Estructuras de Datos con Java

Héctor Tejeda V

Abril, 2010

Índice general

Capítulo 1

Introducción

1.1. Que son las estructuras de datos

Una *estructura de datos* es un arreglo de datos en la memoria de una computadora, y en algunas ocasiones en un disco. Las estructuras de datos incluyen arreglos, listas enlazadas, pilas, árboles binarios, y tablas de dispersión entre otros. Los *algoritmos* manipulan los datos en estas estructuras de varias formas, para buscar un dato particular u ordenar los datos.

1.2. Generalidades de las Estructuras de Datos

Una forma de ver las estructuras de datos es enfocándose en sus fortalezas y debilidades. En el cuadro 1.2 se muestran las ventajas y desventajas de las estructuras de datos que se revisarán, los términos que se usan serán abordados en otros capítulos.

Las estructuras mostradas en el cuadro 1.2, excepto los arreglos, pueden ser pensados como *tipos de datos abstractos* o ***Abstract Data Type*** (ADT). Un ADT, de forma general, es una forma de ver una estructura de datos: enfocándose en lo que esta hace e ignorando como hace su trabajo, es decir, el ADT deberá cumplir con ciertas propiedades, pero la manera como estará implementado puede variar, aún empleando el mismo lenguaje. Por ejemplo, el ATD pila puede ser implementado con un arreglo o bien con una lista enlazada.

Varios de los algoritmos que serán discutidos en este material se aplicarán directamente a estructuras de datos específicas. Para la mayoría de las estructuras de datos, se requiere que hagan las siguientes tareas:

1. Insertar un nuevo dato.
2. Buscar un elemento indicado.
3. Borrar un elemento indicado.

Estructura	**Ventajas**	**Desventajas**
Arreglo	Inserción rápida, acceso muy rápido si se conoce el índice.	Búsqueda y borrado lento, tamaño fijo.
Arreglo ordenado	Búsqueda más rápida que el arreglo desordenado.	Inserción y borrado lento, tamaño fijo.
Pila	Proporciona acceso último en entrar, primero en salir.	Acceso lento a otros elementos.
Cola	Proporciona primero en entrar, primero en salir.	Acceso lento a otros elementos.
Lista enlazada	Inserción y borrado rápido.	Búsqueda lenta.
Árbol binario	Búsqueda, inserción y borrado rápido si el árbol permanece balanceado.	Complejo.
Árbol rojinegro	Búsqueda, inserción y borrado rápido. Árbol siempre balanceado.	Complejo.
Árbol 2-3-4	Búsqueda, inserción y borrado rápido. Árbol siempre balanceado.	Complejo.
Tabla de dispersión	Acceso muy rápido si se conoce la llave. Inserción rápida.	Borrado lento, acceso lento si no se conoce la llave, uso ineficiente de la memoria.
Montículo	Inserción, borrado y acceso al elemento más grande rápido	Acceso lento a otros elementos.
Grafo	Modela situaciones del mundo real	Algunos algoritmos son lentos y complejos.

Cuadro 1.1: Características de las Estructuras de Datos

También se necesitará saber como *iterar* a través de todos los elementos en la estructura de datos, visitando cada uno en su turno ya sea para mostrarlo o para realizar alguna acción en este.

Otro algoritmo importante es el *ordenamiento*. Hay varias formas para ordenar los datos, desde ideas simples hasta otras que permiten realizarlo de manera rápida.

El concepto de *recurrencia* es importante en el diseño de ciertos algoritmos. La recurrencia involucra que un método se llame a si mismo.

Capítulo 2

Arreglos, listas enlazadas y recurrencia

2.1. Arreglos

En esta sección, se exploran unas cuantas aplicaciones con arreglos.

2.1.1. Guardar entradas de juego en un arreglo

La primera aplicación que será revisada es para guardar las entradas en un arreglo—en particular, las mejores marcas para un videojuego. Los arreglos también se podrían usar para guardar registros de pacientes en un hospital o los nombres de estudiantes.

En primer lugar se debe determinar lo que se quiere incluir en el registro de una puntuación alta. Por supuesto, un componente que se deberá incluir es un entero representando los puntos obtenidos, al cual llamaremos `puntuación`. Otra característica que deberá incluirse es el nombre de la persona que obtuvo la puntuación, la cual denominaremos `nombre`.

```
1  public class EntradaJuego {
2
3     protected String nombre; // nombre de la persona que gano esta puntuación
4     protected int puntuacion;  // el valor de la puntuación
5
6     /** Constructor para crear una entrada del juego */
7     public EntradaJuego(String n, int p) {
8        nombre = n;
9        puntuacion = p;
10    }
11
12    /** Recupera el campo nombre */
13    public String getNombre() { return nombre; }
14    /** Recupera el campo puntuación */
15    public int getPuntuacion() { return puntuacion; }
16    /** Regresa una cadena representando esta entrada */
17    public String toString() {
18       return "(" + nombre + ", " + puntuacion + ")";
19    }
```

```
20 }
```

Listado 2.1: Clase `EntradaJuego.java`

Clase para las mayores puntuaciones

En el listado 2.2 se implementa lo que se detalla en los siguientes miembros de la clase `Puntuaciones`. La clase es usada para guardar un determinado número de puntuaciones altas. Adicionalmente se consideran métodos que permiten modificar una instanciación de esta clase, ya sea para agregar nuevos récords, o bien, para quitar algún récord no válido.

```
1  /** Clase para guardar los récords en un arreglo en orden descendente. */
2  public class Puntuaciones {
3
4    public static final int maxEntradas = 10; // núm. de records que se guardan
5    protected int numEntradas;                // núm. actual de entradas
6    protected EntradaJuego[] entradas;  // arreglo de entradas (nomb. y puntuac.)
7
8    /** Constructor por defecto */
9    public Puntuaciones() {
10     entradas = new EntradaJuego[maxEntradas];
11     numEntradas = 0;
12   }
13
14   /** Regresa una cadena de las mayores puntuaciones */
15   public String toString() {
16     String s = "[";
17     for (int i = 0; i < numEntradas; i++) {
18       if (i > 0) s  += ", "; // separar entradas con comas
19       s += entradas[i];
20     }
21     return s + "]";
22   }
23
24   /** Intentar agregar un nuevo récord a la coleccion si es lo suficientemente
25      alto */
26   public void add(EntradaJuego e) {
27     int nuevaPuntuacion = e.getPuntuacion();
28     // ¿es la nueva entrada realmente un récord?
29     if (numEntradas == maxEntradas) { // el arreglo está lleno
30       if ( nuevaPuntuacion <= entradas[numEntradas-1].getPuntuacion() )
31         return;  // la nueva entrada, e, no es un record
32     }
33     else // el arreglo no está lleno
34       numEntradas++;
35     // Encontrar el lugar en donde la nueva entrada «e» estara
36     int i = numEntradas-1;
37     for ( ; (i >= 1) && (nuevaPuntuacion > entradas[i-1].getPuntuacion()); i--)
38       entradas[i] = entradas[i - 1]; // mover entrada i un lugar a la derecha
39     entradas[i] = e;                 // agregar el nuevo récord a entradas
40   }
41
42   /** Quitar el récord del índice i y devolverlo. */
43   public EntradaJuego remove(int i) throws IndexOutOfBoundsException {
44     if ((i < 0) || (i >= numEntradas))
45       throw new IndexOutOfBoundsException("Indice no valido: " + i);
46     // guardar temporlmente el objecto a ser quitado
47     EntradaJuego temp = entradas[i];
48
49     for (int j = i; j < numEntradas - 1; j++)   // contar hacia adelante desde i
50       entradas[j] = entradas[j+1];       // mover una celda a la izquierda
51
```

```
52      entradas[ numEntradas - 1 ] = null;  // anular el menor registro
53      numEntradas--;
54      return temp;                 // regresar objeto eliminado
55    }
56  }
```

Listado 2.2: Clase Puntuciones.java

Se quiere almacenar los mayores registros en un arreglo llamado `entradas`. La cantidad de registros puede variar, por lo que se usará el nombre simbólico, `maxEntradas`, para representar el número de registros que se quieren guardar. Se debe poner esta variable con un valor específico, y usando esta variable en el código, se puede hacer fácilmente un cambio más tarde si se requiere. Se define después el arreglo, `entradas`, para que sea un arreglo de tamaño `maxEntradas`. Inicialmente, el arreglo guarda solamente entradas nulas, pero conforme el usuario juega, se llenará en el arreglo con nuevos objetos `EntradaJuego`. Por lo que se necesitará definir métodos para actualizar las referencias `EntradaJuego` en las entradas del arreglo.

La forma como se mantiene organizado el arreglo `entradas` es simple—se guarda el conjunto de objetos `EntradaJuego` ordenado por sus valores enteros `puntuacion`, de mayor a menor. Si el número de objetos `EntradaJuego` es menor que `maxEntradas`, entonces se permite que al final del arreglo se guarden referencias `null`. Con este diseño se previene tener celdas vacías, u "hoyos", y los registros están desde el índice cero hasta la cantidad de juegos jugados. Se ilustra una instancia de la estructura de datos en la figura 2.1 y se da el código Java para tal estructura en el listado 2.2.

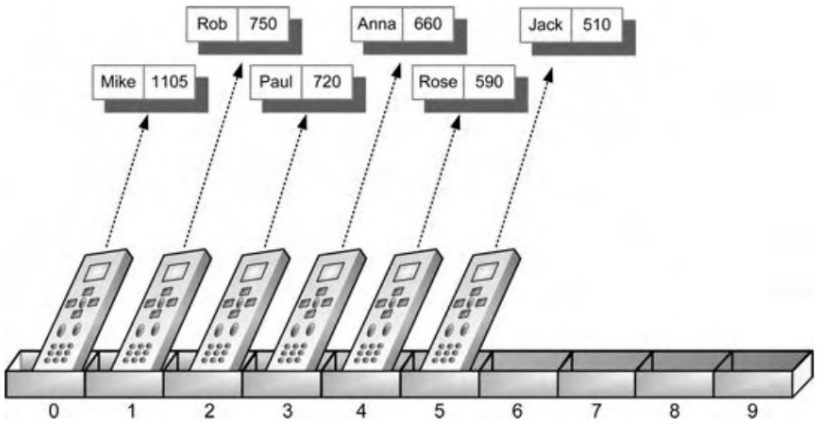

Figura 2.1: Una ilustración de un arreglo de tamaño 10 guardando referencias a 6 objetos `entradaJuego` desde la celda 0 hasta la 5, y el resto siendo referencias `null`.

El método `toString()` produce una cadena representando las mayores puntuaciones del arreglo `entradas`. Puede también ser empleado para propósitos de depuración. La cadena será una lista separada con comas de objetos

`EntradaJuego` del arreglo `entradas`. La lista se genera con un ciclo `for`, el cual agrega una coma justo antes de cada entrada que venga después de la primera.

Inserción

Una de las actualizaciones más comunes que se quiere hacer al arreglo `entradas` es agregar una nueva entrada. Suponiendo que se quiera insertar un nuevo objeto e del tipo `EntradaJuego` se considera como se haría la siguiente operación en una instancia de la clase `Puntuaciones`:

> `add`(e): inserta la entrada de juego e en la colección de las mayores puntuaciones. Si la colección está llena, entonces e es agregado solamente si su puntuación es mayor que la menor puntuación en el conjunto, y en este caso, e reemplaza la entrada con la menor puntuación.

El principal reto en la implementación de esta operación es saber donde e deberá ir en el arreglo `entradas` y hacerle espacio a e.

Se supondrá que las puntuaciones están en el arreglo de izquierda a derecha desde la mayor hasta la menor. Entonces se quiere que dada una nueva entrada, e, se requiere saber donde estará. Se inicia buscando al final del arreglo `entradas`. Si la última referencia en el arreglo no es `null` y su puntuación es mayor que la puntuación de e, entonces se detiene la búsqueda, ya que en este caso, e no es una puntuación alta. De otra forma, se sabe que e debe estar en el arreglo, y también se sabe que el último registro en el arreglo ya no debe estar. Enseguida, se va al penúltima referencia, si esta referencia es `null` o apunta a un objeto `EntradaJuego` con una puntuación que es menor que e, esta referencia deberá ser movida una celda a la derecha en el arreglo `entrada`. Se continua comparando y desplazando referencias de entradas del juego hasta que se alcanza el inicio del arreglo o se compara la puntuación de e con una entrada con una puntuación mayor. En cualquier caso, se habrá identificado el lugar al que e pertenece. Ver figura 2.2

Remoción de un objeto

Suponiendo que algún jugador hace trampa y obtiene una puntuación alta, se desearía tener un método que permita quitar la entrada del juego de la lista de las mayores puntuaciones. Por lo tanto se considera como debería implementarse la siguiente operación:

> `remove`(i): quita y regresa la entrada de juego e con el índice i en el arreglo `entradas`. Si el índice i está fuera de los límites del arreglo `entradas`, entonces el método deberá lanzar una excepción; de otra forma, el arreglo `entradas` será actualizado para quitar el objeto en la posición i y todos los objetos guardados con índices mayores que i serán movidos encima para llenar el espacio del dejado por el objeto quitado.

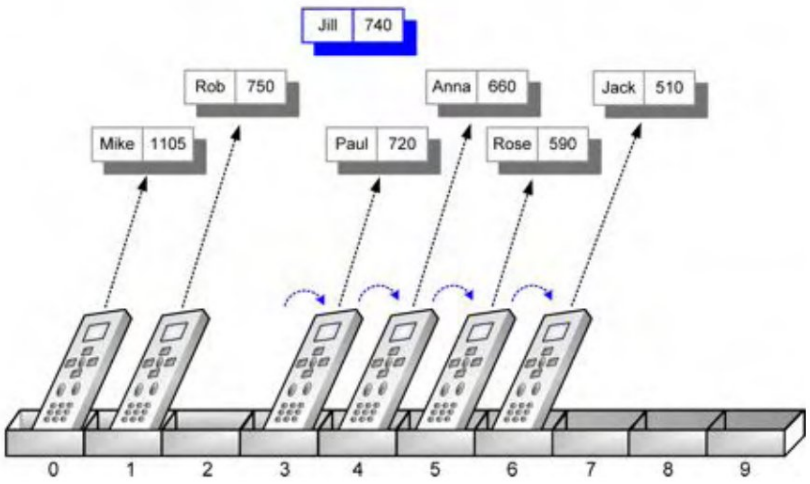

Figura 2.2: Preparación para agregar un nuevo objeto `entradaJuego` al arreglo `Entradas`. Para hacer espacio a una nueva referencia, se tienen que desplazar las referencias con los registros más pequeños que la nueva una celda a la derecha.

La implementación para `remove` será parecida al algoritmo para agregar un objeto, pero en forma inversa. Para remover la referencia al objeto con índice i, se inicia en el índice i y se mueven todas las referencias con índices mayores que i una celda a la izquierda. Ver figura 2.3.

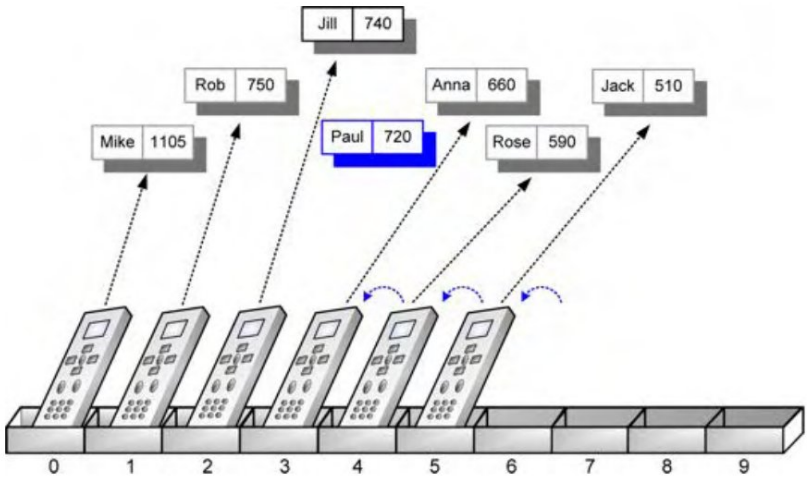

Figura 2.3: Remoción en el índice 3 en el arreglo que guarda las referencias a objetos `entradaJuego`.

Los detalles para hacer la operación de remoción consideran que primeramente se debe guardar la entrada del juego, llamada e, en el índice i del arreglo, en una variable temporal. Se usará esta variable para regresar e cuando se haya terminado de borrar. Para el movimiento de las referencias que son mayores que i una celda a la izquierda, no se hace hasta el final del arreglo—se para en la penúltima referencia, ya que la última referencia no tiene una referencia a su derecha. Para la última referencia en el arreglo `entradas`, es suficiente con anularla. Se termina regresando una referencia de la entrada removida, la cual no tiene una referencia que le apunte desde el arreglo `entradas`.

2.1.2. Ordenar un arreglo

En la sección previa, se mostró como se pueden agregar objetos o quitar estos en un cierto índice i en un arreglo mientras se mantiene el ordenamiento previo de los objetos intacto. En esta sección, se revisa una forma de iniciar con un arreglo con objetos que no están ordenadados y ponerlos en orden. Esto se conoce como el problema de *ordenamiento*.

Un algoritmo simple de inserción ordenada

Se revisarán varios algoritmos de ordenamiento en este material. Como introducción, se describe en esta sección, un algoritmo simple de ordenamiento, llamado *inserción ordenada*. En este caso, se describe una versión específica del algoritmo donde la entrada es un arreglo de elementos comparables. Se consideran tipos de algoritmos de ordenamiento más general mas adelante.

El algoritmo de inserción ordenada trabaja de la siguiente forma. Se inicia con el primer carácter en el arreglo y este carácter por si mismo ya está ordenado. Entonces se considera el siguiente carácter en el arreglo. Si este es menor que el primero, se intercambian. Enseguida se considera el tercer carácter en el arreglo. Se intercambia hacia la izquierda hasta que esté en su ordenamiento correcto con los primeros dos caracteres. De esta manera se considera el resto de los caracteres, hasta que el arreglo completo esté ordenado. Mezclando la descripción informal anterior con construcciones de programación, se puede expresar el algoritmo de inserción ordenada como se muestra a continuación.

Algoritmo InserciónOrdenada(A):
  Entrada: Un arreglo A de n elementos comparables
  Salida: El arreglo A con elementos reacomodados en orden creciente
  Para $i \leftarrow 1$ **Hasta** $n-1$ **Hacer**
    Insertar $A[i]$ en su sitio correcto en $A[0], A[1], \ldots, A[i-1]$.

Esta es una descripción de alto nivel del inserción ordenada y muestra porque este algoritmo recibe tal nombre—porque cada iteración inserta el siguiente elemento en la parte ordenada del arreglo que esta antes de este. Antes de codificar la descripción se requiere trabajar mas en los detalles de como hacer la tarea de inserción.

Se reescribirá la descripción de tal forma que se tenga un ciclo anidado dentro de otro. El ciclo exterior considerará cada elemento del arreglo en turno y el ciclo

interno moverá ese elemento a su posición correcta con el subarreglo ordenado de caracteres que están a su izquierda.

Refinar los detalles para inserción ordenada

Se describe a continuación una descripción en nivel intermedio del algoritmo de inserción ordenada. Esta descripción está más cercana al código actual, ya que hace una mejor descripción para insertar el elemento $A[i]$ en el subarreglo que está antes de este. Todavía emplea una descripción informal para el movimiento de los elementos si no están todavía ordenados.

Algoritmo InserciónOrdenada(A):
  Entrada: Un arreglo A de n elementos comparables
  Salida: El arreglo A con elementos recodados en orden creciente
  Para $i \leftarrow 1$ **Hasta** $n-1$ **Hacer**
    {Insertar $A[i]$ en su sitio correcto en $A[0], A[1], \ldots, A[i-1]$.}
    $actual \leftarrow A[i]$
    $j \leftarrow i-1$
    Mientras $j \geq 0$ Y $A[j] > actual$ **Hacer**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j-1$
    $A[j+1] \leftarrow actual$ {$actual$ está ahora en el lugar correcto}

Se ilustra un ejemplo de la ejecución del algoritmo de ordenamiento en la figura 2.4. La parte ordenada del arreglo está en blanco, y el siguiente elemento a ser insertado está en azul. Este se encuentra guardado en la variable *cur*. Cada renglón corresponde a una iteración del ciclo externo, y cada copia del arreglo en un renglón corresponde a una iteración del ciclo interno. Cada comparación se muestra con un arco. También se indica si la comparación dió un movimiento o no.

Descripción de inserción ordenada con Java

Se proporciona a continuación un código de Java para esta versión simple del algoritmo de inserción ordenada. El código proporcionado es para el caso especial cuando A es un arreglo de caracteres llamado referenciado por a.

```
1  /** Inserción ordenada de un arreglo de caracteres en orden creciente */
2  public static void insercionOrdenada( char[] a ) {
3    int n = a.length;
4    for (int i=1; i<n; i++) { // índice desde el segundo carácter en a
5
6      char actual = a[i];     // el carácter actual a ser insertado
7      int j = i-1;            // iniciar comparando con la celda izq. de i
8      while ((j>=0) && (a[j]>actual)) // mientras a[j] no está ordenado con actual
9        a[ j+1 ] = a[ j-- ]; // mover a[j] a la derecha y decrementar j
10     a[ j+1 ] = actual;      // este es el lugar correcto para actual
11   }
12 }
```

Listado 2.3: Código Java para realizar la inserción ordenada de un arreglo de caracteres

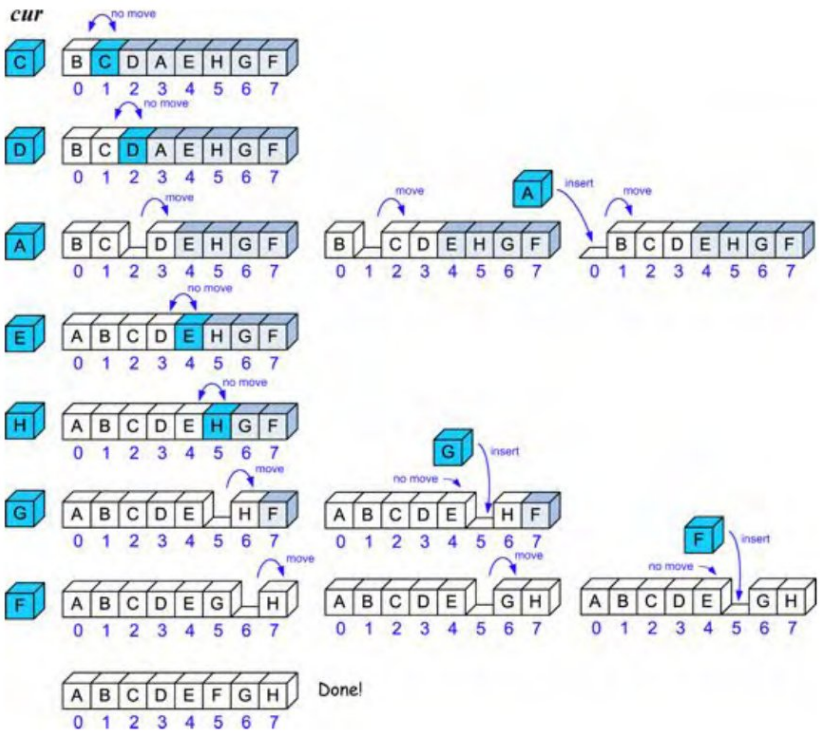

Figura 2.4: Ejecución del algoritmo de ordenamiento de un arreglo de 8 caracteres.

2.1.3. Métodos para arreglos y números aleatorios

Como los arreglos son muy importantes, Java proporciona un determinado número de métodos incorporados para realizar tareas comunes en arreglos. Estos métodos son estáticos en la clase `java.util.Arrays`. Esto es, ellos están asociados con la propia clase, y no con alguna instancia particular de la clase.

Algunos métodos simples

Se listan a continuación algunos métodos simples de la clase `java.util.Arrays` que no requieren explicación adicional:

`equals`(A, B): regresa verdadero si y solo si el arreglo A y el arreglo B son iguales. Se considera que dos arreglos son iguales si estos tienen el mismo número de elementos y cada par correspondiente de elementos en los dos arreglos son iguales. Esto es, A y B tienen los mismos elementos en el mismo orden.

`fill`(A, x): guarda el elemento x en cada celda del arreglo A.

`sort`(A): ordena el arreglo A usando el ordenamiento natural de sus elementos. Se emplea para ordenar el algoritmo ordenamiento rápido, el cual es mucho más rápido que la inserción ordenada.

`toString`(A): devuelve un `String` representante del arreglo A.

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  /** Programa que muestra algunos usos de los arreglos */
5  public class PruebaArreglo {
6    public static void main(String[] args) {
7      int num[] = new int[10];
8      Random rand = new Random(); // un generador de números pseudoaleatorios
9      rand.setSeed(System.currentTimeMillis()); // usar el tiempo act. como sem.
10     // llenar el arreglo num con números pseudoaleatorios entre 0 y 99, incl.
11     for (int i = 0; i < num.length; i++)
12       num[i] = rand.nextInt(100); // el siguiente numero pseudoaleatorio
13     int[] ant = (int[]) num.clone(); // clonar el arreglo num
14     System.out.println("arreglos iguales antes de ordenar: " +
15     Arrays.equals(ant,num));
16     Arrays.sort(num); // ordenar arreglo num (ant esta sin cambios)
17     System.out.println("arreglos iguales después de ordenar: " +
18     Arrays.equals(ant,num));
19     System.out.println("ant = " + Arrays.toString(ant));
20     System.out.println("num = " + Arrays.toString(num));
21   }
22 }
```

Listado 2.4: Programa PruebaArreglo.java que usa varios métodos incorporados en Java

Un ejemplo con números pseudoaleatorios

El programa `PruebaArreglo`, listado 2.4, emplea otra característica en Java—la habilidad para generar números pseudoaleatorios, esto es, números que son

estadísticamente aleatorios (pero no realmente aleatorios). En particular, este usa un objeto `java.util.Random`, el cual es un *generador de números pseudoaletorios*. Tal generador requiere un valor para iniciar, el cual es conocido como su *semilla*. La secuencia de números generados para una semilla dada será siempre la misma. En el programa, se pone el valor de la semilla al valor actual del tiempo en milisegundos a partir del 1° de enero de 1970, empleando el método `System.currentTimeMillis`, el cual será diferente cada vez que se ejecute el programa. Una vez que se ha puesto el valor de la semilla, se obtiene repetidamente un número aleatorio entre 0 y 99 llamando al método `nextInt` con el argumento 100. Un ejemplo de la salida del programa es:

```
arreglos iguales antes de ordenar: true
arreglos iguales después de ordenar: false
ant = [49, 41, 8, 10, 48, 87, 52, 2, 97, 81]
num = [2, 8, 10, 41, 48, 49, 52, 81, 87, 97]
```

Por cierto, hay una leve posibilidad de que los arreglos `ant` y `num` permanezcan igual, aún después de que `num` sea ordenado, es decir, si `num` ya está ordenado antes de que sea clonado. Pero la probabilidad de que esto ocurra es menor que uno en cuatro millones.

Criptografía simple con cadenas y arreglos de caracteres

Una de las aplicaciones primarias de los arreglos es la representación de `String`. Los objetos cadena son usualmente representados internamente como un arreglo de caracteres. Aún si las cadenas pudieran estar representadas de otra forma, hay una relación natural entre cadenas y arreglos de caracteres—ambos usan índices para referirse a sus caracteres. Gracias a esta relación, Java hace fácil la creación de objetos `String` desde arreglos de caracteres y viceversa. Para crear un objeto `String` de un arreglo de caracteres A, se usa la expresión,

`new String`(A)

Esto es, uno de los constructores de la clase `String` toma un arreglo de caracteres como su argumento y regresa una cadena teniendo los mismos caracteres en el mismo orden como en el arreglo. De igual forma, dado un `String` S, se puede crear un arreglo de caracteres de la representación de S usando la expresión,

`S.toCharArray()`

En la clase `String` hay un método, `toCharArray`, el cual regresa un arreglo, del tipo `char[]` con los mismos caracteres de S.

El cifrador César

Una área donde se requiere poder cambiar de una cadena a un arreglo de caracteres y de regreso es útil en *criptografía*, la ciencia de los mensajes secretos y sus aplicaciones. Esta área estudia varias formas de realizar el *encriptamiento*, el

cual toma un mensaje, denominado el *texto plano*, y lo convierte en un mensaje cifrado, llamado el *ciphertext*. Asimismo, la criptografía también estudia las formas correspondientes de realizar el *descifrado*, el cual toma un ciphertext y lo convierte de regreso en el texto plano original.

Es discutible que el primer esquema de encriptamiento es el *cifrado de César*, el cual es nombrado después de que Julio César, quién usaba este esquema para proteger mensajes militares importantes. El cifrador de César es una forma simple para oscurecer un mensaje escrito en un lenguaje que forma palabras con un alfabeto.

El cifrador involucra reemplazar cada letra en un mensaje con una letra que está tres letras después de esta en el alfabeto para ese lenguaje. Por lo tanto, en un mensaje del español, se podría reemplazar cada A con D, cada B con E, cada C con F, y así sucesivamente. Se continúa de esta forma hasta la letra W, la cual es reemplazada con Z. Entonces, se permite que la sustitución del patrón de la vuelta, por lo que se reemplaza la X con A, Y con B, y Z con C.

Usar caracteres como índices de arreglos

Si se fueran a numerar las letras como índices de arreglos, de tal forma que A sea 0, B sea 1, C sea 2, y así sucesivamente, entonces se puede escribir la fórmula del cifrador de César para el alfabeto inglés con la siguiente fórmula:

Reemplazar cada letra i con la letra $i + 3$ mód 26,

donde mód es el operador *módulo*, el cual regresa el residuo después de realizar una división entera. El operador se denota con % en Java, y es el operador que se requiere para dar la vuelta en el final del alfabeto. Para el 26 el módulo es 0, para el 27 es 1, y el 28 es 2. El algoritmo de desciframiento para el cifrador de César es exactamente lo opuesto—se reemplaza cada letra con una que está tres lugares antes que esta, con regreso para la A, B y C.

Se puede capturar esta regla de reemplazo usando arreglos para encriptar y descifrar. Porque cada carácter en Java está actualmente guardado como un número—su valor Unicode—se pueden usar letras como índices de arreglos. Para una letra mayúscula c, por ejemplo, se puede usar c como un índice de arreglo tomando el valor Unicode de c y restando A. Lo anterior sólo funciona para letras mayúsculas, por lo que se requerirá que los mensajes secretos estén en mayúsculas. Se puede entonces usar un arreglo, $encriptar$, que represente la regla de encriptamiento, por lo que $encriptar[i]$ es la letra que reemplaza la letra número i, la cual es $c - A$ para una letra mayúscula c en Unicode. De igual modo, un arreglo, $descifrar$, puede representar la regla de desciframiento, por lo que $descifrar[i]$ es la letra que reemplaza la letra número i.

En el listado 2.5, se da una clase de Java simple y completa para el cifrador de César, el cual usa la aproximación anterior y también realiza las conversiones entre cadenas y arreglos de caracteres.

```
1 /** Clase para hacer encriptamiento y desciframiento usando el cifrador de
2  * César. */
3 public class Cesar {
```

```
4
5     public static final char[] abc = {'A','B','C','D','E','F','G','H', 'I',
6       'J','K','L','M', 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
7     protected char[] encriptar = new char[abc.length];  // Arreglo encriptamiento
8     protected char[] descifrar = new char[abc.length];  // Arreglo desciframiento
9
10    /** Constructor que inicializa los arreglos de encriptamiento y
11     * desciframiento */
12    public Cesar() {
13      for (int i=0; i<abc.length; i++)
14        encriptar[i] = abc[(i + 3) % abc.length]; // rotar alfabeto 3 lugares
15      for (int i=0; i<abc.length; i++)
16        descifrar[encriptar[i]-'A'] = abc[i]; // descifrar inverso a encriptar
17    }
18
19    /** Método de encriptamiento */
20    public String encriptar(String secreto) {
21      char[] mensj = secreto.toCharArray(); // arreglo mensaje
22      for (int i=0; i<mensj.length; i++)    // ciclo de encriptamiento
23        if (Character.isUpperCase(mensj[i])) // se tiene que cambiar letra
24          mensj[i] = encriptar[mensj[i] - 'A']; // usar letra como índice
25      return new String(mensj);
26    }
27
28    /** Método de desciframiento */
29    public String descifrar(String secreto) {
30      char[] mensj = secreto.toCharArray();    // arreglo mensaje
31      for (int i=0; i<mensj.length; i++)       // ciclo desciframiento
32        if (Character.isUpperCase(mensj[i]))   // se tiene letra a cambiar
33          mensj[i] = descifrar[mensj[i] - 'A'];  // usar letra como índice
34      return new String(mensj);
35    }
36
37    /** Metodo main para probar el cifrador de César */
38    public static void main(String[] args) {
39      Cesar cifrador = new Cesar(); // Crear un objeto cifrado de César
40      System.out.println("Orden de Encriptamiento = " +
41        new String(cifrador.encriptar));
42      System.out.println("Orden de Descifrado = " +
43        new String(cifrador.descifrar));
44      String secreto = "ESTRUCTURAS DE DATOS; ARREGLOS Y LISTAS LIGADAS.";
45      secreto = cifrador.encriptar(secreto);
46      System.out.println(secreto);          // el texto cifrado
47      secreto = cifrador.descifrar(secreto);
48      System.out.println(secreto);          // debería ser texto plano
49    }
50  }
```

Listado 2.5: Una clase simple y completa de Java para el cifrador de César

Cuando se ejecuta este programa, se obtiene la siguiente salida:

```
Orden de Encriptamiento = DEFGHIJKLMNOPQRSTUVWXYZABC
Orden de Descifrado = XYZABCDEFGHIJKLMNOPQRSTUVW
HVWUXFWXUDV GH GDWRV; DUUHJORV B OLVWDV OLJDGDV.
ESTRUCTURAS DE DATOS; ARREGLOS Y LISTAS LIGADAS.
```

2.1.4. Arreglos bidimensionales y juegos de posición

Varios juegos de computadora, sean juegos de estrategia, de simulación, o de conflicto de primera persona, usan un "tablero" bidimensional. Los programas que manejan juegos de posición requieren una forma de representar objetos

en un espacio bidimensional. Una forma natural de hacerlo es con un *arreglo bidimensional*, donde se usan dos índices, por ejemplo i y j, para referirse a las celdas en el arreglo. El primer índice se refiere al número del renglón y el segundo al número de la columna. Dado tal arreglo se pueden mantener tableros de juego bidimensionales, así como realizar otros tipos de cálculos involucrando datos que están guardados en renglones y columnas.

Los arreglos en Java son unidimensionales; se puede usar un sólo índice para acceder cada celda de un arreglo. Sin embargo, hay una forma como se pueden definir arreglos bidimensionales en Java—se puede crear un arreglo bidimensional como un arreglo de arreglos. Esto es, se puede definir un arreglo de dos dimensiones para que sea un arreglo con cada una de sus celdas siendo otro arreglo. Tal arreglo de dos dimensiones es a veces llamado una *matriz*. En Java, se declara un arreglo de dos dimensiones como se muestra en el siguiente ejemplo:

```
int[][] Y = new int[8][10];
```

La sentencia crea una "arreglo de arreglos" bidimensional, `Y`, el cual es 8×10, teniendo ocho renglones y diez columnas. Es decir, `Y` es un arreglo de longitud ocho tal que cada elemento de `Y` es un arreglo de longitud de diez enteros. Lo siguiente podría entonces ser usos válidos del arreglo `Y` y siendo `i` y `j` variables tipo `int`:

```
Y[i][i+1] = Y[i][i] + 3;
i = a.length;
j = Y[4].length;
```

Se revisa una aplicación de un arreglo bidimensional implementando un juego simple posicional.

El gato

Es un juego que se practica en un tablero de tres por tres. Dos jugadores—X y O—alternan en colocar sus respectivas marcas en las celdas del tablero, iniciando con el jugador X. Si algún jugado logra obtener tres marcas suyas en un renglón, columna o diagonal, entonces ese jugador gana.

La idea básica es usar un arreglo bidimensional, `tablero`, para mantener el tablero del juego. Las celdas en este arreglo guardan valores para indicar si la celda está vacía o guarda una X o una O. Entonces, el tablero es una matriz de tres por tres, donde por ejemplo, el renglón central son las celdas `tablero[1][0]`, `tablero[1][1]`, `tablero[1][2]`. Para este caso, se decidió que las celdas en el arreglo tablero sean enteros, con un cero indicando una celda vacía, un uno indicando una X, y un -1 indicando O. Esta codificación permite tener una forma simple de probar si en una configuración del tablero es una victoria para X o para O, a saber, si los valores de un renglón, columna o diagonal suman -3 o 3. Se ilustra lo anterior en la figura 2.5.

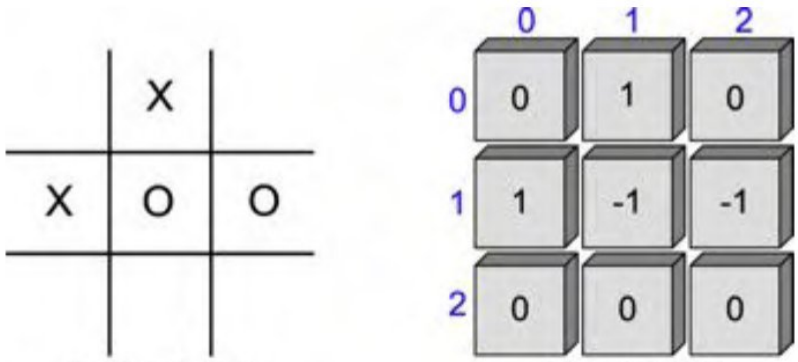

Figura 2.5: Una ilustración del juego del gato a la izquierda, y su representación empleando el arreglo `tablero`.

Se da una clase completa de Java para mantener un tablero del gato para dos jugadores en el listado 2.6. El código es sólo para mantener el tablero del gato y para registrar los movimientos; no realiza ninguna estrategia, ni permite jugar al gato contra la computadora.

```
1  /** Simulación del juego del gato (no tiene ninguna estrategia). */
2  public class Gato {
3
4    protected static final int X = 1, O = -1; // jugadores
5    protected static final int VACIA = 0;   // celda vacía
6    protected int tablero[][] = new int[3][3];  // tablero
7    protected int jugador;        // jugador actual
8
9    /** Constructor */
10   public Gato() { limpiarTablero(); }
11
12   /** Limpiar el tablero */
13   public void limpiarTablero() {
14     for (int i = 0; i < 3; i++)
15       for (int j = 0; j < 3; j++)
16         tablero[i][j] = VACIA;    // cada celda deberá estar vacía
17     jugador = X;        // el primer jugador es 'X'
18   }
19
20   /** Poner una marca X u O en la posición i,j */
21   public void ponerMarca(int i, int j) throws IllegalArgumentException {
22     if ( (i < 0) || (i > 2) || (j < 0) || (j > 2) )
23       throw new IllegalArgumentException("Posición invalida en el tablero");
24     if ( tablero[i][j] != VACIA )
25       throw new IllegalArgumentException("Posición del tablero ocupada");
26     tablero[i][j] = jugador; // colocar la marca para el jugador actual
27     jugador =- jugador; // cambiar jugadores (usar el hecho de que O=-X)
28   }
29
30   /** Revisar si la configuración del tablero es ganadora para un jugador dado*/
31   public boolean esGanador(int marca) {
32     return ((tablero[0][0] + tablero[0][1] + tablero[0][2] == marca*3) // ren. 0
33       || (tablero[1][0] + tablero[1][1] + tablero[1][2] == marca*3) // ren. 1
34       || (tablero[2][0] + tablero[2][1] + tablero[2][2] == marca*3) // ren. 2
35       || (tablero[0][0] + tablero[1][0] + tablero[2][0] == marca*3) // col. 0
36       || (tablero[0][1] + tablero[1][1] + tablero[2][1] == marca*3) // col. 1
37       || (tablero[0][2] + tablero[1][2] + tablero[2][2] == marca*3) // col. 2
38       || (tablero[0][0] + tablero[1][1] + tablero[2][2] == marca*3) // diag.
```

```
39        || (tablero[2][0] + tablero[1][1] + tablero[0][2] == marca*3)); // diag.
40    }
41
42    /** Regresar el jugador ganador o 0 para indicar empate */
43    public int ganador() {
44      if (esGanador(X))
45        return(X);
46      else if (esGanador(O))
47        return(O);
48      else
49        return(0);
50    }
51
52    /** Regresar una cadena simple mostrando el tablero actual */
53    public String toString() {
54      String s = "";
55      for (int i=0; i<3; i++) {
56        for (int j=0; j<3; j++) {
57          switch (tablero[i][j]) {
58            case X:      s += "X"; break;
59            case O:      s += "O"; break;
60            case  VACIA: s += " "; break;
61          }
62          if (j < 2) s += "|";       // límite de columna
63        }
64        if (i < 2) s += "\n-----\n";    // límite de renglon
65      }
66      return s;
67    }
68
69    /** Ejecución de prueba de un juego simple */
70    public static void main(String[] args) {
71      Gato juego = new Gato();
72         /* mueve X: */          /* mueve O: */
73      juego.ponerMarca(1,1);     juego.ponerMarca(0,2);
74      juego.ponerMarca(2,2);     juego.ponerMarca(0,0);
75      juego.ponerMarca(0,1);     juego.ponerMarca(2,1);
76      juego.ponerMarca(1,2);     juego.ponerMarca(1,0);
77      juego.ponerMarca(2,0);
78      System.out.println( juego.toString() );
79      int jugadorGanador = juego.ganador();
80      if (jugadorGanador != 0)
81        System.out.println(jugadorGanador + " gana");
82      else
83        System.out.println("Empate");
84    }
85  }
```

Listado 2.6: Una clase simple y completa para jugar gato entre dos jugadores.

Se muestra en la figura 2.6 el ejemplo de salida de la clase `Gato`.

2.2. Listas simples enlazadas

Los arreglos son elegantes y simples para guardar cosas en un cierto orden, pero tienen el inconveniente de no ser muy adaptables, ya que se tiene que fijar el tamaño del arreglo por adelantado.

Hay otras formas de guardar una secuencia de elementos, que no tiene el inconveniente de los arreglos. En esta sección se explora una implementación alterna, la cual es conocida como lista simple enlazada.

Una *lista enlazada*, en su forma más simple, es una colección de *nodos* que

```
O|X|O
-----
O|X|X
-----
X|O|X
Empate
```

Figura 2.6: Salida del ejemplo del juego del gato

juntos forman un orden lineal. El ordenamiento está determinado de tal forma que cada nodo es un objeto que guarda una referencia a un elemento y una referencia, llamado siguiente, a otro nodo. Ver la figura 2.7.

Figura 2.7: Ejemplo de una lista simple enlazada cuyos elementos son cadenas indicando códigos de aeropuertos. El apuntador `next` de cada nodo se muestra como una flecha. El objeto `null` es denotado por Ø.

Podría parecer extraño tener un nodo que referencia a otro nodo, pero tal esquema trabaja fácilmente. La referencia `sig` dentro de un nodo puede ser vista como un *enlace* o *apuntador* a otro nodo. De igual nodo, moverse de un nodo a otro siguiendo una referencia a `sig` es conocida como *salto de enlace* o *salto de apuntador*. El primer nodo y el último son usualmente llamados la cabeza y la cola de la lista, respectivamente. Así, se puede saltar a través de la lista iniciando en la cabeza y terminando en la cola. Se puede identificar la cola como el nodo que tenga la referencia `sig` como `null`, la cual indica el final de la lista. Una lista enlazada definida de esta forma es conocida como una *lista simple enlazada*.

Como en un arreglo, una lista simple enlazada guarda sus elementos en un cierto orden. Este orden está determinado por la cadenas de enlaces `sig` yendo desde cada nodo a su sucesor en la lista. A diferencia de un arreglo, una lista simple enlazada no tiene un tamaño fijo predeterminado, y usa espacio proporcional al número de sus elementos. Asimismo, no se emplean números índices para los nodos en una lista enlazada. Por lo tanto, no se puede decir sólo por examinar un nodo si este es el segundo, quinto u otro nodo en la lista.

Implementación de una lista simple enlazada

Para implementar una lista simple enlazada, se define una clase `Nodo`, como se muestra en el listado 2.7, la cual indica el tipo de los objetos guardados en los

nodos de la lista. Para este caso se asume que los elementos son `String`. También es posible definir nodos que puedan guardar tipos arbitrarios de elementos. Dada la clase `Nodo`, se puede definir una clase, `ListaSimple`, mostrada en el listado 2.8, definiendo la lista enlazada actual. Esta clase guarda una referencia al nodo cabeza y una variable va contando el número total de nodos.

```
1  /** Nodo de una lista simple enlazada de cadenas. */
2
3  public class Nodo {
4
5    private String elemento; // Los elementos son cadenas de caracteres
6    private Nodo sig;
7
8    /** Crea un nodo con el elemento dado y el nodo sig. */
9    public Nodo(String s, Nodo n) {
10     elemento = s;
11     sig = n;
12   }
13
14   /** Regresa el elemento de este nodo. */
15   public String getElemento() { return elemento; }
16
17   /** Regresa el nodo siguiente de este nodo. */
18   public Nodo getSig() { return sig; }
19
20   // métodos modificadores
21   /** Poner el elemento de este nodo. */
22   public void setElemento(String nvoElem) { elemento = nvoElem; }
23
24   /** Poner el nodo sig de este nodo. */
25   public void setSig(Nodo nvoSig) { sig = nvoSig; }
26 }
```

Listado 2.7: Implementación de un nodo de una lista simple enlazada

```
1  /** Lista simple enlazada. */
2  public class ListaSimple {
3
4    protected Nodo cabeza;  // nodo cabeza de la lista
5    protected long tam;     // cantidad de nodos en la lista
6
7    /** constructor por defecto que crea una lista vacía */
8    public ListaSimple() {
9      cabeza = null;
10     tam = 0;
11   }
12
13   // ... métodos de actualizacion y búsqueda deberían ir aquí
14 }
```

Listado 2.8: Implementación parcial de una lista simple enlazada

2.2.1. Inserción en una lista simple enlazada

Cuando se usa una lista simple enlazada, se puede fácilmente insertar un elemento en la cabeza de la lista, como se muestra en la figura 2.8.

La idea principal es que se cree un nuevo nodo, se pone su enlace *siguiente* para que se refiera al mismo objeto que la *cabeza*, y entonces se pone que la *cabeza* apunte al nuevo nodo. Se muestra enseguida como insertar un nuevo

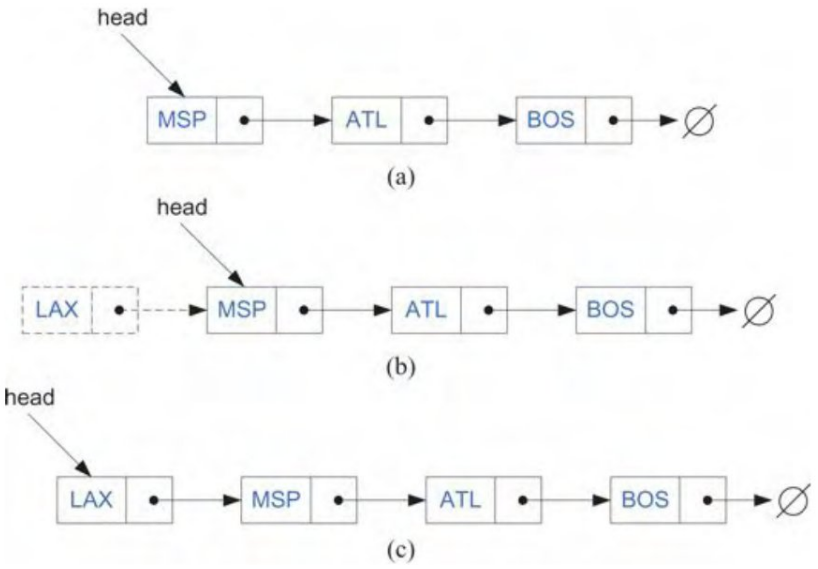

Figura 2.8: Inserción de un elemento en la cabeza de una lista simple enlazada: (a) antes de la inserción; (b) creación de un nuevo nodo; (c) después de la inserción.

nodo v al inicio de una lista simple enlazada. Este método trabaja aún si la lista está vacía. Observar que se ha puesto el apuntador *siguiente* para el nuevo nodo v antes de hacer que la variable *cabeza* apunte a v.

Algoritmo agregarInicio(v):
$v.setSiguiente(cabeza)$ { hacer que v apunte al viejo nodo cabeza }
$cabeza \leftarrow v$ { hacer que la variable *cabeza* apunte al nuevo nodo }
$tam \leftarrow tam + 1$ {incrementar la cuenta de nodos }

2.2.2. Inserción de un elemento en la cola

Se puede también fácilmente insertar un elemento en la cola de la lista, cuando se ha proporcionado una referencia al nodo cola, como se muestra en la figura 2.9.

En este caso, se crea un nuevo nodo, se asigna su referencia *siguiente* para que apunte al objeto `null`, se pone la referencia *siguiente* de la *cola* para que apunte a este nuevo objeto, y entonces se asigna a la referencia *cola* este nuevo nodo. El siguiente algoritmo inserta un nuevo nodo al final de la lista simple enlazada. El método también trabaja si la lista está vacía. Se pone primero el apuntador *siguiente* para el viejo nodo cola *antes* de hacer que la variable *cola* apunte al nuevo nodo.

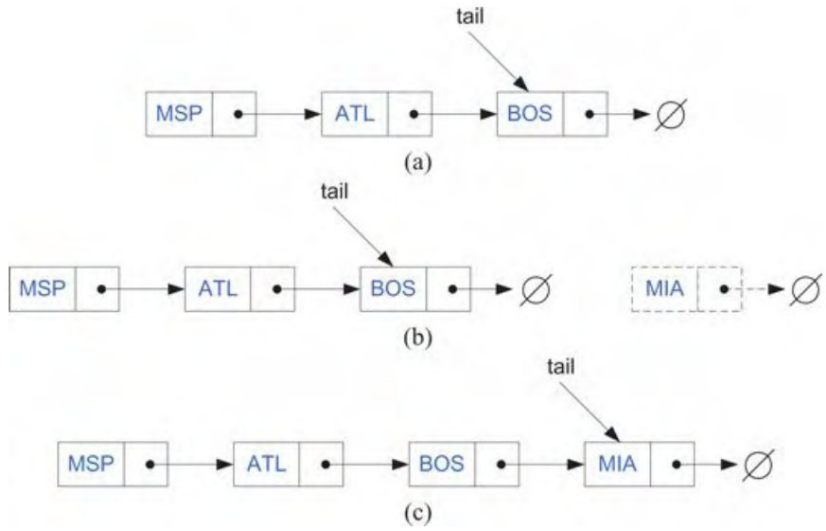

Figura 2.9: Inserción en la cola de una lista simple enlazada: (a) antes de la inserción; (b) creación de un nuevo nodo; (c) después de la inserción. Observar que se puso el enlace `next` para la `tail` en (b) antes de asignar a la variable `tail` para que apunte al nuevo nodo en (c).

Algoritmo agregarFinal(v):
$v.setSiguiente(null)$ { hacer que el nuevo nodo v apunte al objeto null }
$cola.setSiguiente(v)$ { el viejo nodo $cola$ apuntará al nuevo nodo }
$cola \leftarrow v$ { hacer que la variable $cola$ apunte al nuevo nodo }
$tam \leftarrow tam + 1$ {incrementar la cuenta de nodos }

2.2.3. Quitar un elemento de una lista simple enlazada

La operación contraria de insertar un nuevo elemento en la cabeza de una lista enlazada es quitar un elemento en la cabeza. Esta operación es ilustrada en la figura 2.10.

En el algoritmo siguiente se muestra la forma de realizar la remoción del elemento en la cabeza de la lista.

Algoritmo removerPrimero():
Si $cabeza =$ **null Entonces**
Indicar un error: la lista está vacía.
$t \leftarrow cabeza$
$cabeza \leftarrow cabeza.getSiguiente()$ { $cabeza$ apuntará al siguiente nodo }
$t.setSiguiente(null)$ { anular el apuntador siguiente del nodo borrado.}
$tam \leftarrow tam - 1$ { decrementar la cuenta de nodos }

Desafortunadamente, no se puede fácilmente borrar el nodo cola de una lista simple enlazada. Aún si se tiene una referencia `cola` directamente al último nodo de la lista, se necesita poder acceder el nodo *anterior* al último nodo en orden

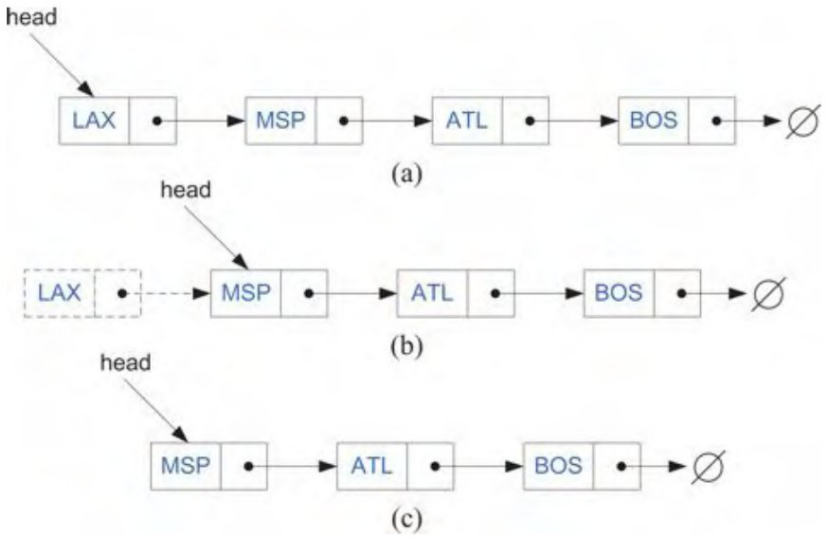

Figura 2.10: Remoción de un elemento en la cabeza de una lista simple enlazada: (a) antes de la remoción; (b) desligando el viejo nodo; (c) después de la remoción.

para quitar el último nodo. Pero no se puede alcanzar el nodo anterior a la cola usando los enlaces siguientes desde la cola. La única forma para acceder este nodo es empezar de la cabeza de la lista y llegar a través de la lista. Pero tal secuencia de operaciones de saltos de enlaces podría tomar un tiempo grande.

2.3. Listas doblemente enlazadas

Como se comentaba en la sección previa, quitar un elemento de la cola de una lista simple enlazada no es fácil. Además, el quitar cualquier otro nodo de la lista diferente de la cabeza de una lista simple enlazada es consumidor de tiempo, ya que no se tiene una forma rápida de acceder al nodo que se desea quitar. Por otra parte, hay varias aplicaciones donde no se tiene un acceso rápido tal como el acceso al nodo predecesor. Para tales aplicaciones, sería bueno que contarán con una forma de ir en ambas direcciones en una lista enlazada.

Hay un tipo de lista enlazada que permite ir en ambas direcciones—hacia adelante y hacia atrás—en una lista enlazada. Esta es la *lista doblemente enlazada*. Tal lista permite una gran variedad de operaciones rápidas de actualización, incluyendo la inserción y el borrado en ambos extremos, y en el centro. Un nodo en una lista doblemente enlazada guarda dos referencias—un enlace `sig`, el cual apunta al siguiente nodo en la lista, y un enlace `prev`, el cual apunta al nodo previo en la lista.

Una implementación de un nodo de una lista doblemente enlazada se muestra en el listado 2.9, donde nuevamente se asume que los elementos son cadenas de

caracteres.

```
1  /** Nodo de una lista doblemente enlazada de una lista de cadenas */
2  public class NodoD {
3
4    protected String elemento;  // Cadena elemento guardada por un nodo
5    protected NodoD sig, prev;  // Apuntadores a los nodos siguiente y previo
6
7    /** Constructor que crea un nodo con los campos dados */
8    public NodoD(String e, NodoD p, NodoD s) {
9      elemento = e;
10     prev = p;
11     sig = s;
12   }
13
14   /** Regresa el elemento de este nodo */
15   public String getElemento() { return elemento; }
16   /** Regresa el nodo previo de este nodo */
17   public NodoD getPrev() { return prev; }
18   /** Regresa el nodo siguiente de este nodo */
19   public NodoD getSig() { return sig; }
20   /** Pone el elemento de este nodo */
21   public void setElemento(String nvoElem) { elemento = nvoElem; }
22   /** Pone el nodo previo de este nodo */
23   public void setPrev(NodoD nvoPrev) { prev = nvoPrev; }
24   /** Pone el nodo siguiente de este nodo */
25   public void setSig(NodoD nvoSig) { sig = nvoSig; }
26 }
```

Listado 2.9: Clase `NodoD` representando un nodo de una lista doblemente enlazada que guarda una cadena de carácteres.

Centinelas cabeza y cola

Para simplificar la programación, es conveniente agregar nodos especiales en los extremos de lista doblemente enlazada: un nodo *cabeza* o *header* justo antes de la cabeza de la lista, y un nodo *cola* o *trailer* justo después de la cola de la lista. Estos nodos "falsos" o centinelas no guardan ningún elemento. La cabeza tiene una referencia `sig` válida pero una referencia `prev` nula, mientras *cola* tiene una referencia `prev` válida pero una referencia `sig` nula. Una lista doblemente enlazada con estos centinelas se muestra en la figura 2.11.

Figura 2.11: Una lista doblemente enlazada con centinelas, `header` y `trailer`, marcando el final de la lista.

Un objeto lista enlazada podría simplemente necesitar guardar referencias a estos dos centinelas y un contador `tam` que guarde el número de elementos, sin contar los centinelas, en la lista.

Insertar o remover elementos en cualquier extremo de la lista doblemente enlazada se hace directo. Además, el enlace `prev` elimina la necesidad de recorrer

la lista para obtener el nodo que está antes de la cola. Se muestra los detalles de la operación remover en la cola de la lista doblemente enlazada en la figura 2.12.

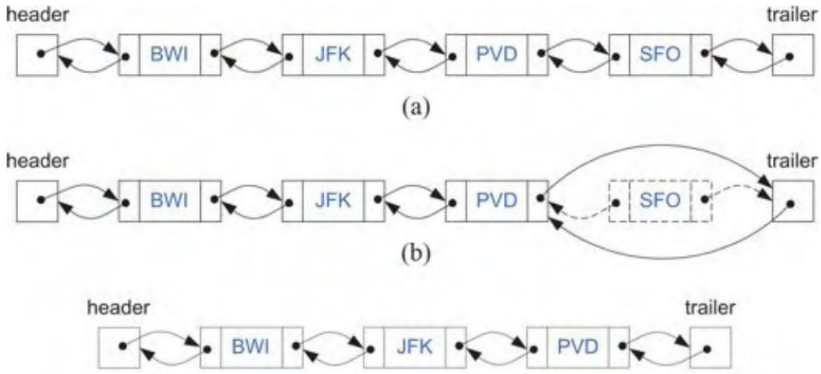

Figura 2.12: Remoción del nodo al final de una lista doblemente enlazada con centinelas: (a) antes de borrar en la cola; (b) borrando en la cola; (c) despueś del borrado.

Los detalles de la remoción en la cola de la lista doblemente enlazada se muestran en el siguiente algoritmo.

Algoritmo removerÚltimo():

Si $tam = 0$ **Entonces**
  Indicar un error: la lista está vacía.
$v \leftarrow terminación.getPrev()$ { último nodo }
$u \leftarrow v.getPrev()$ { nodo anterior al último nodo }
$terminación.setPrev(u)$
$u.setSig(terminación)$
$v.setPrev(null)$
$v.setSig(null)$
$tam \leftarrow tam - 1$ { decrementar la cuenta de nodos }

De igual modo, se puede fácilmente hacer una inserción de un nuevo elemento al inicio de una lista doblemente enlazada, como se muestra en el siguiente algoritmo, ver figura 2.13.

La implementación de la inserción de un nuevo nodo `v` al inicio, donde la variable `tam` guarda la cantidad de elementos en la lista. Este algoritmo también trabaja con una lista vacía.

Algoritmo agregarInicio(v):

$w \leftarrow cabeza.getSig()$ { primer nodo }
$v.setSig(w)$
$v.setPrev(cabeza)$
$w.setPrev(v)$
$cabeza.setSig(v)$
$tam \leftarrow tam + 1$

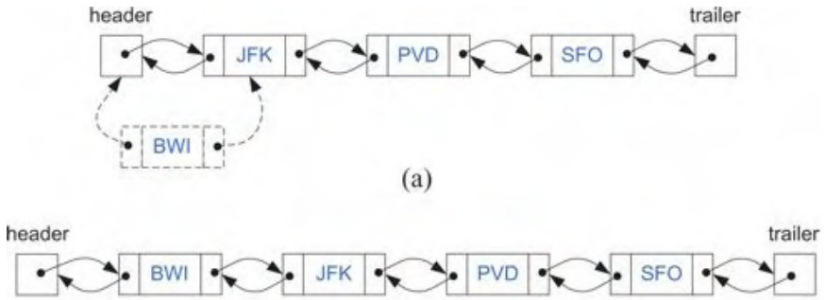

Figura 2.13: Inserción de un elemento en el frente: (a) previo; (b) final.

2.3.1. Inserción entre los extremos de una lista doble enlazada

Las listas dobles enlazadas no nada mas son útiles para insertar y remover elementos en la cabeza o en la cola de la lista. Estas también son convenientes para mantener una lista de elementos mientras se permite inserción y remoción en el centro de la lista. Dado un nodo `v` de una lista doble enlazada, el cual posiblemente podría ser la cabecera pero no la cola, se puede fácilmente insertar un nuevo nodo `z` inmediatamente después de `v`. Específicamente, sea `w` el siguiente nodo de `v`. Se ejecutan los siguientes pasos.

1. Hacer que el enlace `prev` de `z` se refiera a `v`.
2. Hacer que el enlace `sig` de `z` se refiera a `w`.
3. Hacer que el enlace `prev` de `w` se refiera a `z`.
4. Hacer que el enlace `sig` de `v` se refiera a `z`.

Este método está dado en detalle en el siguiente algoritmo y es ilustrado en la figura 2.14. Recordando el uso de los centinelas cabecera y cola, este algoritmo trabaja aún si `v` es el nodo cola, el nodo que está justo antes del centinela cola.

Algoritmo agregarDespues(v, z):
$w \leftarrow v.getSig()$ { nodo después de `v` }
$z.setPrev(v)$ { enlazar `z` a su predecesor, `v` }
$z.setSig(w)$ { enlazar `z` a su sucesor, `w` }
$w.setPrev(z)$ { enlazar `w` a su nuevo predecesor, `z` }
$w.setSig(z)$ { enlazar `v` a su nuevo sucesor, `z` }
$tam \leftarrow tam + 1$

2.3.2. Remover en el centro de una lista doblemente enlazada

De igual modo, es fácil remover un nodo `v` intermedio de una lista doblemente enlazada. Se acceden los nodos `u` y `w` a ambos lados de `v` usando sus métodos

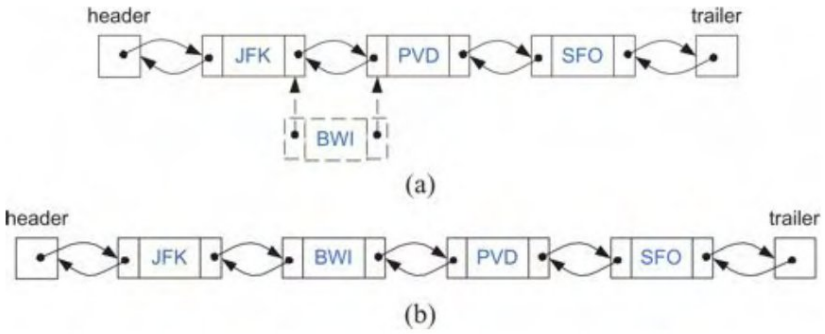

Figura 2.14: Inserción de un nuevo nodo después del nodo que guarda `JFK`: (a) creación de un nuevo nodo con el elemento `BWI` y enlace de este; (b) después de la inserción.

`getPrev` y `getSig`, estos nodos deberán existir, ya que se están usando centinelas. Para quitar el nodo `v`, se tiene que hacer que `u` y `w` se apunten entre ellos en vez de hacerlo hacia `v`. Esta operación es referida como desenlazar a `v`. También se anulan los apuntadores `prev` y `sig` de `v` para que ya no retengan referencias viejas en la lista. El siguiente algoritmo muestra como remover el nodo `v` aún si `v` es el primer nodo, el último o un nodo no centinela y se ilustra en la figura 2.15.

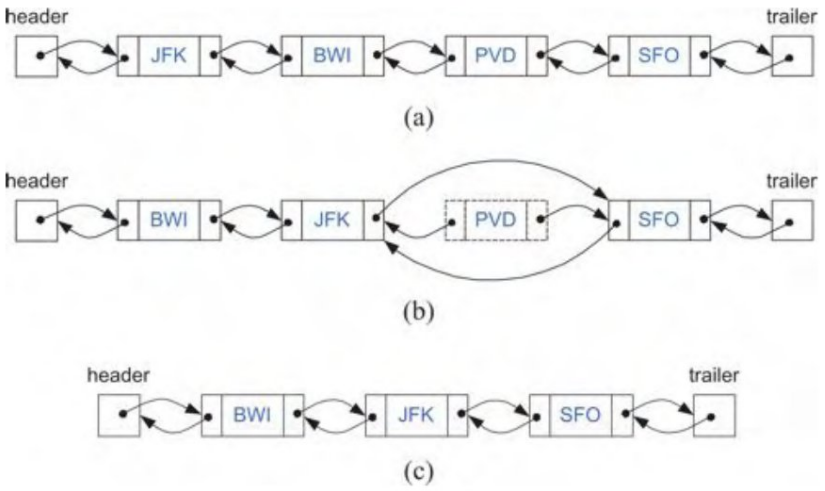

Figura 2.15: Borrado del nodo que guarda `PVD`: (a) antes del borrado; (b) desenlace del viejo nodo; (c) después de la remoción.

Algoritmo remover(v):
$u \leftarrow v.getPrev()$ { nodo predecesor a v }
$w \leftarrow v.getSig()$ { nodo sucesor a v }
$w.setPrev(u)$ { desenlazar v }
$u.setSig(w)$
$v.setPrev(null)$ { anular los campos de v }
$v.setNext(null)$
$tam \leftarrow tam - 1$ { decrementar el contador de nodos }

2.3.3. Una implementación de una lista doblemente enlazada

En el listado 2.10 se muestra una implementación de una lista doblemente enlazada con nodos que guardan `String` como elementos. Se hacen las siguientes observaciones acerca de la clase `ListaDoble`.

- Los objetos de la clase `NodoD`, los cuales guardan elementos `String`, son usados para todos los nodos de la lista, incluyendo los nodos centinela `cabecera` y `terminacion`.
- Se puede usar la clase `ListaDoble` para una lista doblemente enlazada de objetos `String` solamente. Para construir una lista de otros tipos, se puede usar una declaración genérica.
- Los métodos `getPrimero` y `getUltimo` dan acceso directo al primer nodo y el último en la lista.
- Los métodos `getPrev` y `getSig` permiten recorrer la lista.
- Los métodos `tienePrev` y `tieneSig` detectan los límites de la lista.
- Los métodos `agregarInicio` y `agregarFinal` agregan un nuevo nodo al inicio y al final de la lista, respectivamente.
- Los métodos `agregarAntes` y `agregarDespues` agregan un nodo antes y después de un nodo existente, respectivamente.
- Teniendo solamente un sólo método para borrar, `remover`, no es actualmente una restricción, ya que se puede remover al inicio o al final de una lista `L` doblemente enlazada ejecutando `L.remover(L.getPrimero())` o `L.remover(L.getUltimo())`, respectivamente.
- El método `toString` para convertir una lista entera en una cadena es útil para propósitos de prueba y depuración.

```
1 /** Lista doblemente enlazada con nodos de tipo NodoD guardando cadenas. */
2 public class ListaDoble {
3
4   protected int tam;      // número de elementos
5   protected NodoD cabeza, cola; // centinelas
```

```
6
7   /** Constructor que crea una lista vacia */
8   public ListaDoble() {
9     tam = 0;
10    cabeza = new NodoD(null, null, null); // crer cabeza
11    cola = new NodoD(null, cabeza, null); // crear terminación
12    cabeza.setSig(cola);  // cabeza apunta a terminación
13  }
14
15  /** Regresa la cantidad de elementos en la lista */
16  public int tam() { return tam; }
17
18  /** Indica si la lista esta vacía */
19  public boolean estaVacia() { return (tam == 0); }
20
21  /** Regresa el primer nodo de la lista */
22  public NodoD getPrimero() throws IllegalStateException {
23    if (estaVacia()) throw new IllegalStateException("La lista esta vacía");
24    return cabeza.getSig();
25  }
26
27  /** Regresa el último nodo de la lista */
28  public NodoD getUltimo() throws IllegalStateException {
29    if (estaVacia()) throw new IllegalStateException("La lista esta vacía");
30    return cola.getPrev();
31  }
32
33  /** Regresa el nodo anterior al nodo v dado. Un error ocurre si v
34    * es la cabeza */
35  public NodoD getPrev(NodoD v) throws IllegalArgumentException {
36    if (v == cabeza) throw new IllegalArgumentException
37      ("No se puede mover hacia atrás de la cabeza de la lista");
38    return v.getPrev();
39  }
40
41  /** Regresa el nodo siguiente al nodo v dado. Un error ocurre si v
42    * es la terminación */
43  public NodoD getSig(NodoD v) throws IllegalArgumentException {
44    if (v == cola) throw new IllegalArgumentException
45      ("No se puede mover hacia adelante de la terminación de la lista");
46   return v.getSig();
47  }
48
49  /** Inserta el nodo z dado antes del nodo v dado. Un error
50    * ocurre si v es la cabeza */
51  public void agregarAntes(NodoD v, NodoD z) throws IllegalArgumentException {
52    NodoD u = getPrev(v); // podría lanzar un IllegalArgumentException
53    z.setPrev(u);
54    z.setSig(v);
55    v.setPrev(z);
56    u.setSig(z);
57    tam++;
58  }
59
60  /** Inserta el nodo z dado despues del nodo v dado. Un error
61    * ocurre si v es la cabeza */
62  public void agregarDespues(NodoD v, NodoD z) {
63    NodoD w = getSig(v); // podría lanzar un IllegalArgumentException
64    z.setPrev(v);
65    z.setSig(w);
66    w.setPrev(z);
67    v.setSig(z);
68    tam++;
69  }
70
71  /** Inserta el nodo v dado en la cabeza de la lista */
72  public void agregarInicio(NodoD v) {
73    agregarDespues(cabeza, v);
```

```
74    }
75
76    /** Inserta el nodo v dado en la cola de la lista */
77    public void agregarFinal(NodoD v) {
78      agregarAntes(cola, v);
79    }
80
81    /** Quitar el nodo v dado de la lista. Un error ocurre si v es
82      * la cabeza o la cola */
83    public void remover(NodoD v) {
84      NodoD u = getPrev(v); // podría lanzar IllegalArgumentException
85      NodoD w = getSig(v);  // podría lanzar IllegalArgumentException
86      // desenlazar el nodo v de la lista
87      w.setPrev(u);
88      u.setSig(w);
89      v.setPrev(null);
90      v.setSig(null);
91      tam--;
92    }
93
94    /** Regresa si un nodo v dado tiene un nodo previo */
95    public boolean tienePrev(NodoD v) { return v != cabeza; }
96
97    /** Regresa si un nodo v dado tiene un nodo siguiente */
98    public boolean tieneSig(NodoD v) { return v != cola; }
99
100   /** Regresa una cadena representando la lista */
101   public String toString() {
102     String s = "[";
103     NodoD v = cabeza.getSig();
104     while (v != cola) {
105       s += v.getElemento();
106       v = v.getSig();
107       if (v != cola)
108         s += ",";
109     }
110     s += "]";
111     return s;
112   }
113 }
```

Listado 2.10: Una clase de lista doblemente enlazada.

2.4. Listas circulares y ordenamiento

En esta sección se revisan algunas aplicaciones y extensiones de listas enlazadas.

2.4.1. Listas circularmente enlazadas

El juego de niños, "*Pato, Pato, Ganso*", es jugado en varias culturas. Una variación de la lista simple enlazada, llamada la lista circularmente enlazada, es usada para una variedad de aplicaciones involucrando juegos circulares, como el juego que se comenta. Se muestra a continuación este tipo de lista y su aplicación en un juego circular.

Una *lista circularmente enlazada* tiene el mismo tipo de nodos que una lista simple enlazada. Esto es, cada nodo en una lista circularmente enlazada tiene un apuntador siguiente y una referencia a un elemento. Pero no hay una cabeza

o cola en la lista circularmente enlazada. En vez de tener que el apuntador del último nodo sea `null`, en una lista circularmente enlazada, este apunta de regreso al primer nodo. Por lo tanto, no hay primer nodo o último. Si se recorren los nodos de una lista circularmente enlazada desde cualquier nodo usando los apuntadores `sig`, se ciclará a través de los nodos.

Aún cuando una lista circularmente enlazada no tiene inicio o terminación, no obstante se necesita que algún nodo esté marcado como especial, el cual será llamado el *cursor*. El nodo cursor permite tener un lugar para iniciar si se requiere recorrer una lista circularmente inversa. Y si se recuerda esta posición inicial, entonces también se puede saber cuando se haya terminado con un recorrido en la lista circularmente enlazada, que es cuando se regresa al nodo que fue el nodo cursor cuando se inicio.

Se pueden entonces definir algunos métodos simples de actualización para una lista circularmente ligada:

`agregar`(v): inserta un nuevo nodo `v` inmediatamente después del cursor; si la lista está vacía, entonces `v` se convierte en el cursor y su apuntador `sig` apunta a el mismo.

`remover()`: borra y regresa el nodo `v` inmediatamente después del cursor (no el propio cursor, a menos que este sea el único nodo); si la lista queda vacía, el cursor es puesto a `null`.

`avanzar()`: avanza el cursor al siguiente nodo en la lista.

En el listado 2.11, se muestra una implementación Java de una lista circularmente enlazada, la cual usa la clase nodo del listado 2.7 e incluye también un método `toString` para generar una representación de cadena de la lista.

```
1  /** Lista circularmente enlazada con nodos de tipo Nodo guardando cadenas. */
2
3  public class ListaCircular {
4
5    protected Nodo cursor; // el cursor actual
6    protected int tam;     // la cantidad de nodos en la lista
7
8    /** Constructor que crea una lista vacia */
9    public ListaCircular() { cursor = null; tam = 0; }
10
11   /** Regresa la cantidad de nodos */
12   public int tam() { return tam; }
13
14   /** Regresa el cursor */
15   public Nodo getCursor() { return cursor; }
16
17   /** Mueve el cursor hacia adelante */
18   public void avanzar() { cursor = cursor.getSig(); }
19
20   /** Agregar un nodo después del cursor */
21   public void agregar(Nodo nodoNuevo) {
22     if (cursor == null) {    // ¿la lista está vacia?
23       nodoNuevo.setSig(nodoNuevo);
24       cursor = nodoNuevo;
25     }
26     else {
27       nodoNuevo.setSig(cursor.getSig());
28       cursor.setSig(nodoNuevo);
```

```
29      }
30      tam++;
31    }
32
33    /** Quitar el nodo despues del cursor */
34    public Nodo remover() {
35      Nodo nodoViejo = cursor.getSig(); // el nodo siendo removido
36      if ( nodoViejo == cursor )
37        cursor = null; // la lista se está vaciando
38      else {
39        cursor.setSig( nodoViejo.getSig() );  // desenlazando viejo nodo
40        nodoViejo.setSig( null );
41      }
42      tam--;
43      return nodoViejo;
44    }
45    /** Regresar una representación String de la lista,
46      * iniciando desde el cursor */
47    public String toString() {
48      if (cursor == null) return "[]";
49      String s = "[..." + cursor.getElemento();
50      Nodo cursorOrig = cursor;
51      for ( avanzar(); cursorOrig != cursor; avanzar() )
52        s += ", " + cursor.getElemento();
53      return s + "...]";
54    }
55  }
```

Listado 2.11: Una clase de lista circularmente ligada con nodos simples.

Algunas observaciones acerca de la clase `ListaCircular`

Es un programa simple que puede dar suficiente funcionalidad para simular juegos circulares, como Pato, Pato, Ganso, como se ve enseguida. No es un programa robusto, sin embargo. En particular, si una lista circular está vacía, entonces llamar `avanzar` o `remover` en esa lista causará una excepción.

Pato, Pato, Ganso

En el juego de niños Pato, Pato, Ganso, un grupo de niños se sienta en círculo. Uno de ellos es ellos es elegido para quedarse parado y este camina alrededor del círculo por fuera. El niño elegido toca a cada niño en la cabeza, diciendo “Pato” o “Ganso”. Cuando el elegido decide llamar a alguien el “Ganso”, el “Ganso” se levanta y persigue al otro niño alrededor del círculo. Si el “Ganso” no logra tocar al elegido, entonces a él le toca quedarse para la próxima ronda, sino el elegido lo seguirá siendo en la siguiente ronda. El juego continua hasta que los niños se aburran o un adulto les llame para comer un refrigerio.

Simular este juego es una aplicación ideal de una lista circularmente enlazada. Los niños pueden representar nodos en la lista. El niño “elegido” puede ser identificado como el niño que está sentado después del cursor, y puede ser removido del círculo para simular la marcha alrededor. Se puede avanzar el cursor con cada “Pato” que el elegido identifique, lo cual se puede simular con una decisión aleatoria. Una vez que un “Ganso” es identificado, se puede remover este nodo de la lista, hacer una selección aleatoria para simular si el “Ganso” alcanzo al elegido, e insertar el ganador en la lista. Se puede entonces avanzar el

cursor e insertar al ganador y repetir el proceso o termina si este es la última partida del juego.

Usar una lista circularmente enlazada para simular Pato, Pato, Ganso

Se da un código de Java para simular el juego Pato, Pato, Ganso en el listado 2.12.

```
1  import java.util.Random;
2
3  public class PatoPatoGanso {
4
5    /** Simulación de Pato, Pato, Ganso con una lista circularmente enlazada. */
6    public static void main(String[] args) {
7
8      ListaCircular C = new ListaCircular();
9      int N = 3;  // número de veces que se jugará el juego
10     Nodo eleg;  // el jugador que es el elegido
11     Nodo ganso; // el ganso
12     Random rand = new Random();
13
14     // usar el tiempo actual como semilla
15     rand.setSeed( System.currentTimeMillis() );
16
17     // Los jugadores ...
18     String[] noms = {"Alex","Eli","Paco","Gabi","Pepe","Luis","Toño","Pedro"};
19     for ( String nombre: noms ) {
20       C.agregar( new Nodo( nombre, null) );
21       C.avanzar();
22     }
23
24     System.out.println("Los jugadores son: "+C.toString());
25     for (int i = 0; i < N; i++) {   // jugar Pato, Pato, Ganso N veces
26       eleg = C.remover();
27       System.out.println(eleg.getElemento() + " es elegido.");
28       System.out.println("Jugando Pato, Pato, Ganso con: "+C.toString());
29       // marchar alrededor del circulo
30       while (rand.nextBoolean() || rand.nextBoolean()) {
31          C.avanzar(); // avanzar con probabilidad 3/4
32          System.out.println( C.getCursor().getElemento() + " es un Pato.");
33       }
34       ganso = C.remover();
35       System.out.println( "¡"+ganso.getElemento() + " es el ganso!" );
36       if ( rand.nextBoolean() ) {
37         System.out.println("¡El ganso ganó!");
38         C.agregar(ganso); // poner al ganso de regreso en su lugar anterior
39         C.avanzar();  // ahora el cursor está sobre el ganso
40         C.agregar(eleg);  // El "elegido" seguirá siéndolo en la próxima ronda
41       }
42       else {
43         System.out.println("¡El ganso perdió!");
44         C.agregar(eleg);  // poner al elegido en el lugar del ganso
45         C.avanzar();  // ahora el cursor está sobre la persona elegida
46         C.agregar(ganso); // El ganso ahora es el elegido en la siguiente ronda
47       }
48     }
49     System.out.println("El circulo final es " + C.toString());
50   }
51 } // fin de la clase PatoPatoGanso
```

Listado 2.12: El método main de un programa que usa una lista circularmente enlazada para simular el juego de niños Pato–Pato–Ganso.

Se muestra un ejemplo de la salida de la ejecución del programa Pato, Pato, Ganso a continuación:

```
Los jugadores son: [...Pedro, Alex, Eli, Paco, Gabi, Pepe, Luis, Toño...]
Alex es elegido.
Jugando Pato, Pato, Ganso con: [...Pedro, Eli, Paco, Gabi, Pepe, Luis, Toño...]
Eli es un Pato.
Paco es el ganso!
El ganso perdio!
Paco es elegido.
Jugando Pato, Pato, Ganso con: [...Alex, Gabi, Pepe, Luis, Toño, Pedro, Eli...]
Gabi es un Pato.
Pepe es un Pato.
Luis es un Pato.
Toño es un Pato.
Pedro es un Pato.
Eli es un Pato.
Alex es un Pato.
Gabi es un Pato.
Pepe es un Pato.
Luis es un Pato.
Toño es un Pato.
Pedro es un Pato.
Eli es el ganso!
El ganso gano!
Paco es elegido.
Jugando Pato, Pato, Ganso con: [...Eli, Alex, Gabi, Pepe, Luis, Toño, Pedro...]
Alex es un Pato.
Gabi es el ganso!
El ganso gano!
El circulo final es [...Gabi, Paco, Pepe, Luis, Toño, Pedro, Eli, Alex...]
```

Observar que cada iteración en esta ejecución del programa genera una salida diferente, debido a las configuraciones iniciales diferentes y el uso de opciones aleatorias para identificar patos y ganso. Asimismo, si el ganso alcanza al ganso depende de opciones aleatorias.

2.4.2. Ordenando una lista enlazada

Se muestra a continuación el algoritmo de inserción ordenada para una lista doblemente enlazada.

Algoritmo InsercionOrdenada(L):

Entrada: Una lista L doblemente enlazada de elementos comparables
Salida: La lista L con elementos rearreglados en orden creciente
Si $L.tam() \leq 1$ **Entonces**
  regresar
$fin \leftarrow L.getPrimero()$
Mientras fin no sea el último nodo en L **Hacer**
  $pivote \leftarrow fin.getSig()$
  Quitar $pivote$ de L
  $ins \leftarrow fin$
  Mientras ins no sea la cabecera y el elemento de ins sea mayor que el del $pivote$ **Hacer**
    $ins \leftarrow ins.getPrev()$
  Agregar $pivote$ justo después de ins en L
  Si $ins = fin$ **Entonces** { Se agregó $pivote$ después de fin en este caso}
    $fin \leftarrow fin.getSig()$

Una implementación en Java del algoritmo de inserción ordenada para una lista doblemente enlazada representada por la clase `ListaDoble` se muestra en el listado 2.13.

```
1  /** Inserción ordenada para una lista doblemente enlazada
2      de la clase ListaDoble. */
3  public static void ordenar(ListaDoble L) {
4    if (L.tam() <= 1) return; // L ya está ordenada en este caso
5
6    NodoD pivote; // nodo pivote
7    NodoD ins;    // punto de inserción
8    NodoD fin = L.getPrimero();  // fin de la corrida
9
10   while (fin != L.getUltimo()) {
11     pivote = fin.getSig(); // obtener el siguiente nodo de pivote
12     L.remover(pivote);     // quitar pivote de L
13     ins = fin;    // iniciar busqueda desde fin de la corrida ordenada
14
15     while (L.tienePrev(ins) &&
16            ins.getElemento().compareTo(pivote.getElemento()) > 0)
17       ins = ins.getPrev(); // mover a la izquierda
18
19   // agregar el pivote de regreso, despues del punto de inserción
20     L.agregarDespues(ins,pivote);
21
22     if (ins == fin)       // se ha agregado pivote después de fin (se quedo
23       fin = fin.getSig(); // en el mismo lugar) -> incrementar marcador fin
24   }
25 }
```

Listado 2.13: Ordenamiento por inserción para una lista doblemente enlazada de la clase `ListaDoble`.

2.5. Recurrencia

La repetición se puede lograr escribiendo ciclos, como por ejemplo con `for` o con `while`. Otra forma de lograr la repetición es usando *recurrencia*, la cual ocurre cuando una función se llama a sí misma. Se han visto ejemplos de métodos

que llaman a otros métodos, por lo que no debería ser extraño que muchos de los lenguajes de programación moderna, incluyendo Java, permitan que un método se llame a sí mismo. En esta sección, se verá porque esta capacidad da una alternativa y poderosa para realizar tareas repetitivas.

La función factorial

Para ilustrar la recurrencia, se inicia con ejemplo sencillo para calcular el valor de la *función factorial*. El factorial de un entero positivo n, denotada por $n!$, está definida como el producto de los enteros desde 1 hasta n. Si $n = 0$, entonces $n!$ está definida como uno por convención. Mas formalmente, para cualquier entero $n \geq 0$,

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ 1 \cdot 2 \cdots (n-2) \cdot (n-1) \cdot n & \text{si } n \geq 1 \end{cases}$$

Para hacer la conexión con los métodos más clara, se usa la notación `factorial(n)` para denotar $n!$.

La función factorial puede ser definida de una forma que sugiera una formulación recursiva. Para ver esto, observar que

$$\text{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \text{factorial}(4)$$

Por lo tanto, se puede definir factorial(5) en términos del factorial(4). En general, para un entero positivo n, se puede definir factorial(n) como $n \cdot$factorial($n-1$). Esto lleva a la siguiente *definición recursiva*.

$$\text{factorial}(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{si } n \geq 1 \end{cases} \tag{2.1}$$

Esta definición es típica de varias definiciones recursivas. Primero, esta contiene uno o más *casos base*, los cuales están definidos no recursivamente en términos de cantidades fijas. En este caso, $n = 0$ es el caso base. Esta definición también contiene uno o mas *casos recursivos*, los cuales están definidos apelando a la definición de la función que está siendo definida. Observar que no hay circularidad en esta definición, porque cada vez que la función es invocada, su argumento es más pequeño en uno.

Una implementación recursiva de la función factorial

Se considera una implementación en Java, listado 2.14, de la función factorial que está dada por la fórmula 2.1 con el nombre `factorialRecursivo()`. Observar que no se requiere ciclo. Las invocaciones repetidas recursivas de la función reemplazan el ciclo.

```
1 public class Recurrencia {
2
3   public static int factorialRecursivo(int n) { // función recursiva factorial
4     if (n == 0) return 1; // caso base
```

Listado 2.14: Una implementación recursiva de la función factorial.

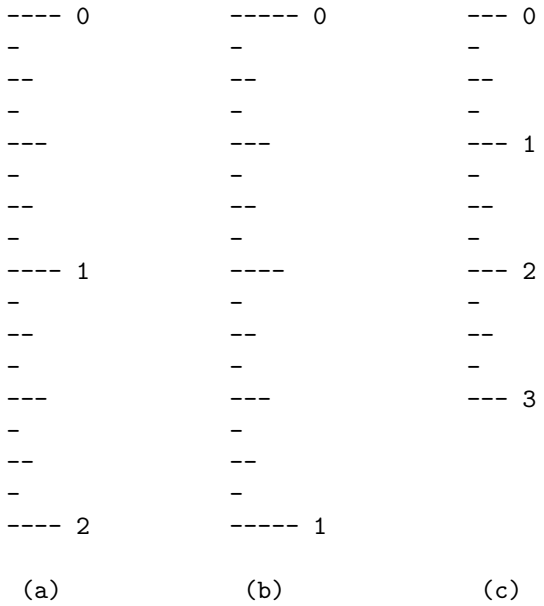

Figura 2.16: Ejemplos de la función dibuja regla: (a) regla de 2" con longitud 4 de la marca principal; (b) regla de 1" con longitud; (c) regla de 3" con longitud 3.

¿Cuál es la ventaja de usar recurrencia? La implementación recursiva de la función factorial es un poco más simple que la versión iterativa, en este caso no hay razón urgente para preferir la recurrencia sobre la iteración. Para algunos problemas, sin embargo, una implementación recursiva puede ser significativamente más simple y fácil de entender que una implementación iterativa como los ejemplos siguientes.

Dibujar una regla inglesa

Como un ejemplo más complejo del uso de recurrencia, considerar como dibujar las marcas de una regla típica Inglesa. Una regla está dividida en intervalos de una pulgada, y cada intervalo consiste de un conjunto de marcas colocadas en intervalos de media pulgada, un cuarto de pulgada, y así sucesivamente. Como el tamaño del intervalo decrece por la mitad, la longitud de la marca decrece en uno, ver figura 2.16.

Cada múltiplo de una pulgada también tiene una etiqueta numérica. La longitud de la marca más grande es llamada la *longitud de la marca principal*. No se consideran las distancias actuales entre marcas, sin embargo, se imprime una marca por línea.

Una aproximación recursiva para dibujar la regla

La aproximación para dibujar tal regla consiste de tres funciones. La función principal `dibujaRegla()` dibuja la regla entera. Sus argumentos son el número total de pulgadas en la regla, `nPulgadas`, y la longitud de la marca principal, `longPrincipal`. La función utilidad `dibujaUnaMarca()` dibuja una sola marca de la longitud dada. También se le puede dar una etiqueta entera opcional, la cual es impresa si esta no es negativa.

El trabajo interesante es hecho por la función recursiva `dibujaMarcas()`, la cual dibuja la secuencia de marcas dentro de algún intervalo. Su único argumento es la longitud de la marca asociada con la marca del intervalo central. Considerar la regla de 1" con longitud 5 de la marca principal mostrada en la figura 2.16(b). Ignorando las líneas conteniendo 0 y 1, considerar como dibujar la secuencia de marcas que están entre estas líneas. La marca central (en 1/2") tiene longitud 4. Los dos patrones de marcas encima y abajo de esta marca central son idénticas, y cada una tiene una marca central de longitud 3. En general, un intervalo con una marca central de longitud $L \geq 1$ está compuesta de lo siguiente:

- Un intervalo con una longitud de marca central $L - 1$
- Un sólo interalo de longitud L
- Un intervalo con una longitud de marca central $L - 1$

Con cada llamada recursiva, la longitud decrece por uno. Cuando la longitud llega a cero, simplemente se debe regresar. Como resultado, este proceso recursivo siempre terminará. Esto sugiere un proceso recursivo, en el cual el primer paso y último son hechos llamando a `dibujaMarcas(L-1)` recursivamente. El paso central es realizado llamando a la función `dibujaUnaMarca(L)`. Esta formulación recursiva es mostrada en el listado 2.15. Como en el ejemplo del factorial, el código tiene un caso base, cuando $L = 0$. En este ejemplo se hacen dos llamadas recursivas a la función.

```
1   }
2
3   // dibuja una marca sin etiqueta
4   public static void dibujaUnaMarca(int longMarca) {
5     dibujaUnaMarca(longMarca, -1);
6   }
7
8   // dibuja una marca
9   public static void dibujaUnaMarca(int longMarca, int etiqMarca) {
10    for (int i = 0; i < longMarca; i++)
11      System.out.print("-");
12    if (etiqMarca >= 0) System.out.print(" " + etiqMarca);
13    System.out.print("\n");
14  }
15
16  // dibuja marcas de longitud dada
17  public static void dibujaMarcas(int longMarca) {
18    if (longMarca > 0) { // parar cuando la longitud llegue a 0
19      dibujaMarcas(longMarca-1); // recursivamente dibujar marca superior
20      dibujaUnaMarca(longMarca); // dibujar marca principal
21      dibujaMarcas(longMarca-1); // recursivamente dibujar marca inferior
22    }
```

```
23    }
24
25    // dibuja regla
26    public static void dibujaRegla(int nPulgadas, int longPrincipal) {
27      dibujaUnaMarca(longPrincipal, 0); // dibujar marca 0 y su etiqueta
28      for (int i = 1; i <= nPulgadas; i++) {
29        dibujaMarcas(longPrincipal-1);    // dibujar marcas para esta pulgada
30        dibujaUnaMarca(longPrincipal, i); // dibujar marca i y su etiqueta
```

Listado 2.15: Una implementación recursiva de una función que dibuja una regla.

Sumar los elementos de un arreglo recursivamente

Suponer que se da un arreglo, A, de n enteros que se desean sumar juntos. Se puede resolver el problema de la suma usando recurrencia lineal observando que la suma de todos los n enteros de A es igual a $A[0]$, si $n = 1$, o la suma de los primeros $n - 1$ enteros más la suma del último elemento en A. En particular, se puede resolver este problema de suma usando el algoritmo recursivo siguiente.

Algoritmo SumaLineal(A, n):
Entrada: Un arreglo A de enteros y un entero $n \geq 1$ tal que A tiene al menos n elementos.
Salida: La suma de los primeros n enteros en A.
Si $n = 1$ **Entonces**
  regresar $A[0]$
Si no
  regresar SumaLineal$(A, n-1) + A[n]$

Este ejemplo, también ilustra una propiedad importante que un método recursivo debería siempre tener—que el método termina. Se asegura esto escribiendo una sentencia no recursiva para el caso $n = 1$. Además, siempre se hace una llamada recursiva sobre un valor más pequeño del parámetro $(n - 1)$ que el que fue dado (n), por lo tanto, en algún punto (en el punto mas bajo de la recurrencia), se hará la parte no recursiva del computo (regresando $A[0]$). En general, un algoritmo que usa recurrencia lineal típicamente tiene la siguiente forma:

- **Prueba para los casos base**. Se inicia probando un conjunto de casos base, debería haber al menos uno. Estos casos basos deberían estar definidos, por lo que cada posible encadenamiento de llamadas recursivas eventualmente alcanzará un caso base, y el manejo de cada caso base no debería usar recurrencia.

- **Recurrencia**. Después de probar los casos base, entonces se realizan llamadas recursivas simples. Este paso recursivo podría involucrar una prueba que decida cual de varias posibles llamadas recursivas hacer, pero finalmente debería escoger hacer sólo una de estas posibles llamadas cada vez que se haga este paso. Mas aún, se debería definir cada posible llamada recursiva para que esta progrese hacia el caso base.

Invertir un arreglo por recurrencia

Se considera ahora el problema de invertir los n elementos de un arreglo, A, para que el primer elemento sea el último, el segundo elemento sea el penúltimo, y así sucesivamente. Se puede resolver este problema usando recurrencia lineal, observando que la inversión de un arreglo se puede lograr intercambiando el primer elemento y el último, y entonces invertir recursivamente el resto de los elementos en el arreglo. Se describen a continuación los detalles de este algoritmo, usando la convención de que la primera vez que se llama el algoritmo se hace como `invertirArreglo`$(A, 0, n-1)$.

Algoritmo InvertirArreglo(A, i, j):
  Entrada: Un arreglo A e índices enteros no negativos i y j.
  Salida: La inversión de los elementos en A iniciando en el índice i
    y terminando en j.
  Si $i < j$ **Entonces**
    Intercambiar $A[i]$ y $A[j]$
    InvertirArreglo$(A, i+1, j-1)$
  Regresar

En este algoritmo se tienen dos casos base implícitos, a saber, cuando $i = j$ y cuando $i > j$. Sin embargo, en cualquier caso, simplemente se termina el algoritmo, ya que una secuencia con cero elementos o un elemento es trivialmente igual a su inversión. Además, en el paso recursivo se está garantizando ir progresando hacia uno de los dos casos base. Si n es par, eventualmente se alcanzará el caso $i = j$, y si n es impar, eventualmente se alcanzará el caso $i > j$. El argumento anterior implica que el algoritmo recursivo está garantizado para terminar.

Definiendo problemas de manera que facilite la recurrencia

Para diseñar un algoritmo recursivo para un problema dado, es útil pensar en las diferentes formas en que se puede subdividir el problema para definir problemas que tengan la misma estructura general que el problema original. Este proceso en ocasiones significa que se necesita redefinir el problema original para facilitar la búsqueda de problemas similares. Por ejemplo, con el algoritmo `InvertirArreglo`, se agregaron los parámetros i y j por lo que una llamada recursiva para invertir la parte interna del arreglo A podrían tener la misma estructura y misma sintaxis, como la llamada para invertir cualquier otro par de valores del arreglo A. Entonces, en vez de que inicialmente se llame al algoritmo como `InvertirArreglo`(A), se llama este inicialmente como `InvertirArreglo`$(A, 0, n-1)$. En general, si se tiene dificultad para encontrar la estructura repetitiva requerida para diseñar un algoritmo recursivo, en algunos casos es útil trabajar el problema con algunos ejemplos concretos pequeños para ver como los subproblemas deberían estar definidos.

Recurrencia de cola

Usar la recurrencia puede ser con frecuencia una herramienta útil para diseñar algoritmos que tienen definiciones cortas y elegantes. Pero esta utilidad viene

acompañado de un costo modesto. Cuando se usa un algoritmo recursivo para resolver un problema, se tienen que usar algunas de las localidades de la memoria para guardar el estado de las llamadas recursivas activas. Cuando la memoria de la computadora es un lujo, entonces, es más útil en algunos casos poder derivar algoritmos no recurrentes de los recurrentes.

Se puede emplear la estructura de datos *pila* para convertir un algoritmo recurrente en uno no recurrente, pero hay algunos casos cuando se puede hacer esta conversión más fácil y eficiente. Específicamente, se puede fácilmente convertir algoritmos que usan *recurrencia de cola*. Un algoritmo usa recurrencia de cola si este usa recursión lineal y el algoritmo hace una llamada recursiva como su última operación. Por ejemplo, el algoritmo `InvertirArreglo` usa recurrencia de cola.

Sin embargo, no es suficiente que la última sentencia en el método definición incluya una llamada recurrente. Para que un método use recurrencia de cola, la llamada recursiva deberá ser absolutamente la última parte que el método haga, a menos que se este en el caso base. Por ejemplo, el algoritmo `SumaLineal` no usa recurrencia de cola, aún cuando su última sentencia incluye una llamada recursiva. Esta llamada recursiva no es actualmente la última tarea que el método realiza. Después de esta recibe el valor regresado de la llamada recursiva, le agrega el valor de $A[n-1]$ y regresa su suma. Esto es, la última tarea que hace el algoritmo es una suma, y no una llamada recursiva.

Cuando un algoritmo emplea recurrencia de cola, se puede convertir el algoritmo recurrente en uno no recurrente, iterando a través de las llamadas recurrentes en vez de llamarlas a ellas explícitamente. Se ilustra este tipo de conversión revisitando el problema de invertir los elementos de un arreglo. El siguiente algoritmo no recursivo realiza la tarea de invertir los elementos iterando a través de llamadas recurrentes del algoritmo `InvertirArreglo`. Inicialmente se llama a este algoritmo como `InvertirArregloIterativo` $(A, 0, n-1)$.

Algoritmo InvertirArregloIterativo(A, i, j):
Entrada: Un arreglo A e índices enteros no negativos i y j.
Salida: La inversión de los elementos en A iniciando en el índice i y terminando en j.
mientras $i < j$ **Hacer**
  Intercambiar $A[i]$ y $A[j]$
  $i \leftarrow i + 1$
  $j \leftarrow j - 1$
Regresar

2.5.1. Recurrencia binaria

Cuando un algoritmo hace dos llamadas recurrentes, se dice que usa *recurrencia binaria*. Estas llamadas pueden, por ejemplo, ser usadas para resolver dos hojas similares de algún problema, como se hizo en la sección 2.5 para dibujar una regla inglesa. Como otra aplicación de recurrencia binaria, se revisitará el problema de sumar los n elementos de un arreglo de enteros A. En este caso, se pueden sumar los elementos de A: *(i)* recurrentemente los elementos en la

primera mitad de A; *(ii)* recurrentemente los elementos en la segunda mitad de A; y *(iii)* agregando estos dos valores juntos. Se dan los detalles en el siguiente algoritmo, el cual es inicialmente llamado como `SumaBinaria`$(A, 0, n)$.

Algoritmo SumaBinaria(A, i, n):
  Entrada: Un arreglo A de enteros y enteros i y n.
  Salida: La suma de los primeros n enteros en A iniciando en el índice i.
  Si $n = 1$ **Entonces**
    regresar $A[i]$
  regresar SumaBinaria$(A, i, \lceil n/2 \rceil)$+SumaBinaria$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

Para analizar el algoritmo `SumaBinaria`, se considera, por simplicidad, el caso cuando n es potencia de 2. La figura 2.17 muestra el trazado de la recursión de una ejecución del método `SumaBinaria`(0,8). Se etiqueta cada recuadro con los valores de los parámetros i y n, los cuales representan el índice inicial y el tamaño de la secuencia de los elementos que serán invertidos, respectivamente. Observar que las flechas en el trazado van de una recuadro etiquetado (i, n) a otro recuadro etiquetado $(i, n/2)$ o $(i + n/2, n/2)$. Esto es, el valor del parámetro n es dividido en dos en cada llamada recursiva. Por lo tanto, la profundidad de la recursión, esto es, el número máximo de instancias del método que están activas al mismo tiempo, es $1 + \log_2 n$. Entoces el algoritmo usa una cantidad de espacio adicional aproximadamente proporcional a su valor. El tiempo de ejecución del algoritmo es todavía proporcional a n, ya que cada recuadro es visitado en tiempo constante y hay $2n - 1$ recuadros.

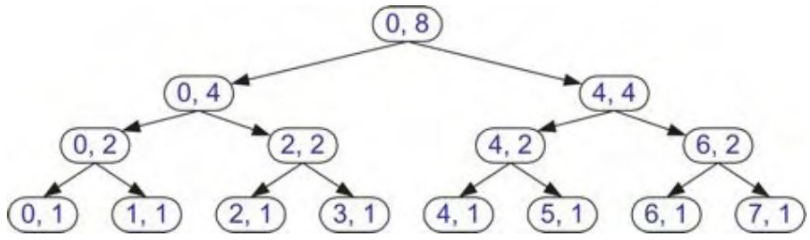

Figura 2.17: Trazado de la recursión de `SumaBinaria`(0,8).

Encontrar números de Fibonacci con recurrencia binaria

Se considera ahora el problema de computar el k-ésimo número de Fibonacci. Los números de Fibonacci están recursivamente definidos como:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ para } i < 1. \end{aligned}$$

Aplicando directamente la definición anterior, el algoritmo `FibBinario`, mostrado a continuación, encuentra la secuencia de los números de Fibonacci empleando recurrencia binaria.

Algoritmo FibBinario(k):
Entrada: Un entero k no negativo.
Salida: El k-ésimo número de Fibonacci F_k.
Si $k \leq 1$ **Entonces**
  regresar k
Si no
  regresar FibBinario($k-1$)+FibBinario($k-2$)

Desafortunadamente, a pesar de la definición de Fibonacci pareciendo una recursión binaria, usando esta técnica es ineficiente en este caso. De hecho, toma un número de llamadas exponencial para calcular el k-ésimo número de Fibonacci de esta forma. Específicamente, sea n_k el número de llamadas hechas en la ejecución de `FibBinario`(k). Entonces, se tienen los siguientes valores para las diferentes n_ks:

$$\begin{aligned} n_0 &= 1 \\ n_1 &= 1 \\ n_2 &= n_1 + n_0 + 1 = 3 \\ n_3 &= n_2 + n_1 + 1 = 5 \\ n_4 &= n_3 + n_2 + 1 = 9 \\ n_5 &= n_4 + n_3 + 1 = 15 \\ n_6 &= n_5 + n_4 + 1 = 25 \\ n_7 &= n_6 + n_5 + 1 = 41 \\ n_8 &= n_7 + n_6 + 1 = 67 \end{aligned}$$

Si se sigue hacia adelante el patrón, se ve que el número de llamadas es más del doble para un índice que esté dos índices anteriores. Esto es, n_4 es más del doble que n_2, n_5 es más del doble que n_3, y así sucesivamente. Por lo tanto, $n_k > 2^{k/2}$, lo que significa que $FibBinario(k)$ hace un número de llamadas que son exponenciales en k. En otras palabras, empleando recurrencia binaria para computar números de Fibonacci es muy ineficiente.

Números de Fibonacci con recurrencia lineal

El principal problema con la aproximación anterior, basada en recurrencia binaria, es que el cómputo de los números de Fibonacci es realmente un problema de recurrencia lineal. No es este un buen candidato para emplear recurrencia binaria. Se empleo recurrencia binaria por la forma como el k-ésimo número de Fibonacci, F_k, dependía de los dos valores previos, F_{k-1} y F_{k-2}. Pero se puede encontrar F_k de una forma más eficiente usando recurrencia lineal.

Para poder emplear recurrencia lineal, se necesita redefinir ligeramente el problema. Una forma de lograr esta conversión es definir una función recursiva que encuentre un par consecutivo de números Fibonacci (F_{k-1}, F_k). Entonces se puede usar el siguiente algoritmo de recurrencia lineal mostrado a continuación.

Algoritmo FibLineal(k):
Entrada: Un entero k no negativo.
Salida: El par de números Fibonacci (F_k, F_{k-1}).
Si $k \leq 1$ **Entonces**
  regresar $(0, k)$
Si no
  $(i, j) \leftarrow$ FibLineal$(k-1)$
  regresar $(j, i+j)$

El algoritmo dado muestra que usando recurrencia lineal para encontrar los números de Fibonacci es mucho más eficiente que usando recurrencia binaria. Ya que cada llama recurrente a `FibLineal` decrementa el argumento k en 1, la llamada original `FibLineal`(k) resulta en una serie de $k-1$ llamadas adicionales. Esto es, computar el k-ésimo número de Fibonacci usando recurrencia lineal requiere k llamadas al método. Este funcionamiento es significativamente más rápido que el tiempo exponencial necesitado para el algoritmo basado en recurrencia binaria. Por lo tanto, cuando se usa recurrencia binaria, se debe primero tratar de particionar completamente el problema en dos, o se debería estar seguro que las llamadas recursivas que se traslapan son realmente necesarias.

Usualmente, se puede eliminar el traslape de llamadas recursivas usando más memoria para conservar los valores previos. De hecho, esta aproximación es una parte central de una técnica llamada *programación dinámica*, la cual está relacionada con la recursión.

2.5.2. Recurrencia múltiple

Generalizando de la recurrencia binaria, se usa *recurrencia múltiple* cuando un método podría hacer varias llamadas recursivas múltiples, siendo ese número potencialmente mayor que dos. Una de las aplicaciones más comunes de este tipo de recursión es usada cuando se quiere enumerar varias configuraciones en orden para resolver un acertijo combinatorio. Por ejemplo, el siguiente es un acertijo de suma:

```
seis+de+enero=reyes
```

Para resolver el acertijo, se necesita asignar un dígito único, esto es, 0,1,. . . ,9, a cada letra en la ecuación, para poder hacer la ecuación verdadera. Típicamente, se resuelve el acertijo usando observaciones humanas del acertijo particular que se está intentando resolver para eliminar configuraciones hasta que se pueda trabajar con las configuraciones restantes, probando la correctez de cada una.

Sin embargo, si el número de configuraciones posibles no es muy grande, se puede usar una computadora simplemente para enumerar todas las posibilidades y probar cada una, sin emplear observaciones humanas. Además, tal algoritmo puede usar recurrencia múltiple para trabajar a través de todas las configuraciones en una forma sistemática. Se proporciona enseguida pseudocódigo para tal algoritmo. Para conservar la descripción general suficiente para ser usado con otros acertijos, el algoritmo enumera y prueba todas las secuencias de longitud

k sin repeticiones de los elementos de un conjunto dado U. Se construyen las secuencias de k elementos mediante los siguientes pasos:

1. Generar recursivamente las secuencias de $k-1$ elementos.
2. Agregar a cada una de tales secuencias un elemento que no esté contenido dentro de esta.

En toda la ejecución del algoritmo, se usa el conjunto U para saber los elementos no contenidos en la secuencia actual, por lo que un elemento e no ha sido usado aún sí y sólo sí e está en U.

Otra forma de ver el algoritmo siguiente es que este enumera cada subconjunto ordenado posible de tamaño, y prueba cada subconjunto para ser una posible solución al acertijo.

Para acertijos de sumas, $U = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ y cada posición en la secuencia corresponde a una letra dada. Por ejemplo, la primera posición podría ser b, la segunda o, la tercera y, y así sucesivamente.

```
Algoritmo ResolverAcertijo(k, S, U):
  Entrada: Un entero k, secuencia S, y conjunto U.
  Salida: Una enumeración de todas las extensiones k-longitud para S
          usando elementos en U sin repeticiones.
  Para cada e en U Hacer
    Remover e de U { e está ahora siendo usado }
    Agregar e al final de S
    Si k = 1 Entonces
      Probar si S es una configuración que resuelva el acertijo
      Si S resuelve el acertijo Entonces
        regresar "Solución encontrada: " S
    Si no
      ResolverAcertijo(k − 1, S, U)
    Agregar e de regreso a U { e está ahora sin uso }
    Remover e del final de S
```

Capítulo 3

Herramientas de análisis

3.1. Funciones

Se comenta brevemente, en esta sección, las siete funciones más importantes usadas en el análisis de algoritmos.

3.1.1. La función constante

La función más simple que se emplea es la *función constante*. Esta es la función

$$f(n) = c$$

para alguna constante fija c, tal como $c = 5$, $c = 27$, o $c = 2^{10}$. Con esta función, para cualquier argumento n, la función constante $f(n)$ asigna el valor c, es decir, no importa cual es el valor de n, $f(n)$ siempre será igual al valor constante c.

Como se emplean generalmente funciones enteras en el análisis, la función constante más usada es $g(n) = 1$, y cualquier otra función constante, $f(n) = c$, puede ser escrita como una constante c veces $g(n)$, es decir, $f(n) = cg(n)$.

Esta función caracteriza el número de pasos requeridos para hacer una operación básica en la computadora, como la suma de dos números, asignar un valor a una variable, o comparar dos números.

3.1.2. La función logarítmica

La *función logarítmica*, $f(n) = \log_b n$, para alguna constante $b > 1$ es ubicua, ya que aparece con bastante frecuencia en el análisis de estructuras de datos y algoritmos. Esta función está definida como sigue:

$$x = log_b n \text{ sí y sólo sí } b^x = n$$

Por definición, $\log_b 1 = 0$. El valor de b es conocido como la *base* del logaritmo.

Para encontrar el valor exacto de esta función para cualquier entero n se requiere usar cálculo, pero se puede usar una buena aproximación que es buena

para el análisis sin emplear cálculo. En particular, se puede calcular el entero más pequeño que sea mayor o igual a $\log_a n$, ya que este número es igual al número de veces que se puede dividir n por a hasta que se obtenga un número menor que o igual a 1. Por ejemplo, $log_3 27$ es 3, ya que $27/3/3/3 = 1$. De igual modo $\log_4 64$ es 3, porque $64/4/4/4 = 1$, y la aproximación a $\log_2 12$ es 4, ya que $12/2/2/2/2 = 0.75 \leq 1$. La aproximación base dos surge en el análisis de algoritmos, ya que es una operación común en los algoritmos "divide y vencerás" por que repetidamente dividen la entrada por la mitad.

Además, como las computadoras almacenan los enteros en binario, la base más común para la función logarítmica en ciencias de la computación es dos, por lo que típicamente no se indica cuando es dos, entonces se tiene,

$$\log n = \log_2 n.$$

Hay algunas reglas importantes para logaritmos, parecidas a las reglas de exponentes.

Proposición 3.1: *(reglas de logaritmos) dados números reales $a > 0$, $b > 1$, $c > 0$, y $d > 1$, se tiene:*

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

Para usar una notación abreviada, se usará $\log^c n$ para denotar la función $(\log n)^c$.

3.1.3. La función lineal

Otra función simple e importante es la *función lineal* ,

$$f(n) = n$$

Esto es, dado un valor de entrada n, la función lineal f asigna el propio valor n.

Esta función surge en el análisis de algoritmos cuando se tiene que hacer una operación básica para cada uno de los n elementos. Por ejemplo, comparar un número x con cada uno de los elementos de una arreglo de tamaño n requerirá n comparaciones. La función lineal también representa el mejor tiempo de ejecución que se espera lograr para cualquier algoritmo que procese una colección de n objetos que no estén todavía en la memoria de la computadora, ya que introducir los n objetos requiere n operaciones.

3.1.4. La funcion N-Log-N

La función

$$f(n) = n \log n$$

es la que asigna a una entrada n el valor de n veces el logaritmo de base dos de n. Esta función crece un poco más rápido que la función lineal y es mucho más lenta que la función cuadrática. Por lo tanto, si se logra mejorar el tiempo de ejecución de algún problema desde un tiempo cuadrático a n-log-n, se tendrá un algoritmo que corra más rápido en general.

3.1.5. La función cuadrática

La *función cuadrática* también aparece muy seguido en el análisis de algoritmos,

$$f(n) = n^2$$

Para un valor de entrada n, la función f asigna el producto de n con ella misma, es decir, "n cuadrada".

La razón principal por la que la función cuadrática aparece en el análisis de algoritmos es porque hay muchos algoritmos que tienen ciclos anidados, donde el ciclo interno realiza un número lineal de operaciones y el ciclo externo es realizado un número lineal de veces. Así, en tales casos, el algoritmo realiza $n \cdot n = n^2$ operaciones.

Ciclos anidados y la función cuadrática

La función cuadrática puede también surgir en los ciclos anidados donde la primera iteración de un ciclo usa una operación, la segunda usa dos operaciones, la tercera usa tres operaciones, y así sucesivamente, es decir, el número de operaciones es

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

En otras palabras, este es el número total de operaciones que serán hechas por el ciclo anidado si el número de operaciones realizadas dentro del ciclo se incrementa por uno con cada iteración del ciclo exterior.

Se cree que Carl Gauss uso la siguiente identidad para resolver el problema que le habían dejado de encontrar la suma de enteros desde el 1 hasta el 100.
Proposición 3.2: *para cualquier entero $n \geq 1$, se tiene*

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}.$$

Lo que se puede aprender de este resultado es que si ejecuta un algoritmo con ciclos anidados tal que las operaciones del ciclo anidado se incrementan en uno cada vez, entonces el número total de operaciones es cuadrático en el número de veces, n, que se hace el ciclo exterior. En particular, el número de operaciones es $n^2/2 + n/2$, en este caso, lo cual es un poco más que un factor constante $(1/2)$ veces la función cuadrática n^2.

3.1.6. La función cúbica y otras polinomiales

De las funciones que son potencias de la entrada, se considera la *función cúbica* ,

$$f(n) = n^3$$

la cual asigna a un valor de entrada n el producto de n con el mismo tres veces. Esta función aparece con menor frecuencia en el contexto del análisis de algoritmos que las funciones constante, lineal o cuadrática pero aparece de vez en vez.

Polinomiales

Algunas de las funciones que se han revisado previamente pueden ser vistas como una parte de una clase de funciones más grandes, las *polinomiales* .

Una función *polinomial* es una función de la forma,

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

donde $a_0, a_1, \ldots, a_d$ son constantes, llamados los coeficientes de la polinomial, y $a_d \neq 0$. El entero d, el cual indica la potencia más grande en el polinomio, es llamado el grado del polinomio.

Sumatorias

Una notación que aparece varias veces en el análisis de las estructuras de datos y algoritmos es la *sumatoria* , la cual está definida como sigue:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b),$$

donde a y b son enteros y $a \leq b$.

Usando una sumatoria se puede escribir un polinomio $f(n)$ de grado d con coeficientes $a_0, \ldots, a_d$ como

$$f(n) = \sum_{i=0}^{d} a_i n^i.$$

3.1.7. La función exponencial

Otra función usada en el análisis de algoritmos es la *función exponencial*

$$f(n) = b^n,$$

donde b es una constante positiva, llamada la base, y el argumento n es el exponente, es decir, la función $f(n)$ asigna al argumento de entrada n el valor obtenido de multiplicar la base b por sí misma n veces. En análisis de algoritmos, la base más común para la función exponencial es $b = 2$. Por ejemplo, si se

tiene un ciclo que inicia realizando una operación y entonces dobla el número de operaciones hechas con cada iteración, entonces el número de operaciones realizadas en la n-ésima iteración es 2^n. Además, una palabra entera conteniendo n bits puede representar todos los enteros no negativos menores que 2^n. La función exponencial también será referida como *función exponente*.

En ocasiones se tienen otros exponentes además de n, por lo que es útil conocer unas cuantas reglas para trabajar con exponentes.

Proposición 3.3: *(reglas de exponentes) dados enteros positivos a, b, y c, se tiene*

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

Sumas geométricas

Cuando se tiene un ciclo donde cada iteración toma un factor multiplicativo mayor que la previa, se puede analizar este ciclo usando la siguiente proposición.

Proposición 3.4: *Para cualquier entero $n \geq 0$ y cualquier número real a tal que $a > 0$ y $a \neq 1$, considerar la sumatoria*

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots a^n$$

recordando que $a^0 = 1$ si $a > 0$. Esta suma es igual a

$$\frac{a^{n+1} - 1}{a - 1}.$$

3.2. Análisis de algoritmos

La herramienta de análisis primario que se usará involucra la caracterización de los tiempos de ejecución de los algoritmos y las estructuras de datos, así como los espacios usados. El tiempo de ejecución es una medida natural de la "bondad", ya que el tiempo es un recurso precioso –las soluciones computacionales deberán correr tan rápido como sea posible.

En general, el tiempo de ejecución de un algoritmo o de un método de una estructura de datos se incrementa con el tamaño de la entrada, sin embargo podría variar para entradas diferentes con el mismo tamaño. También, el tiempo de ejecución es afectado por el ambiente del hardware (como se refleja en el procesador, frecuencia del reloj, memoria, disco, etc.) y el ambiente de software (como se refleja en el sistema operativo, lenguaje de programación, compilador, intérprete, etc.) en el cual el algoritmo está implementado, compilado y ejecutado. Considerando las posibles variaciones que pueden darse por los diferentes factores ambientales, se intenta enfocarse en la relación entre el tiempo de ejecución de un algoritmo y el tamaño de su entrada.

Se está interesado en caracterizar el tiempo de ejecución de un algoritmo como una función del tamaño de la entrada. Entonces se requiere saber cual es una forma apropiada de medirla.

3.2.1. Estudios experimentales

Si un algoritmo ha sido implementado, se puede estudiar su tiempo de ejecución usando varias pruebas de entrada y guardando el tiempo usado en cada ejecución. Para hacer estas mediciones de forma precisa se pueden hacer llamadas al sistema que están construidas en el lenguaje o el sistema operativo, por ejemplo, usando método `System.currentTimeMillis()`. Se está interesado en determinar la dependencia general del tiempo de ejecución de acuerdo al tamaño de la entrada, para lo cual se deben hacer varios experimentos con muchas entradas de prueba diferentes de varios tamaños. Entonces se pueden visualizar los resultados de tales experimentos graficando el rendimiento de cada ejecución del algoritmo como un punto con coordenada x igual al tamaño de la entrada, n, y la coordenada y igual al tiempo de ejecución, t. Este análisis requiere que se escojan buenas entradas de muestra.

Las limitaciones que tienen los estudios experimentales son:

- Los experimentos sólo pueden hacerse en un conjunto limitado de pruebas de entrada, por lo que quedarán entradas no incluidas que podrían ser importantes.
- Es difícil comparar los tiempos experimentales de ejecución de dos algoritmos a menos que los experimentos sean hechos en el mismo hardware y software.
- Se tiene que implementar de forma completa y ejecutar un algoritmo para poder estudiar sus tiempos de ejecución experimentalmente.

3.2.2. Operaciones primitivas

Si se desea analizar un algoritmo particular, se puede realizar un análisis directo en el pseudo-código de alto nivel en vez de realizar experimentos del tiempo de ejecución. Se define un conjunto de *operaciones primitivas* tales como las siguientes:

- Asignar un valor a una variable.
- Llamar un método.
- Realizar una operación aritmética.
- Comparar dos números.
- Indexar en un arreglo.
- Seguir la referencia a un objeto.
- Regresar desde un método.

Conteo de operaciones primitivas

Una operación primitiva corresponde a una instrucción de bajo nivel con un tiempo de ejecución que es constante. Se cuentan cuantas operaciones primitivas son ejecutadas, y se usa este número t como una medida del tiempo de ejecución del algoritmo.

El conteo de las operaciones está correlacionado a un tiempo de ejecución actual en un computadora particular, para cada operación primitiva corresponde a una instrucción constante en el tiempo, y sólo hay número fijo de operaciones primitivas. El número, t, de operaciones primitivas que un algoritmo realiza será proporcional al tiempo de ejecución actual de ese algoritmo.

Un algoritmo podría correr más rápido con algunas entradas que con otras del mismo tamaño. Por lo tanto, se podría desear expresar el tiempo de ejecución de un algoritmo como la función del tamaño de la entrada obtenida de tomar el promedio de todas las posibles entradas del mismo tamaño. Desafortunadamente, el *análisis promedio* suele ser muy difícil, ya que requiere definir una distribución de probabilidad sobre el conjunto de entradas, lo cual es una tarea difícil.

Enfoque en el peor caso

El *análisis del peor caso* es mucho más fácil que el caso promedio, ya que sólo requiere la habilidad de identificar el peor caso de entrada, el cual es frecuentemente simple. Con esta aproximación también se llega a mejores algoritmos, ya que logrando que el algoritmo se comporte bien para el peor caso, entonces para cualquier entrada también lo hará.

3.2.3. Notación asintótica

En general, cada paso básico en una descripción en pseudo-código o en una implementación de lenguaje de alto nivel corresponde a un número pequeño de operaciones primitivas, excepto por las llamadas a métodos. Por lo tanto se puede hacer un análisis simple de un algoritmo que estime el número de operaciones primitivas ejecutadas hasta un factor constante, por lo pasos en el pseudo-código.

En el análisis de algoritmos se enfoca en la relación de crecimiento del tiempo de ejecución como una función del tamaño de la entrada n, tomando una aproximación o una "foto grande".

Se analizan los algoritmos usando una notación matemática para funciones que no tiene en cuenta los factores constantes. Se caracterizan los tiempos de ejecución de los algoritmos usando funciones que mapean el tamaño de la entrada, n, a valores que corresponden al factor principal que determina la relación de crecimiento en términos de n. Esta aproximación permite enfocarse en los aspectos de "foto grande" del tiempo de ejecución de un algoritmo.

La notación "O-Grande"

Sean $f(n)$ y $g(n)$ funciones que mapean enteros no negativos a números reales. Se dice que $f(n)$ es $O(g(n))$ si hay una constante $c > 0$ y una constante entera $n_0 \geq 1$ tal que

$$f(n) \leq cg(n), \text{ para } n \geq n_0.$$

Esta definición es frecuentemente referida como la notación "O-grande", por eso en ocasiones se dice "$f(n)$ es *O-grande* de $g(n)$". Alternativamente, se puede también decir "$f(n)$ *es orden de* $g(n)$". La definición se ilustra en la figura 3.1

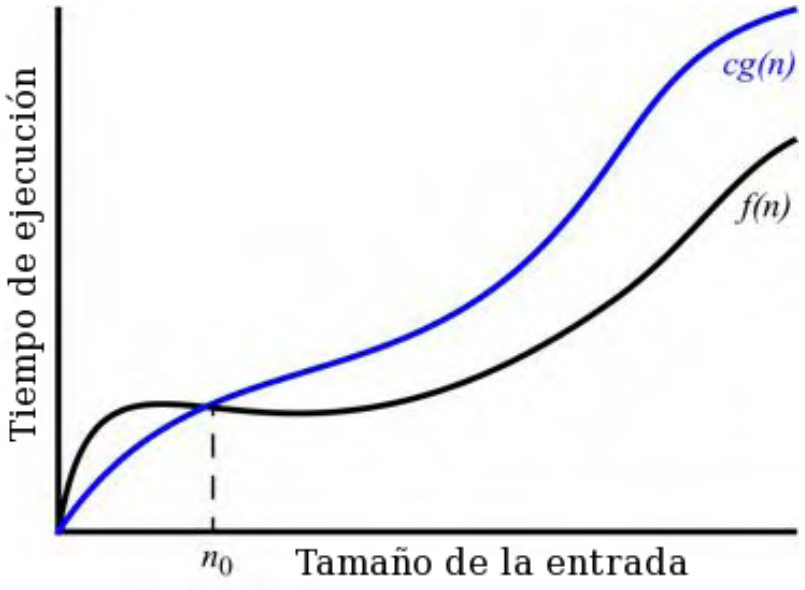

Figura 3.1: Ilustración de la notación "O-grande". La función $f(n)$ es $O(g(n))$ ya que $f(n) \leq c \cdot g(n)$ cuando $n \geq n_0$

Ejemplo 3.1. *La función* $8n - 2$ *es* $O(n)$

Justificación. Por la definición de O-grande, se necesita encontrar una constante real $c > 0$ y una constante entera $n_0 \geq 1$ tal que $8n - 2 \leq cn$ para cada entero $n \geq n_0$. Una posible opción es $c = 8$ y $n_0 = 1$ de la infinidad de opciones disponibles ya que cualquier número real mayor que o igual a 8 funciona para c, y cualquier entero mayor que o igual a 1 trabajará para n_0.

La notación O-grande nos permite decir que una función $f(n)$ es "menor que o igual a" otra función $g(n)$ hasta un factor constante y en el sentido asintótico conforme n tiene a infinito. Esto se debe del hecho de que la definición usar "$\leq$" para comparar $f(n)$ con $g(n)$ veces una constante, c, para los casos asintóticos cuando $n \geq n_0$.

Propiedades de la notación O-Grande

La notación O-grande permite ignorar los factores constantes y los términos de orden menor y enfocarse en los componentes principales de una función que afectan su crecimiento.

Ejemplo 3.2. $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *es* $O(n^4)$

Justificación: Observar que $5n^4+3n^3+2n^2+4n+1 \leq (5+3+2+4+1)n^4 = cn^4$, para $c = 15$, cuando $n \geq n_0 = 1$

Se puede caracterizar la relación de crecimiento de cualquier función polinomial.

Proposición 3.5: *Si* $f(n)$ *es un polinomio de grado* n*, esto es*

$$f(n) = a_0 + a_1 n + \cdots a_d n^d,$$

y $a_d > 0$*, entonces* $f(n)$ *es* $O(n^d)$.

Justificación: observar que, para $n \geq 1$, se tiene $1 \leq n \leq n^2 \cdots n^d$ por lo tanto

$$a_0 + a_1 n + \cdots a_d n_d \leq (a_0 + a_1 + \cdots a_d)n_d.$$

Así, se puede mostrar que $f(n)$ es $O(n^d)$ definiendo $c = a_0 + a_1 + \cdots + a_d$ y $n_0 = 1$.

Así, el término de mayor grado en una función polinomial es el término que determina la razón de crecimiento asintótico del polinomio. Se muestran a continuación ejemplos adicionales, en donde se combinan las funciones vistas al inicio del capítulo.

Ejemplo 3.3. $5n^2 + 3n \log n + 2n + 5$ es $O(n^2)$.

Justificación: $5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2 = cn^2$. para $c = 15$ cuando $n \geq n_0 = 2$. Observar que $n \log n$ es cero para $n = 1$.

Ejemplo 3.4. $20n^3 + 10n \log n + 5$ es $O(n^3)$.

Justificación: $20n^3 + 10n \log n + 5 \leq 35n^3,$ para $n \geq 1$.

Ejemplo 3.5. $3 \log n + 2$ es $O(\log n)$.

Justificación: $3 \log n + 2 \leq 5 \log n,$ para $n \geq 2$. La razón por la que se usa $n \geq 2 = n_0$ es porque $\log n$ es cero para $n = 1$.

Ejemplo 3.6. $2^n + 2$ es $O(2^n)$.

Justificación: $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$; por lo tanto, se puede tomar $c = 4$ y $n_0 = 1$ en este caso.

Ejemplo 3.7. $2n + 100 \log n$ es $O(n)$.

Justificación: $2n+100 \log n \leq 102n$ para $n \geq 2 = n_0$; por lo tanto, se puede tomar a $c = 102$ en este caso.

Caracterización de funciones en términos más simples

Se podría usar la notación O-grande para caracterizar una función tan cercana como sea posible. Para la función $f(n) = 4n^3 + 3n^2$ es cierto que es $O(n^5)$ o aún $O(n^4)$, pero es más preciso decir que $f(n)$ es $O(n^3)$. También se considera de mal gusto incluir factores constantes y términos de orden inferior en la notación O-grande. No está de moda decir que la función $2n^2$ es $O(4n^2 + 6n \log n)$, sin

embargo esto es completamente correcto. Se debe intentar describir la función en O-grande en los términos más simples.

Las funciones descritas en la sección 3 son las funciones más comunes usadas en conjunción con la notación O-grande para caracterizar los tiempos de ejecución y el espacio usado de los algoritmos. Además, se usan los nombres de estas funciones para referirse a los tiempos de ejecución de los algoritmos que están caracterizando. Por ejemplo, se podría decir que un algoritmo que se ejecuta con tiempo $4n^2 + n \log n$ en el peor caso como un algoritmo *tiempo cuadrático*, ya que corre en tiempo $O(n^2)$. De igual forma, un algoritmo ejecutándose en tiempo a lo más $5n + 20 \log n + 4$ podría ser llamado un algoritmo lineal.

Omega-grande

La notación O-grande proporciona una forma asintótica de decir que una función es "menor que o igual a" otra función, la siguiente notación da una forma asintótica para indicar que una función crece a un ritmo que es "mayor que o igual a" otra función.

Sean $f(n)$ y $g(n)$ funciones que mapean enteros no negativos a números reales. Se dice que $f(n)$ es $\Omega(g(n))$ (pronunciada "f(n) es Omega-grande de $g(n)$") si $g(n)$ es $O(f(n))$, esto es, existe una constante real $c > 0$ y una constante entera $n_0 \geq 1$ tal que

$$f(n) \geq cg(n), \text{ para } n \geq n_0.$$

Esta definición permite decir asintóticamente que una función es mayor que o igual a otra, hasta un factor constante.

Ejemplo 3.8. *La función* $3n \log n + 2n$ *es* $\Omega(n \log n)$

Justificación. $3n \log n + 2n \geq 3n \log n$, para $n \geq 2$.

Tetha-grande

Además, hay una notación que permite decir que dos funciones crecen a la misma velocidad, hasta unos factores constantes. Se dice que $f(n)$ es $\Theta(g(n))$ (pronunciado "$f(n)$ es Tetha-grande de $g(n)$") si $f(n)$ es $O(g(n))$ y $f(n)$ es $\Omega(g(n))$, esto es, existen constantes reales $c' > 0$ y $c'' > 0$, y una constante entera $n_0 \geq 1$ tal que

$$c'g(n) \leq f(n) \leq c''g(n), \text{ para } n \geq n_0.$$

Ejemplo 3.9. $3n \log n + 4n + 5 \log n$ *es* $\Theta(n \log n)$

Justificación. $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5)n \log n$ para $n \geq 2$.

3.2.4. Análisis asintótico

Suponiendo dos algoritmos que resuelven el mismo problema: un algoritmo A, el cual tiene tiempo de ejecuión $O(n)$, y un algoritmo B, con tiempo de ejecución $O(n^2)$. Se desea sabe cuál algoritmo es mejor. Se sabe que n es $O(n^2)$, lo cual implica que el algoritmo A es asintóticamente mejor que el algoritmo B, sin

embargo para un valor pequeño de n, B podría tener un tiempo de ejecución menor que A.

Se puede emplear la notación O-grande para ordenar clases de funciones por su velocidad de crecimiento asintótico. Las siete funciones son ordenadas por su velocidad de crecimiento incremental en la siguiente secuencia, esto es, si una función $f(n)$ precede a una función $g(n)$ en la secuencia, entonces $f(n)$ es $O(g(n))$:

$$1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n$$

Se muestra enseguida la tasa de crecimiento de algunas funciones importantes a continuación:

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

La siguiente table explora el tamaño máximo permitido para una instancia de entrada que es procesada por un algoritmo en 1 segundo, 1 minuto, y 1 hora. Se muestra la importancia del diseño de un buen algoritmo, porque un algoritmo lento asintóticamente es derrotado en una ejecución grande por un algoritmo más rápido asintóticamente, aún si el factor constante para el algoritmo más rápido es malo.

Tiempo	Tamaño máximo del problema (n)		
Ejecución (μs)	1 segundo	1 minuto	1 hora
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

La importancia del diseño de un buen algoritmo va más allá de que puede ser resuelto eficientemente en una computadora dada. Como se muestra en la siguiente tabla aún si se logra una aceleración dramática en el hardware, todavía no se puede superar la desventaja de un algoritmo asintóticamente lento. La tabla muestra el tamaño máximo del nuevo problema que se calcula para una cantidad fija de tiempo, suponiendo que los algoritmos con los tiempos de ejecución dados son ejecutados ahora en una computadora 256 veces más rápida que la previa.

Tiempo de ejecución	Tamaño máximo del nuevo problema
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

Capítulo 4

Pilas y colas

4.1. Genéricos

Iniciando con la versión 5.0 de Java se incluye una *estructura genérica* para usar tipos abstractos en una forma que evite muchas conversiones de tipos. Un *tipo genérico* es un tipo que no está definido en tiempo de compilación, pero queda especificado completamente en tiempo de ejecución. La estructura genérica permite definir una clase en términos de un conjunto de *parámetros de tipo formal*, los cuales podrían ser usados, por ejemplo, para abstraer los tipos de algunas variables internas de la clase. Los paréntesis angulares son usados para encerrar la lista de los parámetros de tipo formal. Aunque cualquier identificador válido puede ser usado para un parámetro de tipo formal, los nombres de una sola letra mayúscula por convención son usados. Dada una clase que ha sido definida con tales tipos parametrizados, se instancia un objeto de esta clase usando los *parámetros de tipo actual* para indicar los tipos concretos que serán usados.

En el listado 4.1, se muestra la clase `Par` que guarda el par llave-valor, donde los tipos de la llave y el valor son especificados por los parámetros `L` y `V`, respectivamente. El método `main` crea dos instancias de esta clase, una para un par `String-Integer` (por ejemplo, para guardar un dimensión y su valor), y otra para un par `Estudiante-Double` (por ejemplo, para guardar la calificación dada a un estudiante).

```
1  public class Par<L, V> {
2
3    L llave;
4    V valor;
5
6    public void set(L l, V v) {
7      llave = l;
8      valor = v;
9    }
10   public L getLlave() { return llave; }
11   public V getValor() { return valor; }
12   public String toString() {
13     return "[" + getLlave() + ", " + getValor() + "]";
14   }
```

```
15
16    public static void main (String[] args) {
17      Par<String,Integer> par1 = new  Par<String,Integer>();
18      par1.set(new String("altura"), new Integer(2));
19      System.out.println(par1);
20      Par<Estudiante,Double> par2 = new  Par<Estudiante,Double>();
21      par2.set(new Estudiante("8403725A","Hector",14), new Double(9.5));
22      System.out.println(par2);
23    }
24  }
```

Listado 4.1: Clase Par.java

En la clase `Par` se hace uso de la clase `Estudiante` que se muestra en el listado 4.2.

```
1   public class Estudiante {
2     String matricula;
3     String nombre;
4     int edad;
5
6     public Estudiante (String m, String n, int e) { // constructor simple
7       matricula = m;
8       nombre = n;
9       edad = e;
10    }
11
12    protected int horasEstudio() { return edad/2; } // solo una suposición
13    public String getMatricula() { return matricula; } // matrícula
14    public String getNombre() { return nombre; } // nombre del estudiante
15    public int getEdad() { return edad; } // edad
16
17    public boolean equals (Estudiante otro) { // ¿es la misma persona?
18      return (matricula.equals(otro.getMatricula())); // comparar matrículas
19    }
20
21    public String toString() { // para impresión
22      return "Estudiante(Matrícula: " + matricula +
23        ", Nombre: " + nombre +
24        ", Edad: " + edad + ")";
25    }
26  }
```

Listado 4.2: Clase Estudiante.java

La salida de la ejecución de la clase `Par` se muestra enseguida:

```
[altura, 2]
[Estudiante(Matricula: 8403725A, Nombre: Hector, Edad: 14), 9.5]
```

En el ejemplo, el parámetro del tipo actual puede ser un tipo arbitrario. Para restringir el tipo de parámetro actual se puede usar la cláusula `extends`, como se muestra enseguida, donde la clase `EstudianteParDirectorioGenerico` está definido en términos de un parámetro de tipo genérico `P`, parcialmente especificando que este extiende la clase `Estudiante`.

```
1   public class EstudianteParDirectorioGenerico<E extends Estudiante> {
2
3     //... las variables de instancia podrían ir aquí ...
4
5     public EstudianteParDirectorioGenerico() { /* el constructor
6       por defecto aquí va */ }
```

```
7
8     public void insertar (E estudiante, E otro) { /* código
9       para insertar va aqui */ }
10
11    public P encontrarOtro (E estudiante) { return null; } // para encontrar
12
13    public void remover (E estudiante, P otro) { /* código
14      para remover va aquí */ }
15  }
```

Listado 4.3: Restricción del parámetro usando la cláusula `extends`

Hay una advertencia importante relacionada a los tipos genéricos, a saber, que los elementos guardados en un arreglo no pueden ser de un tipo variable o parametrizado. Java permite que un arreglo sea definido con un tipo parametrizado, pero no permite que un tipo parametrizado sea usado para crear un nuevo arreglo. Afortunadamente, Java permite para un arreglo definido con un tipo parametrizado sea inicializado con un nuevo arreglo creado, no parametrizado, como se muestra en el listado 4.4. Aún así, este mecanismo tardío causa que el compilador de Java emita una advertencia, porque esto no es 100 % tipeo seguro. Se ilustra este punto en lo siguiente:

```
1   public static void main(String[] args) {
2
3     Par<String,Integer>[] a = new Par[10]; // correcto, pero da una advertencia
4
5     Par<String,Integer>[] b = new Par<String,Integer>[10]; // incorrecto
6
7     a[0] = new Par<String,Integer>(); // esto está completamente bien
8
9     a[0].set("Gato",10); // esta y la siguiente sentencia tambien están bien
10
11    System.out.println("Primer par es "+a[0].getLlave()+","+a[0].getValor());
12
13  }
```

Listado 4.4: Ejemplo que muestra que un tipo parametrizado sea usado para crear un nuevo arreglo.

4.2. Pilas

Una *pila* (*stack*) es una colección de objetos que son insertados y removidos de acuerdo al principio *último en e1Gntrar, primero en salir*, *LIFO* (*last-in first-out*). Los objetos pueden ser insertados en una pila en cualquier momento, pero solamente el más reciente insertado, es decir, el "último" objeto puede ser removido en cualquier momento. Una analogía de la pila es el dispensador de platos que se encuentra en el mobiliario de alguna cafetería o cocina. Para este caso, las operaciones fundamentales involucran "push" (empujar) platos y "pop" (sacar) platos de la pila. Cuando se necesita un nuevo plato del dispensador, se saca el plato que está encima de la pila, y cuando se agrega un plato, se empuja este hacia abajo en la pila para que se convierta en el nuevo plato de la cima. Otro ejemplo son los navegadores Web de internet que guardan las direcciones de los sitios recientemente visitados en una pila. Cada vez que un usuario visita

un nuevo sitio, esa dirección del sitio es empujada en la pila de direcciones. El navegador, mediante el botón *Volver atrás*, permite al usuario sacar los sitios previamente visitados.

4.2.1. El tipo de dato abstracto pila

Las pilas son las estructuras de datos más simples, no obstante estas también están entre las más importantes, ya que son usadas en un sistema de diferentes aplicaciones que incluyen estructuras de datos mucho mas sofisticadas. Formalmente, una pila es un tipo de dato abstracto que soporta los siguientes dos métodos:

`push(e)`:inserta el elemento e, para que sea la cima de la pila.
`pop()`:quita el elemento de la cima de la pila y lo regresa; un error ocurre si la pila está vacía.

Adicionalmente, también se podrían definir los siguientes métodos:

`size()`:regresa el número de elementos en la pila.
`isEmpty()`:regresa un booleano indicando si la pila está vacía.
`top()`:regresa el elemento de la cima de la pila, sin removerlo; un error ocurre si la pila está vacía.

La siguiente tabla muestra una serie de operaciones en la pila y su efecto en una pila de enteros inicialmente vacía, observar que los elementos son metidos y sacados por el mismo lado, el lado derecho.

Operación	Salida	Contenido
`push(5)`	–	(5)
`push(3)`	–	(5,3)
`pop()`	3	(5)
`push(7)`	–	(5,7)
`pop()`	7	(5)
`top()`	5	(5)
`pop()`	5	()
`pop()`	"`error`"	()
`isEmpty()`	`true`	()
`push(9)`	–	(9)
`push(7)`	–	(9,7)
`push(3)`	–	(9,7,3)
`push(5)`	–	(9,7,3,5)
`size()`	4	(9,7,3,5)
`pop()`	5	(9,7,3)
`push(8)`	–	(9,7,3,8)
`pop()`	8	(9,7,3)
`pop()`	3	(9,7)

Una interfaz pila en Java

Debido a su importancia, la estructura de datos pila está incluida como una clase en el paquete `java.util`. La clase `java.util.Stack` es una estructura que guarda objetos genéricos Java e incluye, entre otros, los métodos `push()`, `pop()`, `peek()` (equivalente a `top()`), `size()`, y `empty()` (equivalente a `isEmpty()`). Los métodos `pop()` y `peek()` lanzan la excepción `EmptyStackException` del paquete `java.util` si son llamados con pilas vacías. Mientras es conveniente solo usar la clase incluida `java.util.Stack`, es instructivo aprender como diseñar e implementar una pila "desde cero".

Implementar un tipo de dato abstracto en Java involucra dos pasos. El primer paso es la definición de una *interfaz de programación de aplicaciones* o ***Application Programming Interface*** (API), o simplemente *interfaz*, la cual describe los nombres de los métodos que la estructura abstracta de datos soporta y como tienen que ser declarados y usados.

Además, se deben definir excepciones para cualquier cualquier condición de error que pueda originarse. Por ejemplo, la condición de error que ocurre cuando se llama al método `pop()` o `top()` en una cola vacía es señalado lanzando una excepción de tipo `EmptyStackException`, la cual está definida en el listado 4.5

```
1  /**
2   * Excepción Runtime lanzada cuando se intenta hacer una operación top
3   * o pop en una cola vacía.
4   */
5
6  public class EmptyStackException extends RuntimeException {
7    public EmptyStackException(String err) {
8      super(err);
9    }
10 }
```

Listado 4.5: Excepción lanzada por los métodos `pop()` y `top()` de la interfaz pila cuando son llamados con una pila vacía.

Una interfaz completa para el ADT pila está dado en el listado 4.6. Esta interfaz es muy general ya que esta específica que elementos de cualquier clase dada, y sus subclases, pueden ser insertados en la pila. Se obtiene esta generalidad mediante el concepto de *genéricos* (sección 4.1).

Para que un ADT dado sea para cualquier uso, se necesita dar una clase concreta que implemente los métodos de la interfaz asociada con ese ADT. Se da una implementación simple de la interfaz `Stack` en la siguiente subsección.

```
1   /**
2    * Interfaz para una pila: una colección de objetos que son insertados
3    * y removidos de acuerdo al principio último en entrar, primero en salir.
4    * Esta interfaz incluye los métodos principales de java.util.Stack.
5    *
6    * @see EmptyStackException
7    */
8
9   public interface Stack<E> {
10
11   /**
12    * Regresa el número de elementos en la pila.
13    * @return número de elementos en la pila.
14    */
```

```
15    public int size();
16
17   /**
18    * Indica si la pila está vacia.
19    * @return true si la pila está vacía, de otra manera false.
20    */
21    public boolean isEmpty();
22
23   /**
24    * Explorar el elemento en la cima de la pila.
25    * @return el elemento cima en la pila.
26    * @exception EmptyStackException si la pila está vacía.
27    */
28    public E top()
29      throws EmptyStackException;
30
31   /**
32    * Insertar un elemento en la cima de la pila.
33    * @param elemento a ser insertado.
34    */
35    public void push (E elemento);
36
37   /**
38    * Quitar el elemento cima de la pila.
39    * @return elemento removido.
40    * @exception EmptyStackException si la pila está vacía.
41    */
42    public E pop()
43      throws EmptyStackException;
44  }
```

Listado 4.6: Interfaz `Stack` documentada con comentarios en estilo Javadoc. Se usa el tipo parametrizado genérico – `E`– así la pila puede contener elementos de cualquier clase especificada.

4.2.2. Implementación de una pila usando un arreglo

Se puede implementar una pila guardando sus elementos en un arreglo. La pila en esta implementación consiste de una arreglo `S` de `n` elementos además de una variable entera `t` que da el índice del elemento de la cima en el arreglo `S`.

Figura 4.1: Implementación de una pila con un arreglo S. El elemento de la cima está guardado en la celda $S[t]$.

En Java el índice para los arreglos inicia con cero, por lo que `t` se inicializa con -1, y se usa este valor de `t` para identificar una pila vacía. Asimismo, se pueda usar `t` para determinar el número de elementos (`t`+1). Se agrega también una nueva excepción, llamada `FullStackException`, para señalar el error que surge si se intenta insertar un nuevo elemento en una pila llena. La excepción `FullStackException` es particular a esta implementación y no está definida en la ADT pila. Se dan los detalles de la implementación de la pila usando un arreglo en los siguientes algoritmos.

Algoritmo size():
  Regresar $t + 1$

Algoritmo isEmpty():
  Regresar $(t < 0)$

Algoritmo top():
  Si isEmpty() **Entonces**
    lanzar una EmptyStackException
  Regresar $S[t]$

Algoritmo push(e):
  Si size()$= N$ **Entonces**
    lanzar una FullStackException
  $t \leftarrow t + 1$
  $S[t] \leftarrow e$

Algoritmo pop():
  Si isEmpty() **Entonces**
    lanzar una EmptyStackException
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ null
  $t \leftarrow t - 1$
  Regresar e

Análisis de la implementación de pila con arreglos

La correctez de los métodos en la implementación usando arreglos se tiene inmediatamente de la definición de los propios métodos. Hay, sin embargo, un punto medio interesante que involucra la implementación del método `pop`.

Se podría haber evitado reiniciar el viejo $S[t]$ a `null` y se podría tener todavía un método correcto. Sin embargo, hay una ventaja y desventaja en evitar esta asignación que podría pensarse al implementar los algoritmos en Java. El equilibrio involucra el mecanismo *colector de basura* de Java que busca en la memoria objetos que ya no son mas referenciados por objetos activos, y recupera este espacio para su uso futuro. Sea $e = S[t]$ el elemento de la cima antes de que el método `pop` sea llamado. Haciendo a $S[t]$ una referencia nula, se indica que la pila no necesita mas mantener una referencia al objeto e. Además, si no hay otras referencias activas a e, entonces el espacio de memoria tomado por e será recuperado por el colector de basura.

En la siguiente tabla se muestran los tiempos de ejecución para los métodos en la ejecución de una pila con un arreglo. Cada uno de los métodos de la pila en el arreglo ejecuta un número constante de sentencias involucrando operaciones aritméticas, comparaciones, y asignaciones. Además, `pop` también llama a `isEmpty`, el cual también se ejecuta en tiempo constante. Por lo tanto, en esta implementación del ADT pila, cada método se ejecuta en tiempo constante,

esto es, cada uno corre en tiempo $O(1)$.

Método	Tiempo
`size`	$O(1)$
`isEmpty`	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$

Una implementación concreta de Java de los algoritmos para la implementación de la pila se da en el listado 4.7, con la implementación Java de la clase `ArrayStack` implementando la interfaz `Stack`. Se usa un nombre simbólico, `CAPACIDAD`, para indicar el tamaño del arreglo. Este valor permite indicar la capacidad del arreglo en un sólo lugar en el código y que el valor se refleje en todo el código.

```
1  /**
2   * Implementación de la ADT pila usando un arreglo de longitud fija.
3   * Una excepción es lanzada si la operación push es intentada cuando
4   * el tamaño de la pila es igual a la longitud del arreglo.  Esta
5   * clase incluye los métodos principales de la clase agregada
6   * java.util.Stack.
7   *
8   * @see FullStackException
9   */
10 public class ArrayStack<E> implements Stack<E> {
11  /**
12   * Capacidad del arreglo usado para implementar la pila.
13   */
14   protected int capacidad;
15  /**
16   * Capacidad por defecto del arreglo.
17   */
18   public static final int CAPACIDAD = 1000;
19  /**
20   * Arreglo genérico usado para implementar la pila.
21   */
22   protected E S[];
23  /**
24   * Índice del elemento cima de la pila en el arreglo
25   */
26   protected int cima = -1;
27  /**
28   * Inicializa la pila para usar un arreglo de longitud por defecto.
29   */
30   public ArrayStack() {
31     this(CAPACIDAD);
32   }
33  /**
34   * Inicializa la pila para usar un arreglo de longitud dada.
35   * @param cap longitud del arreglo.
36   */
37   public ArrayStack(int cap) {
38     capacidad = cap;
39     S = (E[]) new Object[capacidad]; // el compilador podría dar advertencia
40                                      // pero está bien
41   }
42  /**
43   * Regresa el número de elementos en la pila.
44   * Este método se ejecuta en tiempo O(1)
45   * @return número de elementos en la pila.
46   */
```

```
47    public int size() {
48      return (cima + 1);
49    }
50   /**
51    * Prueba si la pila está vacía.
52    * Este método se ejecuta en tiempo O(1)
53    * @return true si la pila está vacia, false de otra forma.
54    */
55    public boolean isEmpty() {
56      return (cima < 0);
57    }
58   /**
59    * Inserta un elemento en la cima de la pila.
60    * Este método se ejecuta en tiempo O(1)
61    * @return elemento insertado.
62    * @param elemento elemento a ser insertado.
63    * @exception FullStackException si el arreglo de los elementos está lleno.
64    */
65    public void push(E elemento) throws FullStackException {
66      if (size() == capacidad)
67        throw new FullStackException("La pila está llena.");
68      S[++cima] = elemento;
69    }
70   /**
71    * Inspecciona el elemento en la cima de la pila.
72    * Este método se ejecuta en tiempo O(1)
73    * @return elemento cima en la pila.
74    * @exception EmptyStackException si la pila está vacia.
75    */
76    public E top() throws EmptyStackException {
77      if (isEmpty())
78        throw new EmptyStackException("La pila está vacía.");
79      return S[cima];
80      }
81   /**
82    * Quita el elemento cima de la pila.
83    * Este método se ejecuta en tiempo O(1)
84    * @return elemento removido.
85    * @exception EmptyStackException si la pila está vacía.
86    */
87    public E pop() throws EmptyStackException {
88      E elemento;
89      if (isEmpty())
90        throw new EmptyStackException("La pila está vacía.");
91      elemento = S[cima];
92      S[cima--] = null; // desreferenciar S[cima] para el colector de basura.
93      return elemento;
94    }
95   /**
96    * Regresa una representacion de la pila como una lista de elementos,
97    * con el elemento cima al final: [ ... , prev, cima ].
98    * Este metodo corre en tiempo O(n), donde n es el tamaño de la cima.
99    * @return representación textual de la pila.
100   */
101   public String toString() {
102     String s;
103     s = "[";
104     if (size() > 0) s+= S[0];
105     if (size() > 1)
106       for (int i = 1; i <= size()-1; i++) {
107         s += ", " + S[i];
108       }
109     return s + "]";
110   }
111  /**
112   * Imprime información del estado de una operación reciente de la pila.
113   * @param op operación hecha
114   * @param elemento elemento regresado por la operación
```

```
115     * @return información acerca de la operación hecha el elemento
116     * regresado por la operación y el contenido de la pila después de
117     * la operación.
118     */
119     public void estado(String op, Object elemento) {
120       System.out.print("------> " + op);   // imprime esta operación
121       System.out.println(", regresa " + elemento); // que fue regresado
122       System.out.print("resultado: num. elems. = " + size());
123       System.out.print(", ¿está vacio? " + isEmpty());
124       System.out.println(", pila: " + this);       // contenido de la pila
125     }
126    /**
127     * Probar el programa haciendo una serie de operaciones en pilas,
128     * imprimiendo las operaciones realizadas, los elementos regresados y
129     * el contenido de la pila involucrada, despues de cada operación.
130     */
131     public static void main(String[] args) {
132       Object o;
133       ArrayStack<Integer> A = new ArrayStack<Integer>();
134       A.estado("new ArrayStack<Integer> A", null);
135       A.push(7);
136       A.estado("A.push(7)", null);
137       o = A.pop();
138       A.estado("A.pop()", o);
139       A.push(9);
140       A.estado("A.push(9)", null);
141       o = A.pop();
142       A.estado("A.pop()", o);
143       ArrayStack<String> B = new ArrayStack<String>();
144       B.estado("new ArrayStack<String> B", null);
145       B.push("Paco");
146       B.estado("B.push(\"Paco\")", null);
147       B.push("Pepe");
148       B.estado("B.push(\"Pepe\")", null);
149       o = B.pop();
150       B.estado("B.pop()", o);
151       B.push("Juan");
152       B.estado("B.push(\"Juan\")", null);
153     }
154 }
```

Listado 4.7: Implementación de la interfaz `Stack` usando un arreglo con Java.

Salida ejemplo

Se muestra enseguida la salida del programa `ArrayStack`. Con el uso de tipos genéricos, se puede crear un `ArrayStack A` para guardar enteros y otro `ArrayStack B` que guarda `String`.

```
------> new ArrayStack<Integer> A, regresa null
resultado: num. elems. = 0, esta vacio = true, pila: []
------> A.push(7), regresa null
resultado: num. elems. = 1, esta vacio = false, pila: [7]
------> A.pop(), regresa 7
resultado: num. elems. = 0, esta vacio = true, pila: []
------> A.push(9), regresa null
resultado: num. elems. = 1, esta vacio = false, pila: [9]
------> A.pop(), regresa 9
resultado: num. elems. = 0, esta vacio = true, pila: []
```

```
------> new ArrayStack<String> B, regresa null
 resultado: num. elems. = 0, esta vacio = true, pila: []
------> B.push("Paco"), regresa null
 resultado: num. elems. = 1, esta vacio = false, pila: [Paco]
------> B.push("Pepe"), regresa null
 resultado: num. elems. = 2, esta vacio = false, pila: [Paco, Pepe]
------> B.pop(), regresa Pepe
 resultado: num. elems. = 1, esta vacio = false, pila: [Paco]
------> B.push("Juan"), regresa null
 resultado: num. elems. = 2, esta vacio = false, pila: [Paco, Juan]
```

Limitación de la pila con un arreglo

La implementación con un arreglo de una pila es simple y eficiente. Sin embargo, esta implementación tiene un aspecto negativo—esta debe asumir un límite superior fijo, `CAPACIDAD`, en el tamaño de la pila. Para el ejemplo mostrado en el listado 4.7 se escogió el valor de la capacidad igual a 1,000 o un valor arbitrario menor o mayor. Una aplicación podría actualmente necesitar mucho menos espacio que esto, por lo que se tendría un desperdicio de memoria. Alternativamente, una aplicación podría necesitar más espacio que esto, lo cual causaría que la implementación de la pila genere una excepción tan pronto como un programa cliente intente rebasar la capacidad de la pila, por defecto 1,000 objetos. A pesar de la simplicidad y eficiencia, la pila implementada con arreglo no es precisamente ideal.

Afortunadamente, hay otra implementación, la cual se revisa a continuación, que no tiene una limitación del tamaño y usa espacio proporcional al número actual de elementos guardados en la pila. Aún, en casos donde se tiene una buena estimación en el número de elementos que se guardarán en la pila, la implementación basada en el arreglo es difícil de vencer. Las pilas tienen un papel importante en un número de aplicaciones computacionales, por lo tanto es útil tener una implementación rápida del ADT pila como una implementación simple basada en el arreglo.

4.2.3. Implementación de una pila usando lista simple

En esta sección, se explora la implementación del ADT pila usando una lista simple enlazada. En el diseño de tal implementación, se requiere decidir si la cima de la pila está en la cabeza de la lista o en la cola. Sin embargo, hay una mejor opción ya que se pueden insertar elementos y borrar en tiempo constante solamente en la cabeza. Por lo tanto, es más eficiente tener la cima de la pila en la cabeza de la lista. También, para poder hacer la operación `size` en tiempo constante, se lleva la cuenta del número actual de elementos en una variable de instancia.

En vez de usar una lista ligada que solamente pueda guardar objetos de un cierto tipo, como se mostró en la sección 2.2, se quiere, en este caso, implementar una pila genérica usando una lista *genérica* enlazada. Por lo tanto, se necesita

usar un nodo de tipo genérico para implementar esta lista enlazada. Se muestra tal clase `Nodo` en el listado 4.8.

```
1  /**
2   * Nodo de una lista simple enlazada, la cual guarda referencias
3   * a su elemento y al siguiente nodo en la lista.
4   *
5   */
6  public class Nodo<E> {
7    // Variables de instancia:
8    private E elemento;
9    private Nodo<E> sig;
10   /** Crea un nodo con referencias nulas a su elemento y al nodo sig. */
11   public Nodo() {
12     this(null, null);
13   }
14   /** Crear un nodo con el elemento dado y el nodo sig. */
15   public Nodo(E e, Nodo<E> s) {
16     elemento = e;
17     sig = s;
18   }
19   // Métodos accesores:
20   public E getElemento() {
21     return elemento;
22   }
23   public Nodo<E> getSig() {
24     return sig;
25   }
26   // Métodos modificadores:
27   public void setElemento(E nvoElem) {
28     elemento = nvoElem;
29   }
30   public void setSig(Nodo<E> nvoSig) {
31     sig = nvoSig;
32   }
33 }
```

Listado 4.8: La clase `Nodo` implementa un nodo genérico para una lista simple enlazada.

Clase genérica `NodoStack`

Una implementación con Java de una pila, por medio de un lista simple enlazada, se da en el listado 4.9. Todos los métodos de la interfaz `Stack` son ejecutados en tiempo constante. Además de ser eficiente en el tiempo, esta implementación de lista enlazada tiene un requerimiento de espacio que es $O(n)$, donde n es el número actual de elementos en la pila. Por lo tanto, esta implementación no requiere que una nueva excepción sea creada para manejar los problemas de desbordamiento de tamaño. Se usa una variable de instancia `cima` para referirse a la cabeza de la lista, la cual apunta al objeto `null` si la lista está vacía. Cuando se mete un nuevo elemento e en la pila, se crea un nuevo nodo v para e, se referencia e desde v, y se inserta v en la cabeza de la lista. De igual modo, cuando se quita un elemento de la pila, se remueve el nodo en la cabeza de la lista y se regresa el elemento. Por lo tanto, se realizan todas las inserciones y borrados de elementos en la cabeza de la lista.

```
1  /**
2   * Implementación del ADT pila por medio de una lista simple enlazada.
```

```
3   *
4   * @see Nodo
5   */
6
7  public class NodoStack<E> implements Stack<E> {
8    protected Nodo<E> cima; // referencia el nodo cabeza
9    protected int size;     // número de elementos en la pila
10   /** Crear una pila vacía. */
11   public NodoStack() {
12     cima = null;
13     size = 0;
14   }
15   public int size() { return size; }
16   public boolean isEmpty() {
17     if (cima == null) return true;
18     return false;
19   }
20   public void push(E elem) {
21     Nodo<E> v = new Nodo<E>(elem, cima); // crear un nuevo nodo y enlazarlo
22     cima = v;
23     size++;
24   }
25   public E top() throws EmptyStackException {
26     if (isEmpty()) throw new EmptyStackException("La pila está vacía.");
27     return cima.getElemento();
28   }
29   public E pop() throws EmptyStackException {
30     if (isEmpty()) throw new EmptyStackException("La pila está vacía.");
31     E temp = cima.getElemento();
32     cima = cima.getSig(); // desenlazar el antiguo nodo cima
33     size--;
34     return temp;
35   }
36
37
38  /**
39   * Regresar una representación de cadena de la pila como una lista
40   * de elementos con el elemento cima al final: [ ... , prev, cima ].
41   * Este método se ejecuta en tiempo O(n), donde n es el número de elementos
42   * en la pila.
43   * @return representación textual de la pila.
44   */
45   public String toString() {
46     String s;
47     Nodo<E> cur = null;
48     s = "[";
49     int n = size();
50     if (n > 0) {
51       cur = cima;
52       s += cur.getElemento();
53     }
54     if (n > 1)
55       for (int i = 1; i <= n-1; i++) {
56          cur = cur.getSig();
57          s += ", " + cur.getElemento();
58       }
59     s += "]";
60     return s;
61   }
62
63  /**
64   * Imprime información acerca de una operación y la pila.
65   * @param op operación realizada
66   * @param elemento elemento regresado por la operacion
67   * @return información acerca de la operación realizada, el elemento
68   * regresado por la operación y el contenido después de la operación.
69   */
70   public static void estado(Stack S, String op, Object elemento) {
```

```
71      System.out.println("---------------------------------");
72      System.out.println(op);
73      System.out.println("Regresado: " + elemento);
74      String emptyStatus;
75      if (S.isEmpty())
76        emptyStatus = "vacío";
77      else
78        emptyStatus = "no vacío";
79      System.out.println("tam = " + S.size() + ", " + emptyStatus);
80      System.out.println("Pila: " + S);
81    }
82
83   /**
84    * Programa prueba que realiza una serie de operaciones en una pila
85    * e imprime la operación realizada, el elemento regresado
86    * y el contenido de la pila después de cada operación.
87    */
88    public static void main(String[] args) {
89      Object o;
90      Stack<Integer> A = new NodoStack<Integer>();
91      estado (A, "Nueva Pila Vacía", null);
92      A.push(5);
93      estado (A, "push(5)", null);
94      A.push(3);
95      estado (A, "push(3)", null);
96      A.push(7);
97      estado (A, "push(7)", null);
98      o = A.pop();
99      estado (A, "pop()", o);
100     A.push(9);
101     estado (A, "push(9)", null);
102     o = A.pop();
103     estado (A, "pop()", o);
104     o = o = A.top();
105     estado (A, "top()", o);
106   }
107 }
```

Listado 4.9: Clase `NodoStack` implementando la interfaz `Stack` usando una lista simple enlazada con los nodos genéricos del listado 4.8.

4.2.4. Invertir un arreglo con una pila

Se puede usar una pila para invertir los elementos en un arreglo, teniendo así un algoritmo no recursivo para el problema indicado en la sección 2.5. La idea básica es simplemente meter todos los elementos del arreglo en orden en una pila y entonces llenar el arreglo de regreso otra vez sacando los elementos de la pila. En el listado 4.10 se da una implementación en Java del algoritmo descrito. En el método `invertir` se muestra como se pueden usar tipos genéricos en una aplicación simple que usa una pila genérica. En particular, cuando los elementos son sacados de la pila en el ejemplo, estos son automáticamente regresados como elementos del tipo `E`; por lo tanto, estos pueden ser inmediatamente regresados al arreglo de entrada. Se muestran dos ejemplos que usan este método.

```
1  public class InvertirArreglo {
2
3    public static <E> void invertir(E[] a) {
4      Stack<E> S = new ArrayStack<E>(a.length);
5      for(int i=0; i<a.length; i++)
6        S.push(a[i]);
```

```
7       for(int i=0; i<a.length; i++)
8         a[i] = S.pop();
9     }
10
11    /** Rutina probadora para invertir arreglos */
12    public static void main(String args[]) {
13      // autoboxing: a partir de Java 1.5, hace la conversión de tipos
14      //             primitivos a objetos automáticamente y viceversa.
15      //             Se usa a continuación.
16      Integer[] a = {4, 8, 15, 16, 23, 42};
17      String[] s = {"Jorge","Paco","Pedro","Juan","Marta"};
18      System.out.println("a = "+Arrays.toString(a));
19      System.out.println("s = "+Arrays.toString(s));
20      System.out.println("Invirtiendo ...");
21      invertir(a);
22      invertir(s);
23      System.out.println("a = "+Arrays.toString(a));
24      System.out.println("s = "+Arrays.toString(s));
25    }
26  }
```

Listado 4.10: Un método genérico que invierte los elementos en un arreglo genérico mediante una pila de la interfaz `Stack<E>`

La salida del listado 4.10 se muestra a continuación:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jorge, Paco, Pedro, Juan, Marta]
Invirtiendo ...
a = [42, 23, 16, 15, 8, 4]
s = [Marta, Juan, Pedro, Paco, Jorge]
```

4.2.5. Aparear paréntesis y etiquetas HTML

En esta subsección, se exploran dos aplicaciones relacionadas de pilas, la primera es para el apareamiento de paréntesis y símbolos de agrupamiento en expresiones aritméticas.

Las expresiones aritméticas pueden contener varios pares de símbolos de agrupamiento, tales como:

- Paréntesis: "(" y ")"
- Llaves: "{" y "}"
- Corchetes: "[" y "]"
- Símbolos de la función piso: "⌊" y "⌋"
- Símbolos de la función techo: "⌈" y "⌉"

y cada símbolo de apertura deberá aparearse con su correspondiente símbolo de cierre. Por ejemplo, un corchete izquierdo, "[", deberá aparearse con su correspondiente corchete derecho, "]", como en la siguiente expresión:

$$[(5 + x) - (y + z)].$$

Los siguientes ejemplos ilustran este concepto:

- Correcto: ()(())\{([()])\}
- Correcto: (((()(())\{([()])\})))
- Incorrecto:)(())\{([()])\}
- Incorrecto: (\{[ ])\}
- Incorrecto: (

Un algoritmo para el apareamiento de paréntesis

Un problema importante en el procesamiento de expresiones aritméticas es asegurar que los símbolos de agrupamiento se apareen correctamente. Se puede usar una pila S para hacer el apareamiento de los símbolos de agrupamiento en una expresión aritmética con una sola revisión de izquierda a derecha. El algoritmo prueba que los símbolos izquierdo y derecho se apareen y también que ambos símbolos sean del mismo tipo.

Suponiendo que se da una secuencia $X = x_0x_1x_2 \ldots x_{n-1}$, donde cada x_i es un símbolo que puede ser un símbolo de agrupamiento, un nombre de variable, un operador aritmético, o un número. La idea básica detrás de la revisión de que los símbolos de agrupamiento en S empaten correctamente, es procesar los símbolos en X en orden. Cada vez que se encuentre un símbolo de apertura, se mete ese símbolo en S, y cada vez que se encuentre un símbolo de cierre, se saca el símbolo de la cima de la pila S, suponiendo que S no está vacía, y se revisa que esos símbolos son del mismo tipo. Suponiendo que las operaciones `push` y `pop` son implementadas para ejecutarse en tiempo constante, este algoritmo corre en tiempo $O(n)$, esto es lineal. Se da un pseudocódigo de este algoritmo a continuación.

Algoritmo ApareamientoParéntesis(X, n):
  Entrada: Un arreglo X de n símbolos, cada uno de los cuales
    es un símbolo de agrupamiento, una variable, un operador
    aritmético, o un número.
  Salida: `true` si y sólo si todos los símbolos de agrupamiento
    en X se aparean.
  Sea S una pila vacía
  Para $i \leftarrow 0$ **Hasta** $n - 1$ **Hacer**
    Si $X[i]$ es un símbolo de apertura **Entonces**
      S.`push`($X[i]$)
    Si no Si $X[i]$ es símbolo de cierre **Entonces**
      Si S.`isEmpty()` **Entonces**
        regresar `false` { nada para aparear }
      Si S.`pop()` no empata el tipo de $X[i]$ **Entonces**
        regresar `false` { tipo incorrecto }
  Si S.`isEmpty()` **Entonces**
    regresar `true` { cada símbolo apareado }
  Si no
    regresar `false` { algunos símbolos no empataron }

Apareamiento de etiquetas en un documento HTML

Otra aplicación en la cual el apareamiento es importante es en la validación de documentos HTML. HTML es el formato estándar para documentos hiperenlazados en el Internet. En un documento HTML, porciones de texto están delimitados por *etiquetas HTML*. Una etiqueta simple de apertura HTML tiene la forma "`<nombre>`" y la correspondiente etiqueta de cierre tiene la forma "`</nombre>`". Las etiquetas HTML comúnmente usadas incluyen

- `body`: cuerpo del documento
- `h1`: sección cabecera
- `blockquote`: sección sangrada
- `p`: párrafo
- `ol`: lista ordenada
- `li`: elemento de la lista

Idealmente, un documento HTML debería tener todas sus etiquetas apareadas, sin embargo varios navegadores toleran un cierto número de etiquetas no apareadas.

El algoritmo de la sección para el apareamiento de paréntesis puede ser usado para aparear las etiquetas en un documento HTML. En el listado 4.11 se da un programa en Java para aparear etiquetas en un documento HTML leído de la entrada estándar. Por simplicidad, se asume que todas las etiquetas son etiquetas simples de apertura y cierre, es decir sin atributos y que no hay etiquetas formadas incorrectamente. El método `esHTMLApareada` usa una pila para guardar los nombres de las etiquetas

```
1  import java.io.IOException;
2  import java.util.Scanner;
3
4  /** Prueba simplificada de apareamiento de etiquetas en un documento HTML. */
5  public class HTML {
6    /** Quitar el primer carácter y el últimto de una <etiqueta> cadena. */
7    public static String quitarExtremos(String t) {
8      if (t.length() <= 2 ) return null; // esta es una etiqueta degenerada
9      return t.substring(1,t.length()-1);
10   }
11   /** Probar si una etiqueta desnuda (sin los simbolos inicial y final)
12     * esta vacía o es una etiqueta verdadera de apertura. */
13   public static boolean esEtiquetaApertura(String etiqueta) {
14     return etiqueta.length()==0 || etiqueta.charAt(0)!='/';
15   }
16   /** Probar si etiqueta1 desnuda aparea con etiqueta2 de cierre. */
17   public static boolean seAparean(String etiqueta1, String etiqueta2) {
18     // probar contra el nombre después de '/'
19     return etiqueta1.equals(etiqueta2.substring(1));
20   }
21   /** Probar si cada etiqueta de apertura tiene una etiqueta de cierre. */
22   public static boolean esHTMLApareada(String[] etiqueta) {
23     Stack<String> S = new NodoStack<String>(); // Pila para aparear etiquetas
24     for (int i=0; (i<etiqueta.length)&&(etiqueta[i]!=null); i++)
25       if (esEtiquetaApertura(etiqueta[i]))
```

```
26          S.push(etiqueta[i]); // etiqueta apertura; meterla en la pila
27        else {
28          if (S.isEmpty())
29            return false;    // nada para aparear
30          if (!seAparean(S.pop(), etiqueta[i]))
31            return false;    // apareamiento incorrecto
32        }
33      if (S.isEmpty()) return true; // se apareo todo
34      return false;  // hay algunas etiquetas que nunca fueron apareadas
35    }
36    public final static int CAPACIDAD = 1000;  // Capacidad del arreglo etiqueta
37    /* Dividir un documento HTML en un arreglo de etiquetas html */
38    public static String[] parseHTML(Scanner s) {
39      String[] etiqueta = new String[CAPACIDAD]; // el arreglo etiqueta
40      int contador = 0;   // contador de etiquetas
41      String elemento;    // elemento regresado por el scanner s
42      while (s.hasNextLine()) {
43        while ((elemento=s.findInLine("<[^>]*>"))!=null) // encontrar sig etiq
44          etiqueta[contador++]=quitarExtremos(elemento);
45        s.nextLine();     // ir a la siguiente línea
46      }
47      return etiqueta;    // arreglo de etiquetas desnudas
48    }
49    public static void main(String[] args) throws IOException { // probador
50      System.out.print("El archivo de entrada es un documento ");
51      if (esHTMLApareada(parseHTML(new Scanner(System.in))))
52        System.out.println("HTML apareado");
53      else
54        System.out.println("HTML NO apareado");
55    }
56  }
```

Listado 4.11: Clase `HTML` para revisar el apareamiento de las etiquetas en un documento HTML.

4.3. Colas

Otra estructura de datos fundamental es la *cola* (*queue*). Una cola es una colección de objetos que son insertados y removidos de acuerdo al principio *primero en entrar, primero en salir*, *FIFO* (*first-in first-out*). Esto es, los elementos pueden ser insertados en cualquier momento, pero solamente el elemento que ha estado en la cola más tiempo puede ser removido en cualquier momento.

Se dice que los elementos entran a una cola por la parte de atrás y son removidos del frente. La metáfora para esta terminología es una fila de gente esperando para subir a un juego mecánico. La gente que espera para tal juego entra por la parte trasera de la fila y se sube al juego desde el frente de la línea.

4.3.1. Tipo de dato abstracto cola

El tipo de dato abstracto cola define una colección que guarda los objetos en una secuencia, donde el acceso al elemento y borrado están restringidos al primer elemento en la secuencia, el cual es llamado el *frente* de la cola, y la inserción del elemento está restringida al final de la secuencia, la cual es llamada la *parte posterior* de la cola. Esta restricción forza la regla de que los elementos son insertados y borrados en una cola de acuerdo al principio FIFO.

El ADT *cola* soporta los siguientes dos métodos fundamentales:

`enqueue(e)`:inserta el elemento e en la parte posterior de la cola.
`dequeue()`:quita el elemento del frente de la cola y lo regresa; un error ocurre si la cola está vacía.

Adicionalmente, de igual modo como con el ADT pila, el ADT cola incluye los siguientes métodos de apoyo:

`size()`:regresa el número de elementos en la cola.
`isEmpty()`:regresa un booleano indicando si la cola está vacía.
`front()`:regresa el elemento del frente de la cola, sin removerlo; un error ocurre si la cola está vacía.

La siguiente tabla muestra una serie de operaciones de cola y su efecto en una cola Q inicialmente vacía de objetos enteros. Por simplicidad, se usan enteros en vez de objetos enteros como argumentos de las operaciones.

Operación	Salida	$frente \leftarrow Q \leftarrow zaga$
`enqueue(5)`	–	(5)
`enqueue(3)`	–	(5,3)
`dequeue()`	5	(3)
`enqueue(7)`	–	(3,7)
`dequeue()`	3	(7)
`front()`	7	(7)
`dequeue()`	7	()
`dequeue()`	"`error`"	()
`isEmpty()`	`true`	()
`enqueue(9)`	–	(9)
`enqueue(7)`	–	(9,7)
`size()`	2	(9,7)
`enqueue(3)`	–	(9,7,3)
`enqueue(5)`	–	(9,7,3,5)
`dequeue()`	9	(7,3,5)

Aplicaciones de ejemplo

Hay varias aplicaciones posibles para las colas. Tiendas, teatros, centros de reservación, y otros servicios similares que típicamente procesen peticiones de clientes de acuerdo al principio FIFO. Una cola sería por lo tanto una opción lógica para una estructura de datos que maneje el procesamiento de transacciones para tales aplicaciones.

4.3.2. Interfaz cola

Una interfaz de Java para el ADT cola se proporciona en el listado 4.12. Esta interfaz genérica indica que objetos de tipo arbitrario pueden ser insertados en la

cola. Por lo tanto, no se tiene que usar conversión explícita cuando se remuevan elementos.

Los métodos `size` e `isEmpty` tienen el mismo significado como sus contrapartes en el ADT pila. Estos dos métodos, al igual que el método `front`, son conocidos como métodos *accesores*, por su valor regresado y no cambian el contenido de la estructura de datos.

```
1  /**
2    * Interfaz para una cola: una colección de elementos que son insertados
3    * y removidos de acuerdo con el principio primero en entrar, primero en
4    * salir.
5    *
6    * @see EmptyQueueException
7    */
8
9  public interface Queue<E> {
10  /**
11   * Regresa el numero de elementos en la cola.
12   * @return número de elementos en la cola.
13   */
14   public int size();
15  /**
16   * Indica si la cola está vacía
17   * @return true si la cola está vacía, false de otro modo.
18   */
19   public boolean isEmpty();
20  /**
21   * Revisa el elemento en el frente de la cola.
22   * @return elemento en el frente de la cola.
23   * @exception EmptyQueueException si la cola está vacía.
24   */
25   public E front() throws EmptyQueueException;
26  /**
27   * Insertar un elemento en la zaga de la cola.
28   * @param elemento nuevo elemento a ser insertado.
29   */
30   public void enqueue (E element);
31  /**
32   * Quitar el elemento del frente de la cola.
33   * @return elemento quitado.
34   * @exception EmptyQueueException si la cola está vacía.
35   */
36   public E dequeue() throws EmptyQueueException;
37 }
```

Listado 4.12: Interfaz `Queue` documentada con comentarios en estilo Javadoc.

4.3.3. Implementación simple de la cola con un arreglo

Se presenta una realización simple de una cola mediante un arreglo, Q, de capacidad fija, guardando sus elementos. Como la regla principal con el tipo ADT cola es que se inserten y borren los objetos de acuerdo al principio FIFO, se debe decidir como se va a llevar el frente y la parte posterior de la cola.

Una posibilidad es adaptar la aproximación empleada para la implementación de la pila, dejando que $Q[0]$ sea el frente de la cola y permitiendo que la cola crezca desde allí. Sin embargo, esta no es una solución eficiente, para esta se requiere que se muevan todos los elementos hacia adelante una celda cada vez que se haga la operación `dequeue`. Tal implementación podría tomar por lo tanto

tiempo $O(n)$ para realizar el método `dequeue`, donde n es el número actual de objetos en la cola. Si se quiere lograr tiempo constante para cada método de la cola, se necesita una aproximación diferente.

Usar un arreglo de forma circular

Para evitar mover objetos una vez que son colocados en Q, se definen dos variables f y r, que tienen los siguientes usos:

- f es un índice a la celda de Q guardando el primer elemento de la cola, el cual es el siguiente candidato a ser eliminado por una operación `dequeue`, a menos que la cola esté vacía, en tal caso $f = r$.
- r es un índice a la siguiente celda del arreglo disponible en Q.

Inicialmente, se asigna $f = r = 0$, lo cual indica que la cola está vacía. Cuando se quita un elemento del frente de la cola, se incrementa f para indicar la siguiente celda. Igualmente, cuando se agrega un elemento, se guarda este en la celda $Q[r]$ y se incrementa r para indizar la siguiente celda disponible en Q. Este esquema permite implementar los métodos `front`, `enqueue`, y `dequeue` en tiempo constante, esto es, tiempo $O(1)$. Sin embargo, todavía hay un problema con esta aproximación.

Considerar, por ejemplo, que sucede si repetidamente se agrega y se quita un solo elemento N veces. Se tendría $f = r = N$. Si se intentará entonces insertar el elemento una vez mas, se podría obtener un error de arreglo fuera de límites, ya que las N localidades válidas en Q son de $Q[0]$ a $Q[N-1]$, aunque hay espacio en la cola para este caso. Para evitar este problema y poder utilizar todo el arreglo Q, se permite que los índices reinicien al final de Q. Esto es, se ve ahora a Q como un "arreglo circular" que va de $Q[0]$ a $Q[N-1]$ y entonces inmediatamente regresan a $Q[0]$ otra vez, ver figura 4.2.

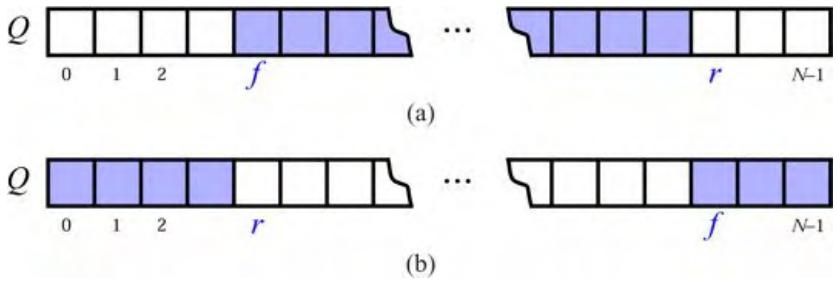

Figura 4.2: Usando arreglo Q en un modo circular: (a) la configuración "normal" con $f \leq r$; (b) la configuración "envuelto alrededor de" con $r < f$

Implementar esta vista circular de Q es fácil. Cada vez que se incremente f o r, se calcula este incremento como "$(f + 1)$ mód N" o "$(r + 1)$ mód N", respectivamente.

El operador "mód" es el operador *módulo*, el cual es calculado tomando el residuo después de la división entera. Por ejemplo, 14 dividido por 4 es 3 con residuo 2, así $14 \mod 4 = 2$. Específicamente, dados enteros x y y tal que $x \geq 0$ y $y \geq 0$, se tiene $x \mod y = x - \lfloor x/y \rfloor y$. Esto es, si $r = x \mod y$, entonces hay un entero no negativo q, tal que $x = qy + r$. Java usa "`%`" para denotar el operador módulo. Mediante el uso del operador módulo, se puede ver a Q como un arreglo circular e implementar cada método de la cola en una cantidad constante de tiempo, es decir, tiempo $O(1)$. Se describe como usar esta aproximación para implementar una cola enseguida.

Algoritmo size():
  return $(N - f + r) \mod N$

Algoritmo isEmpty():
  return $(f = r)$

Algoritmo front():
  Si isEmpty() **Entonces**
    lanzar una EmptyQueueException
  return $Q[f]$

Algoritmo enqueue(e):
  Si size()$= N - 1$ **Entonces**
    lanzar una FullQueueException
  $Q[r] \leftarrow e$
  $r \leftarrow (r + 1) \mod N$

Algoritmo dequeue():
  Si isEmpty() **Entonces**
    lanzar una EmptyQueueException
  $e \leftarrow Q[f]$
  $Q[f] \leftarrow$ null
  $f \leftarrow (f + 1) \mod N$
  return e

La implementación anterior contiene un detalle importante, la cual podría ser ignorada al principio. Considerar la situación que ocurre si se agregan N objetos en Q sin quitar ninguno de ellos. Se tendría $f = r$, la cual es la misma condición que ocurre cuando la cola está vacía. Por lo tanto, no se podría decir la diferencia entre una cola llena y una vacía en este caso. Existen varias formas de manejarlo.

La solución que se describe es insistir que Q no puede tener maś de $N - 1$ objetos. Esta regla simple para manejar una cola llena se considera en el algoritmo `enqueue` dado previamente. Para señalar que no se pueden insertar más elementos en la cola, se emplea una excepción específica, llamada `FullQueueException`. Para determinar el tamaño de la cola se hace con la expresión $(N - f + r) \mod N$, la cual da el resultado correcto en una configuración "normal", cuando $f \leq r$, y en la configuración "envuelta" cuando $r < f$. La implementación Java

de una cola por medio de un arreglo es similar a la de la pila, y se deja como ejercicio.

Al igual que con la implementación de la pila con arreglo, la única real desventaja de la implementación de la cola con arreglo es que artificialmente se pone la capacidad de la cola a algún valor fijo. En una aplicación real, se podría actualmente necesitar más o menos capacidad que esta, pero si se tiene una buena estimación de la capacidad, entonces la implementación con arreglo es bastante eficiente.

La siguiente tabla muestra los tiempos de ejecución de los métodos de una cola empleando un arreglo. Cada uno de los métodos en la realización arreglo ejecuta un número constante de sentencias involucrando operaciones aritméticas, comparaciones, y asignaciones. Por lo tanto, cada método en esta implementación se ejecuta en tiempo $O(1)$.

Método	Tiempo
`size`	$O(1)$
`isEmpty`	$O(1)$
`front`	$O(1)$
`enqueue`	$O(1)$
`dequeue`	$O(1)$

4.3.4. Implementar una cola con una lista enlazada

Se puede implementar eficientemente el ADT cola usando una lista genérica simple ligada. Por razones de eficiencia, se escoge el frente de la cola para que esté en la cabeza de la lista, y la parte posterior de la cola para que esté en la cola de la lista. De esta forma se quita de la cabeza y se inserta en el final. Para esto se necesitan mantener referencias en los dos nodos cabeza y cola de la lista. Se presenta a continuación una implementación para el método `queue` de la cola en el listado 4.13 y para `dequeue` en el listado 4.14.

```
1  public void enqueue(E elem) {
2    Nodo<E> nodo = new Nodo<E>();
3    nodo.setElemento(elem);
4    nodo.setSig(null); // nodo será el nuevo nodo cola
5    if (tam == 0)
6      cabeza = nodo; // caso especial de una cola previamente vacía
7    else
8      cola.setSig(nodo); // agregar nodo en la cola de la lista
9    cola = nodo; // actualizar la referencia al nodo cola
10   tam++;
11 }
```

Listado 4.13: Implementación del método `enqueue` del ADT cola por medio de una lista simple ligada usando nodos de la clase `Nodo`.

```
1  public E dequeue() throws EmptyQueueException {
2    if (tam == 0)
3      throw new EmptyQueueException("La cola está vacía.");
4    E e = cabeza.getElemento();
5    cabeza = cabeza.getSig();
6    tam--;
```

```
7       if (tam == 0)
8         cola = null; // la cola está ahora vacía
9       return e;
10    }
```

Listado 4.14: Implementación del método `dequeue` del ADT cola por medio de una lista simple ligada usando nodos de la clase `Nodo`.

Cada uno de los métodos de la implementación del ADT cola con una implementación de lista ligada simple corre en tiempo $O(1)$. Se evita la necesidad de indicar un tamaño máximo para la cola, como se hizo en la implementación de la cola basada en un arreglo, pero este beneficio viene a expensas de incrementar la cantidad de espacio usado por elemento. A pesar de todo, los métodos en la implementación de la cola con la lista ligada son más complicados que lo que se desearía, se debe tener cuidado especial de como manejar los casos especiales donde la cola está vacía antes de un `enqueue` o cuando la cola se vacía después de un `dequeue`.

4.3.5. Planificador Round Robin

Un uso popular de la estructura de datos cola es para implementar un planificador *round robin*, donde se itera a través de una colección de elementos en forma circular y se "atiende" cada elemento mediante la realización de una acción dada sobre este. Dicha pauta es usada, por ejemplo, para asignar equitativamente un recurso que deberá ser compartido por una colección de clientes. Por ejemplo, se puede usar un planificador round robin para asignar una parte del tiempo del CPU a varias aplicaciones ejecutándose concurrentemente en una computadora.

Se puede implementar un planificador round robin usando una cola, Q, mediante la repetición de los siguientes pasos:

1. $e \leftarrow Q$.`dequeue()`
2. Servir elemento e
3. Q.`enqueue(e)`

El problema de Josefo

En el juego de niños de la "Papa Caliente", un grupo de n niños sentados en círculo se pasan un objeto, llamado la "papa" alrededor del círculo. El juego se inicia con el niño que tenga la papa pasándola al siguiente en el círculo, y estos continúan pasando la papa hasta que un coordinador suena una campana, en donde el niño que tenga la papa deberá dejar el juego y pasar la papa al siguiente niño en el círculo. Después de que el niño seleccionado deja el juego, los otros niños estrechan el círculo. Este proceso es entonces continuado hasta que quede solamente un niño, que es declarado el ganador. Si el coordinador siempre usa la estrategia de sonar la campana después de que la papa ha pasado k veces, para algún valor fijo k, entonces determinar el ganador para una lista de niños es conocido como el *problema de Josefo*.

Resolviendo el problema de Josefo con una cola

Se puede resolver el problema de Josefo para una colección de n elementos usando una cola, mediante la asociación de la papa con el elemento en el frente de la cola y guardando elementos en la cola de acuerdo a su orden alrededor del círculo. Por lo tanto, pasar la papa es equivalente a retirar un elemento de la cola e inmediatamente agregarlo otra vez. Después de que este proceso se ha realizado k veces, se quita el elemento del frente usando `dequeue` y descartándolo. Se muestra en el listado 4.15, un programa Java completo para resolver el problema de Josefo, el cual describe una solución que se ejecuta en tiempo $O(nk)$.

```
1  public class Josefo {
2    /** Solución del problema de Josefo usando una cola. */
3    public static <E> E Josefo(Queue<E> Q, int k) {
4      if (Q.isEmpty()) return null;
5      while (Q.size() > 1 ) {
6        System.out.println("  Cola: " + Q + " k = " + k);
7        for (int i=0; i<k; i++)
8          Q.enqueue(Q.dequeue()); // mover el elemento del frente al final
9        E e = Q.dequeue(); // quitar el elemento del frente de la coleccción
10       System.out.println("\t"+e+" está eliminad@");
11     }
12     return Q.dequeue();   // el ganador
13   }
14   /** Construir una cola desde un arreglo de objetos */
15   public static <E> Queue<E> construirCola(E a[]) {
16     Queue<E> Q = new ColaNodo<E>();
17     for(E e: a)
18       Q.enqueue(e);
19     return Q;
20   }
21   /** Método probador */
22   public static void main(String[] args) {
23     String[] a1 = {"Jorge","Pedro","Carlos"};
24     String[] a2 = {"Gabi","Toño","Paco","Mari","Iván","Alex","Néstor"};
25     String[] a3 = {"Rigo","Edgar","Karina","Edgar","Temo","Héctor"};
26     System.out.println("Primer ganador es "+Josefo(construirCola(a1),7));
27     System.out.println("Segundo ganador es "+Josefo(construirCola(a2),10));
28     System.out.println("Tercer ganador es "+Josefo(construirCola(a3),2));
29   }
30 }
```

Listado 4.15: Un programa Java completo para resolver el problema de Josefo usando una cola. La clase `ColaNodo` es mostrada en los listados 4.13 y 4.14.

4.4. Colas con doble terminación

Considerar una estructura de datos parecida a una cola que soporte inserción y borrado tanto en el frente y la parte posterior de la cola. Tal extensión de una cola es llamada una *cola con doble terminación* o en inglés *double-ended queue*, o *deque*, está última se pronuncia '`dek`', para evitar confusión con el método `dequeue` de un ADT cola regular, que se pronuncia como '`dikiu`'

4.4.1. El tipo de dato abstracto deque

El tipo de dato abstracto es más completo que los ADT pila y cola. Los métodos fundamentales del ADT deque son los siguientes:

`addFirst`(e):inserta un nuevo elemento e al inicio de la deque.
`addLast`(e):inserta un nuevo elemento e al final de la deque.
`removeFirst()`:retira y regresa el primer elemento de la deque; un error ocurre si la deque está vacía.
`removeLast()`:retira y regresa el último elemento de la deque; un error ocurre si la deque está vacía.

Adicionalmente, el ADT deque podría también incluir los siguientes métodos de apoyo:

`getFirst()`:regresa el primer elemento de la deque; un error ocurre si la deque está vacía.
`getLast()`:regresa el último elemento de la deque; un error ocurre si la deque está vacía.
`size()`:regresa el número de elementos de la deque.
`isEmpty()`:determina si la deque está vacía.

La siguiente tabla muestra una serie de operaciones y sus efectos sobre una deque D inicialmente vacía de objetos enteros. Por simplicidad, se usan enteros en vez de objetos enteros como argumentos de las operaciones.

Operación	Salida	D
`addFirst(3)`	–	(3)
`addFirst(5)`	–	(5,3)
`removeFirst()`	5	(3)
`addLast(7)`	–	(3,7)
`removeFirst()`	3	(7)
`removeLast()`	7	()
`removeFirst()`	"`error`"	()
`isEmpty()`	`true`	()

4.4.2. Implementación de una deque

Como la deque requiere inserción y borrado en ambos extremos de una lista, usando una lista simple ligada para implementar una deque sería ineficiente. Sin embargo, se puede usar una lista doblemente enlazada, para implementar una deque eficientemente. La inserción y remoción de elementos en cualquier extremo de una lista doblemente enlazada es directa para hacerse en tiempo $O(1)$, si se usan nodos centinelas para la cabeza y la cola.

Para una inserción de un nuevo elemento e, se puede tener acceso al nodo p que estará antes de e y al nodo q que estará después del nodo e. Para insertar un nuevo elemento entre los nodos p y q, cualquiera o ambos de estos podrían

ser centinelas, se crea un nuevo nodo t, haciendo que los enlaces `prev` y `sig` de t hagan referencia a p y q respectivamente.

De igual forma, para quitar un elemento guardado en un nodo t, se puede acceder a los nodos p y q que están a los lados de t, y estos nodos deberán existir, ya que se usan centinelas. Para quitar el nodo t entre los nodos p y q, se tiene que hacer que p y q se apunten entre ellos en vez de t. No se requiere cambiar ninguno de los campos en t, por ahora t puede ser reclamado por el colector de basura, ya que no hay nadie que esté apuntando a t.

Todos los métodos del ADT deque, como se describen previamente, están incluidos en la clase `java.util.LinkedList<E>`. Por lo tanto, si se necesita usar una deque y no se desea implementarla desde cero, se puede simplemente usar la clase anterior.

De cualquier forma, se muestra la interfaz `Deque` en el listado 4.16 y una implementación de esta interfaz en el listado 4.17.

```
1  /**
2   * Interfaz para una deque: una colección de objetos que pueden ser
3   * insertados y quitados en ambos extremos; un subconjunto de los
4   * métodos de java.util.LinkedList.
5   *
6   */
7
8  public interface Deque<E> {
9   /**
10   * Regresa el número de elementos en la deque.
11   */
12   public int size();
13  /**
14   * Determina si la deque está vacía.
15   */
16   public boolean isEmpty();
17  /**
18   * Regresa el primer elemento; una excepción es lanzada si la deque
19   * está vacía.
20   */
21   public E getFirst() throws EmptyDequeException;
22  /**
23   * Regresa el último elemento; una excepción es lanzada si la deque
24   * está vacía.
25   */
26   public E getLast() throws EmptyDequeException;
27  /**
28   * Insertar un elemento para que sea el primero en la deque.
29   */
30   public void addFirst (E element);
31  /**
32   * Insertar un elemento para que sea el último en la deque.
33   */
34   public void addLast (E element);
35  /**
36   * Quita el primer elemento; una excepción es lanzada si la deque
37   * está vacía.
38   */
39   public E removeFirst() throws EmptyDequeException;
40  /**
41   * Quita el último elemento; una excepción es lanzada si la deque
42   * está vacía.
43   */
44   public E removeLast() throws EmptyDequeException;
45 }
```

Listado 4.16: Interfaz `Deque` documentada con comentarios en estilo Javadoc. Se usa el tipo parametrizado genérico E con lo cual una deque puede contener elementos de cualquier clase especificada.

```
1  /**
2   * Implementación de la interfaz Deque mediante una lista doblemente
3   * enlazada. Esta clase usa la clase NodoDL, la cual implementa un nodo de
4   * la lista.
5   *
6   */
7
8  public class DequeNodo<E> implements Deque<E> {
9    protected NodoDL<E> cabeza, cola;  // centinelas
10   protected int tam;    // número de elementos
11
12   /** Crea una deque vacía. */
13   public DequeNodo() {  // inicializar una deque vacía
14     cabeza = new NodoDL<E>();
15     cola = new NodoDL<E>();
16     cabeza.setSig(cola);  // hacer que cabeza apunte a cola
17     cola.setPrev(cabeza);  // hacer que cola apunte a cabeza
18     tam = 0;
19   }
20
21 /**
22  * Regresar el tamaño de la deque, esto es el número de elementos que tiene.
23  * @return  Número de elementos en la deque
24  */
25   public int size() {
26     return tam;
27   }
28
29 /**
30  * Esta función regresa true si y sólo si la deque está vacía.
31  * @return  true si la deque está vacía, false de otro modo.
32  */
33   public boolean isEmpty() {
34     if (tam == 0)
35       return true;
36     return false;
37   }
38
39 /**
40  * Revisar el primer elemento sin modificar la deque.
41  * @return El primer elemento en la secuencia.
42  */
43   public E getFirst() throws EmptyDequeException {
44     if (isEmpty())
45       throw new EmptyDequeException("Deque está vacía.");
46     return cabeza.getSig().getElemento();
47   }
48
49   public E getLast() throws EmptyDequeException {
50     if (isEmpty())
51       throw new EmptyDequeException("Deque está vacía.");
52     return cola.getPrev().getElemento();
53   }
54
55   public void addFirst(E o) {
56     NodoDL<E> segundo = cabeza.getSig();
57     NodoDL<E> primero = new NodoDL<E>(o, cabeza, segundo);
58     segundo.setPrev(primero);
59     cabeza.setSig(primero);
60     tam++;
```

```
61    }
62
63    public void addLast(E o) {
64      NodoDL<E> penultimo = cola.getPrev();
65      NodoDL<E> ultimo = new NodoDL<E>(o, penultimo, cola);
66      penultimo.setSig(ultimo);
67      cola.setPrev(ultimo);
68      tam++;
69    }
70
71    public E removeFirst() throws EmptyDequeException {
72      if (isEmpty())
73        throw new EmptyDequeException("Deque está vacía.");
74      NodoDL<E> primero = cabeza.getSig();
75      E e = primero.getElemento();
76      NodoDL<E> segundo = primero.getSig();
77      cabeza.setSig(segundo);
78      segundo.setPrev(cabeza);
79      tam--;
80      return e;
81     }
82
83    public E removeLast() throws EmptyDequeException {
84      if (isEmpty())
85        throw new EmptyDequeException("Deque está vacía.");
86      NodoDL<E> ultimo = cola.getPrev();
87      E e = ultimo.getElemento();
88      NodoDL<E> penultimo = ultimo.getPrev();
89      cola.setPrev(penultimo);
90      penultimo.setSig(cola);
91      tam--;
92      return e;
93    }
94
95  }
```

Listado 4.17: Clase `DequeNodo` implementando la interfaz `Deque` con la clase `NodoDL` (que no se muestra). `NodoDL` es un nodo de una lista doblemente enlazada genérica.

Capítulo 5

Listas e Iteradores

5.1. Lista arreglo

Suponer que se tiene una colección S de n elementos guardados en un cierto orden lineal, de este modo se puede referir a los elementos en S como primero, segundo, tercero, etc. Tal colección es generalmente referida como una *lista* o *secuencia*. Se puede referir únicamente a cada elemento e en S usando un entero en el rango $[0, n-1]$ esto es igual al número de elementos de S que preceden a e en S. El *índice* de un elemento e en S es el número de elementos que están antes que e en S. Por lo tanto, el primer elemento en S tiene índice cero y el último índice tiene índice $n-1$. También, si un elemento de S tiene índice i, su elemento previo, si este existe, tiene índice $i-1$, y su siguiente elemento, si este existe, tiene índice $i+1$. Este concepto de índice está relacionado al de *rango* de un elemento en una lista, el cual es usualmente definido para ser uno más que su índice; por lo que el primer elemento está en el rango uno, el segundo está en rango dos, etc.

Una secuencia que soporte acceso a sus elementos mediante sus índices es llamado una *lista arreglo* o también con el término más viejo *vector*. Como la definición de índice es más consistente con la forma como los arreglos son índizados en Java y otros lenguajes de programación, como C y C++, se hará referencia al lugar donde un elemento es guardado en una lista arreglo como su "índice", no su "rango".

Este concepto de índice es simple pero potente, ya que puede ser usado para indicar donde insertar un nuevo elemento en una lista o de donde quitar un elemento viejo.

5.1.1. El tipo de dato abstracto lista arreglo

Como un ADT, una lista arreglo S tiene los siguientes métodos, además de los métodos estándares `size()` y `isEmpty()`:

`get(`i`)`: regresa el elemento de S con índice i; una condición de error ocurre si $i < 0$ o $i >$ `size()`-1.

`set(`i, e`)`: reemplaza con e y regresar el elemento en el índice i; una condición de error ocurre si $i < 0$ o $i >$ size() $- 1$.

`add(`i, e`)`: inserta un nuevo elemento e en S para tener índice i; una condición de error ocurre si $i < 0$ o $i >$ size().

`remove(`i`)`: quita de S el elemento en el índice i; una condición de error ocurre si $i < 0$ o $i >$ size() $- 1$.

No se insiste que un arreglo debería ser usado para implementar una lista arreglo, así el elemento en el índice 0 está guardado en el índice 0 en el arreglo, aunque esta es una posibilidad muy natural. La definición de índice ofrece una forma para referirse al "lugar" donde un elemento está guardado en una secuencia sin tener que preocuparse acerca de la implementación exacta de esa secuencia. El índice de un elemento podría cambiar cuando la secuencia sea actualizada, como se muestra en el siguiente ejemplo.

Operación	Salida	S
`add(0,7)`	–	(7)
`add(0,4)`	–	(4,7)
`get(1)`	7	(4,7)
`add(2,2)`	–	(4,7,2)
`get(3)`	"`error`"	(4,7,2)
`remove(1)`	7	(4,2)
`add(1,5)`	–	(4,5,2)
`add(1,3)`	–	(4,3,5,2)
`add(4,9)`	–	(4,3,5,2,9)
`get(2)`	5	(4,3,5,2,9)
`set(3,8)`	2	(4,3,5,8,9)

5.1.2. El patrón adaptador

Las clases son frecuentemente escritas para dar funcionalidad similar a otras clases. El patrón de diseño *adaptador* se aplica a cualquier contexto donde se quiere modificar una clase existente para que sus métodos empaten con los de una clase relacionada diferente o interfaz. Una forma general para aplicar el patrón adaptador es definir la nueva clase de tal forma que esta contenga una instancia de la clase vieja como un campo oculto, e implementar cada método de la nueva clase usando métodos de esta variable de instancia oculta. El resultado de aplicar el patrón adaptador es que una nueva clase que realiza casi las mismas funciones que una clase previa, pero en una forma más conveniente, ha sido creada.

Con respecto al ADT lista arreglo, se observa que este es suficiente para definir una clase adaptadora para el ADT deque, como se ve en la tabla 5.1:

Método Deque	Realización con Métodos de Lista Arreglo
size(), isEmpty()	size(),isEmpty()
getFirst()	get(0)
getLast()	get(size()-1)
addFirst(e)	add(0,e)
addLast(e)	add(size(),e)
removeFirst()	remove(0)
removeLast()	remove(size()-1)

Cuadro 5.1: Realización de una deque por medio de una lista arreglo.

5.1.3. Implementación simple con un arreglo

Una opción obvia para implementar el ADT lista arreglo es usar un arreglo A, donde $A[i]$ guarda una referencia al elemento con índice i. Se escoge el tamaño N del arreglo A lo suficientemente grande, y se mantiene el número de elementos en una variable de instancia, $n < N$.

Los detalles de esta implementación del ADT lista arreglo son simples. Para implementar la operación `get(i)`, por ejemplo, sólo se regresa $A[i]$. La implementación de los métodos `add`(i, e) y `remove(i)` son dados a continuación. Una parte importante y consumidora de tiempo de esta implementación involucra el desplazamiento de elementos hacia adelante y hacia atrás para mantener las celdas ocupadas en el arreglo de forma continua. Estas operaciones de desplazamiento son requeridas para mantener la regla de guardar siempre un elemento cuyo índice en la lista es i en el índice i en el arreglo A, ver figura 5.1.

Algoritmo add(i, e):
  Para $j \leftarrow n-1, n-2, \ldots, i$ **Hacer**
    $A[j+1] \leftarrow A[j]$ {hacer espacio para el nuevo elemento}
  $A[i] \leftarrow e$
  $n \leftarrow n+1$

Algoritmo remove(i):
  $e \leftarrow A[i]$ { e es una variable temporal }
  Para $j \leftarrow i, i+1, \ldots, n-2$ **Hacer**
    $A[j] \leftarrow A[j+1]$ {llenar el espacio del elemento quitado}
  $n \leftarrow n-1$
  Regresar e

El rendimiento de una implementación simple con arreglo

El cuadro 5.2 muestra los tiempos de ejecución en el peor caso de un arreglo lista con n elementos realizados por medio de un arreglo. Los métodos `isEmpty`, `size`, `get` y `set` se ejecutan en tiempo $O(1)$, pero los métodos de inserción y borrado pueden tomar más tiempo que este. En particular, `add`(i, e) se ejecuta

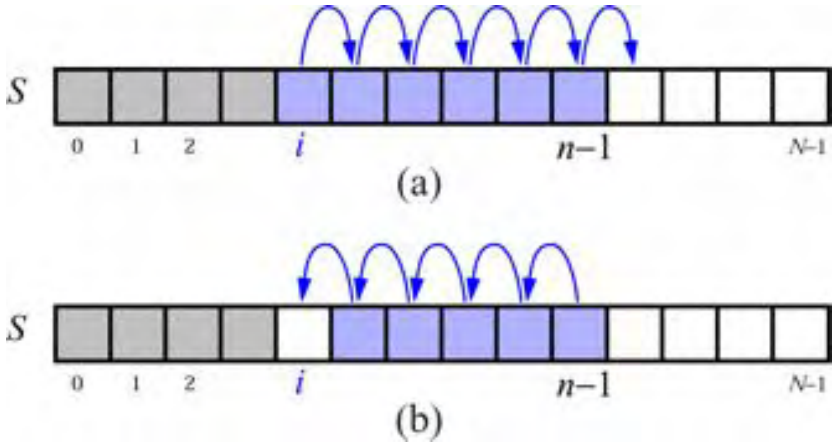

Figura 5.1: Implementación basada en un arreglo de una lista arreglo S que está guardando n elementos: (a) desplazando hacia adelante para una inserción en el índice i; (b) desplazamiento hacia atrás para un borrado en el índice i.

en tiempo $O(n)$. En efecto, el peor caso para esta operación ocurre cuando $i = 0$, ya que todos los n elementos existentes tienen que ser desplazados hacia adelante. Un argumento similar se aplica al método `remove(`i`)`, el cual corre en tiempo $O(n)$, porque se tiene que desplazar hacia atrás $n - 1$ elementos en el peor caso cuando $i = 0$. De hecho, suponiendo que cada índice posible es igualmente probable de que sea pasado como un argumento a estas operaciones, su tiempo promedio es $O(n)$, ya que se tienen que desplazar $n/2$ elementos en promedio.

Método	Tiempo
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Cuadro 5.2: Rendimiento de un lista arreglo con n elementos usando un arreglo. El espacio usado es $O(N)$, donde N es el tamaño del arreglo.

Observando más detenidamente en `add`(i, e) y `remove(`i`)`, se deduce que cada tiempo de ejecución es $O(n - i + 1)$, solamente para aquellos elementos en el índice i y mayores tienen que ser desplazadas hacia atrás o adelante. Así, insertar y quitar un elemento al final de una lista arreglo, usando los métodos `add`(n, e) y `remove`$(n - 1)$, respectivamente toman tiempo $O(1)$ cada uno. Por otra parte, esta observación tiene una consecuencia interesante para la adaptación del ADT

lista arreglo al ADT deque, dado en la sección 4.16, si el ADT lista arreglo en este caso es implementado por medio de un arreglo como se describe previamente, entonces los métodos `addLast` y `removeLast` del deque se ejecutan en tiempo $O(1)$. Sin embargo, los métodos `addFirst` y `removeFirst` del deque se ejecutan en tiempo $O(n)$.

Actualmente, con un poco de esfuerzo, se puede generar una implementación basada en un arreglo para que el ADT lista arreglo logre tiempo $O(1)$ para inserciones y borrados en el índice 0, al igual que inserciones y borrados al final de la lista arreglo. Para lograrlo se requiere que se de en la regla, que un elemento en el índice es guardado en el arreglo en el índice i, sin embargo, se tendría que usar una aproximación de un arreglo circular como el que fue usado en la sección 4.3 para implementar una cola.

5.1.4. Una interfaz simple y la clase java.util.ArrayList

Para construir una implementación Java del ADT lista arreglo, se muestra, en el listado 5.1, una interfaz Java, `ListaIndice`, que captura los métodos principales del ADT lista de arreglo. En este caso, se usa una `IndexOutOfBoundsException` para señalar un argumento índice inválido.

```
1  /**
2   * Una interfaz para listas arreglo.
3   */
4  public interface ListaIndice<E> {
5    /** Regresa el número de elementos en esta lista. */
6    public int size();
7    /** Prueba si la lista vacía. */
8    public boolean isEmpty();
9    /** Insertar un elemento e para estar en el índice i,
10     * desplazando todos los elementos después de este. */
11   public void add(int i, E e)
12     throws IndexOutOfBoundsException;
13   /** Regresar el elemento en el índice i, sin quitarlo. */
14   public E get(int i)
15     throws IndexOutOfBoundsException;
16   /** Quita y regresa el elemento en el índice i,
17     * desplazando los elementos después de este. */
18   public E remove(int i)
19     throws IndexOutOfBoundsException;
20   /** Reemplaza el elemento en el índice i con e,
21     * regresando el elemento previo en i. */
22   public E set(int i, E e)
23     throws IndexOutOfBoundsException;
24 }
```

Listado 5.1: La interfaz `ListaIndice` para el ADT lista arreglo.

La clase java.util.ArrayList

Java proporciona una clase, `java.util.ArrayList`, que implementa todos los métodos que se dieron previamente para el ADT lista arreglo. Esta incluye todos los métodos dados en el listado 5.1 para la interfaz `ListaIndice`. Por otra parte, la clase `java.util.ArrayList` tiene características adicionales a las que fueron dadas en el ADT lista arreglo simplificado. Por ejemplo, la clase

`java.util.ArrayList` también incluye un método, `clear()`, el cual remueve todos los elementos de la lista de arreglo, y un método, `toArray()`, el cual regresa un arreglo conteniendo todos los elementos de la lista arreglo en el mismo orden. Además, la clase `java.util.ArrayList` también tiene métodos para buscar en la lista, incluyendo un método `indexOf`(e), el cual regresa el índice de la primera ocurrencia de un elemento igual a e en la lista arreglo, y un método `lastIndexOf`(e), el cual regresa el índice de la última ocurrencia de un elemento igual a e en la lista arreglo. Ambos métodos regresan el valor índice inválido -1 si no se encuentra un elemento igual a e.

5.1.5. Lista arreglo usando un arreglo extendible

Además de implementar los métodos de la interfaz `ListaIndice` y algunos otros métodos útiles, la clase `java.util.ArrayList` proporciona una característica interesante que supera una debilidad en la implementación simple de la lista arreglo.

La mayor debilidad de la implementación simple para el ADT lista arreglo dado en la sección 5.1.3, es que este requiere una especificación avanzada de una capacidad fija, N, para el número total de elementos que podrían ser guardados en la lista arreglo. Si el número actual de elementos, n, de la lista arreglo es mucho menor que N, entonces la implementación desperdiciará espacio. Peor aún, si n se incrementa y rebasa a N, entonces esta implementación fallará.

Para evitar lo anterior, `java.util.ArrayList` usa una técnica interesante de arreglo extendible así nunca se debe preocupar de que el arreglo sea desbordado cuando se usa esta clase.

Al igual que la clase `java.util.ArrayList`, se proporciona un medio para crecer el arreglo A que guarda los elementos de una lista arreglo S. Por supuesto, en Java y en otros lenguajes de programación, no se puede crecer el arreglo A; su capacidad está fija a algún número N, como ya se ha visto. Cuando un *desbordamiento* ocurre, esto es, cuando $n = N$ y se hace una llamada al método `add`, se realizan los siguientes pasos adicionales.

1. Reservar un nuevo arreglo B de capacidad $2N$.
2. Hacer $B[i] \leftarrow A[i]$, para $i = 0, \ldots, N-1$.
3. Hacer $A \leftarrow B$, esto es, se usa B como el arreglo que soporta a S.
4. Insertar el nuevo elemento en A.

Esta estrategia de reemplazo de arreglo es conocida como *arreglo extendible*, esta puede ser vista como extender la terminación del arreglo subyacente para hacerle espacio para más elementos, ver figura 5.2. Intuitivamente, este estrategia es parecida a la del cangrejo ermitaño, el cual se cambia a un caracol más grande cuando sobrepasa al anterior.

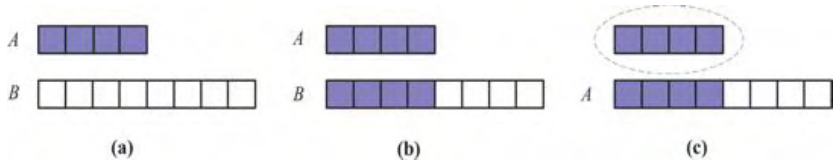

Figura 5.2: Los tres pasos para "crecer" un arreglo extendible: (a) crea un nuevo arreglo B; (b) copiar los elementos de A a B; (c) reasignar referencia A al nuevo arreglo

Implementar la interfaz `ListaIndice` con un arreglo extendible

Se dan porciones de una implementación Java del ADT lista arreglo usando un arreglo extendible en el código 5.2. Esta clase sólo proporciona los medios para crecer. Un ejercicio es la implementación para poder encoger.

```
1  /** Realización de una lista indizada por medio de un arreglo, el cual es
2    * doblado cuando el tamaño de la lista excede la capacidad del arreglo.
3    */
4  public class ArrayListaIndice<E> implements ListaIndice<E> {
5    private E[] A; // arreglo almacenando los elementos de la lista indizada
6    private int capacidad = 16;  // tamaño inicial del arreglo A
7    private int tam = 0;  // número de elementos guardados en la lista
8    /** Crear la lista indizada con capacidad inicial de 16. */
9    public ArrayListaIndice() {
10     A = (E[]) new Object[capacidad]; // el compilador podría advertir,
11                                      // pero está bien
12   }
13   /** Insertar un elemento en el índice dado. */
14   public void add(int r, E e)
15     throws IndexOutOfBoundsException {
16     revisarIndice(r, size() + 1);
17     if (tam == capacidad) {  // un sobreflujo
18       capacidad *= 2;
19       E[] B =(E[]) new Object[capacidad]; // el compilador podría advertir
20       for (int i=0; i<tam; i++)
21         B[i] = A[i];
22       A = B;
23     }
24     for (int i=tam-1; i>=r; i--)  // desplazar elementos hacia adelante
25       A[i+1] = A[i];
26     A[r] = e;
27     tam++;
28   }
29   /** Quitar el elemento guardado en el índice dado. */
30   public E remove(int r)
31     throws IndexOutOfBoundsException {
32     revisarIndice(r, size());
33     E temp = A[r];
34     for (int i=r; i<tam-1; i++)  // desplazar elementos hacia atrás
35       A[i] = A[i+1];
36     tam--;
37     return temp;
38   }
39   /** Revisa si el índice dado está en el rango [0, n - 1] */
40   protected void revisarIndice(int r, int n) //
41     throws IndexOutOfBoundsException {  //
42     if (r < 0 || r >= n)
43       throw new IndexOutOfBoundsException("Índice ilegal : " + r);
44   }
45   /** Regresar el número de elementos en la lista indizada. */
```

Listado 5.2: Parte de la clase `ArrayListaIndice` realizando el ADT lista arreglo por medio de un arreglo extendible. El método `revisarIndice` vigila que el índice r está entre 0 y $n-1$.

5.2. Listas nodo

Usar un índice no es la única forma de referirse al lugar donde un elemento aparece en una secuencia. Si se tiene una secuencia S implementada con una lista enlazada, simple o doble, entonces podría ser más natural y eficiente usar un nodo en vez de un índice como medio de identificación para acceder a S o actualizar. Se define en esta sección el ADT lista nodo el cual abstrae la estructura de datos lista enlazada concreta, secciones 2.2 y 2.3, usando un ADT posición relacionada que abstrae la noción de "lugar" en una lista nodo.

5.2.1. Operaciones basadas en nodos

Sea S una lista enlazada simple o doble. Se desearía definir métodos para S que tomen nodos como parámetros y proporcionen nodos como tipos de regreso. Tales métodos podrían dar aumentos de velocidad significativos sobre los métodos basados en índices, porque encontrar el índice de un elemento en una lista enlazada requiere buscar a través de la lista incrementalmente desde el inicio o final, contando los elementos conforme se recorra.

Por ejemplo, se podría definir un método hipotético `remove`(v) que quite el elemento de S guardado en el nodo v de la lista. Usando un nodo como un parámetro permite remover un elemento en tiempo $O(1)$ yendo directamente al lugar donde está guardado el nodo y entonces "desenlazando" este nodo mediante una actualización a los enlaces *sig* y *prev* de sus vecinos. De igual modo, se puede insertar, en tiempo O(1), un nuevo elemento e en S con una operación tal como `addAfter`(v, e), la cual indica el nodo v después del cual el nodo del nuevo elemento debería ser insertado. En este caso se "enlaza" el nuevo nodo.

Para abstraer y unificar las diferentes formas de guardar elementos en las diversas implementaciones de una lista, se introduce el concepto de *posición*, la cual formaliza la noción intuitiva de "lugar" de un elemento relativo a los otros en la lista.

5.2.2. Posiciones

A fin de ampliar de forma segura el conjunto de operaciones para listas, se abstrae una noción de "posición" que permite usar la eficiencia de la implementación de las listas simples enlazadas o dobles sin violar los principios del diseño orientado a objetos. En este marco, se ve una lista como una colección de elementos que guardan cada elemento en una posición y que mantienen estas posiciones organizadas en un orden lineal. Una posición es por sí misma un tipo de dato abstracto que soporta el siguiente método simple:

`elemento`():regresa el elemento guardado en esta posición.

Una posición está siempre definida *relativamente*, esto es, en términos de sus vecinos. En una lista, una posición p siempre estará "después" de alguna posición q y "antes" de alguna posición s, a menos que p sea la primera posición o la última. Una posición p, la cual está asociada con algún elemento e en una lista S, no cambia, aún si el índice de e cambia en S, a menos que se quite explícitamente e, y por lo tanto, se destruya la posición p. Por otra parte, la posición p no cambia incluso si se reemplaza o intercambia el elemento e guardado en p con otro elemento. Estos hechos acerca de las posiciones permiten definir un conjunto de métodos de lista basados en posiciones que usan objetos posición como parámetros y también proporcionan objetos posición como valores de regreso.

5.2.3. El tipo de dato abstracto lista nodo

Usando el concepto de posición para encapsular la idea de "nodo" en una lista, se puede definir otro tipo de ADT secuencia llamado el ADT *lista nodo*. Este ADT soporta los siguientes métodos para una lista S:

`first`():regresa la posición del primer elemento de S; un error ocurre si S está vacía.
`last`():regresa la posición del último elemento de S; un error ocurre si S está vacía.
`prev`(p):regresa la posición del elemento de S precediendo al de la posición p; un error ocurre si p está en la primera posición.
`next`(p):regresa la posición del elemento de S siguiendo al de la posición p; un error ocurre si p está en la última posición.

Los métodos anteriores permiten referirse a posiciones relativas en una lista, comenzando en el inicio o final, y moviéndose incrementalmente a la izquierda o a la derecha de la lista. Estas posiciones pueden ser pensadas como nodos en la lista, pero observar que no hay referencias específicas a objetos nodos. Además, si se proporciona una posición como un argumento a un método lista, entonces la posición deberá representar una posición válida en aquella lista.

Métodos de actualización de la lista nodo

Además de los métodos anteriores y los métodos genéricos `size` e `isEmpty`, se incluyen también los siguientes métodos de actualización para el ADT lista nodo, los cuales usan objetos posición como parámetros y regresan objetos posición como valores de regreso algunos de ellos.

`set`(p, e):reemplaza el elemento en la posición p con e, regresando el elemento anterior en la posición p.
`addFirst`(e):inserta un nuevo elemento e en S como el primer elemento.
`addLast`(e):inserta un nuevo elemento e en S como el último elemento.
`addBefore`(p, e):inserta un nuevo elemento e en S antes de la posición p.
`addAfter`(p, e):inserta un nuevo elemento e en S después de la posición p.
`remove`(p):quita y regresa el elemento e en la posición p en S, invalidando esta posición en S.

El ADT lista nodo permite ver una colección ordenada de objetos en términos de sus lugares, sin tenerse que preocupar acerca de la forma exacta como estos lugares están representados, ver figura 5.3.

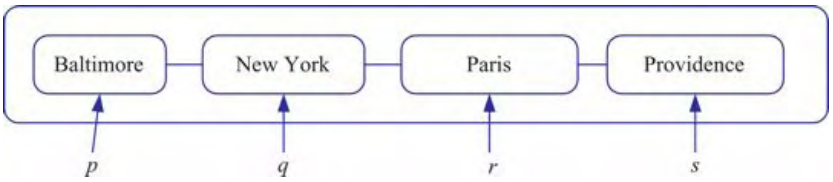

Figura 5.3: Una lista nodo. Las posiciones en el orden actual son p, q, r, y s.

Podría parecer a primera vista que hay redundancia en el repertorio anterior de operaciones para el ADT lista nodo, ya que se puede hacer la operación `addFirst`(e) con `addBefore(first(),`e`)`, y la operación `addLast`(e) con `addAfter(last(),e)`. Pero estas sustituciones sólo pueden ser hechas para una lista no vacía.

Observar que una condición de error ocurre si una posición pasada como argumento a una de las operaciones de la lista es inválida. Las razones para que una posición p sea inválida incluyen:

- $p =$ **null**
- p fue previamente borrada de la lista
- p es una posición de una lista diferente
- p es la primera posición de la lista y se llama `prev`(p)
- p es la última posición de la lista y se llama `next`(p)

Se muestra enseguida una serie de operaciones para una lista nodo S inicialmente vacía. Se usan las variables p_1, p_2, etc., para denotar diferentes posiciones, y se muestra el objeto guardado actualmente en tal posición entre paréntesis.

Operación	Salida	S
`addFirst`(8)	–	(8)
$p_1 \leftarrow$ `first`$()$	[8]	(8)
`addAfter`$(p_1, 5)$	–	(8,5)
$p_2 \leftarrow$ `next`(p_1)	[5]	(8,5)
`addBefore`$(p_2, 3)$	–	(8,3,5)
$p_3 \leftarrow$ `prev`(p_2)	[3]	(8,3,5)
`addFirst`(9)	–	(9,8,3,5)
$p_2 \leftarrow$ `last`$()$	[5]	(9,8,3,5)
`remove`(`first`$())$	9	(8,3,5)
`set`$(p_3, 7)$	3	(8,7,5)
`addAfter`(`first`$(), 2)$	–	(8,2,7,5)

El ADT lista nodo, con su noción incorporada de posición, es útil para realizar ajustes. Por ejemplo, un programa que simula un juego de cartas podría simular la mano de cada persona como una lista nodo. Ya que la mayoría de las personas guarda las cartas de la misma figura juntas, insertar y sacar cartas de la mano de una persona podría ser implementado usando los métodos del ADT lista nodo, con las posiciones siendo determinadas por un orden natural de las figuras. Igualmente, un editor de texto simple encaja la noción de inserción posicional y remoción, ya que tales editores típicamente realizan todas las actualizaciones relativas a un *cursor*, el cual representa la posición actual en la lista de caracteres de texto que está siendo editada.

Una interfaz Java representando el ADT posición está dado en el listado 5.3.

```
1 public interface Posicion<E> {
2   /** Regresa el elemento guardado en esta posición. */
3   E elemento();
4 }
```

Listado 5.3: Una interfaz Java para el ADT posición.

Una interfaz, con los métodos requeridos, para el ADT lista nodo, llamada `ListaPosicion`, está dada en el listado 5.4. Esta interfaz usa las siguientes excepciones para indicar condiciones de error.

BoundaryViolationException: lanzada si un intento es hecho al acceder una elemento cuya posición está fuera del rango de posiciones de la lista, por ejemplo llamando al método `next` en la última posición de la secuencia.

InvalidPositionException: lanzada si una posición dada como argumento no es válida, por ejemplo, esta es una referencia `null` o esta no tiene lista asociada.

```
1 import java.util.Iterator;
2 /**
3  * Una interfaz para listas posicionales.
4  *
5  */
6 public interface ListaPosicion<E> extends Iterable<E> {
7   /** Regresa un iterador de todos los elementos en la lista. */
```

```
8   public Iterator<E> iterator();
9   /** Regresa una colección iterable de todos los nodos en la lista. */
10  public Iterable<Posicion<E>> positions();
11  /** Regresa el número de elementos en esta lista. */
12  public int size();
13  /** Indica si la lista está vacía. */
14  public boolean isEmpty();
15  /** Regresa el primer nodo en la lista. */
16  public Posicion<E> first();
17  /** Regresa el último nodo en la lista. */
18  public Posicion<E> last();
19  /** Regresa el nodo siguiente a un nodo dado en la lista. */
20  public Posicion<E> next(Posicion<E> p)
21    throws InvalidPositionException, BoundaryViolationException;
22  /** Regresa el nodo anterior a un nodo dado en la lista. */
23  public Posicion<E> prev(Posicion<E> p)
24    throws InvalidPositionException, BoundaryViolationException;
25  /** Inserta un elemento en el frente de la lista,
26    * regresando la nueva posición. */
27  public void addFirst(E e);
28  /** Inserta un elemento en la parte de atrás de la lista,
29    * regresando la nueva posición. */
30  public void addLast(E e);
31  /** Inserta un elemento después del nodo dado en la lista. */
32  public void addAfter(Posicion<E> p, E e)
33    throws InvalidPositionException;
34  /** Inserta un elemento antes del nodo dado en la lista. */
35  public void addBefore(Posicion<E> p, E e)
36    throws InvalidPositionException;
37  /** Quita un nodo de la lista, regresando el elemento guardado. */
38  public E remove(Posicion<E> p) throws InvalidPositionException;
39  /** Reemplaza el elemento guardado en el nodo dado,
40     regresando el elemento viejo. */
41  public E set(Posicion<E> p, E e) throws InvalidPositionException;
42 }
```

Listado 5.4: Interfaz para el ADT lista nodo.

Otro adaptador Deque

El ADT lista nodo es suficiente para definir una clase adaptadora para el ADT deque, como se muestra en la tabla 5.3.

Método Deque	Realización con Métodos de la lista nodo
size(), isEmpty()	size(), isEmpty()
getFirst()	first().element()
getLast()	last().element()
addFirst(e)	addFirst(e)
addLast(e)	addLast(e)
removeFirst()	remove(first())
removeLast()	remove(last())

Cuadro 5.3: Realización de una deque por medio de una lista nodo.

5.2.4. Implementación de una lista doblemente enlazada

Si se quisiera implementar el ADT lista nodo usando una lista doblemente enlazada, sección 2.3, se puede hacer que los nodos de la lista enlazada implementen el ADT posición. Esto es, se tiene cada nodo implementando la interfaz `Posicion` y por lo tanto definen un método, `element()`, el cual regresa el elemento guardado en el nodo. Así, los propios nodos actúan como posiciones. Son vistos internamente por la lista enlazada como nodos, pero desde el exterior, son vistos solamente como posiciones. En la vista interna, se puede dar a cada nodo v variables de instancia `prev` y `next` que respectivamente refieran a los nodos predecesor y sucesor de v, los cuales podrían ser de hecho nodos centinelas cabeza y cola marcando el inicio o el final de la lista. En lugar de usar las variables `prev` y `next` directamente, se definen los métodos `getPrev`, `setPrev`, `getNext`, y `setNext` de un nodo para acceder y modificar estas variables.

En el listado 5.5, se muestra una clase Java, `NodoD`, para los nodos de una lista doblemente enlazada implementando el ADT posición. Esta clase es similar a la clase `DNodo` mostrada en el listado 2.9, excepto que ahora los nodos guardan un elemento genérico en lugar de una cadena de caracteres. Observar que las variables de instancia `prev` y `next` en la clase `NodoD` son referencias privadas a otros objetos `NodoD`.

```
1  public class NodoD<E> implements Posicion<E> {
2    private NodoD<E> prev, next;  // Referencia a los nodos anterior y posterior
3    private E elemento; // Elemento guardado en esta posición
4    /** Constructor */
5    public NodoD(NodoD<E> newPrev, NodoD<E> newNext, E elem) {
6      prev = newPrev;
7      next = newNext;
8      elemento = elem;
9    }
10   // Método de la interfaz Posicion
11   public E elemento() throws InvalidPositionException {
12     if ((prev == null) && (next == null))
13       throw new InvalidPositionException("¡La posición no está en una lista!");
14     return elemento;
15   }
16   // Métodos accesores
17   public NodoD<E> getNext() { return next; }
18   public NodoD<E> getPrev() { return prev; }
19   // Métodos actualizadores
20   public void setNext(NodoD<E> nuevoNext) { next = nuevoNext; }
21   public void setPrev(NodoD<E> nuevoPrev) { prev = nuevoPrev; }
22   public void setElemento(E nuevoElemento) { elemento = nuevoElemento; }
23 }
```

Listado 5.5: Clase `NodoD` realizando un nodo de una lista doblemente enlazada implementando la interfaz `Posicion`.

Dada una posición p en S, se puede "desenvolver" p para revelar el nodo v subyacente mediante una *conversión* de la posición a un nodo. Una vez que se tiene un nodo v, se puede, por ejemplo, implementar el método `prev(`p`)` con v`.getPrev`, a menos que el nodo regresado por v`.getPrev` sea la cabeza, en tal caso se señala un error. Por lo tanto, posiciones en una implementación lista doblemente enlazada se pueden soportar en una forma orientada al objeto sin ningún gasto adicional de tiempo o espacio.

Considerar ahora como se podría implementar el método `addAfter`(p, e), para insertar un elemento e después de la posición p. Se crea un nuevo nodo v para mantener el elemento e, se enlaza v en su lugar en la lista, y entonces se actualizan las referencias `next` y `prev` de los dos vecinos de v. Este método está dado en el siguiente pseudocódigo, y se ilustra en la figura 5.4, recordando el uso de centinelas, observar que este algoritmo trabaja aún si p es la última posición real.

Algoritmo addAfter(p, e):
Crear un nuevo nodo v
v.setElement(e)
v.setPrev(p) { enlazar v a su predecesor }
v.setNext$(p$.getNext$())$ { enlazar v a su sucesor }
$(p$.getNext$())$.setPrev(v) { enlazar el viejo sucesor de p a v }
p.setNext(v) { enlazar p a su nuevo sucesor, v }

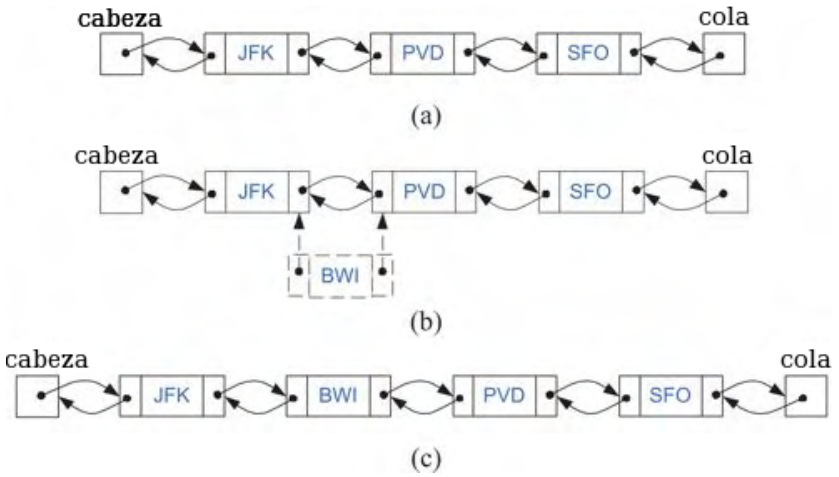

Figura 5.4: Agregar un nuevo nodo después de la posición "ACA":(a) antes de la inserción; (b) creación de un nodo v con el elemento "CUN" y el enlazado a la lista; (c) después de la inserción.

Los algoritmos para los métodos `addBefore`, `addFirst`, y `addLast` son similares al método `addAfter`. Se dejan sus detalles como un ejercicio.

Considerar, ahora, el método `remove`(e), el cual quita el elemento e guardado en la posición p. Para realizar esta operación, se enlazan los dos vecinos de p para referirse entre ellos como nuevos vecinos—desenlazando p. Observar que después de que p está desenlazado, no hay nodos apuntando a p; por lo tanto, el recolector de basura puede reclamar el espacio de p. Este algoritmo está dado en el siguiente pseudocódigo y se ilustra en la figura 5.5. Recordando el uso de los centinelas cabeza y cola, este algoritmo trabaja aún si p es la primera posición, la última, o solamente una posición real en la lista.

Algoritmo remove(p):
  $t \leftarrow p$.element() { una variable temporal para guardar el valor de regreso }
  (p.getPrev()).setNext(p.getNext()) { desenlazar p }
  (p.getNext()).setPrev(p.getPrev())
  p.setPrev(**null**) { invalidando la posición p }
  p.setNext(**null**)
  regresar t

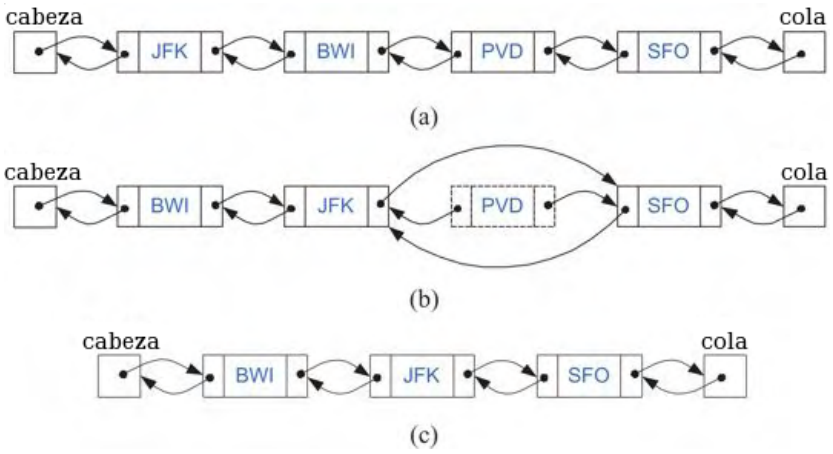

Figura 5.5: Quitando el objeto guardado en la posición para "MEX": (a) antes de la remoción; (b) desenlazando el nodo viejo; (c) después de la remoción.

En conclusión, usando una lista doblemente enlazada, se pueden ejecutar todos los métodos del ADT lista nodo en tiempo $O(1)$. Así, una lista doblemente enlazada es una implementación eficiente del ADT lista nodo.

Implementación de la lista nodo en Java

La clase Java `ListaNodoPosicion`, la cual implementa el ADT lista nodo usando una lista doblemente enlazada, se muestra en el listado 5.6. En esta clase se tienen declaradas variables de instancia para guardar el número de elementos en la lista, así como los centinelas especiales para manejar más fácilmente la lista. También se tiene un constructor sin parámetros que permite crear una lista nodo vacía. En esta clase además de implementar los métodos declarados en la interfaz `ListaPosicion` se tienen otros métodos de conveniencia, por ejemplo para revisar si una posición es válida, probar si una posición es la primera o la última, intercambiar los contenidos de dos posiciones, u obtener una representación textual de la lista.

```
1 import java.util.Iterator;
2 /**
3  * Realización de una ListaPosición usando una lista doblemente enlazada
4  * de nodos.
5  *
```

```
 */
public class ListaNodoPosicion<E> implements ListaPosicion<E> {
  protected int numElts;              // número de elementos en la lista
  protected NodoD<E> cabeza, cola; // centinelas especiales
  /** Constructor que crea una lista vacía; tiempo O(1) */
  public ListaNodoPosicion() {
    numElts = 0;
    cabeza = new NodoD<E>(null, null, null);  // crea cabeza
    cola = new NodoD<E>(cabeza, null, null); // crea cola
    cabeza.setNext(cola);  // hacer que cabeza apunte a cola
  }
 /** Regresa el número de elementos en la lista; tiempo O(1) */
  public int size() { return numElts; }
  /** Valida si la lista está vacía;  tiempo O(1) */
  public boolean isEmpty() { return (numElts == 0); }
  /** Regresa la primera posición en la lista; tiempo O(1) */
  public Posicion<E> first()
      throws EmptyListException {
    if (isEmpty())
      throw new EmptyListException("La lista está vacía");
    return cabeza.getNext();
  }
  /** Regresa la última posición en la lista; tiempo O(1) */
  public Posicion<E> last()
      throws EmptyListException {
    if (isEmpty())
      throw new EmptyListException("La lista está vacía");
    return cola.getPrev();
  }
  /** Regresa la posición anterior de la dada; tiempo O(1) */
  public Posicion<E> prev(Posicion<E> p)
      throws InvalidPositionException, BoundaryViolationException {
    NodoD<E> v = revisaPosicion(p);
    NodoD<E> prev = v.getPrev();
    if (prev == cabeza)
      throw new BoundaryViolationException
        ("No se puede avanzar más allá del principio de la lista");
    return prev;
  }
  /** Regresa la posición después de la dada; tiempo O(1) */
  public Posicion<E> next(Posicion<E> p)
      throws InvalidPositionException, BoundaryViolationException {
    NodoD<E> v = revisaPosicion(p);
    NodoD<E> next = v.getNext();
    if (next == cola)
      throw new BoundaryViolationException
        ("No se puede avanzar más allá del final de la lista");
    return next;
  }
  /** Insertar el elemento dado antes de la posición dada: tiempo O(1) */
  public void addBefore(Posicion<E> p, E elemento)
      throws InvalidPositionException {
    NodoD<E> v = revisaPosicion(p);
    numElts++;
    NodoD<E> nodoNuevo = new NodoD<E>(v.getPrev(), v, elemento);
    v.getPrev().setNext(nodoNuevo);
    v.setPrev(nodoNuevo);
  }
  /** Insertar el elemento dado después de la posición dada: tiempo O(1) */
  public void addAfter(Posicion<E> p, E elemento)
      throws InvalidPositionException {
    NodoD<E> v = revisaPosicion(p);
    numElts++;
    NodoD<E> nodoNuevo = new NodoD<E>(v, v.getNext(), elemento);
    v.getNext().setPrev(nodoNuevo);
    v.setNext(nodoNuevo);
  }
  /** Insertar el elemento dado al inicio de la lista, regresando
```

```
74      * la nueva posición; tiempo O(1) */
75    public void addFirst(E elemento) {
76      numElts++;
77      NodoD<E> nodoNuevo = new NodoD<E>(cabeza, cabeza.getNext(), elemento);
78      cabeza.getNext().setPrev(nodoNuevo);
79      cabeza.setNext(nodoNuevo);
80    }
81    /** Insertar el elemento dado al final de la lista, regresando
82      * la nueva posición; tiempo O(1) */
83    public void addLast(E elemento) {
84      numElts++;
85      NodoD<E> ultAnterior = cola.getPrev();
86      NodoD<E> nodoNuevo = new NodoD<E>(ultAnterior, cola, elemento);
87      ultAnterior.setNext(nodoNuevo);
88      cola.setPrev(nodoNuevo);
89    }
90    /** Quitar la posición dada de la lista; tiempo O(1) */
91    public E remove(Posicion<E> p)
92        throws InvalidPositionException {
93      NodoD<E> v = revisaPosicion(p);
94      numElts--;
95      NodoD<E> vPrev = v.getPrev();
96      NodoD<E> vNext = v.getNext();
97      vPrev.setNext(vNext);
98      vNext.setPrev(vPrev);
99      E vElem = v.elemento();
100     // desenlazar la posición de la lista y hacerlo inválido
101     v.setNext(null);
102     v.setPrev(null);
103     return vElem;
104   }
105   /** Reemplaza el elemento en la posición dada con el nuevo elemento
106     * y regresar el viejo elemento; tiempo O(1) */
107   public E set(Posicion<E> p, E elemento)
108       throws InvalidPositionException {
109     NodoD<E> v = revisaPosicion(p);
110     E elemVjo = v.elemento();
111     v.setElemento(elemento);
112     return elemVjo;
113   }
114   /** Regresa un iterador de todos los elementos en la lista. */
115   public Iterator<E> iterator() { return new IteradorElemento<E>(this); }
116   /** Regresa una colección iterable de todos los nodos en la lista. */
117   public Iterable<Posicion<E>> positions() {  // crea una lista de posiciones
118     ListaPosicion<Posicion<E>> P = new ListaNodoPosicion<Posicion<E>>();
119     if (!isEmpty()) {
120       Posicion<E> p = first();
121       while (true) {
122         P.addLast(p); // agregar posición p como el últ. elemento de la lista P
123         if (p == last())
124           break;
125         p = next(p);
126       }
127     }
128     return P; // regresar P como el objeto iterable
129   }
130   // Métodos de conveniencia
131   /** Revisa si la posición es válida para esta lista y la convierte a
132     * NodoD si esta es válida: tiempo O(1) */
133   protected NodoD<E> revisaPosicion(Posicion<E> p)
134     throws InvalidPositionException {
135     if (p == null)
136       throw new InvalidPositionException
137         ("Posición Null pasada a ListaNodo");
138     if (p == cabeza)
139       throw new InvalidPositionException
140         ("El nodo cabeza no es una posición válida");
141     if (p == cola)
```

```
142         throw new InvalidPositionException
143           ("El nodo terminación no es una posición válida");
144       try {
145         NodoD<E> temp = (NodoD<E>) p;
146         if ((temp.getPrev() == null) || (temp.getNext() == null))
147           throw new InvalidPositionException
148             ("Posición no pertenece a una ListaNodo válida");
149         return temp;
150       } catch (ClassCastException e) {
151         throw new InvalidPositionException
152         ("Posición es de un tipo incorrecto para esta lista");
153       }
154     }
155     /** Valida si una posición es la primera; tiempo O(1). */
156     public boolean isFirst(Posicion<E> p)
157       throws InvalidPositionException {
158       NodoD<E> v = revisaPosicion(p);
159       return v.getPrev() == cabeza;
160     }
161     /** Valida si una posición es la última; tiempo O(1). */
162     public boolean isLast(Posicion<E> p)
163         throws InvalidPositionException {
164       NodoD<E> v = revisaPosicion(p);
165       return v.getNext() == cola;
166     }
167     /** Intercambia los elementos de dos posiciones dadas; tiempo O(1) */
168     public void swapElements(Posicion<E> a, Posicion<E> b)
169         throws InvalidPositionException {
170       NodoD<E> pA = revisaPosicion(a);
171       NodoD<E> pB = revisaPosicion(b);
172       E temp = pA.elemento();
173       pA.setElemento(pB.elemento());
174       pB.setElemento(temp);
175     }
176     /** Regresa una representación textual de una lista nodo usando for-each */
177     public static <E> String forEachToString(ListaPosicion<E> L) {
178       String s = "[";
179       int i = L.size();
180       for (E elem: L) {
181         s += elem; // moldeo implícito del elemento a String
182         i--;
183         if (i > 0)
184           s += ", "; // separar elementos con una coma
185       }
186       s += "]";
187       return s;
188     }
189     /** Regresar una representación textual de una lista nodo dada */
190     public static <E> String toString(ListaPosicion<E> l) {
191       Iterator<E> it = l.iterator();
192       String s = "[";
193       while (it.hasNext()) {
194         s += it.next();  // moldeo implícito del elemento a String
195         if (it.hasNext())
196           s += ", ";
197         }
198       s += "]";
199       return s;
200     }
201     /** Regresa una representación textual de la lista */
202     public String toString() {
203       return toString(this);
204     }
205 }
```

Listado 5.6: La clase `ListaNodoPosicion` implementando el ADT lista nodo con una lista doblemente enlazada.

5.3. Iteradores

Un cómputo típico en una lista arreglo, lista, o secuencia es recorrer sus elementos en orden, uno a la vez, por ejemplo, para buscar un elemento específico.

5.3.1. Los tipos de dato abstracto Iterador e Iterable

Un *iterador* (*iterator*) es un patrón de diseño de software que abstrae el proceso de explorar a través de una colección de elementos un elemento a la vez. Un iterador consiste de una secuencia S, un elemento actual en S, y una forma de pasar al siguiente elemento en S y hacerlo el elemento actual. Así, un iterador extiende el concepto del ADT posición que se introdujo en la sección 5.2. De hecho, una posición puede ser pensada como un iterador que no va a ninguna parte. Un iterador encapsula los conceptos de "lugar" y "siguiente" en una colección de objetos.

Se define el ADT iterador soportando los siguientes dos métodos:

`hasNext`():prueba si hay elementos dejados en el iterador.
`next`():regresa el siguiente elemento en el iterador.

El ADT iterador tiene la noción de (*cursor* del elemento "actual" en recorrido de una secuencia. El primer elemento en un iterador es regresado por la primera llamada al método `next`, asumiendo que el iterador contiene al menos un elemento.

Un iterador da un esquema unificado para acceder a todos los elementos de una colección de objetos en una forma que es independiente de la organización específica de la colección. Un iterador para una lista arreglo, lista nodo, o secuencia deberán regresar los elementos de acuerdo a su orden lineal.

Iteradores simples en Java

Java proporciona un iterador a través de su interfaz `java.util.Iterator`. La clase `java.util.Scanner` implementa esta interfaz. Esta interfaz soporta un método adicional opcional para quitar el elemento previamente regresado de la colección. Esta funcionalidad, de quitar elementos a través de un iterador, es algo controversial desde un punto de vista orientado al objeto, sin embargo, no es sorprendente que su implementación por las clases es opcional. Java también da la interfaz `java.util.Enumeration`, la cual es históricamente más vieja que la interfaz iterador y usa los nombres `hasMoreElements()` y `nextElement()`.

El tipo de dato abstracto Iterable

Para dar un mecanismo unificado genérico para explorar a través de una estructura de datos, los ADT que guardan colecciones de objetos deberían soportar el siguiente método:

`iterator`():regresa un iterador de los elementos en la colección.

Este método está soportado por la clase `java.util.ArrayList` y es muy importante, que hay una interfaz completa `java.lang.Iterable` conteniendo sólo este método. Este método puede simplificar al usuario la tarea de recorrer los elementos de una lista. Para garantizar que una lista nodo soporta el método anterior, por ejemplo, se podría agregar este método a la interfaz `ListaPosicion`, como se muestra en el listado 5.4. En este caso, también se quisiera indicar que `ListaPosicion` extiende `Iterable`. Por lo tanto, se asume que las listas arreglo y listas nodo soportan el método `iterator()`.

```
1 public interface ListaPosicion<E> extends Iterable<E> {
2   /** Regresa un iterador de todos los elementos en la lista. */
3   public Iterator<E> iterator();
```

Listado 5.7: Agregando el método `iterator` a la interfaz `ListaPosicion`

Dada tal definición de una `ListaPosicion`, se podría usar un iterador regresado por el método `iterator()` para crear una representación de cadena de una lista nodo, como se muestra en el listado 5.8.

```
1   /** Regresar una representación textual de una lista nodo dada */
2   public static <E> String toString(ListaPosicion<E> l) {
3     Iterator<E> it = l.iterator();
4     String s = "[";
5     while (it.hasNext()) {
6       s += it.next();  // moldeo implícito del elemento a String
7       if (it.hasNext())
8         s += ", ";
9       }
10    s += "]";
11    return s;
12  }
```

Listado 5.8: Ejemplo de un iterador Java usado para convertir una lista nodo a una cadena.

5.3.2. El ciclo Java For-Each

Como recorrer los elementos regresados por un iterador es una construcción común, Java proporciona una notación breve para tales ciclos, llamada el *ciclo for-each*. La sintaxis para tal ciclo es como sigue:

for (*tipo nombre : expresión*)
  sentencia del ciclo

donde *expresión* evalúa a una colección que implementa la interfaz `java.lang.Iterable`, *tipo* es el tipo de objeto regresado por el iterador para esta clase, y *nombre* es el nombre de una variable que tomará los valores de los elementos de este iterador en la *sentencia del ciclo*. Esta notación es la versión corta de la siguiente:

```
for (Iterator<tipo> it=expresión.iterator(); it.hasNext(); ) {
   Tipo nombre= it.next();
   sentencia del ciclo
}
```

Por ejemplo, si se tiene una lista, `valores`, de objetos `Integer`, y `valores` implementa `java.lang.Iterable`, entonces se pueden sumar todos los enteros en valores como sigue:

```
List<Integer> valores;
 ...sentencias que crean una nueva lista y la llenan con Integers.
int suma = 0;
for (Integer i : valores)
   suma += i; // desencajonar permite esto
```

Se podría leer el ciclo anterior como, "para cada `Integer i` en `valores`, hacer el cuerpo del ciclo, en este caso, sumar `i` a `suma`"

Además de la forma anterior del ciclo for-each, Java también permite que un ciclo for-each sea definido cuando *expresión* es un arreglo de tipo *Tipo*, el cual puede ser un tipo base o un tipo objeto. Por ejemplo se puede totalizar los enteros en un arreglo, v, el cual guarda los primeros diez enteros positivos, como sigue:

```
int[] v=1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
int total = 0;
for (int i:v)
   total += i;
```

5.3.3. Implementación de los iteradores

Una forma de implementar un iterador para una colección de elementos es hacer una "foto" de esta e iterar sobre esta. Esta aproximación involucraría guardar la colección en una estructura de datos separada que soporte acceso secuencial a sus elementos. Por ejemplo, se podrían insertar todos los elementos de la colección en una cola, en tal caso el método `hasNext()` podría corresponder a `!isEmpty()` y `next()` podría corresponder a `dequeue()`. Con esta aproximación, el método `iterator()` toma tiempo $O(n)$ para una colección de tamaño n. Como esta sobrecarga de copiado es relativamente costosa, se prefiere, en muchos casos, tener iteradores que operen sobre la misma colección, y no una copia.

Para implementar esta aproximación directa, se necesita solo realizar un seguimiento a donde el cursor del iterador apunta en la colección. Así, crear un nuevo iterador en este caso involucra crear un objeto iterador que represente un cursor colocado solo antes del primer elemento de la colección. Igualmente, hacer el método `next()` involucra devolver el siguiente elemento, si este existe, y mover el cursor pasando esta posición del elemento. Así, en esta aproximación, crear

un iterador cuesta tiempo $O(1)$, al igual que hacer cada uno de los métodos del iterador. Se muestra una clase implementando tal iterador en el listado 5.9, y se muestra en el listado 5.10 como este iterador podría ser usado para implementar el método `iterator()` en la clase `ListaNodoPosicion`.

```
1  public class ElementoIterador<E> implements Iterator<E> {
2    protected ListaPosicion<E> lista; // la lista subyacente
3    protected Posicion<E> cursor; // la siguiente posición
4    /** Crea un elemento iterador sobre la lista dada. */
5    public ElementoIterador(ListaPosicion<E> L) {
6      lista = L;
7      cursor = (lista.isEmpty())? null : lista.first();
8    }
9    /** Valida si el iterador tiene un objeto siguiente. */
10   public boolean hasNext() { return (cursor != null);  }
11   /** Regresa el siguiente objeto en el iterador. */
12   public E next() throws NoSuchElementException {
13     if (cursor == null)
14       throw new NoSuchElementException("No hay elemento siguiente");
15     E aRegresar = cursor.elemento();
16     cursor = (cursor == lista.last())? null : lista.next(cursor);
17     return aRegresar;
18   }
```

Listado 5.9: Una clase elemento iterador para una `ListaPosicion`.

```
1    /** Regresa un iterador de todos los elementos en la lista. */
2    public Iterator<E> iterator() { return new IteradorElemento<E>(this); }
```

Listado 5.10: El método `iterator()` de la clase `ListaNodoPosicion`.

Iteradores posición

Para ADTs que soporten la noción de posición, tal como las ADT lista y secuencia, se puede también dar el siguiente método:

`positions()`:regresa un objeto `Iterable` (como una lista arreglo o lista nodo) conteniendo las posiciones en la colección como elementos.

Un iterador regresado por este método permite recorrer las posiciones de una lista. Para garantizar que una lista nodo soporta este método, se podría agregar la interfaz `ListaPosicion`, entonces se debe agregar una implementación del método `positions()` en la clase `ListaNodoPosicion`, como se muestra en el listado 5.12. Este método usa la propia clase `ListaNodoPosicion` para crear una lista que contiene las posiciones de la lista original como sus elementos. Regresando esta lista posición como su objeto `Iterable` permitiendo llamar `iterator()` sobre este objeto para obtener un iterador de posiciones de la lista original.

```
1  public interface ListaPosicion<E> extends Iterable<E> {
2    // ... todos los otros métodos de la ADT lista ...
3    /** Regresa una colección iterable de todos los nodos en la lista. */
4    public Iterable<Posicion<E>> positions();
5  }
```

Listado 5.11: Agregar métodos de `Iterable` a la interfaz `ListaPosicion`. label

```
1  public Iterable<Posicion<E>> positions() {  // crea una lista de posiciones
2    ListaPosicion<Posicion<E>> P = new ListaNodoPosicion<Posicion<E>>();
3    if (!isEmpty()) {
4      Posicion<E> p = first();
5      while (true) {
6        P.addLast(p); // agregar posición p como el últ. elemento de la lista P
7        if (p == last())
8          break;
9        p = next(p);
10     }
11   }
12   return P; // regresar P como el objeto iterable
13 }
```

Listado 5.12: El método `positions()` de la clase `ListaNodoPosicion`.

El iterador regresado por `iterator()` y otros objetos `Iterable` definen un tipo restringido de iterador que permite solo una pasada a los elementos. Sin embargo, iteradores más poderosos pueden también ser definidos, los cuales permiten moverse hacia atrás y hacia adelante sobre un cierto ordenamiento de los elementos.

5.3.4. Iteradores lista en Java

La clase `java.util.LinkedList` no expone un concepto posición a los usuarios en su API. En vez, la forma preferida para acceder y actualizar un objeto `LinkedList` en Java, sin usar índices, es usar un `ListIterator` que es generado por la lista ligada, usando un método `listIterator`. Tal iterador da métodos de recorrido hacia atrás y hacia adelante al igual que métodos locales para actualizar. Este ve su posición actual como estando antes del primer elemento, entre dos elementos, o después del último elemento. Esto es, usa un cursor lista, parecido a como un cursor de pantalla es visto localizado entre dos caracteres en una pantalla. La interfaz `java.util.ListIterator` incluye los siguientes métodos:

`add`(e):agrega el elemento e en la posición actual del iterador.
`hasNext`():verdadero si y solo si hay un elemento después de la posición actual del iterador.
`hasPrevious`():verdadero si y solo si hay un elemento antes de la posición actual del iterador.
`previous`():regresa el elemento e antes de la posición actual y mueve el cursor antes de e, por lo que puede usarse para iterar hacia atrás.
`next`():regresa el elemento e después de la posición actual y mueve el cursor después de e, por lo que puede usarse para iterar hacia adelante.
`nextIndex`():regresa el índice del siguiente elemento.
`previousIndex`():regresa el índice del elemento previo.
`set`(e):reemplaza el último elemento regresado por `next` o `previous` con el elemento indicado.
`remove`():quita de la lista el último elemento que fue regresado por `next` o `previous`.

Es riesgoso usar múltiples iteradores sobre la misma lista mientras se modifica su contenido. Si inserciones, borrados, o reemplazos son ocupados en múltiples "lugares" en una lista, es más seguro usar posiciones para indicar estas localidades. Pero la clase `java.util.LinkedList` no expone sus objetos posición al usuario. Por lo tanto, para evitar los riesgos de modificar una lista que ha creado múltiples iteradores, por llamadas a su método `iterator`, los objetos `java.util.Iterator` tienen una capacidad "fail-fast" que inmediatamente invalida tal iterador si su colección subyacente es modificada inesperadamente. Java permite muchos iteradores lista para estar recorriendo una lista ligada al mismo tiempo, pero si uno de ellos la modifica, usando `add`, `remove` o `set`, entonces todos los otros iteradores se hacen inválidos.

La interfaz `java.util.List` y sus implementaciones

Java proporciona funcionalidad similar a los ADT lista arreglo y lista nodo en la interfaz `java.util.List`, la cual es implementada con un arreglo en `java.util.ArrayList` y con una lista ligado en `java.util.LinkedList`. Hay algunas compensaciones entre estas dos implementaciones. Java usa iteradores para lograr una funcionalidad similar a la del ADT lista nodo derivada de posiciones. La siguiente table muestra los métodos correspondientes entre los ADT lista (arreglo y nodo) y las interfaces `java.util` `List` y `ListIterator`, así como el tiempo de ejecución. Se usa A y L como abreviaturas para `java.util.ArrayList` y `java.util.LinkedList`.

Método del ADT lista	Método java.util.List	Método ListIterator	Notas
`size`()	`size`()		$O(1)$
`isEmpty`()	`isEmpty`()		$O(1)$
`get`(i)	`get`(i)		A es $O(1)$, L es $O(\text{mín}\{i, n-i\})$
`first`()	`listIterator`()		1er elemento es next
`last`()	`listIterator`($size()$)		Último elemento es previous
`prev`(p)		`previous`()	$O(1)$
`next`(p)		`next`()	$O(1)$
`set`(p, e)		`set`(e)	$O(1)$
`set`(i, e)	`set`(i, e)		A es $O(1)$, L es $O(\text{mín}\{i, n-i\})$
`add`(i, e)	`add`(i, e)		$O(n)$
`remove`(i)	`remove`(i)		A es $O(1)$, L es $O(\text{mín}\{i, n-i\})$
`addFirst`(e)	`add`($0, e$)		A es $O(n)$, L es $O(1)$
`addFirst`(e)	`addFirst`(e)		$O(1)$ solo en L
`addLast`(e)	`add`(e)		$O(1)$
`addLast`(e)	`addLast`(e)		$O(1)$ solo en L
`addAfter`(p, e)		`add`(e)	A es $O(1)$, L es $O(1)$
`addBefore`(p, e)		`add`(e)	A es $O(1)$, L es $O(1)$
`remove`(p)		`remove`()	A es $O(1)$, L es $O(1)$

5.4. ADT listas y el marco de colecciones

Se revisa en esta sección los ADT lista general, los cuales usan métodos de los ADT deque, lista arreglo, y lista nodo. Antes de describir estos ADT, se mencionan el contexto en el cual existen.

5.4.1. El marco de colecciones Java

Java proporciona un paquete interfaces y clases para estructuras de datos, las cuales definen el *marco de colecciones Java* (*Java Collections Framework*. Este paquete, `java.util`, incluye versiones de varias de las estructuras de datos comentadas en este texto, algunas de las cuales ya han sido discutidas y otras serán revisadas posteriormente. El paquete `java.util` incluyen las siguientes interfaces, entre otras:

`Collection`:una intefaz general para cualquier estructura de datos que contiene una colección de elementos. Extiende a `java.lang.Iterable` por lo que incluye un método `iterator`(), el cual regresa un iterador de los elementos en la colección.

`Iterator`:una interfaz para el ADT iterador simple.

`List`:una interfaz que extiende a `Collection` para incluir el ADT lista arreglo. También incluye el método `lisIterator`() para regresar un objeto `ListIterator` de esta lista.

`ListIterator`:una interfaz iterador que proporciona recorrido hacia adelante y hacia atrás sobre una lista, al igual que métodos de actualización basados en cursor.

`Map`:una interfaz para mapear llaves a valores.

`Queue`:una interfaz para un ADT cola, pero usando nombres de métodos diferentes. Incluye a los métodos `peek`() (equivalente a `front`()), `offer`(e) (equivalente a `enqueue`(e)), y `poll`() (equivalente a `dequeue`()).

`Set`:una interfaz que extiende a `Collections` para conjuntos.

El marco de colecciones Java también incluye varias clases que implementan varias combinaciones de las interfaces anteriores. Estas clases se irán revisando de acuerdo al material que se esté discutiendo, en vez de revisarlas todas ellas de una vez. Se hace notar que cualquier clase implementando la interfaz `java.util.Collection` también implementa la interfaz `java.lang.Iterable`; por lo que incluye al método `iterator`() que puede ser usado en un ciclo `for` avanzado. Además, cualquier clase implementando la interfaz `java.util.List` también incluye al método `listIterator`(). Estas interfaces son útiles para recorrer los elementos de una colección o lista.

5.4.2. La clase `java.util.LinkedList`

Esta clase contiene muchos métodos, incluyendo todos los métodos del ADT deque, sección 4.16, y todos los métodos del ADT lista arreglo, sección 5.1. Además también proporciona funcionalidad similar al ADT lista nodo mediante el uso de lista iterador.

Rendimiento

En la documentación se indica que la clase es implementada con una lista doblemente enlazada. Por lo tanto, todos los métodos de actualización del iterador de lista asociado corren en tiempo $O(1)$, como sucede con los métodos del ADT deque, ya que actualizar o pedir se hace en los extremos de la lista. Pero los métodos del ADT lista arreglo están incluidos en `java.util.LinkedList`, los cuales, en general, no se adaptan bien a una implementación de una lista doblemente enlazada.

Como una lista enlazada no permite acceso indexado a sus elementos, realizar la operación `get`(i), para regresar el elemento en un índice i dado, requiere

que se hagan "saltos" de uno de los extremos de la lista, incrementando o decrementando, hasta que se ubique el nodo que guarda el elemento con índice i. Como una leve optimización, se puede iniciar saltando del extremo más cercano de la lista, logrando así el siguiente tiempo de ejecución

$$O(\text{mín}(i+1, n-1))$$

donde n es el número de elementos en la lista. El peor caso de este tipo de búsquedas ocurre cuando

$$r = \lfloor n/2 \rfloor$$

Por lo tanto, el tiempo de ejecución todavía es $O(n)$.

Las operaciones `add`(i, e) y `remove`(i) también deben hacer saltos para ubicar el nodo que guarda el elemento con índice i, y entonces insertar o borrar un nodo. Los tiempos de ejecución de estas implementaciones son también

$$O(\text{mín}(i+1, n-1))$$

lo cual es $O(n)$. Una ventaja de esta aproximación es que, si $i = 0$ o $i = n-1$, como en el caso de la adaptación del ADT lista arreglo para el ADT deque dado en la sección 5.1.2, entonces estos métodos se ejecutan en tiempo $O(1)$. Pero, en general, usar métodos lista arreglo con un objeto `java.util.LinkedList` es ineficiente.

5.4.3. Secuencias

Una *secuencia* es un ADT que soporta todos los métodos del ADT deque (sección 4.16), el ADT lista arreglo (sección 5.1, y el ADT lista nodo (sección 5.2). Entonces, este proporciona acceso explícito a los elementos en la lista ya sea por sus índices o por sus posiciones. Por otra parte, como se proporciona capacidad de acceso dual, se incluye en el ADT secuencia, los siguientes dos métodos "puente" que proporcionan conexión entre índices y posiciones:

`atIndex`(i):regresar la posición del elemento con índice i; una condición de error ocurre si $i < 0$ o $i > size() - 1$.
`indexOf`(p):regresar el índice del elemento en la posición p.

Herencia múltiple en el ADT secuencia

La definición del ADT secuencia incluye todos los métodos de tres ADT diferentes siendo un ejemplo de *herencia múltiple*. Esto es, el ADT secuencia hereda métodos de tres "super" tipos de datos abstractos. En otras palabras, sus métodos incluyen la unión de los métodos de estos super ADT. Ver el listado 5.13 para una especificación Java del ADT como una interfaz Java.

```
1 /**
2  * Una interfaz para una secuecia, una estructura de datos que
3  * soporta todas las operaciones de un deque, lista indizada y
4  * lista posición.
```

```
5  */
6  public interface Secuencia<E>
7      extends Deque<E>, ListaIndice<E>, ListaPosicion<E> {
8    /** Regresar la posición conteniendo el elemento en el índice dado. */
9    public Posicion<E> atIndex(int r) throws BoundaryViolationException;
10   /** Regresar el índice del elemento guardado en la posición dada. */
11   public int indexOf(Posicion<E> p) throws InvalidPositionException;
12 }
```

Listado 5.13: La interfaz `Secuencia` definida por herencia múltiple por las interfaces `Deque`, `ListaIndice`, y `ListaPosicion` y agregando los dos métodos puente.

Implementando una secuencia con un arreglo

Si se implementa el ADT secuencia S con una lista doblemente enlazada, se obtiene un rendimiento similar a la clase `java.util.LinkedList`. Supóngase que se quiere implementar la secuencia S guardando cada elemento e de S en una celda $A[i]$ de un arreglo A. Se puede definir un objeto posición p que guarde un índice i y una referencia a un arreglo A, como variables de instancia en este caso. Se puede entonces implementar el método `elemento`(p) regresando $A[i]$. Sin embargo, una desventaja es que las celdas en A no tendrían forma de referirse a su correspondiente posición. Así, después de realizar la operación `addFirst`, no se tiene forma de informar las posiciones existentes en S tal que los índices se incrementaron por 1 (la posiciones en una secuencia están definidas relativamente a las posiciones vecinas, no a los índices). Por lo tanto, si se implementa una secuencia con un arreglo, se necesita una aproximación diferente.

Considerar una solución alterna en la cual, en vez de guarda los elementos de S en el arreglo A, se guada un nuevo tipo de objeto posición en cada celda de A, y se guardan elementos en posiciones. El nuevo objeto posición p guarda el índice i y el elemento asociado con p.

Con esta estructura de datos, ilustrada en la figura 5.6, se puede buscar a través del arreglo para actualizar la variable índice i para cada posición cuyo índice cambie debido a una inserción o borrado.

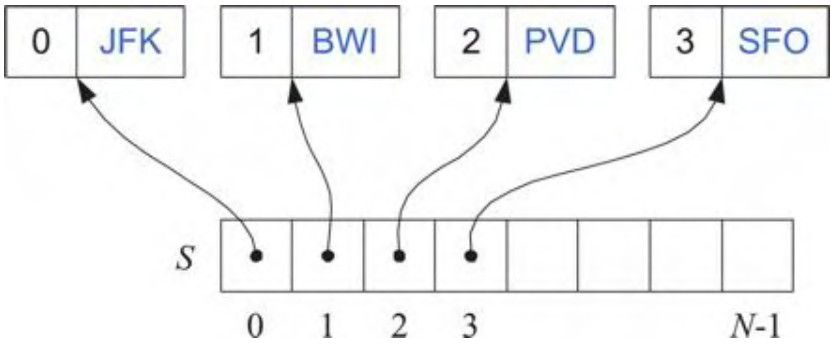

Figura 5.6: Una implementación basada en un arreglo de la secuencia ADT.

Compromiso eficiencia con una secuencia basada en arreglo

En esta implementación usando un arreglo de una secuencia, los métodos `addFirst`, `addBefore`, `addAfter`, y `remove` toman tiempo $O(n)$, por que se tienen que desplazar objetos posición para hacer espacio para la nueva posición o para llenar el espacio creado por la remoción de una vieja posición (como en los métodos insertar y remover basados en índice). Todos los otros métodos basados en la posición toman tiempo $O(1)$.

5.5. Heurística mover al frente

Supóngase que se desea mantener una colección de elementos además de saber el número de veces que cada elemento es accedido. Conocer el número de accesos permite, por ejemplo, conocer cuales son los primeros diez elementos más populares. Ejemplos de tales escenarios incluyen los navegadores Web que mantienen un registro de las direcciones Web más populares visitadas por un usuario o programa álbum de fotos que mantiene una lista de las imágenes más populares que un usuario ve. Adicionalmente, una lista de favoritos podría ser usada en una interfaz gráfica para conocer las acciones más populares de un menú desplegable, para después presentar al usuario un menú condensando conteniendo sólo las opciones más populares.

Se considera en esta sección como se puede implementar el ADT *lista de favoritos*, el cual soporta los métodos `size()` e `isEmpty()` además de los siguientes:

`access`(e): accede el elemento e, incrementando su cuenta de acceso, y agregándolo a la lista de favoritos si todavía no estaba presente.

`remove`(e): quitar el elemento e de la lista de favoritos, dado que este se encuetra allá.

`top`(): regresar un colección iterable de los k elementos más accedidos.

5.5.1. Implementación con una lista ordenada y una clase anidada

La primera implementación de una lista de favoritos que se considera (listado 5.14) es construyendo una clase, `ListaFavoritos`, guardando referencias a los objectos accesados en una lista ligada ordenada de forma decreciente, por la cantidad de accesos. La clase usa una característica de Java que permite definir una clase anidada y relacionada dentro de la definición de una clase. Tal *clase anidada* deberá ser declarada `static`, para indicar que esta definición está relacionada a la clase que la contiene, y no a una instancia especifica de esa clase. Mediante clases anidadas se pueden definir clases de "ayuda" o "soporte" que pueden ser protegidas de uso externo.

La clase anidada, `Entrada`, guarda para cada elemento e en la lista, un par (c, v), donde c es la cuenta de accesos para e y v es una referencia *valor* al propio

elemento e. Cada vez que un elemento es accedido, se busca este en la liga ligada (agregándolo si todavía no está) e incrementando su cuenta de acceso. Remover un elemento requiere encontralo y quitarlo de la lista ligado. Devolver los k elementos más accedidos involucra copiar los valores de entrada en un lista de salida de acuerdo a su orden en lista ligada interna.

```
1  /** Lista de elementos favoritos, con sus cuentas de acceso. */
2  public class ListaFavoritos<E> {
3    protected ListaPosicion<Entrada<E>> listaF; // lista de entradas
4    /** Constructor; tiempo O(1) */
5    public ListaFavoritos() {
6      listaF = new ListaNodoPosicion<Entrada<E>>();
7    }
8    /** Regresa el ùnmero de elementos en la lista; tiempo O(1). */
9    public int size() { return listaF.size(); }
10   /** Prueba si la lista está vacía; tiempo O(1) */
11   public boolean isEmpty() { return listaF.isEmpty(); }
12   /** Quitar un elemento dado, dado que esta en la lista; tiempo O(n) */
13   public void remove(E obj) {
14     Posicion<Entrada<E>> p = encontrar(obj); // búsqueda de obj
15     if( p!=null )
16       listaF.remove(p); // quitar la entrada
17   }
18   /** Incrementar la cuenta de accesos para el elemento dado y lo inserta
19    * si todavía no está presente; tiempo O(n) */
20   public void access(E obj) {
21     Posicion<Entrada<E>> p = encontrar(obj); // búsqueda de obj
22     if( p!=null )
23       p.elemento().incrementarCuenta(); // incrementar cuenta de acceso
24     else {
25       listaF.addLast(new Entrada<E>(obj)); // agregar la entrada al final
26       p = listaF.last();
27     }
28     moverArriba(p);
29   }
30   /** Encontar la posición de un elemento dado, o regresar null; tiempo O(n) */
31   protected Posicion<Entrada<E>> encontrar(E obj) {
32     for( Posicion<Entrada<E>> p: listaF.positions())
33       if( valor(p).equals(obj))
34         return p; // encontrado en posición p
35     return null;    // no encontrado
36   }
37   protected void moverArriba(Posicion<Entrada<E>> cur) {
38     Entrada<E> e = cur.elemento();
39     int c = cuenta(cur);
40     while( cur!=listaF.first() ) {
41       Posicion<Entrada<E>> prev = listaF.prev(cur); // posición previa
42       if( c<=cuenta(prev) ) break; // entrada en su posición correcta
43       listaF.set(cur, prev.elemento()); // descender la entrada previa
44       cur = prev;
45     }
46     listaF.set(cur, e); // guardar la entrada en su posición final
47   }
48   /** Regresa los k elementos más accedidos, para un k; tiempo O(k) */
49   public Iterable<E> top(int k) {
50     if( k<0 || k>size() )
51       throw new IllegalArgumentException("Argumento inválido");
52     ListaPosicion<E> T = new ListaNodoPosicion<E>(); // lista top-k
53     int i=0; // contador de las entradas agregadas a la lista
54     for( Entrada<E> e: listaF ) {
55       if (i++ >= k)
56         break;  // todas las k entradas han sido agregadas
57       T.addLast( e.valor() ); // agregar una entrada a la lista
58     }
59     return T;
60   }
```

```
61    /** Representación String de lista de favoritos */
62    public String toString() { return listaF.toString(); }
63    /** Método auxiliar que extrae el valor de la entrada en una posición */
64    protected E valor(Posicion<Entrada<E>> p) { return p.elemento().valor(); }
65    /** Método auxiliar que extrae el contador de la entrada en una pos. */
66    protected int cuenta(Posicion<Entrada<E>> p) { return p.elemento().cuenta(); }
67
68    /** Clase interna para guardar elementos y sus cuentas de acceso. */
69    protected static class Entrada<E> {
70      private E valor; // elemento
71      private int cuenta; // cuenta de accesos
72      /** Constructor */
73      Entrada(E v) {
74        cuenta = 1;
75        valor = v;
76      }
77      /** Regresar el elemento */
78      public E valor() { return valor; }
79      public int cuenta() { return cuenta; }
80      public int incrementarCuenta() { return ++cuenta; }
81      public String toString() { return "["+cuenta+","+valor+"]"; }
82    }
83  } // Fin de la clase ListaFavoritos
```

Listado 5.14: La clase `ListaFavoritos` que incluye una clase anidada, `Entrada`, la cual representa los elementos y su contador de accesos

5.5.2. Lista con heurística mover al frente

La implementación previa de la lista de favoritos hace el método `access`(e) en tiempo proporcional al índice de e en la lista de favoritos, es decir, si e es el k-ésimo elemento más popular en la lista, entonces accederlo toma tiempo $O(k)$. En varias secuencias de acceso de la vida real, es común, que una vez que el elemento es accedido, es muy probable que sea accedido otra vez en el futuro cercano. Se dice que tales escenarios poseen *localidad de referencia*.

Una *heurística*, o regla de oro, que intenta tomar ventaja de la localidad de referencia que está presente en una secuencia de acceso es la *heurística mover al frente*. Para aplicar esta huerística, cada vez que se accede un elemento se mueve al frente de la lista. La esperanza es que este elemento sea accedido otra vez en el futuro cercano. Por ejemplo, considerar un escenario en el cual se tienen n elementos donde cada elemento es accedido n veces, en el siguiente orden: se accede el primer elemento n veces, luego el segundo n veces, y así hasta que el n-ésimo elemento se accede n veces.

Si se guardan los elementos ordenados por la cantidad de accesos, e insertando cada elemento la primera vez que es accedido, entonces:

- cada acceso al elemento 1 se ejecuta en tiempo $O(1)$;
- cada acceso al elemento 2 se ejecuta en tiempo $O(2)$;
- ...
- cada acceso al elemento n se ejecuta en tiempo $O(n)$;

Así, el tiempo totar para realizar las series de accesos es proporcional a

$$n + 2n + 3n + \ldots + nn = n(1 + 2 + 3 + \ldots + n) = n(n + 1)/2$$

lo cual es $O(n^3)$.

Por otra parte, si se emplea la heurística mover al frente, insertando cada elemento la primera vez, entonces

- cada acceso al elemento 1 se ejecuta en tiempo $O(1)$;
- cada acceso al elemento 2 se ejecuta en tiempo $O(1)$;
- ...
- cada acceso al elemento n se ejecuta en tiempo $O(1)$;

Por lo que el tiempo de ejecución para realizar todos los accesos en este caso es $O(n^2)$. Así, la implementación mover al frente tiene tiempos de acceso más rápidos para este escenario. Sin embargo este beneficio vien a un costo.

Implementación de la heurística mover al frente

En el listado 5.15 se da una implementación de una lista de favoritos usando la heurística mover al frente, definiendo una nueva clase `ListaFavoritosMF`, la cual extiende a la clase `ListaFavoritos` y entonces se anulan las definiciones de los métodos `moverArriba` y `top`. El método `moverArriba` en este caso quita el elemento accedido de su posición frente en la lista ligada y la inserta de regreso en la lista en el frente. El método `top`, por otra parte, es más complicado.

```
1  /** Lista de elementos favoritos con la heurística mover al frente */
2  public class ListaFavoritosMF<E> extends ListaFavoritos<E>{
3    /** Constructor por defecto */
4    public ListaFavoritosMF() { }
5    /** Mueve una entrada a la primera posición: tiempo O(1) */
6    protected void moverArriba(Posicion<Entrada<E>> pos) {
7      listaF.addFirst(listaF.remove(pos));
8    }
9    /** Regresa los k elementos más accedidos, para un k; tiempo O(k) */
10   public Iterable<E> top(int k) {
11     if( k<0 || k>size() )
12       throw new IllegalArgumentException("Argumento inválido");
13     ListaPosicion<E> T = new ListaNodoPosicion<E>(); // lista top-k
14     if( !isEmpty() ){
15       // copiar las entradas en una lista temporal C
16       ListaPosicion<Entrada<E>> C = new ListaNodoPosicion<Entrada<E>>();
17       for( Entrada<E> e: listaF )
18         C.addLast(e);
19       // encontrar los top k elementos, uno a la vez
20       for( int i=0; i<k; i++) {
21         Posicion<Entrada<E>> posMax=null; // posición de la entrada sup.
22         int cuentaMax=-1; // cuenta de accesos de la entrada top
23         for(Posicion<Entrada<E>> p: C.positions()) {
24           // examinar todas las entradas de C
25           int c=cuenta( p );
26           if( c>cuentaMax) { // encontró una entrada con una cuenta mayor
27             cuentaMax = c;
28             posMax = p;
29           }
```

```
30          }
31          T.addLast( valor(posMax) ); // agregar entrada top a T
32          C.remove(posMax); // quitar la posición que ya fue agregada
33        }
34      }
35      return T;
36    }
37  } // Fin de la clase ListaFavoritosMF
```

Listado 5.15: La clase `ListaFavoritosMF` implementa la heurística mover al frente. Esta clase extiende a la clase `ListaFavoritos` y anula los métodos `moverArriba` y `top`.

El listado 5.16 muestra una aplicación que crea dos listas de favoritos, una emplea la clase `ListaFavoritos` y la otra la clase `ListaFavoritosMF`. Las listas de favoritos son construidas usando un arreglo de direcciones Web, y después se genera 20 números pseudoaleatorio que están en el rango de 0 al tamaño del arreglo menos uno. Cada vez que se marca un acceso se muestra el estado de las listas. Posteriormente se obtiene, de cada lista, el *top* del tamaño de la lista. Usando el sitio más popular de una de las listas se abre en una ventana.

```
1  import java.io.*;
2  import javax.swing.*;
3  import java.awt.*;
4  import java.net.*;
5  import java.util.Random;
6  /** Programa ejemplo para las clases ListaFavoritos y ListaFavoritosMF */
7  public class ProbadorFavoritos {
8    public static void main(String[] args) {
9      String[] arregloURL = {"http://google.com","http://mit.edu",
10       "http://bing.com","http://yahoo.com","http://unam.mx"};
11     ListaFavoritos<String> L1 = new ListaFavoritos<String>();
12     ListaFavoritosMF<String> L2 = new ListaFavoritosMF<String>();
13     int n = 20; // cantidad de operaciones de acceso
14     // Escenario de simulación: acceder n veces un URL aleatorio
15     Random rand = new Random();
16     for(int k=0; k<n; k++) {
17       System.out.println("_____________________________________________");
18       int i = rand.nextInt(arregloURL.length); // índice aleatorio
19       String url = arregloURL[i];
20       System.out.println("Accediendo: "+url);
21       L1.access(url);
22       System.out.println("L1 = " + L1);
23       L2.access(url);
24       System.out.println("L2 = " + L2);
25     }
26     int t = L1.size()/2;
27     System.out.println("---------------------------------------------");
28     System.out.println("Top " + t + " en L1 = " + L1.top(t));
29     System.out.println("Top " + t + " en L2 = " + L2.top(t));
30     // Mostrar una ventana navegador del URL más popular de L1
31     try {
32       String popular = L1.top(1).iterator().next(); // el más popular
33       JEditorPane jep = new JEditorPane(popular);
34       jep.setEditable(false);
35       JFrame frame = new JFrame("URL más popular en L1: "+popular);
36       frame.getContentPane().add(new JScrollPane(jep),BorderLayout.CENTER);
37       frame.setSize(640,480);
38       frame.setVisible(true);
39       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40     } catch( IOException e) { /* ignorar excepciones I/O */ }
41   }
42 }
```

Listado 5.16: Se emplea en la clase `ProbadorFavoritos` las clases `ListaFavoritos` y `ListaFavoritosMF` para contar la simulación de accesos a un conjunto de páginas Web. Se muestra también la página más popular.

El compromiso con la heurística mover al frente

La lista de favoritos no será mantenida en orden por la cantidad de accesos, cuando se pide encontrar los k elementos más accedidos, se necesita buscarlos. Se puede implementar el método `top`(k) como sigue:

1. Se copian las entradas de la lista de favoritos en otra lista, C, y se crea una lista vacía, T.
2. La lista C se explora k veces. En cada ocasión, se encuentra una entrada de C con la cuenta de acceso más grande, quitando esta entrada de C, e insertando su valor al final de T.
3. Se regresa la lista T.

La implementación del método `top` toma tiempo $O(kn)$. Así, cuando k es una constante, el método se ejecuta en tiempo $O(n)$. Esto ocurre si que quiere obtener la lista del "*top-ten*". Sin embargo, si k es proporcional a n, entonces la ejecución toma tiempo $O(n^2)$ y sucede cuando se quiere la lista del "top 25 %".

Capítulo 6

Árboles

6.1. Árboles generales

En este capítulo, se discute una de las estructuras de datos no lineal más importante en computación—*árboles*. Las estructuras árbol son además un progreso en la organización de datos, ya que permiten implementar un conjunto de algoritmos más rápidos que cuando se usan estructuras lineales de datos, como una lista. Los árboles también dan una organización natural para datos, y por lo tanto, se han convertido en estructuras indispensables en sistemas de archivos, interfaces gráficas de usuario, bases de datos, sitios Web, y otros sistemas de cómputo.

Cuando se dice que los árboles "no son lineales", se hace referencia a una relación organizacional que es más rica que las relaciones simples "antes" y "después" entre objetos en secuencias. Las relaciones en un árbol son *jerárquicas*, con algunos objetos estando "encima" y algunos otros "abajo". Actualmente, la terminología principal para estructuras de dato árbol viene de los árboles genealógicos, con algunos términos "padre", "hijo", "ancestro", y "descendiente" siendo las palabras comunes más usadas para describir relaciones.

6.1.1. Definiciones de árboles y propiedades

Un *árbol* es un tipo de dato abstracto que guarda elementos jerárquicamente. Con la excepción del elemento cima, cada elemento en un árbol tiene un elemento *padre* y cero o mas elementos *hijos*. Un árbol es visualizado colocando elementos dentro de óvalos o rectángulos, y dibujando las conexiones entre padres e hijos con líneas rectas, ver figura 6.1. Se llama al elemento cima la *raíz* del árbol, pero este es dibujado como el elemento más alto, con los otros elementos estando conectados abajo, justo lo opuesto de un árbol vivo.

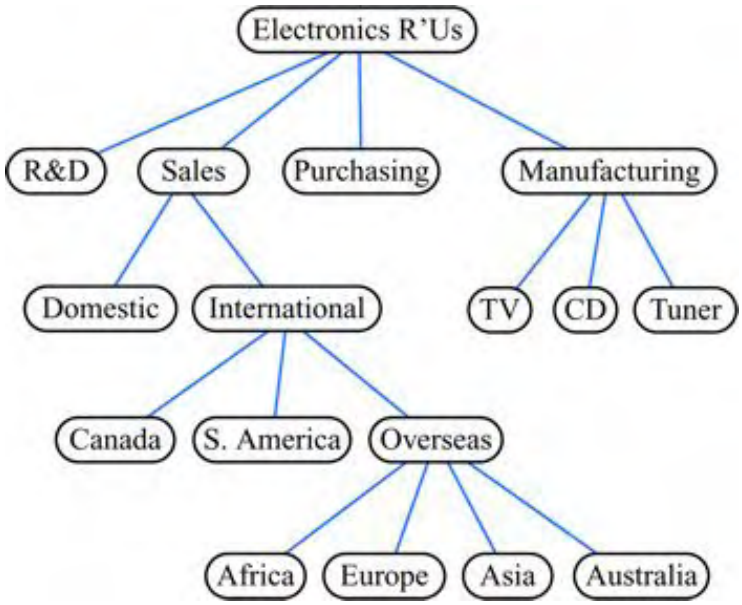

Figura 6.1: Un árbol con 17 nodos que representan la organización de una corporación.

6.1.2. Definición formal de árbol

Se define un *árbol* T como un conjunto de *nodos* guardando elementos tal que los nodos tienen una relación *padre-hijo*, que satisfacen las siguientes propiedades:

- Si T no está vacío, este tiene un nodo especial, llamado la *raíz* (root) de T, que no tiene padre.
- Cada nodo v de T diferente de la raíz tiene un nodo *padre* único w; cada nodo con padre w es un *hijo* de w.

De acuerdo a lo anterior, un árbol puede estar vacío; es decir, que este no tiene ningún nodo. Esta convención también permite definir un árbol recursivamente, tal que un árbol T está vacío o consiste de un nodo r, llamado la raíz de T, y un conjunto de árboles, posiblemente vacío, cuyas raíces son los hijos de r.

Dos nodos que son hijos del mismo padre son *hermanos*. Un nodo v es *externo* si v no tiene hijos. Un nodo v es interno si este tiene uno o más hijos. Los nodos externos son también conocidos como *hojas*.

Ejemplo 6.1. *En la mayoría de los sistemas operativos, los archivos son organizados jerárquicamente en directorios anidados, también conocidos como carpetas, los cuales son presentados al usuario en la forma de un árbol, ver figura 6.2. Más específicamente, los nodos internos del árbol están asociados con directorios y los nodos externos están asociados con archivos regulares. En el sistema operativo*

UNIX. la raíz del árbol es llamada el "directorio raíz", y es representada por el símbolo "/".

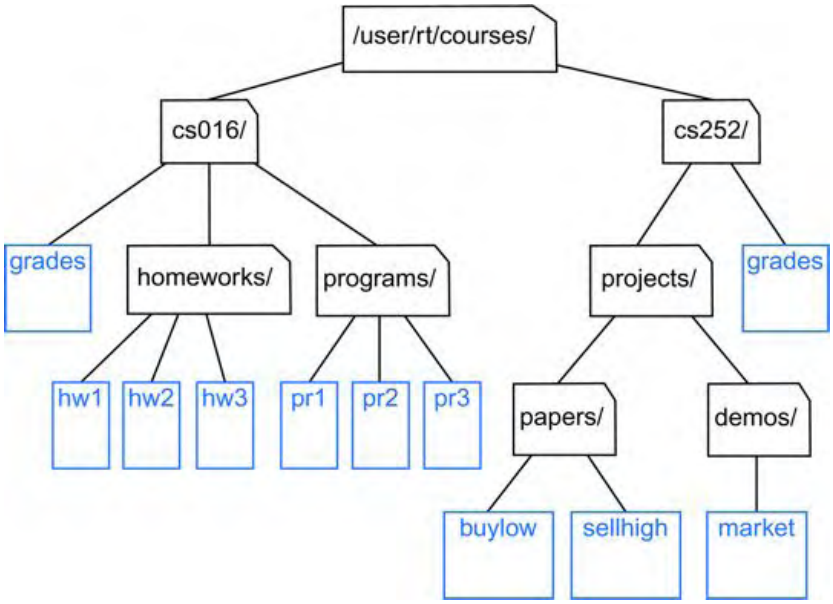

Figura 6.2: Representación de árbol de una porción de un sistema de archivos.

Un nodo u es un *ancestro* de un nodo v si $u = v$ o u es un ancestro del padre de v. A la inversa, se dice que un nodo v es un *descendiente* de un nodo u si u es un ancestro de v. En la figura 6.2 `cs252` es un ancestro de `papers`, y `pr3` es un descendiente de `cs016/`. El *subárbol* enraízado de T es el árbol consistente de todos los descendientes de v en T, incluyendo al propio v. En la figura 6.2, el subárbol enraizado en `cs016/` consiste de los nodos `cs016/`, `grades`, `homeworks/`, `programs/`, `hw1`, `hw2`, `hw3`, `pr1`, `pr2` y `pr3`.

Aristas y caminos en árboles

Una *arista* del árbol T es un par de nodos (u, v) tal que u es el padre de v, o viceversa. Un *camino* de T es una secuencia de nodos tal que dos nodos consecutivos cualesquiera en la secuencia forman una arista. Por ejemplo, el árbol de la figura 6.2 contiene el camino `cs016/programs/pr2`.

Ejemplo 6.2. *La relación de herencia entre clases en un programa Java forman un árbol. La raíz, java.lang.Object, es un ancestro de todas las otras clases. Cada clase, C, es una descendiente de esta raíz y es la raíz de un subárbol de las clases que extienden C. Así, hay un camino de C a la raíz, java.lang.Object, en este árbol de herencia.*

Árboles ordenados

Un árbol es *ordenado* si hay un ordenamiento lineal definido para los hijos de cada nodo; esto es, se puede identificar los hijos de un nodo como siendo el primero, segundo, tercero, etc. Tal ordenamiento es visualizado arreglando los hermanos de izquierda a derecha, de acuerdo a su ordenamiento. Los árboles ordenados indican el orden lineal entre hermanos listándolos en el orden correcto.

Ejemplo 6.3. *Los componentes de un documento estructurado, tal como un libro, están jerárquicamente organizados como un árbol cuyos nodos internos son partes, capítulos, y secciones, y cuyos nodos externos son párrafos, tablas, figuras, etcétera, ver figura 6.3. La raíz del árbol corresponde al propio libro. Se podría considerar expandir más el árbol para mostrar párrafos consistentes de sentencias, sentencias consistentes de palabras, y palabras consistentes de caracteres. Tal árbol es un ejemplo de un árbol ordenado, porque hay un ordenamiento bien definido entre los hijos de cada nodo.*

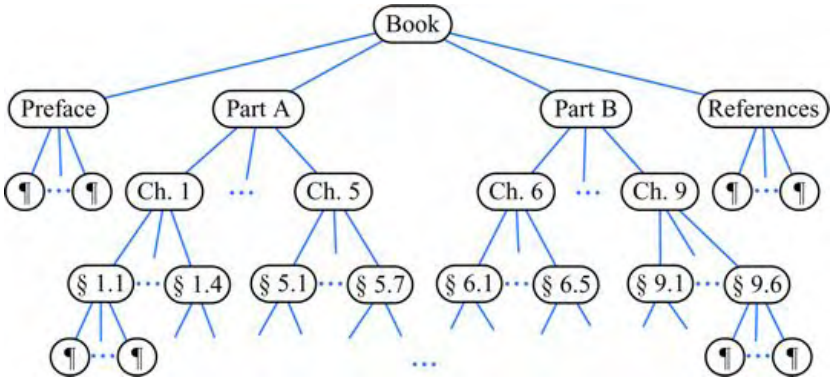

Figura 6.3: Un árbol ordenado asociado con un libro.

6.1.3. El tipo de dato abstracto árbol

El ADT árbol guarda elementos en posiciones, las cuales, al igual que las posiciones en una lista, están definidas relativamente a posiciones vecinales. Las *posiciones* en un árbol son sus *nodos* y las posiciones vecinales satisfacen las relaciones padre-hijo que definen un árbol válido. Por lo tanto, se usan los términos "posición" y "nodo" indistintamente para árboles. Al igual que con una lista posición, un objeto posición para un árbol soporta el método:

`elemento()`:regresa el objeto guardado en esta posición.

Sin embargo, el poder real de posiciones nodo en un árbol, viene de los *métodos accesores* del ADT árbol que aceptan y regresan posiciones, tales como las siguientes:

`root()`:regresa la raíz del árbol; un error ocurre si el árbol está vacío.
`parent(`v`)`:regresa el padre de v; un error ocurre si v es la raíz.
`children(`v`)`:regresa una colección iterable que lleva los hijos del nodo v.

Si un árbol T esta ordenado, entonces la colección iterable, `children(`v`)`, guarda los hijos de v en orden. Si v es un nodo externo, entonces `children(`v`)` está vacío.

Además de los métodos anteriores fundamentales accesores, también se incluyen los siguientes *métodos de consulta*:

`isInternal(`v`)`:indica si el nodo v es interno.
`isExternal(`v`)`:indica si el nodo v es externo.
`isRoot(`v`)`:indica si el nodo v es la raíz.

Estos métodos hacen la programación con árboles más fácil y más legible, ya que pueden ser usados en las condiciones de las sentencias **if** y de los ciclos **while**, en vez de usar un condicional no intuitivo.

Hay también un número de *métodos genéricos* que un árbol probablemente podría soportar y que no están necesariamente relacionados con la estructura del árbol, incluyendo los siguientes:

`size()`:regresa el número de nodos en el árbol.
`isEmpty()`:valida si el árbol tiene algún nodo o no.
`iterator()`:regresa un iterador de todos los elementos guardados en nodos del árbol.
`positions()`:regresa una colección iterable de todos los nodos del árbol.
`replace(`v, e`)`:regresa el elemento del nodo v antes de cambiarlo por e.

Cualquier método que tome una posición como un argumento podría generar una condición de error si esa posición es inválida. No se definen métodos de actualización especializados para árboles. En su lugar, se describen métodos de actualización diferentes en conjunción con aplicaciones específicas de árboles.

6.1.4. Implementar un árbol

La interfaz Java mostrada en el listado 6.1 representa el ADT árbol. Las condiciones de error son manejadas como sigue: cada método puede tomar una posición como un argumento, estos métodos podrían lanzar `InvalidPositionException` para indicar que la posición es inválida. El método `parent` lanza `BoundaryViolationException` si este es llamado sobre un árbol vacío. En el caso de que se intente obtener la raíz de un árbol T vacío se lanza `EmpthTreeException`.

```
1 import java.util.Iterator;
2 /**
3  * Una interfaz para un árbol donde los nodos pueden tener
4  * una cantidad arbitraria de hijos.
5  */
6 public interface Tree<E> {
7   /** Regresa el número de nodos en el árbol. */
8   public int size();
9   /** Valida si el árbol está vacío. */
```

```
10    public boolean isEmpty();
11    /** Regresa un iterador de los elementos guardados en el árbol. */
12    public Iterator<E> iterator();
13    /** Regresa una colección iterable de los nodos. */
14    public Iterable<Posicion<E>> positions();
15    /** Reemplaza el elemento guardado en un nodo dado. */
16    public E replace(Posicion<E> v, E e)
17      throws InvalidPositionException;
18    /** Regresa la raíz del árbol. */
19    public Posicion<E> root() throws EmptyTreeException;
20    /** Regresa el padre de un nodo dado. */
21    public Posicion<E> parent(Posicion<E> v)
22      throws InvalidPositionException, BoundaryViolationException;
23    /** Regresa una colección iterable de los hijos de un nodo dado. */
24    public Iterable<Posicion<E>> children(Posicion<E> v)
25      throws InvalidPositionException;
26    /** Valida si un nodo dado es interno. */
27    public boolean isInternal(Posicion<E> v)
28      throws InvalidPositionException;
29    /** Valida si un nodo dado es externo. */
30    public boolean isExternal(Posicion<E> v)
31      throws InvalidPositionException;
32    /** Valida si un nodo dado es la raíz del árbol. */
33    public boolean isRoot(Posicion<E> v)
34      throws InvalidPositionException;
35  }
```

Listado 6.1: La interfaz Java `Tree` representando el ADT árbol. Métodos adicionales de actualización podrían ser agregados dependiendo de la aplicación.

Una estructura enlazada para árboles generales

Una forma natural para crear un árbol T es usar una *estructura enlazada*, donde se representa cada nodo v de T por un objeto posición, ver figura 6.4 (a), con los siguientes campos: una referencia al elemento guardado en v, un enlace al padre de v, y algún tipo de colección, por ejemplo, una lista o arreglo, para guardar enlaces a los hijos de v. Si v es la raíz de T entonces el campo `parent` de v es `null`. También, se guarda una referencia a la raíz de T y el número de nodos de T en variables internas. Se muestra en la figura 6.4 (b) la estructura esquemáticamente.

La tabla 6.1 resume el rendimiento de la implementación de un árbol general usando una estructura enlazada. Se observa que usando una colección para guardar los hijos de cada nodo v, se puede implementar `children(`v`)` regresando una referencia a esta colección.

6.2. Algoritmos de recorrido para árbol

En esta sección, se presentan algoritmos para hacer recorridos en un árbol accediendo este a través de los métodos del ADT árbol.

6.2.1. Profundidad y altura

Sea v un nodo de un árbol T. La *profundidad* de v es el número de ancestros de v, excluyendo al propio v. Esta definición implica que la profundidad de la

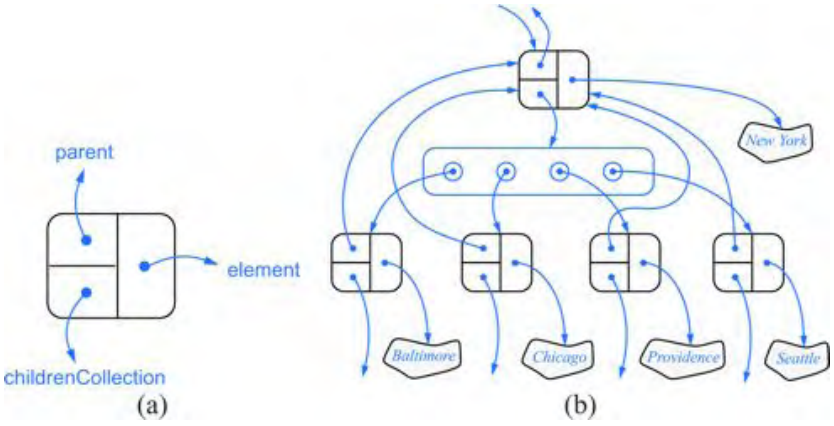

Figura 6.4: La estructura enlazada para un árbol general: (a) el objeto posición asociado con un nodo; (b) la porción de la estructura de dato asociada con un nodo y sus hijos.

Operación	Tiempo
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

Cuadro 6.1: Tiempos de ejecución de los métodos de un árbol general n-nodos con una estructura enlazada. c_v denota el número de hijos de un nodo v. El espacio usado es $O(n)$.

raíz de T es cero.

La profundidad de un nodo v puede también ser definida recursivamente como sigue:

- Si v es la raíz, entonces la profundidad de v es cero.
- De otra forma, la profundidad de v es uno mas la profundidad del padre de v.

Basada en esta definición se presenta un algoritmo recursivo simple, `depth` para computar la profundidad de un nodo v de T. Este método se llama a sí mismo recursivamente en el padre de v, y agrega uno al valor regresado.

Algoritmo depth(T, v):
  Si v es la raíz de T **Entonces**
    Regresar 0
  Si no
    Regresar 1+depth(T, w), donde w es el padre de v en T

Una implementación Java simple de este algoritmo se muestra en el listado 6.2.

```
1  public static <E> int depth (Tree<E> T, Posicion<E> v) {
2     if ( T.isRoot(v) )
3        return 0;
4     else
5        return 1 + depth(T, T.parent(v) );
6  }
```

Listado 6.2: El método `depth` escrito en Java.

El tiempo de ejecución del algoritmo `depth`(T, v) es $O(d_v)$, donde d_v denota la profundidad del nodo v en el árbol T, porque el algoritmo realiza un paso recursivo tiempo constante para cada ancestro de v. Así, el algoritmo `depth`(T, v) se ejecuta en tiempo $O(n)$ en el peor caso, donde n es el número total de nodos de T, ya que un nodo de T podría tener profundidad $n - 1$ en el peor caso. Aunque tal tiempo de ejecución es una función del tamaño de entrada, es más preciso caracterizar el tiempo de ejecución en términos del parámetro d_v, debido a que este parámetro puede ser mucho menor que n.

Altura

La *altura* de un nodo v en un árbol T está también definida recursivamente:

- Si v es un nodo externo, entonces la altura de v es cero.
- De otra forma, la altura de v es uno más la altura máxima de los hijos de v.

La *altura* de un árbol no vacío T es la altura de la raíz de T. Además la altura puede también ser vista como sigue.

Proposición 6.1: *La altura de un árbol no vacío T es igual a la máxima profundidad de un nodo externo de T.*

Se presenta enseguida un algoritmo, `height1`, para hallar la altura de un árbol no vacío T usando la definición anterior y el algoritmo `depth`. El listado 6.3 muestra su implementación en Java.

Algoritmo height1(T):
  $h \leftarrow 0$
  Para cada vértice v en T **Hacer**
    Si v es un nodo externo de T **Entonces**
      $h \leftarrow max(h, \text{depth}(T, v))$
  Regresar h

```
1  public static <E> int height1 (Tree<E> T) {
2     int h = 0;
3     for ( Posicion<E> v : T.positions() )
4        if ( T.isExternal(v) )
5           h = Math.max( h, depth(T,v) );
6     return h;
7  }
```

Listado 6.3: El método `height1` escrito en Java. Se emplea el método `max` de la clase `java.lang.Math`

Desafortunadamente, el algoritmo `height1` no es muy eficiente. Como `height1` llama al algoritmo `depth(`v`)` en cada nodo externo v de T, el tiempo de ejecución de `height1` está dado por $O(n + \sum_v (1 + d_v))$, donde n es el número de nodos de T, d_v es la profundidad del nodo v, y la suma se hace sólo con el conjunto de nodos externos de T. En el peor caso, la suma $\sum_v (1 + d_v)$ es proporcional a n^2. Así, el algoritmo `height1` corre en tiempo $O(n^2)$.

El algoritmo `height2`, mostrado a continuación e implementado en Java en el listado 6.4, computa la altura del árbol T de una forma más eficiente usando la definición recursiva de altura.

Algoritmo height2(T, v):
  Si v es un nodo externo de T **Entonces**
    Regresar 0
  Si no
    $h \leftarrow 0$
    Para cada hijo w de v en T **Hacer**
      $h \leftarrow max(h, \text{height2}(T, w))$
    Regresar 1+h

```
1  public static <E> int height2 (Tree<E> T, Posicion<E> v) {
2     if ( T.isExternal(v) ) return 0;
3     int h = 0;
4     for ( Posicion<E> w : T.children(v) )
5        h = Math.max(h, height2( T, w ));
6     return 1 + h;
7  }
```

Listado 6.4: El método `height2` escrito en Java.

El algoritmo `height2` es más eficiente que `height1`. El algoritmo es recursivo, y, si este es inicialmente llamado con la raíz de T, este eventualmente será llamado con cada nodo de T. Así, se puede determinar el tiempo de ejecución de este método sumando, sobre todos los nodos, la cantidad de tiempo gastado en cada

nodo. Procesar cada nodo en `children(`v`)` toma tiempo $O(c_v)$, donde c_v es el número de hijos de un nodo v. También, el ciclo **while** tiene c_v iteraciones y cada iteración en el ciclo toma tiempo $O(1)$ más el tiempo para la llamada recursiva con un hijo de v. Así, el algoritmo `height2` gasta tiempo $O(1 + c_v)$ en cada nodo v, y su tiempo de ejecución es $O(\sum_v (1 + c_v))$.

Sea un árbol T con n nodos, y c_v denota el número de hijos de un nodo v de T, entonces sumando sobre los vértices en T, $\sum_v c_v = n - 1$, ya que cada nodo de T, con la excepción de la raíz, es un hijo de otro nodo, y así contribuye una unidad a la suma anterior.

Con el resultado anterior, el tiempo de ejecución del algoritmo `height2`, cuando es llamado con la raíz de T, es $O(n)$, donde n es el número de nodos de T.

6.2.2. Recorrido en preorden

Un *recorrido* de un árbol T es una forma sistemática de acceder, "visitar", todos los nodos de T. Se presenta un esquema básico de recorrido para árboles, llamado recorrido en preorden. En la siguiente sección, se revisa otro esquema básico de recorrido, llamado recorrido en postorden.

En un *recorrido en preorden* de un árbol T, la raíz de T es visitada primero y entonces los subárboles enraizados con sus hijos son recorridos recursivamente. Si el árbol está ordenado, entonces los subárboles son recorridos de acuerdo al orden de los hijos. La acción específica asociada con la "visita" de un nodo v depende de la aplicación de este recorrido, y podría involucrar desde incrementar un contador hasta realizar algún cálculo complejo para v. El pseudocódigo para el recorrido en preorden del subárbol enraizado en un nodo v se muestra enseguida.

Algoritmo preorden(T, v):
  Realizar la acción "visita" para el nodo v
  Para cada hijo w de v en T **Hacer**
    preorden(T, w) {recursivamente recorrer el subárbol en w}

El algoritmo de recorrido en preorden es útil para producir un ordenamiento lineal de los nodos de un árbol donde los padres deberán estar antes que sus hijos en el ordenamiento. Tales ordenamientos tienen varias aplicaciones diferentes. Se comenta a continuación un ejemplo simple de tal aplicación en el siguiente ejemplo.

Ejemplo 6.4. *El recorrido en preorden del árbol asociado con un documento examina un documento entero secuencialmente, desde el inicio hasta el fin. Si los nodos externos son quitados antes del recorrido, entonces el recorrido examina la tabla de contenido del documento. Ver figura 6.5.*

El recorrido en preorden es también una forma eficiente para acceder a todos los nodos de un árbol. El análisis es similar al del algoritmo `height2`. En cada nodo v, la parte no recursiva del algoritmo de recorrido en preorden requiere tiempo $O(1 + c_v)$, donde c_v es el número de hijos de v, Así, el tiempo total de ejecución en el recorrido en preorden de T es $O(n)$. También se puede observar que cada nodo es visitado sólo una vez.

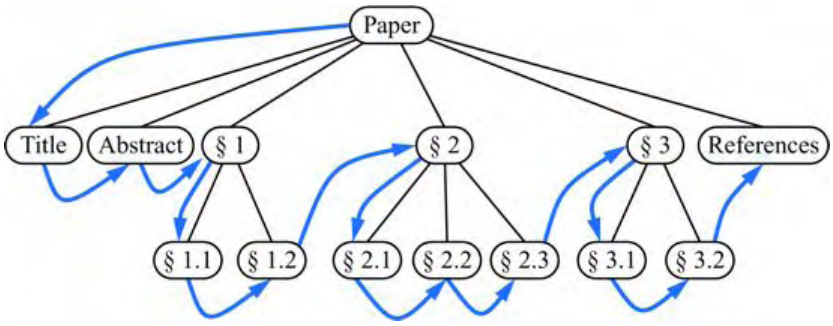

Figura 6.5: Recorrido en preorden de un árbol ordenado, donde los hijos de cada nodo están ordenados de izquierda a derecha.

El algoritmo `toStringPreorden` implementado con Java, listado 6.5, hace una impresión en preorden del subárbol de un nodo v de T, es decir, realiza el recorrido en preorden del subárbol enraizado en v e imprime los elementos guardados en un nodo cuando el nodo es visitado. El método $T.children(v)$ regresa una colección iterable que accede a los hijos de v en orden.

```
1  public static <E> String toStringPreorden(Tree<E> T, Posicion<E> v) {
2     String s = v.elemento().toString(); // acción principal de visita
3     for (Posicion<E> h: T.children(v))
4        s += ", " + toStringPreorden(T, h);
5     return s;
6  }
```

Listado 6.5: Método `toStringPreorden`que hace una impresión en preorden de los elementos en el subárbol del nodo v de T.

Hay una aplicación interesante del algoritmo de recorrido en preorden que produce una representación de cadena de un árbol entero. Suponiendo que para cada elemento e guardado en el árbol T, la llamada e.`toString()` regresa una cadena asociada con e. La *representación parentética de cadena* $P(T)$ del árbol T definida recursivamente como sigue. Si T tiene un sólo nodo v, entonces

$$P(T) = v.\text{elemento}().\text{toString}().$$

De otra forma,

$$P(T) = v.\text{elemento}().\text{toString}() + \text{“(”} + P(T_1) + \text{“,”} + \cdots + \text{“,”} + P(T_k) + \text{“)”},$$

donde v es la raíz de T y $T_1, T_2, \ldots, T_k$ son los subárboles enraizados de los hijos de v, los cuales están dados en orden si T es un árbol ordenado.

La definición de $P(T)$ es recursiva y se usa el operador “+” para indicar la concatenación de cadenas. La representación parentética del árbol de la figura 6.1 se muestra enseguida, donde el sangrado y los espacios han sido agregados para claridad.

```
Electronics R'Us (
   R&D
   Sales (
      Domestic
      International (
         Canada
         S. America
         Overseas ( Africa Europe Asia Australia )
      )
   )
   Purchasing
   Manufacturing ( TV CD Tuner )
)
```

El método Java `representacionParentetica`, listado 6.6, es una variación del método `toStringPreorden`, listado 6.5. Está implementado por la definición anterior para obtener una representaión parentética de cadena de un árbol T. El método `representacionParentetica` hace uso del método `toString` que está definido para objeto Java. Se puede ver a este método como un tipo de método `toString()` para objetos árbol.

```
1  public static <E> String representacionParentetica(
2                              Tree<E> T, Posicion<E> v) {
3     String s = v.elemento().toString(); // acción principal de visita
4     if (T.isInternal(v)) {
5        Boolean primeraVez = true;
6        for (Posicion<E> w : T.children(v))
7           if (primeraVez) {
8              s += " ( "+representacionParentetica(T, w); // 1er hijo
9              primeraVez = false;
10          }
11          else
12             s += ", "+representacionParentetica(T, w); // hijos siguientes
13       s += " )"; // cerrar paréntesis
14    }
15    return s;
16 }
```

Listado 6.6: Método `representacionParentetica` donde se emplea el operador + para concatenar dos cadenas.

6.2.3. Recorrido en postorden

El algoritmo de *recorrido en postorden* puede ser visto como el opuesto al recorrido en preorden, porque este recursivamente recorre los subárboles enraizados en los hijos de la raíz primero, y después visita la raíz. Al igual que en el recorrido en preorden se emplea el algoritmo para resolver un problema particular especializando una acción asociada con la "visita" de un nodo v. Si el árbol está ordenado, se hacen llamadas recursivas para los hijos de un nodo v de acuerdo a su orden indicado. El pseudocódigo es dado a continuación.

Algoritmo postorden(T, v):
  Para cada hijo w de v en T **Hacer**
    postorden(T, w) {recursivamente recorrer el subárbol en w}
  Realizar la acción "visita" para el nodo v

El nombre de recorrido en postorden se debe a que este método de recorrido visitará un nodo v después de que este ha visitado todos los otros nodos en los subárboles enraizados en v. Ver figura 6.6.

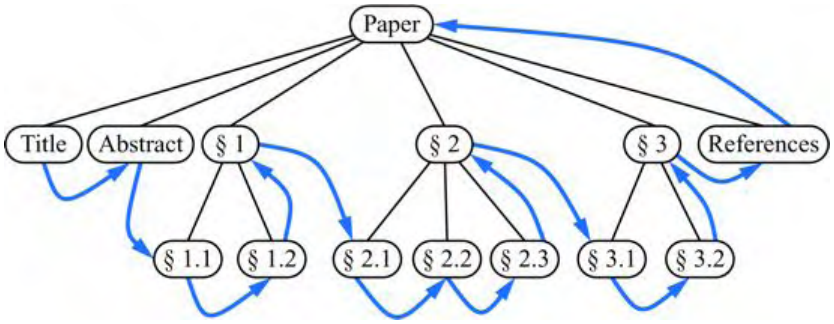

Figura 6.6: Recorrido en postorden del árbol ordenado de la figura 6.5

El análisis del tiempo de ejecución del recorrido en postorden es análogo al del recorrido en preorden. El tiempo total gastado en las porciones no recursivas del algoritmo es proporcional al tiempo gastado en visitar los hijos de cada nodo en el árbol. Entonces el recorrido de un árbol T con n nodos toma tiempo $O(n)$, suponiendo que cada visita tome tiempo $O(1)$, así el recorrido se ejecuta en tiempo lineal.

En el listado 6.7 se muestra el método Java `toStringPostOrden` el cual realiza un recorrido en postorden de un árbol T. Este método muestra el elemento guardado en un nodo cuando este es visitado. Este método también llama implícitamente llama al método `toString` con los elementos, cuando están involucrados en una operación de concatenación de cadenas.

```
1  public static <E> String toStringPostorden(Tree<E> T, Posicion<E> v) {
2     String s = "";
3     for (Posicion<E> h: T.children(v))
4        s += ", " + toStringPreorden(T, h)+" ";
5     s += v.elemento();
6     return s;
7  }
```

Listado 6.7: Método `toStringPostorden` que realiza una impresión en postorden de los elementos en el subárbol del nodo v de T.

El método de recorrido en postorden es útil para resolver problemas donde se desea calcular alguna propiedad para cada nodo v en un árbol, pero para poder calcular esa propiedad sobre v se requiere que ya se haya calculado esa propiedad para los hijos de v. Se muestra una aplicación en el siguiente ejemplo.

Ejemplo 6.5. *Considerar un árbol sistema de archivos T, donde los nodos externos representan archivos y los nodos internos representan directorios. Suponer*

que se quiere calcular el espacio en disco usado en un directorio, ver figura 6.7, lo cual es recursivamente dado por la suma de:

- El tamaño del directorio propio.
- Los tamaños de los archivos en el directorio.
- El espacio usado por los directorios hijos.

El cálculo puede ser hecho en un recorrido en postorden del árbol T. Después de que los subárboles de un nodo interno v han sido recorridos, se calcula el espacio usado por v agregando el tamaño del propio directorio v y los tamaños de los archivos contenidos en v al espacio usado por cada hijo interno de v, el cual fue calculado por los recorridos recursivos en postorden de los hijos de v.

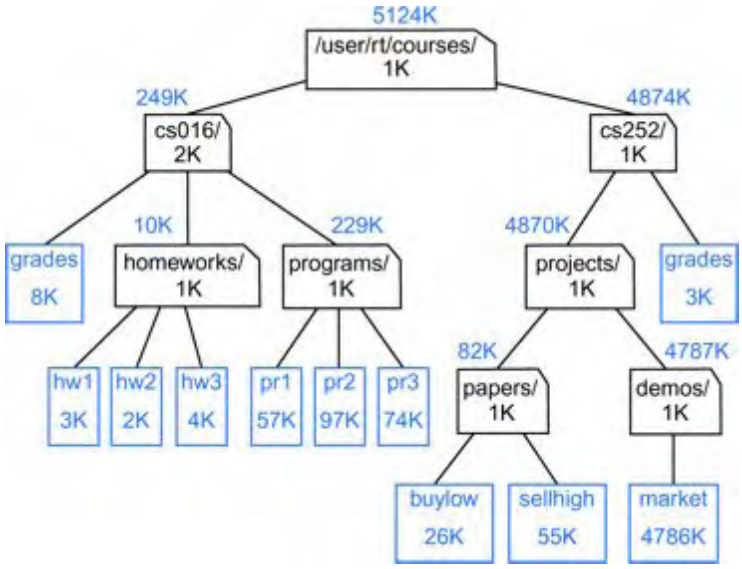

Figura 6.7: El árbol de la figura 6.2 representando el sistema de archivos, mostrando el nombre y el tamaño asociado para cada archivo o directorio dentro de cada nodo, y el espacio en disco usado por el directorio encima de cada nodo interno.

Calcular el espacio en disco

De acuerdo con el ejemplo anterior, el método `espacioDisco`, mostrado en el listado 6.8, realiza un recorrido en postorden de un árbol sistema de archivos T, imprimiendo el nombre y el espacio usado por el directorio asociado con cada nodo interno de T. Cuando se llama desde la raíz de T, `espacioDisco` se ejecuta en tiempo $O(n)$, donde n es el número de nodos de T dados que los métodos auxiliares *nombre* y *tam* toman tiempo $O(1)$.

```
1  public static <E> int espacioDisco(Tree<E> T, Posicion<E> v) {
2     int t = tam(v);  //iniciar con el tamaño del propio nodo
3     for (Posicion<E> h: T.children(v))
4        // agregar espacio calculado recursivamente usado por los hijos
5        t += espacioDisco(T, h);
6     if (T.isInternal(v))
7        // imprimir nombre y espacio en disco usado
8        System.out.print(nombre(v) + ": " + t);
9     return t;
10 }
```

Listado 6.8: El método `espacioDisco` imprime el nombre y el espacio en disco usado por los directorios asociados con cada nodo interno de un árbol sistema de archivos. Los métodos auxiliares *nombre* y *tam* deben ser definidos para regresar el nombre y el tamaño asociado con un nodo.

Otros tipos de recorridos

A pesar de que los recorridos en preorden y postorden son formas comunes de visitar los nodos de un árbol, se pueden tener otros recorridos. Por ejemplo, se podría recorrer un árbol del que se visiten todos los nodos de una profundidad p antes que se visiten los nodos en una profundidad $d + 1$. Este recorrido es conocido "por niveles". Cuando se recorren los nodos del árbol consecutivamente numerándoles conforme se visitan se llama la *numeración por nivel* de los nodos de T.

6.3. Árboles Binarios

Un *árbol binario* es un árbol ordenado con las siguientes propiedades:

1. Cada nodo tiene a lo más dos hijos.
2. Cada nodo hijo está etiquetado ya se como un *hijo izquierdo* o como un *hijo derecho*.
3. Un hijo izquierdo precede a un hijo derecho en el ordenamiento de los hijos de un nodo.

El subárbol enraizado en un hijo izquierdo o un hijo derecho de un nodo interno v es llamado un *subárbol izquierdo* o *subárbol derecho*, respectivamente, de v. Un árbol binario es *propio* si cada nodo tiene cero o dos hijos. Algunos autores también se refieren a tales árboles como árboles binarios *completos*. Así, en un árbol binario propio, cada nodo interno tiene exactamente dos hijos. Un árbol binario que no es propio es *impropio*.

Ejemplo 6.6. *Una clase de árboles binarios se emplean en los contextos donde se desea representar un número de diferentes salidas que pueden resultar de contestar una serie de preguntas si o no. Cada nodo interno está asociado con una pregunta. Iniciando en la raíz, se va con el hijo izquierdo o derecho del nodo actual, dependiendo de si la respuesta a la pregunta fue "Si" o "No". Con cada*

decisión, se sigue una arista desde un padre a un hijo, eventualmente trazando un camino en el árbol desde la raíz a un nodo externo. Tales árboles binarios son conocidos como árboles de decisión*, porque cada nodo externo V en tal árbol representa una decisión de que hacer si la pregunta asociada con los ancestros de v fueron contestadas en una forma que llevan a v. Un árbol de decisión es un árbol binario propio. La figura 6.8 muestra un árbol de decisión para ordenar en forma ascendente tres valores guardados en las variables A, B y C, donde los nodos internos del árbol representan comparaciones y los nodos externos representan la salida ordenada.*

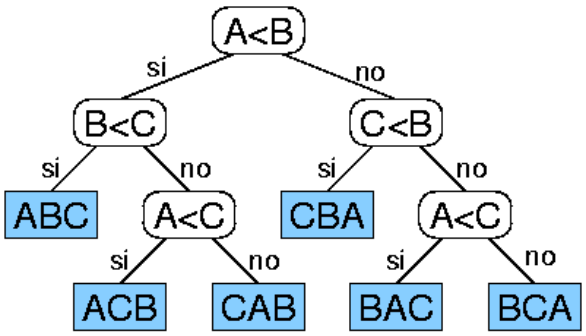

Figura 6.8: Un árbol de decisión para ordenar tres variables en orden creciente

Ejemplo 6.7. *Una expresión aritmética puede ser representada con un árbol binario cuyos nodos externos están asociados con variables o constantes, y cuyos nodos internos están asociados con los operadores $+$, $-$, $\times$ y $/$. Ver figura 6.9. Cada nodo en tal árbol tiene un valor asociado con este.*

- Si un nodo es externo, entonces su valor es aquel de su variable o constante.
- Si un nodo es interno, entonces su valor está definido aplicando la operación a los valores de sus hijos.

Un árbol de expresión aritmética es un árbol binario propio, ya que los operadores $+$, $-$, $\times$ y $/$ requieren exactamente dos operandos.

Definición recursiva de un árbol binario

Se puede definir un árbol binario en una forma recursiva tal que un árbol binario está vacío o consiste de:

- Un nodo r, llamado la raíz de T y que guarda un elemento.
- Un árbol binario, llamado el subárbol izquierdo de T.
- Un árbol binario, llamado el subárbol derecho de T.

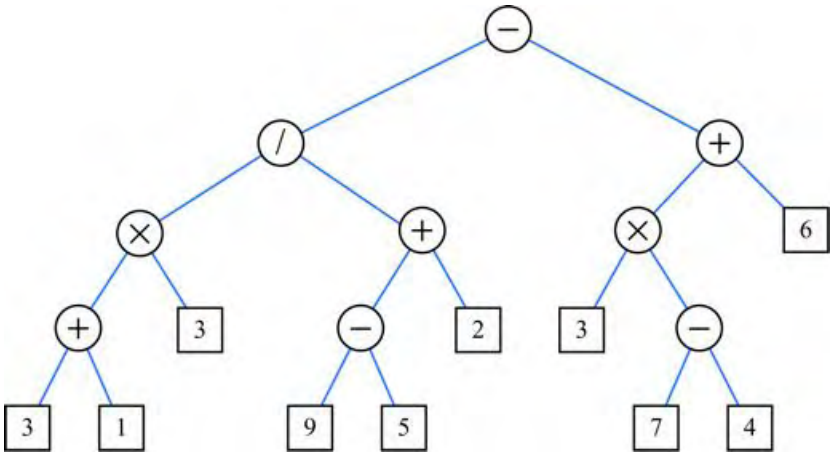

Figura 6.9: Un árbol binario representando la expresión aritmética $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$.

6.3.1. El ADT árbol binario

Como un tipo de dato abstracto, un árbol binario es una especialización de un árbol que soporta cuatro métodos accesores adicionales:

`left(`v`)`: Regresa el hijo izquierdo de v; una condición de error ocurre si v no tiene hijo izquierdo.
`right(`v`)`: Regresa el hijo derecho de v; una condición de error ocurre si v no tiene hijo derecho.
`hasLeft(`v`)`: Prueba si v tiene hijo izquierdo.
`hasRight(`v`)`: Prueba si v tiene hijo derecho.

No se definen métodos de actualización especializados sino que se consideran algunas posibles actualizaciones de métodos cuando se describan implementaciones específicas y aplicaciones de árboles binarios.

6.3.2. Una interfaz árbol binario en Java

Se modela un árbol binario como un tipo de dato abstracto que extiende el ADT árbol y agrega los cuatro métodos especializados para un árbol binario. En el listado 6.9, se muestra la interfaz que se define con esta aproximación. Como los árboles binarios son árboles ordenados, la colección iterable regresada por el método `children(`v`)`, que se hereda de la interfaz `Tree` guarda el hijo izquierdo de v antes que el hijo derecho.

```
1 /**
2  * Una interfaz para un árbol binario donde cada nodo tiene
3  * cero, uno o dos hijos.
4  *
5  */
```

```
6  public interface ArbolBinario<E> extends Tree<E> {
7    /** Regresa el hijo izquierdo de un nodo. */
8    public Posicion<E> left(Posicion<E> v)
9      throws InvalidPositionException, BoundaryViolationException;
10   /** Regresa el hijo derecho de un nodo. */
11   public Posicion<E> right(Posicion<E> v)
12     throws InvalidPositionException, BoundaryViolationException;
13   /** Indica si un un nodo tiene hijo izquierdo. */
14   public boolean hasLeft(Posicion<E> v) throws InvalidPositionException;
15   /** Indica si un un nodo tiene hijo derecho. */
16   public boolean hasRight(Posicion<E> v) throws InvalidPositionException;
17 }
```

Listado 6.9: Interfaz Java `ArbolBinario` para el ADT árbol binario que extiende a `Tree` (listado 6.1).

6.3.3. Propiedades del árbol binario

Las propiedades de los árboles binarios se relacionan con sus alturas y el número de nodos Se denota al conjunto de todos los nodos de un árbol T que están a la misma profundidad d como el *nivel* d de T. Para el árbol binario, el nivel 0 tiene a lo más un nodo, la raíz, el nivel 1 tiene a los más dos nodos, los hijos de la raíz, el nivel 2 tiene a lo más 4 hijos, y así sucesivamente. Ver figura 6.10. En general, el nivel d tiene a lo más 2^d nodos.

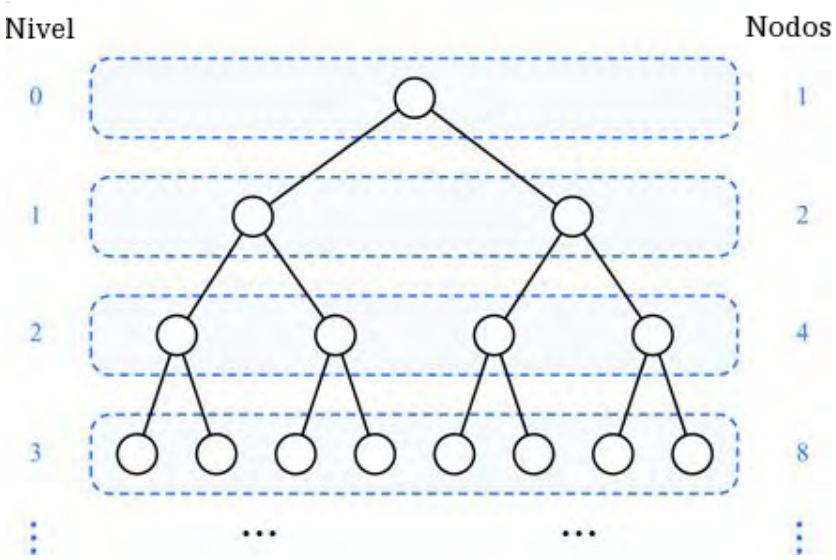

Figura 6.10: Número máximo de nodos en los niveles de un árbol binario.

Se observa que el número máximo de nodos en los niveles del árbol binario crece exponencialmente conforme se baja en el árbol. De esta observación, se pueden derivar las siguientes propiedades de relación entre la altura de un árbol

binario T con el número de nodos.

Proposición 6.2: *Sea T un árbol binario no vacío, y sean n, n_E, n_I y h el número de nodos, número de nodos externos, numero de nodos internos, y la altura de T, respectivamente. Entonces T tiene las siguientes propiedades:*

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq n - 1$

También, si T es propio, entonces T tiene las siguientes propiedades

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

También se cumple la siguiente relación entre el número de nodos internos y los nodos externos en un árbol binario propio

$$n_E = n_I + 1.$$

Para mostrar lo anterior se pueden quitar los nodos de T y dividirlos en dos "pilas", una pila con los nodos internos y otra con los nodos externos, hasta que T quede vacío. Las pilas están inicialmente vacías. Al final, la pila de nodos externos tendrá un nodo más que la pila de nodos internos, considerar dos casos:

> **Caso 1**: Si T tiene un sólo nodo v, se quita v y se coloca en la pila externa. Así, la pila de nodos externos tiene un sólo nodo y la pila de nodos internos está vacía.
>
> **Caso 2**: De otra forma, T tiene más de un nodo, se quita de T un nodo externo arbitrario w y su padre v, el cual es un nodo interno. Se coloca a w en la pila de nodos externos y v en la pila de nodos internos. Si v tiene un padre u, entonces se reconecta u con el ex-hermano z de w, como se muestra en la figura 6.11. Esta operación quita un nodo interno y uno externo, y deja al árbol como un árbol binario propio.
>
> Se repite esta operación, hasta que eventualmente se tiene un árbol final consistente de un sólo nodo. Entonces se quita el nodo del árbol final se coloca en la pila externa, teniendo así esta pila un nodo más que la pila interna.

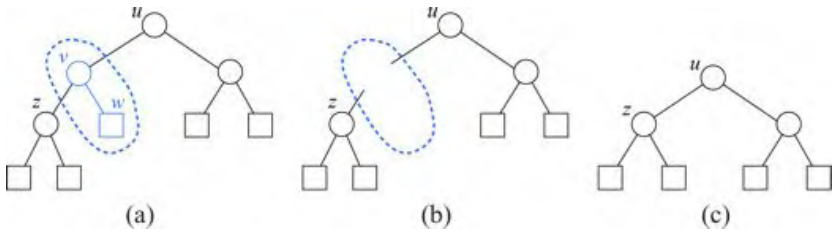

Figura 6.11: Operación para quitar un nodo externo y su nodo padre, usado en la demostración

6.3.4. Una estructura enlazada para árboles binarios

Una forma natural de realizar un árbol binario T es usar una estructura ligada, donde se representa cada nodo v de T por un objeto `Posicion`, ver figura 6.12, con los campos dando referencias al elemento guardado en v y a objetos `Posicion` asociados con los hijos y el padre de v.

Si v es la raíz de T, entonces el campo `padre` de v es `null`. También, se guarda el número de nodos de T en la variable `tam`.

Implementación Java de un nodo del árbol binario

Se emplea la interfaz Java `PosicionAB` para representar un nodo del árbol binario. Esta interfaz extiende `Posicion`, por lo que se hereda el método `elemento`, y tiene métodos adicionales para poner el elemento en el nodo, `setElemento` y para poner y devolver el padre, el hijo izquierdo y el hijo derecho, como se muestra en el listado 6.10.

```
1  /**
2   * Interfaz para un nodo de un árbol binario. Esta mantiene un elemento,
3   * un nodo padre, un nodo izquierdo, y nodo derecho.
4   *
5   */
6  public interface PosicionAB<E> extends Posicion<E> {  // hereda elemento()
7    public void setElemento(E o);
8    public PosicionAB<E> getIzq();
9    public void setIzq(PosicionAB<E> v);
10   public PosicionAB<E> getDer();
11   public void setDer(PosicionAB<E> v);
12   public PosicionAB<E> getPadre();
13   public void setPadre(PosicionAB<E> v);
14 }
```

Listado 6.10: Interfaz Java `PosicionAB` para definir los métodos que permiten acceder a los elementos de un nodo del árbol binario.

La clase `NodoAB`, listado 6.11, implementa la interfaz `PosicionAB` para un objeto con los campos `elemento`, `izq`, `der`, y `padre`, lo cual, para un nodo v, se refiere al elemento en v, el hijo izquierdo de v, el hijo derecho de v, y el padre de v respectivamente.

```
1  /**
2   * La clase implementa un nodo de un árbol binario guardando las referencias
```

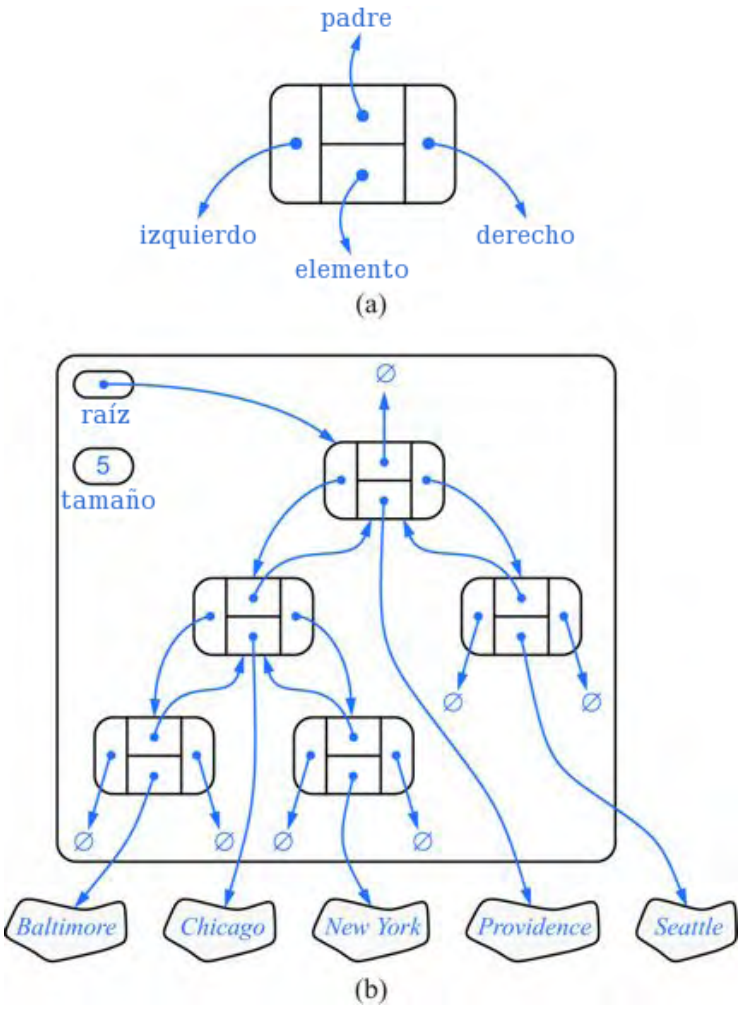

Figura 6.12: Un nodo (a) y una estructura enlazada (b) para representar un árbol binario.

```
3   * a un elemento, un nodo padre, un nodo izquierdo, y un nodo derecho.
4   *
5   */
6  public class NodoAB<E> implements PosicionAB<E> {
7    private E elemento;  // elemento guardado en este nodo
8    private PosicionAB<E> izq, der, padre;  // nodos adyacentes
9    /** Constructor por defecto */
10   public NodoAB() { }
11   /** Constructor principal */
12   public NodoAB(E elemento, PosicionAB<E> padre,
13     PosicionAB<E> izq, PosicionAB<E> der) {
14     setElemento(elemento);
15     setPadre(padre);
16     setIzq(izq);
17     setDer(der);
18   }
19   /** Regresa el elemento guardado en esta posición */
20   public E elemento() { return elemento; }
21   /** Pone el elemento guardado en esta posición */
22   public void setElemento(E o) { elemento=o; }
23   /** Regresa el hijo izquierdo de esta posición */
24   public PosicionAB<E> getIzq() { return izq; }
25   /** Pone el hijo izquierdo de esta posición */
26   public void setIzq(PosicionAB<E> v) { izq=v; }
27   /** Regresa el hijo derecho de esta posición */
28   public PosicionAB<E> getDer() { return der; }
29   /** Pone el hijo derecho de esta posición */
30   public void setDer(PosicionAB<E> v) { der=v; }
31   /** Regresa el padre de esta posición */
32   public PosicionAB<E> getPadre() { return padre; }
33   /** Pone el padre de esta posición */
34   public void setPadre(PosicionAB<E> v) { padre=v; }
35 }
```

Listado 6.11: Clase auxiliar `NodoAB` para implementar los nodos del árbol binario.

Implementación Java del árbol binario enlazado

En el listado 6.12 se muestra la clase `ArbolBinarioEnlazado` que implementa la interfaz `ArbolBinario`, listado 6.9, usando una estructura de datos enlazada. Esta clase guarda el tamaño del árbol y una referencia al objeto `NodoAB` asociado con la raíz del árbol en variables internas. Adicional a los métodos de la interfaz `ArbolBinario`, la clase tiene otros métodos, incluyendo el método accesor `sibling(`v`)`, el cual regresa el hermano de un nodo v, y los siguientes métodos de actualización:

`addRoot`(e): Crea y regresa un nuevo nodo r que guarda el elemento e y hace a r la raíz del árbol, un error ocurre si el árbol no está vacío.

`insertLeft`(v, e): Crea y regresa un nuevo nodo w que guarda a e, agrega a w como el hijo izquierdo de v y regresa w, un error ocurre si v ya tiene un hijo izquierdo.

`insertRight`(v, e): Crea y regresa un nuevo nodo w que guarda a e, agrega a w como el hijo derecho de v y regresa w, un error ocurre si v ya tiene un hijo derecho.

`remove`(v): Quita el nodo v y lo reemplaza con su hijo, si tiene, y regresa el elemento guardado en v; un error ocurre si v tiene dos hijos.

`attach`(v, T_1, T_2): Une T_1 y T_2 como los subárboles izquierdo y derecho del nodo externo v, una condición de error ocurre si v no es externo.

La clase `ArbolBinarioEnlazado` tiene un constructor sin argumentos que regresa un árbol binario vacío. Iniciando con este árbol vacío, se puede construir cualquier árbol binario creando el primer nodo con el método `addRoot` y aplicando repetidamente los métodos `insertLeft`, `insertRight` o el método `attach`. De igual forma, se puede desmantelar cualquier árbol binario T usando la operación `remove`, llegando hasta un árbol binario vacío.

Cuando una posición v es pasado como un argumento a alguno de los métodos de la clase `ArbolBinarioEnlazado`, su validez se revisa llamando un método auxiliar de ayudo, `checkPosition`(v). Una lista de los nodos visitados en un recorrido en preorden del árbol es construido por un método recursivo `preorderPositions`. Las condiciones de error son indicadas lanzando excepciones `InvalidPositionException`, `BoundaryViolationException`, `EmptyTreeException`, y `NonEmptyTreeException`.

```
1  import java.util.Iterator;
2  /**
3    * Una implementación de la interfaz ArbolBinario usando una estructura
4    * enlazada.
5    * @see ArbolBinario */
6  public class ArbolBinarioEnlazado<E> implements ArbolBinario<E> {
7    protected PosicionAB<E> raiz; // referencia a la raíz
8    protected int tam;    // número de nodos
9    /**  Crea un árbol binario vacío */
10   public ArbolBinarioEnlazado() {
11     raiz = null;  // empezar con un árbol binario
12     tam = 0;
13   }
14   /** Regresa el número de nodos en el árbol. */
15   public int size() {
16     return tam;
17   }
18   /** Indica si el árbol está vacío. */
19   public boolean isEmpty() {
20     return (tam == 0);
21   }
22   /** Indica si un nodo es interno. */
23   public boolean isInternal(Posicion<E> v) throws InvalidPositionException {
24     checkPosicion(v);   // método auxiliar
```

```
25      return (hasLeft(v) || hasRight(v));
26    }
27    /** Indica si un nodo es externo */
28    public boolean isExternal(Posicion<E> v) throws InvalidPositionException {
29      return !isInternal(v);
30    }
31    /** Indica si un nodo es la raíz */
32    public boolean isRoot(Posicion<E> v) throws InvalidPositionException {
33      checkPosicion(v);
34      return (v == root());
35    }
36    /** Indica si un nodo tiene hijo izquierdo */
37    public boolean hasLeft(Posicion<E> v) throws InvalidPositionException {
38      PosicionAB<E> vv = checkPosicion(v);
39      return (vv.getIzq() != null);
40    }
41    /** Indica si un nodo tiene hijo derecho */
42    public boolean hasRight(Posicion<E> v) throws InvalidPositionException {
43      PosicionAB<E> vv = checkPosicion(v);
44      return (vv.getDer() != null);
45    }
46    /** Regresa la raíz del árbol */
47    public Posicion<E> root() throws EmptyTreeException {
48      if (raiz == null)
49        throw new EmptyTreeException("El árbol está vacío");
50      return raiz;
51    }
52    /** Regresa el hijo izquierdo de un nodo. */
53    public Posicion<E> left(Posicion<E> v)
54      throws InvalidPositionException, BoundaryViolationException {
55      PosicionAB<E> vv = checkPosicion(v);
56      Posicion<E> leftPos = vv.getIzq();
57      if (leftPos == null)
58        throw new BoundaryViolationException("Sin hijo izquierdo");
59      return leftPos;
60    }
61    /** Regresa el hijo izquierdo de un nodo. */
62    public Posicion<E> right(Posicion<E> v)
63      throws InvalidPositionException, BoundaryViolationException {
64      PosicionAB<E> vv = checkPosicion(v);
65      Posicion<E> rightPos = vv.getDer();
66      if (rightPos == null)
67        throw new BoundaryViolationException("Sin hijo derecho");
68      return rightPos;
69     }
70    /** Regresa el padre de un nodo. */
71    public Posicion<E> parent(Posicion<E> v)
72      throws InvalidPositionException, BoundaryViolationException {
73      PosicionAB<E> vv = checkPosicion(v);
74      Posicion<E> parentPos = vv.getPadre();
75      if (parentPos == null)
76        throw new BoundaryViolationException("Sin padre");
77      return parentPos;
78    }
79    /** Regresa una colección iterable de los hijos de un nodo. */
80    public Iterable<Posicion<E>> children(Posicion<E> v)
81      throws InvalidPositionException {
82     ListaPosicion<Posicion<E>> children = new ListaNodoPosicion<Posicion<E>>();
83      if (hasLeft(v))
84        children.addLast(left(v));
85      if (hasRight(v))
86        children.addLast(right(v));
87      return children;
88    }
89    /** Regresa una colección iterable de los nodos de un árbol. */
90    public Iterable<Posicion<E>> positions() {
91     ListaPosicion<Posicion<E>> positions = new ListaNodoPosicion<Posicion<E>>();
92      if(tam != 0)
```

```
93        preordenPosiciones(root(), positions);  // asignar posiciones en preorden
94      return positions;
95    }
96   /** Regresar un iterado de los elementos guardados en los nodos. */
97    public Iterator<E> iterator() {
98      Iterable<Posicion<E>> positions = positions();
99     ListaPosicion<E> elements = new ListaNodoPosicion<E>();
100    for (Posicion<E> pos: positions)
101      elements.addLast(pos.elemento());
102     return elements.iterator();  // Un iterador de elementos
103   }
104   /** Reemplaza el elemento en un nodo. */
105   public E replace(Posicion<E> v, E o)
106     throws InvalidPositionException {
107     PosicionAB<E> vv = checkPosicion(v);
108     E temp = v.elemento();
109     vv.setElemento(o);
110     return temp;
111   }
112   // Métodos adicionales accesores
113   /** Regresa el hermano de un nodo. */
114   public Posicion<E> sibling(Posicion<E> v)
115     throws InvalidPositionException, BoundaryViolationException {
116       PosicionAB<E> vv = checkPosicion(v);
117       PosicionAB<E> parentPos = vv.getPadre();
118       if (parentPos != null) {
119         PosicionAB<E> sibPos;
120         PosicionAB<E> leftPos = parentPos.getIzq();
121         if (leftPos == vv)
122           sibPos = parentPos.getDer();
123         else
124           sibPos = parentPos.getIzq();
125         if (sibPos != null)
126          return sibPos;
127       }
128       throw new BoundaryViolationException("Sin hermano");
129   }
130   // Métodos adicionales de actualización
131   /** Agrega un nodo raíz a un árbol vacío */
132   public Posicion<E> addRoot(E e) throws NonEmptyTreeException {
133     if(!isEmpty())
134       throw new NonEmptyTreeException("El árbol ya tiene raíz");
135     tam = 1;
136     raiz = createNode(e,null,null,null);
137     return raiz;
138   }
139   /** Insertar un hijo izquierdo en un nodo dado. */
140   public Posicion<E>  insertLeft(Posicion<E> v, E e)
141     throws InvalidPositionException {
142     PosicionAB<E> vv = checkPosicion(v);
143     Posicion<E> leftPos = vv.getIzq();
144     if (leftPos != null)
145       throw new InvalidPositionException("El nodo ya tiene hijo izquierdo");
146     PosicionAB<E> ww = createNode(e, vv, null, null);
147     vv.setIzq(ww);
148     tam++;
149     return ww;
150   }
151   /** Insertar un hijo derecho en un nodo dado. */
152   public Posicion<E>  insertRight(Posicion<E> v, E e)
153     throws InvalidPositionException {
154     PosicionAB<E> vv = checkPosicion(v);
155     Posicion<E> rightPos = vv.getDer();
156     if (rightPos != null)
157       throw new InvalidPositionException("El nodo ya tiene hijo derecho");
158     PosicionAB<E> w = createNode(e, vv, null, null);
159     vv.setDer(w);
160     tam++;
```

```
161     return w;
162   }
163   /** Quitar un nodo con un hijo o sin hijos. */
164   public E remove(Posicion<E> v)
165     throws InvalidPositionException {
166     PosicionAB<E> vv = checkPosicion(v);
167     PosicionAB<E> leftPos = vv.getIzq();
168     PosicionAB<E> rightPos = vv.getDer();
169     if (leftPos != null && rightPos != null)
170       throw new InvalidPositionException("No se puede remover nodo con dos hijos");
171     PosicionAB<E> ww;   // el único hijo de v, si tiene
172     if (leftPos != null)
173       ww = leftPos;
174     else if (rightPos != null)
175       ww = rightPos;
176     else    // v es una hoja
177       ww = null;
178     if (vv == raiz) {   // v es la raíz
179       if (ww != null)
180         ww.setPadre(null);
181         raiz = ww;
182     }
183     else {    // v no es la raíz
184       PosicionAB<E> uu = vv.getPadre();
185       if (vv == uu.getIzq())
186         uu.setIzq(ww);
187       else
188         uu.setDer(ww);
189       if(ww != null)
190         ww.setPadre(uu);
191     }
192     tam--;
193     return v.elemento();
194   }
195
196   /** Conecta dos árboles para ser los subárboles de un nodo externo. */
197   public void attach(Posicion<E> v, ArbolBinario<E> T1, ArbolBinario<E> T2)
198     throws InvalidPositionException {
199     PosicionAB<E> vv = checkPosicion(v);
200     if (isInternal(v))
201       throw new InvalidPositionException("No se pueda conectar del nodo interno");
202     if (!T1.isEmpty()) {
203       PosicionAB<E> r1 = checkPosicion(T1.root());
204       vv.setIzq(r1);
205       r1.setPadre(vv);    // T1 deberá ser inválidado
206     }
207     if (!T2.isEmpty()) {
208       PosicionAB<E> r2 = checkPosicion(T2.root());
209       vv.setDer(r2);
210       r2.setPadre(vv);    // T2 deberá ser inválidado
211     }
212   }
213   /** Intercambiar los elementos en dos nodos */
214   public void swapElements(Posicion<E> v, Posicion<E> w)
215     throws InvalidPositionException {
216     PosicionAB<E> vv = checkPosicion(v);
217     PosicionAB<E> ww = checkPosicion(w);
218     E temp = w.elemento();
219     ww.setElemento(v.elemento());
220     vv.setElemento(temp);
221   }
222   /** Expandir un nodo externo en un nodo interno con dos nodos
223     * externos hijos */
224   public void expandExternal(Posicion<E> v, E l, E r)
225     throws InvalidPositionException {
226     if (!isExternal(v))
227       throw new InvalidPositionException("El nodo no es externo");
228     insertLeft(v, l);
```

```
229       insertRight(v, r);
230     }
231     /** Quitar un nodo externo v y reemplazar su padre con el hermano de v*/
232     public void removeAboveExternal(Posicion<E> v)
233       throws InvalidPositionException {
234       if (!isExternal(v))
235         throw new InvalidPositionException("El nodo no es externo");
236       if (isRoot(v))
237         remove(v);
238       else {
239         Posicion<E> u = parent(v);
240         remove(v);
241         remove(u);
242       }
243     }
244     // Métodos auxiliares
245     /** Si v es un nodo de un árbol binario, convertir a PosicionAB,
246       * si no lanzar una excepción */
247     protected PosicionAB<E> checkPosicion(Posicion<E> v)
248       throws InvalidPositionException {
249       if (v == null || !(v instanceof PosicionAB))
250         throw new InvalidPositionException("La posición no es válida");
251       return (PosicionAB<E>) v;
252     }
253     /** Crear un nuevo nodo de un árbol binario */
254     protected PosicionAB<E> createNode(E element, PosicionAB<E> parent,
255             PosicionAB<E> left, PosicionAB<E> right) {
256       return new NodoAB<E>(element,parent,left,right); }
257     /** Crear una lista que guarda los nodos en el subárbol de un nodo,
258       * ordenada de acuerdo al recorrido en preordel del subárbol. */
259     protected void preordenPosiciones(Posicion<E> v, ListaPosicion<Posicion<E>> pos)
260       throws InvalidPositionException {
261       pos.addLast(v);
262       if (hasLeft(v))
263         preordenPosiciones(left(v), pos); // recursividad en el hijo izquierdo
264       if (hasRight(v))
265         preordenPosiciones(right(v), pos);  // recursividad en el hijo derecho
266     }
267     /** Crear una lista que guarda los nodos del subárbol de un nodo,
268        ordenados de acuerdo al recorrido en orden del subárbol. */
269     protected void posicionesEnorden(Posicion<E> v, ListaPosicion<Posicion<E>> pos)
270       throws InvalidPositionException {
271       if (hasLeft(v))
272         posicionesEnorden(left(v), pos);  // recursividad en el hijo izquierdo
273       pos.addLast(v);
274       if (hasRight(v))
275         posicionesEnorden(right(v), pos); // recursividad en el hijo derecho
276     }
277   }
```

Listado 6.12: `ArbolBinarioEnlazado` implementando la interfaz `ArbolBinario`.

Rendimiento de `ArbolBinarioEnlazado`

Los tiempos de ejecución de los métodos de la clase `ArbolBinarioEnlazado`, el cual usa una representación de estructura enlazada, se muestran enseguida:

- Los métodos `size()` e `isEmpty()` usan una variable de instancia que guarda el número de nodos de T, y cada uno toma tiempo $O(1)$.
- Los métodos accesores `root`, `left`, `right`, `sibling`, y `parent` toman tiempo $O(1)$.

- El método `replace`(v, e) toma tiempo $O(1)$.
- Los métodos `iterator()` y `positions()` están implementados haciendo un recorrido en preorden del árbol, usando el método auxiliar `preorden-Posiciones`. El iterador de salida es generado con el método `iterator()` de la clase `ListaNodoPosicion`. Los métodos `iterator()` y `positions()` toma tiempo $O(n)$.
- El método `children` emplea una aproximación similar para construir la colección iterable que se regresa, pero este corre en tiempo $O(1)$, ya que hay a lo más dos hijos para cualquier nodo en un árbol binario.
- Los métodos de actualización `insertLeft`, `insertRight`, `attach`, y `remove` corren en tiempo $O(1)$, ya que involucran manipulación en tiempo constante de un número constante de nodos.

6.3.5. Representación lista arreglo para el árbol binario

Esta representación alterna de un árbol binario T está basado en una forma de numerar los nodos de T. Para cada nodo v de T, sea $p(v)$ el entero definido como sigue:

- Si v es la raíz de T, entonces $p(v) = 1$.
- Si v es el hijo izquierdo de un nodo u, entonces $p(v) = 2p(u)$.
- Si v es el hijo derecho de un nodo u, entonces $p(v) = 2p(u) + 1$.

La función de numeración p se conoce como *numeración de nivel* de los nodos en un árbol binario T, esta numera los nodos en cada nivel de T en orden creciente de izquierda a derecha, pudiendo brincarse algunos números, ver figura 6.13.

La función de numeración de nivel p sugiere una representación de un árbol binario T por medio de una lista arreglo S tal que cada nodo v de T es el elemento de S en el índice $p(v)$. Como se menciono en el capítulo previo, se lleva a cabo la lista arreglo S mediante un arreglo extendible, ver sección 5.1. Tal implementación es simple y eficiente, para poder usarlo para los métodos `root`, `parent`, `left`, `right`, `hasLeft`, `hasRight`, `isInternal`, `isExternal`, e `isRoot` usando operaciones aritméticas sobre los números $p(v)$ asociados con cada nodo v involucrado en la operación.

Se muestra un ejemplo de representación lista arreglo de un árbol binario en la figura 6.14.

Sea n el número de nodos de T, y sea p_M el valor máximo de $p(v)$ sobre todos los nodos de T. La lista arreglo S tiene tamaño $N = p_M + 1$ ya que el elemento de S en el índice 0 no está asociado con ningún nodo de T. También, S tendrá, en general, un número de elementos vacíos que no se refieren a nodos existentes de T. En el peor caso, $N = 2^n$.

Los tiempos de ejecución para un árbol binario T implementado con una lista arreglo S, son iguales a los dados para el árbol binario enlazado, sección 6.3.4.

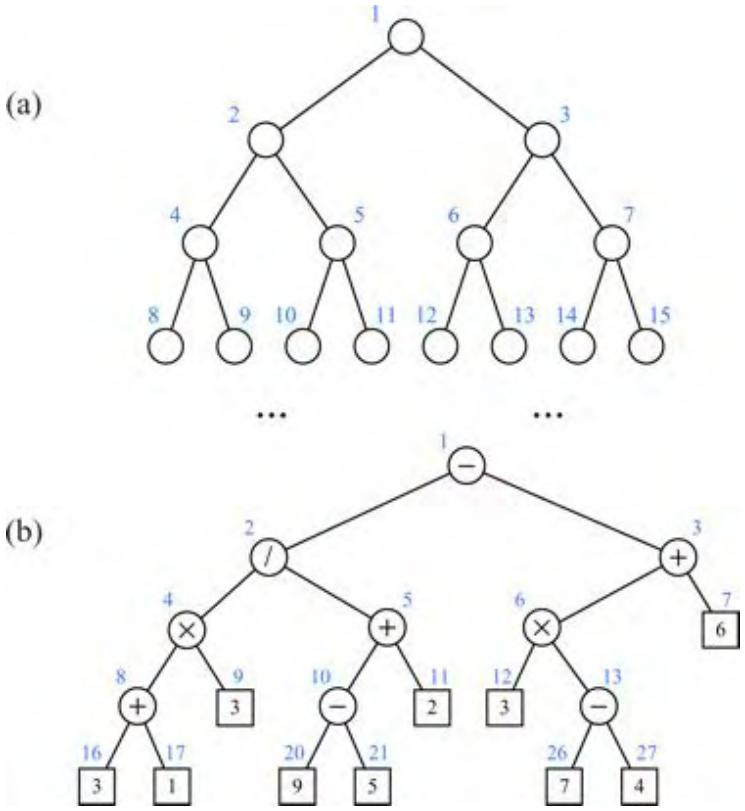

Figura 6.13: Numeración por nivel de un árbol binario: (a) esquema general; (b) un ejemplo.

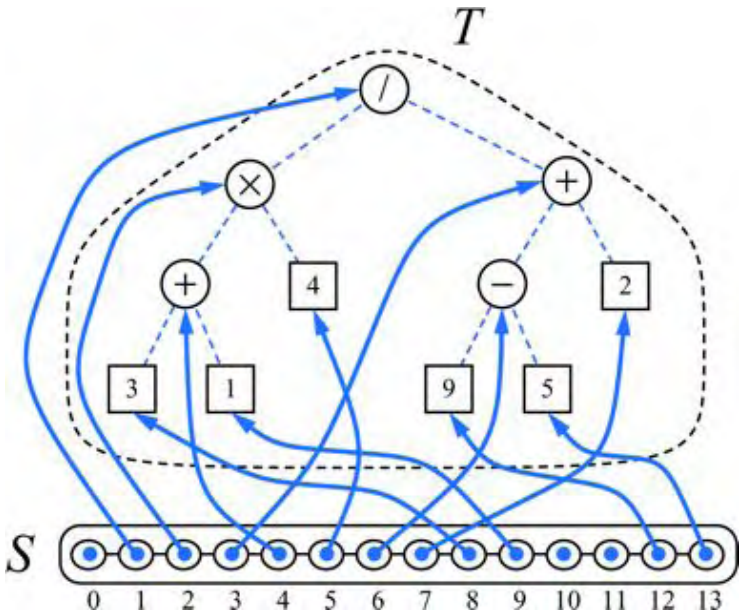

Figura 6.14: Representación de un árbol binario T por medio de una lista arreglo S.

6.3.6. Recorrido de árboles binarios

Como en los árboles generales, los cálculos de árboles binarios involucran recorridos. Se muestra enseguida ejemplos de recorridos que pueden ocuparse para resolver algunos problemas.

Construcción de un árbol de expresión

Para el problema de construir un árbol de expresión de una expresión aritmética completamente parentetizada de tamaño n se da el algoritmo `construir Expresion`, suponiendo que todas las operaciones aritméticas son binarias y las variables no están parentetizadas. Así, cada subexpresión parentetizada contiene un operador en el centro. El algoritmo usa una pila P mientras se explora la expresión de entrada E buscando variables, operadores, y paréntesis derechos.

- Cuando se ve una variable u operador x, se crea un árbol binario de un sólo nodo T, cuya raíz guarda x y se mete T a la pila.

- Cuando se ve un paréntesis derecho, ")", se sacan de la cima tres árboles de la pila S, los cuales representan una subexpresión $(E_1 \circ E_2)$. Entonces se conectan los árboles para E_1 y E_2 en uno para $\circ$, y se mete el resultado de regreso a la pila P.

Se repite esto hasta que la expresión E ha sido procesada, en ese momento el elemento de la cima en la pila es el árbol de expresión para E. El tiempo total de ejecución es $O(n)$.

Algoritmo construirExpresion(E):
  Entrada: Una expresión aritmética completamente parentetizada $E = e_0, e_1, \ldots, e_{n-1}$, siendo cada e_i una variable, operador, o símbolo paréntesis
  Salida: Un árbol binario T representando la expresión aritmética E
  $P \leftarrow$ una nueva pila inicialmente vacía
  Para $i \leftarrow 0$ hasta $n-1$ **Hacer**
    Si e_i es una variable o un operador **Entonces**
      $T \leftarrow$ un nuevo árbol binario vacío
      T.addRoot(e_i)
      P.push(T)
    si no si $e_i = '('$ **Entonces**
      Continuar el ciclo
    si no { $e_i = ')'$ }
      $T_2 \leftarrow P$.pop() { el árbol que representa a E_2 }
      $T \leftarrow P$.pop() { el árbol que representa a $\circ$ }
      $T_1 \leftarrow P$.pop() { el árbol que representa a E_1 }
      T.attach(T.root(), T_1, T_2)
      P.push(T)
  Regresar P.pop()

Recorrido en preorden de un árbol binario

Como cualquier árbol binario puede ser visto como un árbol general, el recorrido en preorden para árboles generales, sección 6.2.2, puede ser aplicado a cualquier árbol binario y simplificado como se muestra enseguida.

Algoritmo preordenBinario(T, v):
  Realizar la acción "visita" para el nodo v
  Si v tiene un hijo izquierdo u en T **Entonces**
    preordenBinario(T, u) { recursivamente recorrer el subárbol izquierdo }
  Si v tiene un hijo derecho w en T **Entonces**
    preordenBinario(T, w) { recursivamente recorrer el subárbol derecho }

Como en el caso de los árboles generales, hay varias aplicaciones de recorrido en preorden para árboles binarios.

Recorrido en postorden de un árbol binario

De manera análoga, el recorrido en postorden para árboles generales, sección 6.2.3, puede ser especializado para árboles binarios, como se muestra enseguida.

Algoritmo postordenBinario(T, v):
  Si v tiene un hijo izquierdo u en T **Entonces**
    postordenBinario(T, u) { recursivamente recorrer el subárbol izquierdo }
  Si v tiene un hijo derecho w en T **Entonces**
    postordenBinario(T, w) { recursivamente recorrer el subárbol
  Realizar la acción "visita" para el nodo v
derecho }

Evaluación de un árbol de expresión

El recorrido en postorden de un árbol binario puede ser usado para resolver el problema de evaluación del árbol de expresión. En este problema, se da un árbol de expresión aritmética, es decir, un árbol binario donde cada nodo externo tiene asociado un valor con este y cada nodo interno tiene un operador aritmético asociado con este, y se quiere calcular el valor de la expresión aritmética representada por el árbol.

El algoritmo `evaluarExpresion`, dado enseguida, evalúa la expresión asociada con el subárbol enraizado en el nodo v de un árbol T de expresión aritmética realizando un recorrido en postorden de T iniciando en v. En este caso, la acción "visita" consiste realizar una sola operación aritmética. Se supone que el árbol de expresión aritmética es un árbol binario propio.

Algoritmo evaluarExpresion(T, v):
  Si v es un nodo interno en T **Entonces**
    Sea $\circ$ el operador guardado en v
    $x \leftarrow$ evaluarExpresion(T, T.left(v))
    $y \leftarrow$ evaluarExpresion(T, T.right(v))
    Regresar $x \circ y$
  Si no
    Regresar el valor guardado en v

El algoritmo `evaluarExpresion`, al hacer un recorrido en postorden, da un tiempo $O(n)$ para evaluar un expresión aritmética representada por un árbol binario con n nodos. Al igual que en el recorrido en postorden general, el recorrido en postorden para árboles binarios puede ser aplicado a otros problemas de evaluación "bottom-up" también, tal como calcular el tamaño dado en el ejemplo 6.5.

Recorrido en orden de un árbol binario

Un método de recorrido adicional para un árbol binario es el recorrido *en orden* . En este recorrido, se visita un nodo entre los recorridos recursivos de sus subárboles izquierdo y derecho. El recorrido en orden del subárbol enraizado en un nodo v de un árbol binario T se da en el siguiente algoritmo.

Algoritmo enorden(T, v):
  Si v tiene un hijo izquierdo u en T **Entonces**
    enordenBinario(T, u) { recursivamente recorrer el subárbol izquierdo }
  Realizar la acción "visita" para el nodo v
  Si v tiene un hijo derecho w en T **Entonces**
    enordenBinario(T, w) { recursivamente recorrer el subárbol derecho }

El recorrido en orden de un árbol binario T puede ser visto informalmente como visitar los nodos de T "de izquierda a derecha". Además, para cada nodo v, el recorrido en orden visita v después de que todos los nodos en el subárbol izquierdo de v y antes que todos los nodos en subárbol derecho de v, ver figura 6.15.

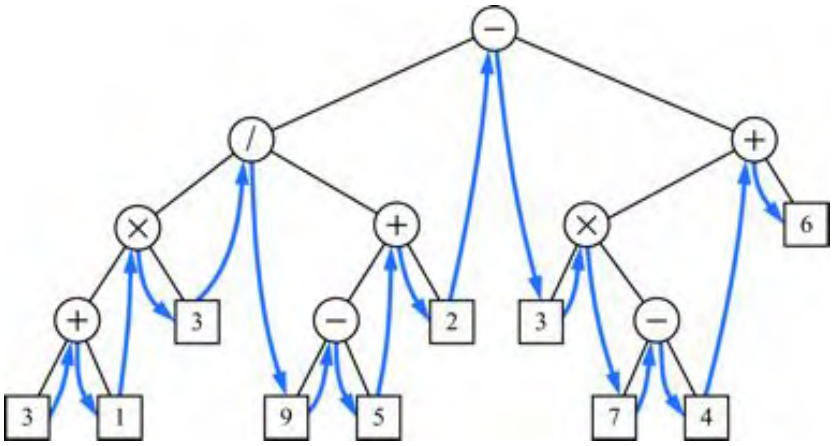

Figura 6.15: Recorrido en orden de un árbol binario.

Árboles binarios de búsqueda

Sea S un conjunto cuyos elementos tienen una relación de orden. Por ejemplo, S podría ser un conjunto de enteros. Un árbol *binario de búsqueda* para S es un árbol binario propio T tal que

- Cada nodo interno v de T guarda un elemento de S, denotado con $x(v)$.
- Para cada nodo interno v de T, los elementos guardados en el subárbol izquierdo de v son menores que o iguales a $x(v)$ y los elementos guardados en el subárbol derecho de v son mayores que $x(v)$.
- Los nodos externos de T no guardan ningún elemento.

Un recorrido en orden de los nodos internos de un árbol binario de búsqueda T visita los elementos en orden creciente, ver figura 6.16.

Se puede usar un árbol binario de búsqueda T para un conjunto S para encontrar si un valor de búsqueda y está en S, recorriendo el camino descendente

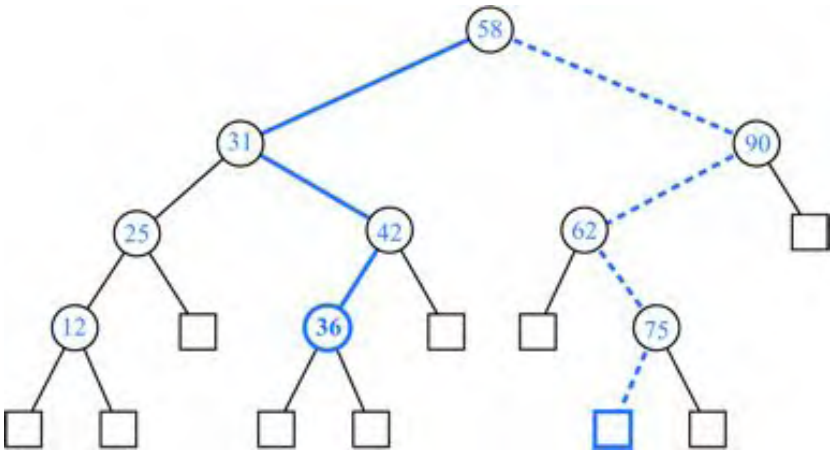

Figura 6.16: Un árbol binario de búsqueda ordenando enteros. El camino azul sólido es recorrido cuando se busca exitosamente el 36. El camino azul discontinuo es recorrido cuando se busca sin éxito el 70.

del árbol T, iniciando con la raíz, En cada nodo interno v encontrado, se compara el valor de búsqueda y con el elemento $x(v)$ guardado en v. Si $y < x(v)$, entonces la búsqueda continúa en el subárbol izquierdo de v. Si $y = x(v)$, entonces la búsqueda termina exitosamente. Si $y > x(v)$, entonces la búsqueda continúa en el subárbol derecho de v. Finalmente, si se alcanza un nodo externo, la búsqueda termina sin éxito. El árbol binario de búsqueda puede ser visto como un árbol de decisión, donde la pregunta hecha en cada nodo interno es si el elemento en ese nodo es menor que, igual a, o mayor que el elemento que se está buscando.

El tiempo de ejecución de la búsqueda en un árbol binario de búsqueda T es proporcional a la altura de T. La altura de un árbol binario propio con n nodos puede ser tan pequeño como $\log(n+1) - 1$ o tan grande como $(n-1)/2$, ver la propiedades del árbol binario, sección 6.3.3. Por lo anterior los árboles binarios de búsqueda son más eficientes cuando tienen alturas pequeñas.

Recorrido en orden para dibujar un árbol

El recorrido en orden puede ser aplicado al problema de dibujar un árbol binario. Se puede dibujar un árbol binario T con un algoritmo que asigne coordenadas x e y a un nodo de v de T usando las siguientes reglas, ver figura 6.17:

- $x(v)$ es el número de nodos visitados antes de v en el recorrido en orden de T.
- $y(v)$ es la profundidad de v en T.

En esta aplicación, se toma la convención común en computación para las gráficas de que la coordenada x se incrementa de izquierda a derecha y la

coordenada y se incrementa de arriba hacia abajo. Así el origen está en la esquina superior izquierda de la pantalla de la computadora.

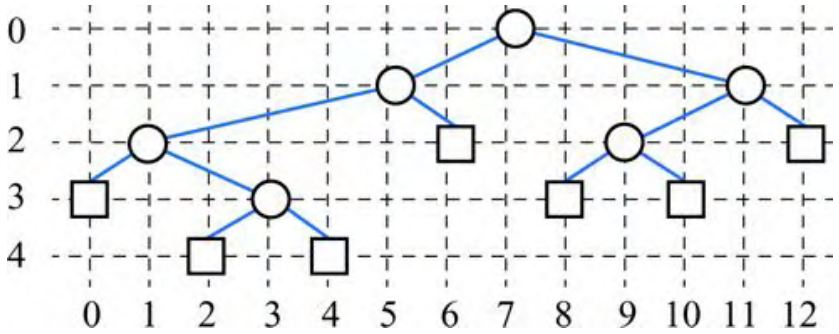

Figura 6.17: Un dibujo en orden de un árbol binario.

Recorrido de Euler de un árbol binario

Los algoritmos de recorrido para árbol que se han visto, todos ellos son de la forma de iteradores. Cada recorrido visita los nodos de un árbol en un cierto orden, y se garantiza visitar cada nodo exactamente una vez. Se pueden unificar los tres algoritmos de recorrido dados en una sola estructura, relajando los requerimientos de que cada nodo sea visitado una sola vez. La ventaja de este recorrido es que permite tipos más generales de algoritmos para ser expresados sencillamente.

El recorrido de Euler de un árbol binario T puede ser informalmente definido como una "caminata" alrededor de T, que se inicia yendo desde la raíz hacia su hijo izquierdo, viendo a las aristas de T como "paredes" que siempre se conservan a la izquierda, ver figura 6.18. Cada nodo v de T es encontrado tres veces por el recorrido de Euler:

- "A la izquierda", antes del recorrido de Euler de subárbol izquierdo de v.
- "Desde abajo", entre los recorridos de Euler de los dos subárboles de v.
- "A la derecha", después del recorrido de Euler de subárbol derecho de v.

Si v es un nodo externo, entonces estas tres "visitas" suceden todas al mismo tiempo. Se describe el recorrido de Euler del subárbol enraizado en v en el siguiente algoritmo:

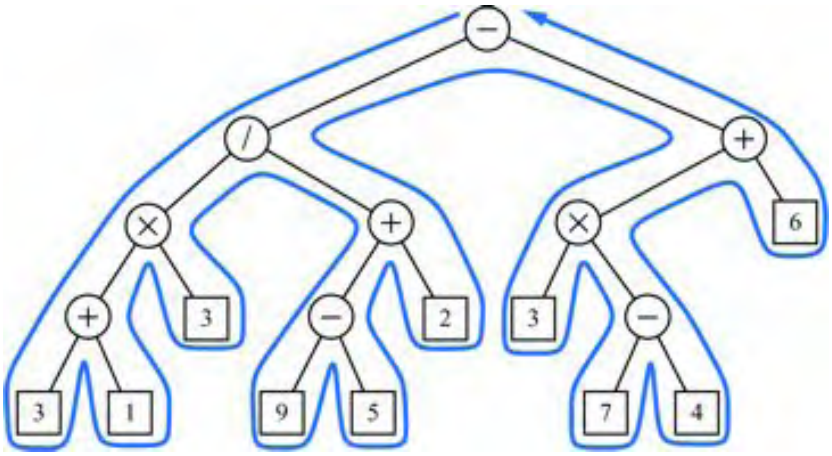

Figura 6.18: Un recorrido de Euler de un árbol binario.

Algoritmo RecorridoEuler(T, v):
realizar la acción para visitar nodo v a la izquierda
Si v tiene un hijo izquierdo u en T **Entonces**
RecorridoEuler(T, u) { recursivamente recorrer subárbol izquierdo de v }
realizar la acción para visitar nodo v desde abajo
Si v tiene un hijo derecho w en T **Entonces**
RecorridoEuler(T, w) { recursivamente recorrer subárbol derecho de v }
realizar la acción para visitar nodo v a la derecha

El tiempo de ejecución del recorrido de Euler de un árbol de n-nodos se obtiene así, suponiendo que cada acción visita toma tiempo $O(1)$. Como se usa una cantidad constante de tiempo en cada nodo del árbol durante el recorrido, el tiempo de ejecución total es $O(n)$.

El recorrido en preorden de un árbol binario es equivalente a un recorrido de Euler tal que cada nodo tiene asociado una acción "visita" ocurriendo solamente cuando este es encontrada a la izquierda. De la misma forma, los recorridos en orden y postorden son equivalentes a un recorrido de Euler tal encontrado desde abajo o a la derecha, respectivamente. Se puede usar el recorrido de Euler para hacer otros tipos de recorridos. Por ejemplo, suponer que se desea calcular el número de descendientes de cada nodo v en un árbol binario de n nodos. Se inicia un recorrido de Euler inicializando un contador a cero, y entonces incrementando el contador cada vez que se visita un nodo a la izquierda. Para determinar el número de descendientes de un nodo v, se calcula la diferencia entre los valores del contador cuando v es visitado a la izquierda y cuando es visitado a la derecha, y agregado uno. Esta regla da el número de descendientes de v, porque cada nodo en el subárbol enraizado es contado la visita de v y la visita de v a la derecha. Por lo tanto, se tiene un método tiempo $O(n)$ para calcular el número de descendientes de cada nodo.

Otra aplicación del recorrido de Euler es imprimir una expresión aritmética

completamente parentizada a partir de su árbol de expresión. El algoritmo `imprimirExpresión`, mostrado a continuación, realiza esta tarea realizando las siguientes acciones en un recorrido de Euler:

- Acción "a la izquierda": si el nodo es interno, imprimir "("
- Acción "desde abajo": imprimir el valor u operador guardado en el nodo
- Acción "a la derecha": si el nodo es interno, imprimir ")"

Algoritmo imprimirExpresión(T, v):
  Si T.isInternal(v) **Entonces**
    imprimir "("
  Si T.hasLeft(v) **Entonces**
    imprimirExpresión$(T, T$.left$(v))$
  Si T.isInternal(v) **Entonces**
    imprimir el operador guardado en v
  Si no
    imprimir el valor guardado en v
  Si T.hasRight(v) **Entonces**
    imprimirExpresión$(T, T$.right$(v))$
  Si T.isInternal(v) **Entonces**
    imprimir ")"

6.3.7. Plantilla método patrón

Los métodos de recorrido para árboles descritos previamente son ejemplos de patrones de diseño de software orientado al objeto, la *plantilla método patrón*. La plantilla método patrón describe un mecanismo genérico de cálculo que puede ser especializado para una aplicación particular redefiniendo cierto pasos. Siguiendo la plantilla método patrón, se diseña un algoritmo que implementa un recorrido de Euler genérico de un árbol binario. Se muestra enseguida el algoritmo plantillaRecorridoEuler.

Algoritmo plantillaRecorridoEuler(T, v):
  $r \leftarrow$ nuevo objeto del tipo $ResultadoRecorrido$
  visitarIzquierda(T, v, r)
  Si T.hasLeft(v) **Entonces**
    r.izq $\leftarrow$ plantillaRecorridoEuler$(T, T$.left$(v))$
  visitarAbajo(T, v, r)
  Si T.hasRight(v) **Entonces**
    r.der $\leftarrow$ plantillaRecorridoEuler$(T, T$.right$(v))$
  visitarDerecha(T, v, r)
  regresar r.salida

Cuando se llama con un nodo v, el método `plantillaRecorridoEuler` llama varios métodos auxiliares en diferentes fases del recorridos. Como se describe enseguida:

- Crear una varible local r del tipo *ResultadoRecorrido*, la cual es usada para guardar resultados intermedios del cálculo, que tiene campos *izq*, *der* y *salida*.
- Se llama al método auxiliar `visitarIzquierda`(T, v, r), el cual realiza los cálculos asociados encontrándose a la izquierda del nodo.
- Si v tiene hijo izquierdo, recursivamente se llama sobre el mismo con el hijo izquierdo de v y guarda el valor regresado en r.`izq`.
- Se llama al método auxiliar `visitarAbajo`(T, v, r), el cual realiza los cálculos asociados encontrándose abajo del nodo.
- Si v tiene hijo derecho, recursivamente se llama sobre el mismo con el hijo derecho de v y guarda el valor regresado en r.`der`.
- Se llama al método auxiliar `visitarDerecha`(T, v, r), el cual realiza los cálculos asociados encontrándose a la derecha del nodo.
- Regresar r.salida.

El método `plantillaRecorridoEuler` puede ser visto como una *plantilla* o "esqueleto" de un recorrido Euleriano.

Implementación Java

La clase `RecorridoEuler`, mostrada en el listado 6.13, implementa un recorrido transversal de Euler usando la plantilla método patrón. El recorrido transversal es hecha por el método `RecorridoEuler`. Los métodos auxiliares llamados por `RecorridoEuler` son lugares vacíos, tienen cuerpos vacíos o solo regresan `null`. La clase `RecorridoEuler` es abstracta y por lo tanto no puede ser instanciada. Contiene un método abstracto, llamado `ejecutar`, el cual necesita ser especificado en las subclases concretas de `RecorridoEuler`.

```
1  /**
2   * Plantilla para algoritmos que recorren un árbol binario usando un
3   * recorrido euleriano. Las subclases de esta clase redefinirán
4   * algunos de los métodos de esta clase para crear un recorrido
5   * específico.
6   */
7  public abstract class RecorridoEuler<E, R> {
8    protected ArbolBinario<E> arbol;
9    /** Ejecución del recorrido. Este método abstracto deberá ser
10   * especificado en las subclases concretas. */
11   public abstract R ejecutar(ArbolBinario<E> T);
12   /** Inicialización del recorrido */
13   protected void inicializar(ArbolBinario<E> T) { arbol = T; }
14   /** Método plantilla */
15   protected R recorridoEuler(Posicion<E> v) {
16     ResultadoRecorrido<R> r = new ResultadoRecorrido<R>();
17     visitarIzquierda(v, r);
18     if (arbol.hasLeft(v))
19       r.izq = recorridoEuler(arbol.left(v));  // recorrido recursivo
20     visitarAbajo(v, r);
21     if (arbol.hasRight(v))
22       r.der = recorridoEuler(arbol.right(v)); // recorrido recursivo
```

```
23      visitarDerecha(v, r);
24      return r.salida;
25    }
26    // Métodos auxiliares que pueden ser redefinidos por las subclases
27    /** Método llamado para visitar a la izquierda */
28    protected void visitarIzquierda(Posicion<E> v, ResultadoRecorrido<R> r) {}
29    /** Método llamado para visitar abajo */
30    protected void visitarAbajo(Posicion<E> v, ResultadoRecorrido<R> r) {}
31    /** Método llamado para visitar a la derecha */
32    protected void visitarDerecha(Posicion<E> v, ResultadoRecorrido<R> r) {}
33
34    /* Clase interna para modelar el resultado del recorrido */
35    public class ResultadoRecorrido<R> {
36      public R izq;
37      public R der;
38      public R salida;
39    }
40  }
```

Listado 6.13: Clase `RecorridoEuler.java` que define un recorrido Euleriano genérico de un árbol binario

La clase, `RecorridoEuler`, por si misma no hace ningún cálculo útil. Sin embargo, se puede se puede extender y reemplazar los métodos auxiliares vacíos para hacer cosas útiles. Se ilustra este concepto usando árboles de expresión aritmética, ver ejemplo 6.7. Se asume que un árbol de expresión aritmética tiene objetos de tipo `TerminoExpresion` en cada nodo. La clase `TerminoExpresion` tiene subclases `VariableExpresión` (para variables) y `OperadorExpresion` (para operadores). A su vez, la clase `OperadorExpresion` tiene subclases para los operadores aritméticos, tales como `OperadorSuma` y `OperadorMultiplicacion`. El método `valor` de `TerminoExpresion` es reemplazado por sus subclases. Para una variable, este regresa el valor de la variable. Para un operador, este regresa el resultado de aplicar el operador a sus operandos. Los operandos de un operador son puesto por el método `setOperandos` de `OperadorExpresion`. En los códigos de los listados 6.14, 6.15, 6.16 y 6.17 se muestran las clases anteriores.

```
1  /** Clase para un término (operador o variable de una expresión
2   * aritmética.
3   */
4  public class TerminoExpresion {
5    public Integer getValor() { return 0; }
6    public String toString() { return new String(""); }
7  }
```

Listado 6.14: Clase `TerminoExpresion.java` para una expresión

```
1  /** Clase para una variable de una expresión artimética. */
2  public class VariableExpresion extends TerminoExpresion {
3    protected Integer var;
4    public VariableExpresion(Integer x) { var = x; }
5    public void setVariable(Integer x) { var = x; }
6    public Integer getValor() { return  var; }
7    public String toString() { return var.toString(); }
8  }
```

Listado 6.15: Clase `VariableExpresion.java` para una variable

```
/** Clase para un operador de una expresión aritmética. */
public class OperadorExpresion extends TerminoExpresion {
  protected Integer primerOperando, segundoOperando;
  public void setOperandos(Integer x, Integer y) {
    primerOperando = x;
    segundoOperando = y;
  }
}
```

Listado 6.16: Clase `OperadorExpresion.java` para un operador genérico

```
/** Clase para el operador adición en una expresión aritmética. */
public class OperadorSuma extends OperadorExpresion {
  public Integer getValor() {
    // desencajonar y después encajonar
    return (primerOperando + segundoOperando);
  }
  public String toString() { return new String("+"); }
}
```

Listado 6.17: Clase `OperadorSuma.java` el operador suma

En los listados 6.18 y 6.19, se muestran las clases `RecorridoEvaluarExpresion` y `RecorridoImprimirExpresion`, especializando `RecorridoEuler`, que evalúa e imprime una expresión aritmética guardada en un árbol binario, respectivamente. La clase `RecorridoEvaluarExpresion` reemplaza el método auxiliar `visitarDerecha`(T, v, r) con el siguiente cálculo:

- Si v es un nodo externo, poner r.`out` igual al valor de la variable guardada en v;
- si no (v es un nodo interno), combinar r.`izq` y r.`der` con el operador guardado en v, y pone r.`salida` igual al resultado de la operación.

```
/** Calcular el valor de un árbol de expresión aritmética. */
public class RecorridoEvaluarExpresion
  extends RecorridoEuler<TerminoExpresion, Integer> {

  public Integer ejecutar(ArbolBinario<TerminoExpresion> T) {
    inicializar(T); // llamar al método de la superclase
    return recorridoEuler(arbol.root()); // regresar el valor de la expresión
  }

  protected void visitarDerecha(Posicion<TerminoExpresion> v,
      ResultadoRecorrido<Integer> r) {
    TerminoExpresion termino = v.elemento();
    if (arbol.isInternal(v)) {
      OperadorExpresion op = (OperadorExpresion) termino;
      op.setOperandos(r.izq, r.der);
    }
    r.salida = termino.getValor();
  }
}
```

Listado 6.18: Clase `RecorridoEvaluarExpresion.java` especializa `RecorridoEuler` para evaluar la expresión asociada con un árbol de expresión aritmética.

La clase `RecorridoImprimirExpresion` reemplaza los métodos `visitarIzquierda`, `visitarAbajo`, `visitarDerecha` siguiendo la aproximación de la versión pseudocódigo dada previamente.

```
1  /** Imprimir la expresión guardada en un árbol de expresión aritmética. */
2  public class RecorridoImprimirExpresion
3    extends RecorridoEuler<TerminoExpresion, String> {
4
5    public Integer ejecutar(ArbolBinario<TerminoExpresion> T) {
6      inicializar(T);
7      System.out.print("Expresión: ");
8      recorridoEuler(T.root());
9      System.out.println();
10     return null;  // nada que regresar
11   }
12
13   protected void visitarIzquierda(Posicion<TerminoExpresion> v,
14       ResultadoRecorrido<String> r) {
15     if (arbol.isInternal(v)) System.out.print("("); }
16
17   protected void visitarAbajo(Posicion<TerminoExpresion> v,
18       ResultadoRecorrido<String> r) {
19     System.out.print(v.elemento()); }
20
21   protected void visitarDerecha(Posicion<TerminoExpresion> v,
22       ResultadoRecorrido<String> r) {
23     if (arbol.isInternal(v)) System.out.print(")"); }
24 }
```

Listado 6.19: Clase `RecorridoImprimirExpresion.java` especializa `RecorridoEuler` para imprimir la expresión asociada con un árbol de expresión aritmética.

Bibliografía

[1] R. Lafore, *Data Structures & Algorithms*, Second Edition, Sams, 2003.

[2] M. Goodrich, R. Tamassia, *Data Structures and Algorithms in Java*, Fourth Edition, John Wiley & Sons, 2005.

Índice alfabético