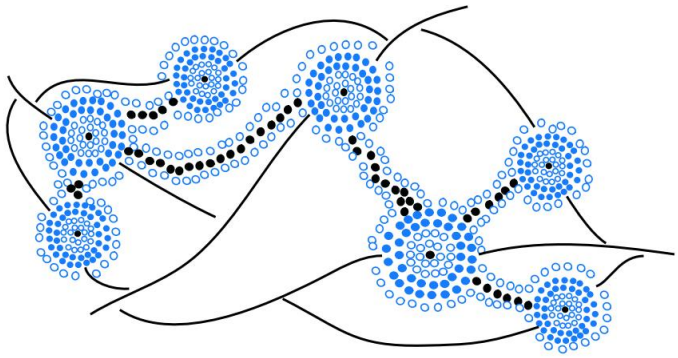
Capítulo

3 estructuras de datos fundamentales



Contenido

3.1 Uso de matrices. 104
3.1.1 Almacenamiento de entradas de juego en una matriz. 104
3.1.2 Ordenar una matriz 110
3.1.3 Métodos java.util para matrices y números aleatorios 112
3.1.4 Criptografía simple con matrices de caracteres. 115
3.1.5 Matrices bidimensionales y juegos posicionales. 118
3.2 Listas simplemente enlazadas. 122
3.2.1 Implementación de una clase de lista simplemente enlazada. 126
3.3 Listas enlazadas circularmente. 128
3.3.1 Programación round-robin. 128
3.3.2 Diseño e implementación de una lista enlazada circularmente. 129
3.4 Listas doblemente enlazadas 132
3.4.1 Implementación de una clase de lista doblemente enlazada. 135
3.5 Prueba de equivalencia. 138
3.5.1 Pruebas de equivalencia con matrices. 139
3.5.2 Pruebas de equivalencia con listas enlazadas. 140
3.6 Clonación de estructuras de datos 141
3.6.1 Clonación de matrices. 142
3.6.2 Clonación de listas enlazadas 144
3.7 Ejercicios 145

3.1 Uso de matrices

En esta sección, exploramos algunas aplicaciones de las matrices: las estructuras de datos concretas introducido en la Sección 1.3 que acceden a sus entradas utilizando índices enteros.

3.1.1 Almacenamiento de entradas de juego en una matriz

La primera aplicación que estudiamos es almacenar una secuencia de entradas con puntuaciones altas para un vídeo. juego en una matriz. Esto es representativo de muchas aplicaciones en las que una secuencia de objetos deben almacenarse. Podríamos haber optado por almacenar registros para pacientes en un hospital o los nombres de los jugadores de un equipo de fútbol. Sin embargo, dejemos Nos centramos en almacenar entradas con puntuaciones altas, lo cual es una aplicación sencilla que ya está Lo suficientemente rico como para presentar algunos conceptos importantes de estructuración de datos.

Para comenzar, consideremos qué información incluir en un objeto que representa una Entrada de puntuación alta. Obviamente, un componente a incluir es un número entero que representa La partitura en sí, que identificamos como puntuación. Otra cosa útil para incluir es la Nombre de la persona que obtuvo esta puntuación, que identificamos como nombre. Podríamos ir A partir de aquí, agregue campos que representen la fecha en que se obtuvo el puntaje o el juego. estadísticas que llevaron a esa puntuación. Sin embargo, omitimos esos detalles para mantener nuestro ejemplo. Simple. Una clase Java, GameEntry, que representa una entrada de juego, se proporciona en el código. Fragmento 3.1.

```
1 clase pública GameEntry {
2 cadena privada nombre; // el                    // nombre de la persona que obtuvo esta puntuación
3 int privado puntuación; /      valor de la puntuación
4              Construye una entrada de juego con los parámetros dados.    /
5       Entrada de juego pública (Cadena n, int s) {
6           nombre = n;
7           puntuación = s;
8       }
9       /       Devuelve el campo de nombre.    /
10      public String getName() { return nombre; }
11      /       Devuelve el campo de puntuación.    /
12      public int getScore() { devolver puntuación; }
13      /       Devuelve una representación de cadena de esta entrada.    /
14      Cadena pública toString() {
15 return "(" + nombre + ", " + puntuación + ")";
16      }
17 }
```

Fragmento de código 3.1: Código Java para una clase GameEntry simple. Tenga en cuenta que incluimos métodos para devolver el nombre y la puntuación de un objeto de entrada de juego, así como un método para devolver una representación de cadena de esta entrada.

Una clase para obtener puntuaciones altas

Para mantener una secuencia de puntuaciones altas, desarrollamos una clase llamada Marcador.
El marcador está limitado a una cierta cantidad de puntuaciones altas que se pueden guardar; una vez que eso sucede,
Se alcanza el límite, una nueva puntuación solo califica para el marcador si es estrictamente superior
que la puntuación más baja del tablero. La longitud del marcador deseado
Puede depender del juego, quizás 10, 50 o 500. Dado que ese límite puede variar,
permitir que se especifique como parámetro para nuestro constructor de Scoreboard.

Internamente, usaremos una matriz llamada board para administrar las instancias de GameEntry que representan las puntuaciones más altas. La matriz se asigna con el valor especificado.
Capacidad máxima, pero todas las entradas son inicialmente nulas. A medida que se agreguen entradas,
Mantenlos desde la puntuación más alta hasta la más baja, comenzando en el índice 0 de la matriz. Nosotros
ilustrar un estado típico de la estructura de datos en la Figura 3.1 y proporcionar el código Java para
Construya dicha estructura de datos en el Fragmento de Código 3.2.

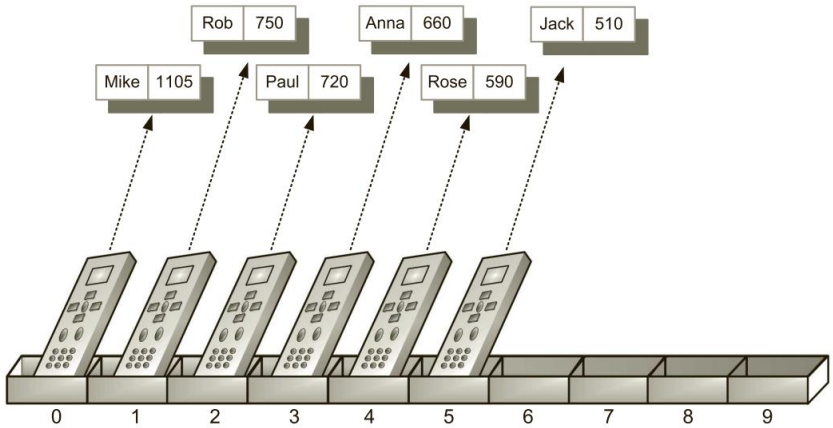


Figura 3.1: Una ilustración de una matriz de longitud diez que almacena referencias a seis
Objetos GameEntry en las celdas con índices 0 a 5; el resto son referencias nulas.

```
1 /       Clase para almacenar puntuaciones altas en una matriz en orden no decreciente.    /
2 Marcador de clase pública {
3       int privado numEntries = 0; private                    // número de entradas reales
4       GameEntry[ ] board; // matriz de entradas de juego (nombres y puntuaciones)
5       /       Construye un marcador vacío con la capacidad dada para almacenar entradas.    /
6       Marcador público(int capacidad) {
7           tablero = nueva GameEntry[capacidad];
8       }
...     // Más métodos irán aquí
36 }
```

Fragmento de código 3.2: El comienzo de una clase Scoreboard para mantener un conjunto de
Puntuaciones como objetos GameEntry. (Completado en los fragmentos de código 3.3 y 3.4).

Agregar una entrada

Una de las actualizaciones más comunes que podríamos querer hacer en un marcador es agregar una nueva entrada. Tenga en cuenta que no todas las entradas calificarán necesariamente como altas Puntuación. Si el tablero aún no está lleno, se conservará cualquier nueva entrada. Una vez que el tablero esté lleno completo, una nueva entrada solo se conserva si es estrictamente mejor que una de las otras puntuaciones, en particular, la última entrada del marcador, que es la más baja de las puntuaciones altas.

El fragmento de código 3.3 proporciona una implementación de un método de actualización para el Clase de marcador que considera la adición de una nueva entrada de juego.

```
9   /      Intenta agregar una nueva puntuación a la colección (si es lo suficientemente alta)    /
10  público void agregar(GameEntry e) {
11      int nuevaPuntuación = e.obtenerPuntuación( );
12      // ¿La nueva entrada es realmente una puntuación alta?
13      si (numEntradas < tablero.longitud || nuevaPuntuación > tablero[numEntradas−1].obtenerPuntuación()) {
14          si (numEntradas < tablero.longitud)                    // No se baja ninguna puntuación del tablero
15              numEntries++; // entonces el número total aumenta
16          // desplaza las puntuaciones más bajas hacia la derecha para hacer espacio para la nueva entrada
17          int j = numEntradas − 1;
18          mientras (j > 0 && tablero[j−1].getScore( ) < nuevoScore) {
19              tablero[j] = tablero[j−1]; j−−; }                  // desplazar la entrada de j-1 a j
20                                                                 // y decrementar j
21
22          tablero[j] = e; }                                      // cuando termine, agregue una nueva entrada
23
24  }
```

Fragmento de código 3.3: Código Java para insertar un objeto GameEntry en un marcador.

Al considerar una nueva puntuación, el primer objetivo es determinar si se considera una puntuación alta. Este será el caso (véase la línea 13) si el marcador está por debajo de su... capacidad, o si la nueva puntuación es estrictamente mayor que la puntuación más baja del tablero.

Una vez que se ha determinado que se debe mantener una nueva entrada, hay dos Tareas restantes: (1) actualizar correctamente el número de entradas y (2) colocar las nuevas entrada en la ubicación apropiada, cambiando las entradas con puntuaciones inferiores según sea necesario.

La primera de estas tareas se maneja fácilmente en las líneas 14 y 15, ya que el número total El número de inscripciones solo se puede aumentar si el tablero aún no está al máximo de su capacidad. (Cuando esté lleno, La adición de una nueva entrada se verá contrarrestada por la eliminación de la entrada con puntuación más baja.)

La colocación de la nueva entrada se implementa en las líneas 17 a 22. El índice j es Inicialmente se establece en numEntries − 1, que es el índice en el que se colocará la última GameEntry. residen después de completar la operación. O bien j es el índice correcto para el más reciente La entrada, o una o más inmediatamente anteriores, tendrán puntuaciones menores. El bucle while verifica la condición compuesta, desplazando las entradas hacia la derecha y disminuyendo j, como siempre que haya otra entrada en el índice j −1 con una puntuación menor que la nueva puntuación.

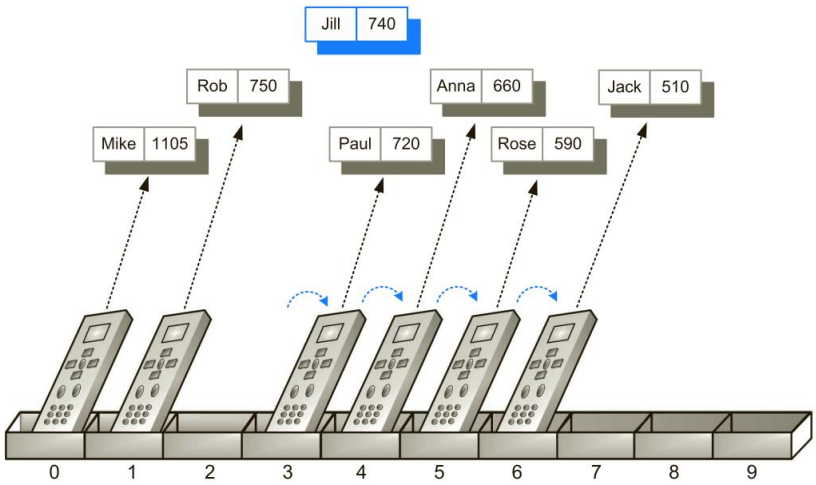


Figura 3.2: Preparación para añadir el objeto GameEntry de Jill al array del tablero. Para dejar espacio para la nueva referencia, debemos desplazar una celda a la derecha las referencias a entradas de juego con puntuaciones inferiores a la nueva.

La Figura 3.2 muestra un ejemplo del proceso, justo después de desplazar las entradas existentes, pero antes de añadir la nueva. Al completarse el bucle, j será el índice correcto para la nueva entrada. La Figura 3.3 muestra el resultado de una operación completa, tras la asignación de board[j] = e, realizada en la línea 22 del código.

En el ejercicio C-3.19, exploramos cómo se podría simplificar la adición de entradas de juego para el caso en el que no necesitamos preservar los órdenes relativos.

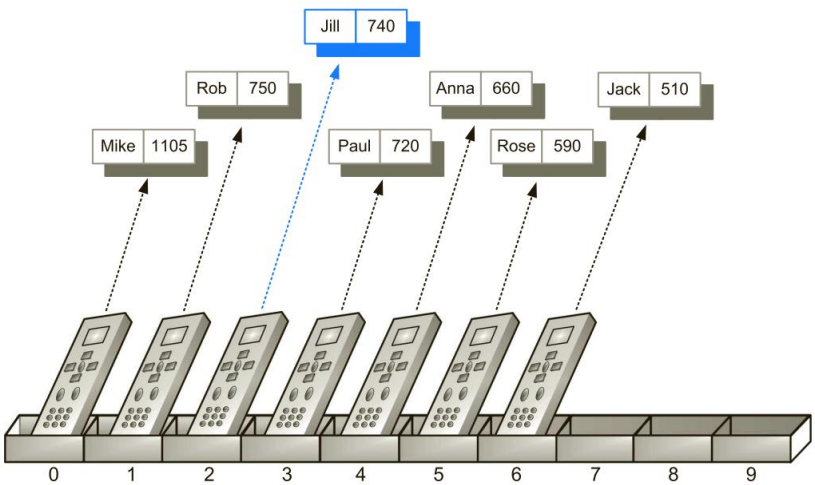


Figura 3.3: Añadiendo una referencia al objeto GameEntry de Jill al array del tablero. La referencia ahora se puede insertar en el índice 2, ya que hemos desplazado a la derecha todas las referencias a objetos GameEntry con puntuaciones inferiores a la nueva.

Eliminar una entrada

Supongamos que alguien importante juega a nuestro videojuego y su nombre aparece en nuestra lista de favoritos. lista de puntuaciones, pero luego nos enteramos de que hubo trampa. En este caso, podríamos querer tener un método que nos permita eliminar una entrada de juego de la lista de puntuaciones altas. Por lo tanto, consideremos cómo podríamos eliminar una referencia a un objeto GameEntry de un marcador.

Elegimos agregar un método a la clase Scoreboard, con la firma remove(i), Donde i designa el índice actual de la entrada que debe eliminarse y devolverse. Al eliminar una puntuación, las puntuaciones inferiores se desplazarán hacia arriba para completarla. Para la entrada eliminada. Si el índice i está fuera del rango de entradas actuales, el método lanzará una IndexOutOfBoundsException.

Nuestra implementación para eliminar implicará un bucle para cambiar entradas, mucho Como nuestro algoritmo de suma, pero a la inversa. Para eliminar la referencia al objeto En el índice i, comenzamos en el índice i y movemos todas las referencias en índices superiores a i una celda a la izquierda. (Ver Figura 3.4.)

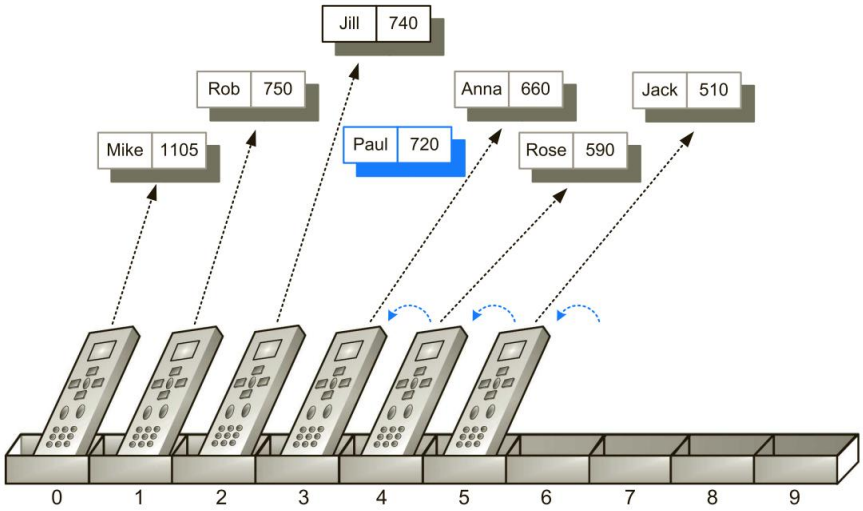


Figura 3.4: Una ilustración de la eliminación de la puntuación de Paul del índice 3 de una matriz almacenar referencias a objetos GameEntry.

Se proporciona nuestra implementación del método remove para la clase Scoreboard en el fragmento de código 3.4. Los detalles para realizar la operación de eliminación contienen algunos puntos sutiles. El primero es que, para eliminar y devolver la entrada del juego (vamos Llamémoslo e) en el índice i de nuestra matriz, primero debemos guardar e en una variable temporal. Usaremos esta variable para devolver e cuando terminemos de eliminarla.

```
25 /      Eliminar y devolver la puntuación más alta en el índice i.    /
26 public GameEntry remove(int i) lanza IndexOutOfBoundsException {
27         si (i < 0 || i >= numEntradas)
28             lanzar nueva IndexOutOfBoundsException("Índice inválido: + i);
29         GameEntry temp = tablero[i]; para                    //guarda el objeto a eliminar
30         (int j = i; j < numEntradas − 1; j++) tablero[j] =   // contar hacia arriba desde i (no hacia abajo)
31             tablero[j+1];                                    // mover una celda a la izquierda
32         tablero[numEntradas −1 ] = null;                     // anula la última puntuación anterior
33         numEntradas−−;
34         devolver temperatura; }                              // devuelve el objeto eliminado
35
```

Fragmento de código 3.4: Código Java para realizar la operación Scoreboard.remove.

El segundo punto sutil es que, al mover referencias superiores a i una celda a otra,
A la izquierda, no llegamos hasta el final del array. Primero, basamos nuestro bucle
en el número de entradas actuales, no en la capacidad de la matriz, porque hay
No hay razón para “desplazar” una serie de referencias nulas que pueden estar al final de la
matriz. También definimos cuidadosamente la condición del bucle, j < numEntries − 1, de modo que
la última iteración del bucle asigna board[numEntries−2] = board[numEntries−1].
No hay ninguna entrada para cambiar a la celda board[numEntries−1], por lo que devolvemos esa celda a
null justo después del bucle. Concluimos devolviendo una referencia a la entrada eliminada.
(que ya no tiene ninguna referencia que lo apunte dentro del arreglo del tablero).

Conclusiones

En la versión de la clase Scoreboard que está disponible en línea, incluimos una implementación del
método toString(), que nos permite mostrar el contenido de
El marcador actual, separado por comas. También incluimos un método principal que...
realiza una prueba básica de la clase.

Los métodos para agregar y eliminar objetos en una matriz de puntuaciones altas son
simples. Sin embargo, forman la base de técnicas que se utilizan repetidamente.
para construir estructuras de datos más sofisticadas. Estas otras estructuras pueden ser más
general que la estructura de matriz anterior, por supuesto, y a menudo tendrán mucho
Pueden realizar más operaciones que simplemente agregar y quitar. Pero estudiar la
La estructura de datos de matriz concreta, como la que estamos haciendo ahora, es un excelente punto
de partida para comprender estas otras estructuras, ya que cada estructura de datos debe implementarse.
utilizando medios concretos.

De hecho, más adelante en este libro, estudiaremos una clase de colecciones de Java, ArrayList,
que es más general que la estructura de matriz que estamos estudiando aquí. La ArrayList
tiene métodos para operar en una matriz subyacente; pero también elimina el error que
ocurre cuando se agrega un objeto a una matriz completa copiando automáticamente los objetos
en una matriz más grande cuando sea necesario. Analizaremos la clase ArrayList con más detalle.
detalles en la Sección 7.2.

3.1.2 Ordenar una matriz

En la subsección anterior, consideramos una aplicación en la que añadimos un objeto a un array en una posición dada, desplazando los demás elementos para mantener el orden anterior. En esta sección, utilizamos una técnica similar para resolver el problema de ordenación ; es decir, partimos de un array desordenado de elementos y los reorganizamos en orden no decreciente.

El algoritmo de ordenación por inserción

En este libro estudiamos varios algoritmos de ordenamiento, la mayoría de los cuales se describen en el Capítulo 12. A modo de introducción, en esta sección describimos un algoritmo de ordenamiento sencillo conocido como ordenamiento por inserción. El algoritmo procede considerando un elemento a la vez, colocándolo en el orden correcto con respecto a los anteriores. Comenzamos con el primer elemento del array, que se ordena trivialmente por sí mismo. Al considerar el siguiente elemento del array, si es menor que el primero, los intercambiamos. A continuación, consideramos el tercer elemento del array, intercambiándolo hacia la izquierda hasta que esté en el orden correcto con respecto a los dos primeros. Continuamos de esta manera con el cuarto elemento, el quinto, y así sucesivamente, hasta que todo el array esté ordenado. Podemos expresar el algoritmo de ordenamiento por inserción en pseudocódigo, como se muestra en el Fragmento de Código 3.5.

Algoritmo de ordenamiento por inserción (A):
Entrada: Una matriz A de n elementos comparables
Salida: La matriz A con elementos reorganizados en orden no decreciente para k desde 1 hasta n−1
Inserte A[k] en su ubicación adecuada dentro de A[0], A[1], ..., A[k].

Fragmento de código 3.5: Descripción de alto nivel del algoritmo de ordenación por inserción.

Esta es una descripción simple y detallada del ordenamiento por inserción. Si revisamos el Fragmento de Código 3.3 en la Sección 3.1.1, observamos que la tarea de insertar una nueva entrada en la lista de puntuaciones más altas es casi idéntica a la de insertar un nuevo elemento considerado en el ordenamiento por inserción (excepto que las puntuaciones del juego se ordenaron de mayor a menor). Proporcionamos una implementación en Java del ordenamiento por inserción en el Fragmento de Código 3.6, utilizando un bucle externo para considerar cada elemento por turno y un bucle interno que mueve el nuevo elemento considerado a su ubicación correcta en relación con el subconjunto (ordenado) de elementos a su izquierda. Ilustramos un ejemplo de ejecución del algoritmo de ordenamiento por inserción en la Figura 3.5.

Observamos que si un array ya está ordenado, el bucle interno del ordenamiento por inserción solo realiza una comparación, determina que no se requiere intercambio y regresa al bucle externo. Por supuesto, podríamos tener que hacer mucho más trabajo si el array de entrada está extremadamente desordenado. De hecho, tendremos que hacer el mayor trabajo si el array de entrada está en orden decreciente.

```
/ Ordenación por inserción de una matriz de caracteres en orden no decreciente /
1 2 public static void inserciónSort(char[ ] datos) {
        int n = datos.longitud;
3 4 para (int k = 1; k < n; k++) { 5 char cur =          // comienza con el segundo caracter
datos[k]; 6 int j = k; mientras (j > 0 &&                // hora de insertar cur=data[k]
datos[j−1] > cur) { 7                                    // encuentra el índice correcto j para cur
            datos[j] = datos[j−1]; 8 9 j−−; 10 }         // por lo tanto, data[j-1] debe ir después de cur
                                                         // desliza los datos[j-1] hacia la derecha
                                                         // y considera la j anterior para cur

11 datos[j] = cur; }                                     //este es el lugar apropiado para cur

12 }
```

Fragmento de código 3.6: Código Java para realizar una ordenación por inserción en una matriz de caracteres.

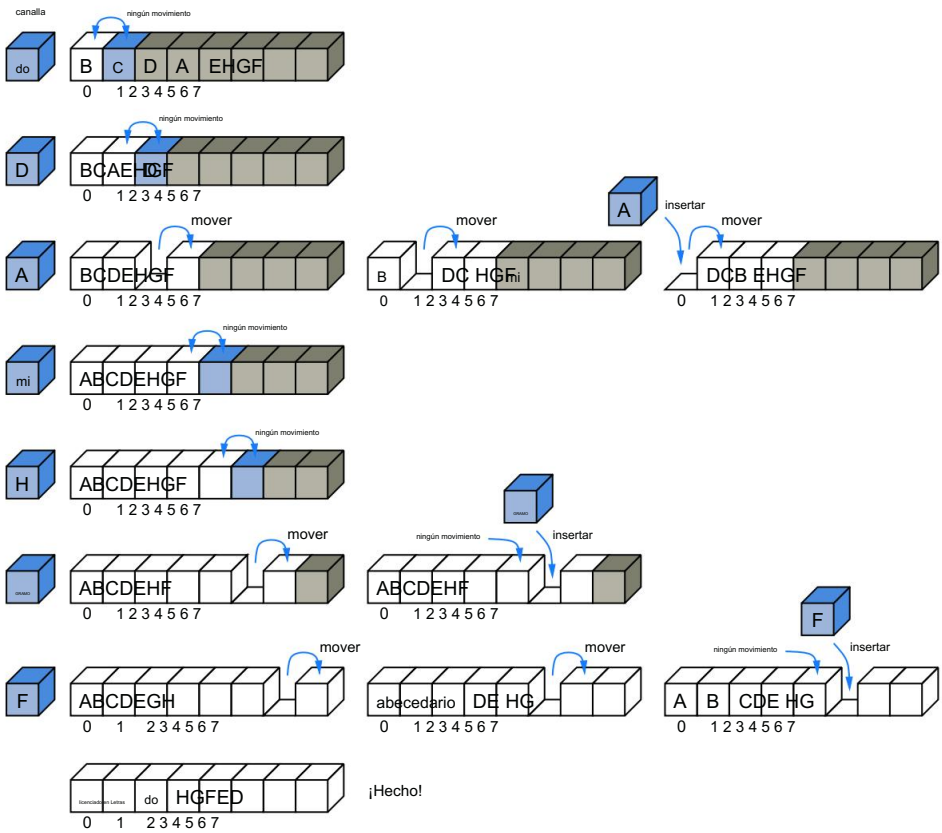


Figura 3.5: Ejecución del algoritmo de ordenamiento por inserción en un array de ocho caracteres. Cada fila corresponde a una iteración del bucle externo, y cada copia del La secuencia en una fila corresponde a una iteración del bucle interno. El elemento actual Lo que se está insertando se resalta en la matriz y se muestra como el valor actual.

3.1.3 Métodos java.util para matrices y números aleatorios

Dada la importancia de los arrays, Java proporciona una clase, java.util.Arrays, con varios métodos estáticos integrados para realizar tareas comunes con ellos. Más adelante en este libro, describiremos los algoritmos en los que se basan varios de estos métodos. Por ahora, ofrecemos una descripción general de los métodos más utilizados de esta clase, como se indica a continuación (más información en la Sección 3.5.1):

equals(A, B): Devuelve verdadero solo si los arrays A y B son iguales. Dos arrays se consideran iguales si tienen el mismo número de elementos y cada par de elementos correspondiente es igual. Es decir, A y B tienen los mismos valores en el mismo orden.

fill(A, x): almacena el valor x en cada celda de la matriz A, siempre que el tipo de la matriz A esté definido de modo que se le permita almacenar el valor x.

copyOf(A, n): Devuelve una matriz de tamaño n tal que los primeros k elementos de esta matriz se copian desde A, donde k = min{n,A.length}. Si n > A.length, entonces los últimos n − A.length elementos en esta matriz se rellenarán con valores predeterminados, por ejemplo, 0 para una matriz de int y null para una matriz de objetos.

copyOfRange(A, s, t): Devuelve una matriz de tamaño t − s tal que los elementos de esta matriz se copian en orden desde A[s] hasta A[t −1], donde s < t, rellenado como con copyOf() si t > A.length.

toString(A): Devuelve una representación en cadena del array A, comenzando con [y terminando con], y con los elementos de A separados por la cadena ", ". La representación en cadena de un elemento A[i] se obtiene mediante String.valueOf(A[i]), que devuelve la cadena "null" para una referencia nula y, en caso contrario, llama a A[i].toString(). sort(A): Ordena el array A según el orden natural de sus elementos, que deben ser comparables. Los algoritmos de ordenación son el tema central del Capítulo 12.

binarySearch(A, x): Busca el valor x en la matriz ordenada A , devolviendo el índice donde se encuentra o el índice donde podría insertarse, manteniendo el orden. El algoritmo de búsqueda binaria se describe en la Sección 5.1.3.

Como métodos estáticos, se invocan directamente en la clase java.util.Arrays, no en una instancia específica de la clase. Por ejemplo, si data fuera un array, podríamos ordenarlo con la sintaxis java.util.Arrays.sort(data), o con la sintaxis más corta Arrays.sort(data) si primero importamos la clase Arrays (véase la Sección 1.8).

Generación de números pseudoaleatorios

Otra característica integrada en Java, que suele ser útil al probar programas que manejan matrices, es la capacidad de generar números pseudoaleatorios, es decir, números que parecen aleatorios (pero no necesariamente lo son). En particular, Java cuenta con una clase integrada, java.util.Random, cuyas instancias son generadores de números pseudoaleatorios, es decir, objetos que calculan una secuencia de números estadísticamente aleatorios. Sin embargo, estas secuencias no son realmente aleatorias, ya que es posible predecir el siguiente número de la secuencia dada la lista anterior de números. De hecho, un generador de números pseudoaleatorios popular consiste en generar el siguiente número, next, a partir del número actual, cur, según la fórmula (en sintaxis de Java):

siguiente = (a cur + b) % n;

Donde a, b y n son enteros seleccionados apropiadamente, y % es el operador de módulo. Algo similar es, de hecho, el método utilizado por los objetos java.util.Random, con n = 248. Resulta que se puede demostrar que dicha secuencia es estadísticamente uniforme, lo cual suele ser suficiente para la mayoría de las aplicaciones que requieren números aleatorios, como los juegos. Para aplicaciones como la seguridad informática, donde se requieren secuencias aleatorias impredecibles, no se debe utilizar este tipo de fórmula. En su lugar, idealmente se debería utilizar una muestra de una fuente realmente aleatoria, como la estática de radio procedente del espacio exterior.

Dado que el siguiente número en un generador pseudoaleatorio se determina a partir del o los números anteriores, dicho generador siempre necesita un punto de partida, denominado semilla. La secuencia de números generada para una semilla dada siempre será la misma.
La semilla para una instancia de la clase java.util.Random se puede establecer en su constructor o con su método setSeed().

Un truco común para obtener una secuencia diferente cada vez que se ejecuta un programa es usar una semilla que sea distinta en cada ejecución. Por ejemplo, podríamos usar una entrada cronometrada del usuario o establecer la semilla con la hora actual en milisegundos desde el 1 de enero de 1970 (proporcionada por el método System.currentTimeMillis).

Los métodos de la clase java.util.Random incluyen los siguientes:

nextBoolean( ): Devuelve el siguiente valor booleano pseudoaleatorio.

nextDouble(): Devuelve el próximo valor doble pseudoaleatorio, entre 0,0 y 1,0.

nextInt( ): Devuelve el próximo valor int pseudoaleatorio.

nextInt(n): Devuelve el próximo valor int pseudoaleatorio en el rango desde 0 hasta n pero sin incluirlo.

setSeed(s): establece la semilla de este generador de números pseudoaleatorios en s larga .

Un ejemplo ilustrativo

Proporcionamos un programa ilustrativo breve (pero completo) en el Fragmento de Código 3.7.

```
1 importar java.util.Arrays;
2 importar java.util.Random;
3 /      Programa que muestra algunos usos de matrices.    /
4 clase pública ArrayTest {
      public static void main(String[ ] args) {
5 6 int datos[ ] = nuevo int[10];
7         Random rand = new Random(); // un generador de números pseudoaleatorios
          rand.setSeed(System.currentTimeMillis()); // usa la hora actual como semilla
8         // llena la matriz de datos con números pseudoaleatorios de 0 a 99, inclusive
9 10      para (int i = 0; i < data.length; i++)
11            datos[i] = rand.nextInt(100);                    // el siguiente número pseudoaleatorio
12        int[ ] orig = Arrays.copyOf(data, data.length); // hacer una copia de la matriz de datos
13        System.out.println("matrices iguales antes de ordenar: "+Arrays.equals(data, orig));
14        Arrays.sort(data); // ordena la matriz de datos (el origen no cambia)
15        System.out.println("matrices iguales después de ordenar: + Arrays.equals(data, orig));
16        System.out.println("orig = + Arrays.toString(orig));
17        System.out.println("datos = + Arrays.toString(datos));
18    }
19 }
```

Fragmento de código 3.7: Una prueba simple de algunos métodos integrados en java.util.Arrays.

A continuación mostramos un ejemplo de salida de este programa:

matrices iguales antes de ordenar: verdadero
matrices iguales después de ordenar: falso
origen = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
datos = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]

En otra ejecución, obtuvimos el siguiente resultado:

matrices iguales antes de ordenar: verdadero
matrices iguales después de ordenar: falso
origen = [87, 49, 70, 2, 59, 37, 63, 37, 95, 1]
datos = [1, 2, 37, 37, 49, 59, 63, 70, 87, 95]

Al utilizar un generador de números pseudoaleatorios para determinar los valores del programa,
Recibimos una entrada diferente para nuestro programa cada vez que lo ejecutamos. Esta función es, de hecho, lo que...
hace que los generadores de números pseudoaleatorios sean útiles para probar código, particularmente cuando
Trabajando con matrices. Aun así, no deberíamos usar ejecuciones de pruebas aleatorias como reemplazo.
para razonar sobre nuestro código, ya que podríamos pasar por alto casos especiales importantes en las ejecuciones de pruebas.
Tenga en cuenta, por ejemplo, que existe una pequeña posibilidad de que las matrices orig y data sean
Igual incluso después de ordenar los datos, es decir, si el origen ya está ordenado. Las probabilidades de esto
Las probabilidades de que ocurra son menores a 1 en 3 millones, por lo que es poco probable que ocurra incluso durante unos pocos meses.
mil ejecuciones de prueba; sin embargo, debemos razonar que esto es posible.

3.1.4 Criptografía simple con matrices de caracteres

Una aplicación importante de las matrices y cadenas de caracteres es la criptografía, la ciencia de los mensajes secretos. Este campo implica el proceso de cifrado, en el que un mensaje, llamado texto plano, se convierte en un mensaje codificado, llamado texto cifrado. Asimismo, la criptografía estudia las formas correspondientes de realizar el descifrado, devolviendo un texto cifrado a su texto plano original.

Se podría decir que el esquema de cifrado más antiguo es el cifrado César, que lleva el nombre de Julio César, quien utilizó este esquema para proteger mensajes militares importantes. (Todos los mensajes de César fueron escritos en latín, por supuesto, lo que los hace ilegibles para la mayoría de nosotros). El cifrado César es una forma sencilla de ocultar un mensaje escrito en un idioma que forma palabras con un alfabeto.

El cifrado César implica reemplazar cada letra de un mensaje por la letra que se encuentra un cierto número de letras después en el alfabeto. Por lo tanto, en un mensaje en inglés, podríamos reemplazar cada A por D, cada B por E, cada C por F, y así sucesivamente, si se desplaza tres caracteres. Continuamos este método hasta llegar a la W, que se reemplaza por la Z. Luego, dejamos que el patrón de sustitución se repita, de modo que reemplazamos la X por A, la Y por B y la Z por C.

Conversión entre cadenas y matrices de caracteres

Dado que las cadenas son inmutables, no podemos editar directamente una instancia para cifrarla. En su lugar, nuestro objetivo será generar una nueva cadena. Una técnica práctica para realizar transformaciones de cadenas es crear un array equivalente de caracteres, editarlo y, a continuación, reensamblar una (nueva) cadena a partir de él.

Java tiene soporte para conversiones de cadenas a matrices de caracteres y viceversa. Dada una cadena S, podemos crear una nueva matriz de caracteres que coincida con S mediante el método S.toCharArray(). Por ejemplo, si s="bird", el método devuelve la matriz de caracteres A={'b', 'i', 'r', 'd'}. Por el contrario, existe una forma del constructor String que acepta una matriz de caracteres como parámetro. Por ejemplo, con la matriz de caracteres A={'b', 'i', 'r', 'd'}, la sintaxis new String(A) produce "bird".

Uso de matrices de caracteres como códigos de reemplazo

Si numeráramos nuestras letras como índices de matriz, de modo que A sea 0, B sea 1 y C sea 2, podemos representar la regla de reemplazo como una matriz de caracteres y un codificador, de modo que A se asigne a encoder[0], B a encoder[1], y así sucesivamente. Para encontrar un reemplazo para un carácter en nuestro cifrado César, necesitamos asignar los caracteres de la A a la Z a sus respectivos números del 0 al 25. Afortunadamente, podemos confiar en que los caracteres se representan en Unicode mediante puntos de código enteros, y los puntos de código para las letras mayúsculas del alfabeto latino son consecutivos (para simplificar, restringimos nuestro cifrado a mayúsculas).

Java nos permite restar dos caracteres entre sí, con un resultado entero igual a su distancia de separación en la codificación. Dada una variable c que se sabe que es una letra mayúscula, el cálculo de Java, j = c − 'A', produce el índice deseado j. Como comprobación, si el carácter c es 'A', entonces j = 0. Si c es 'B', la diferencia es 1. En general, el entero j resultante de dicho cálculo puede utilizarse como índice en nuestra matriz de codificador precalculada, como se ilustra en la Figura 3.6.

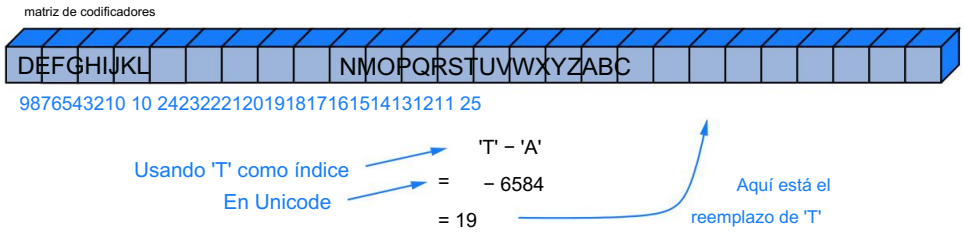


Figura 3.6: Ilustración del uso de caracteres en mayúsculas como índices, en este caso para realizar la regla de reemplazo para el cifrado con cifrado César.

El proceso de descifrado del mensaje se puede implementar simplemente utilizando una matriz de caracteres diferente para representar la regla de reemplazo (una que efectivamente desplaza los caracteres en la dirección opuesta).

En el Fragmento de Código 3.8, presentamos una clase Java que realiza el cifrado César con un desplazamiento rotacional arbitrario. El constructor de la clase crea las matrices de traducción del codificador y del decodificador para la rotación dada. Nos basamos en gran medida en la aritmética modular, ya que un cifrado César con una rotación de r codifica la letra que tiene índice k con la letra que tiene índice (k + r) módulo 26, donde mod es el operador módulo , que devuelve el residuo después de realizar una división entera. Este operador se denota con % en Java, y es exactamente el operador que necesitamos para realizar fácilmente el ajuste al final del alfabeto, para 26 módulo 26 es 0, 27 módulo 26 es 1 y 28 módulo 26 es 2. La matriz del decodificador para el cifrado César es justo lo opuesto: reemplazamos cada letra con la r que le precede, con ajuste; Para evitar sutilezas que involucran números negativos y el operador módulo, reemplazaremos la letra que tiene código k con la letra que tiene código (k−r +26) mod 26.

Con las matrices de codificador y decodificador disponibles, los algoritmos de cifrado y descifrado son esencialmente los mismos, por lo que realizamos ambos mediante un método de utilidad privado llamado "transform". Este método convierte una cadena en una matriz de caracteres, realiza la traducción diagramada en la Figura 3.6 para cualquier símbolo del alfabeto en mayúsculas y, finalmente, devuelve una nueva cadena, construida a partir de la matriz actualizada.

El método principal de la clase, como prueba simple, produce la siguiente salida: Código de

cifrado = DEFGHIJKLMNOPQRSTUVWXYZABC Código de
descifrado = XYZABCDEFGHIJKLMNOPQRSTUVW Secreto: WKH
HDJOH LV LQ SODB; PHHW DW MRH'V.
Mensaje: EL ÁGUILA ESTÁ EN JUEGO; ENCUÉNTRATE EN JOE'S.

```
1 /       Clase para realizar cifrado y descifrado utilizando el cifrado César.    /
2 clase pública CaesarCipher {
3 protected char[ ] encoder = new char[26]; // Matriz de cifrado
        protected char[ ] decoder = new char[26]; // Matriz de descifrado
4 /       Constructor que inicializa las matrices de cifrado y descifrado    /
5 público CaesarCipher(int rotación) {
6 para (int k=0; k < 26; k++) {
7
8               encoder[k] = (char) ('A' + (k + rotación) % 26);
                decodificador[k] = (char) ('A' + (k − rotación + 26) % 26);
9           }
10      }
11 12 /       Devuelve una cadena que representa un mensaje cifrado.    /
13      public String cifrar(String mensaje) {
14 return transform(mensaje, codificador); }                    // usar matriz de codificadores
15
16 /       Devuelve el mensaje descifrado dado el secreto cifrado.    /
17      public String decrypt(String secreto) {
   return transform(secreto, decodificador); 18 }                // usar matriz decodificadora
19
20 /       Devuelve la transformación de la cadena original utilizando el código dado.    /
21 transformación de cadena privada (cadena original, char[ ] código) {
22          char[ ] msg = original.toCharArray( );
23          para (int k=0; k < longitud del mensaje; k++)
24              si (Carácter.isUpperCase(msg[k])) { int j =          //Tenemos una carta para cambiar
25                  msg[k] − 'A'; msg[k] =                          // será un valor de 0 a 25
26                  código[j]; }                                    // reemplazar el caracter
27
28          devuelve nueva cadena (msg);
29      }
30 /       Método principal simple para probar el cifrado César    /
31      public static void main(String[ ] args) {
32          CaesarCipher cifrado = nuevo CaesarCipher(3);
33          System.out.println("Código de cifrado = + new String(cipher.encoder));
34          System.out.println("Código de descifrado = + new String(cipher.decoder));
35          String message = "EL ÁGUILA ESTÁ EN JUEGO; NOS ENCONTRAMOS EN CASA DE JOE.";
36          Cadena codificada = cipher.encrypt(mensaje);
37          System.out.println("Secreto: " + codificado);
38 Cadena respuesta = cipher.decrypt(codificado);
39 System.out.println("Mensaje: " + respuesta); 40 }              // debería ser texto plano nuevamente

41 }
```

Fragmento de código 3.8: Una clase Java completa para realizar el cifrado César.

3.1.5 Matrices bidimensionales y juegos posicionales

Muchos juegos de computadora, ya sean juegos de estrategia, juegos de simulación o juegos en primera persona, Los juegos de conflicto involucran objetos que residen en un espacio bidimensional. Software para Estos juegos posicionales necesitan una forma de representar objetos en dos dimensiones. espacio. Una forma natural de hacer esto es con una matriz bidimensional, donde usamos dos índices, por ejemplo i y j, para referirse a las celdas de la matriz. El primer índice suele referirse a un número de fila y el segundo a un número de columna. Dado un array de este tipo, podemos Mantener tableros de juego bidimensionales y realizar otros tipos de cálculos que involucra datos almacenados en filas y columnas.

Las matrices en Java son unidimensionales; utilizamos un único índice para acceder a cada celda de una matriz. Sin embargo, existe una forma de definir matrices bidimensionales en Java: podemos crear una matriz bidimensional como una matriz de matrices. Es decir, podemos Definir una matriz bidimensional como una matriz en la que cada una de sus celdas es otra matriz. Esta matriz bidimensional a veces también se denomina matriz. En Java, puede declarar una matriz bidimensional de la siguiente manera:

```
int[ ][ ] datos = nuevo int[8][10];
```

Esta declaración crea una “matriz de matrices” bidimensional, datos, que son 8×10, que tiene 8 filas y 10 columnas. Es decir, los datos son una matriz de longitud 8 tal que cada El elemento de datos es una matriz de longitud 10 de enteros. (Véase la Figura 3.7). Lo siguiente Entonces serían usos válidos de los datos de la matriz y de las variables int i, j y k:

```
datos[i][i+1] = datos[i][i] + 3;
j = datos.longitud; k =                    // j es 8
 datos[4].longitud;                        // k es 10
```

Las matrices bidimensionales tienen muchas aplicaciones para el análisis numérico. En lugar de entrar en los detalles de tales aplicaciones, sin embargo, exploramos una aplicación de matrices bidimensionales para implementar un juego posicional simple.

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13		77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figura 3.7: Ilustración de una matriz de enteros bidimensional, datos, que tiene 8 filas y 10 columnas. El valor de los datos[3][5] es 100 y el de los datos[6][2] es 632.

Tres en raya

Como la mayoría de los escolares saben, el tres en raya se juega en un tablero de tres por tres. Dos jugadores —X y O— se alternan colocando sus respectivas marcas en las casillas del tablero, empezando por el jugador X. Si alguno de los jugadores consigue colocar tres de sus marcas en una fila, columna o diagonal, gana.

Es cierto que no se trata de un juego posicional sofisticado, ni siquiera es muy divertido, ya que un buen jugador siempre puede forzar un empate. La ventaja del tres en raya es que es un ejemplo claro y sencillo que muestra cómo se pueden usar matrices bidimensionales para juegos posicionales. El software para juegos posicionales más sofisticados, como las damas, el ajedrez o los populares juegos de simulación, se basa en el mismo enfoque que ilustramos aquí para usar una matriz bidimensional para el tres en raya.

La idea básica es usar una matriz bidimensional, "board", para mantener el tablero de juego. Las celdas de esta matriz almacenan valores que indican si la celda está vacía o contiene una X o una O. Es decir, "board" es una matriz de tres por tres, cuya fila central consta de las celdas "board[1][0]", " board[1][1]" y "board[1][2]. En nuestro caso, optamos por que las celdas de la matriz "board" sean números enteros: 0 indica una celda vacía, 1 indica una X y −1 indica una O. Esta codificación nos permite comprobar de forma sencilla si una configuración de tablero dada es ganadora para X u O, es decir, si los valores de una fila, columna o diagonal suman 3 o −3, respectivamente. Ilustramos este enfoque en la Figura 3.8.

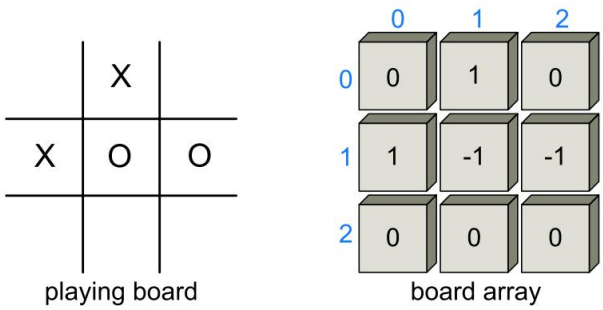


Figura 3.8: Una ilustración de un tablero de tres en raya y la matriz de números enteros bidimensionales, tablero, que lo representa.

En los fragmentos de código 3.9 y 3.10, proporcionamos una clase Java completa para el mantenimiento de un tablero de tres en raya para dos jugadores. Mostramos un ejemplo de salida en la Figura 3.9. Tenga en cuenta que este código solo sirve para mantener el tablero de tres en raya y registrar movimientos; no ejecuta ninguna estrategia ni permite jugar al tres en raya contra la computadora. Los detalles de dicho programa quedan fuera del alcance de este capítulo, pero aun así podría ser un buen proyecto de curso (véase el Ejercicio P-8.67).

```
1 /      Simulación de un juego de tres en raya (no hace estrategia).    /
2 clase pública TicTacToe {
      público estático final int X = 1, O = −1; 3 público estático                  // jugadores
4 final int VACÍO = 0; privado int tablero[ ][ ] = nuevo int[3][3];                  // celda vacía
5     privado int jugador; /      Constructor    /                                   // tablero de juego
6                                                                                    // jugador actual
7
8     público TicTacToe() { clearBoard(); }
9     /      Limpia el tablero    /
10    público void clearBoard() {
11       para (int i = 0; i < 3; i++)
12          para (int j = 0; j < 3; j++)
13             tablero[i][j] = VACÍO;                                                //cada celda debe estar vacía
14       jugador = X; }                                                             // el primer jugador es 'X'
15
16    /      Coloca una marca X u O en la posición i,j.    /
17    público void putMark(int i, int j) lanza IllegalArgumentException {
18       si ((i < 0) || (i > 2) || (j < 0) || (j > 2))
19          lanzar nueva IllegalArgumentException("Posición de tablero inválida");
20       si (tablero[i][j] != VACÍO)
21          lanzar nueva IllegalArgumentException("Posición del tablero ocupada");
22       tablero[i][j] = jugador;                  // coloca la marca para el jugador actual
23       jugador = − jugador; }                    // cambia de jugador (usa el hecho de que O = - X)
24
25    /      Comprueba si la configuración del tablero es una victoria para el jugador dado.    /
26    público booleano isWin(int marca) {
27       devuelve ((tablero[0][0] + tablero[0][1] + tablero[0][2] == marca    3)                    // fila 0
28                || (tablero[1][0] + tablero[1][1] + tablero[1][2] == marca    3) // fila 1
29                || (tablero[2][0] + tablero[2][1] + tablero[2][2] == marca    3) // fila 2
30                || (tablero[0][0] + tablero[1][0] + tablero[2][0] == marca    3) // columna 0
31                || (tablero[0][1] + tablero[1][1] + tablero[2][1] == marca    3) // columna 1
32                || (tablero[0][2] + tablero[1][2] + tablero[2][2] == marca    3) // columna 2
33                || (tablero[0][0] + tablero[1][1] + tablero[2][2] == marca    3) // diagonal
34                || (tablero[2][0] + tablero[1][1] + tablero[0][2] == marca    3)); // diagnóstico de revoluciones
35    }
36    /      Devuelve el código del jugador ganador , o 0 para indicar un empate (o juego inacabado).    /
37    público int ganador( ) {
38       si (isWin(X))
39          devolver(X);
40       de lo contrario si (isWin(O))
41          retorno(O);
42       demás
43          retorno(0);
44    }
```

Fragmento de código 3.9: Una clase Java simple y completa para jugar al tres en raya entre Dos jugadores. (Continúa en el fragmento de código 3.10.)

```
45        /       Devuelve una cadena de caracteres simple que muestra el tablero actual.    /
46 cadena pública toString() {
47            StringBuilder sb = nuevo StringBuilder();
48 para (int i=0; i<3; i++) {
49                para (int j=0; j<3; j++) {
50 interruptores (placa[i][j]) {
51 caso X: sb.append("O");                sb.append("X"); romper;
52 caso O: caso VACÍO:          break;
                  sb.append(" "); break;
54
53 }
55 si (j < 2) sb.append("|"); 56 }                                    // límite de columna

                si (i < 2) sb.append("\n-----\n"); 57 }                    // límite de fila
58
59 devolver sb.toString();
60 }
61 /     Prueba de funcionamiento de un juego simple    /
62 public static void main(String[ ] args) {
63 /          Juego TicTacToe = new TicTacToe();
64    X movimientos:    / /    O movimientos:    /
65 juego.putMark(1,1); juego.putMark(0,2);
66 juego.putMark(2,2); juego.putMark(0,0);
67            juego.putMark(0,1); juego.putMark(2,1);
68            juego.putMark(1,2); juego.putMark(1,0);
69            juego.putMark(2,0);
70            System.out.println(juego);
71            int jugadorGanador = juego.ganador();
72            String[ ] result = {"O gana", "Empate", "X gana"}; // confiar en el orden
73            System.out.println(resultado[1 + jugadorganador]);
74 }
75 }
```

Fragmento de código 3.10: Una clase Java simple y completa para jugar al tres en raya entre dos jugadores. (Continuación del Fragmento de código 3.9).

```
O|X|O
-----
O|X|X
-----
X|O|X
Atar
```

Figura 3.9: Ejemplo de salida de un juego de tres en raya.

3.2 Listas enlazadas simples

En la sección anterior, presentamos la estructura de datos de la matriz y analizamos algunos de sus aplicaciones. Las matrices son excelentes para almacenar cosas en un orden determinado, pero... Tienen desventajas. La capacidad de la matriz debe ser fija al crearse, y Las inserciones y eliminaciones en posiciones interiores de una matriz pueden consumir mucho tiempo si Muchos elementos deben ser desplazados.

En esta sección, presentamos una estructura de datos conocida como lista enlazada, que ofrece una alternativa a una estructura basada en matrices. Una lista enlazada, en su forma más simple, es una colección de nodos que colectivamente forman una secuencia lineal. En un enlace simple lista, cada nodo almacena una referencia a un objeto que es un elemento de la secuencia, como así como una referencia al siguiente nodo de la lista (ver Figura 3.10).

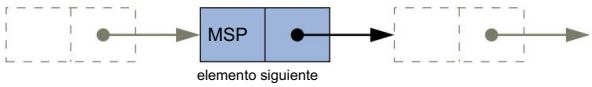


Figura 3.10: Ejemplo de una instancia de nodo que forma parte de una lista enlazada simple. El campo de elemento del nodo se refiere a un objeto que es un elemento de la secuencia (el código de aeropuerto MSP, en este ejemplo), mientras que el siguiente campo se refiere al siguiente nodo de la lista enlazada (o nulo si no hay más nodos).

La representación de una lista enlazada se basa en la colaboración de muchos objetos (ver Figura 3.11). Como mínimo, la instancia de la lista enlazada debe mantener una referencia a la primera nodo de la lista, conocido como la cabecera. Sin una referencia explícita a la cabecera, No habría forma de localizar ese nodo (ni indirectamente, ningún otro). El último El nodo de la lista se conoce como la cola. La cola de una lista se puede encontrar recorriendo el Lista enlazada: comienza en la cabeza y se mueve de un nodo a otro siguiendo La siguiente referencia de cada nodo. Podemos identificar la cola como el nodo que tiene nulo como su Siguiente referencia. Este proceso también se conoce como salto de enlace o salto de puntero. Sin embargo, almacenar una referencia explícita al nodo de cola es una eficiencia común para evitar tal recorrido. De manera similar, es común que una instancia de lista enlazada mantener un recuento del número total de nodos que componen la lista (también conocida como tamaño de la lista), para evitar recorrer la lista para contar los nodos.

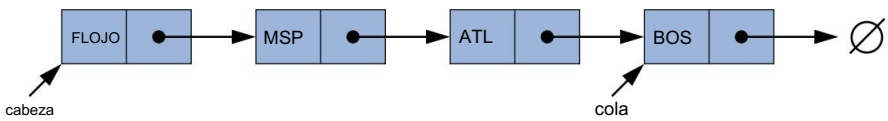


Figura 3.11: Ejemplo de una lista enlazada simple cuyos elementos son cadenas que indican códigos de aeropuerto. La instancia de lista mantiene un miembro llamado "head" que hace referencia a los primer nodo de la lista, y otro miembro llamado cola que se refiere al último nodo de la lista. El valor nulo se denota como Ø.

Inserción de un elemento al principio de una lista enlazada simple

Una propiedad importante de una lista enlazada es que no tiene un valor fijo predeterminado.
tamaño; utiliza espacio proporcional a su número actual de elementos. Al usar un
En una lista enlazada simple, podemos insertar fácilmente un elemento al principio de la lista, como se muestra
en la Figura 3.12, y descrito con pseudocódigo en el Fragmento de Código 3.11. El principal
La idea es que creamos un nuevo nodo, establecemos su elemento en el nuevo elemento, establecemos su siguiente
enlace para hacer referencia al encabezado actual y establecer el encabezado de la lista para apuntar al nuevo nodo.

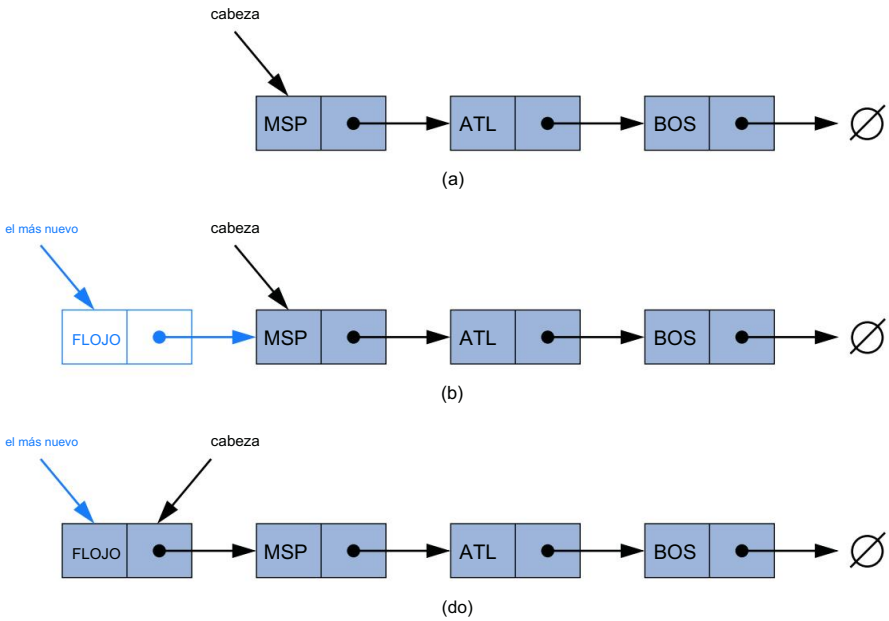


Figura 3.12: Inserción de un elemento al principio de una lista enlazada simple: (a) antes del
inserción; (b) después de crear un nuevo nodo y vincularlo al nodo principal existente; (c) después
Reasignación de la referencia de cabecera al nodo más nuevo.

Algoritmo addFirst(e):

más nuevo = Nodo(e) {crea una nueva instancia de nodo que almacena la referencia al elemento e}
latest.next = head {establece el próximo nodo nuevo para que haga referencia al antiguo nodo principal}
cabeza = más {establecer la variable head para hacer referencia al nuevo nodo}
tamaño = tamaño + 1 nuevo {incrementar el número de nodos}

Fragmento de código 3.11: Inserción de un nuevo elemento al comienzo de una lista enlazada simple.
Tenga en cuenta que establecemos el siguiente puntero del nuevo nodo antes de reasignar la variable head
Si la lista estuviera inicialmente vacía (es decir, la cabecera es nula), entonces una consecuencia natural
es que el nuevo nodo tiene su próxima referencia establecida en nula.

Inserción de un elemento al final de una lista enlazada simple

También podemos insertar fácilmente un elemento al final de la lista, siempre que mantengamos un referencia al nodo de cola, como se muestra en la Figura 3.13. En este caso, creamos un nuevo nodo, asigna su próxima referencia a nulo, establece la próxima referencia de la cola para que apunte a este nuevo nodo y luego actualizamos la referencia de cola a este nuevo nodo. Damos pseudocódigo para el proceso en el Fragmento de Código 3.12.

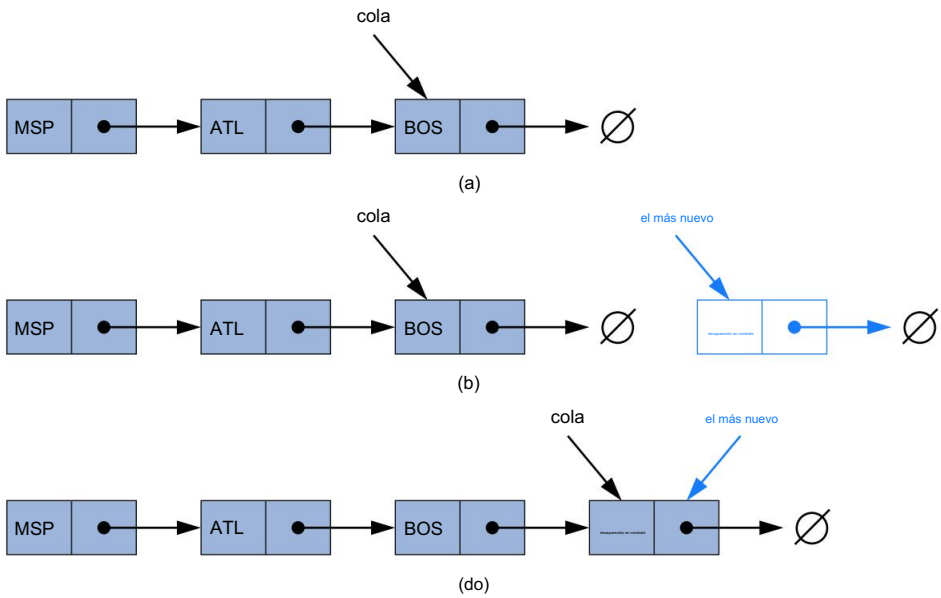


Figura 3.13: Inserción en la cola de una lista enlazada simple: (a) antes de la inserción; (b) tras la creación de un nuevo nodo; (c) tras la reasignación de la referencia de cola. Nota que debemos establecer el siguiente enlace del nodo de cola en (b) antes de asignar la variable de cola para señalar el nuevo nodo en (c).

Algoritmo addLast(e):

```
más nuevo = Nodo(e) {crea una nueva instancia de nodo que almacena la referencia al elemento e}
latest.next = null {establecer el próximo nuevo nodo para hacer referencia al objeto nulo}
tail.next = latest {hace que el antiguo nodo de cola apunte al nuevo nodo}
cola = más nuevo {establecer la variable tail para hacer referencia al nuevo nodo}
tamaño = tamaño+1 {incrementar el número de nodos}
```

Fragmento de código 3.12: Inserción de un nuevo nodo al final de una lista enlazada simple. Nota que establezcamos el siguiente puntero para el nodo de cola antiguo antes de hacer que el punto de cola sea variable al nuevo nodo. Este código debería ajustarse para insertarlo en un nodo vacío. lista, ya que no habría un nodo de cola existente.

Cómo eliminar un elemento de una lista enlazada simple

Quitar un elemento del encabezado de una lista enlazada simple es esencialmente lo inverso.
Operación de insertar un nuevo elemento en la cabecera. Esta operación se ilustra en
Figura 3.14 y se describe en detalle en el Fragmento de código 3.13.

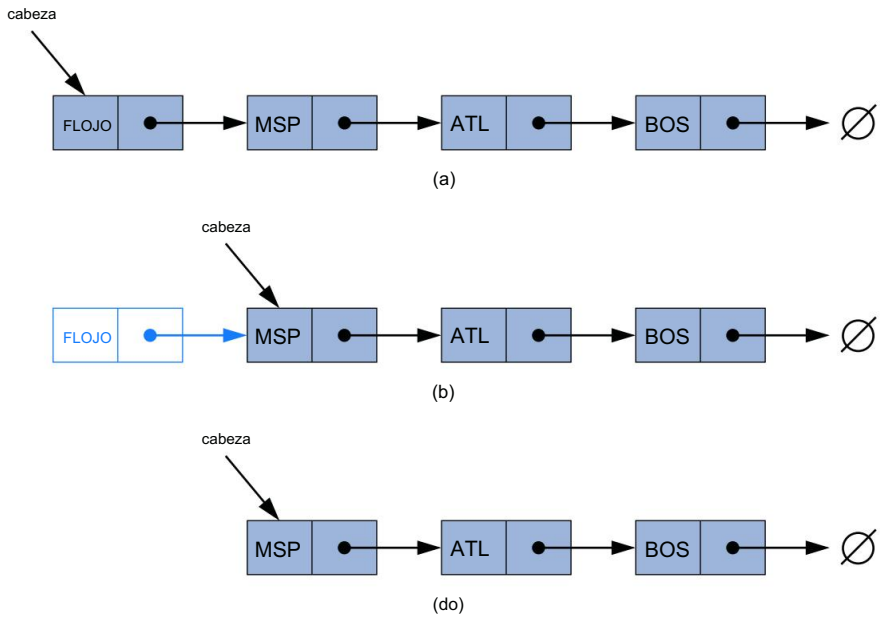


Figura 3.14: Eliminación de un elemento al principio de una lista enlazada simple: (a) antes
la eliminación; (b) después de "conectar" el cabezal antiguo; (c) configuración final.

```
Algoritmo removeFirst( ):
    si cabeza == nulo entonces
        La lista está vacía.
    cabeza = cabeza.siguiente          {hacer que la cabeza apunte al siguiente nodo (o nulo)}
    tamaño = tamaño−1                  {disminuir el número de nodos}
```

Fragmento de código 3.13: Eliminar el nodo al comienzo de una lista enlazada simple.

Desafortunadamente, no podemos eliminar fácilmente el último nodo de una lista enlazada simple. Incluso
Si mantenemos una referencia de cola directamente al último nodo de la lista, debemos poder
para acceder al nodo anterior al último nodo para eliminar el último nodo. Pero nosotros
No se puede llegar al nodo anterior a la cola siguiendo los siguientes enlaces desde la cola. El único
La forma de acceder a este nodo es comenzar desde el principio de la lista y buscar hasta el final.
a través de la lista. Pero tal secuencia de operaciones de salto de enlace podría tomar mucho tiempo.
tiempo. Si queremos apoyar una operación de este tipo de manera eficiente, necesitaremos hacer nuestro
lista doblemente enlazada (como hacemos en la Sección 3.4).

3.2.1 Implementación de una clase de lista enlazada simple

En esta sección, presentamos una implementación completa de una clase SinglyLinkedList, apoyando los siguientes métodos:

tamaño(): Devuelve el número de elementos en la lista.

isEmpty( ): devuelve verdadero si la lista está vacía y falso en caso contrario.

first(): Devuelve (pero no elimina) el primer elemento de la lista.

last(): Devuelve (pero no elimina) el último elemento de la lista.

addFirst(e): agrega un nuevo elemento al frente de la lista.

addLast(e): agrega un nuevo elemento al final de la lista.

removeFirst( ): elimina y devuelve el primer elemento de la lista.

Si se llaman a first(), last() o removeFirst() en una lista que está vacía, simplemente devuelve una referencia nula y deja la lista sin cambios.

Como no nos importa qué tipo de elementos se almacenan en la lista, Utilice el marco de genéricos de Java (ver Sección 2.5.2) para definir nuestra clase con un formato formal. parámetro de tipo E que representa el tipo de elemento deseado por el usuario.

Nuestra implementación también aprovecha el soporte de Java para clases anidadas. (véase la Sección 2.6), ya que definimos una clase Nodo privada dentro del ámbito de la clase SinglyLinkedList pública. El fragmento de código 3.14 presenta la definición de la clase Nodo. y el Fragmento de Código 3.15, el resto de la clase SinglyLinkedList. Con Node como... La clase anidada proporciona una encapsulación robusta, protegiendo a los usuarios de nuestra clase de los detalles subyacentes sobre nodos y enlaces. Este diseño también permite a Java diferenciar Este tipo de nodo proviene de formas de nodos que podemos definir para su uso en otras estructuras.

```
1 clase pública SinglelyLinkedList<E> {
      //---------- Clase de nodo anidada ----------
2 clases estáticas privadas Node<E> {
4
3 elemento E privado; Nodo<E>                  // referencia al elemento almacenado en este nodo
5       privado siguiente; Nodo                // referencia al nodo subsiguiente en la lista
6       público(E e, Nodo<E> n) {
7         elemento = e;
8         siguiente = n;
9       }
10      public E getElement() { return elemento; }
11      público Nodo<E> getNext() { devolver siguiente; }
12      público void setNext(Nodo<E> n) { siguiente = n; }
13    } //---------- fin de la clase Node anidada ----------
      ... el resto de la clase SinglyLinkedList seguirá...
```

Fragmento de código 3.14: Una clase Node anidada dentro de la clase SinglyLinkedList. (El resto de la clase SinglyLinkedList se proporcionará en el fragmento de código 3.15).

```
clase pública SinglelyLinkedList<E> {
...     (la clase Node anidada va aquí)

    // variables de instancia de SinglyLinkedList
    Nodo privado<E> head = null;                    // nodo principal de la lista (o nulo si está vacío)
privado Node<E> tail = null; privado int size       // último nodo de la lista (o nulo si está vacío)
    = 0;                                            // número de nodos en la lista
SinglyLinkedList() { } // métodos de acceso         // construye una lista inicialmente vacía

público int tamaño () { devolver tamaño; }
    público booleano isEmpty() { devolver tamaño == 0; }
público E first() { si                              // devuelve (pero no elimina) el primer elemento
(estáVacío()) devuelve nulo;
devolver cabeza.getElement();
    }
público E último() {                                // devuelve (pero no elimina) el último elemento
        si (isEmpty()) devuelve nulo;
        devolver tail.getElement();
}
// métodos de actualización
    público void addFirst(E e) {                    // agrega el elemento e al frente de la lista
        cabeza = nuevo Nodo<>(e, cabeza);           // crear y vincular un nuevo nodo
        si (tamaño == 0)
            cola = cabeza;                          // caso especial: el nuevo nodo también se convierte en cola
        tamaño++;
}
    público void addLast(E e) {                     // agrega el elemento e al final de la lista
        Nodo<E> más nuevo = nuevo Nodo<>(e, null); // el nodo eventualmente será la cola
        si (estáVacío())
            cabeza = más nuevo; de lo               // caso especial: lista previamente vacía
        contrario
tail.setNext(más nuevo);                            // nuevo nodo después de la cola existente
nuevo;                                              // el nuevo nodo se convierte en la cola
tamaño++;

    }
público E removeFirst() {                           // elimina y devuelve el primer elemento
(estáVacío()) devuelve nulo;                        // nada que quitar
E respuesta =
cabeza.getElement();
        cabeza = cabeza.getNext();                  // se volverá nulo si la lista solo tiene un nodo
tamaño−−;
        si (tamaño == 0)
cola = nulo;                                        // Caso especial ya que la lista ahora está vacía
devolver
respuesta;
}
}
```

Fragmento de código 3.15: La definición de la clase SinglyLinkedList (cuando se combina con la clase Node anidada del Fragmento de Código 3.14).

3.3 Listas enlazadas circularmente

Tradicionalmente, las listas enlazadas se consideran como el almacenamiento de una secuencia de elementos en orden lineal, de principio a fin. Sin embargo, existen muchas aplicaciones en las que los datos pueden visualizarse de forma más natural como si tuvieran un orden cíclico, con relaciones vecinas bien definidas, pero sin un principio ni un final fijos.

Por ejemplo, muchos juegos multijugador son por turnos: el jugador A juega un turno, luego el jugador B, luego el jugador C, y así sucesivamente, pero finalmente regresa al jugador A y al jugador B, repitiéndose el patrón. Otro ejemplo: los autobuses y el metro urbanos suelen circular en un bucle continuo, con paradas programadas, pero sin una primera ni una última parada designadas. A continuación, consideraremos otro ejemplo importante de orden cíclico en el contexto de los sistemas operativos.

3.3.1 Programación por turnos

Una de las funciones más importantes de un sistema operativo es la gestión de los numerosos procesos activos en una computadora, incluyendo la programación de dichos procesos en una o más unidades centrales de procesamiento (CPU). Para garantizar la capacidad de respuesta de un número arbitrario de procesos concurrentes, la mayoría de los sistemas operativos permiten que los procesos compartan eficazmente el uso de las CPU mediante un algoritmo conocido como programación round-robin. A cada proceso se le asigna un breve turno para ejecutarse, conocido como franja de tiempo, pero se interrumpe al finalizar esta, incluso si su tarea aún no ha finalizado. Cada proceso activo recibe su propia franja de tiempo, turnándose en un orden cíclico. Se pueden añadir nuevos procesos al sistema y eliminar aquellos que completan su trabajo.

Se podría implementar un programador round-robin con una lista enlazada tradicional, mediante realizando repetidamente los siguientes pasos en la lista enlazada L (ver Figura 3.15):

1. proceso p = L.removeFirst()
2. Asigne una porción de tiempo al proceso p 3. L.addLast(p)

Desafortunadamente, el uso de una lista enlazada tradicional para este propósito presenta desventajas. Resulta innecesariamente ineficiente eliminar repetidamente un nodo de un extremo de la lista para luego crear uno nuevo para el mismo elemento al reinsertarlo, por no mencionar las diversas actualizaciones que se realizan para aumentar y disminuir el tamaño de la lista y para desvincular y volver a vincular nodos.

En el resto de esta sección, demostramos cómo se puede utilizar una ligera modificación en nuestra implementación de lista enlazada simple para proporcionar una estructura de datos más eficiente para representar un orden cíclico.

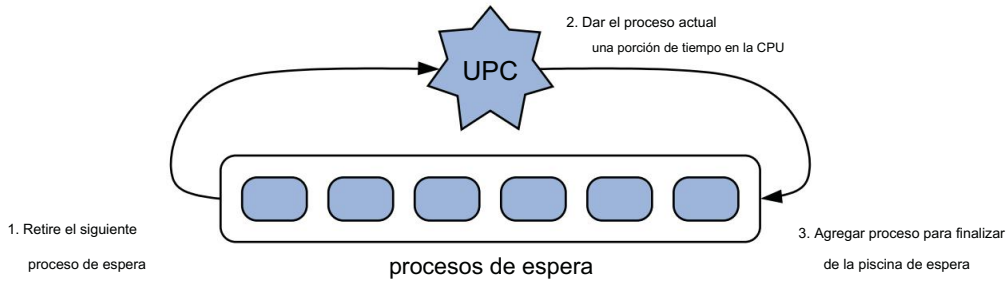


Figura 3.15: Los tres pasos iterativos para la programación round-robin.

3.3.2 Diseño e implementación de una lista enlazada circularmente

En esta sección, diseñamos una estructura conocida como lista enlazada circularmente, que es
Esencialmente, una lista enlazada singularmente en la que se establece la siguiente referencia del nodo de cola.
para hacer referencia al encabezado de la lista (en lugar de a nulo), como se muestra en la Figura 3.16.

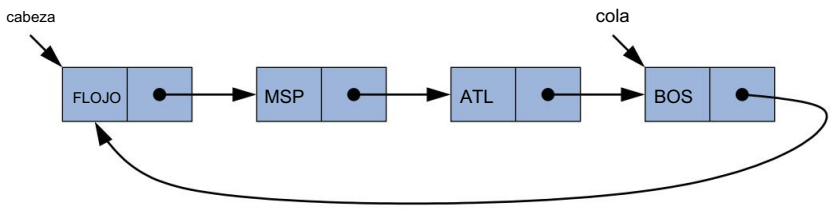


Figura 3.16: Ejemplo de una lista enlazada simple con estructura circular.

Usamos este modelo para diseñar e implementar una nueva clase CircularlyLinkedList,
que admite todos los comportamientos públicos de nuestra clase SinglyLinkedList y uno
método de actualización adicional:

rotate(): Mueve el primer elemento al final de la lista.

Con esta nueva operación, la programación por turnos se puede implementar de manera eficiente
realizando repetidamente los siguientes pasos en una lista enlazada circularmente C:

1. Asigne una porción de tiempo al proceso C.first()
2. C.rotate()

Optimización adicional

Al implementar una nueva clase, realizamos una optimización adicional: ya no
Mantenemos explícitamente la referencia principal. Siempre que mantengamos una referencia a la
cola, podemos localizar la cabeza como tail.getNext(). Manteniendo solo la referencia de la cola.
No solo ahorra un poco en el uso de memoria, sino que también hace que el código sea más simple y eficiente.
ya que elimina la necesidad de realizar operaciones adicionales para mantener una referencia de cabezal
actual. De hecho, nuestra nueva implementación es posiblemente superior a nuestra original.
Implementación de lista enlazada, incluso si no estamos interesados en el nuevo método de rotación.

Operaciones en una lista enlazada circularmente

Implementar el nuevo método de rotación es bastante sencillo. No movemos ningún nodo.
o elementos, simplemente avanzamos la referencia de cola para apuntar al nodo que sigue
(la cabecera implícita de la lista). La figura 3.17 ilustra esta operación utilizando un método más
Visualización simétrica de una lista enlazada circularmente.

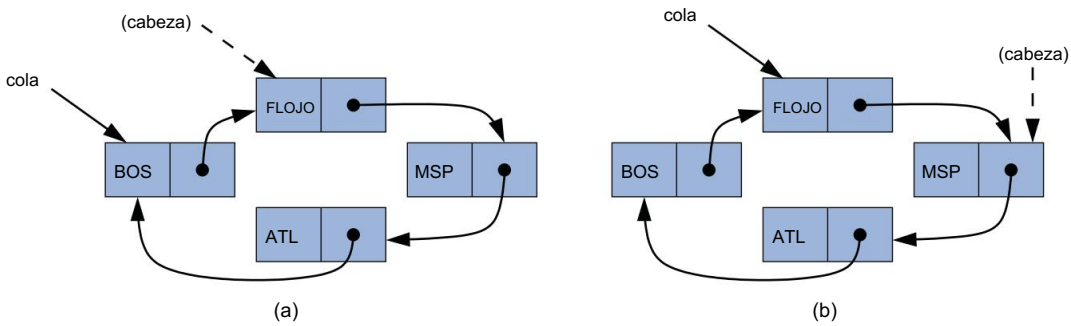


Figura 3.17: La operación de rotación en una lista enlazada circularmente: (a) antes de la rotación, representa la secuencia {LAX, MSP, ATL, BOS}; (b) después de la rotación, representa la secuencia {MSP, ATL, BOS, LAX}. Se muestra la referencia de la cabeza implícita, que se identifica únicamente como tail.getNext() en la implementación.

Podemos agregar un nuevo elemento al frente de la lista creando un nuevo nodo y vinculándolo justo después del final de la lista, como se muestra en la Figura 3.18. Para implementar el método addLast, podemos confiar en el uso de una llamada a addFirst y luego inmediatamente avanzar la referencia de cola para que el nodo más nuevo se convierta en el último.

La eliminación del primer nodo de una lista enlazada circularmente se puede lograr mediante simplemente actualizando el siguiente campo del nodo de cola para omitir el encabezado implícito. Un Java La implementación de todos los métodos de la clase CircularlyLinkedList se proporciona en el Código Fragmento 3.16.

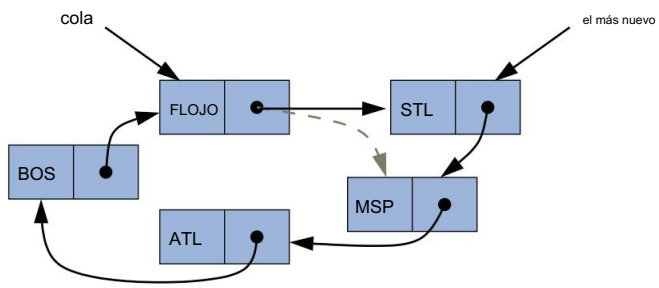


Figura 3.18: Efecto de una llamada a addFirst(STL) en la lista enlazada circular de la Figura 3.17(b). La variable "news" tiene alcance local durante la ejecución de la método. Observe que cuando se completa la operación, STL es el primer elemento de la lista, tal como se almacena dentro de la cabecera implícita, tail.getNext().

```
clase pública CircularlyLinkedList<E> {
    (clase de nodo anidada idéntica a la de la clase SinglyLinkedList)
    // variables de instancia de CircularlyLinkedList
privado Node<E> tail = null; privado int size = 0; público CircularlyLinkedList()
    { } // métodos de acceso

    público int tamaño() { devolver tamaño; }
public boolean isEmpty() { return size == 0; }
    público E primero()
    { si (estáVacío()) devuelve nulo;
devolver tail.getNext().getElement(); }

si  público E last() {
(estáVacío()) devuelve nulo;
devolver tail.getElement();
    }
// métodos de actualización
public void rotate() { if (tail != null)
        tail =
tail.getNext(); }

public void addFirst(E e) { if (tamaño
== 0) {
cola = nuevo Nodo<>(e, null);
de      cola.setNext(cola); }
lo contrario {
Nodo<E> más nuevo = nuevo Nodo<>(e, tail.getNext());
    tail.setNext(más nuevo);
}
}
    tamaño++;

public void addLast(E e) { addFirst(e);
    tail =
tail.getNext( ); }

público E removeFirst() { si
    (estáVacío()) devuelve nulo;
Nodo<E> cabeza = cola.getNext();
    si (cabeza == cola) cola = nulo; de lo
contrario cola.setNext(cabeza.getNext());
tamaño−

devolver cabeza.getElement();
    }
}
```

// Almacenamos la cola (pero no la cabeza)
// número de nodos en la lista
// construye una lista inicialmente vacía
// devuelve (pero no elimina) el primer elemento
// la cabeza está *después* de la cola
// devuelve (pero no elimina) el último elemento
// rotar el primer elemento al final de la lista
// si está vacío, no hacer nada
//la vieja cabeza se convierte en la nueva cola
// agrega el elemento e al frente de la lista
// enlace a sí mismo circularmente
// agrega el elemento e al final de la lista
// Insertar nuevo elemento al principio de la lista
// ahora el nuevo elemento se convierte en la cola
// elimina y devuelve el primer elemento
// nada que quitar
// debe ser el único nodo restante
// elimina “cabeza” de la lista

Fragmento de código 3.16: Implementación de la clase CircularlyLinkedList.

3.4 Listas doblemente enlazadas

En una lista enlazada simple, cada nodo mantiene una referencia al nodo inmediatamente posterior. Hemos demostrado la utilidad de esta representación cuando gestionar una secuencia de elementos. Sin embargo, existen limitaciones derivadas de La asimetría de una lista enlazada simple. En la Sección 3.2, demostramos que podemos insertar eficientemente un nodo en cualquier extremo de una lista enlazada simple y puede eliminar un nodo en el encabezado de una lista, pero no podemos eliminar de manera eficiente un nodo en la cola de la lista. De manera más general, no podemos eliminar de manera eficiente un nodo arbitrario de una posición interior de la lista si solo se nos da una referencia a ese nodo, porque no podemos determinar el nodo que precede inmediatamente al nodo que se va a eliminar (sin embargo, ese nodo debe tener su próxima referencia actualizada).

Para proporcionar una mayor simetría, definimos una lista enlazada en la que cada nodo mantiene una referencia explícita al nodo anterior y una referencia al nodo posterior. Tal Una estructura se conoce como lista doblemente enlazada. Estas listas permiten una mayor variedad de Operaciones de actualización en tiempo O(1), incluyendo inserciones y eliminaciones en posiciones arbitrarias dentro de la lista. Seguimos usando el término "siguiente" para referirnos a la nodo que sigue a otro, e introducimos el término “prev” para la referencia al nodo que lo precede.

Centinelas de encabezado y tráiler

Para evitar algunos casos especiales al operar cerca de los límites de un doble lista enlazada, ayuda a agregar nodos especiales en ambos extremos de la lista: un nodo de encabezado en el al principio de la lista y un nodo de tráiler al final. Estos nodos "ficticios" Se conocen como centinelas (o guardias) y no almacenan elementos del sistema primario. secuencia. En la Figura 3.19 se muestra una lista doblemente enlazada con dichos centinelas.

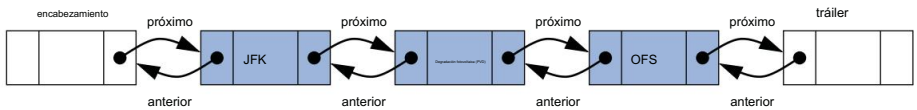


Figura 3.19: Una lista doblemente enlazada que representa la secuencia { JFK, PVD, SFO }, utilizando el encabezado y el tráiler de Sentinels para delimitar los extremos de la lista.

Al utilizar nodos centinela, se inicializa una lista vacía para que el siguiente campo del encabezado apunta al tráiler, y el campo anterior del tráiler apunta al encabezado; los campos restantes de los centinelas son irrelevantes (presumiblemente nulos, en Java). Para una lista no vacía, el siguiente del encabezado hará referencia a un nodo que contiene el primer elemento real de una secuencia, al igual que el anterior del tráiler hace referencia al nodo que contiene el último elemento de una secuencia.

Ventajas de usar centinelas

Aunque podríamos implementar una lista doblemente enlazada sin nodos centinela (como
Como hicimos con nuestra lista enlazada simple en la Sección 3.2), la ligera memoria adicional dedicada a la
Sentinels simplifica enormemente la lógica de nuestras operaciones. En particular, el encabezado y
Los nodos de remolque nunca cambian, solo cambian los nodos entre ellos. Además,
Podemos tratar todas las inserciones de manera unificada, porque siempre habrá un nuevo nodo.
colocado entre un par de nodos existentes. De manera similar, cada elemento que se va a
Se garantiza que lo eliminado se almacenará en un nodo que tenga vecinos en cada lado.

A modo de contraste, observamos nuestra implementación de SinglyLinkedList de la Sección 3.2.
Su método addLast requería un condicional (líneas 39-42 del Fragmento de Código 3.15) para
Gestionar el caso especial de inserción en una lista vacía. En el caso general, el nuevo
El nodo se vinculó después de la cola existente. Pero al agregarlo a una lista vacía, hay
no existe cola, en su lugar es necesario reasignar la cabeza para hacer referencia al nuevo nodo.
El uso de un ganglio centinela en esa implementación eliminaría el caso especial,
ya que siempre habría un nodo existente (posiblemente el encabezado) antes de un nuevo nodo.

Inserción y eliminación con una lista doblemente enlazada

Cada inserción en nuestra representación de lista doblemente enlazada tendrá lugar entre
un par de nodos existentes, como se diagrama en la Figura 3.20. Por ejemplo, cuando un nuevo
Si el elemento se inserta al principio de la secuencia, simplemente agregaremos el nuevo nodo.
Entre el encabezado y el nodo que se encuentra actualmente después del encabezado. (Ver Figura 3.21).

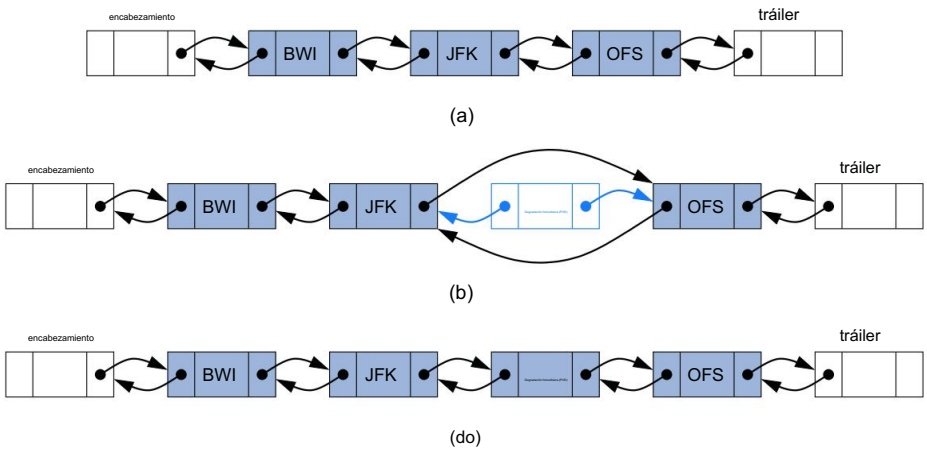


Figura 3.20: Adición de un elemento a una lista doblemente enlazada con centinelas de encabezado y final:
(a) antes de la operación; (b) después de crear el nuevo nodo; (c) después de enlazar el
vecinos del nuevo nodo.

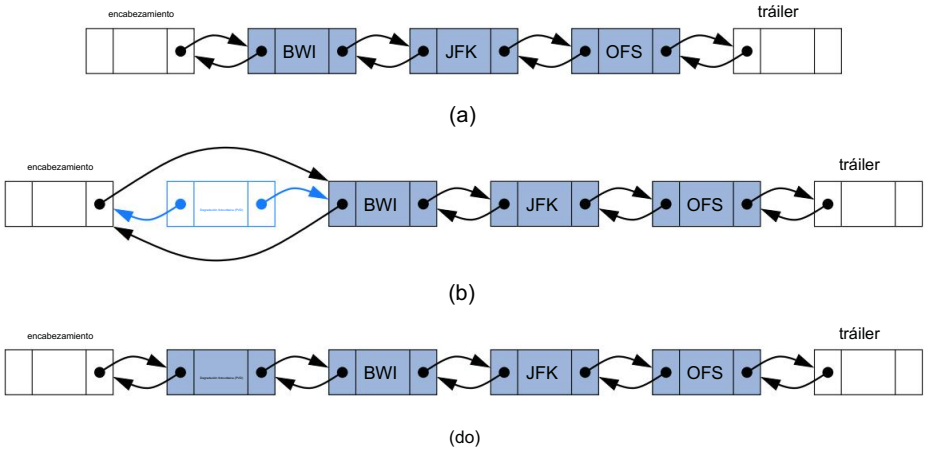


Figura 3.21: Adición de un elemento al frente de una secuencia representada por una lista doblemente enlazada con centinelas de encabezado y final: (a) antes de la operación; (b) después creando el nuevo nodo; (c) después de vincular los vecinos al nuevo nodo.

La eliminación de un nodo, representada en la Figura 3.22, se realiza de forma opuesta a una inserción. Los dos vecinos del nodo que se va a eliminar están vinculados directamente. entre sí, omitiendo así el nodo original. Como resultado, ese nodo ya no... ya no se considerará parte de la lista y podrá ser reclamado por el sistema. Porque De nuestro uso de centinelas, se puede utilizar la misma implementación al eliminar el primer o el último elemento de una secuencia, porque incluso dicho elemento se almacenará en un nodo que se encuentra entre otros dos.

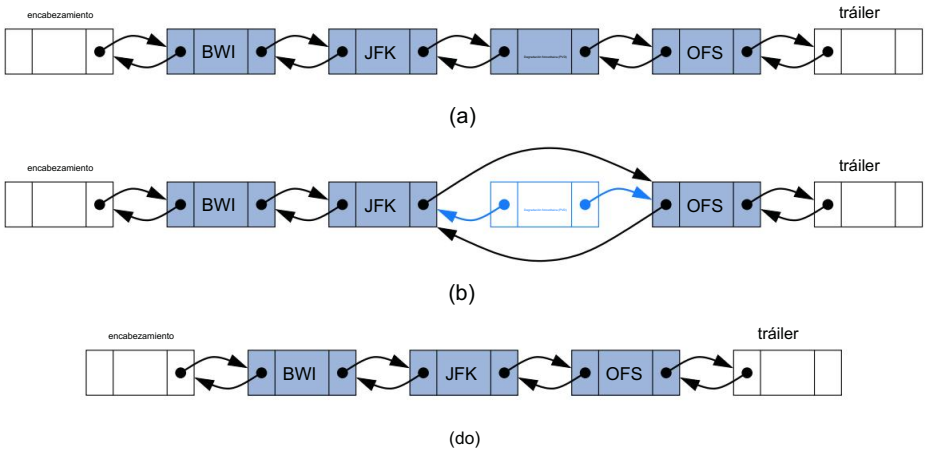


Figura 3.22: Eliminación del elemento PVD de una lista doblemente enlazada: (a) antes la eliminación; (b) después de vincular el nodo antiguo; (c) después de la eliminación (y la basura) recopilación).

3.4.1 Implementación de una clase de lista doblemente enlazada

En esta sección, presentamos una implementación completa de una clase DoublyLinkedList, que admite los siguientes métodos públicos:

tamaño(): Devuelve el número de elementos en la lista.

isEmpty( ): devuelve verdadero si la lista está vacía y falso en caso contrario.

first( ): Devuelve (pero no elimina) el primer elemento de la lista. last( ): Devuelve (pero no elimina) el último elemento de la lista.

addFirst(e): agrega un nuevo elemento al frente de la lista.

addLast(e): agrega un nuevo elemento al final de la lista.

removeFirst( ): elimina y devuelve el primer elemento de la lista. removeLast( ): elimina y devuelve el último elemento de la lista.

Si se llaman a first(), last(), removeFirst() o removeLast() en una lista que está vacía, devolveremos una referencia nula y dejaremos la lista sin cambios.

Aunque hemos visto que es posible añadir o eliminar un elemento en una posición interna de una lista doblemente enlazada, esto requiere el conocimiento de uno o más nodos para identificar la posición donde debe realizarse la operación. En este capítulo, preferimos mantener la encapsulación mediante una clase Node privada y anidada. En el capítulo 7, revisaremos el uso de listas doblemente enlazadas, ofreciendo una interfaz más avanzada que admite inserciones y eliminaciones internas, manteniendo la encapsulación.

Los fragmentos de código 3.17 y 3.18 presentan la implementación de la clase ListaDoblementeEnlazada. Al igual que con la clase ListaSinglyEnlazada, utilizamos el marco de genéricos para aceptar cualquier tipo de elemento. La clase Nodo anidada para la lista doblemente enlazada es similar a la de la lista simple, excepto que admite una referencia "prev" adicional al nodo anterior.

El uso de nodos centinela, encabezado y tráiler, afecta la implementación de varias maneras. Creamos y enlazamos los centinelas al construir una lista vacía (líneas 25-29). También tenemos en cuenta que el primer elemento de una lista no vacía se almacena en el nodo inmediatamente después del encabezado (no en el encabezado mismo), y de forma similar, el último elemento se almacena en el nodo inmediatamente antes del tráiler.

Los centinelas facilitan enormemente la implementación de los distintos métodos de actualización. Proporcionaremos un método privado, addBetween, para gestionar el caso general de una inserción, y luego usaremos esta utilidad como un método sencillo para implementar addFirst y addLast. De forma similar, definiremos un método privado de eliminación que permite implementar fácilmente removeFirst y removeLast.

```
1 /      Una implementación básica de lista doblemente enlazada.    /
2 clase pública DoublyLinkedList<E> {
      //---------- Clase de nodo anidada ----------
4
3 clase estática privada Node<E> {
5         elemento E privado; // referencia al elemento almacenado en este nodo
6         private Node<E> prev; //                    // referencia al nodo anterior en la lista
7         Nodo privado<E> siguiente; referencia al nodo subsiguiente en la lista
8         público(E e, Nodo<E> p, Nodo<E> n) {
9            elemento = e;
10           anterior = p;
11           siguiente = n;
12        }
13        public E getElement() { return elemento; }
14        público Nodo<E> getPrev() { devolver prev; }
15        público Nodo<E> getNext() { devolver siguiente; }
16        público void setPrev(Nodo<E> p) { prev = p; }
17        público void setNext(Nodo<E> n) { siguiente = n; }
18    } //---------- fin de la clase Node anidada ----------
19
20    // variables de instancia de la lista doblemente enlazada
21    encabezado privado Node<E> ;                    // centinela de encabezado
22    tráiler privado Node<E>; tamaño                 // centinela del tráiler
23    int privado = 0; /                              // número de elementos en la lista
24    Construye una nueva lista vacía.    /
25    pública DoblementeEnlazada( ) {
26       encabezado = nuevo Nodo<>(nulo, nulo, nulo); tráiler =   // crear encabezado
27       nuevo Nodo<>(nulo, encabezado, nulo);           // el tráiler está precedido por el encabezado
28       encabezado.setNext(tráiler); }                  // El encabezado es seguido por el tráiler
29
30    /      Devuelve el número de elementos en la lista enlazada.    /
31    público int tamaño() { devolver tamaño; }
32    /      Comprueba si la lista enlazada está vacía.    /
33    público booleano isEmpty() { devolver tamaño == 0; }
34    /      Devuelve (pero no elimina) el primer elemento de la lista.    /
35    público E primero( ) {
36       si (isEmpty()) devuelve nulo;
37       devolver encabezado.getNext().getElement(); }   // El primer elemento está más allá del encabezado
38
39    /      Devuelve (pero no elimina) el último elemento de la lista.    /
40    público E último() {
41       si (isEmpty()) devuelve nulo;
42       devolver trailer.getPrev() .getElement() ; }    // El último elemento está antes del tráiler.
43
```

Fragmento de código 3.17: Implementación de la clase DoublyLinkedList. (Continúa en Fragmento de código 3.18.)

```
44 // métodos de actualización públicos
45 /      Agrega el elemento e al principio de la lista.    /
46 público void addFirst(E e) {
            addBetween(e, encabezado, encabezado.getNext());          // colocar justo después del encabezado
47 }
48 /      Agrega el elemento e al final de la lista.    /
50 público void addLast(E e) {
51 addBetween(e, trailer.getPrev(), trailer); 52 }                    // colocar justo antes del tráiler

53      /      Elimina y devuelve el primer elemento de la lista.    /
54 público E removeFirst( ) {
            si (isEmpty()) devuelve nulo; 55                          // nada que quitar
devuelve remove(header.getNext()); 56 57 }                            // El primer elemento está más allá del encabezado

58 /      Elimina y devuelve el último elemento de la lista.    /
59 público E removeLast() {
            si (isEmpty()) devuelve nulo; 60                          // nada que quitar
devuelve remove(trailer.getPrev()); 61 62 }                           // El último elemento está antes del tráiler.

63
64 // métodos de actualización privados
65 /      Agrega el elemento e a la lista enlazada entre los nodos dados.    /
66 private void addBetween(E e, Nodo<E> predecesor, Nodo<E> sucesor) {
67          // crear y vincular un nuevo nodo
68 Nodo<E> más nuevo = nuevo Nodo<>(e, predecesor, sucesor);
69 predecesor.setNext(más nuevo);
70 successor.setPrev(más nuevo);
71 tamaño++;
72 }
73      /      Elimina el nodo dado de la lista y devuelve su elemento.    /
74 privado E remove(Nodo<E> nodo) {
75          Nodo<E> predecesor = nodo.getPrev();
76          Nodo<E> sucesor = nodo.getNext( );
77          predecesor.setNext(sucesor);
78          sucesor.setPrev(predecesor);
79          tamaño−−;
80          retorna nodo.getElement();
81      }
82 } //---------- fin de la clase DoublyLinkedList ----------
```

Fragmento de código 3.18: Implementación de los métodos de actualización públicos y privados para La clase DoublyLinkedList. (Continuación del fragmento de código 3.17).

3.5 Pruebas de equivalencia

Al trabajar con tipos de referencia, existen diversas nociones sobre lo que significa que una expresión sea igual a otra. En el nivel más básico, si a y b son variables de referencia, la expresión a == b comprueba si a y b hacen referencia al mismo objeto (o si ambos tienen el valor nulo).

Sin embargo, para muchos tipos existe una noción de nivel superior: dos variables se consideran "equivalentes" incluso si no se refieren a la misma instancia de la clase. Por ejemplo, normalmente consideramos dos instancias de String como equivalentes si representan la misma secuencia de caracteres.

Para respaldar una noción más amplia de equivalencia, todos los tipos de objeto admiten un método denominado equals. Los usuarios de tipos de referencia deben usar la sintaxis a.equals(b), a menos que necesiten comprobar específicamente la noción más específica de identidad. El método equals se define formalmente en la clase Object, que actúa como superclase para todos los tipos de referencia, pero dicha implementación devuelve el valor de la expresión a == b. Definir una noción de equivalencia más significativa requiere conocimientos sobre una clase y su representación.

El autor de cada clase tiene la responsabilidad de proporcionar una implementación del método equals, que sobrescribe el heredado de Object, si existe una definición más relevante para la equivalencia de dos instancias. Por ejemplo, la clase String de Java redefine equals para comprobar la equivalencia carácter por carácter.

Se debe tener mucho cuidado al anular la noción de igualdad, ya que la consistencia de las bibliotecas de Java depende de que el método equals defina lo que se conoce como una relación de equivalencia en matemáticas, satisfaciendo las siguientes propiedades: Tratamiento de null: para cualquier variable de referencia no nula x, la llamada x.equals(null) debe devolver falso (es decir, nada es igual a null excepto null).

Reflexividad: para cualquier variable de referencia no nula x, la llamada x.equals(x) debe devolver verdadero (es decir, un objeto debe ser igual a sí mismo).

Simetría: para cualquier variable de referencia no nula x e y, las llamadas x.equals(y) e y.equals(x) deben devolver el mismo valor.

Transitividad: para cualquier variable de referencia no nula x, y, z, si ambas llamadas x.equals(y) e y.equals(z) devuelven verdadero, entonces la llamada x.equals(z) también debe devolver verdadero.

Aunque estas propiedades puedan parecer intuitivas, implementar correctamente el método equals en algunas estructuras de datos puede resultar complicado, especialmente en un contexto orientado a objetos, con herencia y genéricos. En la mayoría de las estructuras de datos de este libro, omitimos la implementación de un método equals válido (dejándolo como ejercicio). Sin embargo, en esta sección, consideramos el tratamiento de las pruebas de equivalencia tanto para arrays como para listas enlazadas, incluyendo un ejemplo concreto de una implementación correcta del método equals para nuestra clase SinglyLinkedList.

3.5.1 Pruebas de equivalencia con matrices

Como mencionamos en la Sección 1.3, los arrays son un tipo de referencia en Java, pero técnicamente no son una clase. Sin embargo, la clase java.util.Arrays, presentada en la Sección 3.1.3, Proporciona métodos estáticos adicionales que son útiles al procesar matrices. A continuación, se presenta un resumen del tratamiento de la equivalencia para matrices, suponiendo que las variables a y b hacen referencia a objetos de matriz:

a == b: prueba si a y b hacen referencia a la misma instancia de matriz subyacente.

a.equals(b): Curiosamente, esto es idéntico a a == b. Las matrices no son una tipo de clase verdadero y no anule el método Object.equals.

Arrays.equals(a,b): Esto proporciona una noción más intuitiva de equivalencia, devolviendo verdadero si las matrices tienen la misma longitud y todos los pares de elementos correspondientes son “iguales” entre sí. Más específicamente, si los elementos de la matriz son primitivos, entonces utiliza El estándar == para comparar valores. Si los elementos del array-ray son de tipo de referencia, se realizan comparaciones por pares con a[k].equals(b[k]) para evaluar la equivalencia.

Para la mayoría de las aplicaciones, el comportamiento de Arrays.equals captura el concepto de equivalencia. Sin embargo, existe una complicación adicional al usar Matrices multidimensionales. El hecho de que las matrices bidimensionales en Java sean realmente... Las matrices unidimensionales anidadas dentro de una matriz unidimensional común plantean una cuestión interesante con respecto a cómo pensamos acerca de los objetos compuestos, que son Objetos —como una matriz bidimensional— que se componen de otros objetos. En particular, plantea la cuestión de dónde empieza y termina un objeto compuesto.

Por lo tanto, si tenemos una matriz bidimensional, a, y otra matriz bidimensional, b, que tiene las mismas entradas que a, probablemente queramos pensar que a es igual a b. Pero las matrices unidimensionales que conforman las filas de a y b (como a[0] y b[0]) se almacenan en diferentes ubicaciones de memoria, aunque tienen la El mismo contenido interno. Por lo tanto, una llamada al método java.util.Arrays.equals(a,b) devolverá falso en este caso, porque prueba a[k].equals(b[k]), que invoca la Definición de iguales de la clase de objeto.

Para apoyar la noción más natural de que las matrices multidimensionales son iguales si tienen contenidos iguales, la clase proporciona un método adicional:

Arrays.deepEquals(a,b): idéntico a Arrays.equals(a,b) excepto cuando los elementos de a y b son en sí mismos matrices, en cuyo caso se llama Arrays.deepEquals(a[k],b[k]) para las entradas correspondientes, en lugar de a[k].equals(b[k]).

3.5.2 Pruebas de equivalencia con listas enlazadas

En esta sección, desarrollamos una implementación del método equals en el contexto de la clase SinglyLinkedList de la Sección 3.2.1. Utilizando una definición muy similar a la Tratamiento de matrices mediante el método java.util.Arrays.equals, consideramos dos listas para Serían equivalentes si tienen la misma longitud y contenido elemento por elemento. equivalente. Podemos evaluar dicha equivalencia recorriendo simultáneamente dos listas, verificando que x.equals(y) para cada par de elementos correspondientes x e y.

La implementación del método SinglyLinkedList.equals se proporciona en el Código Fragmento 3.19. Aunque nos centramos en comparar dos listas enlazadas simples, la El método equals debe tomar un objeto arbitrario como parámetro. Tomamos un método conservador. enfoque, exigiendo que dos objetos sean instancias de la misma clase para tener cualquier posibilidad de equivalencia. (Por ejemplo, no consideramos una lista simplemente enlazada para sería equivalente a una lista doblemente enlazada con la misma secuencia de elementos.) Después Para garantizar, en la línea 2, que el parámetro o no sea nulo, la línea 3 utiliza el método getClass() compatible con todos los objetos para probar si las dos instancias pertenecen a la misma clase.

Al llegar a la línea 4, nos aseguramos de que el parámetro fuera una instancia de la clase SinglyLinkedList (o una subclase apropiada), por lo que podemos convertir de forma segura a una SinglyLinkedList, para que podamos acceder a sus variables de instancia size y head. Hay cierta sutileza en el tratamiento del marco de genéricos de Java. Aunque Nuestra clase SinglyLinkedList tiene un parámetro de tipo formal declarado <E>, no podemos Detectar en tiempo de ejecución si la otra lista tiene un tipo coincidente. (Para aquellos interesados, (Busque en línea una discusión sobre el borrado en Java). Así que volvemos a usar un método más clásico. enfoque con el tipo no parametrizado SinglyLinkedList en la línea 4 y declaraciones de nodo no parametrizadas en las líneas 6 y 7. Si las dos listas tienen tipos incompatibles, Esto se detectará al llamar al método equals en los elementos correspondientes.

```
público booleano igual a (Objeto o) {
1    si (o == null) devuelve falso;
2    si (getClass() != o.getClass()) devuelve falso;
3    SinglyLinkedList other = (SinglyLinkedList) o; si (tamaño !        // utilizar tipo no parametrizado
4 5  = otro.tamaño) devuelve falso;
6    Nodo walkA = cabeza;                                               // recorrer la lista principal
7    Nodo walkB = otro.cabeza;                                          // recorrer la lista secundaria
     mientras (walkA != null) {
8        si (!walkA.getElement( ).equals(walkB.getElement( ))) devuelve falso; //no coincide
9        caminarA = caminarA.getNext( );
10 11    caminarB = caminarB.getNext( );
12   }
13   devuelve verdadero; // si alcanzamos esto, todo coincidió exitosamente
14 }
```

Fragmento de código 3.19: Implementación del método SinglyLinkedList.equals.

3.6 Clonación de estructuras de datos

La belleza de la programación orientada a objetos es que la abstracción permite que una estructura de datos sea tratada como un solo objeto, aunque la implementación encapsulada de la estructura pueda depender de una combinación más compleja de muchos objetos.
En esta sección, consideramos lo que significa hacer una copia de dicha estructura.

En un entorno de programación, se espera comúnmente que una copia de un objeto tenga su propio estado y que, una vez creada, sea independiente del original (por ejemplo, para que los cambios en uno no afecten directamente al otro). Sin embargo, cuando los objetos tienen campos que son variables de referencia que apuntan a objetos auxiliares, no siempre es evidente si una copia debe tener un campo correspondiente que haga referencia al mismo objeto auxiliar o a una nueva copia de dicho objeto.

Por ejemplo, si una clase hipotética AddressBook tiene instancias que representan una libreta de direcciones electrónica (con información de contacto, como números de teléfono y direcciones de correo electrónico, de los amigos y conocidos de una persona), ¿cómo podríamos imaginar una copia de la libreta de direcciones? ¿Debería aparecer una entrada añadida a una libreta en la otra? Si cambiamos el número de teléfono de una persona en un libro, ¿esperaríamos que ese cambio se sincronice en el otro?

No existe una respuesta universal para preguntas como esta. En cambio, cada clase en Java es responsable de definir si sus instancias se pueden copiar y, de ser así, cómo se construye la copia. La superclase universal Object define un método llamado clone, que permite generar una copia superficial de un objeto. Esta utiliza la semántica de asignación estándar para asignar el valor de cada campo del nuevo objeto al campo correspondiente del objeto existente que se está copiando. Esto se conoce como copia superficial porque, si el campo es un tipo de referencia, una inicialización con la forma duplicate.field = original.field hace que el campo del nuevo objeto haga referencia a la misma instancia subyacente que el campo del objeto original.

Una copia superficial no siempre es apropiada para todas las clases; por lo tanto, Java deshabilita intencionalmente el uso del método clone() al declararlo como protegido y al hacer que genere una excepción CloneNotSupportedException al ser llamado. El autor de una clase debe declarar explícitamente la compatibilidad con la clonación, declarando formalmente que la clase implementa la interfaz Cloneable y una versión pública del método clone(). Este método público puede simplemente llamar al método protegido para realizar la asignación campo por campo que resulta en una copia superficial, si corresponde. Sin embargo, para muchas clases, la clase puede optar por implementar una versión más avanzada de la clonación, en la que se clonan algunos de los objetos referenciados.

Para la mayoría de las estructuras de datos de este libro, omitimos la implementación de un método de clonación válido (dejándolo como ejercicio). Sin embargo, en esta sección, consideramos enfoques para clonar tanto arrays como listas enlazadas, incluyendo una implementación concreta del método de clonación para la clase SinglyLinkedList.

3.6.1 Clonación de matrices

Aunque las matrices admiten algunas sintaxis especiales como a[k] y a.length, es importante recordar que son objetos y que las variables de matriz son referencias.
Variables. Esto tiene consecuencias importantes. Como primer ejemplo, considere el siguiente código:

```
int[ ] datos = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] copia de seguridad;
copia de seguridad = datos;                    // advertencia; no es una copia
```

La asignación de una copia de seguridad variable a los datos no crea ninguna matriz nueva; simplemente crea un nuevo alias para la misma matriz, como se muestra en la Figura 3.23.

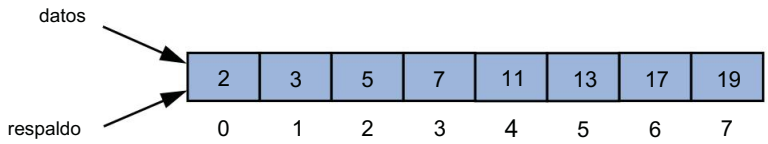


Figura 3.23: El resultado del comando backup = data para matrices int.

En cambio, si queremos hacer una copia de la matriz, datos y asignar una referencia a Para la nueva matriz a variable, copia de seguridad, debemos escribir:

```
copia de seguridad = datos.clone();
```

El método clone, cuando se ejecuta en una matriz, inicializa cada celda de la nueva matriz. al valor almacenado en la celda correspondiente de la matriz original. Esto da como resultado en una matriz independiente, como se muestra en la Figura 3.24.

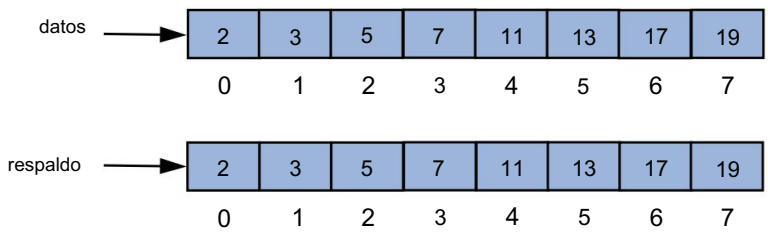


Figura 3.24: El resultado del comando backup = data.clone() para matrices int.

Si posteriormente realizamos una asignación como data[4] = 23 en esta configuración, La matriz de respaldo no se ve afectada.

Hay más consideraciones al copiar una matriz que almacena referencias tipos en lugar de tipos primitivos. El método clone() produce una copia superficial. de la matriz, produciendo una nueva matriz cuyas celdas hacen referencia a los mismos objetos referenciados por la primera matriz.

Por ejemplo, si la variable contactos se refiere a una matriz de personas hipotéticas
En algunas instancias, el resultado del comando guest = contacts.clone() produce una copia superficial, como se muestra en la Figura 3.25.

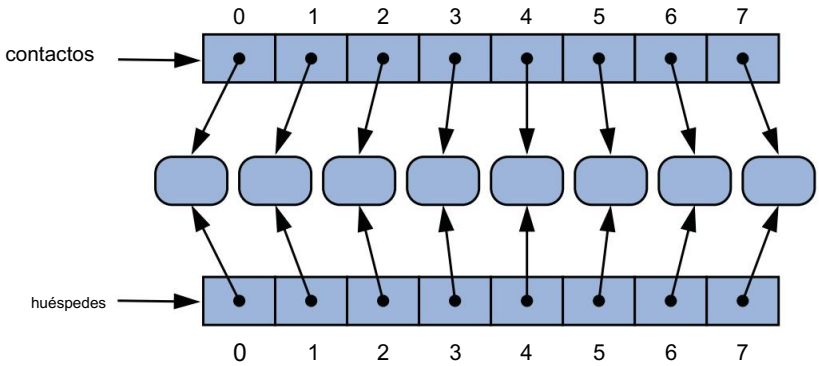


Figura 3.25: Una copia superficial de una matriz de objetos, resultante del comando invitados = contactos.clone( ).

Se puede crear una copia profunda de la lista de contactos clonando iterativamente los elementos individuales, de la siguiente manera, pero solo si la clase Persona se declara como Clonable.

```
Persona[ ] invitados = new Persona[contactos.length];
para (int k=0; k < contactos.longitud; k++)
   invitados[k] = (Persona) contactos[k].clone( );          // devuelve el tipo de objeto
```

Porque una matriz bidimensional es en realidad una matriz unidimensional que almacena otras matrices unidimensionales, la misma distinción entre una copia superficial y profunda Existe. Desafortunadamente, la clase java.util.Arrays no proporciona ningún "deepClone". método. Sin embargo, podemos implementar nuestro propio método clonando el individuo filas de una matriz, como se muestra en el Fragmento de Código 3.20, para una matriz bidimensional de números enteros.

```
1 público estático int[ ][ ] deepClone(int[ ][ ] original) {
2     int[ ][ ] backup = nuevo int[longitud original][ ]; para     // crea una matriz de matrices de nivel superior
3     (int k=0; k < longitud original; k++)
4         copia de seguridad[k] = original[k].clone();             // copiar la fila k
5 devolver copia de seguridad;
6 }
```

Fragmento de código 3.20: Un método para crear una copia profunda de una matriz bidimensional de números enteros.

3.6.2 Clonación de listas enlazadas

En esta sección, agregamos soporte para clonar instancias de la clase SinglyLinkedList
De la Sección 3.2.1. El primer paso para que una clase sea clonable en Java es declarar
que implementa la interfaz Cloneable. Por lo tanto, ajustamos la primera línea del
La definición de clase aparecerá de la siguiente manera:

```
clase pública SinglyLinkedList<E> implementa Cloneable {
```

La tarea restante es implementar una versión pública del método clone() de
la clase, que presentamos en el fragmento de código 3.21. Por convención, ese método
debe comenzar creando una nueva instancia usando una llamada a super.clone(), que en nuestro
El caso invoca el método de la clase Object (línea 3). Dado que la versión heredada devuelve un Object,
realizamos una conversión de tipo SinglyLinkedList<E>.

En este punto de la ejecución, la otra lista se ha creado como una copia superficial.
del original. Dado que nuestra clase de lista tiene dos campos, tamaño y encabezado, lo siguiente
Se han realizado las siguientes asignaciones:

```
otro.tamaño = este.tamaño;
otro.cabeza = este.cabeza;
```

Si bien la asignación de la variable de tamaño es correcta, no podemos permitir que la nueva lista
Comparten el mismo valor de cabecera (a menos que sea nulo). Para que una lista no vacía tenga un estado
independiente, debe tener una cadena de nodos completamente nueva, cada una almacenando una referencia.
al elemento correspondiente de la lista original. Por lo tanto, creamos una nueva cabecera.
nodo en la línea 5 del código y luego realizar un recorrido por el resto del
lista original (líneas 8 a 13) mientras se crean y vinculan nuevos nodos para la nueva lista.

```
1  público SinglyLinkedList<E> clone() lanza CloneNotSupportedException {
2    // utilice siempre Object.clone() heredado para crear la copia inicial
3    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // conversión segura
4    si (tamaño > 0) { // necesitamos una cadena de nodos independiente
5      otro.cabeza = nuevo Nodo<>(cabeza.getElement( ), null);
6      Node<E> walk = head.getNext( ); // recorre el resto de la lista original
7      Node<E> otherTail = other.head; // recordar el nodo creado más recientemente
8      mientras (walk != null) { // crea un nuevo nodo que almacena el mismo elemento
9        Nodo<E> más nuevo = nuevo Nodo<>(walk.getElement( ), null);
10       otherTail.setNext(más nuevo);                    // vincula el nodo anterior a este
11       otherTail = más nuevo;
12       caminar = caminar.getNext( );
13     }
14   }
15   devolver otro;
16 }
```

Fragmento de código 3.21: Implementación del método SinglyLinkedList.clone.