Figura 6.27 Modelo de cola: la introducción de datos se realiza mediante `enqueue`, la consulta mediante `getFront` y el borrado mediante `dequeue`.

Las operaciones con una cola consumen un tiempo constante.

Puesto que las operaciones con una cola y las operaciones con una pila están restringidas de forma similar, cabe esperar que también necesitan un tiempo constante por cada consulta, y efectivamente es así. Todas las operaciones básicas con la cola tardan un tiempo $O(1)$. Veremos varias aplicaciones de las colas en los casos de estudio.

6.6.4 Pilas y colas en la API de Colecciones

La API de colecciones proporciona una clase `Stack`, pero no proporciona ninguna clase para colas. Java 5 añade una interfaz `Queue`.

La API de Colecciones proporciona una clase `Stack` para la pilas, pero no proporciona ninguna clase para las colas. Los métodos de `Stack` son `push`, `pop` y `peek`. Sin embargo, la clase `Stack` amplía `Vector` y es más lenta de lo necesario; como `Vector`, su uso ya no se recomienda y puede sustituirse por operaciones con `List`. Antes de Java 1.4, el único soporte de `java.util` para las operaciones con colas era utilizar una `LinkedList` (por ejemplo, `addLast`, `removeFirst` y `getFirst`). Java 5 añade una interfaz `Queue` para las colas, parte de la cual se muestra en la Figura 6.28. Sin embargo, necesitamos seguir empleando métodos de `LinkedList`. Los nuevos métodos son `add`, `remove` y `element`.

6.7 Conjuntos

Un conjunto `Set` no contiene duplicados.

Un conjunto `Set` es un contenedor que no contiene duplicados. Soporta todos los métodos de `Collection`. Lo más importante es que, tal como explicamos en la Sección 6.5.3, `contains` para una lista `List` es ineficiente, independientemente de si la `List` es un `ArrayList` o una `LinkedList`. Por el contrario, una implementación de librería de `Set` debe soportar de manera eficiente la operación `contains`. De forma similar, el método `remove` de `Collection` (que tiene como parámetro un objeto especificado, no un índice especificado) es ineficiente para una `List`, porque está implícito que lo primero que tiene que hacer `remove` es encontrar el elemento que hay que eliminar; esencialmente, esto hace que `remove` sea al menos tan difícil como `contains`. Para un `Set`, también se espera que `remove` esté implementado eficientemente. Y finalmente, se espera asimismo que `add` tenga una implementación eficiente. No hay sintaxis en Java que pueda utilizarse para especificar que una operación deba satisfacer una determinada restricción de tiempo de ejecución, o para indicar que una cierta colección no contenga duplicados; por ello, la Figura 6.29 ilustra que la interfaz `Set` hace poco más que declarar un tipo.

Un `SortedSet` es un `Set` que mantiene (internamente) sus elementos ordenados. Los objetos añadidos al `SortedSet` deben ser comparables, o sino habrá que proporcionar un `Comparator` en

`SortedSet` es un contenedor ordenado. No permite duplicados.

el momento de instanciar un contenedor. Un `SortedSet` soporta todos los métodos de `Set`, pero se garantiza que su iterador recorra todos los elementos por orden. El `SortedSet` también nos permite encontrar los elementos mayor y menor. La interfaz para nuestro subconjunto de `SortedSet` se muestra en la Figura 6.30.

```
1  package weiss.util;
2
3  /**
4   * Interfaz Queue.
5   */
6  public interface Queue<AnyType> extends Collection<AnyType>
7  {
8     /**
9      * Devuelve pero no elimina el elemento situado al "principio"
10     * de la cola.
11     * @return el elemento inicial o null si la cola está vacía.
12     * @throws NoSuchElementException si la cola está vacía.
13     */
14    AnyType element( );
15
16    /**
17     * Devuelve pero no elimina el elemento situado al "principio"
18     * de la cola.
19     * @return el elemento inicial.
20     * @throws NoSuchElementException si la cola está vacía.
21     */
22    AnyType remove( );
23 }
```

Figura 6.28 Posible interfaz `Queue`.

```
1  package weiss.util;
2
3  /**
4   * Interfaz Set.
5   */
6  public interface Set<AnyType> extends Collection<AnyType>
7  {
8  }
```

Figura 6.29 Posible interfaz `Set`.

```
1  package weiss.util;
2
3  /**
4   * Interfaz SortedSet.
5   */
6  public interface SortedSet<AnyType> extends Set<AnyType>
7  {
8     /**
9      * Devuelve el comparador utilizado por este SortedSet.
10     * @return el comparador o null si se utiliza
11     * el comparador predeterminado.
12     */
13    Comparator<? super AnyType> comparator( );
14
15    /**
16     * Encuentra el elemento más pequeño del conjunto.
17     * @return el elemento más pequeño.
18     * @throws NoSuchElementException si el conjunto está vacío.
19     */
20    AnyType first( );
21
22    /**
23     * Encuentra el elemento más grande del conjunto.
24     * @return el elemento más grande.
25     * @throws NoSuchElementException si el conjunto está vacío.
26     */
27    AnyType last( );
28 }
```

Figura 6.30 Posible interfaz `SortedSet`.

6.7.1 La clase TreeSet

`TreeSet` es una implementación de `SortedSet`.

El `SortedSet` está implementado mediante un `TreeSet`. La implementación subyacente del `TreeSet` es un árbol equilibrado de búsqueda binaria y se explica en el Capítulo 19.

Si no se especifica lo contrario, la ordenación utiliza el comparador predeterminado. Podemos especificar una ordenación alternativa proporcionando un comparador al constructor. Como ejemplo, la Figura 6.31 ilustra cómo construir un `SortedSet` que almacene cadenas de caracteres. La llamada a `printCollection` permitirá imprimir los elementos en orden decreciente.

El `TreeSet`, como todos los conjuntos `Set`, no permite duplicados. Dos elementos se consideran iguales si el método `compare` del comparador devuelve 0.

```
1    public static void main( String [] args )
2    {
3       Set<String> s = new TreeSet<String>( Collections.reverseOrder( ) );
4       s.add( "joe" );
5       s.add( "bob" );
6       s.add( "hal" );
7       printCollection( s ); // Figura 6.11
8    }
```

Figura 6.31 Ilustración de `TreeSet`, utilizando ordenación inversa.

En la Sección 5.6 hemos examinado el problema de la búsqueda estática y vimos que si se nos presentan los elementos ordenados, podemos soportar la operación `find` en un tiempo logarítmico de caso peor. Esto es una búsqueda estática, porque después de que se nos presenten los elementos, no podemos añadir ni eliminar ningún elemento. El `SortedSet` sí que nos permite añadir y eliminar elementos.

Esperamos que el coste de caso peor de las operaciones `contains`, `add` y `remove` sea $O(\log N)$, porque eso se correspondería con la cota obtenida para la búsqueda binaria estática. Lamentablemente, para la implementación más simple de `TreeSet`, esto no se cumple. El caso medio es logarítmico, pero el caso peor es $O(N)$ y se presenta de forma bastante frecuente. Sin embargo, aplicando algunos trucos algorítmicos, podemos obtener una estructura más compleja que sí que tenga un coste de $O(\log N)$ por operación. Se garantiza que el `TreeSet` de la API de Colecciones tenga este rendimiento, y en el Capítulo 19 explicaremos cómo obtenerlo utilizando el *árbol de búsqueda binaria* y sus variantes, y proporcionaremos una implementación de `TreeSet` con un iterador.

Podemos también utilizar un árbol de búsqueda binaria para acceder al *K*-ésimo elemento más pequeño en un tiempo logarítmico.

Mencionemos, para terminar, que aunque es posible encontrar los elementos mayor y menor en un `SortedSet` en un tiempo $O(\log N)$, encontrar el *K-ésimo* elemento más pequeño, donde K sea un parámetro, no está soportado en la API de Colecciones. Y sin embargo, es posible realizar esta operación en un tiempo $O(\log N)$, mientras se preserva el tiempo de ejecución de las restantes operaciones, aunque para ello hace falta algo más de trabajo.

6.7.2 La clase `HashSet`

El `HashSet` implementa la interfaz `Set`. No requiere un comparador.

Además del `TreeSet`, la API de Colecciones proporciona una clase `HashSet` que implementa la interfaz `Set`. El `HashSet` difiere de `TreeSet` en que no puede utilizarse para enumerar elementos por orden, ni tampoco para obtener los elementos mayor y menor. De hecho, los elementos del `HashSet` no tienen por qué ser comparables en modo alguno. Esto significa que el `HashSet` es menos potente que el `TreeSet`. Si no es importante tener que enumerar los elementos de un `Set` en orden, entonces a menudo es preferible utilizar un `HashSet`, porque al no tener que mantener la ordenación, el `HashSet` permite obtener un mejor rendimiento. Para poder hacer esto, los elementos incluidos en el `HashSet` deben proporcionar una serie de pistas a los algoritmos de `HashSet`. Esto se

```
1    public static void main( String [] args )
2    {
3       Set<String> s = new HashSet<String>( );
4       s.add( "joe" );
5       s.add( "bob" );
6       s.add( "hal" );
7       printCollection( s ); // Figura 6.11
8    }
```

Figura 6.32 Una ilustración de `HashSet`, en la que los elementos se imprimen en un orden desconocido.

lleva a cabo haciendo que cada elemento implemente un método `hashCode` especial; describiremos este método más adelante dentro de esta subsección.

La Figura 6.32 ilustra el uso del `HashSet`. Se garantiza que, si iteramos a través de todo el `HashSet`, visualizaremos cada elemento una sola vez, pero el orden en el que se recorrerán los elementos no está definido. Es casi seguro que dicho orden no coincidirá con el orden de inserción, ni tampoco con ningún otro criterio de ordenación.

Como todos los conjuntos `Set`, el `HashSet` no permite duplicados. Dos elementos se consideran iguales, si el método `equals` dice que lo son. Por tanto, cualquier objeto que se inserte en el `HashSet` debe tener un método `equals` apropiadamente definido.

Recuerde que, en la Sección 4.9, explicamos que es esencial sustituir la definición de `equals` (proporcionando una nueva versión de `equals` que admita un `Object` como parámetro), en lugar de sobrecargar dicho método.

Implementación de `equals` y `hashCode`

`equals` debe ser simétrica. Esto puede resultar problemático cuando está implicada la herencia.

La sustitución de `equals` es bastante complicada cuando están implicados los mecanismos de herencia. La especificación de `equals` establece que si `p` y `q` no son `null`, `p.equals(q)` debe devolver el mismo valor que `q.equals(p)`. Esto no es así en la Figura 6.33. En ese ejemplo, claramente `b.equals(c)` devuelve `true`, como cabía esperar. La expresión `a.equals(b)` también devuelve `true`, porque se utiliza el método `equals` de `BaseClass`, y ese método solo compara los componentes `x`. Sin embargo, `b.equals(a)` devuelve `false`, porque se utiliza el método `equals` de `DerivedClass` y, en la línea 29, la comprobación `instanceof` fallará (`a` no es una instancia de `DerivedClass`).

Hay dos soluciones estándar para este problema. Una consiste en hacer que el método `equals` sea final en `BaseClass`. Esto evita el problema de tener métodos `equals` que entran en conflicto. La otra solución consiste en forzar la comprobación `equals` para exigir que los tipos sean idénticos y no simplemente compatibles, ya que es la compatibilidad unidireccional lo que hace que deje de funcionar `equals`. En este ejemplo, un objeto de tipo `BaseClass` y otro de tipo `DerivedClass` nunca se declararían como iguales. La Figura 6.34 muestra una implementación correcta. La línea 8 contiene la comprobación fundamental. `getClass` devuelve un objeto especial de tipo `Class` (observe la `C` mayúscula) que contiene información acerca de clase de un objeto. `getClass` es un método final de la clase `Object`. Si al invocarlo

La solución 1 consiste en no sustituir `equals` por debajo de la clase base. La solución 2 consiste en exigir que se comparen objetos de tipo idéntico utilizando para ello `getClass`.

```
1  class BaseClass
2  {
3     public BaseClass( int i )
4        { x = i; }
5
6        public boolean equals( Object rhs )
7     {
8        // Esta es una comprobación errónea (ok para una clase final)
9        if( !( rhs instanceof BaseClass ) )
10          return false;
11
12       return x == ( (BaseClass) rhs ).x;
13    }
14
15    private int x;
16 }
17
18 class DerivedClass extends BaseClass
19 {
20    public DerivedClass( int i, int j )
21    {
22       super( i );
23       y = j;
24    }
25
26    public boolean equals( Object rhs )
27    {
28       // Esta es una comprobación errónea.
29       if( !( rhs instanceof DerivedClass ) )
30          return false;
31
32       return super.equals( rhs ) &&
33       y == ( (DerivedClass) rhs ).y;
34    }
35
36    private int y;
37 }
38
39 public class EqualsWithInheritance
40 {
41    public static void main( String [ ] args )
42    {
43       BaseClass a = new BaseClass( 5 );
44       DerivedClass b = new DerivedClass( 5, 8 );
45       DerivedClass c = new DerivedClass( 5, 8 );
46
47       System.out.println( "b.equals(c): " + b.equals( c ) );
48       System.out.println( "a.equals(b): " + a.equals( b ) );
49       System.out.println( "b.equals(a): " + b.equals( a ) );
50    }
51 }
```

Figura 6.33 Una ilustración de una implementación incorrecta de `equals`.

```
1  class BaseClass
2  {
3     public BaseClass( int i )
4        { x = i; }
5
6     public boolean equals( Object rhs )
7     {
8        if( rhs == null || getClass( ) != rhs.getClass( ) )
9           return false;
10
11       return x == ( (BaseClass) rhs ).x;
12    }
13
14    private int x;
15 }
16
17 class DerivedClass extends BaseClass
18 {
19    public DerivedClass( int i, int j )
20    {
21       super( i );
22       y = j;
23    }
24
25    public boolean equals( Object rhs )
26    {
27          // Comprobación de clase no necesaria; getClass() se invoca
28          // en el método equals de la superclase.
29       return super.equals( rhs ) &&
30          y == ( (DerivedClass) rhs ).y;
31    }
32
33    private int y;
34 }
```

Figura 6.34 Implementación correcta de `equals`.

con dos objetos diferentes devuelve la misma instancia de `Class`, entonces los dos objetos tienen tipos idénticos.

Al utilizar un `HashSet`, también tenemos que sustituir el método especial `hashCode` especificado en `Object`; `hashCode` devuelve un valor `int`. Piense en `hashCode` como en algo que proporcionara una pista de confianza acerca del lugar en el que los elementos están almacenados. Si la pista es errónea, no podrá encontrarse el elemento, así que si dos elementos son iguales deberían proporcionar

Si se sustituye `equals` hay que sustituir también el método `hashCode`, porque sino el `hashSet` no funcionará.

pistas idénticas. La especificación de `hashCode` dice que si dos objetos son declarados iguales por el método `equals`, entonces el método `hashCode` debe devolver el mismo valor para ellos, y si se viola esta especificación, el `HashSet` no podrá encontrar los objetos, aun cuando `equals` declare que hay una correspondencia. Si `equals` declara que los objetos no son iguales, el método `hashCode` debería devolver un valor distinto para ellos, aunque esto no es obligatorio. Sin embargo, aunque no sea obligatorio sí que es muy beneficioso para el rendimiento de `HashSet` que `hashCode` solo proporcione resultados idénticos para objetos desiguales en raras ocasiones. En el Capítulo 20 explicaremos cómo interactúan `hashCode` y `HashSet`.

La Figura 6.35 ilustra una clase `SimpleStudent` en la que dos objetos `SimpleStudent` son iguales si ambos tienen el mismo nombre (y ambos son de tipo `SimpleStudent`). Esto podría sustituirse utilizando las técnicas de la Figura 6.34 de la forma necesaria, o bien declarando este método como `final`. Si se le declara `final`, entonces la comprobación utilizada solo permitirá declarar como iguales a dos objetos `SimpleStudent` que sean de tipo idéntico. Si, teniendo un `equals` final, sustituimos la comprobación de la línea 40 por una comprobación `instanceof`, entonces cualesquiera dos objetos de la jerarquía podrán ser declarados iguales si sus nombres se corresponden.

El método `hashCode` de las líneas 47 y 48 simplemente utiliza el `hashCode` del campo `name`. Así, si dos objetos `SimpleStudent` tienen el mismo nombre (tal como lo declare `equals`), tendrán también el mismo `hashCode`, ya que, presumiblemente, los implementadores de `String` habrán respetado la especificación de `hashCode`.

El programa de prueba de acompañamiento forma parte de una prueba de mayor tamaño que ilustra todos los contenedores básicos. Observe que si el `hashCode` no se ha implementado, se añadirán al `HashSet` los tres objetos `SimpleStudent`, porque no se detectará el duplicado.

Resulta que, como promedio, las operaciones `HashSet` puede realizarse en un tiempo constante. Esto parece un resultado bastante sorprendente, porque implica que el coste de una única operación `HashSet` no depende de si el `HashSet` contiene 10 elementos o 10.000. La teoría que subyace al concepto de `HashSet` es fascinante y se describe en el Capítulo 20.

6.8 Mapas

Un `Map` se utiliza para almacenar una colección de entradas compuestas de *claves* y sus correspondientes *valores*. El mapa asigna claves a valores.

Un `Map` se utiliza para almacenar una colección de entradas formadas por sus *claves* y sus *valores*. El `Map` asigna claves a valores. Las claves deben ser diferentes, pero pueden asignarse varias claves a un mismo valor. Por tanto, los que no necesitan ser diferentes son los valores. Existe una interfaz `SortedMap` que mantiene el mapa ordenado, desde el punto de vista lógico, según las claves.

No es sorprendente que existan dos implementaciones: `HashMap` y `TreeMap`. `HashMap` no mantiene las claves en orden, mientras que `TreeMap` sí que lo hace. Por simplicidad, no vamos a implementar la interfaz `SortedMap`, pero sí que implementaremos `HashMap` y `TreeMap`.

El `Map` puede implementarse como un `Set` instanciado con un *par* (véase la Sección 3.9), cuyo comparador o implementación `equals`/`hashCode` solo hace referencia a la clave. La interfaz `Map` no amplía `Collection`; por el contrario, es independiente. En las Figuras 6.36 y 6.37 se muestra una interfaz de ejemplo que contiene los métodos más importantes.

```
1  /**
2   * Programa de pruebas para HashSet.
3   */
4  class IteratorTest
5  {
6     public static void main( String [ ] args )
7     {
8        List<SimpleStudent> stud1 = new ArrayList<SimpleStudent>( );
9        stud1.add( new SimpleStudent( "Bob", 0 ) );
10       stud1.add( new SimpleStudent( "Joe", 1 ) );
11       stud1.add( new SimpleStudent( "Bob", 2 ) ); // duplicate
12
13          // Si hashCode está implementado, solo tendrá 2 elementos.
14          // En caso contrario, tendrá 3 porque
15          // no se detectará el duplicado.
16       Set<SimpleStudent> stud2 = new HashSet<SimpleStudent>( stud1 );
17
18       printCollection( stud1 ); // Bob Joe Bob (orden no especificado)
19       printCollection( stud2 ); // Dos elementos en orden no especificado
20    }
21 }
22
23 /**
24  * Ilustra el uso de hashCode/equals para una clase definida por el usuario.
25  * Los estudiantes se ordenan basándose solo en el nombre.
26  */
27 class SimpleStudent implements Comparable<SimpleStudent>
28 {
29    String name;
30    int id;
31
32    public SimpleStudent( String n, int i )
33       { name = n; id = i; }
34
35    public String toString( )
36       { return name + " " + id; }
37
38    public boolean equals( Object rhs )
39    {
40       if( rhs == null || getClass( ) != rhs.getClass( ) )
41          return false;
42
43       SimpleStudent other = (SimpleStudent) rhs;
44       return name.equals( other.name );
45    }
46
47    public int hashCode( )
48       { return name.hashCode( ); }
49 }
```

Figura 6.35 Ilustra los métodos `equals` y `hashCode` que se emplean en `HashSet`.

```
1  package weiss.util;
2
3  /**
4   * Interfaz Map.
5   * Un mapa almacena parejas clave/valor.
6   * En nuestra implementación, no están permitidas las claves duplicadas.
7   */
8  public interface Map<KeyType,ValueType> extends java.io.Serializable
9  {
10   /**
11    * Devuelve el número de claves en este mapa.
12    */
13   int size( );
14
15   /**
16    * Comprueba si este mapa está vacío.
17    */
18   boolean isEmpty( );
19
20   /**
21    * Comprueba si este mapa contiene una clave especificada.
22    */
23   boolean containsKey( KeyType key );
24
25   /**
26    * Devuelve el valor que se corresponde con la clave o null
27    * si no se encuentra la clave. Dado que están permitidos los valores
28    * null, comprobar si el valor de retorno es null puede no ser una forma
29    * segura de determinar si la clave está presente en el mapa.
30    */
31   ValueType get( KeyType key );
32
33   /**
34    * Añade la pareja clave/valor al mapa, sustituyendo el valor
35    * original en caso de que la clave ya estuviera presente.
36    * Devuelve el antiguo valor asociado con la clave o
37    * null si la clave  no estaba presente antes de esta llamada.
38 */
39 ValueType put( KeyType key, ValueType value );
40
41   /**
42    * Elimina la clave y su valor del mapa.
43    * Devuelve el valor previamente asociado con la clave
44    * o null si la clave no estaba presente antes de esta llamada.
45    */
46 ValueType remove( KeyType key );
```

Figura 6.36 Una interfaz `Map` de ejemplo (parte 1).

```
47    /**
48     * Elimina todas las parejas clave/valor del mapa.
49     */
50    void clear( );
51
52    /**
53     * Devuelve las claves del mapa.
54     */
55    Set<KeyType> keySet( );
56
57    /**
58     * Devuelve los valores del mapa. Puede haber duplicados.
59     */
60    Collection<ValueType> values( );
61
62    /**
63     * Devuelve un conjunto de objetos Map.Entry correspondientes
64     * a las parejas clave/valor contenidas en el mapa.
65     */
66    Set<Entry<KeyType,ValueType>> entrySet( );
67
68    /**
69     * Interfaz utilizada para acceder a las parejas clave/valor de un mapa.
70     * A partir de un mapa, utilice entrySet().iterator para obtener un
71     * iterador sobre un Set de parejas. El método next() de este iterador
72     * proporciona objetos de tipo Map.Entry<KeyType,ValueType>.
73     */
74    public interface Entry<KeyType,ValueType> extends java.io.Serializable
75    {
76       /**
77        * Devuelve la clave de esta pareja.
78        */
79       KeyType getKey( );
80
81       /**
82        * Devuelve el valor de esta pareja.
83     */
84       ValueType getValue( );
85
86       /**
87        * Cambia el valor de esta pareja.
88        * @return el valor antiguo asociado con esta pareja.
89        */
90       ValueType setValue( ValueType newValue );
91    }
92 }
```

Figura 6.37 Una interfaz `Map` de ejemplo (parte 2).

La mayoría de los métodos tienen un semántica intuitiva. El método `put` se utiliza para añadir un pareja clave/valor, `remove` se emplea para eliminar una pareja clave/valor (solo se especifica la clave) y `get` devuelve el valor asociado con una clave. Están permitidos valores `null`, lo que complica las cosas para `get`, ya que el valor de retorno proporcionado por `get` no permitirá distinguir entre una búsqueda fallida y una búsqueda que haya tenido éxito y que devuelva `null` como valor. Si se sabe que existen valores `null` en el mapa, puede emplearse `containsKey`.

La interfaz `Map` no proporciona un método `iterator` ni una clase iteradora. En lugar de ello, devuelve una `Collection` que puede utilizarse para visualizar los contenidos del mapa.

El método `keySet` proporciona una `Collection` que contiene todas las claves. Puesto que no están permitidas las claves duplicadas, el resultado de `keySet` es un `Set`, para el cual podemos obtener un iterador. Si el `Map` es un `SortedMap`, el `Set` es un `SortedSet`.

De forma similar, el método `values` devuelve una `Collection` que contiene todos los valores. En este caso se trata realmente de una `Collection`, ya que los valores duplicados están permitidos.

`Map.Entry` abstrae la noción de una pareja dentro del mapa.

Finalmente, el método `entrySet` devuelve una `Collection` de parejas clave/valor. De nuevo, se trata de un `Set`, porque las parejas deben tener claves distintas. Los objetos del `Set` devueltos por el `entrySet` son parejas; debe haber un tipo que represente a esas parejas clave/valor. Este tipo es especificado por la interfaz `Entry` anidada dentro de la interfaz `Map`. Por tanto, el tipo de objeto almacenado en el `entrySet` es `Map.Entry`.

La Figura 6.38 ilustra el uso del `Map` con `TreeMap`. En la línea 23 se crea un mapa vacío y luego se rellena con una serie de llamadas a `put` en las líneas 25 a 29. La última llamada a `put` simplemente sustituye un valor con `"unlisted"`. Las líneas 31 y 32 imprimen el resultado de una llamada a `get`, que se utiliza para obtener el valor correspondiente a la clave `"Jane Doe"`. Más interesante es la rutina `printMap` que abarca las líneas 8 a 19.

En `printMap`, en la línea 12, obtenemos un `Set` que contiene parejas `Map.Entry`. A partir del `Set`, podemos utilizar un bucle `for` avanzado para visualizar las entradas `Map.Entry` y podemos obtener la información de clave y valor utilizando `getKey` y `getValue`, como se muestra en las líneas 16 y 17.

`keySet`, `values` y `entrySet` devuelven vistas.

Volviendo a `main`, vemos que `keySet` devuelve un conjunto de claves (en la línea 37) que pueden imprimirse en la línea 38 llamando a `printCollection` (en la Figura 6.11); de forma similar, en las líneas 41 y 42, `values` devuelve una colección de valores que se puede imprimir. Más interesante resulta el que el conjunto de claves y la colección de valores son *vistas* del mapa, por lo que los cambios en el mapa se ven inmediatamente reflejados en el conjunto de claves y la colección de valores, y las eliminaciones que efectuemos en el conjunto de claves o en el de valores se convierten en eliminaciones en el mapa subyacente. Por tanto, la línea 44 no solo elimina la clave del conjunto de claves, sino también la entrada asociada en el mapa. De forma similar, la línea 45 elimina una entrada del mapa. Por tanto, la operación de impresión en la línea 49 refleja un mapa en el que se han eliminado dos entradas.

Las vistas en sí mismas constituyen un concepto interesante y explicaremos los detalles específicos acerca de cómo se implementan más adelante, cuando implementemos las clases de mapas. En la Sección 6.10 se exponen algunos ejemplos adicionales de vistas.

La Figura 6.39 ilustra otro uso del mapa, en un método que devuelve los elementos de una lista que aparecen más de una vez. En este código, se está utilizando internamente un mapa para agrupar

```
1  import java.util.Map;
2  import java.util.TreeMap;
3  import java.util.Set;
4  import java.util.Collection;
5
6  public class MapDemo
7  {
8     public static <KeyType,ValueType>
9     void printMap( String msg, Map<KeyType,ValueType> m )
10    {
11       System.out.println( msg + ":" );
12       Set<Map.Entry<KeyType,ValueType>> entries = m.entrySet( );
13
14       for( Map.Entry<KeyType,ValueType> thisPair : entries )
15       {
16          System.out.print( thisPair.getKey( ) + ": " );
17       System.out.println( thisPair.getValue( ) );
18       }
19    }
20
21    public static void main( String [ ] args )
22    {
23       Map<String,String> phone1 = new TreeMap<String,String>( );
24
25       phone1.put( "John Doe", "212-555-1212" );
26       phone1.put( "Jane Doe", "312-555-1212" );
27       phone1.put( "Holly Doe", "213-555-1212" );
28       phone1.put( "Susan Doe", "617-555-1212" );
29       phone1.put( "Jane Doe", "unlisted" );
30
31       System.out.println( "phone1.get(\"Jane Doe\"): " +
32                           phone1.get( "Jane Doe" ) );
33       System.out.println( "\nThe map is: " );
34       printMap( "phone1", phone1 );
35
36       System.out.println( "\nThe keys are: " );
37       Set<String> keys = phone1.keySet( );
38       printCollection( keys );
39
40       System.out.println( "\nThe values are: " );
41       Collection<String> values = phone1.values( );
42       printCollection( values );
43
44       keys.remove( "John Doe" );
45       values.remove( "unlisted" );
46
47       System.out.println( "After John Doe and 1 unlisted are removed" );
48       System.out.println( "\nThe map is: " );
49       printMap( "phone1", phone1 );
50    }
51 }
```

Figura 6.38 Una ilustración de cómo se utiliza la interfaz `Map`.

```
1  public static List<String> listDuplicates( List<String> coll )
2  {
3     Map<String,Integer> count = new TreeMap<String,Integer>( );
4     List<String> result = new ArrayList<String>( );
5
6     for( String word : coll )
7     {
8        Integer occurs = count.get( word );
9        if( occurs == null )
10          count.put( word, 1 );
11       else
12          count.put( word, occurs + 1 );
13    }
14
15    for( Map.Entry<String,Integer> e : count.entrySet( ) )
16       if( e.getValue( ) >= 2 )
17          result.add( e.getKey( ) );
18
19    return result;
20 }
```

Figura 6.39 Un uso típico de un mapa.

los duplicados: la clave del mapa es un elemento y el valor es el número de veces que el elemento ha aparecido. Las líneas 8-12 ilustran la idea típica que podemos ver a la hora de construir un mapa de esta forma. Si el elemento nunca ha sido insertado en el mapa, lo hacemos con un recuento igual a 1. En caso contrario, actualizamos el recuento. Observe el juicioso uso de los mecanismos de *autoboxing* y *unboxing*. Después, en las líneas 15-17 utilizamos un iterador para recorrer el conjunto de entradas, obteniendo las claves que aparezcan con un recuento mayor o igual que dos en el mapa.

6.9 Colas con prioridad

La cola con prioridad solo permite acceder al elemento mínimo.

Aunque los trabajos de impresión enviados a una impresora suelen colocarse en una cola, esa política puede no ser siempre la más adecuada. Por ejemplo, un trabajo de impresión podría ser particularmente importante, por lo que desearíamos poder permitir que ese trabajo se imprima en cuanto la impresora esté disponible. A la inversa, cuando la impresora finalice un trabajo y haya varios trabajos de 1 página y uno de 100 páginas esperando, puede ser razonable imprimir el último trabajo al final, aun cuando no sea el último trabajo enviado. (Lamentablemente, la mayoría de los sistemas no hacen esto, lo cual puede resultar particularmente molesto en ciertas ocasiones.)

De forma similar, en un entorno multiusuario, el planificador del sistema operativo debe decidir cuál de entre varios procesos ejecutar. En general, a cada proceso solo se le permite ejecutarse durante un periodo de tiempo fijo. Un algoritmo no muy adecuado para implementar este tipo de