6.5 La interfaz List

Una *lista* es una colección de elementos en la que los elementos tienen una posición.

La interfaz `List` amplía la interfaz `Collection` y abstrae la noción de posición.

Una *lista* es una colección de elementos en la que los elementos tienen una posición. El ejemplo más obvio de lista es una matriz. En una matriz, los elementos se colocan en las posiciones 0, 1, etc.

La interfaz `List` amplia la interfaz `Collection` y abstrae la noción de posición. La interfaz en `java.util` añade numerosos métodos a la interfaz `Collection`. Nosotros nos conformaremos con añadir los tres mostrados en la Figura 6.16.

Los primeros dos métodos son `get` y `set`, que son similares a los métodos que ya hemos visto en `ArrayList`. El tercer método devuelve un iterador más flexible, el `ListIterator`.

6.5.1 La interfaz ListIterator

Como se muestra en la Figura 6.17, `ListIterator` es simplemente como un `Iterator`, salvo porque es bidireccional. Por tanto, podemos tanto avanzar como retroceder. Debido a esto, hay que proporcionar al método factoría `listIterator` que lo crea un valor que sea lógicamente igual al

```
1  package weiss.util;
2
3  /**
4   * Interfaz List. Contiene mucho menos que la de java.util
5   */
6  public interface List<AnyType> extends Collection<AnyType>
7  {
8     AnyType get( int idx );
9     AnyType set( int idx, AnyType newVal );
10
11    /**
12     * Obtiene un objeto ListIterator que se usa para recorrer
13     * la colección bidireccionalmente.
14     * @return un iterador posicionado
15     * antes del elemento solicitado.
16     * @param pos el índice para iniciar el iterador.
17     * Utilice size() para realizar un recorrido inverso completo.
18     * Utilice 0 para realizar un recorrido completo en dirección normal.
19     * @throws IndexOutOfBoundsException si pos no está
20     * entre 0 y size(), ambos incluidos.
21     */
22    ListIterator<AnyType> listIterator( int pos );
23 }
```

Figura 6.16 Una interfaz `List` de ejemplo.

número de elementos que ya hayan sido visitados en la dirección normal de avance. Si este valor es cero, el `ListIterator` se inicializa en la primera posición, igual que un `Iterator`. Si este valor tiene el tamaño del objeto `List`, el iterador se inicializa como si ya hubiera procesado todos los elementos en la dirección normal de avance. Por tanto, en este estado, `hasNext` devuelve `false`, pero podemos utilizar `hasPrevious` y `previous` para recorrer la lista en sentido inverso.

`ListIterator` es una versión bidireccional de `Iterator`.

La Figura 6.18 ilustra que podemos utilizar `itr1` para recorrer una lista en la dirección normal de avance y, después de alcanzar el final podemos volver a recorrer la lista hacia atrás. También ilustra `itr2`, que se coloca al final y que simplemente procesa la lista `ArrayList` en sentido inverso. Finalmente, se ilustra el bucle `for` avanzado.

Una dificultad con `ListIterator` es que la semántica de `remove` debe ser modificada ligeramente. La nueva semántica es que `remove` elimina del objeto `List` el último objeto devuelto como resultado de la invocación de `next` o de `previous`, pudiéndose invocar `remove` una única vez entre

```
1 package weiss.util;
2
3 /**
4  * Interfaz ListIterator para la interfaz List.
5  */
6 public interface ListIterator<AnyType> extends Iterator<AnyType>
7 {
8    /**
9     * Comprueba si hay más elementos en la colección al realizar
10    * la iteración en sentido inverso.
11    * @return true si hay más elementos en la colección al
12    * recorrerla en sentido inverso.
13    */
14   boolean hasPrevious( );
15
16   /**
17    * Obtiene el elemento anterior de la colección.
18    * @return el elemento anterior (todavía no visualizado) de la colección
19    * al recorrerla en sentido inverso.
20    */
21    AnyType previous( );
22
23   /**
24    * Elimina el último elemento devuelto por next o previous.
25    * Solo puede invocarse una única vez después de next o previous.
61    */
27   void remove( );
28 }
```

Figura 6.17 Interfaz `ListIterator` de ejemplo.

llamadas sucesivas a `next` o `previous`. Para sustituir la salida *javadoc* generada para `remove`, el método `remove` se incluye en la interfaz `ListIterator`.

La interfaz de la Figura 6.17 es solo una interfaz parcial. Hay algunos otros métodos adicionales en `ListIterator` que no vamos a explicar en este libro, pero que se emplean como ejercicios. Entre estos métodos se incluyen `add` y `set`, que permiten al usuario realizar cambios en la lista `List` en la posición actualmente ocupada por el iterador.

```
1  import java.util.ArrayList;
2  import java.util.ListIterator;
3
4  class TestArrayList
5  {
6     public static void main( String [ ] args )
7     {
8        ArrayList<Integer> lst = new ArrayList<Integer>( );
9        lst.add( 2 ); lst.add( 4 );
10       ListIterator<Integer> itr1 = lst.listIterator( 0 );
11       ListIterator<Integer> itr2 = lst.listIterator( lst.size( ) );
12
13       System.out.print( "Forward: " );
14       while( itr1.hasNext( ) )
15          System.out.print( itr1.next( ) + " " );
16       System.out.println( );
17
18       System.out.print( "Backward: " );
19       while( itr1.hasPrevious( ) )
20          System.out.print( itr1.previous( ) + " " );
21       System.out.println( );
22
23       System.out.print( "Backward: " );
24       while( itr2.hasPrevious( ) )
25          System.out.print( itr2.previous( ) + " " );
26       System.out.println( );
27
28       System.out.print( "Forward: ");
29       for( Integer x : lst )
30          System.out.print( x + " " );
31       System.out.println( );
32    }
33 }
```

Figura 6.18 Un programa de ejemplo que ilustra la iteración bidireccional.

6.5.2 La clase LinkedList

La clase LinkedList implementa una lista enlazada.

Hay dos implementaciones básicas de `List` en la API de Colecciones. Una implementación es `ArrayList`, que ya hemos visto anteriormente. La otra es la lista enlazada `LinkedList`, que almacena internamente los elementos de una forma distinta de la que emplea `ArrayList`, lo que conduce a una serie de compromisos en lo que se refiere al rendimiento. Una tercera versión es `Vector`, que es como `ArrayList`, pero proviene de una librería más antigua y que está presente fundamentalmente por compatibilidad con el código *heredado* (antiguo). La utilización de `Vector` ya no es común.

`ArrayList` puede resultar apropiada si las inserciones se efectúan solo al final de la matriz (utilizando `add`), por las razones explicadas en la Sección 2.4.3. `ArrayList` duplica la capacidad interna de la matriz si una inserción al final de la misma hace que se exceda la capacidad interna. Aunque esto nos da un buen rendimiento O mayúscula, especialmente si añadimos un constructor que permita al llamante sugerir la capacidad de la matriz interna, `ArrayList` es una elección poco adecuada si las inserciones no se hacen al final, porque entonces nos veremos obligados a desplazar elementos para hacer sitio a los elementos nuevos.

La lista enlazada se utiliza para evitar tener que mover grandes cantidades de datos. Almacena los elementos con un gasto adicional de una referencia por cada elemento.

En una lista enlazada, los elementos se almacenan de forma no contigua, en lugar de emplearse la matriz contigua usual. Para hacer esto, almacenamos cada objeto en un nodo que contiene el objeto y una referencia al siguiente nodo de la lista, como se muestra en la Figura 6.19. En este escenario, lo que hacemos es mantener referencias al primer y último nodo de la lista.

Para ser concretos, un nodo típico tendría el aspecto siguiente:

```
class ListNode
{
   Object data;   // Algún elemento
   ListNode next;
}
```

En cualquier punto, podemos añadir un nuevo elemento `x` al final haciendo lo siguiente:

```
last.next = new ListNode( ); // Añadir un nuevo ListNode
last = last.next;            // Ajustar la referencia al último nodo
last.data = x;               // Colocar x en el nodo
last.next = null;            // Es el último; ajustar la ref. al siguiente.
```

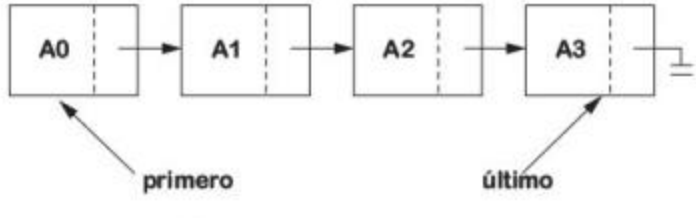

Figura 6.19 Una lista enlazada simple.

Ahora ya no podemos encontrar un único elemento arbitrario con un único acceso. En lugar de ello, tendremos que recorrer la lista. Esto es similar a la diferencia entre acceder a un elemento que se encuentra en un disco compacto (un acceso) o en una cinta (un acceso secuencial). Aunque parezca que esto hace que las listas enlazadas sean menos atractivas que las matrices, siguen teniendo ventajas. En primer lugar, una inserción en mitad de la lista no exige mover todos los elementos situados después del punto de inserción. Los movimientos de datos son muy caros en la práctica, y la lista enlazada permite realizar una operación de inserción con solo un número constante de instrucciones de asignación.

El compromiso básico entre `ArrayList` y `LinkedList` es que `get` no es eficiente para `LinkedList`, mientras que la inserción y eliminación en mitad de un contenedor están soportadas de manera más eficiente por `LinkedList`.

Comparando `ArrayList` y `LinkedList`, vemos que las inserciones y borrados en la mitad de la secuencia son ineficientes en `ArrayList` pero pueden ser eficientes en una lista `LinkedList`. Sin embargo, un `ArrayList` permite un acceso directo utilizando un índice, mientras que una `LinkedList` no debería permitirlo. En realidad, lo cierto es que en la API de Colecciones, `get` y `set` forman parte de la interfaz `List`, por lo que `LinkedList` soporta esta operaciones, aunque lo hace de una forma muy lenta. Por tanto, `LinkedList` puede utilizarse siempre, a menos que haga falta una indexación eficiente. `ArrayList` puede seguir siendo una mejor elección si las inserciones solo pueden producirse al final de la lista.

El acceso a la lista se realiza a través de una clase iteradora.

Para acceder a los elementos de la lista, necesitamos una referencia al nodo correspondiente en lugar de un índice. La referencia al nodo estará normalmente oculta dentro de una clase iteradora.

Puesto que `LinkedList` realiza las operaciones de adición y borrado de forma más eficiente, tiene más operaciones que `ArrayList`. Algunas de las operaciones adicionales disponibles para `LinkedList` son las siguientes:

```
void addLast( AnyType element )
```
Añade `element` al final de esta `LinkedList`.

```
void addFirst( AnyType element )
```
Añade `element` al principio de esta `LinkedList`.

```
AnyType getFirst( )
AnyType element( )
```
Devuelve el primer elemento de esta `LinkedList`. `element` se añadió en Java 5.

```
AnyType getLast( )
```
Devuelve el último elemento de esta `LinkedList`.

```
AnyType removeFirst( )
AnyType remove( )
```
Elimina y devuelve el primer elemento de esta `LinkedList`. `remove` se añadió en Java 5.

```
AnyType removeLast( )
```
Elimina y devuelve el último elemento de esta `LinkedList`.

Implementaremos la clase `LinkedList` en la Parte Cuatro.

6.5.3 Tiempo de ejecución para las distintas listas

En la Sección 6.5.2 vimos que, para algunas operaciones, `ArrayList` representa una mejor elección que `LinkedList`, mientras que para otras operaciones sucede justo lo contrario. En esta sección, en lugar de analizar los tiempos de ejecución de manera informal, vamos a analizarlos en términos de O mayúscula. Inicialmente, nos concentraremos en el siguiente subconjunto de operaciones:

- `add` (al final)
- `add` (al principio)
- `remove` (al final)
- `remove` (al principio)
- `get` y `set`
- `contains`

Costes de `ArrayList`

Para `ArrayList`, la adición al final significa simplemente colocar un elemento en la siguiente posición de la matriz, e incrementar el tamaño actual. Ocasionalmente, tendremos que redimensionar la capacidad de la matriz, pero como esta es una operación extremadamente rara, se puede argumentar con cierta razón que no afecta al tiempo de ejecución. Por tanto, el coste de añadir al final de una lista `ArrayList` no depende del número de elementos almacenados en la lista y es, por tanto, $O(1)$.

De forma similar, eliminar del final del `ArrayList` implica simplemente reducir el tamaño actual y es también $O(1)$, `get` y `set` en `ArrayList` se convierten en operaciones de indexación de la matriz, que normalmente se asume que requieren un tiempo constante y son, por tanto, operaciones $O(1)$.

No hace falta decir que, cuando hablamos del coste de una única operación sobre una colección, resulta difícil concebir algo que sea mejor que $O(1)$ (es decir, un tiempo constante) por cada operación. Para poder obtener un rendimiento mejor, haría falta que las operaciones fueran cada vez más rápidas a medida que la colección aumentara de tamaño, lo que sería bastante extraño.

Sin embargo, no todas las operaciones son $O(1)$ en un `ArrayList`. Como hemos visto, si añadimos al principio del `ArrayList`, entonces cada elemento de la lista debe ser desplazado una posición de índice hacia arriba. Por tanto, si hubiera N elementos en el `ArrayList`, la adición de un elemento al principio sería una operación $O(N)$. De forma similar, eliminar un elemento del principio del `ArrayList` requiere desplazar todos los elementos una posición de índice hacia abajo, lo que también es una operación $O(N)$. Y una comprobación `contains` sobre un `ArrayList` es una operación $O(N)$, porque potencialmente tendremos que examinar cada elemento del `ArrayList`.

No hace falta decir que $O(N)$ por cada operación no es tan conveniente como $O(1)$ por cada operación. De hecho, cuando consideramos que la operación `contains` es $O(N)$ que consiste básicamente en una búsqueda exhaustiva, podríamos argumentar que $O(N)$ por cada operación para una operación básica con una colección parece el peor resultado posible.

Costes de `LinkedList`

Si examinamos las operaciones de `LinkedList`, podemos ver que añadir un elemento al principio o al final es una operación $O(1)$. Para añadir al principio, simplemente creamos un nuevo nodo y lo añadimos al principio actualizando `first`. Esta operación no depende de conocer cuántos nodos

posteriores contiene la lista. Para añadir un elemento al final, simplemente creamos un nuevo nodo y lo añadimos al final, ajustando `last`.

Eliminar el primer elemento de la lista enlazada, es una operación $O(1)$, porque nos limitamos a hacer avanzar `first` al siguiente nodo de la lista. Eliminar el último elemento de la lista enlazada parece ser también $O(1)$, ya que necesitamos mover `last` al último nodo y actualizar en este nodo el enlace que apunta al nodo siguiente. Sin embargo, llegar hasta el penúltimo nodo no es tan fácil en una lista enlazada, como se deduce de la Figura 6.19.

En una lista enlazada clásica, en la que cada nodo almacena un enlace al nodo siguiente, el disponer de un enlace al último nodo no nos proporciona ninguna información acerca del penúltimo nodo. La idea obvia de mantener un enlace al penúltimo nodo no funciona, porque ese tercer enlace también tendría que ser actualizado durante una operación de borrado. En lugar de ello, lo que haremos será obligar a cada nodo a mantener un enlace al nodo anterior de la lista. Esto se muestra en la Figura 6.20 y se conoce con el nombre de *lista doblemente enlazada*.

En una lista doblemente enlazada, las operaciones `add` y `remove` en cualquiera de los dos extremos requieren un tiempo $O(1)$. Como sabemos, existe un compromiso, sin embargo, porque `get` y `set` no son eficientes con este tipo de lista. En lugar de acceder directamente a través de una matriz tenemos que seguir una serie de enlaces. En algunos casos, podremos optimizar ese recorrido comenzando por el final en lugar de por el principio, pero si la operación `get` o `set` se refiere a un elemento que está situado cerca de la parte central de la lista, necesitará un tiempo $O(N)$.

`contains` en una lista enlazada es igual que en `ArrayList`: el algoritmo básico es una búsqueda secuencial que termina examinando potencialmente cada uno de los elementos, y por tanto se trata de una operación $O(N)$.

Comparación de los costes de `ArrayList` y `LinkedList`

La Figura 6.21 compara los tiempos de ejecución de las operaciones simples en `ArrayList` y `LinkedList`.

Para ver la diferencia entre la utilización de `ArrayList` y `LinkedList` en una rutina de mayor tamaño, vamos a echar un vistazo a algunos métodos que operan sobre un objeto `List`. En primer lugar, suponga que construimos un objeto `List` añadiendo elementos al final del mismo.

```
public static void makeList1( List<Integer> lst, int N )
{
    lst.clear( );
    for( int i = 0; i < N; i++ )
        lst.add( i );
}
```

Independientemente de si se pasa como parámetro un `ArrayList` o una `LinkedList`, el tiempo de ejecución de `makeList1` es $O(N)$, porque cada llamada a `add`, estando situados al final de la

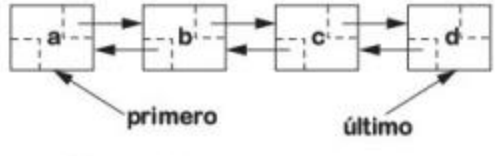

Figura 6.20 Una lista doblemente enlazada.

	ArrayList	LinkedList
add/remove al final	$O(1)$	$O(1)$
add/remove al principio	$O(N)$	$O(1)$
get/set	$O(1)$	$O(N)$
contains	$O(N)$	$O(N)$

Figura 6.21 Costes de cada una de las operaciones sencillas para `ArrayList` y `LinkedList`.

lista, requiere un tiempo constante. Por otro lado, si construimos una `List` añadiendo elementos al principio,

```
public static void makeList2( List<Integer> lst, int N )
{
   lst.clear( );
   for( int i = 0; i < N; i++ )
        lst.add( 0, i );
}
```

el tiempo de ejecución será $O(N)$ para una `LinkedList`, pero $O(N^2)$ para un `ArrayList`, porque en un `ArrayList`, la operación de adición de elementos al principio de la lista es una operación $O(N)$.

La siguiente rutina trata de calcular la suma de los números de una lista:

```
public static int sum( List<Integer> lst )
{
   int total = 0;
   for( int i = 0; i < N; i++ )
        total += lst.get( i );
}
```

Aquí, el tiempo de ejecución es $O(N)$ para un `ArrayList`, pero $O(N^2)$ para una `LinkedList`, ya que en una `LinkedList`, las llamadas a `get` son operaciones $O(N)$. En lugar de ello, utilice un bucle `for` avanzado, que hará que el tiempo de ejecución sea $O(N)$ para cualquier `List`, porque el iterador podrá avanzar de manera eficiente de un elemento al siguiente.

6.5.4 Eliminación y adición de elementos en mitad de una colección `List`

La interfaz `List` contiene dos operaciones:

```
void add( int idx, AnyType x );
void remove( int idx );
```

que permiten la adición de un elemento en un índice especificado y la eliminación de un elemento en un índice especificado. Para un `ArrayList`, estas operaciones son en general $O(N)$, debido al desplazamiento de elementos requerido.

Para una `LinkedList`, en principio cabe esperar que, si sabemos dónde se está realizando el cambio, entonces deberíamos poder llevarlo a cabo de forma eficiente dividiendo los enlaces de la lista enlazada. Por ejemplo, es fácil ver que, en principio, eliminar un único nodo en una lista doblemente enlazada requiere modificar algunos enlaces en los nodos posterior y anterior. Sin embargo, estas operaciones siguen siendo $O(N)$ en una `LinkedList` porque hace falta un tiempo $O(N)$ para encontrar el nodo.

Esta es precisamente la razón por la que `Iterator` proporciona un método `remove`. La idea es que, a menudo, un elemento solo se elimina después de haberlo examinado y de haber decidido que podemos descartarlo. Esto es similar a la idea de recoger elementos del suelo: a medida que buscamos por el suelo, si vemos un elemento, lo tomamos inmediatamente, porque ya estamos ahí.

Como ejemplo, vamos a proporcionar una rutina que elimine todos los elementos con valor par de una lista. Por tanto, si la lista contiene 6, 5, 1, 4, 2, entonces después de invocar el método contendrá 5, 1.

Hay varias posibles ideas para un algoritmo que elimine elementos de la lista a medida que los va encontrado. Por supuesto, una idea consiste en construir una nueva lista que contenga todos los números impares y luego borrar la lista original y copiar de nuevo en ella esos número impares. Pero nos interesa más escribir una versión limpia que evite hacer una copia y que lo que haga sea borrar los elementos de la lista a medida que los va encontrando.

Esta estrategia resulta, casi con seguridad, incorrecta para un `ArrayList`, porque el eliminar un elemento de un lugar arbitrario en un `ArrayList` es caro. (Es posible diseñar un algoritmo diferente para `ArrayList` que funcione mejor, pero ahora no nos vamos a preocupar por esto.) En una `LinkedList`, podemos tener algo más de esperanza, ya que, como sabemos, eliminar un elemento de una posición conocida puede hacerse de forma eficiente reordenando algunos enlaces.

La Figura 6.22 muestra el primer intento. En un `ArrayList`, como era de esperar, la operación `remove` no es eficiente, por lo que la rutina requiere un tiempo cuadrático. Una `LinkedList` manifiesta dos problemas. En primer lugar, la llamada a `get` no es eficiente, por lo que la rutina requiere un tiempo cuadrático. Además, la llamada a `remove` es igualmente ineficiente, porque como ya hemos visto, resulta muy caro llegar hasta la posición `i`.

La Figura 6.23 muestra un intento de rectificar el problema. En lugar de utilizar `get`, empleamos un iterador para recorrer la lista. Esto es bastante eficiente, pero cuando empleamos el método `remove` de `Collection` para eliminar un valor par, la operación no es eficiente, porque el método `remove` tiene que buscar de nuevo el elemento, lo que requiere un tiempo lineal. Pero si ejecutamos

```
1 public static void removeEvensVer1( List<Integer> lst )
2 {
3    int i = 0;
4    while( i < lst.size( ) )
5       if( lst.get( i ) % 2 == 0 )
6          lst.remove( i );
7       else
8          i++;
9 }
```

Figura 6.22 Elimina los números pares de una lista: cuadrático para todos los tipos de listas.

```
1 public static void removeEvensVer2( List<Integer> lst )
2 {
3    for( Integer x : lst )
4       if( x % 2 == 0 )
5          lst.remove( x );
6 }
```

Figura 6.23 Elimina los números pares de una lista; no funciona debido a `ConcurrentModificationException`.

el código, vemos que la situación es todavía peor: el programa genera una excepción `ConcurrentModificationException` porque, al eliminar un elemento, el iterador subyacente empleado por el bucle `for` avanzado queda invalidado. (El código de la Figura 6.22 explica por qué: no podemos esperar que el bucle `for` avanzado entienda que solo debe avanzar si no se está eliminando un elemento.)

La Figura 6.24 muestra una idea que sí funciona: después de que el iterador encuentre un elemento con valor par, podemos emplear el iterador para eliminar el valor que acaba de encontrar. Para una `LinkedList`, la llamada al método `remove` del iterador solo requiere un tiempo constante, porque el iterador se encuentra en el nodo que hay que eliminar (o cerca de él). Por tanto, para una `LinkedList`, la rutina completa requiere un tiempo lineal, en lugar de un tiempo cuadrático.

Para un `ArrayList`, incluso aunque el iterador se encuentre en el punto que hay que eliminar, la operación `remove` sigue siendo cara, porque habrá que desplazar los elementos de la matriz; así que, como cabía esperar, toda la rutina sigue requiriendo un tiempo cuadrático para un `ArrayList`.

Si ejecutamos el código de la Figura 6.24, pasándole un `LinkedList<Integer>`, necesita 0,015 segundos para una `LinkedList` de 400.000 elementos y 0,031 segundos para una `LinkedList` de 800.000 elementos, por lo que se trata claramente de una rutina con tiempo lineal, dado que el tiempo de ejecución se multiplica por el mismo factor que el tamaño de la entrada. Cuando pasamos un `ArrayList<Integer>`, la rutina tarda aproximadamente 1,25 minutos para un `ArrayList` de 400.000 elementos y cerca de cinco minutos para un `ArrayList` de 800.000 elementos; la multiplicación por cuatro del tiempo de ejecución cuando la entrada se multiplica solo por un factor de dos es coherente con el comportamiento cuadrático.

Para la operación de adición de elementos se produce una situación similar. La interfaz `Iterator` no proporciona un método `add`, pero `ListIterator` sí que lo hace. No hemos mostrado dicho método en la Figura 6.17; en el Ejercicio 6.24 le pediremos que lo utilice.

```
1 public static void removeEvensVer3( List<Integer> lst )
2 {
3    Iterator<Integer> itr = lst.iterator( );
4
5    while( itr.hasNext( ) )
6    if( itr.next( ) % 2 == 0 )
7    itr.remove( );
8 }
```

Figura 6.24 Elimina los números pares de una lista; cuadrático en `ArrayList`, pero requiere solo un tiempo lineal para `LinkedList`.