Parte IV

Implementaciones

CAPÍTULO

15 Pilas y colas

Este capítulo discute las implementaciones de las estructuras de datos de pilas y colas. Recuerde del Capítulo 6, que sus operaciones básicas deberían realizarse en tiempo constante. Para las pilas y las colas existen dos técnicas básicas de conseguir operaciones con coste constante. La primera consiste en almacenar los elementos de forma contigua empleando un vector, mientras que la segunda consiste en almacenarlos de forma no contigua en una lista enlazada. En este capítulo se presentan las implementaciones de las dos estructuras de datos empleando ambas técnicas. El código implementa las interfaces presentadas en el Capítulo 6.

En este capítulo veremos:

- Una implementación de las pilas basada en un vector.
- Una implementación de las colas basada en un vector.
- Una implementación de las pilas empleando listas enlazadas.
- Una implementación de las colas empleando listas enlazadas.
- La utilización de la herencia para obtener una nueva estructura de datos, llamada *cola doble*.

15.1 Implementación dinámica de vectores

En esta sección implementaremos las pilas y colas empleando un vector. Los algoritmos resultantes son extremadamente eficientes y sencillos de codificar.

15.1.1 Pilas

Una pila puede implementarse mediante un vector y un entero, que indica el índice del elemento situado en la cima de la pila.

Como muestra la Figura 15.1, una pila puede implementarse usando un vector y un entero. El entero `cdp` (*cima de pila*) es el índice del vector correspondiente al elemento situado en la cima de la pila. Así, cuando `cdp` es -1, la pila está vacía. Para `apilar`, se incrementa `cdp` y se inserta el nuevo elemento en la posición `cdp` del vector. El acceso a la cima de la pila resulta trivial, y la operación de `desapilar` se realiza decrementando `cdp`. En la Figura 15.1 mostramos la pila resultante tras las tres operaciones siguientes: `apilar(a)`, `apilar(b)` y `desapilar`, partiendo de una pila vacía.

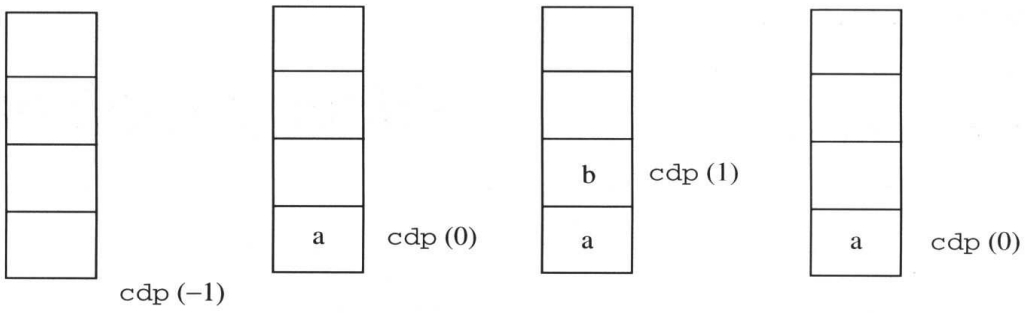

Figura 15.1 Comportamiento de las rutinas de las pilas: pila vacía; `apilar(a)`; `apilar(b)`; `desapilar`.

La Figura 15.2 muestra el esqueleto de la clase `PilaVec` basada en un vector. En ella se especifican dos atributos. `elVector`, que se expande siempre que es necesario, almacena los elementos de la pila. `cimaDePila` es el índice de la cima actual de la pila. En una pila vacía dicho índice es -1. El constructor correspondiente se muestra en la Figura 15.3.

La mayoría de las rutinas son aplicaciones de ideas vistas previamente.

Los métodos públicos se enumeran en las líneas 27 a 38 del esqueleto. La implementación de muchas de estas rutinas es simple. El método `apilar` se muestra en la Figura 15.4. Cuando no es necesario doblar el tamaño del vector, lo que es un ejercicio estándar ya estudiado, su código se reduce al que aparece en la línea 9. Recuérdese, de la Sección 1.4.3, que el uso prefijo del operador `++` significa que `cimaDePila` se incrementa primero, y su nuevo valor se emplea para indexar `elVector`. Las rutinas restantes son igualmente breves, tal y como se muestra en las Figuras 15.5 y 15.6. El operador postfijo `--` empleado en la Figura 15.6 significa que, aunque `cimaDePila` se decrementa, es su valor original el que se emplea para indexar `elVector`.

Obsérvese que la duplicación del vector no afecta a la eficiencia del uso prolongado de la estructura.

Cuando el vector no se duplica, cada operación se realiza en tiempo constante. Por su parte, la complejidad de una operación `apilar` que incluya la duplicación del vector es $O(N)$. Si fuese necesaria la duplicación con mucha frecuencia, deberíamos preocuparnos; sin embargo, esto claramente no es así, ya que la duplicación de un vector de N elementos debe estar precedida por, al menos, $N/2$ operaciones `apilar` que no necesitan de la duplicación del vector. Como consecuencia, podemos repartir el coste $O(N)$ de la duplicación entre las operaciones de `apilar`, aumentando su coste sólo en una pequeña constante. Esta técnica recibe el nombre de *amortización*.

Un ejemplo de amortización en la vida real es el pago de impuestos. En lugar de pagar la cuantía total el 15 de abril, el gobierno recibe el dinero a través de las retenciones. La cuantía total es la misma, la única diferencia es *cuándo* se pagan los impuestos. Lo mismo es cierto para el tiempo empleado en las operaciones `apilar`. Podemos considerar el coste de la duplicación en el momento en el que se produce, o suponerlo repartido entre las operaciones `apilar` previas, igualando su coste. Una cota amortizada indica que valoramos cada operación de una secuencia compartiendo de forma justa el coste total. En nuestro ejemplo, usando este concepto obtenemos que el coste de la duplicación del vector no es en absoluto excesivo, conservándose de hecho el coste (amortizado) constante.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PilaVec
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ************************OPERACIONES PÚBLICAS**************************
10 // void apilar( x )         --> Inserta x
11 // void desapilar( )        --> Elimina el último elemento insertado
12 // Object cima( )           --> Devuelve el último elemento insertado
13 // Object cimaYDesapilar( )--> Devuelve, y elimina, el elemento más reciente
14 // boolean esVacia( )       --> Devuelve true si pila vacía; si no, false
15 // void vaciar( )           --> Elimina todos los elementos
16 // ****************************ERRORES*********************************
17 // desapilar, cima y cimaYDesapilar sobre una pila vacía
18
19 /**
20  * Implementación de las pilas basada en un vector.
21  */
22 public class PilaVec implements Pila
23 {
24     public PilaVec( )
25       { /* Figura 15.3 */ }
26
27     public boolean esVacia( )
28       { return cimaDePila == -1; }
29     public void vaciar( )
30       { cimaDePila = -1; }
31     public void apilar( Object x )
32       { /* Figura 15.4 */ }
33     public Object cima( ) throws DesbordamientoInferior
34       { /* Figura 15.5 */ }
35     public void desapilar( ) throws DesbordamientoInferior
36       { /* Figura 15.5 */ }
37     public Object cimaYDesapilar( ) throws DesbordamientoInferior
38       { /* Figura 15.6 */ }
39
40     private Object [ ] elVector;
41     private int cimaDePila;
42
43     static final int CAPACIDAD_POR_DEFECTO = 10;
44
45     private void duplicarVector( )
46       { /* Implementación usual, no se muestra */ }
47 }
```

Figura 15.2 Esqueleto de la clase `PilaVec`.

```
1  /**
2   * Construye la pila.
3   */
4  public PilaVec( )
5  {
6      elVector = new Object[ CAPACIDAD_POR_DEFECTO ];
7      cimaDePila = -1;
8  }
```

Figura 15.3 Constructor sin parámetros de las `Pilas` basadas en un vector.

```
1  /**
2   * Inserta un nuevo elemento en la pila.
3   * @param x el elemento a insertar.
4   */
5  public void apilar( Object x )
6  {
7      if( cimaDePila + 1 == elVector.length )
8          duplicarVector( );
9      elVector[ ++cimaDePila ] = x;
10 }
```

Figura 15.4 Método `apilar` de las `Pilas` basadas en un vector.

```
1  /**
2   * Devuelve el último elemento insertado en la pila.
3   * @return el último elemento insertado en la pila.
4   * @exception DesbordamientoInferior cuando la pila esté vacía.
5   */
6  public Object cima( ) throws DesbordamientoInferior
7  {
8      if( esVacia( ) )
9          throw new DesbordamientoInferior( "Cima" );
10     return elVector[ cimaDePila ];
11 }
12
13 /**
14  * Elimina el último elemento insertado en la pila.
15  * @exception DesbordamientoInferior cuando la pila esté vacía.
16  */
17 public void desapilar( ) throws DesbordamientoInferior
18 {
19     if( esVacia( ) )
20         throw new DesbordamientoInferior( "Desapilar" );
21     cimaDePila--;
22 }
```

Figura 15.5 Métodos `cima` y `desapilar` de las `Pilas` basadas en un vector.

```
1  /**
2   * Devuelve y elimina el último elemento insertado
3   * en la pila.
4   * @return el último elemento insertado en la pila
5   * @exception DesbordamientoInferior si la pila está vacía.
6   */
7  public Object cimaYDesapilar( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "CimaYDesapilar" );
11     return elVector[ cimaDePila-- ];
12 }
```

Figura 15.6 Método `cimaYDesapilar` de las `Pilas` basadas en un vector.

15.1.2 Colas

Almacenar los elementos de la cola desde el principio del vector provoca que la operación `quitarPrimero` sea muy costosa.

La forma más sencilla de implementar una cola consiste en almacenar sus elementos en un vector, colocando el elemento en cabeza en la primera posición del mismo (es decir, en el índice 0). Si `fin` representa la posición del último elemento de la cola, entonces para insertar un elemento no tenemos más que incrementar `fin`, insertando el elemento en esa posición. El problema es que la operación `quitarPrimero` es muy costosa. Ello es debido a la exigencia de posicionar los elementos de la cola desde el principio del vector, con lo que forzamos a la rutina `quitarPrimero` a desplazar una posición todos los elementos del vector, una vez eliminado el primero.

La acción de `quitarPrimero` se implementa incrementando la posición `cabeza`.

En la Figura 15.7 se muestra cómo se resuelve este problema, bastaría con incrementar `cabeza` cuando se realiza una operación `quitarPrimero`, en lugar de desplazar todos los elementos. Entonces, cuando la cola tiene un único elemento, `fin` y `cabeza` representan la posición en el vector de dicho elemento. Consistentemente, en una cola vacía, `fin` debe inicializarse a `cabeza-1`.

Esta implementación hace que tanto `insertar` como `quitarPrimero` se ejecuten en tiempo constante. El principal problema que presenta esta codificación se aprecia en la primera línea de la Figura 15.8. Después de tres ejecuciones más de `quitarPrimero`, no podríamos añadir más elementos, aunque la cola (más exactamente, el vector que la sustenta) no esté realmente llena. La duplicación del vec-

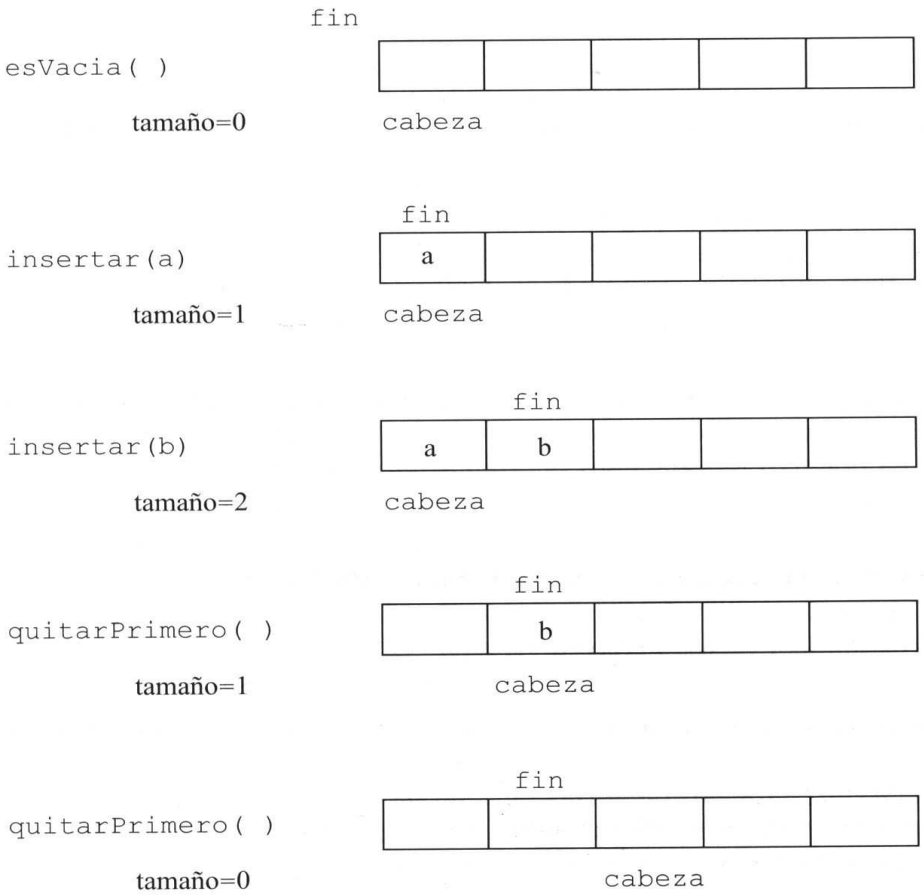

Figura 15.7 Implementación de las colas basada en un vector.

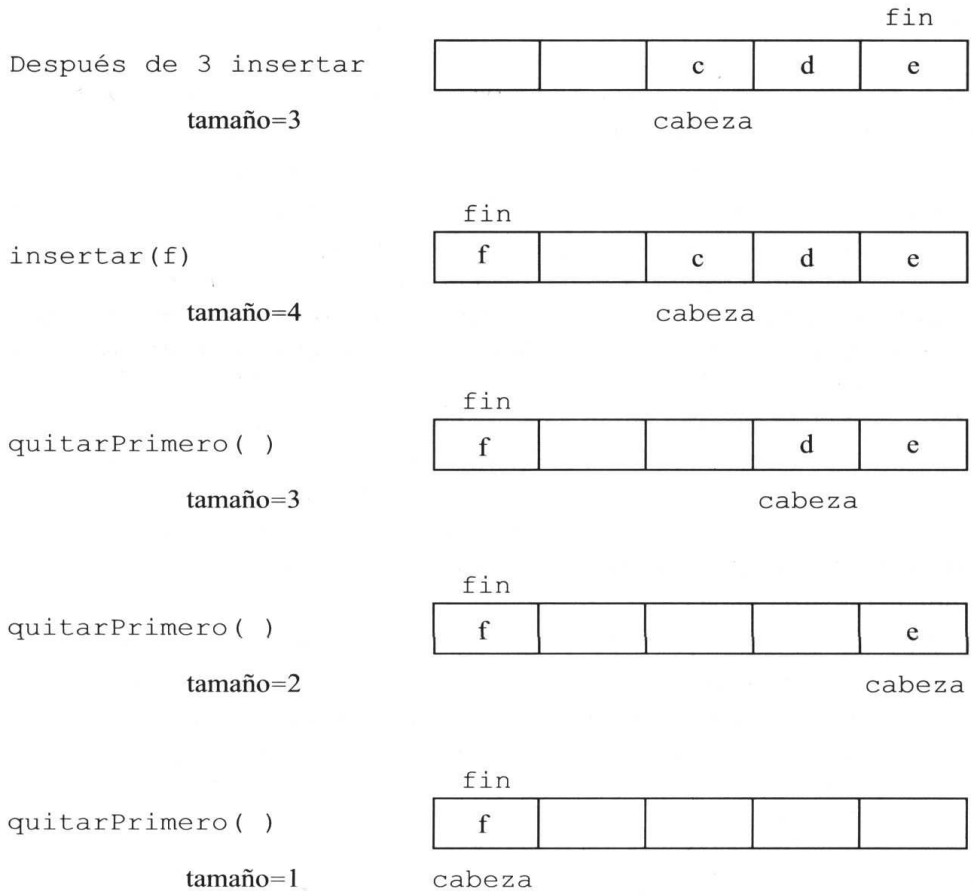

Figura 15.8 Implementación de las colas empleando la implementación circular.

tor sólo resolvería momentáneamente el problema. Esto es debido a que, incluso si el tamaño del vector fuera 1.000, después de 1.000 operaciones `insertar`, no queda espacio libre en la cola, independientemente de su tamaño actual. Incluso si se han realizado 1.000 acciones de `quitarPrimero`, vaciando con ello a nivel abstracto la cola, no podemos añadir ningún elemento más.

La *circularidad* devuelve `cabeza` y `fin` al principio del vector cuando rebasan la última posición. El uso de la circularidad en la implementación de las colas se denomina *implementación circular.*

Sin embargo, tal y como muestra la Figura 15.8, existe un montón de espacio libre; todas las posiciones anteriores a `cabeza` están vacías, por lo que podrían reciclarse. Con tal objetivo introducimos la *circularidad*: cuando `fin` o `cabeza` rebasan la última posición del vector, se reposicionan en el principio del mismo. Esta implementación recibe el nombre de *implementación circular.* Ahora sólo necesitamos duplicar el vector cuando el número de elementos en la cola sea igual al número de elementos insertados en el vector. Para realizar `insertar(f)` situamos `fin` al principio del vector y colocamos `f` en esa posición. Ahora, tras las tres acciones `quitarPrimero`, también situamos `cabeza` al principio del vector.

El esqueleto de la clase genérica `ColaVec` se muestra en la Figura 15.9. La clase `ColaVec` tiene cuatro atributos: un vector que puede expandirse dinámicamente, el número de elementos que están en cada momento en la cola, y las posiciones en el vector del primer y último elemento de la cola.

Si la cola está llena, debemos realizar la duplicación del vector con cuidado.

En la sección privada de la clase se declaran dos métodos. Estos métodos son empleados internamente por los métodos de `ColaVec`, pero no están disponibles para el usuario de la clase. El primero de ellos es la rutina `incrementar`, que

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ColaVec
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ************************OPERACIONES PÚBLICAS**************************
10 // void insertar( x )       --> Inserta x
11 // Object primero( )        --> Devuelve el elemento más antiguo insertado
12 // Object quitarPrimero( ) --> Devuelve y elimina el elemento más antiguo
13 // boolean esVacia( )       --> Devuelve true si cola vacía; si no, false
14 // void vaciar( )           --> Elimina todos los elementos
15 // ***************************ERRORES************************************
16 // primero o quitarPrimero de una cola vacía
17
18 /**
19  * Implementación de las colas basada en un vector.
20  */
21 public class ColaVec implements Cola
22 {
23     public ColaVec( )
24       { /* Figura 15.11 */ }
25
26     public boolean esVacia( )
27       { return tamanyoActual == 0; }
28     public void insertar( Object x )
29       { /* Figura 15.12 */ }
30     public void vaciar( )
31       { /* Figura 15.15 */ }
32     public Object quitarPrimero( ) throws DesbordamientoInferior
33       { /* Figura 15.14 */ }
34     public Object primero( ) throws DesbordamientoInferior
35       { /* Figura 15.14 */ }
36
37     private Object [ ] elVector;
38     private int tamanyoActual;
39     private int cabeza;
40     private int fin;
41
42     static final int CAPACIDAD_POR_DEFECTO = 10;
43
44     private int incrementar( int x )
45       { /* Figura 15.10 */ }
46     private void duplicarCola( )
47     { /* Figura 15.13 */ }
48 }
```

Figura 15.9 Esqueleto de la clase `ColaVec`.

añade 1 a su parámetro y devuelve el nuevo valor. Como servirá para implementar la circularidad, si su resultado es igual al tamaño del vector, se ajusta a cero. Esto se muestra en la Figura 15.10. La otra rutina es `duplicarCola`, que es invocada cuando `insertar` precisa de la duplicación del vector. Es ligeramente más complicada que `duplicarVector`, ya que ahora los elementos de la cola no están ne-

```
1  /**
2   * Método interno para el incremento con circularidad.
3   * @param x cualquier índice en el rango de elVector.
4   * @return x+1, o 0 si x está al final de elVector.
5   */
6  private int incrementar( int x )
7  {
8      if( ++x == elVector.length )
9          x = 0;
10     return x;
11 }
```

Figura 15.10 Rutina de generación de la circularidad.

cesariamente almacenados empezando en la posición cero del vector. Como consecuencia, los elementos deben ser copiados con cuidado. `duplicarCola` se discute junto a `insertar`.

La mayoría de los métodos públicos se asemejan a sus equivalentes en las pilas, incluyendo el constructor mostrado en la Figura 15.11. Este constructor no es particularmente singular, salvado el hecho de que debemos asegurarnos que `cabeza` y `fin` reciben los valores adecuados.

Cuando duplicamos el vector de una cola, no podemos copiar directamente el contenido del vector.

En la Figura 15.12 se muestra la rutina `insertar`. La estrategia básica es muy sencilla, tal y como se muestra en las líneas 9 a 11. `duplicarCola`, mostrado en la Figura 15.13, es similar a la rutina de duplicación de las pilas, salvo en lo que se refiere a la copia de los vectores, en la que el recorrido del vector viejo no comienza en la primera posición. En este caso, en las líneas 11 a 13 el método salta a lo largo del vector antiguo y copia cada elemento en el nuevo. Los resultados se copian en el nuevo vector, recorriendo éste desde la primera posición.

```
1  /**
2   * Construye la cola.
3   */
4  public ColaVec( )
5  {
6      elVector = new Object[ CAPACIDAD_POR_DEFECTO ];
7      vaciar( );
8  }
```

Figura 15.11 Constructor sin parámetros de las `Colas` basadas en un vector.

```
1  /**
2   * Inserta un nuevo elemento en la cola.
3   * @param x el elemento a insertar.
4   */
5  public void insertar( Object x )
6  {
7      if( tamanyoActual == elVector.length )
8          duplicarCola( );
9      fin = incrementar( fin );
10     elVector[ fin ] = x;
11     tamanyoActual++;
12 }
```

Figura 15.12 Método `insertar` de las `Colas` basadas en un vector.

```
1  /**
2   * Método interno para expandir elVector.
3   */
4  private void duplicarCola( )
5  {
6      Object [ ] nuevoVector;
7
8      nuevoVector = new Object[ elVector.length * 2 ];
9
10             // Copia de los elementos de la cola
11     for( int i = 0; i < tamanyoActual;
12                          i++, cabeza = incrementar( cabeza ) )
13         nuevoVector[ i ] = elVector[ cabeza ];
14
15     elVector = nuevoVector;
16     cabeza = 0;
17     fin = tamanyoActual - 1;
18 }
```

Figura 15.13 Expansión dinámica de la clase de las `Colas` basadas en un vector.

Observe los nuevos valores de `cabeza` y `fin`, en las líneas 16 y 17. Las rutinas `quitarPrimero` y `primero` se muestran en la Figura 15.14; ambas son cortas y sencillas. Finalmente, `vaciar` se muestra en la Figura 15.15. De nuevo es claro que las rutinas de las colas se ejecutan en tiempo constante. Del mismo modo que en el caso de las pilas, el coste de la duplicación de vectores puede amortizarse sobre la secuencia de operaciones `insertar`.

```
1  /**
2   * Devuelve y elimina el elemento más antiguo insertado
3   * en la cola.
4   * @return el elemento más antiguo insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object quitarPrimero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "QuitarPrimero" );
11     tamanyoActual--;
12
13     Object valorDevuelto = elVector[ cabeza ];
14     cabeza = incrementar( cabeza );
15     return valorDevuelto;
16 }
17
18 /**
19  * Devuelve el elemento más antiguo insertado en la cola.
20  * @return el elemento más antiguo insertado en la cola.
21  * @exception DesbordamientoInferior si la cola está vacía.
22  */
23 public Object primero( ) throws DesbordamientoInferior
24 {
25     if( esVacia( ) )
26         throw new DesbordamientoInferior( "Primero" );
27     return elVector[ cabeza ];
28 }
```

Figura 15.14 Métodos `quitarPrimero` y `primero` de las `Colas` basadas en un vector.

```
1  /**
2   * Vaciado lógico de la cola.
3   */
4  public void vaciar( )
5  {
6      tamanyoActual = 0;
7      cabeza = 0; fin = -1;
8  }
```

Figura 15.15 Rutina `vaciar` para las `Colas` basadas en un vector.

15.2 Implementaciones con listas enlazadas

La ventaja de las implementaciones con listas enlazadas es que el exceso de memoria consiste en una única referencia por elemento. El inconveniente es que la asignación de memoria consume bastante tiempo.

Una alternativa a la implementación contigua empleando vectores, consiste en utilizar una lista enlazada. Recuerde de la Sección 6.4, que en una lista enlazada almacenamos cada elemento en un objeto distinto, que contiene además una referencia al siguiente objeto de la lista.

La ventaja de las listas enlazadas es que el exceso de gasto de memoria en que se incurre se reduce a una referencia por elemento, mientras que el almacenamiento contiguo en vectores exige un espacio eventualmente mayor, igual al número actual de posiciones vacías del vector (además de algo de memoria adicional durante el proceso de duplicación). Es cierto que en Java esta ventaja es poco relevante ya que las posiciones vacías del vector representan referencias nulas, que consumen muy poco espacio. Sin embargo, en otros lenguajes de programación, esta ventaja sería importante ya que las posiciones vacías suelen guardar instancias de objetos no inicializadas, que consumen un notable espacio. Además de por este motivo, las implementaciones con listas enlazadas se discuten por las siguientes razones:

1. Es importante comprender implementaciones alternativas que pueden ser útiles en otros lenguajes de programación.
2. Las implementaciones que emplean listas enlazadas son más reducidas que las correspondientes versiones con vectores, especialmente en el caso de las colas.
3. Dichas implementaciones ilustran algunos principios que se esconden tras las operaciones generales con listas enlazadas presentadas en el Capítulo 16.

Para que la implementación sea competitiva con las implementaciones contiguas con un vector, debemos ser capaces de ejecutar las operaciones básicas de listas enlazadas en tiempo constante. Esto es sencillo, ya que los cambios en la lista enlazada se limitan a los elementos de los dos extremos (`cabeza` y `fin`) de la lista.

15.2.1 Pilas

Para implementar las pilas, la cima de las mismas está representada por el primer elemento de la lista.

El interfaz `Pila` puede implementarse empleando una lista enlazada, en la que la cima de la pila está representada por el primer elemento de la lista. Esto se ilustra en la Figura 15.16. Para implementar la operación `apilar`, creamos un nuevo nodo y lo incluimos en la lista como primer elemento. Dicho nodo se genera por la

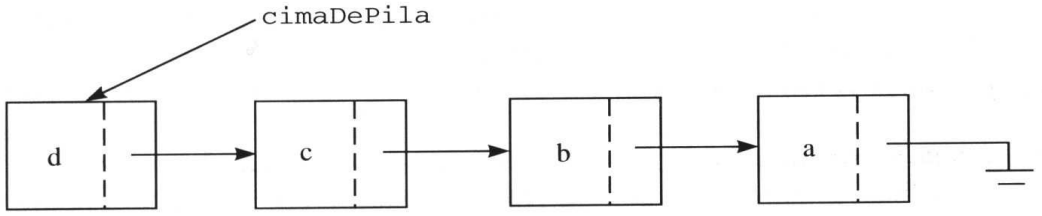

Figura 15.16 Implementación de una pila mediante una lista enlazada.

llamada a `new`. Para implementar el método `desapilar`, basta con desplazar la cima de la pila al segundo elemento de la lista (si es que existe). La pila vacía se representa mediante la lista vacía.

Claramente, cada operación es constante, ya que al restringir las operaciones al primer elemento de la lista, conseguimos que todos los cálculos sean independientes del tamaño de la misma. Todo esto se conserva en las implementaciones en Java.

La Figura 15.17 muestra la declaración de los nodos de la lista. Un `NodoLista` está compuesto por dos atributos: `dato` guarda un elemento y `siguiente` almacena una referencia que apunta al siguiente nodo de la lista. Se dispone de dos constructores para `NodoLista`. El primero de ellos se puede emplear en la forma siguiente

```
NodoLista nodo1 = new NodoLista( x );
```

donde `x` es un elemento. El segundo puede usarse para inicializar la referencia siguiente por medio de una referencia `NodoLista` (o `null`):

```
NodoLista nodo2 = new NodoLista( x, nodo1 );
```

Obsérvese que el acceso a la nueva clase `NodoLista` es amistoso. Esto significa que este tipo no pertenece al ámbito global, pero es visible para el resto de clases

```
1  package EstructurasDatos;
2
3  // Nodo básico almacenado en una lista enlazada.
4  // Nótese que esta clase no es accesible fuera
5  // del paquete EstructurasDatos.
6
7  class NodoLista
8  {
9          // Constructores
10     NodoLista( Object elElemento )
11       { this( elElemento, null ); }
12
13     NodoLista( Object elElemento, NodoLista n )
14       { dato = elElemento; siguiente = n; }
15
16         // Atributos amistosos; accesibles desde otras rutinas del paquete
17     Object dato;
18     NodoLista siguiente;
19 }
```

Figura 15.17 Clase `NodoLista`.

El acceso a la clase NodoLista es amistoso para las estructuras de datos del paquete, por lo que no es visible para el usuario general.

del paquete EstructurasDatos. Esto es positivo, ya que fuerza el ocultamiento de la información. Los NodoLista son un detalle interno del paquete de estructuras de datos, pero son completamente invisibles para los usuarios de la clase Pila.

La Figura 15.18 muestra la clase PilaLi que implementa el interfaz Pila usando una lista enlazada. La pila se representa mediante un único atributo: cimaDePila es una referencia que apunta al primer NodoLista de la lista enlazada. El constructor de la línea 24 indica que se crea una pila vacía inicializando cimaDePila a null.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PilaLi
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ************************OPERACIONES PÚBLICAS************************
10 // void apilar( x )          --> Inserta x
11 // void desapilar( )         --> Elimina el último elemento insertado
12 // Object cima( )            --> Devuelve el último elemento insertado
13 // Object cimaYDesapilar( ) --> Devuelve y elimina el elemento más reciente
14 // boolean esVacia( )        --> Devuelve true si pila vacía; si no, false
15 // void vaciar( )            --> Elimina todos los elementos
16 // ************************ERRORES*************************************
17 // desapilar, cima y cimaYDesapilar sobre una pila vacía
18
19 /**
20  * Implementación de las pilas basada en una lista enlazada.
21  */
22 public class PilaLi implements Pila
23 {
24     public PilaLi( )
25       { cimaDePila = null; }
26
27     public boolean esVacia( )
28       { return cimaDePila == null; }
29     public void vaciar( )
30       { cimaDePila = null; }
31     public void apilar( Object x )
32       { cimaDePila = new NodoLista( x, cimaDePila ); }
33
34     public void desapilar( ) throws DesbordamientoInferior
35       { /* Figura 15.19 */ }
36     public Object cima( ) throws DesbordamientoInferior
37       { /* Figura 15.19 */ }
38     public Object cimaYDesapilar( ) throws DesbordamientoInferior
39       { /* Figura 15.19 */ }
40
41     private NodoLista cimaDePila;
42 }
```

Figura 15.18 Esqueleto de la clase Pila basada en una lista enlazada (PilaLi).

La operación `apilar` requiere una única línea de código: la creación de un nuevo `NodoLista`. El atributo contiene el elemento `x` a insertar, y la referencia `siguiente` para el nuevo nodo es la `cimaDePila` inicial. El nodo creado pasa a ser la nueva cima. Todo esto se realiza en la línea 32

Las rutinas de las pilas pueden implementarse en una sola línea.

Las rutinas restantes se muestran el la Figura 15.19. La operación `desapilar` también es muy sencilla. Tras el test obligatorio para detectar la pila vacía, se actualiza `cimaDePila` de modo que apunte al segundo elemento de la lista. Análogamente, `cima` y `cimaYDesapilar` son rutinas muy breves.

```
1  /**
2   * Devuelve el último elemento insertado en la pila.
3   * No modifica la pila.
4   * @return el último elemento insertado en la lista.
5   * @exception DesbordamientoInferior si la pila está vacía.
6   */
7  public Object cima( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "Cima" );
11     return cimaDePila.dato;
12 }
13
14 /**
15  * Devuelve el último elemento insertado en la pila.
16  * @exception DesbordamientoInferior si la pila está vacía.
17  */
18 public void desapilar( ) throws DesbordamientoInferior
19 {
20     if( esVacia( ) )
21         throw new DesbordamientoInferior( "Desapilar" );
22     cimaDePila = cimaDePila.siguiente;
23 }
24
25 /**
26  * Devuelve y elimina el último elemento insertado
27  * en la pila.
28  * @return el último elemento insertado en la pila.
29  * @exception DesbordamientoInferior si la pila está vacía.
30  */
31 public Object cimaYDesapilar( ) throws DesbordamientoInferior
32 {
33     if( esVacia( ) )
34         throw new DesbordamientoInferior( "cimaYDesapilar" );
35
36     Object datoCima = cimaDePila.dato;
37     cimaDePila = cimaDePila.siguiente;
38     return datoCima;
39 }
```

Figura 15.19 Componentes sencillas de la clase `Pila` basada en una lista enlazada (`PilaLi`).

15.2.2 Colas

Puede usarse una lista enlazada en la que se mantienen referencias al principio y al final de la lista, para implementar la funcionalidad de la cola.

Las colas pueden implementarse empleando una lista enlazada, siempre que se mantengan referencias que apunten al principio y al final de la lista. La Figura 15.20 ilustra la idea general.

Los métodos de las colas son prácticamente idénticos a los de las pilas. En la Figura 15.21 se muestra el esqueleto de la clase `ColaLi`. En ella no hay nada nuevo, fuera del hecho de que mantenemos dos referencias en lugar de una.

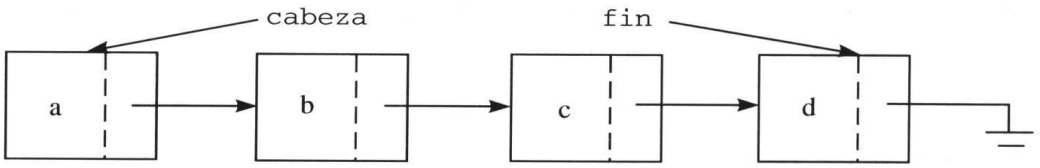

Figura 15.20 Implementación de una cola basada en una lista enlazada.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ColaLi
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ********************OPERACIONES PÚBLICAS*************************
10 // void insertar( x )        --> Inserta x
11 // Object primero( )         --> Devuelve el elemento más antiguo insertado
12 // Object quitarPrimero( )   --> Devuelve y elimina el elemento más reciente
13 // boolean esVacia( )        --> Devuelve true si cola vacía; si no, false
14 // void vaciar( )            --> Elimina todos los elementos
15 // ************************ERRORES*********************************
16 // primero o quitarPrimero sobre una cola vacía
17
18 /**
19  * Implementación de las colas basada en una lista enlazada.
20  */
21 public class ColaLi implements Cola
22 {
23     public ColaLi( )
24       { vaciar( ); }
25
26     public boolean esVacia( )
27       { return cabeza == null; }
28     public void vaciar( )
29       { cabeza = fin = null; }
30     public Object quitarPrimero( ) throws DesbordamientoInferior
31       { /* Figura 15.22 */ }
32     public Object primero( ) throws DesbordamientoInferior
33       { /* Figura 15.23 */ }
34     public void insertar( Object x )
35       { /* Figura 15.25 */
36
37     private NodoLista cabeza;
38     private NodoLista fin;
39 }
```

Figura 15.21 Esqueleto de la clase `Cola` basada en una lista enlazada (`ColaLi`).

En la Figura 15.22 se implementa `quitarPrimero`. Dicha rutina es lógicamente equivalente a la operación de las pilas `desapilar`. `primero` es muy sencillo y se muestra en la Figura 15.23.

El método `insertar` distingue dos casos. Si la cola está vacía, creamos una cola con un único elemento, invocando a `new`, y actualizando `cabeza` y `fin` de modo que ambos referencien al único nodo existente. En caso contrario, creamos un nuevo nodo con el dato `x`, lo añadimos al final de la lista y actualizamos el final de la lista a ese nuevo nodo. Todo esto se ilustra en la Figura 15.24. Nótese que la inserción del primer elemento es un caso especial, ya que no existe ninguna referencia `siguiente` a la que pueda apuntar el nuevo nodo. Todo esto se hace en la Figura 15.25.

La inserción del primer elemento es un caso especial, ya que no existe ninguna referencia `siguiente` a la que pueda apuntar el nuevo nodo.

```
1  /**
2   * Devuelve y elimina el elemento más antiguo insertado
3   * en la cola.
4   * @return el elemento más antiguo insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object quitarPrimero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "QuitarPrimero" );
11
12     Object valorDevuelto = cabeza.dato;
13     cabeza = cabeza.siguiente;
14     return valorDevuelto;
15 }
```

Figura 15.22 Método `quitarPrimero` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

```
1  /**
2   * Devuelve el último elemento insertado en la cola.
3   * No modifica la cola.
4   * @return el último elemento insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object primero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "Primero" );
11     return cabeza.dato;
12 }
```

Figura 15.23 Método `primero` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

15.3 Comparación de los dos métodos

Las operaciones de las dos versiones presentadas, basadas en vectores y listas enlazadas, se ejecutan en tiempo constante. Por tanto, son tan rápidas como para no suponer un cuello de botella para ningún algoritmo. Como consecuencia, raramente importa qué versión se esté empleando.

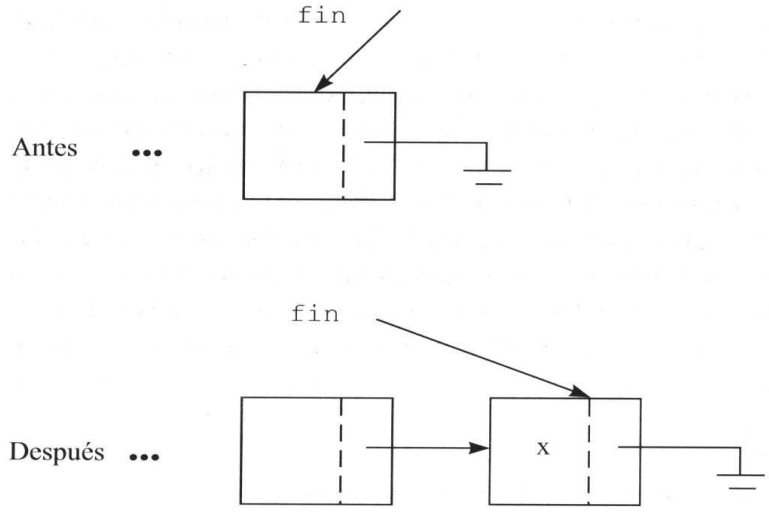

Figura 15.24 Operación `insertar` para la implementación basada en una lista enlazada.

```
1  /**
2   * Inserta un nuevo elemento en la cola.
3   * @param x el elemento a insertar.
4   */
5  public void insertar( Object x )
6  {
7      if( esVacia( ) ) // Genera una cola de un solo dato
8          fin = cabeza = new NodoLista( x );
9      else // Caso regular
10         fin = fin.siguiente = new NodoLista( x );
11 }
```

Figura 15.25 Método `insertar` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

Las versiones basadas en vectores parecen ser más rápidas que las basadas en listas enlazadas, especialmente cuando se dispone a priori de una estimación ajustada de la capacidad. Si se cuenta con un constructor adicional que permita especificar la capacidad inicial (*véase* el Ejercicio 15.3) y la estimación es correcta, no será necesario realizar ninguna duplicación. Además, el acceso secuencial permitido por el vector, es típicamente más rápido que el acceso potencialmente no secuencial de la memoria dinámica.

Sin embargo, la implementación basada en vectores presenta dos inconvenientes. En primer lugar, en el caso de las colas, la implementación mediante vectores es más compleja que la implementación mediante listas enlazadas, debido al código necesario para la circularidad y sus implicaciones en la duplicación del vector. Nuestra implementación de la duplicación de vectores no es tan eficiente como podría hacerse (*véase* el Ejercicio 15.8). Una implementación más rápida necesitaría algunas líneas adicionales de código. También la implementación de las pilas basada en un vector es algo más larga que la correspondiente basada en una lista enlazada.

El segundo inconveniente aparece en otros lenguajes de programación, pero no en Java. Cuando duplicamos un vector, necesitamos temporalmente tres veces más espacio de lo que la cantidad de datos podría sugerir. Esto es debido a que en la duplicación se necesita memoria para almacenar tanto el vector viejo como el nuevo (de tamaño doble). Más aún, si la cola alcanza un gran tamaño, el índice de ocupación del vector estará entre el 50 y el 100 por ciento: en media, esto supone que está lleno en un 75 por ciento, es decir, que por cada tres elementos del vector, una posición está vacía. Como consecuencia, el espacio perdido en media es el 33 por ciento del utilizado, y cuando la tabla está medio llena llega a ser del 100 por ciento. En Java esto no supone un gran problema, ya que cada elemento del vector es sólo una referencia. En otros lenguajes, como C++, los objetos se almacenan directamente, en lugar de ser referenciados. En estos lenguajes, el espacio perdido puede ser significativo, cuando lo comparamos con las versiones basadas en listas enlazadas, que requieren únicamente una referencia extra por cada elemento.

15.4 Colas dobles

Una *cola doble* permite el acceso por ambos extremos de la estructura. La mayor parte de su funcionalidad puede obtenerse de la clase de las colas.

Este capítulo concluye con una discusión acerca del uso de la herencia para derivar una nueva estructura de datos. Una *cola doble* es similar a una cola, excepto por el hecho de que está permitido el acceso por ambos extremos. El Ejercicio 14.11 describe una aplicación de las colas dobles. Los términos empleados en lugar de `insertar` y `quitarPrimero`, son `anyadirPorDelante`, `anyadirPorDetras`, `eliminarPorDelante` y `eliminarPorDetras`. La Figura 15.26 muestra la clase derivada `ColaDoble`. La derivamos a partir de `ColaVec` ya que la eliminación de nodos al final de una lista no está soportada de forma eficiente.

La implementación de una cola doble es sencilla si se emplea herencia.

El constructor por defecto de esta clase vendrá dado por una llamada a `super`, lo que resulta adecuado. `insertar` y `quitarPrimero` pueden emplearse, puesto que al extender clases no podemos eliminar métodos. El método `primero` de la clase de las colas no se cambia, se hereda sin modificación alguna. `anyadirPorDetras` y `eliminarPorDelante` invocan a las rutinas `insertar` y `quitarPrimero`. Las únicas rutinas que han de ser completamente implementadas son `anyadirPorDelante`, `eliminarPorDetras` y `verFinal`. Para hacerlo, necesitamos que los atributos de la clase original sean protegidos, tras lo cual podemos escribir los nuevos métodos. Esto se deja como ejercicio al lector en el Ejercicio 15.5.

Resumen

En este capítulo se describen las implementaciones de las clases de las pilas y de las colas. Ambos tipos de estructuras pueden representarse empleando un vector o una lista enlazada. En ambos casos, las operaciones se ejecutan en tiempo constante, por lo que todas ellas son muy rápidas.

En otros lenguajes de programación, la implementación basada en vectores emplea más memoria pero menos tiempo que la correspondiente implementación basada en listas enlazadas, conduciéndonos a la típica disquisición de tiempo versus memoria. En Java, esta discusión no es muy relevante, debido al tratamiento de los objetos con referencias.

```
1  package EstructurasDatos;
2  // Clase de las colas dobles
3  //
4  // CONSTRUCCIÓN: sin ninguna inicialización
5  //
6  // ***************************OPERACIONES PÚBLICAS*****************************
7  // void anyadirPorDelante( Object x )--> Inserta x por delante
8  // void anyadirPorDetras( Object x ) --> Inserta x por detrás
9  // Object eliminarPorDelante( )      --> Elimina el elemento de la cabeza
10 // Object eliminarPorDetras( )       --> Elimina el elemento del final
11 // Object primero( )                 --> Devuelve el elemento de la cabeza
12 // Object verFinal( )                --> Devuelve el elemento del final
13 // boolean esVacia( )                --> Devuelve true si vacía; si no, false
14 // void vaciar( )                    --> Elimina todos los elementos
15 // insertar y quitarPrimero están disponibles, pero no deben emplearse
16 // ****************************ERRORES*************************************
17 // Se lanza una excepción si se elimina o se consulta una cola doble vacía
18
19 class ColaDoble extends ColaVec
20 {
21     public void anyadirPorDelante( Object x )
22       { /* Ejercicio 15.5 */ }
23     public void anyadirPorDetras( Object x )
24       { insertar( x ); }
25     public Object eliminarPorDelante( ) throws DesbordamientoInferior
26       { return quitarPrimero( ); }
27     public Object eliminarPorDetras( ) throws DesbordamientoInferior
28       { /* Ejercicio 15.5 */ }
29     public Object verFinal( ) throws DesbordamientoInferior
30       { /* Ejercicio 15.5 */ }
31     // esVacia, vaciar y primero son métodos heredados
32 }
```

Figura 15.26 Clase de las colas dobles `ColaDoble` derivada a partir de `ColaVec`.

Elementos del juego

circularidad Se produce cuando `cabeza` o `fin` regresan al principio del vector al haber rebasado la última posición del mismo.

cola doble Cola en la que está permitido el acceso por ambos extremos. La mayor parte de su funcionalidad se deriva de la clase de las colas.

implementación circular de vectores Implementación en la que se emplea la circularidad para representar una cola.

Errores comunes

1. La implementación circular de vectores requerida en el caso de las colas puede hacerse fácilmente de forma errónea cuando se intenta reducir el código necesario. Por ejemplo, no debe evitarse el uso de `tamanyoActual`, intentando calcular el tamaño del vector a partir de `fin` y `cabeza`.
2. Utilizar una implementación de estas estructuras en la que no pueda garantizarse el acceso a la estructura de datos en tiempo constante es un grave error. No existe ninguna justificación para una ineficiencia tal.

En Internet

Están disponibles los ficheros que se citan a continuación, encontrándose en el directorio **DataStructures**. Algunos programas de prueba se vieron en el Capítulo 6, mientras que otros se emplean en las aplicaciones de la Parte III del libro.

StackAr.java	Implementa una pila empleando un vector. Es la versión inglesa de la clase `PilaVec`.
StackLi.java	Implementa una pila empleando una lista enlazada. Es la versión inglesa de la clase `PilaLi`.
QueueAr.java	Implementa una cola empleando un vector. Es la versión inglesa de la clase `ColaVec`.
QueueLi.java	Implementa una cola empleando una lista enlazada. Es la versión inglesa de la clase `ColaLi`.

Ejercicios

Cuestiones breves

15.1. Dibuje el valor de las estructuras que soportan la implementación de las pilas y de las colas (para las implementaciones con vectores y con listas enlazadas) tras cada paso de la siguiente secuencia de operaciones: *añadir*(1), *añadir*(2), *eliminar*, *añadir*(3), *añadir*(4), *eliminar*, *eliminar*, *añadir*(5). En la implementación con vectores, suponer que el tamaño inicial de la estructura es tres.

Problemas prácticos

15.3. Compare los tiempos de ejecución de las dos versiones de la clase `Pila`, la basada en vectores y la basada en listas enlazadas. Emplee objetos de tipo `Integer`.

15.3. Añada constructores a las clases de las pilas y de las colas en sus versiones basadas en vectores. Dichos constructores deben permitir especificar la capacidad inicial del vector.

15.4. Escriba un método `main` que cree y manipule simultáneamente dos pilas, una de objetos de tipo `Integer` y otra de objetos de tipo `Double`.

15.5. Complete la implementación de la clase `DobleCola`.

15.6. Implemente la clase de las pilas basadas en un vector empleando la clase predefinida `Vector`. ¿Cuáles son las ventajas y desventajas de esta aproximación al problema?

15.7. Implemente la clase de las pilas basadas en un vector extendiendo la clase predefinida `Vector`. ¿Cuáles son las ventajas y desventajas de esta aproximación al problema?

15.8. El código de la duplicación de vectores de la clase `ColaVec` es excesivamente costoso debido a las repetidas llamadas a `incrementar`. Escriba una versión más eficiente, evitando todas las llamadas a `incrementar`. Para hacerlo debe controlar cuando se emplea la circularidad. De ser así, emplee dos bucles separados para realizar la copia. En cualquier otro caso, use un solo bucle.

15.9. Usar un `Vector` para implementar una cola basada en un vector es algo problemático, ya que el método `setSize` no es lo que se necesita. Muestre cómo redistribuir los elementos de la cola tras la operación `setSize`, de modo que sólo cambien, a lo sumo, la mitad de las referencias del `Vector`. (No obstante, esto sigue empleando más movimientos que la implementación que no usa la clase `Vector`.)

Prácticas de programación

15.10. Una cola doble restringida permite insertar elementos en ambos extremos de la estructura, pero sólo permite accesos y eliminaciones en la cabeza. Se pide hacer lo siguiente:

a) Emplee herencia para derivar esta nueva clase a partir de `Cola`.
b) Emplee herencia para derivar esta nueva clase a partir de `ColaDoble`.

15.11. Supongamos que deseamos añadir la operación `encontrarMinimo` (pero no `eliminarMinimo`) en el repertorio de las pilas. Se pide hacer lo siguiente:

a) Emplee herencia para derivar la nueva clase, e implemente `encontrarMinimo` de modo que se recorra secuencialmente la pila de elementos.
b) En lugar de emplear herencia, implemente la nueva clase empleando dos pilas, tal y como se propuso en el Ejercicio 6.5.
c) Emplee herencia, junto con el algoritmo descrito en el Ejercicio 6.5, para la derivación de la nueva clase.

15.12. Supongamos que queremos añadir la operación `encontrarMinimo` (pero no `eliminarMinimo`) en el repertorio de las colas dobles. Se pide hacer lo siguiente:

a) Emplee herencia para derivar la nueva clase, e implemente `encontrarMinimo` de modo que se recorra secuencialmente la pila de elementos. Debe hacer las elecciones adecuadas para decidir que atributos serán protegidos.
b) En lugar de emplear herencia, implemente la nueva clase empleando cuatro colas. Si una eliminación vacía una cola, deben reorganizarse los elementos restantes.

15.13. Implemente un applet que muestre cómo varía el estado de una pila cuando se aplican sobre ella las operaciones básicas. Incluya una GUI que permita al usuario especificar las operaciones. Además, debe permitirse la elección (mediante una componente de elección) entre la implementación basada en vectores y la implementación basada en listas enlazadas.

15.14. Repita el Ejercicio 15.13 para el caso de una cola.