CAPÍTULO

16 Listas enlazadas

En el Capítulo 15 se mostró que se pueden usar las listas enlazadas para almacenar elementos de forma no contigua. Las listas enlazadas usadas en ese capítulo eran simples debido a que todos los accesos se realizaban en uno de los dos extremos. En este capítulo veremos:

- Cómo permitir el acceso a cualquier elemento usando una lista enlazada general.
- Los algoritmos generales usados para las operaciones sobre listas enlazadas.
- Cómo la *clase iteradora* proporciona un mecanismo seguro para recorrer y acceder a las listas enlazadas.
- Variaciones de las listas, como las *listas doblemente enlazadas* y las *listas enlazadas circulares*.
- Cómo podemos usar la herencia para derivar una clase de *listas enlazadas ordenadas*.

16.1 Ideas básicas

Este capítulo implementa las listas enlazadas y permite acceso general (operaciones de inserción, eliminación y consulta arbitrarias de un elemento) a la lista. La lista enlazada básica consta de una colección de nodos situados en la memoria dinámica conectados entre sí. En una lista simplemente enlazada, cada nodo se compone de un dato y una referencia al siguiente nodo de la lista. El último nodo de la lista contiene una referencia siguiente `null`. Supondremos que el nodo viene dado por la declaración `NodoLista` del Capítulo 15:

```
class NodoLista
{
    Object    dato;
    NodoLista siguiente;
        // Constructores
}
```

Se puede acceder al primer nodo de la lista a través de un puntero, lo cual se muestra en la Figura 16.1. Podemos imprimir la lista enlazada o buscar en ella comenzando en el primer elemento y siguiendo la cadena de punteros `siguiente`.

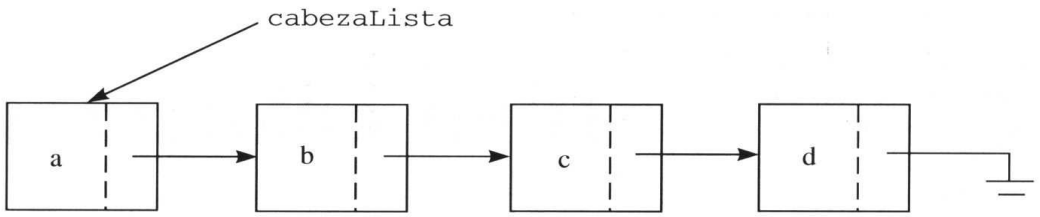

Figura 16.1 Lista enlazada básica.

Las dos operaciones básicas que se pueden realizar son la inserción y eliminación de un elemento arbitrario `x`.

La inserción consiste en introducir un nodo dentro de la lista y se puede llevar a cabo con una instrucción.

En el caso de la inserción, debemos determinar dónde va a tener lugar la inserción. Si partimos de una referencia a algún nodo de la lista, el lugar donde más fácilmente se puede insertar es el inmediatamente posterior a ese elemento. Como ejemplo, en la Figura 16.2 se muestra cómo insertamos `x` después del elemento `a` en una lista enlazada. Debemos llevar a cabo los siguientes pasos:

```
tmp = new NodoLista( );                  // Crear un nodo nuevo
tmp.dato = x;                            // Colocar x en el campo dato
tmp.siguiente = actual.siguiente;        // El siguiente al nodo x es b
actual.siguiente = tmp;                  // El siguiente al nodo a es x
```

Como resultado de estas instrucciones, la lista anterior `...a,b,...` ahora es `...a,x,b,...`. Podemos simplificar el código, puesto que `NodoLista` tiene un constructor que inicializa los atributos directamente. Así, obtenemos

```
tmp = new NodoLista( x, actual.siguiente ); // Crear un nuevo nodo
actual.siguiente = tmp;     // El siguiente al nodo a es x
```

En este punto, queda claro que ya no se necesita `tmp`, por lo que podemos limitarnos a la siguiente línea:

```
actual.siguiente = new NodoLista( x, actual.siguiente );
```

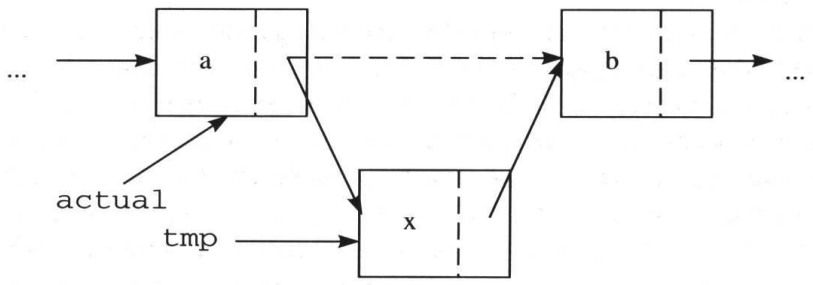

Figura 16.2 Inserción en una lista enlazada: se crea un nodo nuevo (`tmp`), se copia `x` en él y se determinan los valores de la referencia `siguiente` de `tmp` y de `actual`.

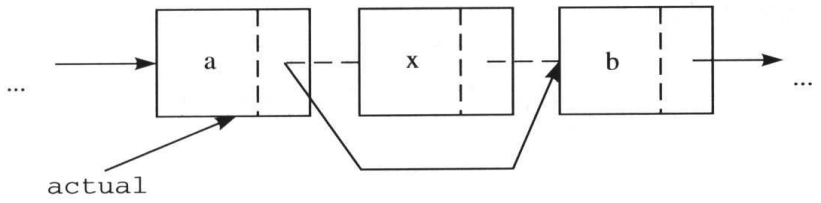

Figura 16.3 Eliminación en una lista enlazada.

La eliminación se puede ejecutar con un simple cambio de una referencia. La Figura 16.3 muestra que para eliminar el elemento `x` de la lista enlazada, hacemos que `actual` sea el nodo anterior a `x`, tras lo que hacemos que la referencia `siguiente` de `actual` salte por encima de `x`. Esto se expresa mediante la siguiente instrucción

La eliminación se puede llevar a cabo saltando por encima del nodo. Necesitamos una referencia al nodo anterior al que queremos eliminar.

```
actual.siguiente = actual.siguiente.siguiente;
```

La lista `...a,x,b,...` es ahora `...a,b,...`.

Esto resume el comportamiento básico de insertar y eliminar en posiciones arbitrarias de una lista enlazada. La propiedad fundamental de las listas enlazadas es que se pueden hacer cambios en una lista realizando una cantidad constante de movimientos de datos, lo cual es una gran mejora respecto a la implementación usando vectores, ya que la contigüidad en un vector implica que siempre que se añade o elimina un elemento, se deben desplazar todos los elementos que le siguen.

Las operaciones sobre las listas enlazadas solamente realizan una cantidad constante de movimientos de datos.

16.1.1 Nodos cabecera

Un problema de la descripción básica es que asume que siempre que se elimina un elemento, hay algún elemento anterior a él. En consecuencia, la eliminación del primer elemento de una lista enlazada se convierte en un caso especial. De forma análoga, la rutina de inserción no nos permite insertar un elemento en la primera posición de la lista, ya que las inserciones están restringidas a aquellas posiciones posteriores a alguna otra. Así, aunque el algoritmo básico funciona bien, hay algunos casos especiales molestos que debemos tratar.

Los *nodos cabecera* nos evitan tener que tratar de forma explícita casos especiales, como la inserción de un elemento en la primera posición o la eliminación del primer elemento.

Los casos especiales son siempre problemáticos en el diseño de algoritmos y conducen con frecuencia a errores en el código. Por ello es preferible escribir código que evite dichos casos especiales. Una forma de lograr esto en este caso consiste en introducir un *nodo cabecera*.

Un nodo cabecera es un nodo extra de la lista enlazada que no guarda ningún dato pero que sirve para satisfacer el requerimiento de que cada nodo que contenga un elemento tenga un nodo anterior. En la Figura 16.4 se muestra el nodo cabecera para la lista `a,b,c`. Observe que `a` ya no es un caso especial. Se puede borrar como cualquier otro nodo, colocando `actual` en el nodo anterior a él. También podemos añadir fácilmente un nuevo elemento al principio de la lista, colocando `actual` en el nodo cabecera y llamando a la rutina de inserción. Usando el nodo cabecera, simplificamos el código a cambio de una penalización despreciable en espacio. En aplicaciones más complejas, la cabecera no solamente simplifica el código sino que también aumenta la velocidad, ya que, a la postre, un menor número de tests requiere un menor tiempo.

El nodo cabecera no guarda ningún dato, pero sirve para satisfacer el requerimiento de que cada nodo tenga uno anterior.

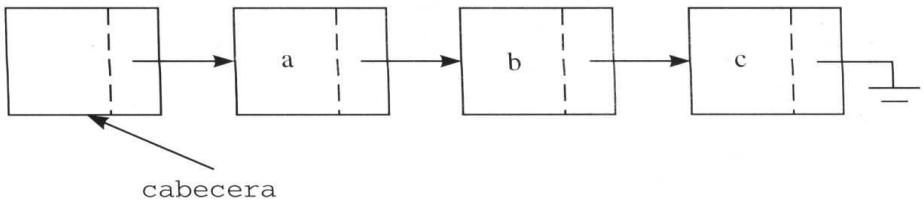

Figura 16.4 Uso de un nodo cabecera en la lista enlazada.

El uso de nodos cabecera es controvertido. Algunos argumentan que el hecho de evitar casos especiales no es justificación suficiente para añadir celdas ficticias, considerando el uso del nodo cabecera como poco un truco sucio al viejo estilo de los primeros programadores. Aun así, nosotros los usaremos pues nos permiten concentrarnos en las manipulaciones básicas de los punteros sin oscurecer el código con casos especiales. Si se deben o no usar nodos cabecera es cuestión de preferencias personales. Además, en una implementación de la clase su uso sería completamente transparente al usuario. Pero debemos ser cuidadosos; la rutina de impresión no debe imprimir el contenido de la cabecera y las rutinas de búsqueda deben saltarse ese nodo. Colocarnos al principio de la lista ahora significa colocar la posición actual en `cabecera.siguiente`. Como se muestra en la Figura 16.5, una lista con nodo cabecera es vacía si `cabecera.siguiente` es `null`.

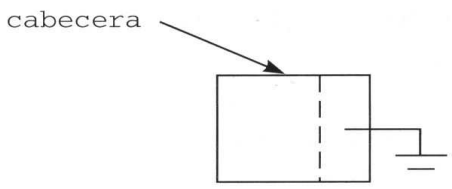

Figura 16.5 Lista vacía cuando se usa un nodo cabecera.

16.1.2 Clases iteradoras

Almacenando un puntero al nodo actual en la clase de las listas, nos aseguramos de que el acceso es controlado.

La *clase iteradora* mantiene una posición actual y lleva a cabo todas las rutinas que dependen del conocimiento de la posición en la lista. La clase iteradora está emparejada con la clase `Lista`.

La estrategia primitiva habitual identifica una lista enlazada con un puntero al nodo cabecera. Entonces se puede acceder a cada elemento individual de la lista proporcionando un puntero al nodo que lo contiene. El problema con esta estrategia es que es difícil comprobar si se producen errores. Un usuario podría cambiar un puntero de forma que apuntase a algo que está en otra lista. Una forma de garantizar que esto no puede pasar es almacenar la posición actual como parte de la clase `Lista`. Para ello, añadimos un segundo atributo, `actual`. Entonces, puesto que todos los accesos a la lista se realizan a través de los métodos de la clase, podemos estar seguros de que `actual` siempre representa un puntero a un nodo de la lista, un puntero al nodo cabecera o bien es igual a `null`.

Este esquema tiene un problema: puesto que solamente se dispone de una posición, no se soporta el caso en que dos iteradores necesiten acceder independientemente a la lista. Una forma de evitar este problema consiste en definir una *clase iteradora* separada. La clase `Lista` no mantendría entonces ninguna noción de posición actual y solamente contendría los métodos que tratan a la lista como una unidad, como `esVacia` y `vaciar`. Las rutinas que dependen del conocimiento de

la posición de la lista en la que nos encontramos se encontrarían en la clase iteradora. Ésta mantiene una noción de la posición actual y el nodo cabecera de la lista que representa (el puntero al nodo cabecera puede ser inicializado por el constructor). El posibilitar el acceso a la lista se consigue haciendo que la clase iteradora pertenezca al mismo paquete.

Podemos ver cada instancia de una clase iteradora como un marco en el que solamente se permiten operaciones legales sobre las listas como, por ejemplo, avanzar en la lista. En este capítulo implementaremos la interfaz `Lista` mediante la clase `ListaEnlazada` y la interfaz `ListaIter` mediante la clase `ListaEnlazadaIter`.

Para ver cómo funciona todo esto, estudiemos el método estático mostrado en la Figura 16.6, que se puede usar en cualquier punto, el cual devuelve la longitud de una lista enlazada. Se declara `itr` como un iterador que puede acceder a la lista enlazada `laLista`. Esto requiere el uso de un constructor para `ListEnlazadaIter` que tome una `ListaEnlazada` como parámetro.

La variable `itr` se inicializa con el primer elemento de `laLista` (saltando por encima del nodo cabecera), pero de cualquier modo llamamos al método `primero`. La llamada al método `estaDentro` refleja el test `p!=null`, que se haría si `p` fuera un puntero normal a un `NodoLista`. Finalmente, la llamada al método `avanzar` refleja la asignación `p=p.siguiente`.

Por tanto, puesto que la clase iteradora define unos pocos métodos simples, podemos iterar sobre las listas de una forma muy natural. La siguiente sección muestra cómo se implementa todo esto en Java. Las rutinas son increíblemente simples.

```
1  // Devuelve el número de elementos que contiene laLista
2  static public int longLista( ListaEnlazada laLista )
3  {
4      int longitud = 0;
5      ListaEnlazadaIter itr = new ListaEnlazadaIter( laLista );
6
7      for( itr.primero( ); itr.estaDentro( ); itr.avanzar( ) )
8          longitud++;
9      return longitud;
10 }
```

Figura 16.6 Método que devuelve la longitud de una lista.

16.2 Implementación en Java

Comenzamos la implementación de las listas enlazadas mostrando el esqueleto de su clase. Implementamos la interfaz descrita en la Sección 6.4. La clase `ListaEnlazada` se muestra en la Figura 16.7. En la línea 28, declaramos el único atributo de la clase, esto es, el puntero al nodo cabecera.

El constructor de la línea 20 asigna memoria al nodo cabecera. Observe que el atributo `dato` del nodo cabecera almacena la referencia `null`. Se declaran dos métodos, `esVacia` y `vaciar`, ambos consistentes en una sola línea. La clase `ListaEnlazada` es simple porque la mayor parte del trabajo se traslada a la clase iteradora.

La clase `ListaEnlazada` es simple porque la mayor parte del trabajo se traslada a la clase iteradora.

```
1  package EstructurasDatos;
2
3  // Clase ListaEnlazada
4  //
5  // CONSTRUCCIÓN: sin inicializador
6  // El acceso se realiza a través de la clase ListaEnlazadaIter
7  //
8  // ******************OPERACIONES PÚBLICAS*******************
9  // boolean esVacia( ) --> Devuelve true si es vacía
10 // void vaciar( )     --> Elimina todos los elementos
11 // ************************ERRORES*************************
12 // No hay errores especiales
13
14 /**
15  * Implementación de las listas usando listas enlazadas con nodo
16  * cabecera. El acceso a la lista es vía ListaEnlazadaIter.
17  */
18 public class ListaEnlazada implements Lista
19 {
20    public ListaEnlazada( )
21      { cabecera = new NodoLista( null ); }
22    public boolean esVacia( )
23      { return cabecera.siguiente == null; }
24    public void vaciar( )
25      { cabecera.siguiente = null; }
26
27      // Atributos amistosos, luego ListaEnlazadaIter puede acceder
28    NodoLista cabecera;    // Referencia al nodo cabecera
29 }
```

Figura 16.7 Clase `ListaEnlazada`.

El iterador está unido irrevocablemente a la lista enlazada para la cual fue construido.

Las Figuras 16.8 y 16.9 muestran el esqueleto de `ListaEnlazadaIter`. Ésta es una clase muy interesante que requiere una atención especial a los detalles. Los atributos de la clase son `laLista`, que es una referencia a la lista, y `actual`, que es una referencia al nodo actual. En el constructor se inicializa `laLista`, que después ya no debe cambiar. Luego un iterador está irrevocablemente unido a una sola lista[1].

Inicialmente, `actual` apunta al primer nodo, o a la cabecera si la lista es vacía.

En las líneas 11 a 16 se declara un constructor. El parámetro es un objeto `ListaEnlazada`, y como resultado de la construcción, `laLista` pasa a referenciar a ese parámetro y `actual` apunta al primer nodo de la lista (saltándose la cabecera, por supuesto). Si la lista es vacía, `actual` apuntará al nodo cabecera, ya que esperamos que la siguiente operación que se produzca sea `insertar`. No hay constructor de cero parámetros, por lo que si intentáramos declarar un objeto `ListaEnlazadaIter` sin un inicializador de lista, se produciría un error en tiempo de compilación. Por conveniencia, en las líneas 24 a 26 se proporciona otro constructor que acepta una `Lista`.

Nos quedan por ver los métodos de la clase. Tenemos que ser muy cuidadosos al explicar lo que hace cada método. Por ejemplo, `insertar` añade el elemento `x` en la posición inmediatamente posterior a `actual` en la lista. ¿Qué sucede si `actual` es `null`? Por supuesto, se produce un error. ¿Cuál es el nuevo valor de `actual`?

[1] En Java 1.1, esto se puede forzar haciendo que `laLista` sea un atributo `final`, ya que Java 1.1 permite la inicialización de un atributo `final` en los constructores. Sin embargo, en este libro no hemos usado esta característica.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ListaEnlazadaIter; mantiene la "posición actual"
6  //
7  // CONSTRUCCIÓN: con la ListaEnlazada a la que ListaIter
8  //      está permanentemente ligada
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // void insertar( x )     --> Inserta x tras la posición actual
12 // void eliminar( x )     --> Elimina x
13 // boolean buscar( x )    --> Coloca actual para poder ver x
14 // void cero( )           --> Coloca actual antes de la posición 1
15 // void primero( )        --> Coloca actual en la posición 1
16 // void avanzar( )        --> Avanza
17 // boolean estaDentro( ) --> True si es una posición válida
18 // Object recuperar       --> Elemento en la posición actual
19 // ************************ERRORES*************************
20 // Excepciones por accesos, inserciones o eliminaciones ilegales.
21
22 /**
23  * Implementación mediante listas enlazadas con nodo cabecera
24  *      del iterador de listas.
25  */
```

Figura 16.8 Sección de comentarios de `ListaEnlazadaIter`.

En nuestro caso, si la inserción ha tenido éxito, `actual` apuntará al nodo recientemente insertado, pero si no, no se modificará. Usamos en cada método comentarios *javadoc* para documentar el comportamiento, incluyendo la especificación de los errores que pueden aparecer y los cambios en el estado del iterador.

Discutamos el resto de los métodos. El método `buscar` busca `x`. Si `x` está en la lista, se mueve `actual` para apuntar al nodo que lo contiene y `buscar` devuelve `true`. En caso contrario, `buscar` devuelve `false` y `actual` no se ve modificado.

El método `eliminar` busca el elemento `x`. Si está en la lista lo elimina y `actual` pasa a apuntar al nodo cabecera. En caso contrario lanza una excepción y `actual` no se ve modificado. Es importante cambiar `actual`, ya que no deseamos en absoluto que apunte a un nodo que ya ha sido eliminado. Una alternativa perfectamente razonable sería que apuntara al nodo anterior al recientemente eliminado. Ninguno de estas dos alternativas es infalible, ya que si hubiera dos iteradores recorriendo la lista y uno de ellos eliminara un nodo, el otro recorrido podría verse afectado negativamente.

Si hubiera dos iteradores recorriendo la lista y uno de ellos eliminara un nodo, el otro podría verse afectado negativamente.

Una rutina que en ocasiones (*véase* la Sección 13.1) resulta muy útil, es `eliminarSig`, cuya implementación se deja como Ejercicio 16.9.

El método `estaDentro` se usa para comprobar si la posición actual está en la lista. No altera `actual`. El método `recuperar` devuelve el elemento situado en la posición actual (o bien `null` si la posición actual no es un nodo perteneciente a la lista). Tampoco modifica `actual`. El método `cero` coloca `actual` en el nodo cabecera, permitiendo así las inserciones al principio de la lista. El método `primero` coloca `actual` en el primer nodo de la lista, aunque si ésta es vacía le asignará `null`. Finalmente, `avanzar` es el método que lleva a cabo la iteración. Asigna

```
1  public class ListaEnlazadaIter implements ListaIter
2  {
3      /**
4       * Construye la lista.
5       * Como resultado, la posición actual es el primer
6       * elemento, a menos que la lista sea vacía, en cuyo caso
7       * la posición actual es el elemento 0-ésimo.
8       * @param unaLista un objeto ListaEnlazada a la que este
9       *     iterador está permanentemente ligado.
10      */
11     public ListaEnlazadaIter( ListaEnlazada unaLista )
12     {
13         laLista = unaLista;
14         actual = laLista.esVacia( ) ?
15                  unaLista.cabecera : unaLista.cabecera.siguiente;
16     }
17
18     /**
19      * Este constructor se proporciona por conveniencia. Si
20      * unaLista no es un objeto ListaEnlazada, se lanzará una
21      * excepción ClassCastException. En caso contrario, tiene
22      * el mismo comportamiento que el constructor de arriba.
23      */
24     public ListaEnlazadaIter( Lista unaLista ) throws
25                                                 ClassCastException
26       { this( ( ListaEnlazada ) unaLista );}
27
28     /**
29      * Coloca la posición actual en el nodo cabecera.
30      */
31     public void cero( )
32       { actual = laLista.cabecera; }
33
34     public void insertar( Object x ) throws ElementoNoEncontrado
35       { /* Figura 16.12 */ }
36     public boolean buscar( Object x )
37       { /* Figura 16.13 */ }
38     public void eliminar( Object x ) throws ElementoNoEncontrado
39       { /* Figura 16.14 */ }
40     public Object recuperar( )
41       { /* Figura 16.15 */ }
42     public void primero( )
43       { /* Figura 16.15 */ }
44     public void avanzar( )
45       { /* Figura 16.15 */ }
46     public boolean estaDentro( )
47       { /* Figura 16.15 */ }
48
49     protected ListaEnlazada laLista;    // Lista
50     protected NodoLista actual;         // Posición actual
51 }
```

Figura 16.9 Esqueleto para la clase `ListaEnlazadaIter`.

`actual.siguiente` a `actual`, siempre que `actual` no sea `null`, en cuyo caso no sucede nada.

Los métodos `primero`, `avanzar`, `estaDentro` y `eliminar` no lanzan excepciones porque el uso habitual de métodos en el seno de bucles `for` requeriría un bloque `try/catch` que en este caso no está justificado.

Podríamos haber añadido más métodos, pero este conjunto básico es bastante potente. Algunos métodos, como `retroceder`, no están soportados de forma eficiente con esta versión de las listas enlazadas. Más adelante en este capítulo, se estudian otras versiones de la lista enlazada que permiten la implementación en tiempo constante de éste y otros métodos. Por ahora veamos cómo se puede usar la clase `ListaEnlazadaIter` para implementar la copia y la impresión de las listas enlazadas.

El método `retroceder` no está soportado de forma eficiente. Al efecto se podría usar una lista doblemente enlazada si estuviera garantizada su fiabilidad.

En la Figura 16.10 se muestra la implementación de un método de copia de listas. En la línea 5, hacemos una comprobación de alias y terminamos inmediatamente si dicha comprobación devuelve `true`. En caso contrario, llamamos a `vaciar` para vaciar la lista destino. Después declaramos un iterador de listas (`itrIzq`) para recorrer la lista destino `lizq` y un segundo iterador (`itrDer`) para recorrer `lder`. Al principio, `itrIzq` apunta al nodo cabecera, pues la lista está vacía, e `itrDer` apunta al primer nodo de la lista `lder`. En las líneas 14 y 15 se ejecuta un bucle con el que llamamos repetidamente a `insertar` para añadir los elementos de `lder` en `lizq`. Puesto que la posición actual de la lista destino es siempre el último nodo insertado, `lizq` se construye como una copia exacta de `lder`. De una forma prácticamente idéntica, en la Figura 16.11 se usa un bucle para imprimir el contenido de la lista enlazada `laLista`.

El método de copia para las listas enlazadas se puede implementar usando dos iteradores.

```
1  // Método que copia lder en lizq;
2  // Solamente llama a métodos públicos
3  public static void copiar(ListaEnlazada lizq,ListaEnlazada lder )
4  {
5      if( lizq == lder ) // Comprobación de alias
6          return;
7
8      lizq.vaciar( );
9      ListaEnlazadaIter itrIzq = new ListaEnlazadaIter( lizq );
10     ListaEnlazadaIter itrDer = new ListaEnlazadaIter( lder );
11
12     try
13     {
14         for( ; itrDer.estaDentro( ); itrDer.avanzar( ) )
15             itrIzq.insertar( itrDer.recuperar( ) );
16     }
17     catch( ElementoNoEncontrado e ) { } // No puede ocurrir
18 }
```

Figura 16.10 Método de copia para las listas enlazadas, usando `ListaEnlazadaIter` para recorrerlas.

```
1  // Método simple de impresión
2  public static void imprimir( ListaEnlazada laLista )
3  {
4      if( laLista.esVacia( ) )
5          System.out.print( "Lista vacía" );
6      else
7      {
8          ListaEnlazadaIter itr = new ListaEnlazadaIter( laLista );
9          for( ; itr.estaDentro( ); itr.avanzar( ) )
10             System.out.print( itr.recuperar( ) + " " );
11     }
12     System.out.println( );
13 }
```

Figura 16.11 Método de impresión de `ListaEnlazada`.

De no ser por las comprobaciones de error muchas de las rutinas estarían formadas por una única línea de código.

En la línea 10 de la rutina `buscar` y en la parte correspondiente de la rutina `eliminar` se hace uso de la evaluación no estricta.

Ahora que hemos visto qué aspecto tiene `ListaEnlazadaIter` y cómo se usa, podemos implementar sus métodos. La Figura 16.12 muestra el cuerpo de `insertar`. Como hemos dicho anteriormente en el capítulo, si no hiciéramos las comprobaciones de errores la rutina estaría formada por una sola línea de código. La comprobación de errores añade una línea; además, para una mayor claridad, hemos decidido añadir una variable intermedia. Aun así se sigue tratando de un fragmento de código muy reducido.

En la Figura 16.13 se muestra la rutina `buscar`, en la que recorremos la lista elemento a elemento hasta que o bien llegamos al puntero `null` del final de la lista, o bien encontramos el elemento buscado. Observe cuidadosamente que el orden en que se llevan a cabo los tests de la línea 10 es importante porque de esa forma aprovechamos la evaluación no estricta. Como se estudió en la Sección 1.5.2, si la primera mitad del `&&` se evalúa a `false`, entonces el resultado es `false`, y no es necesario ejecutar la segunda mitad.

Análogamente, el método `eliminar` de la Figura 16.14 busca el elemento `x`. Al final del bucle, `itr` o bien apunta al nodo anterior al que contiene `x`, o bien al último nodo, si no se ha encontrado `x`. Entonces se lleva a cabo la eliminación usando el algoritmo descrito previamente. El resto de las rutinas se muestran en la Figura 16.15. Son todas ellas rutinas muy sencillas.

```
1  /**
2   * Inserta tras la posición actual.
3   * actual apunta al nodo insertado con éxito.
4   * @param x el elemento a insertar.
5   * @exception ElementoNoEncontrado si la posición actual es null.
6   */
7  public void insertar( Object x ) throws ElementoNoEncontrado
8  {
9      if( actual == null )
10         throw new ElementoNoEncontrado( "Error de inserción" );
11     NodoLista nuevoNodo = new NodoLista( x, actual.siguiente );
12     actual = actual.siguiente = nuevoNodo;
13 }
```

Figura 16.12 Rutina insertar para la clase `ListaEnlazadaIter`.

```
1  /**
2   * Coloca la posición actual en el primer nodo que contiene
3   * un elemento. Si no encontramos x, actual no se ve modificado.
4   * @param x el elemento a buscar.
5   * @return true si encontramos el elemento x; si no, false .
6   */
7  public boolean buscar( Object x )
8  {
9      NodoLista itr = laLista.cabecera.siguiente;
10     while( itr != null && !itr.dato.equals( x ) )
11         itr = itr.siguiente;
12
13     if( itr == null )
14         return false;
15
16     actual = itr;
17     return true;
18 }
```

Figura 16.13 Rutina `buscar` para `ListaEnlazadaIter`.

```
1  /**
2   * Elimina la primera aparición de un elemento.
3   * Si aparece en la lista se coloca actual al principio;
4   * pero si no aparece en la lista, no se modifica.
5   * @param x elemento a eliminar.
6   * @exception ElementoNoEncontrado si no se encuentra.
7   */
8  public void eliminar( Object x ) throws ElementoNoEncontrado
9  {
10     NodoLista itr = laLista.cabecera;
11     while( itr.siguiente != null &&
12            !itr.siguiente.dato.equals( x ) )
13         itr = itr.siguiente;
14
15     if( itr.siguiente == null )
16         throw new ElementoNoEncontrado( "Fallo en eliminación" );
17
18     itr.siguiente = itr.siguiente.siguiente; // Eliminación
19     actual = laLista.cabecera;               // Inicializa actual
20 }
```

Figura 16.14 Rutina `eliminar` para la clase `ListaEnlazadaIter`.

```
1  /**
2   * Devuelve el elemento almacenado en la posición actual.
3   * @return el elemento almacenado o null si la posición
4   *     actual no está en la lista.
5   */
6  public Object recuperar( )
7  {
8      return estaDentro( ) ? actual.dato : null;
9  }
10
11 /**
12  * Coloca la posición actual en el primer nodo.
13  * Esta operación es válida para listas vacías.
14  */
15 public void primero( )
16 {
17     actual = laLista.cabecera.siguiente;
18 }
19
20 /**
21  * Avanzar la posición actual al nodo siguiente de la lista.
22  * Si la posición actual es null, no hace nada.
23  * En esta rutina no se lanzan excepciones porque su uso más
24  * habitual (dentro de un bucle for) requeriría
25  * que el programador añadiera un bloque try/catch.
26  */
27 public void avanzar( )
28 {
29     if( actual != null )
30         actual = actual.siguiente;
31 }
32
33 /**
34  * Comprueba si la posición actual apunta a un elemento válido.
35  * @return true si la posición actual no es null y no está
36  *     apuntando al nodo cabecera.
37  */
38 public boolean estaDentro( )
39 {
40     return actual != null && actual != laLista.cabecera;
41 }
```

Figura 16.15 Rutinas de iteración para la clase `ListaEnlazadaIter`.

16.3 Listas doblemente enlazadas y listas enlazadas circulares

Las *listas doblemente enlazadas* permiten recorridos en ambos sentidos gracias al almacenamiento de dos referencias por nodo.

Como se mencionó en la Sección 16.2, las listas enlazadas simples no soportan de forma eficiente algunas operaciones importantes. Por ejemplo, aunque es fácil ir al principio de la lista, ir al final consume bastante tiempo. Aunque podemos avanzar fácilmente usando `avanzar`, la implementación de `retroceder` no se puede hacer de forma eficiente con una sola referencia `siguiente`. En algunas aplicaciones esto puede ser crítico. Por ejemplo, cuando diseñamos un editor de textos, podemos mantener la imagen interna del fichero como una lista enlazada de líneas. En este caso, desearíamos movernos fácilmente por el fichero tanto hacia abajo como hacia arriba, insertar antes y después de una línea, en lugar de solamente detrás, y poder saltar rápidamente a la última línea. Un momento de reflexión nos sugiere que para implementar estas operaciones de forma eficiente, deberíamos tener dos referencias en cada nodo: una al siguiente nodo de la lista y otro al anterior. Para terminar de hacer simétrica la representación, añadiríamos además un nodo final además del nodo cabecera. Estas listas reciben el nombre de *doblemente enlazadas*. La Figura 16.16 muestra la lista doblemente enlazada que contiene los elementos `a`, `b`. Cada nodo contiene ahora dos referencias (`siguiente` y `anterior`), con lo que se pueden realizar de forma sencilla búsquedas y recorridos en ambos sentidos. Obviamente, son necesarios algunos cambios pequeños pero importantes.

La simetría exige que usemos un nodo `cabecera` y un nodo `final`, y que, para sacar partido de ella, proporcionemos el doble de operaciones.

En primer lugar, una lista vacía consiste ahora en un nodo `cabecera` y uno `final` conectados entre sí, como se muestra en la Figura 16.17. Observe que no son necesarios en los algoritmos `cabecera.anterior` y `final.siguiente`, por lo que ni siquiera se inicializan. La comprobación de lista vacía es ahora

```
cabecera.siguiente == final
```

o bien

```
final.anterior == cabecera
```

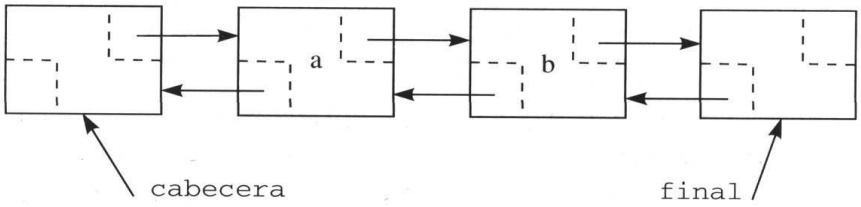

Figura 16.16 Lista doblemente enlazada.

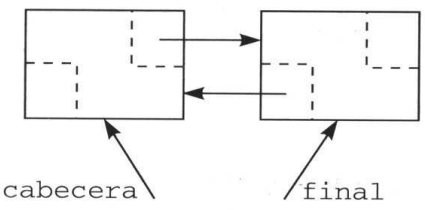

Figura 16.17 Lista doblemente enlazada vacía.

Ya no volvemos a usar `null` para decidir si un avance nos ha llevado hasta el final de la lista, sino que lo sabemos cuando llegamos a `cabecera` o a `final` (recuerde que podemos ir en cualquier sentido). El método `retroceder` se puede implementar mediante

Cuando avanzamos más allá del final de la lista, ahora llegamos al nodo `final` en lugar de a `null`.

```
actual = actual.anterior;
```

Antes de pasar a algunas de las operaciones adicionales que están disponibles, veamos cómo cambian las operaciones de inserción y eliminación. Naturalmente ahora podemos hacer `insertarAntes` e `insertarDespues`. En `insertarDespues` se hallan implicados el doble de cambios de punteros en las listas doblemente enlazadas que en las listas enlazadas simples. Si escribimos cada instrucción de forma explícita, obtenemos

La inserción y la eliminación implican el doble de cambios de punteros, en comparación con las listas enlazadas simples.

```
nNodo = new NodoListaEnlazadaDoble( );
nNodo.anterior = actual;                // Asigna valor a x.anterior
nNodo.siguiente = actual.siguiente; // Asigna valor a x.siguiente
nNodo.anterior.siguiente = nNodo;    // Asigna valor a a.siguiente
nNodo.siguiente.anterior = nNodo;    // Asigna valor a b.anterior
actual = nNodo;
```

Como se ha mostrado antes en el capítulo, los dos primeros cambios de referencias se podrían juntar en el constructor. En la Figura 16.18 se ilustran los cambios de `anterior` y `siguiente` (siguiendo el orden *1, 2, 3, 4*).

La Figura 16.18 también se puede usar para guiarnos en el algoritmo de eliminación. Al contrario que en las listas enlazadas simples, ahora podemos eliminar el nodo actual porque tenemos acceso inmediato al nodo anterior. Por tanto, para `eliminar x` en la Figura 16.18, tenemos que cambiar la referencia `siguiente` de `a` y la `anterior` de `b`. Los cambios básicos son

`eliminar` puede empezar a recorrer la lista a partir del nodo `actual` porque ahora podemos obtener el nodo anterior instantáneamente.

```
actual.anterior.siguiente = actual.siguiente;
                                        // Actualiza a.siguiente
actual.siguiente.anterior = actual.anterior;
                                        // Actualiza b.anterior
actual = cabecera;
```

Para realizar una implementación completa de una lista doblemente enlazada, necesitamos decidir qué operaciones se van a soportar. Es razonable esperar que haya el doble de operaciones que en las listas enlazadas simples. Cada procedi-

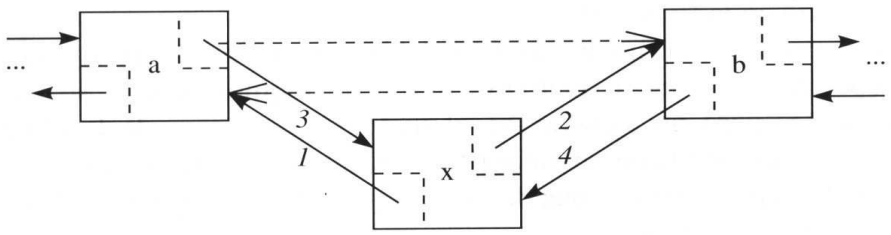

Figura 16.18 Inserción en una lista doblemente enlazada mediante la construcción de un nuevo nodo y la modificación posterior de las referencias, en el orden indicado.

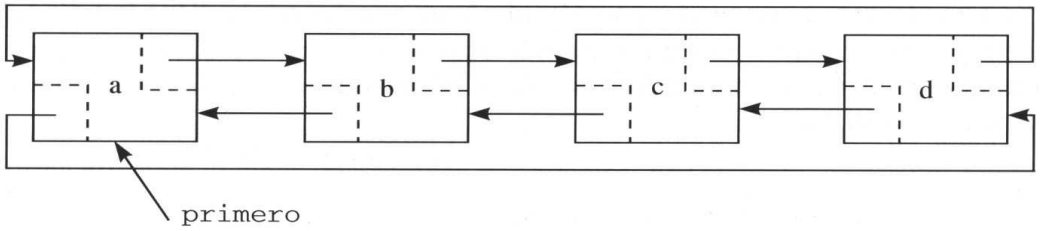

Figura 16.19 Lista circular doblemente enlazada.

miento individual será muy similar a las rutinas de las listas enlazadas simples, sólo las operaciones dinámicas implican cambios adicionales en las referencias. Para la mayoría de las rutinas, el código está dominado por las comprobaciones de error. Aunque algunas de ellas cambian, por ejemplo ya no comparamos con la referencia `null`, éstas no se hacen más complejas. Dejamos la implementación de la clase como práctica de programación en el Ejercicio 16.14.

En una *lista enlazada circular*, la referencia `siguiente` del último nodo apunta al primero. Esto es útil en relación a tareas circulares.

Una práctica habitual es crear una lista enlazada circular, en la que el último nodo apunta al primero. Esto se puede hacer con o sin cabecera, aunque normalmente se hace sin ella, puesto que el propósito principal de la cabecera es asegurar que cada nodo tiene uno anterior, lo cual ya es cierto para una lista enlazada circular no vacía. Sin cabecera solamente tendremos que tratar como un caso especial la lista vacía. Se mantiene una referencia al primer nodo, pero éste ya no es el nodo cabecera. Podemos usar simultáneamente listas enlazadas circulares y listas doblemente enlazadas, como se muestra en la Figura 16.19. La lista circular es útil en situaciones en las que queremos que las búsquedas permitan circularidad, como por ejemplo en algunos editores de texto en los que una vez que se llega al final del fichero, se sigue buscando desde el principio hasta el punto en el que nos encontrábamos. En el Ejercicio 16.16 se le pide implementar una lista circular doblemente enlazada.

16.4 Listas enlazadas ordenadas

Podemos mantener ordenados los elementos de una lista derivando una clase `ListaIterOrd` a partir de `ListaEnlazadaIter`.

A veces desearíamos guardar los elementos de forma ordenada en la lista enlazada. La diferencia fundamental entre una lista enlazada ordenada y una no ordenada es la rutina de inserción. De hecho, podemos obtener una clase de listas ordenadas limitándonos a modificar la rutina de inserción que ya tenemos escrita. El hecho de que la rutina `insertar` sea parte de la clase `ListaEnlazadaIter` nos lleva a pensar que deberíamos tener una nueva clase `ListaIterOrd` basada en ella, lo cual se muestra en la Figura 16.20.

El constructor, mostrado en las líneas 18 y 19, simplemente llama al constructor de la superclase. El nuevo método `insertar` requiere que `x` sea de la clase `Comparable`, pero no sobrescribe el método `insertar` de la clase `ListaEnlazadaIter` porque tiene un valor de la clase `Object` como parámetro. En las líneas 21 a 28 proporcionamos una definición que sí lo sobrescribe e intenta llamar al método `insertar` en orden.

Como ejemplo, la Figura 16.21 ilustra llamadas a varios métodos `insertar`. Los métodos `insertar` que tienen un manejador de excepciones son los que esperan un valor de la clase `Object` como parámetro. La llamada de la línea 7, a la

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4  import Soporte.*; import Soporte.Comparable;
5
6  // Clase ListaIterOrd; mantiene la "posición actual"
7  //
8  // CONSTRUCCIÓN: con una ListaEnlazada a la que la
9  //      ListaIterOrd está permanentemente ligada
10 // ******************OPERACIONES PÚBLICAS*******************
11 // void insertar( x )      --> Inserta x en orden
12 // El resto de operaciones idénticas a las de ListaEnlazadaIter
13 // ************************ERRORES*************************
14 // Excepciones lanzadas por acceso o eliminación ilegal.
15
16 public class ListaIterOrd extends ListaEnlazadaIter
17 {
18     public ListaIterOrd ( Lista unaLista )
19       { super( unaLista ); }
20
21     public void insertar( Object x ) throws ElementoNoEncontrado
22     {
23         if( x instanceof Comparable )
24             insertar( (Comparable) x );
25         else
26             throw new ElementoNoEncontrado( "ListaIterOrd " +
27             " insertar requiere un objeto de tipo Comparable" );
28     }
29
30         // Inserta de forma ordenada.
31     public void insertar( Comparable x )
32     {
33         ListaEnlazadaIter ant = new ListaEnlazadaIter( laLista );
34         ListaEnlazadaIter act = new ListaEnlazadaIter( laLista );
35
36         ant.cero( );
37         act.primero( );
38         while( act.estaDentro( ) &&
39              ( (Comparable)( act.recuperar( ))).menorQue( x ) )
40         {
41             act.avanzar( );
42             ant.avanzar( );
43         }
44
45         try
46           { ant.insertar( x ); }
47         catch( ElementoNoEncontrado e ) { } // No puede ocurrir
48         actual = ant.actual;
49     }
50 }
```

Figura 16.20 Clase iteradora para las listas enlazadas ordenadas en la que las inserciones deben respetar el orden.

que se le pasa como parámetro una referencia `Comparable` estáticamente tipada, es la única llamada directa al método que acepta un valor de la clase `Comparable`. La llamada de la línea 9 lanzará una excepción, ya que `obj` apunta a un valor de

```
1  public static void main( String [ ] args )
2  {
3      MiEntero m = new MiEntero( 5 );
4      Object obj = new Integer( 5 );
5      ListaIterOrd itr = new ListaIterOrd( new ListaEnlazada( ) );
6
7      itr.insertar( m ); // Llama a insertar( Comparable )
8
9      try { itr.insertar( obj ); }
10     catch( ElementoNoEncontrado e )
11       { System.out.println( "Elemento1 no Comparable" ); }
12
13     obj = m;
14     try { itr.insertar( obj ); }
15     catch( ElementoNoEncontrado e )
16       { System.out.println( "Elemento2 no Comparable" ); }
17 }
```

Figura 16.21 Programa que ilustra qué método `insertar` se está usando.

tipo `Integer`, el cual no implementa la interfaz `Comparable`. La llamada de la línea 14 invoca al método `insertar` que acepta un valor de la clase `Object`, ya que aunque en tiempo de ejecución es cierto que `obj` apunta a un objeto de la clase `Comparable`, es el tipo estático el usado para decidir a qué método se está llamando. Aun así, ese método `insertar` llamará al método `insertar` que procesa valores de la clase `Comparable` (línea 24 de la Figura 16.20). Ésta es la razón por la que sobrescribimos el método `equals` con un `Object` como parámetro.

El código del nuevo método `insertar`, mostrado en las líneas 31 a 49 de la Figura 16.20, usa dos objetos `ListaIterOrd` para recorrer la lista correspondiente hasta que se encuentra el punto correcto de inserción. En ese punto, podemos aplicar la rutina `insertar` de `ListaEnlazadaIter`. Para usar objetos de la clase `ListaIterOrd`, debemos saber a qué objeto de la clase `Lista` se refieren. Se trata de `laLista`, que es un atributo protegido de la clase `ListaEnlazadaIter` y por tanto es accesible.

¿Necesitamos realmente usar `ListaEnlazadaIter` en la nueva rutina `insertar`? La respuesta es no, puesto que tanto `actual` como `laLista` son accesibles. El Ejercicio 16.17 le pide rescribir el código para evitar usar `ListaEnlazadaIter`.

Una deficiencia de nuestro esquema es que se puede acceder a cualquier lista usando métodos tanto de `ListaEnlazadaIter` como de `ListaIterOrd`, por lo que no tenemos garantías de que los métodos de `ListaIterOrd` estén trabajando sobre una lista ordenada. Podemos arreglar la situación añadiendo un atributo extra a la clase `ListaEnlazada` que especifique si la lista está ordenada o no. Este atributo podría inicializarse en la primera llamada a `insertar`, y las siguientes llamadas verificarían que no estamos mezclando los dos tipos

Resumen

En este capítulo se han implementado las listas enlazadas usando una clase iteradora. Nuestra clase iteradora `ListaIter` trabaja con la clase `Lista`. Cualquier operación que dependa de alguna manera del posicionamiento en la lista se

define en la clase `ListaIter`. La clase `Lista` define solamente aquellas operaciones que ven a las listas como una entidad (por ejemplo, `esVacia`). La clase `ListaIter` mantiene una noción de posición actual que se actualiza al avanzar, buscar, insertar e incluso eliminar en la lista. El capítulo ha examinado también variaciones de las listas enlazadas, incluyendo las listas doblemente enlazadas, que permiten el recorrido de la lista en ambos sentidos. Finalmente, hemos visto que es relativamente fácil derivar una clase de listas enlazadas ordenadas a partir de la clase base de listas enlazadas.

Elementos del juego

clase iteradora Clase que mantiene la posición actual y contiene todas las rutinas que dependen del conocimiento de la posición de la lista en que nos encontramos.

lista doblemente enlazada Lista enlazada que permite recorridos en ambos sentidos mediante el almacenamiento de dos referencias por nodo.

lista enlazada circular Lista enlazada en la que la referencia siguiente del último nodo apunta al primero. Esto es útil en tareas que implican circularidad.

lista enlazada ordenada Lista enlazada en la que los elementos están ordenados. Se deriva una clase `ListaIterOrd` a partir de `ListaEnlazadaIter`.

nodo cabecera Nodo extra en la lista enlazada que no almacena ningún dato pero que sirve para satisfacer el requerimiento de que cada nodo tenga uno anterior. Este nodo nos permite evitar los casos especiales, como la inserción de un nuevo primer elemento y la eliminación del primer elemento de la lista.

Errores comunes

1. El error más común en las listas enlazadas consiste en conectar incorrectamente los nodos al llevar a cabo una inserción. La conexión es especialmente delicada cuando se trabaja con listas doblemente enlazadas.
2. Los métodos de una clase no deberían permitir desreferenciar un puntero `null`.
3. Se pueden producir problemas cuando hay varios iteradores accediendo a una misma lista simultáneamente. Por ejemplo, ¿qué sucedería si un iterador elimina un nodo al que otro iterador va a acceder? Resolver este tipo de problemas requiere un trabajo adicional.
4. Tenga cuidado al sobrescribir un método con otro cuyos parámetros sean un subtipo de los originales (como es el caso de `equals`).

En Internet

La clase de listas enlazadas, incluyendo las listas enlazadas ordenadas, está disponible en el directorio **DataStructures**. En el directorio **Chapter16** se proporciona un programa de prueba que también incluye los métodos de la Figura 16.6 y de la Figura 16.11, y el programa de la Figura 16.21.

LinkedList.java	Contiene la implementación mediante listas enlazadas de las listas (traducida por `ListaEnlazada`).
LinkedListItr.java	Contiene la implementación mediante listas enlazadas de un iterador para las listas (traducido por `ListaEnlazadaIter`).
SortListItr.java	Contiene la implementación de un iterador para listas ordenadas (traducido por `ListaIterOrd`).
TestList.java	Contiene un programa de prueba para las listas.
WhichInsert.java	Contiene la rutina de la Figura 16.21.

Ejercicios

Cuestiones breves

16.1. Dibuje una lista enlazada vacía con nodo cabecera.

16.2. Dibuje una lista doblemente enlazada vacía con nodo cabecera y final.

Problemas teóricos

16.3. Escriba un algoritmo para imprimir de forma inversa una lista enlazada simple, usando solamente un espacio extra constante, lo cual implica que no puede usar recursión.

16.4. Discuta sobre si sería mejor convertir `primero` en ilegal sobre las listas vacías.

Problemas prácticos

16.5. Modifique la rutina `buscar` de la clase `ListaEnlazadaIter` para devolver la última aparición del elemento `x`.

16.6. Mirar el contenido del siguiente objeto en una `ListaEnlazadaIter` requiere la aplicación de `avanzar`. En algunos casos, puede ser preferible mirar el elemento siguiente de la lista, sin avanzar. Escriba el método con la siguiente declaración que realiza esto en el caso general. El método `vistazo` devuelve una `ListaEnlazadaIter` que se corresponde con las `k` posiciones siguientes a `actual`.

```
ListaEnlazadaIter vistazo( int k );
```

16.7. Explique lo que le pasará al método añadido en el Ejercicio 16.6 cuando se derive la clase `ListaIterOrd` a partir de `ListaEnlazadaIter`.

16.8. Modifique el método `eliminar` de la clase `ListaEnlazadaIter` de forma que se eliminen todas las apariciones de `x`.

16.9. Añada la rutina `eliminarSig` a la clase `ListaEnlazadaIter`. Él mismo eliminará el elemento situado tras la posición actual. ¿Cómo se tratan los posibles errores?

16.10. Implemente una clase `Pila` eficiente usando una `ListaEnlazada` como atributo. Necesitará un iterador `ListaEnlazadaIter`, que puede ser o bien un atributo o una variable local en aquellas rutinas que lo necesiten.

16.11. Implemente una clase `Cola` eficiente usando (como en el Ejercicio 16.10):

a) Una lista enlazada e iteradores apropiados. ¿Cuántos de estos iteradores deben ser atributos para conseguir una implementación eficiente?
b) Una lista doblemente enlazada e iteradores apropiados.

16.12. Implemente `retroceder` para las listas enlazadas simples. Observe que tardará un tiempo lineal.

16.13. Implemente las listas enlazadas sin nodo cabecera.

Prácticas de programación

16.14. Implemente una clase de listas doblemente enlazadas, incluyendo:

a) Métodos `irAPrincipio` e `irAFinal`.
b) Métodos `primero` y `ultimo`.
c) Métodos `insertarAntes` e `insertarDespues`.
d) Métodos `buscarPrimero` y `buscarUltimo` (la búsqueda comienza desde el principio y el final, respectivamente).
e) Métodos `eliminarPrimero` y `eliminarUltimo`

16.15. Escriba un editor de líneas. La sintaxis de los comandos es similar al editor de líneas *ed* de Unix. Se mantiene una copia interna del fichero como una lista enlazada de líneas. Para poder moverse hacia arriba y hacia abajo en el fichero, necesitará una lista doblemente enlazada. La mayor parte de los comandos, los cuales se muestran en la Figura 16.22, están representados mediante una cadena de un solo carácter, aunque algunos están formados por dos caracteres y requieren un argumento (o más).

16.16. Implemente una lista circular doblemente enlazada.

16.17. Diseñe de nuevo `ListaEnlazada`, `ListaEnlazadaIter` y `ListaIterOrd` para no permitir el intercalamiento de comandos `insertar` procedentes de `ListaEnlazadaIter` y `ListaIterOrd`.

16.18. Si el orden en que se almacenan los elementos en una lista no es importante, entonces podemos acelerar las búsquedas usando la siguiente heurística, conocida como *mover-al-frente*: siempre que acceda a un elemento, muévalo al principio de la lista. La razón por la que esto constituye una mejora, es que los elementos a los que se accede frecuentemente tienden a migrar hacia el principio de la lista, mientras que los menos frecuentemente accedidos migran hacia el final de la lista. En consecuencia, los elementos más frecuentemente accedidos requieren una búsqueda más breve. Implemente la heurística mover-al-frente para las listas enlazadas.

16.19. Escriba rutinas `union` e `interseccion` que devuelvan la unión e intersección de dos listas enlazadas. Asuma que las listas de entrada están ordenadas.

Comando	Función
`1`	Ir al principio.
`a`	Añadir texto tras la línea actual hasta encontrar un punto(.).
`d`	Borrar línea actual.
`dr num num`	Borrar varias líneas.
`f name`	Cambiar el nombre del fichero actual (para la siguiente escritura).
`g num`	Ir a una línea.
`h`	Obtener ayuda.
`i`	Como concatenar, pero añade las líneas antes de la línea actual.
`m num`	Mover la línea actual después de otra línea.
`mr num num num`	Mover varias líneas como una unidad después de otra línea.
`n`	Activar y desactivar la numeración de líneas.
`p`	Imprimir la línea actual.
`pr num num`	Imprimir varias líneas.
`q!`	Salir sin guardar.
`r name`	Leer y pegar otro fichero en el actual.
`s text text`	Sustituir un texto por otro.
`t num`	Copiar la línea actual después de otra línea.
`tr num num num`	Copiar varias líneas después de otra línea.
`w`	Guardar fichero en disco.
`x!`	Salir guardando.
`$`	Ir a la última línea.
`-`	Subir una línea.
`+`	Bajar una línea.
`=`	Imprimir el número actual de línea.
`/ text`	Buscar hacia delante un patrón.
`? text`	Buscar hacia atrás un patrón.
`#`	Imprimir el número de líneas y caracteres del fichero.

Figura 16.22 Comandos para el editor del Ejercicio 16.15.