructuras de datos porque debemos escribir programas que utilicen esas estructuras de datos sin tener en cuenta su implementación. Esto hace a la interfaz más limpia, más flexible (más reutilizable) y generalmente más fácil de implementar.

Todas las estructuras son fáciles de implementar si no nos preocupamos del rendimiento. Esto nos permite colocar componentes «baratas» dentro de nuestro programa en la fase de depuración del mismo. Los ejercicios al final de este capítulo le piden escribir implementaciones ineficientes, útiles para procesar pequeñas cantidades de datos. Más tarde, reemplazaremos las implementaciones de estructuras de datos «baratas» por implementaciones que tienen un mejor rendimiento en tiempo y/o espacio, y que son útiles para manejar grandes cantidades de datos. Ya que las interfaces están fijadas, estos cambios no requerirán virtualmente ningún cambio en los programas que utilizan las estructuras de datos.

Este capítulo describe las estructuras de datos utilizando interfaces. En las Partes IV y V implementamos cada interfaz, derivando entonces una nueva clase. Por ejemplo, las pilas se especifican con la interfaz `Pila`. Cuando la implementemos en el Capítulo 15, la clase resultante será denominada `PilaVec` (implementación basada en vectores). En todos los casos, la nueva clase implementará las especificaciones en la interfaz y podrá incluir alguna funcionalidad adicional.

Por ejemplo, la Figura 6.1 presenta una interfaz para la celda de memoria descrita en la Sección 4.6. La interfaz describe las funciones disponibles; la clase derivada concreta debe proporcionar su definición. La implementación se muestra en la Figura 6.2 y es idéntica a la de la Figura 4.16, excepto por lo que se refiere a la cláusula `implements`. La rutina `main` de la Figura 4.17 puede utilizarse sin cambios.

Es importante observar que las estructuras de datos implementadas aquí almacenan referencias a los elementos insertados y no realizan copias internas.

Para ahorrar espacio, las interfaces mostradas en este texto contienen comentarios concisos al comienzo de cada clase, en vez de los comentarios más extensos para Javadoc. El código en Internet tiene ambos tipos de comentarios. Es importante observar que las estructuras de datos implementadas aquí almacenan referencias a los elementos insertados y no realizan copias internas. Por tanto, es una buena

```
1  // Interfaz CeldaMem: simula una celda RAM genérica
2  //
3  // ******************OPERACIONES PÚBLICAS**********************
4  // Object lectura( )          --> Devuelve el valor almacenado
5  // void escritura( Object x ) --> Almacena x
6
7  public interface CeldaMem
8  {
9      Object leer( );
10     void escribir( Object x );
11 }
```

Figura 6.1 Interfaz para la clase de celdas abstractas de memoria.

```
1  // Clase CeldaMemoria
2  //
3  // ******************OPERACIONES PÚBLICAS**********************
4  //
5  // Object leer( )              --> Devuelve el valor almacenado
6  // void escribir( Object x )   --> Almacena x
7
8  public class CeldaMemoria implements CeldaMem
9  {
10         // Métodos públicos
11     public Object leer( ) { return valorAlmacenado; }
12     public void escribir( Object x ) { valorAlmacenado = x; }
13
14         // Representación interna privada del dato
15     private Object valorAlmacenado;
16 }
```

Figura 6.2 Implementación de la clase concreta de celdas de memoria.

práctica colocar objetos inmutables en estas estructuras de datos, de tal forma que un agente externo no pueda cambiar el estado de un objeto que se encuentra dentro de una estructura de datos contenedora.

6.2 Las pilas

Una *pila* es una estructura de datos en la cual el acceso está limitado al elemento más recientemente insertado. Se comporta de forma muy parecida a una pila de facturas, de platos o de periódicos. El último elemento añadido a la pila es colocado en la cima, donde es accedido muy fácilmente, mientras que los elementos que llevan más tiempo son más difíciles de acceder. Por tanto, una pila es apropiada si sólo queremos acceder al elemento en la cima; el resto de elementos son inaccesibles (en principio).

Una *pila* limita el acceso al elemento insertado más recientemente.

En una pila, las tres operaciones naturales de `insertar`, `eliminar` y `buscar`, se renombran por `apilar`, `desapilar` y `cima`. Estas operaciones básicas se ilustran en la Figura 6.3. En la Figura 6.4 se muestra una interfaz Java para una pila abstracta. Se incluye un método `cimaYDesapilar`, que combina estas dos operaciones. En este ejemplo no aparece ninguna característica nueva de Java. La Figura 6.5 muestra cómo se utiliza una clase `Pila` y proporciona la salida correspondiente. Observe que una pila puede ser utilizada para dar la vuelta a una lista de elementos.

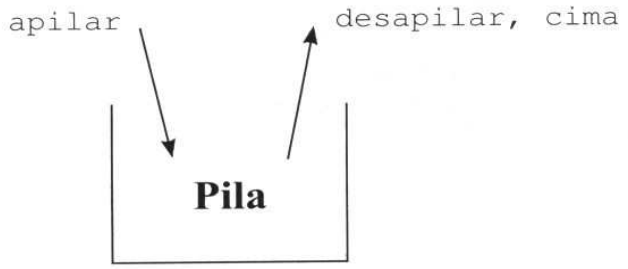


Figura 6.3 Modelo de una pila: la entrada a la pila se realiza con `apilar`, el acceso con `cima` y el borrado con `desapilar`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz Pila
6  //
7  // ********************OPERACIONES PÚBLICAS**********************
8  // void apilar( x )            --> Inserta x
9  // void desapilar( )           --> Elimina el último elemento insertado
10 // Object cima( )              --> Devuelve el último elemento insertado
11 // Object cimaYDesapilar( )    --> Devuelve y elimina el último elemento
12 // boolean esVacia( )          --> Devuelve true si vacía; si no false
13 // void vaciar( )              --> Elimina todos los elementos
14 // ************************ERRORES*******************************
15 // cima, desapilar, o cimaYDesapilar sobre la pila vacía
16
17 public interface Pila
18 {
19     void    apilar( Object x );
20     void    desapilar( )         throws DesbordamientoInferior;
21     Object  cima( )              throws DesbordamientoInferior;
22     Object  cimaYDesapilar( )    throws DesbordamientoInferior;
23     boolean esVacia( );
24     void    vaciar( );
25 }
```

Figura 6.4 Interfaz para las pilas.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las pilas
5
6  public final class TestPila
7  {
8      public static void main( String [ ] args )
9      {
10         Pila p = new PilaVec( );
11
12         for( int i = 0; i < 5; i++ )
13             p.apilar( new Integer( i ) );
14
15         System.out.print( "Contenido:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + p.cimaYDesapilar( ) );
20         }
21         catch( DesbordamientoInferior e ) { }
22
23         System.out.println( );
24     }
25 }
```

Figura 6.5 Programa ejemplo con pilas: la salida es `Contenido: 4 3 2 1 0`.

Cada operación de las pilas debería tardar una cantidad constante de tiempo en ejecutarse, independientemente del número de elementos apilados. De forma análoga, encontrar el periódico de hoy en una pila de periódicos es rápido, independientemente de lo alta que sea la pila. Sin embargo, el acceso arbitrario a una pila no se soporta de forma eficiente, por lo que no se cataloga como opción.

Las operaciones sobre las pilas deberían tardar una cantidad constante de tiempo en ejecutarse.

Lo que hace que las pilas sean útiles es que hay muchas aplicaciones en las que sólo es necesario acceder al elemento más recientemente insertado. Como ilustración, en la próxima sección se describe una utilización importante de las pilas en el diseño de compiladores.

6.2.1 Las pilas y los lenguajes de programación

Los compiladores comprueban los programas buscando errores sintácticos. A menudo, sin embargo, la falta de un símbolo (como por ejemplo la falta de un finalizador de comentario `*/` o `}`) causa que el compilador desparrame cientos de líneas de mensajes de error sin identificar el error real.

Una herramienta útil en esta situación es un programa que comprueba si todo está equilibrado: es decir, si para todo `{` existe un `}` correspondiente, todo `[` tiene un `]`, etc. La secuencia `[()]` es correcta, pero `[(])` no lo es, por lo que contar simplemente el número de apariciones de cada símbolo no es suficiente. (Suponga por ahora que estamos procesando una secuencia de tokens y no se preocupe de problemas tales como que la constante carácter `'{'` no necesita emparejarse con `'}'`.)

Una pila es útil para comprobar si hay símbolos desequilibrados, porque sabemos que cuando se encuentra un símbolo de terminación como `)`, debe emparejarse con el símbolo no cerrado `(` más recientemente visto.

Se puede utilizar una pila para comprobar si hay símbolos desequilibrados.

Por tanto, colocando los símbolos de apertura en una pila, podemos fácilmente comprobar si un símbolo de cierre tiene sentido.

En concreto, tenemos el siguiente algoritmo:

1. Construir una pila vacía.
2. Leer símbolos hasta el final del fichero.
   a) Si el símbolo es un símbolo de apertura, apilarlo en la pila.
   b) Si es un símbolo de terminación y la pila está vacía, entonces producir un mensaje de error.
   c) En otro caso, desapilar de la pila. Si el símbolo desapilado no es el símbolo de apertura correspondiente, producir un mensaje de error.
3. Al finalizar el fichero, si la pila no está vacía, mostrar un error.

En la Sección 11.1 desarrollaremos este algoritmo para que trabaje sobre (casi) todos los programas Java. Detallaremos la devolución de errores así como el procesamiento de comentarios, cadenas, constantes de tipo carácter y secuencias de escape.

El algoritmo para comprobar si los símbolos están equilibrados sugiere una forma de implementar las llamadas a procedimientos. El problema es que cuando se hace una llamada a un nuevo método, todas las variables locales al método que realiza la llamada deben ser salvadas por el sistema; de otra forma, el nuevo método podría sobrescribir estas variables. Es más, la posición actual dentro de la rutina

Las pilas son utilizadas en la mayoría de los lenguajes para implementar las llamadas a métodos.

llamante debe ser guardada de manera que el nuevo método sepa dónde volver cuando termine. Este problema es parecido al de los símbolos equilibrados porque una llamada a un método y el regreso de un método son esencialmente iguales que un paréntesis abierto y un paréntesis cerrado, así que se deberían poderse aplicar las mismas ideas. Efectivamente, esto es así. Como se discute en la Sección 7.3, la pila se utiliza en muchos lenguajes para implementar las llamadas a métodos.

El algoritmo *de evaluación con precedencia entre operadores* utiliza una pila para evaluar expresiones.

Una aplicación final importante de las pilas es la evaluación de expresiones en lenguajes de programación. En la expresión `1 + 2 * 3`, en el momento en que nos encontramos el `*`, el operador `+` y los operandos `1` y `2`, ya han sido leídos. ¿Debe operar `*` sobre `2` o sobre `1 + 2`? Las reglas de precedencia nos dicen que `*` opera sobre `2`, que es el operando más recientemente visto. Después de leer el `3`, podemos evaluar `2 * 3`, obteniendo `6`, para después aplicar el operador `+`. Esto sugiere que los operandos deberían ser guardados en una pila (ya que el `+` es retenido hasta que el operador `*` de mayor precedencia se evalúa). Un algoritmo que utiliza esta estrategia es el algoritmo de *evaluación de expresiones con precedencia entre operadores*. Su funcionamiento se describe en la Sección 11.2.

6.3 Las colas

Las *colas* restringen el acceso al elemento insertado menos recientemente.

Otra estructura de datos simple son las *colas*. En muchos casos, es importante poder encontrar y/o eliminar el elemento más recientemente insertado. Pero en otras ocasiones, no sólo no es importante, sino que es la forma errónea de hacerlo. Por ejemplo, en un sistema multiprocesador, cuando los trabajos se mandan a una impresora, esperamos que el trabajo menos reciente o más viejo sea impreso antes. Esto no es solamente justo, sino que es necesario para garantizar que el primer trabajo no espera para siempre. En consecuencia, uno espera encontrar colas de impresión en todos los grandes sistemas.

Las operaciones básicas permitidas por las colas son:

- `insertar` — inserción al final de la línea,
- `quitarPrimero` — eliminación del elemento al frente de la línea, y
- `primero` — acceso al elemento en el frente de la línea.

La Figura 6.6 ilustra estas operaciones sobre colas. Históricamente, `quitarPrimero` y `primero` han sido combinadas en una sola operación. Seguimos aquí esta idea, de modo que `quitarPrimero` devuelve el primer elemento además de eliminarlo de la cola, si bien mantenemos también la operación de acceso `primero`.

Las operaciones de las colas consumen un tiempo constante por petición.

La Figura 6.7 ilustra la interfaz de las colas, mientras que la Figura 6.8 muestra cómo se usan las colas, produciendo un ejemplo de salida. Ya que las operaciones de las colas están restringidas de forma similar a las de las pilas, es de esperar

Figura 6.6 Modelo de cola: la entrada se realiza con `insertar`, la salida con `primero` y la eliminación con `quitarPrimero`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz Cola
6  //
7  // ******************OPERACIONES PÚBLICAS*************************
8  // void insertar( x )       --> Inserta x
9  // Object primero( )        --> Devuelve el primer elemento
10 // Object quitarPrimero( ) --> Devuelve y elimina el primer elemento
11 // boolean esVacia( )       --> Devuelve true si vacía; si no false
12 // void vaciar( )           --> Elimina todos los elementos
13 // ***************************ERRORES*******************************
14 // primero y quitarPrimero sobre una cola vacía
15
16 public interface Cola
17 {
18     void    insertar( Object X );
19     Object  primero( ) throws DesbordamientoInferior;
20     Object  quitarPrimero( ) throws DesbordamientoInferior;
21     boolean esVacia( );
22     void    vaciar( );
23 }
```

Figura 6.7 Interfaz de las colas.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las colas
5
6  public final class TestCola
7  {
8      public static void main( String [ ] args )
9      {
10         Cola c = new ColaVec( );
11
12         for( int i = 0; i < 5; i++ )
13             c.insertar( new Integer( i ) );
14
15         System.out.print( "Contenido:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + c.quitarPrimero( ) );
20         }
21         catch( DesbordamientoInferior e ) { }
22
23         System.out.println( );
24     }
25 }
```

Figura 6.8 Programa ejemplo con colas: la salida es `Contenido: 0 1 2 3 4`.

que también tarden un tiempo constante en ejecutarse. Así es en efecto. Todas las operaciones básicas sobre las colas tardan un tiempo $O(1)$. En los casos de estudio veremos varias aplicaciones de las colas.

6.4 Listas enlazadas

Las *listas enlazadas* se usan para evitar movimientos de grandes cantidades de datos. Almacenan elementos con el coste añadido de una referencia adicional por elemento.

En una *lista enlazada*, los elementos se almacenan de forma no contigua, en vez de en un vector de posiciones de memoria consecutivas. Para conseguir esto, cada elemento se almacena en un *nodo* que contiene el objeto y una referencia al siguiente nodo en la lista, como se muestra en la Figura 6.9. En este marco, se mantienen referencias al primer y último nodo de la lista. Para ser más precisos, un nodo típico es el siguiente:

```
class NodoLista
{
    Object dato; // Algún elemento
    NodoLista siguiente;
}
```

En cualquier momento, podemos añadir un nuevo último elemento `x` haciendo lo siguiente:

```
ultimo.siguiente = new NodoLista( ); // odjunta un nuevo NodoLista
ultimo = ultimo.siguiente;           // Ajusta el último
ultimo.dato = x;                     // Coloca x en el nodo
ultimo.siguiente = null;             // Es el último; ajusta siguiente
```

Ahora, un elemento cualquiera no puede encontrarse con un único acceso; para ello debemos ir recorriendo la lista. Esto es similar a la diferencia entre acceder a un elemento en un disco compacto (un acceso) o en una cinta (secuencial). Aunque esto puede hacer pensar que las listas son menos atractivas que los vectores, tienen sus ventajas. Primero, una inserción en medio de la lista no requiere mover todos los elementos que se encuentran después del punto de inserción. Los movimientos de datos son muy costosos en la práctica, y las listas enlazadas permiten la inserción con sólo una cantidad constante de instrucciones de asignación.

Merece la pena hacer notar que si permitimos acceso sólo al `primero`, entonces tenemos una pila, y si permitimos inserciones sólo por el `ultimo` y accesos sólo por el `primero`, tenemos una cola.

Sin embargo, en estas situaciones necesitamos habitualmente operaciones más generales, tales como encontrar o eliminar cualquier elemento en la lista, guiados por su nombre. También necesitamos ser capaces de insertar un elemento en cualquier posición. Esto es mucho más que lo que permite una pila o una cola. La Figura 6.10 ilustra estas operaciones de las listas enlazadas.

Para acceder a los elementos en la lista, necesitamos una referencia al correspondiente nodo. Sin embargo, conceder este acceso viola claramente el principio de ocultamiento de la información. Necesitamos asegurar que cualquier acceso a

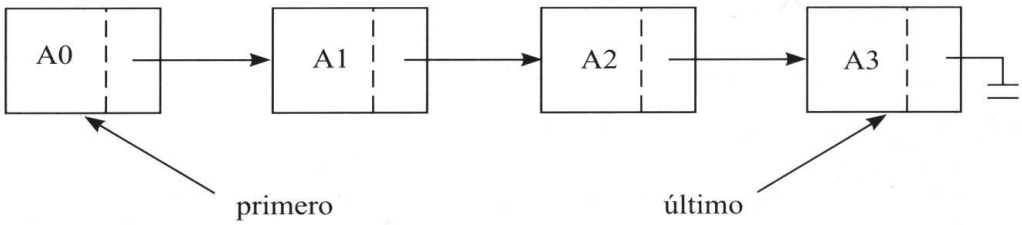


Figura 6.9 Una lista enlazada simple.

Figura 6.10 Modelo de lista enlazada: las entradas son arbitrarias, cualquier elemento puede salir y se soporta la iteración, aunque esta estructura de datos no es eficiente en tiempo.

la lista a través de una referencia es seguro. Para conseguir esto, definimos la lista en dos partes: una clase para las listas y una clase iteradora. La Figura 6.11 muestra la interfaz básica para las listas enlazadas, facilitando métodos que describen solamente el estado de la lista.

El acceso a la lista se realiza a través de una clase iteradora. La clase lista tiene operaciones que reflejan el estado de la lista. El resto de operaciones están en la clase iteradora.

La Figura 6.12 define una clase iteradora utilizada en todos los accesos a la lista. Para ver cómo se utiliza, veamos el código estándar para imprimir todos los elementos de la lista. Si la lista estuviera almacenada en un vector contiguo, un código típico sería el siguiente:

```
// Recorre el vector v, mostrando cada elemento
for( int indice = 0; indice < tamanyo; indice++ )
    System.out.println( v[ indice ] );
```

En Java básico, el código para iterar a través de toda la lista enlazada es

```
// Recorre la lista unaLista, mostrando cada elemento
for( NodoLista p = unaLista.primero; p != null; p = p.siguiente )
    System.out.println( p.dato );
```

```
 1  package EstructurasDatos;
 2
 3  // Interfaz Lista
 4  //
 5  // Acceso mediante la clase ListaIter
 6  //
 7  // ********************OPERACIONES PÚBLICAS*******************
 8  // boolean esVacia( )  --> Devuelve true si vacía; si no, false
 9  // void vaciar( )      --> Elimina todos los elementos
10  // ************************ERRORES*****************************
11  // Ningún error especial
12
13  public interface Lista
14  {
15      boolean esVacia( );
16      void vaciar( );
17  }
```

Figura 6.11 Interfaz para la lista abstracta.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz ListaIter; mantiene la posición actual
6  //
7  // ********************OPERACIONES PÚBLICAS************************
8  // void insertar( x )    --> Inserta x después de la posición actual
9  // void eliminar( x )    --> Elimina x
10 // boolean buscar( x )   --> Actualiza la posición actual accediendo a x
11 // void cero( )          --> Coloca actual antes que el primer elemento
12 // void primero( )       --> Coloca actual en el primer elemento
13 // void avanzar( )       --> Avanza
14 // boolean estaDentro( )--> true, si está en una posición valida
15 // Object recuperar      --> Devuelve el elemento actual
16 // ************************ERRORES*******************************
17 // Excepciones lanzadas en acceso, inserción o eliminación ilegal.
18
19 public interface ListaIter
20 {
21     void    insertar( Object x )  throws ElementoNoEncontrado;
22     boolean buscar( Object x );
23     void    eliminar ( Object x ) throws ElementoNoEncontrado;
24     boolean estaDentro( );
25     Object  recuperar( );
26     void    cero( );
27     void    primero( );
28     void    avanzar( );
29 }
```

Figura 6.12 Interfaz para el iterador de lista abstracto.

El mecanismo de iteración que Java usaría es similar al siguiente (ya que `ListaIter` es una interfaz, la `ListaIter` que sigue a `new` debería ser reemplazada por alguna clase que implementara a la interfaz `ListaIter`)

```
// Recorre laLista, utilizando la abstracción y el iterador
ListaIter iter = new ListaIter( laLista );
for( iter.primero( ); iter.estaDentro( ); iter.avanzar( ) )
    System.out.println( iter.recuperar( ) );
```

La inicialización anterior al bucle `for` se consigue llamando al constructor de `ListaIter`. El test utiliza el método `estaDentro` definido en la clase `ListaIter`. El método `avanzar` avanza al siguiente nodo de la lista enlazada. Podemos acceder al elemento actual llamando al método `recuperar` definido en `ListaIter`. El principio general es que garantizamos la seguridad, ya que todos los accesos a la lista se hacen a través de la clase `ListaIter`. Por último, indicaremos que podríamos tener varios iteradores recorriendo, simultánea pero independientemente, una única lista.

Para realizar esta tarea, la clase `ListaIter` debe mantener dos objetos internos. Primero, necesita una referencia al elemento «actual». Y en segundo lugar, necesita una referencia a la lista. Esta referencia es inicializada (por única vez) en el constructor. La Figura 6.13 muestra la interacción entre la lista y su iterador. Los detalles de la implementación de la lista enlazada se verán en el Capítulo 16. El uso de las técnicas con pilas y colas se verá en el Capítulo 15.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las listas
5
6  public final class TestLista
7  {
8      public static void main( String [ ] args )
9      {
10         Lista laLista   = new ListaEnlazada( );
11         ListaIter iter  = new ListaEnlazadaIter( laLista );
12
13             // Insertar repetidamente elementos por el principio
14         for( int i = 0; i < 5; i++ )
15         {
16             try
17               { iter.insertar( new Integer( i ) ); }
18             catch( ElementoNoEncontrado e ) { } // No puede ocurrir
19             iter.cero( );                  // Mueve iter al principio
20         }
21
22         System.out.print( "Contenido:" );
23         for( iter.primero( ); iter.estaDentro( ); iter.avanzar( ) )
24             System.out.print( " " + iter.recuperar( ) );
25         System.out.println( " fin" );
26     }
27 }
```

Figura 6.13 Ejemplo de uso de las listas: la salida es `Contenido: 4 3 2 1 0 fin.`

Aunque empezamos la discusión utilizando listas enlazadas, las interfaces de las Figuras 6.11 y 6.12 pueden utilizarse para cualquier lista, independientemente de la implementación subyacente. La interfaz no especifica que deban utilizarse listas enlazadas.

6.5 Árboles generales

Un *árbol* está formado por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos.

Los *árboles* son una estructura de datos utilizada muy a menudo, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos. En este texto sólo se consideran *árboles con raíz*. Un árbol con raíz tiene las siguientes características:

- Un nodo es distinguido como la raíz.
- Todo nodo c, excepto la raíz, está conectado por medio de una arista a un único nodo p. p es el padre de c, y c es uno de los hijos de p.
- Hay un único camino desde la raíz a cada nodo. El número de aristas que deben atravesarse es la longitud del camino.

Existen varias relaciones entre los nodos de un árbol.

La Figura 6.14 muestra un árbol. El nodo raíz es A. Los hijos de A son B, C, D y E. Ya que A es la raíz, no tiene padre. El resto de nodos tiene padre. Por ejemplo, el padre de B es A. A los nodos que no tienen hijos se les llama *hojas*. Las hojas de este árbol son los nodos C, F, G, H, I y K. La longitud del camino de A hasta K es tres (aristas). La longitud del camino desde A a A es cero aristas. Pro-

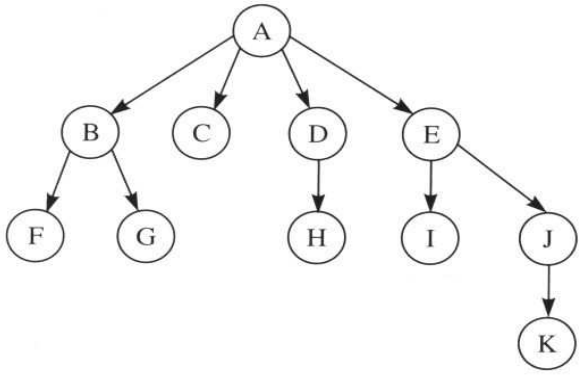


Figura 6.14 Un árbol.

piedades como ascendiente, descendiente o hermano, pueden también definirse con el significado usual. Así, los hermanos de *C* son los nodos *B*, *D* y *E*.

Los árboles se utilizan en sistemas de ficheros.

Los árboles son una estructura de datos fundamental en computación. Casi todos los sistemas operativos almacenan los ficheros en estructuras que son árboles o similares. Por ejemplo, bajo DOS, VMS o Unix, los directorios se almacenan en los nodos internos que no son hojas del árbol, mientras que el resto de ficheros se almacenan en las hojas. Atravesar una arista, es lo mismo que descender a un subdirectorio. Una arista especial que conecta de vuelta al padre (a saber, la entrada `..`) permite fácilmente la vuelta atrás en el árbol (aunque por crear ciclos, la entrada `..` hace que ésta estructura sea sólo *parecida* a un árbol, en vez de un árbol propiamente dicho).

Con esto en mente, podemos ver qué tipo de operaciones permiten los árboles. Queremos añadir, buscar y eliminar nodos del árbol. Como en el caso de las listas enlazadas, definiremos un iterador que nos permita acceder a los nodos, manteniendo la noción de nodo actual. En particular, debemos proporcionar métodos que nos permitan recorrer el árbol de manera ordenada. De esta forma, en la Figura 6.15 presentamos las interfaces para la clase de los árboles y su iterador, de forma análoga a como hicimos para las listas enlazadas. Las operaciones `irARaiz`, `primerHijo` y `siguienteHermano` son suficientes para permitirnos recorrer el árbol de diferentes maneras. En el Capítulo 17 describiremos una jerarquía de clases que permiten recorridos de árboles utilizando solamente los métodos `primerHijo`, `recuperar`, `esValido` y `siguienteHermano`. Esto incluye las clases `OrdenSim`, `PreOrden`, `PostOrden` y `PorNiveles`.

Los *árboles sintácticos de expresiones* se utilizan en el análisis sintáctico. En el árbol sintáctico de una expresión, el valor de un nodo es el resultado de aplicar el operador en el nodo, tomando como operandos los valores de los hijos. Un *árbol binario* tiene, a lo sumo, dos hijos por nodo.

Una segunda aplicación de los árboles corresponde a los denominados *árboles de expresiones*, que se ilustran en la Figura 6.16. En el árbol sintáctico de una expresión, el valor de un nodo es el resultado de aplicar el operador en el nodo utilizando como operandos los valores de los hijos. Las hojas se evalúan a sí mismas. Por consiguiente, el árbol sintáctico de la expresión de la Figura 6.16 se evalúa a `(a+b)*(c-d)`. Los árboles sintácticos de expresiones, y el análisis sintáctico del árbol correspondiente, son estructuras de datos esenciales en las etapas de análisis sintáctico y generación de código de un compilador. Se darán más detalles en la Sección 11.2.

El árbol de expresión en la Figura 6.16 es un *árbol binario*, pues el número de hijos está limitado a como mucho dos por nodo. Un uso importante de los árboles binarios se examina en la próxima sección.

```
1  // Interfaz Arbol; los detalles son parecidos a los de Lista
2
3  public interface Arbol
4  {
5      Boolean esVacio( );
6      void    vaciar( );
7  }
8
9
10 // Interfaz ArbolIter; mantiene la posición actual
11
12 public interface ArbolIter
13 {
14     void    insertar( Object x ) throws ElementoNoEncontrado;
15     boolean buscar( Object x );
16     void    eliminar( Object x ) throws ElementoNoEncontrado;
17
18     void    irARaiz( );
19     void    primerHijo( );
20     void    siguienteHermano( );
21     boolean esValido( );
22     Object  recuperar( );
23 }
```

Figura 6.15 Interfaces para la clase abstracta de los árboles y su iterador.

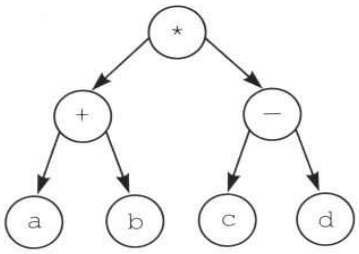


Figura 6.16 Árbol de la expresión `(a+b)*(c-d)`.

6.6 Árboles binarios de búsqueda

En la Sección 5.6, examinamos el problema de la búsqueda estática, y vimos que si los elementos se nos dan ordenados, podemos realizar la operación `buscar` en tiempo logarítmico, en el caso peor. Se trata de búsqueda estática, pues una vez que se nos dan los elementos, no podemos añadir ni eliminar ninguno.

Supongamos ahora que sí necesitamos insertar o eliminar elementos dinámicamente. Una estructura de datos que lo permite son los *árboles binarios de búsqueda*. La Figura 6.17 muestra las operaciones básicas permitidas con los árboles binarios de búsqueda. Su interfaz se muestra en la Figura 6.18. Observe que sólo objetos que implementen la interfaz `Comparable` se pueden almacenar en un árbol binario de búsqueda.

Los *árboles binarios de búsqueda* permiten inserción, eliminación y búsqueda.

El conjunto de operaciones permitidas se extiende para permitir aplicaciones arbitrarias de `buscar` (por nombre) así como de `insertar` y `eliminar`. El método `buscar` devuelve una referencia al objeto que se ajusta al elemento buscado.

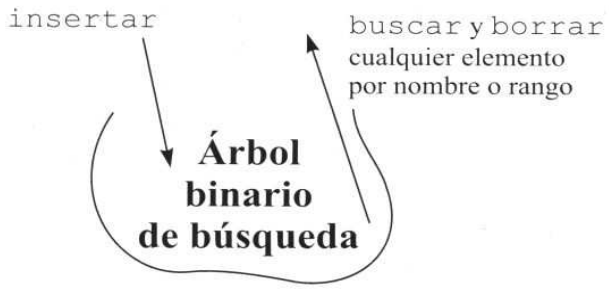


Figura 6.17 Modelo de árbol binario de búsqueda. La búsqueda binaria se generaliza para permitir inserciones y eliminaciones.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Interfaz ArbolBusqueda
7  //
8  // ******************OPERACIONES PÚBLICAS*********************
9  // void insertar( x )       --> Inserta x
10 // void eliminar( x )       --> Elimina x
11 // void eliminarMin( )      --> Elimina el menor elemento
12 // Comparable buscar( x )   --> Devuelve el elemento que ajusta con x
13 // Comparable buscarMin( )  --> Devuelve el menor elemento
14 // Comparable buscarMax( )  --> Devuelve el mayor elemento
15 // boolean esVacio( )       --> Devuelve true si vacío; si no false
16 // void vaciar( )           --> Elimina todos los elementos
17 // void imprimirArbol( )    --> Imprime el árbol ordenadamente
18 // ********************ERRORES*******************************
19 // Muchas rutinas lanzan ElementoNoEncontrado en varias
20 // condiciones degeneradas
21 // insertar lanza ElementoDuplicado si el elemento ya está en el árbol
22
23 public interface ArbolBusqueda
24 {
25    void       insertar( Comparable x ) throws ElementoDuplicado;
26    void       eliminar( Comparable x ) throws ElementoNoEncontrado;
27    void       eliminarMin( )           throws ElementoNoEncontrado;
28    Comparable buscar( Comparable x )   throws ElementoNoEncontrado;
29    Comparable buscarMin( )             throws ElementoNoEncontrado;
30    Comparable buscarMax( )             throws ElementoNoEncontrado;
31    boolean    esVacio( );
32    void       vaciar( );
33    void       imprimirArbol( );
34 }
```

Figura 6.18 Interfaz de árbol binario de búsqueda.

Este ajuste se basa en el método `compara` que debe ser implementado por los objetos de la clase `Comparable`. Si un elemento no se encuentra, `buscar` lanza una excepción. Ésta es una decisión de diseño; otra posibilidad habría sido devolver simplemente `null` cuando no se encuentra el elemento buscado. La diferencia bá-

sica de estos dos enfoques es que en el nuestro el programador debe explícitamente proporcionar código para el caso en el que la búsqueda falle. En el enfoque alternativo, si se devuelve `null` y no se hace ninguna comprobación, el resultado podría ser una `NullPointerException` en tiempo de ejecución. En términos de eficiencia, la versión con excepción puede ser menos eficiente, pero seguramente las consecuencias no se notarían, excepto quizá en alguna sección de código que se ejecutara muy a menudo. Sin embargo, al forzar al programador a tratar este caso, hemos hecho que el código sea menos susceptible a un error en tiempo de ejecución.

De la misma manera, la inserción de un elemento que ya se encuentra en el árbol se señala con la excepción `ElementoDuplicado`. De nuevo otras alternativas son posibles; una es permitir que la nueva inserción sobrescriba el valor almacenado. Esto es lo que se hace en la Sección 6.7, cuando se discuten las tablas hash.

Consideramos ahora un árbol binario de búsqueda que almacena cadenas de caracteres. Ya que sólo se pueden utilizar objetos `Comparable`, no podemos utilizar directamente el tipo `String`. De modo que definimos una clase `MiCadena`, mostrada en la Figura 6.19 (esta clase también implementa el interfaz `Hashable`, que se utiliza en la Sección 6.7). La Figura 6.20 muestra cómo funciona un árbol binario de búsqueda con objetos `MiCadena`.

Utilizando un árbol binario de búsqueda, podemos acceder al *K*-ésimo menor elemento. El coste es logarítmico en promedio, con una implementación simple, y logarítmico en el caso peor, con una implementación más cuidadosa.

Nuestra interfaz `ArbolBusqueda` también proporciona dos métodos adicionales: uno para buscar el elemento mínimo y otro para buscar el elemento máximo. Resulta que con un poco de trabajo adicional, también podemos soportar acceso al *K*-ésimo menor elemento, para un *K* cualquiera dado como parámetro. Esto se conoce como *búsqueda por posición en el orden*.

```
1  import Soporte.*;
2  import EstructurasDatos.*;
3
4  public final class MiCadena implements Comparable, Hashable
5  {
6      public MiCadena( String x )
7        { valor = x; }
8
9      public String toString( )
10       { return valor; }
11
12     public int compara( Comparable lder )
13       { return valor.compareTo( ((MiCadena) lder).valor ); }
14
15     public boolean menorQue( Comparable lder )
16       { return compara( rhs ) < 0; }
17
18     public boolean equals( Object lder )
19       { return valor.equals( ((MiCadena) lder).valor ); }
20
21     public int hash( int tamanyoTabla )
22       { return TablaExploracionCuadratica.hash( valor,
23                                                 tamanyoTabla ); }
24
25     private String valor;
26 }
```

Figura 6.19 Clase `MiCadena` para probar los programas de las Figuras 6.20 y 6.23.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar los árboles de búsqueda
5
6  public final class TestArbolBusqueda
7  {
8      public static void main( String [ ] args )
9  {
10         ArbolBusqueda a = new ArbolBinarioBusqueda ( );
11         MiCadena resultado = null;
12
13         try { a.insertar( new MiCadena( "Susana" ) ); }
14         catch( ElementoDuplicado e ) { } // No puede ocurrir
15
16         try
17         {
18            resultado =
19                 (MiCadena) a.buscar( new MiCadena( "Susana" ) );
20            System.out.print( "Encontrada " + resultado + ";" );
21         }
22         catch( ElementoNoEncontrado e )
23           { System.out.print( "Susana no encontrada;" ); }
24
25         try
26         {
27            resultado =
28                 (MiCadena) a.buscar( new MiCadena( "Juan" ) );
29            System.out.print( "Encontrado " + resultado + ";" );
30         }
31         catch( ElementoNoEncontrado e )
32           { System.out.print( " Juan no encontrado;" ); }
33
34          System.out.println( );
35     }
36 }
```

Figura 6.20 Programa ejemplo con árboles de búsqueda: la salida es `Encontrada Susana; Juan no encontrado;`.

Resumimos ahora los tiempos de ejecución de estas operaciones. Tenemos esperanzas de que el coste de los métodos `buscar`, `insertar` y `eliminar` sea, en el caso peor, $O(\log N)$, como sucede en el caso de la búsqueda binaria estática. Desgraciadamente, con la implementación más simple de los árboles binarios de búsqueda, esto no ocurre. El caso medio es en efecto logarítmico, pero el caso peor es $O(N)$, y puede presentarse con cierta frecuencia. Sin embargo, aplicando algunos trucos algorítmicos, podemos obtener una estructura algo más compleja, que sí tiene un coste $O(\log N)$ por operación.

¿Qué ocurre con los procedimientos `buscarMin`, `buscarMax` y el más general `buscarKesimo`? En el caso de la búsqueda binaria, son claramente operaciones constantes, porque indexamos un vector. En el Capítulo 18 veremos que estas operaciones tardan lo mismo en un árbol binario de búsqueda que la operación `buscar`, esto es, $O(\log N)$ en promedio, aunque $O(N)$ en el caso peor. Con suficiente cuidado, la cota en el caso peor puede ser reducida a $O(\log N)$. Como sugiere el nombre de la estructura, el árbol binario de búsqueda se implementa como

un árbol binario, y por tanto requiere una sobrecarga de dos referencias por elemento. Una variante más elaborada que soporta accesos eficientes en el caso peor necesita más espacio adicional por elemento. El Capítulo 18 da más detalles sobre la implementación de los árboles de búsqueda binarios.

6.7 Tablas hash

Muchas aplicaciones requieren búsqueda dinámica basada sólo en un nombre. La aplicación clásica es la *tabla de símbolos* de un compilador. Mientras compila un programa, el compilador debe guardar los nombres (incluyendo tipos, ámbito y asignación de memoria) de todos los identificadores declarados. Cuando encuentra un identificador fuera de una instrucción de declaración, el compilador comprueba si ya ha sido declarado. Si es así, el compilador extrae la información correspondiente de la tabla de símbolos.

Ya que los árboles binarios de búsqueda soportan un acceso en tiempo logarítmico a elementos con nombre arbitrarios, ¿por qué necesitamos una nueva estructura de datos? La respuesta es que un árbol binario podría producir un coste lineal en tiempo por acceso, y asegurar un coste logarítmico requeriría algoritmos bastante sofisticados.

Las *tablas hash* son una estructura de datos que evitan este caso peor, consiguiéndo además operaciones en tiempo constante, casi con seguridad. Un rendimiento degenerado es posible, pero extremadamente improbable. Por tanto, en la práctica el tiempo de acceso a cualquier elemento no depende del número de elementos en la tabla. Las tablas hash también evitan repetidas llamadas a las rutinas de gestión de la memoria. Esto las hace también más rápidas en la práctica. Un beneficio adicional de las tablas hash es que, al contrario de los árboles binarios de búsqueda, no requieren que los objetos implementados implementen la interfaz `Comparable`.

Las operaciones permitidas se muestran en la Figura 6.21, y la interfaz se muestra en la Figura 6.22. Ahora, la inserción de un elemento repetido no produce el lanzamiento de una excepción. En vez de ello, el nuevo elemento reemplaza al viejo. Las tablas hash sólo funcionan con elementos que implementan la interfaz `Hashable`. La interfaz `Hashable` requiere una *función hash*, que convierte el objeto `Hashable` en un entero. Tiene la siguiente declaración:

```
// Devuelve un entero entre 0 y tamanyoTabla - 1
int hash( int tamanyoTabla );
```

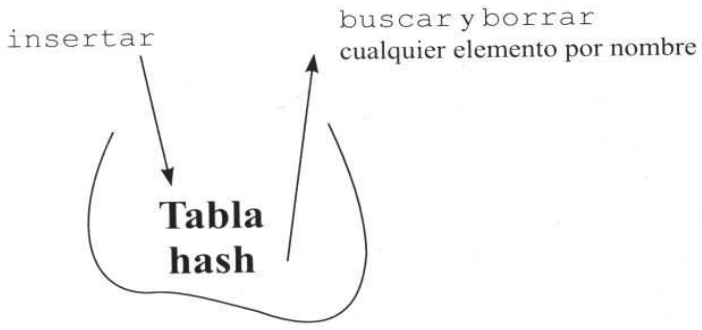


Figura 6.21 Modelo de las tablas hash: cualquier elemento con nombre puede ser accedido o borrado en un tiempo esencialmente constante.

Las *tablas hash* soportan inserciones, eliminaciones y búsquedas en tiempo constante en promedio.

Las tablas hash no son tan proclives al caso peor como lo son los árboles binarios de búsqueda básicos.

Para poder utilizar una tabla hash debemos proporcionar una función hash que convierta un objeto especifico en un entero.

```
1  package EstructurasDatos;
2
3  import Soporte.*;
4  import Excepciones.*;
5
6  // Interfaz TablaHash
7  //
8  // ********************OPERACIONES PÚBLICAS*******************
9  // void insertar( x )    --> Inserta x
10 // void eliminar( x )    --> Elimina x
11 // Hashable buscar( x ) --> Devuelve el elemento que ajusta con x
12 // void vaciar( )        --> Elimina todos los elementos
13 // ************************ERRORES*****************************
14 // buscar y eliminar lanzan ElementoNoEncontrado
15 // insertar sobrescribe el valor anterior en caso de repetición
16
17 public interface TablaHash
18 {
19     void     insertar( Hashable x );
20     void     eliminar( Hashable x ) throws ElementoNoEncontrado;
21     Hashable buscar( Hashable x )   throws ElementoNoEncontrado;
22     void     vaciar( );
23 }
```

Figura 6.22 Interfaz para la clase de tablas hash.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las tablas hash
5
6  public final class TestTablaHash
7  {
8      public static void main( String [ ] args )
9      {
10         TablaHash h = new TablaExploracionCuadratica( );
11         MiCadena resultado = null;
12
13         h.insertar( new MiCadena( new String( "Susana" ) ) );
14
15         try
16         {
17             resultado = (MiCadena)
18                         h.buscar( new MiCadena( "Susana" ) );
19             System.out.println( "Encontrada " + resultado );
20         }
21         catch( ElementoNoEncontrado e )
22           { System.out.println( "Susana no se ha encontrado" ); }
23     }
24 }
```

Figura 6.23 Programa de ejemplo con tablas hash. La salida es `Encontrada Susana`.

(Los detalles de cómo implementar la función hash se verán en el Capítulo 19, donde se proporciona una función que funciona para valores de tipo `String`.) El método `equals` debe ser también sobrescrito. La Figura 6.19 muestra cómo los objetos de `MiCadena` implementan la interfaz `Hashable`, proporcionando las funciones `hash` y `equals`. En la Figura 6.23, se muestra un ejemplo del uso de tablas hash sobre objetos `MiCadena`.

Un *diccionario* almacena claves y definiciones.

Un uso común de las tablas hash lo representan los diccionarios. Un *diccionario* almacena objetos formados por una clave, por la cual se busca en el diccionario, y su definición, que es lo que se devuelve. Podemos utilizar una tabla hash para implementar un diccionario instanciándola de la siguiente manera:

1. El objeto almacenado es de una clase que almacena la clave y su definición.
2. Las funciones de igualdad y desigualdad y la función hash se basan solamente en la parte clave del objeto almacenado.
3. Una búsqueda se realiza construyendo un objeto `e` con la clave y realizando `buscar` en la tabla hash.
4. La definición se obtiene utilizando una referencia `f` a la que se le asigna el valor devuelto por `buscar`. El campo definición de `f` es lo que buscamos.

6.8 Colas de prioridad

Las *colas de prioridad* permiten acceso únicamente al mínimo elemento.

Aunque los trabajos enviados a una impresora generalmente se colocan en una cola, ésta puede no siempre ser la mejor opción. Por ejemplo, un trabajo puede ser particularmente importante, por lo que querríamos ejecutarlo tan pronto como la impresora esté disponible. A la inversa, cuando la impresora termina un trabajo y están esperando varios trabajos de una página y un trabajo de 100 páginas, podría ser razonable imprimir el trabajo largo el último, aun cuando no sea el último trabajo enviado. (Desgraciadamente, la mayoría de los sistemas no hacen esto, lo que puede ser particularmente molesto en ocasiones.)

De la misma manera, en un entorno multiusuario, el planificador de tareas del sistema operativo debe decidir qué proceso ejecutará entre varios. Generalmente, a cada proceso sólo se le permite estar ejecutándose durante un periodo de tiempo fijo. Un posible algoritmo de gestión utiliza una cola. Los trabajos se colocan inicialmente al final de la cola. El planificador hará repetidamente lo siguiente: tomará el primer trabajo en la cola, lo ejecutará hasta que termine o agote su límite de tiempo, y lo volverá a colocar al final de la cola si no ha terminado. Generalmente, esta estrategia no es muy apropiada porque los trabajos cortos deben esperar mucho, por lo que parece que tardan mucho en ejecutarse. Claramente, los usuarios que están utilizando un editor no deberían ver con gran retraso los caracteres escritos. En consecuencia, los trabajos cortos (es decir, aquéllos que consumen pocos recursos) deberían tener preferencia sobre trabajos que ya han consumido gran cantidad de recursos. Pero también algunos trabajos de consumo intensivo, como por ejemplo los ejecutados por el administrador del sistema, podrían ser muy importantes, por lo que deberían también tener preferencia.

Si asignamos a cada trabajo un número para medir su prioridad, los valores menores (páginas impresas, recursos utilizados) asignarán una mayor importancia. Por tanto, queremos ser capaces de acceder al elemento menor de una colección

de elementos, y eliminarlo de la colección. Se obtienen así las operaciones `buscarMin` y `eliminarMin`. La estructura de datos que soporta estas operaciones es la denominada *cola de prioridad*. La Figura 6.24 ilustra las operaciones básicas de las colas de prioridad.

La Figura 6.25 muestra la interfaz de las colas de prioridad, y un programa ejemplo que ilustra su uso se muestra en la Figura 6.26. El programa ejemplo utiliza una cola de prioridad bastante sofisticada, denominada montículo de emparejamientos, que se discute en el Capítulo 22. Como es usual, uno no necesita saber nada acerca del montículo de emparejamientos, excepto que implementa la interfaz de las colas de prioridad.

Una vez más, debemos preguntar: ¿por qué no usar un árbol binario de búsqueda? Y una vez más, la respuesta es que los árboles binarios de búsqueda son innecesariamente potentes, tienen un rendimiento pobre en el caso peor, y requieren la sobrecarga de dos referencias a nodos por elemento. Utilizando un árbol binario de búsqueda sofisticado garantizaríamos un rendimiento logarítmico en el caso peor. Pero para lograrlo necesitaríamos un código elaborado que nos condu-

Figura 6.24 Modelo de cola de prioridad: sólo el elemento mínimo es accesible.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Interfaz ColaPrioridad
7  //
8  // ********************OPERACIONES PÚBLICAS*****************
9  // void insertar( x )        --> Inserta x
10 // Comparable buscarMin( )   --> Devuelve el elemento menor
11 // Comparable eliminarMin( ) --> Devuelve y elimina el elemento menor
12 // void vaciar( )            --> Elimina todos los elementos
13 // boolean esVacia( )        --> Devuelve true si vacía; si no, false
14 // ************************ERRORES****************************
15 // buscarMin y eliminarMin lanzan DesbordamientoInferior cuando vacía.
16
17 public interface ColaPrioridad
18 {
19     void       insertar( Comparable x );
20     Comparable buscarMin( )   throws DesbordamientoInferior;
21     Comparable eliminarMin( ) throws DesbordamientoInferior;
22     void       vaciar( );
23     boolean    esVacia( );
24 }
```

Figura 6.25 Interfaz para las colas de prioridad.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3  import Soporte.*;
4
5  // Programa simple para probar las colas de prioridad
6  public final class TestColaPrioridad
7  {
8      public static void main( String [ ] args )
9      {
10         ColaPrioridad cp = new MonticuloEmparejamientos( );
11
12         cp.insertar( new MiEntero ( 4 ) );
13         cp.insertar( new MiEntero ( 2 ) );
14         cp.insertar( new MiEntero ( 1 ) );
15         cp.insertar( new MiEntero ( 3 ) );
16         cp.insertar( new MiEntero ( 0 ) );
17
18         System.out.print( "Contenido:" );
19         try
20         {
21             for( ; ; )
22                 System.out.print( " " + cp.eliminarMin( ) );
23         }
24         catch( DesbordamientoInferior e ) { }
25
26         System.out.println( );
27     }
28 }
```

Figura 6.26 Programa ejemplo con colas de prioridad. La salida es: `Contenido: 0 1 2 3 4.`

ciría a un rendimiento injustificadamente bajo en la práctica[1]. Ya que las colas de prioridad soportan únicamente las operaciones `buscarMin` y `eliminarMin`, podríamos esperar un rendimiento que fuera un compromiso entre el tiempo constante de las colas y el tiempo logarítmico de los árboles binarios de búsqueda.

Éste es, en efecto, el caso. Las colas de prioridad básicas soportan todas las operaciones en un tiempo logarítmico en el caso peor, no requieren el coste adicional de dos referencias a otros nodos, soportan la inserción en tiempo constante en promedio y son muy fáciles de implementar. La estructura resultante se conoce como *montículo binario* y es una de las estructuras más elegantes conocidas. El Capítulo 20 da detalles sobre la implementación de los montículos binarios.

Los *montículos binarios* implementan las colas de prioridad con coste logarítmico por operación, utilizando un vector.

El montículo de emparejamientos, utilizado en la Figura 6.26, es una alternativa complicada, útil cuando se necesita cierta funcionalidad adicional, incluyendo la habilidad de combinar eficientemente dos colas de prioridad separadas.

Una aplicación importante de las colas de prioridad se encuentra en el área de la *simulación dirigida por eventos*. Considere, por ejemplo, un sistema como un banco en el que los clientes llegan y esperan en fila hasta que una de las K ventanillas está disponible. La llegada de los clientes está gobernada por una función de distribución de probabilidad, así como el tiempo de servicio (el tiempo necesario para que el empleado proporcione el servicio completo a un cliente). Estamos intere-

Una aplicación importante de las colas de prioridad es la *simulación dirigida por eventos*.

[1] Sin embargo, los árboles de ensanchamiento, discutidos en el Capítulo 21, son una notable excepción en algunas aplicaciones.

sados en estadísticas como ¿cuánto tiempo debe esperar un cliente en promedio? o ¿cómo de larga podría ser la fila?

Con ciertas distribuciones de probabilidad y para ciertos valores de K, podemos calcular estos valores de forma exacta. Sin embargo, al hacerse K mayor, el análisis se hace considerablemente más difícil, por lo que hay que usar un computador para simular el funcionamiento del banco. De esta forma, los gerentes del banco pueden determinar cuántas ventanillas son necesarias para asegurar un servicio razonable. Una *simulación dirigida por eventos* consiste en el procesamiento de eventos. Aquí los dos eventos son: (1) la llegada de un cliente y (2) la salida de un cliente, dejando libre una ventanilla. En cualquier momento, tenemos una colección de eventos esperando a producirse. Para ejecutar la simulación, debemos determinar el *siguiente* evento que se producirá. Éste es el evento cuyo tiempo de acontecimiento es mínimo; por tanto, para procesar la lista de eventos eficientemente se utiliza una cola de prioridad que extrae el elemento con tiempo mínimo. En la Sección 13.2 se presenta una discusión completa y una implementación de la simulación dirigida por eventos.

Resumen

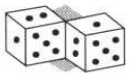
En este capítulo se han examinado las estructuras de datos básicas utilizadas en el resto del libro. Para cada una de ellas se ha proporcionado una interfaz e indicado cual debería ser el tiempo de ejecución de las operaciones. En la Parte IV daremos una implementación de dichas estructuras de datos que se ajuste a las cotas de tiempo afirmadas aquí. La Figura 6.27 resume los resultados que se obtendrán.

En el próximo capítulo estudiamos una herramienta muy importante para resolver problemas: la *recursión*. La recursión permite resolver de forma eficiente muchos problemas, utilizando algoritmos cortos y sencillos, y es fundamental para la implementación de los árboles binarios de búsqueda.

Estructura de datos	Acceso	Comentarios
Pilas	Sólo al más reciente, `cima`, $O(1)$	Muy, muy rápidas
Colas	Sólo al menos reciente, `quitarPrimero`, $O(1)$	Muy, muy rápidas
Listas enlazadas	Cualquier elemento	$O(N)$
Árboles de búsqueda	Cualquier elemento, por nombre o rango, $O(\log N)$	Caso medio; también puede lograrse en el caso peor
Tablas hash	Cualquier elemento, con nombre, $O(1)$	Casi seguro
Colas de prioridad	`buscarMin`, $O(1)$ `eliminarMin`, $O(\log N)$	`insertar` es $O(1)$ en promedio, $O(\log N)$ en el caso peor

Figura 6.27 Resumen de algunas estructuras de datos.

Elementos del juego



análisis sintáctico de precedencia de operadores Algoritmo que utiliza una pila para evaluar expresiones.

árbol Estructura de datos muy utilizada, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos. En todo este libro, asumimos que los árboles tienen raíz.

árbol binario Un árbol con, a lo sumo, dos hijos por nodo.

árbol binario de búsqueda Un árbol que permite la inserción, eliminación y búsqueda. También se puede utilizar para acceder al *K*-ésimo menor elemento. El coste en tiempo es logarítmico en promedio para una implementación simple, y logarítmico en el caso peor, con una implementación más cuidada.

árbol con raíz Árbol con un nodo designado como raíz.

árbol sintáctico de una expresión Un árbol utilizado en análisis sintáctico, en el cual el valor de un nodo es el resultado de aplicar el operador en el nodo, utilizando los valores de sus hijos como operandos.

clase iteradora Clase que permite acceder a una lista. La clase de la lista tiene métodos que reflejan el estado de la lista; el resto de operaciones están en la clase iterador.

cola Estructura de datos que limita el acceso al elemento insertado menos recientemente.

cola de prioridad Estructura de datos que soporta el acceso únicamente al menor elemento.

diccionario Almacena claves, que son las que se buscan en el diccionario, y sus correspondientes definiciones.

estructura de datos Una representación de los datos junto con unas operaciones permitidas sobre dichos datos. Las estructuras de datos nos permiten lograr la reutilización de componentes.

función hash Función que convierte un objeto `Hashable` en un entero.

hoja En un árbol, un nodo sin hijos.

longitud de un camino En un árbol, el número de aristas que hay que atravesar, desde la raíz, para alcanzar un nodo.

montículo binario Implementa sobre un vector las colas de prioridad con un tiempo logarítmico por operación.

pila Estructura de datos que restringe el acceso al elemento más recientemente insertado.

tabla de símbolos Estructura de datos utilizada por un compilador para almacenar los identificadores. Implementada generalmente con una tabla hash.

tabla hash Estructura de datos que soporta inserciones, eliminaciones y búsquedas, en tiempo constante en promedio.

Errores comunes

1. No documentar correctamente las características de la interfaz de una clase es un error grave.
2. Es un error acceder o eliminar de una pila, cola o cola de prioridad, vacía. El que implementa la clase debe asegurarse de detectar el error y lanzar una excepción. Las operaciones de acceso o eliminación están permitidas

en árboles o tablas hash vacías porque el acceso en un árbol o tabla hash vacía es un caso particular de búsqueda sin éxito.
3. Pueden ocurrir varios errores mientras se accede a una lista o a un árbol. Éstos deben ser señalizados por la clase.
4. La implementación de un objeto `Hashable` debe suministrar una buena función hash; en caso contrario, se deteriorará el rendimiento.
5. La función `equals` (que toma un `Object` como parámetro) debería escribirse para los objetos que se insertan en tablas hash o árboles de búsqueda.
6. Una cola de prioridad no es una cola. Sólo suena igual.

En Internet

Algunos de los programas de prueba de este capítulo, incluyendo la clase `MiCadena`, están en el directorio **Chapter06**. Por otra parte, las interfaces para las clases que serán implementadas en las Partes IV y V se encuentran en el directorio **DataStructures**, siendo parte del paquete `DataStructures`, traducido como `EstructurasDatos`. Se tratan, en concreto, de los siguientes:

Stack.java	La interfaz `Stack`, traducida por `Pila` en la Figura 6.4.
Queue.java	La interfaz `Queue`, traducida por `Cola` en la Figura 6.7.
List.java	La interfaz `List`, traducida por `Lista` en la Figura 6.11.
ListItr.java	La interfaz `ListItr`, traducida por `ListaIter` en la Figura 6.12.
SearchTree.java	La interfaz `SearchTree`, traducida por `ArbolBusqueda` en la Figura 6.18.
HashTable.java	La interfaz `HashTable`, traducida por `TablaHash` en la Figura 6.22.
PriorityQueue.java	La interfaz `PriorityQueue`, traducida por `ColaPrioridad` en la Figura 6.25.

Ejercicios

Cuestiones breves

6.1. Muestre los resultados de la siguiente secuencia de instrucciones: `insertar(4)`, `insertar(8)`, `insertar(1)`, `insertar(6)`, `eliminar()` y `eliminar()`, cuando `insertar` y `eliminar` son las operaciones básicas de

a) Las pilas.
b) Las colas.
c) Las colas de prioridad.

Problemas teóricos

6.2. Suponga que quiere soportar, exclusivamente, las tres operaciones: `insertar`, `buscarMax` y `eliminarMax`. ¿Cómo de rápido cree que se pueden implementar estas operaciones?

6.3. ¿Pueden todas las operaciones siguientes ser implementadas en tiempo logarítmico: `insertar`, `eliminarMin`, `eliminarMax`, `buscarMin` y `buscarMax`?

6.4. ¿Qué estructuras de datos de la Figura 6.27 conducirían a algoritmos de ordenación que se ejecutaran en un tiempo menor que cuadrático?

6.5. Muestre que es posible soportar simultáneamente las operaciones siguientes en tiempo constante: `apilar`, `cima`, y `buscarMin`. Observe que `eliminarMin` no forma parte del repertorio. *Pista*: mantenga dos pilas, una para almacenar los elementos y otra para almacenar los mínimos según van apareciendo.

6.6. Una cola doble permite inserciones y eliminaciones en ambos extremos de la línea. ¿Cuál cree que es el tiempo de ejecución por operación?

Problemas prácticos

6.7. Escriba una rutina que imprima en orden inverso los elementos de una lista enlazada. Para hacerlo, recorra la lista con una `ListaIter`, apilando cada elemento en una pila. Cuando alcance el fin de la lista enlazada, extraiga repetidamente la `cima` hasta que la pila se quede vacía.

6.8. Una eliminación en una lista enlazada o en un árbol de búsqueda binario, nos provoca el siguiente problema: cuando el elemento actual es borrado, ¿cuál se convierte en el nuevo elemento actual? Discuta varias alternativas.

6.9. Muestre como implementar eficientemente una pila utilizando una lista enlazada como atributo.

6.10. Muestre como implementar eficientemente una cola utilizando una lista enlazada como atributo y manteniendo un objeto `ListaIter` que apunta siempre al último elemento de la lista enlazada.

6.11. Partiendo de una clase de tablas hash, implemente una clase `Diccionario` que permita las operaciones de `insertar` y `buscar`. Para facilitar su tarea incluimos a continuación algunas de las líneas de esta clase.

```
public class Diccionario
{
      // Algunos de los métodos
    void insertar( Hashable clave, Object definicion );
    Object buscar( Hashable clave ) throws ElementoNoEncontrado;
}
```

Prácticas de programación

6.12. Una pila puede ser implementada utilizando un vector y manteniendo el tamaño actual. Los elementos de la pila se almacenan en posiciones consecutivas del vector, con el elemento de la cima siempre en la posición 0. Observe que éste no es el método más eficiente. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `apilar`, `desapilar` y `cima`.
b) ¿Cuál es el tiempo de ejecución en notación *O* de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.13. Una cola puede ser implementada utilizando un vector y manteniendo su tamaño actual. Los elementos de la cola se almacenan en posiciones consecutivas del vector, con el primer elemento siempre en la posición 0. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `primero`, `insertar` y `quitarPrimero`.
b) ¿Cuál es el tiempo de ejecución en notación *O* de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

Éste no es el método más eficiente de implementar una cola utilizando un vector. Trate de localizar la razón principal por la que alguna de las operaciones es ineficiente, y a partir de la idea de corregir esta situación proponga una implementación alternativa que resulte plenamente satisfactoria.

6.14. Las operaciones permitidas por un árbol de búsqueda pueden implementarse también con un vector, manteniendo su tamaño actual. Los elementos del vector se almacenan ordenados en posiciones consecutivas del vector. En consecuencia, `buscar` puede implementarse con una búsqueda binaria. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `insertar` y `eliminar`.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.15. Una tabla hash puede implementarse utilizando un vector, manteniendo su tamaño actual. Los elementos del vector se almacenan en posiciones consecutivas del vector, pero no de forma ordenada. En lugar de ello, se van insertando en la siguiente posición libre del vector. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `insertar`, `eliminar` y `buscar`.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.16. Una cola de prioridad puede implementarse utilizando un vector ordenado, como se indicó en el Ejercicio 6.14. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin` e `insertar`.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.17. Una cola de prioridad puede implementarse almacenando los elementos en un vector sin ordenar e insertando elementos en la siguiente posición libre, como se indicó en el Ejercicio 6.15. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin` e `insertar`.

b) ¿Cuál es el tiempo de ejecución, en notación *O*, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.18. Añadiendo un atributo más a la clase de las colas de prioridad del Ejercicio 6.17, es posible implementar `insertar` y `buscarMin` en tiempo constante. El atributo extra almacenará la posición del vector donde se encuentra el mínimo. Sin embargo, `eliminarMin` seguirá siendo costosa. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `insertar` y `eliminarMin`.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de `eliminarMin`?
c) Escriba una implementación en Java de dichos algoritmos.

6.19. Una cola de prioridad doble permite accesos tanto al elemento mínimo como al máximo. En consecuencia, se permiten todas las operaciones siguientes: `buscarMin`, `eliminarMin`, `buscarMax` y `eliminarMax`. Discutir distintas implementaciones posibles basadas en un vector, haciendo para cada una de ellas lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin`, `buscarMax` y `eliminarMax`.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.20. Un montículo de medianas soporta las siguientes operaciones: `insertar`, `buscarKesimo` y `eliminarKesimo`. Las dos últimas buscan y eliminan, respectivamente, el *K*-ésimo menor elemento. La implementación más simple mantiene los elementos del montículo ordenados en un vector. Se pide hacer lo siguiente:

a) Describa los algoritmos que pueden ser utilizados para soportar las operaciones de los montículos de medianas.
b) ¿Cuál es el tiempo de ejecución, en notación *O*, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

Bibliografía

La bibliografía correspondiente a las estructuras de datos presentadas en este capítulo se proporciona en la Parte IV.