CAPÍTULO

13 Simulación

Un uso importante del computador es el de la *simulación*. En una simulación, el computador emula el funcionamiento de un sistema real y recoge estadísticas. Como ejemplo, puede que queramos simular el funcionamiento de un banco con k ventanillas para determinar cuál es el valor mínimo de k con el que se consigue un tiempo de servicio razonable. Usar un computador proporcionaría grandes ventajas. En primer lugar, la información se recogería sin molestar a los clientes. En segundo lugar, una simulación por computador puede ser mucho más rápida que la implementación real debido a la velocidad del mismo. En tercer lugar, se podría replicar fácilmente la simulación. En muchos casos, la elección de una estructura de datos apropiada nos puede ayudar a mejorar la eficiencia de la simulación.

En este capítulo veremos:

- Cómo simular un juego modelado sobre el *problema Josephus*.
- Cómo simular el comportamiento de un banco de módems.

Un uso importante de los computadores es el de la *simulación*. En una simulación, el computador emula el funcionamiento de un sistema real y recoge estadísticas.

13.1 El problema Josephus

El *problema Josephus* es el siguiente juego. Se sientan N personas, numeradas de 1 a N, formando un círculo. Se pasa una patata caliente empezando en la persona 1. Después de pasar M veces la patata, se elimina a la persona que tiene la patata, el círculo se estrecha y el juego continúa, tomando la patata la persona sentada después de la que ha sido eliminada. La persona que se mantiene hasta el final gana. Es bastante habitual considerar que M es una constante del juego, aunque se puede usar un generador de números aleatorios para cambiar M tras cada eliminación.

En el *problema Josephus*, se pasa repetidamente una patata caliente. Cuando se termina de pasar, el jugador que tiene la patata queda eliminado. El juego continúa, y gana el jugador que queda al final.

El problema Josephus surgió en el primer siglo A.C. en una cueva de una montaña en Israel, donde los celotes judíos estaban siendo asediados por soldados romanos. El historiador Josephus se encontraba entre ellos. Para consternación de Josephus, los celotes votaron hacer un pacto de suicidio en lugar de rendirse a los romanos. Él sugirió el juego aquí mencionado. La patata caliente era la sentencia de muerte para la persona sentada junto a la que tenía la patata. Josephus amañó el juego para quedar el último y convenció a la penúltima víctima de que debían ren-

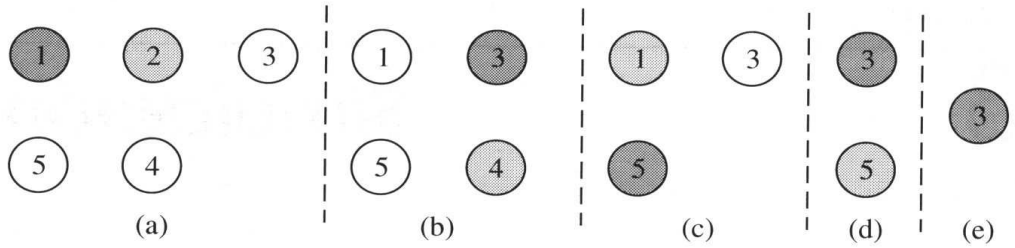


Figura 13.1 El problema Josephus. En cada paso el círculo más oscuro representa a la persona que tiene la patata inicialmente, y el ligeramente sombreado representa al jugador que recibe la patata caliente (y que por tanto es eliminado). Los pases se realizan en el sentido de las agujas del reloj.

dirse. Gracias a ello hemos podido saber de este juego; en efecto, Josephus hizo trampas[1].

Si $M = 0$, entonces se van eliminando los jugadores en orden y el último jugador siempre gana. Para otros valores de M, las cosas no son tan obvias. La Figura 13.1 muestra que si $N = 5$ y $M = 1$, entonces los jugadores son eliminados en el siguiente orden: 2, 4, 1, 5. En este caso gana el jugador 3. Los pasos son los siguientes:

1. Al principio la patata está en manos del jugador número 1. Después de pasarla una vez, la tiene el jugador número 2.
2. El jugador 2 es eliminado. El jugador 3 toma la patata y se la pasa al jugador 4.
3. El jugador 4 es eliminado. El jugador 5 toma la patata y se la pasa al jugador 1.
4. El jugador 1 es eliminado. El jugador 3 toma la patata y se la pasa al jugador 5.
5. El jugador 5 es eliminado, y por tanto el jugador 3 gana.

En primer lugar escribiremos un programa que simule, paso a paso, el juego para cualesquiera valores de N y M. El tiempo de ejecución de la simulación es $O(MN)$, lo cual es aceptable si el número de veces que se pasa la patata es pequeño. Cada paso lleva un tiempo $O(M)$ porque en él se llevan a cabo M pases. Mostraremos entonces cómo implementar cada paso en un tiempo $O(\log N)$, independientemente del número de pases que se realizan. El tiempo de la simulación es entonces $O(N \log N)$.

13.1.1 La solución simple

Podemos representar los jugadores mediante una lista enlazada y usar el iterador para simular los pases.

La fase de pasar la patata en el problema Josephus sugiere que representemos los jugadores mediante una lista enlazada. Creamos una lista enlazada en la que los elementos 1, 2, ..., N son insertados en orden. Inicializamos entonces un iterador con el primer elemento. Cada pase de la patata corresponde a una operación `avanzar` sobre el iterador. Cuando llegamos al último jugador de los que queden

[1] Agradezco a David Teague el relato de esta historia. La versión que nosotros resolvemos difiere de la explicación histórica. En el Ejercicio 13.11 se le pide que resuelva la versión histórica.

en la lista, implementamos el pase reiniciando el iterador con el primer elemento. Esta acción imita un círculo. Cuando hemos terminado de pasar, eliminamos el elemento sobre el que ha aterrizado el iterador.

Por razones explicadas en el Capítulo 16, es más fácil eliminar el elemento posterior a la posición actual usando el método `eliminarSig`. Esto se refleja en el código de la Figura 13.2, donde inicializamos `actual` con el último jugador, en lugar del primero. En las líneas 7 y 8 se declaran la lista enlazada anónima y su iterador. Usando un bucle construimos la lista inicial en las líneas 12 y 13. La semántica de `insertar`, tal y como se especifica en la clase de la Figura 16.7, nos dice que las inserciones se realizan tras la posición del iterador y que el resultado de una inserción coloca el iterador en la posición recién insertada. De esta forma obtenemos la lista deseada.

Mantenemos el iterador en el jugador anterior al actual, para que se pueda aplicar `eliminarSig`.

```
1  /**
2   * Devuelve el ganador del problema Josephus.
3   * Implementación mediante una lista enlazada.
4   */
5  public static int josephus( int personas, int pases )
6  {
7      ListaEnlazadaIter p = new ListaEnlazadaIter (
8                                          new ListaEnlazada( ) );
9          // Construye la lista
10     try
11     {
12         for( int i = 1; i <= personas; i++ )
13            p.insertar( new Integer( i ) );
14     }
15     catch( ElementoNoEncontrado e ) { } // No puede ocurrir
16
17         // Juega al juego;
18         // Observe: p está siempre en el jugador anterior
19     while( personas-- != 1 )
20     {
21         for( int i = 0; i < pases; i++ )
22         {
23            p.avanzar( );          // Avanzar
24            if( !p.estaDentro( ) ) // Si nos pasamos del último
25               p.primero( );       // jugador volvemos al primero
26         }
27
28         if( !p.eliminarSig( ) )   // Elimina el siguiente jugador
29         {
30            // eliminarSig falla si p es el último elemento, luego
31            p.cero( );       // para el último elemento, asigna a p
32            p.eliminarSig( );// el elemento 0 para eliminar el 1
33         }
34     }
35         // Obtiene el primer y único jugador y devuelve el # de
              jugadores
36     p.primero( );
37     return ( (Integer)( p.recuperar( ) ) ).intValue( );
38 }
```

Figura 13.2 Implementación del problema Josephus usando listas enlazadas.

En la Figura 13.2, el código de las líneas 21 a 33 lleva a cabo un paso del algoritmo pasando la patata (líneas 21 a 26) y eliminando después un jugador (líneas 28 a 33). Esto se repite hasta que el test de la línea 19 nos dice que solamente queda un jugador. En ese momento, obtenemos el jugador restante llamando al método `primero` en la línea 36. (Recuerde que durante la mayor parte del algoritmo, el iterador se encuentra en el jugador anterior al actual). En la línea 37 accedemos al jugador y devolvemos su número.

El tiempo de ejecución es $O(MN)$.

Es fácil ver que el tiempo de ejecución de esta rutina es $O(MN)$ porque éste es exactamente el número de pases que se producen durante el algoritmo, lo cual es aceptable para M pequeño. Sin embargo, observe que en el caso en que $M = 0$, el tiempo de ejecución no es $O(0)$, sino $O(N)$, ya que cuando se interpreta una expresión O no basta con multiplicar.

13.1.2 Un algoritmo más eficiente

Si implementamos cada ronda de pases con una única operación logarítmica, la simulación será más rápida.

Se puede obtener un algoritmo más eficiente si usamos una estructura de datos que soporte el acceso al k-ésimo elemento más pequeño (en tiempo logarítmico). Esto nos permite implementar cada ronda de pases con una única operación. La Figura 13.1 muestra por qué. Supongamos que quedan N jugadores y que estamos actualmente en el jugador P empezando por el principio. Inicialmente N es el número total de jugadores y P es 1. Después de M pases, estaremos en el jugador $((M + P) \bmod N)$, excepto si tenemos que devolver el jugador 0, en cuyo caso vamos al jugador N. El cálculo es algo rebuscado, pero el concepto no lo es.

El cálculo es rebuscado debido a la circularidad.

Aplicando este cálculo a la Figura 13.1, vemos que como M es 1, N es inicialmente 5, y P es inicialmente 1, el nuevo valor de P es 2. Tras la eliminación, N se reduce a 4, pero aún estamos en la posición 2 (como la parte (b) de la figura sugiere). El siguiente valor de P es 3 (como se muestra en la parte (b)), luego se elimina el tercer elemento de la lista y N se reduce a 3. El siguiente valor de P es 4 mod 3, es decir 1, con lo que volvemos al primer jugador de la lista (como se muestra en la parte (c)). Se elimina este jugador y N pasa a valer 2. Puesto que 2 mod 2 es 0, hacemos que P valga N, y por tanto el último jugador de la lista es el eliminado, lo cual concuerda con la parte (d). Tras la eliminación, N es 1 y hemos terminado.

Un árbol de búsqueda puede soportar una rutina `buscarKesimo`.

Todo lo que necesitamos es una estructura que soporte de forma eficiente la operación `buscarKesimo` y un método para insertar secuencialmente los jugadores en la estructura de datos. Hay varias alternativas parecidas. Todas ellas usan el hecho de que un árbol binario de búsqueda soporta la operación `buscarKesimo` en tiempo medio logarítmico o si usamos un árbol binario de búsqueda sofisticado en un tiempo logarítmico en el caso peor. En consecuencia, si somos cuidadosos podemos esperar un algoritmo $O(N \log N)$.

El método más simple consiste en insertar los elementos secuencialmente en un árbol binario de búsqueda eficiente en el peor de los casos, como un árbol rojinegro, un AA-árbol o un árbol de ensanchamiento (todos estos árboles se estudian en capítulos posteriores). Entonces cuando sea oportuno podemos llamar a `buscarKesimo` y a `eliminar`. Resulta que los árboles de ensanchamiento son una elección excelente para esta aplicación, porque las operaciones `buscarKesimo` e `insertar` son inusualmente eficientes y `eliminar` no es demasiado difícil de codificar. No obstante aquí usaremos una estructura alternativa, porque la imple-

mentación de esta estructura de datos, que se proporciona en capítulos posteriores, deja como ejercicio la implementación de `buscarKesimo`.

Usamos la clase `ABBConRango` que soporta la operación `buscarKesimo` y que se implementa completamente en la Sección 18.2. Se basa en el árbol binario de búsqueda ordinario y por tanto no tiene eficiencia logarítmica en el caso peor sino solamente en media. En consecuencia, no podemos limitarnos a insertar los elementos secuencialmente; esto provocaría que el árbol de búsqueda exhibiera su peor comportamiento.

Un árbol de búsqueda equilibrado funcionará, pero no es imprescindible si somos cuidadosos y construimos al principio un árbol binario de búsqueda ordinario que no esté desequilibrado. Se puede definir un método de clase que construya un árbol perfectamente equilibrado en un tiempo lineal.

```
1  /**
2   * Construye recursivamente un árbol perfectamente equilibrado
3   * en tiempo O( N log N ) mediante repetidas inserciones.
4   * En la llamada inicial a debería estar vacío.
5   */
6  public static void construirArbol( ABBConRango a,
7                                     int inf, int sup )
8  {
9      int centro = ( inf + sup ) / 2;
10
11     if( inf <= sup )
12     {
13         try
14           { a.insertar( new MiEntero( centro ) ); }
15         catch( ElementoDuplicado e ) { } // No puede suceder
16
17         construirArbol( a, inf, centro - 1 );
18         construirArbol( a, centro + 1, sup );
19     }
20 }
21
22 /**
23  * Devuelve el ganador del problema Josephus.
24  * Implementación usando un árbol de búsqueda.
25  */
26 public static int josephus( int personas, int pases )
27 {
28     ABBConRango a = new ABBConRango( );
29     try
30     {
31         construirArbol( a, 1, personas );
32
33         int rango = 1;
34         while( personas > 1 )
35         {
36             rango = ( rango + pases ) % personas;
37             if( rango == 0 )
38                 rango = personas;
39
40             a.eliminar( a.buscarKesimo( rango ) );
41             personas--;
42         }
43
44         return ( ( MiEntero ) ( a.buscarKesimo( 1 ) ) ).intValue( );
45     }
46     catch( Exception e )
47       { return -1; } // No puede suceder
48 }
```

Figura 13.3 Solución $O(N \log N)$ para el problema Josephus.

Hay varias opciones, una de las cuales consiste en insertar una permutación aleatoria de 1, ..., N en el árbol de búsqueda. Otra, consiste en construir un árbol binario de búsqueda perfectamente equilibrado usando un método de la clase. Puesto que un método de la clase tendría acceso al manejo interno del árbol de búsqueda, se podría hacer en tiempo lineal. Se deja escribir esta rutina como el Ejercicio 18.19, una vez se hayan estudiado los árboles.

Mediante inserciones recursivas construiríamos el mismo árbol pero requiriendo un tiempo $O(N \log N)$.

El método que utilizamos consiste en escribir una rutina que inserte elementos en orden equilibrado. Insertando el elemento central en la raíz y construyendo recursivamente los dos subárboles de la misma manera, obtenemos un árbol equilibrado. El coste de nuestra rutina es un aceptable $O(N \log N)$. Aunque no es tan eficiente como lo sería la rutina lineal de la clase, no afecta de forma adversa al tiempo de ejecución asintótico del algoritmo completo. Se garantiza que las operaciones `eliminar` son logarítmicas. Esta rutina se llama `construirArbol`, y su código aparece en la Figura 13.3, junto con el método `josephus`.

13.2 Simulación dirigida por eventos

Pasamos ahora al problema de la simulación de un banco descrito en la introducción. Tenemos un sistema en el que los clientes llegan y esperan en una cola hasta que una de las ventanillas está libre. La llegada de los clientes está gobernada por una función de distribución de probabilidades, al igual que el tiempo de servicio al cliente (cantidad de tiempo durante el que es atendido una vez que una ventanilla está libre). Estamos interesados en estadísticas como cuál es el tiempo medio que tiene que esperar un cliente en la cola y qué porcentaje del tiempo dedica cada empleado a atender peticiones. (Si hay demasiadas ventanillas, algunos no harán nada durante largos períodos.)

Para ciertas distribuciones de probabilidad y valores de k, se pueden calcular estos valores estadísticos de forma exacta. Pero a medida que k se hace más grande el análisis se hace considerablemente más difícil, por lo que resulta atractivo usar un computador para simular el funcionamiento del banco. De esta forma, los directores del banco pueden decidir cuántas ventanillas se necesitan para asegurar un servicio razonablemente desahogado. La mayor parte de las simulaciones implican conocimientos de probabilidad, estadística y teoría de colas.

13.2.1 Ideas básicas

Una simulación de eventos discretos consiste en el procesamiento de eventos. Aquí los dos eventos a considerar son (a) la llegada de un cliente y (b) la salida de un cliente, dejando libre una ventanilla.

Un *tic* es la unidad de tiempo en una simulación.

Podemos usar una función de probabilidad para generar una secuencia de entrada formada por pares ordenados de tiempos de llegada y servicio de cada cliente, ordenados por tiempo de llegada[2]. No necesitamos usar la hora exacta del día, sino que usamos una unidad a la que nos referimos como *tic*.

[2] La función de probabilidad genera intervalos de tiempo entre llegadas, garantizando que éstas se producen en orden cronológico.

Inicializaremos el tiempo de la simulación en cero tics. Entonces avanzamos el reloj un tic cada vez, comprobando si se produce un evento. Si se produce, procesamos el evento (o eventos) y recalculamos los valores estadísticos. Cuando ya no quedan clientes en la secuencia de entrada y todas las ventanillas están libres, la simulación ha terminado. Ésta es una *simulación dirigida por tiempos discretos*.

Una *simulación dirigida por tiempos discretos* procesa cada unidad de tiempo consecutivamente. Resulta inapropiada si es grande el intervalo entre eventos consecutivos.

El problema de esta estrategia de simulación es que su tiempo de ejecución no depende del número de clientes o de eventos (en este caso hay dos eventos por cliente), sino del número de tics, que en realidad no es parte de la entrada. Para ver por qué esto es importante, supongamos que cambiamos las unidades del reloj a microtics y que multiplicamos todos los tiempos de la entrada por 1 millón. El resultado sería que la simulación tardaría 1 millón de veces más en ejecutarse.

La clave para evitar este problema es ir avanzando el reloj en cada paso hasta el momento en el que se produce el siguiente evento. Luego tenemos una *simulación dirigida por eventos*. Esto es conceptualmente fácil de hacer. En cualquier punto, el siguiente evento que puede producirse es o bien la llegada del siguiente cliente, o bien la salida de uno de los clientes de una ventanilla. Puesto que los tiempos en los que se producen eventos están disponibles, solamente necesitamos encontrar el evento más cercano en el tiempo y procesar dicho evento (actualizando el reloj con el tiempo en el que se produce el evento).

Una *simulación dirigida por eventos* avanza en cada paso el reloj hasta el siguiente evento.

Si el evento es una salida, el procesamiento incluye la actualización de los valores estadísticos con los datos del cliente que se marcha y mirar la cola para ver si hay otro cliente esperando. Si es así, añadimos ese cliente, recalculamos los valores estadísticos que proceda, calculamos la hora en que se marchará el cliente y añadimos su salida al conjunto de eventos a la espera de producirse.

Si el evento es una llegada, comprobamos si hay una ventanilla libre. Si no hay ninguna colocamos al cliente en la cola. En caso contrario, le asignamos al cliente una ventanilla, calculamos el tiempo de salida del cliente, y añadimos dicha salida al conjunto de eventos a la espera de producirse.

La cola de clientes se puede implementar como una cola. Puesto que necesitamos encontrar el evento más cercano en el tiempo, el conjunto de eventos debe organizarse como una cola de prioridad. El siguiente evento es o bien una llegada o una salida (lo que ocurra antes); ambos tipos de eventos están igualmente disponibles. Una simulación dirigida por eventos es apropiada si se espera que el número de tics entre eventos sea grande.

El conjunto de eventos (es decir, los eventos a la espera de producirse) se organiza como una cola de prioridad.

13.2.2 Ejemplo: simulación de un banco de módems

El punto crucial en la gestión de una simulación es la organización de los eventos en una cola de prioridad. Para centrarnos en este punto, consideraremos una simulación de un sistema muy simple: un *banco de módems* del centro de computación de una universidad.

Un banco de módems consiste en una gran colección de módems. Por ejemplo, la Universidad Internacional de Florida (FIU) tiene 96 módems disponibles para los estudiantes. A un módem se accede marcando un número de teléfono. Si alguno de los 96 módems está disponible, entonces el usuario se conecta a uno de

ellos, pero si se están usando todos, entonces el teléfono dará la señal de ocupado. Nuestra simulación modeliza el servicio proporcionado por el banco de módems. Las variables son las siguientes:

- El número de módems en el banco.
- La distribución de probabilidad que gobierna los intentos de marcado.
- La distribución de probabilidad que gobierna el tiempo de conexión.
- El tiempo (real) que cubrirá la simulación.

El *banco de módems* elimina la cola de espera de la simulación, con lo que solamente se precisa una estructura de datos.

La simulación del banco de módems es una versión simplificada de la simulación de los cajeros del banco porque no hay cola de espera. Cada marcado es una llegada, y el tiempo de conexión es el tiempo de servicio. Eliminando la cola de espera, eliminamos la necesidad de mantener una estructura cola, por lo que solamente necesitamos una estructura de datos, la cola de prioridad. En el Ejercicio 13.17 se le pide incorporar una cola, de forma que si todos los módems están ocupados se irán colocando en la cola hasta L llamadas. Para simplificar las cosas, no calculamos valores estadísticos, sino que en su lugar listamos cada evento a medida que se producen. También suponemos que los intentos de conexión suceden a intervalos constantes, mientras que en una buena simulación sería necesario modelizar el tiempo entre llegadas mediante un proceso aleatorio. En la Figura 13.4 se muestra la salida de una simulación.

Listamos cada evento a medida que sucede; recoger valores estadísticos es una extensión simple.

```
 1 El usuario 0 marca en minuto 0 y se conecta durante 1 minutos
 2 El usuario 0 cuelga en minuto 1
 3 El usuario 1 marca en minuto 1 y se conecta durante 5 minutos
 4 El usuario 2 marca en minuto 2 y se conecta durante 4 minutos
 5 El usuario 3 marca en minuto 3 y se conecta durante 11 minutos
 6 El usuario 4 marca en minuto 4 pero recibe señal de ocupado
 7 El usuario 5 marca en minuto 5 pero recibe señal de ocupado
 8 El usuario 6 marca en minuto 6 pero recibe señal de ocupado
 9 El usuario 1 cuelga en minuto 6
10 El usuario 2 cuelga en minuto 6
11 El usuario 7 marca en minuto 7 y se conecta durante 8 minutos
12 El usuario 8 marca en minuto 8 y se conecta durante 6 minutos
13 El usuario 9 marca en minuto 9 pero recibe señal de ocupado
14 El usuario 10 marca en minuto 10 pero recibe señal de ocupado
15 El usuario 11 marca en minuto 11 pero recibe señal de ocupado
16 El usuario 12 marca en minuto 12 pero recibe señal de ocupado
17 El usuario 13 marca en minuto 13 pero recibe señal de ocupado
18 El usuario 3 cuelga en minuto 14
19 El usuario 14 marca en minuto 14 y se conecta durante 6 minutos
20 El usuario 8 cuelga en minuto 14
21 El usuario 15 marca en minuto 15 y se conecta durante 3 minutos
22 El usuario 7 cuelga en minuto 15
23 El usuario 16 marca en minuto 16 y se conecta durante 5 minutos
24 El usuario 17 marca en minuto 17 pero recibe señal de ocupado
25 El usuario 15 cuelga en minuto 18
26 El usuario 18 marca en minuto 18 y se conecta durante 7 minutos
27 El usuario 19 marca en minuto 19 pero recibe señal de ocupado
```

Figura 13.4 Ejemplo de salida para la simulación del banco de módems: tenemos 3 módems; se intenta un marcado cada minuto; el tiempo medio de conexión es de 5 minutos; la simulación se ejecuta durante 19 minutos.

La clase de la simulación requiere otra clase que representa los eventos, la cual recibe el nombre de clase `Evento` y se muestra en la Figura 13.5. Los atributos son el número de cliente, el instante en que sucederá el evento y una indicación del tipo del evento (`MARCADO` o `COLGADO`). Si esta simulación fuera más compleja, con varios tipos de eventos, con seguridad convertiríamos a `Evento` en una clase base y derivaríamos subclases de ella. No lo hacemos aquí para no complicar las cosas ni oscurecer el funcionamiento básico del algoritmo de simulación. La clase `Evento` contiene constructores e implementa la interfaz de la clase `Comparable`. Observe que tiene un constructor de cero parámetros de forma que se puede crear un centinela $-\infty$ para una cola de prioridad que almacena una colección de objetos de la clase `Evento`. La clase `Evento` usa atributos amistosos para las clases dentro del mismo paquete.

La clase `Evento` representa los eventos. En una simulación compleja, se derivarían todos los posible tipos de eventos como subclases. Usar la herencia en la clase `Evento` complicaría el código.

La clase para la simulación de los módems, `SimModem`, se muestra en la Figura 13.6. Consta de un conjunto de atributos, un constructor y dos métodos. Los atributos incluyen un número aleatorio `r` mostrado en la línea 25. En la línea 26, `conjEventos` se declara como una `ColaPrioridad` de objetos `Evento`. Hay tres atributos más. Uno de ellos es `modemsLibres`, que inicialmente es el número de módems en la simulación, pero que cambia a medida que los usuarios se conectan y cuelgan. Los otros dos son `tpoMedioLlamada` y `frecLlamadas`, que son parámetros de la simulación. Recuerde que se llevará a cabo un intento de marcado cada `frecLlamadas` tics. El constructor en las líneas 18 y 19, que se implementa en la Figura 13.7, inicializa estos atributos y coloca la primera llegada en la cola de prioridad `conjEventos`.

```
1  import Soporte.*; import Soporte.Comparable;
2  /**
3   * La clase de los eventos.
4   * Implementa el interfaz Comparable
5   * para ordenar los eventos por el instante en que se producen.
6   */
7  class Evento implements Comparable
8  {
9      static final int MARCADO = 1;
10     static final int COLGADO = 2;
11
12     public Evento( )
13       { this( 0, 0, MARCADO ); }
14
15     public Evento( int nombre, long tpo, int tipo )
16       { quien = nombre; tiempo = tpo; que = tipo; }
17
18     public boolean menorQue( Comparable lder )
19       { return tiempo < ( (Evento) lder ).tiempo; }
20
21     public int compara( Comparable lder )
22     {
23         return menorQue( lder ) ? -1 :
24                lder.menorQue( tnis ) ? 1 : 0;
25     }
26
27     int quien;     // Número del usuario
28     long tiempo;   // Cuándo sucederá el evento
29     int que;       // MARCADO o COLGADO
30 }
```

Figura 13.5 Clase `Evento` usada en la simulación del banco de módems.

```
1  import Excepciones.*;
2  import java.util.*;
3  import EstructurasDatos.*;
4  // Clase SimModem: lleva a cabo una simulación
5  //
6  // CONSTRUCTOR: con tres parámetros: número de módems,
7  //      tiempo medio de conexión y tiempo entre
8  //      llegadas
9  //
10 // **********************MÉTODOS PÚBLICOS**********************
11 // void ejecSim( )        --> Ejecuta una simulación
12
13 /**
14  * Clase SimModem.
15  */
16 public class SimModem
17 {
18     public SimModem( int modems, double tpoMedio,
19                      long intervalo )
20       { /* Figura 13.7 */ }
21
22     public void ejecSim( long tiempoParada )
23       { /* Figura 13.9 */ }
24
25     private Random r;                    // Número aleatorio
26     private ColaPrioridad conjEventos;// Eventos pendientes
27
28        // Parámetros básicos de la simulación
29     private int modemsLibres;            // Número de módems sin usar
30     private double tpoMedioLlamada;      // Longitud de una llamada
31     private long frecLlamadas;           // Intervalo entre llamadas
32
33     private void sigLlamada( long delta )
34       { /* Figura 13.8 */ }
35 }
```

Figura 13.6 Esqueleto de la clase `SimModem`.

```
1  /**
2   * Constructor.
3   * @param modems número de módems.
4   * @param tpoMedio duración media de una llamada.
5   * @param intervalo tiempo medio entre llamadas.
6   */
7  public SimModem( int modems, double tpoMedio,
8                   long intervalo )
9  {
10     conjEventos      = new MonticuloBinario( new Evento( ) );
11     modemsLibres     = modems;
12     tpoMedioLlamada  = tpoMedio;
13     frecLlamadas     = intervalo;
14     r                = new Random( );
15     sigLlamada( frecLlamadas ); // Planifica la primera llamada
16 }
```

Figura 13.7 Constructor `SimModem`.

La rutina `sigLlamada` añade un `MARCADO` al conjunto de eventos.

La clase para la simulación consta solamente de dos métodos. El primero, `sigLlamada`, mostrado en la Figura 13.8, añade una petición de `MARCADO` al conjunto de eventos. Mantiene dos variables de clase (globales): el número del si-

```
1       // Usado solamente por sigLlamada
2   private int numUsuario = 0;
3   private long tpoSigLlamada = 0;
4
5   /**
6    * Coloca un nuevo evento MARCADO en la cola de eventos.
7    * Luego avanza el reloj hasta que sucede el siguiente
8    * evento MARCADO. En la práctica, usaríamos un número
9    * aleatorio para determinar el tiempo.
10   */
11  private void sigLlamada( long delta )
12  {
13      conjEventos.insertar( new Evento( numUsuario++, tpoSigLlamada,
14                                        Evento.MARCADO ) );
15      tpoSigLlamada += delta;
16  }
```

Figura 13.8 Rutina `sigLlamada`, que coloca un nuevo evento `MARCADO` en la cola de eventos y avanza el reloj hasta el momento en el que se produce el siguiente evento `MARCADO`.

guiente usuario que intentará marcar y el tiempo en el que se producirá el evento. De nuevo, suponemos, para simplificar, que las llamadas se producen a intervalos regulares. En la práctica, usaríamos un generador de números aleatorios para modelizar la secuencia de marcados.

La rutina `ejecSim` realiza la simulación.

El otro método es `ejecSim`, mostrado en la Figura 13.9, al que se llama para ejecutar la simulación completa. Es la rutina que hace casi todo el trabajo. Se llama con un único parámetro que indica la duración de la simulación. Mientras el conjunto de eventos no sea vacío, procesamos eventos. Observe que ello no debería suceder nunca, porque cuando llegamos a la línea 13, hay exactamente una petición de marcado en la cola de prioridad, más una petición de colgado para cada módem conectado en ese momento. Siempre que eliminamos un evento en la línea 13 y se confirma que es un marcado, generamos un nuevo evento de marcado en la línea 42. Además, en la línea 37 se genera un evento de colgado cuando el marcado tiene éxito. Por tanto, la única forma de terminar la rutina es haciendo que en algún momento `sigLlamada` no genere un evento, o ejecutando a ejecutar la instrucción `break` de la línea 17 (lo cual es más probable).

Cuando el usuario de un módem cuelga, `modemsLibres` se incrementa. Si un usuario marca se comprueba si hay algún módem disponible, y si es así, se decrementa `modemsLibres`.

A continuación se muestra un resumen de cómo se procesan los distintos eventos. Si el evento es de colgado, incrementamos `modemsLibres` en la línea 21 e imprimimos un mensaje en las líneas 22 y 23. Si el evento es de marcado, generamos una parte de una frase como salida que registra el intento, y entonces si hay módems disponibles, conectamos al usuario. Para ello, decrementamos `modemsLibres` en la línea 31 y generamos un tiempo de conexión (usando una distribución de Poisson en lugar de una uniforme) en la línea 32. Entonces imprimimos el resto de la línea de salida en las líneas 33 y 34 y añadimos un evento de colgado al conjunto de eventos (líneas 35 a 37). Si no hay módems libres, generamos un mensaje de señal de ocupado. En ambos casos, se concluye generando un nuevo evento de marcado. La Figura 13.10 muestra el estado de la cola de prioridad después de cada `eliminarMin` para las primeras fases de la salida del ejemplo que se muestra en la Figura 13.4. El momento en que se produce cada evento se muestra en negrita, y el número de módems libres (si hay alguno) se muestra a la derecha de la cola de prioridad. La secuencia de pasos es la siguiente:

```
1  /**
2   * Ejecuta la simulación hasta llegar a tpoParada.
3   * Produce una salida como la de la Figura 13.4.
4   */
5  public void ejecSim( long tpoParada )
6  {
7      Evento e = null;
8      long duracion;
9
10     while( !conjEventos.esVacia( ) )
11     {
12         try
13           { e = (Evento) conjEventos.eliminarMin( ); }
14         catch( DesbordamientoInferior ex ) { } // No puede suceder
15
16         if( e.tiempo > tpoParada )
17             break;
18
19         if( e.que == Evento.COLGADO ) // COLGADO
20         {
21             modemsLibres++;
22             System.out.println( "El usuario " + e.quien +
23                                 " cuelga en minuto "+ .tiempo );
24         }
25         else                              // MARCADO
26         {
27             System.out.print( "El usuario " + e.quien +
28                     " marca en minuto " + e.tiempo + " " );
29             if( modemsLibres > 0 )
30             {
31                 modemsLibres--;
32                 duracion = r.poisson( tpoMedioLlamada );
33                 System.out.println( "y se conecta durante "
34                                     + duracion + " minutos" );
35                 e.tiempo += duracion;
36                 e.que = Evento.COLGADO;
37                 conjEventos.insertar( e );
38             }
39             else
40                 System.out.println( "pero recibe señal de ocupado" );
41
42             sigLlamada( frecLlamadas );
43         }
44     }
45 }
```

Figura 13.9 Rutina básica de simulación.

1. Se inserta el primer MARCADO.
2. Después de eliminar MARCADO, se conecta al usuario, lo cual genera un COLGADO y una nueva petición de MARCADO.
3. Se procesa una petición de COLGADO.
4. Se procesa una petición de MARCADO que resulta en una conexión, por lo que se añaden tanto un evento MARCADO como un evento COLGADO (tres veces).
5. Una petición de MARCADO falla; en su lugar se genera una nueva petición de MARCADO (tres veces).

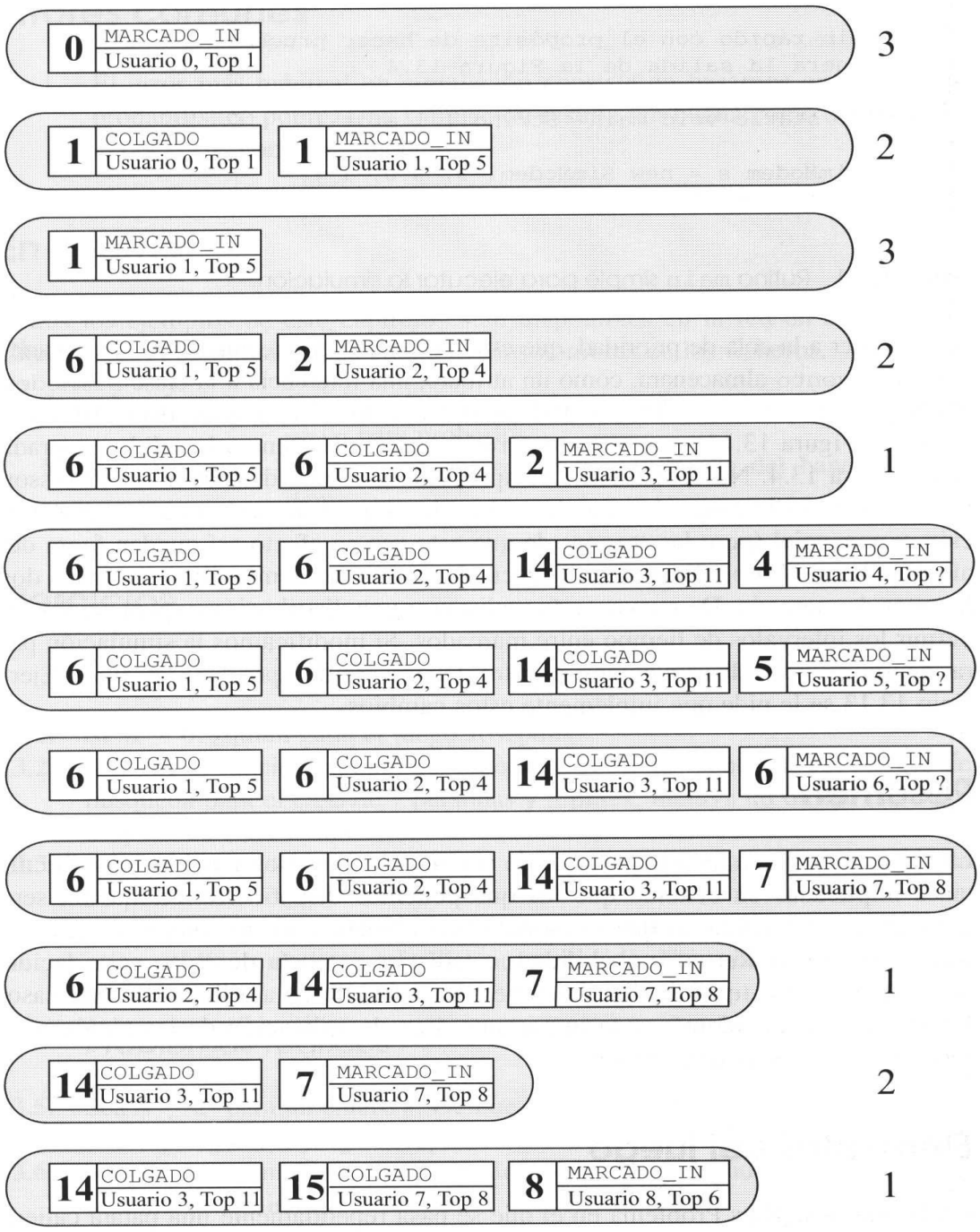


Figura 13.10 Estado de la cola de prioridad para la simulación del banco de módems tras cada paso de la simulación.

6. Se procesa una petición de `COLGADO` (dos veces).
7. Una petición de `MARCADO` tiene éxito, y se añaden un evento `MARCADO` y otro `COLGADO`.

De nuevo, si `Evento` fuera una clase base abstracta, sería de esperar que se definiera un procedimiento `procesarEvento` a través de la jerarquía `Evento`, con lo que no necesitaríamos largas cadenas de instrucciones `if/else`. Sin embargo,

```
1  /**
2   * main rápido con el propósito de hacer pruebas.
3   * Genera la salida de la Figura 13.4
4   */
5  public static void main( String [ ] args )
6  {
7      SimModem s = new SimModem( 3, 5.0, 1 );
8      s.ejecSim( 20 );
9  }
```

Figura 13.11 Rutina `main` simple para ejecutar la simulación.

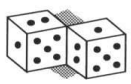
para acceder a la cola de prioridad, que está en la clase de la simulación, necesitaríamos que `Evento` almacenara, como un atributo, una referencia a la clase `SimModem` (en la que está contenido). Esto se haría en el momento de aplicación del constructor.

La simulación usa un modelo probabilístico pobre. Una distribución exponencial modelizaría de forma más exacta el tiempo entre intentos de marcado y el tiempo de conexión total.

En la Figura 13.11 se muestra una rutina `main` que genera la salida mostrada en la Figura 13.4. No obstante, no es apropiado usar una distribución de Poisson para modelizar el tiempo de conexión. Una elección mejor sería usar una distribución exponencial (pero las razones de que esto sea en efecto así quedan fuera del alcance de este libro). Tampoco es adecuado asumir un tiempo fijo entre cada dos intentos de marcado. De nuevo, sería mejor usar una distribución exponencial para definir los intervalos de tiempo entre marcados. Si modificamos la simulación para incorporar estas distribuciones, el reloj debería ser de tipo `double`. En el Ejercicio 13.13 se le pide que implemente estos cambios.

Resumen

La simulación es un área importante de la computación. Hay muchas más dificultades implicadas en el tema que las que podemos discutir. La simulación será acertada en la medida en que el modelo probabilístico lo sea, luego se requiere una sólida formación en probabilidad, estadística y teoría de colas para decidir acertadamente los tipos de distribuciones de probabilidad a utilizar en cada caso. La simulación es, además, una importante área de aplicación de las técnicas de modelización orientadas a objetos.



Elementos del juego

problema Josephus Problema en el que se pasa repetidamente una patata caliente. Cuando se termina de pasar, el jugador que se queda con la patata queda eliminado. El juego continúa entonces, y el jugador que quede el último gana.

simulación Uso importante de los computadores en el que el computador emula el funcionamiento de un sistema real y genera medidas estadísticas de su comportamiento.

simulación dirigida por eventos Simulación en la que en cada paso el reloj se avanza hasta el siguiente evento.

simulación dirigida por tiempos discretos Simulación en la que cada unidad de tiempo se procesa consecutivamente. Resulta inapropiada si el intervalo entre eventos sucesivos es grande.

tic Unidad de tiempo en una simulación.

Errores comunes

1. El error más habitual en simulación consiste en basarnos en un modelo probabilístico pobre. Una simulación resulta tan buena como acertada sea la del correspondiente modelo.

En Internet

Todos los ejemplos de este capítulo están disponibles en la red en el directorio **Chapter13**. A continuación se enumeran los nombres de los ficheros.

Josephus.java	Contiene las dos implementaciones de `josephus` y un `main` para probarlos.
ModemSim.java	Contiene el código de la simulación del banco de módems.

Ejercicios

Cuestiones breves

13.1. Si $M = 0$, ¿quién gana el juego Josephus?

13.2. Muestre el funcionamiento del algoritmo Josephus con un árbol binario de búsqueda para el caso de 7 personas y 3 pases. Incluya un dibujo del árbol después de cada eliminación.

13.3. ¿Hay algún valor de M para el que el jugador 1 gana un juego Josephus de 30 personas?

13.4. Muestre el estado de la cola de prioridad después de cada una de las diez primeras líneas de la simulación mostrada en la Figura 13.4.

Problemas teóricos

13.5. Sea $N = 2^k$ para un entero k. Demuestre que si M es 1, entonces el jugador 1 siempre gana el juego Josephus.

13.6. Sea $J(N)$ el ganador de un juego Josephus con N jugadores y $M = 1$. demuestre lo siguiente:

a) Si N es par, $J(N) = 2J(N/2) - 1$.
b) Si N es impar y $J(\lceil N/2 \rceil) \neq 1$, entonces $J(N) = 2J(\lceil N/2 \rceil) - 3$.
c) Si N es impar y $J(\lceil N/2 \rceil) = 1$, entonces $J(N) = N$.

13.7. Use los resultados del Ejercicio 13.6 para escribir un algoritmo que devuelva el ganador de un juego Josephus con N jugadores y $M = 1$. ¿Cuál es el tiempo de ejecución de dicho algoritmo?

13.8. Proporcione una fórmula general para el ganador de un juego Josephus con N jugadores cuando $M = 2$.

13.9. Usando el algoritmo para $N = 20$, determine el orden de inserción en el `ABBConRango`.

13.10. Demuestre que después de aplicar `construirArbol` (de la Figura 13.3), cada hoja está en uno de los dos últimos niveles del árbol.

Problemas prácticos

13.11. Escriba un programa que resuelva la versión histórica del problema Josephus. Proporcione los algoritmos que hacen uso de la lista enlazada y del árbol de búsqueda.

13.12. Implemente el algoritmo Josephus usando una cola. Cada pase de la patata viene dado por un `quitarPrimero` seguido de un `insertar`.

13.13. Modifique la simulación de forma que el reloj se represente mediante una variable de tipo `double`, y tanto el tiempo entre dos intentos de marcado como el tiempo de conexión se modelicen con una distribución exponencial.

13.14. Modifique la simulación del banco de módems de forma que `Evento` sea una clase base abstracta, y `EventoMarcado` y `EventoColgado` sean clases derivadas de ella. La clase `Evento` debería almacenar una referencia `SimModem` como atributo adicional, que se inicializaría en el constructor. También debería proporcionarse un método abstracto implementado en las clases derivadas, llamado `procesarEvento`, al que se llamaría desde `ejecSim` para procesar el evento.

Prácticas de programación

13.15. Implemente el algoritmo Josephus usando árboles de ensanchamiento (*véase* el Capítulo 21) e inserción secuencial. (La clase de árboles de ensanchamiento está disponible en la red, pero necesitará añadir un método `buscarKesimo`). Compare su eficiencia con el que aparece en el libro y con un algoritmo que use un algoritmo lineal de construcción de un árbol equilibrado.

13.16. Rescriba el algoritmo Josephus de la Figura 13.3 para usar un *montículo de medianas* (Ejercicio 6.20). Use una implementación simple del montículo de medianas en la que los elementos se mantengan ordenados. Compare los tiempos de ejecución de este algoritmo y del que usa un árbol binario de búsqueda.

13.17. Suponga que la FIU ha instalado un sistema el cual, cuando los módems están ocupados, pone las llamadas de teléfono en una cola. Rescriba la rutina de simulación para permitir colas de distintos tamaños e incluso de tamaño infinito.

13.18. Rescriba la simulación para recoger mediciones estadísticas en lugar de producir como salida cada evento. Después compare la velocidad de la simulación, suponiendo que se dispone de varios cientos de módems y que la simulación es muy larga, con otras que utilicen otras versiones de colas con prioridad (algunas de las cuales están disponibles en la red), como por ejemplo las siguientes:

a) La cola de prioridad asintóticamente ineficiente que se describe en el Ejercicio 6.16.

b) La cola de prioridad asintóticamente ineficiente que se describe en el Ejercicio 6.17.
c) Árboles de ensanchamiento (*véase* el Capítulo 21).
d) Montículos sesgados[3](*véase* el Capítulo 22).
e) Montículos de emparejamientos[4](*véase* el Capítulo 22).

13.19. Implemente un applet que ilustre el funcionamiento del juego Josephus usando tanto el algoritmo que utiliza una lista enlazada como el que utiliza un árbol de búsqueda.

[3] *N. del T.:* En inglés «skew heaps».
[4] *N. del T.:* En inglés «pairing heaps».