

ESTRUCTURAS DE DATOS CON JAVA MODERNO

Comportamiento + objetos = programas

Canek Peláez

TEMAS DE COMPUTACIÓN



Estructuras de datos con Java moderno

*Comportamiento + objetos =
programas*

Canek Peláez

Esta obra fue financiada por el proyecto PAPIME **PE100418**.

Estructuras de datos con Java moderno.

1^a edición electrónica, 5 de septiembre de 2018.

© D.R. 2018. Universidad Nacional Autónoma de México.

Facultad de Ciencias, Ciudad Universitaria, Delegación Coyoacán,
C.P. 04510, Ciudad de México.

editoriales@ciencias.unam.mx

<https://tienda.fciencias.unam.mx/>

ISBN: 978-607-30-0966-9

Prohibida la reproducción parcial o total de la obra por cualquier medio sin la autorización por escrito del titular de los derechos patrimoniales.

Impreso y hecho en México.

Índice general

Índice de figuras

[Índice de listados](#)

Agradecimientos

[Requisitos para este libro](#)

[Acerca del idioma](#)

1. [Introducción](#)

2. [Genéricos](#)

2.1. [Tipos genéricos](#)

2.2. [Borradura de tipos](#)

2.3. [Acotamiento de tipos](#)

2.4. [Empacamiento y desempacamiento](#)

2.5. [Los genéricos en este libro](#)

2.6. [Los genéricos en este libro](#)

3. [Iteradores](#)

3.1. [El operador *for-each*](#)

3.2. [El operador *for-each*](#)

4. Colecciones

4.1. None

5. [Listas](#)

5.1. [Definición de listas](#)

5.2. [Algoritmos para listas](#)

5.3. [Iteradores para listas](#)

5.4. [Iteradores para listas](#)

6. [Complejidad computacional](#)

6.1. [La notación de \$O\$ grandeza](#)

6.2. [Complejidades en tiempo y en espacio](#)

6.3. [Complejidades en tiempo y en espacio](#)

7. [Arreglos](#)

7.1. [El polinomio de redireccionamiento](#)

7.2. [Arreglos y genéricos](#)

7.3. [Arreglos y genéricos](#)

8. [Pilas y colas](#)

8.1. [Algoritmos de la clase abstracta](#)

8.2. [Algoritmos para pilas](#)

8.3. [Algoritmos para colas](#)

8.4. [Pilas y colas en el resto del libro](#)

8.5. [Pilas y colas en el resto del libro](#)

9. [Lambdas](#)

9.1. [Lambdas y funciones de primera clase](#)

9.2. [Clases internas anónimas](#)

9.3. [Lambdas en Java con interfaces funcionales](#)

9.4. [Alcance de variables](#)

9.5. [Alcance de variables](#)

10. [Ordenamientos](#)

10.1. [Ordenamientos en arreglos](#)

10.1.1. [Algoritmo SELECTIONSORT](#)

10.1.2. [Algoritmo QUICKSORT](#)

10.1.3.

[Manteniendo arreglos ordenados](#)

10.2. [Ordenamientos en listas](#)

10.2.1. [Algoritmo MERGESORT](#)

10.2.2. [Manteniendo listas ordenadas](#)

10.3. [La razón para ordenar colecciones](#)

10.4. [La razón para ordenar colecciones](#)

11. [Búsquedas](#)

11.1. [Búsquedas en arreglos](#)

11.2. [Búsquedas en listas](#)

11.3. [Búsquedas en listas](#)

12. [Árboles binarios](#)

12.1. [Definición de árboles binarios](#)

12.2. [Propiedades de árboles binarios](#)

12.3. [Implementación en Java](#)

12.4. [Algoritmos para árboles binarios](#)

12.5. [Aprovechando referencias no utilizadas](#)

12.6. [Aprovechando referencias no utilizadas](#)

13. [Árboles binarios completos](#)

13.1. [Definición de árboles binarios completos](#)

13.2. [Recorriendo árboles por amplitud](#)

13.2.1. [Acciones para vértices de árboles binarios](#)

13.3. [Algoritmos para árboles binarios completos](#)

13.4. [Algoritmos para árboles binarios completos](#)

14. [Árboles binarios ordenados](#)

14.1. [Definición de árboles binarios ordenados](#)

14.2. [Recorriendo árboles por profundidad](#)

14.2.1. [DFS *pre-order*](#)

14.2.2. [DFS *post-order*](#)

14.2.3. [DFS *in-order*](#)

14.3. [Algoritmos para árboles binarios ordenados](#)

14.4. [Complejidades en tiempo y en espacio](#)

14.5. [Complejidades en tiempo y en espacio](#)

15. [Árboles rojinegros](#)

15.1. [Definición de árboles rojinegros](#)

15.2. [Algoritmos de los árboles rojinegros](#)

15.2.1. [Algoritmo para agregar](#)

15.2.2. [Algoritmo para eliminar](#)

15.3. [Usos de árboles rojinegros](#)

15.4. [Usos de árboles rojinegros](#)

16. [Árboles AVL](#)

16.1. [Definición de árboles AVL](#)

16.2. [Algoritmos de los árboles AVL](#)

16.2.1. [Algoritmo de rebalanceo](#)

16.3. [Algoritmos de los árboles AVL](#)

17. [Gráficas](#)

17.1. [Definición de gráficas](#)

17.2. [Propiedades de gráficas](#)

17.3. [Implementación en Java](#)

17.4. [Recorridos en gráficas](#)

17.4.1. [BFS en gráficas](#)

17.4.2. [DFS en gráficas](#)

17.5. [Algoritmos para gráficas](#)

17.6. [Implementaciones alternativas de gráficas](#)

17.7. [Implementaciones alternativas de gráficas](#)

18. [Montículos mínimos](#)

18.1. [Definición de montículos mínimos](#)

18.2. [Acomodando hacia arriba y hacia abajo](#)

18.2.1. [Acomodando hacia arriba](#)

18.2.2.

[Acomodando hacia abajo](#)

18.3. [Implementación en Java](#)

18.4. [Algoritmos para montículos mínimos](#)

18.5. [Algoritmo HEAPSORT](#)

18.6. [Montículos de arreglos](#)

18.7. [Montículos de arreglos](#)

19. [Algoritmo de Dijkstra](#)

19.1. [Definición de trayectoria de peso mínimo](#)

19.2. [Implementación en Java](#)

19.3. [Algoritmos para gráficas ponderadas](#)

19.4. [Algoritmo de trayectoria mínima](#)

19.5. [Algoritmo de Dijkstra](#)

19.6. [Reconstruyendo trayectorias](#)

19.7. [Pesos con matrices de adyacencias](#)

19.8. [Pesos con matrices de adyacencias](#)

20. [Funciones de dispersión](#)

20.1. [Colisiones en funciones de dispersión](#)

20.2. [Implementación en Java](#)

20.2.1. [Cascando huevos](#)

20.3. [Función de dispersión XOR](#)

20.4. [Función de dispersión Bob Jenkins](#)

20.5. [Función de dispersión de Daniel J. Bernstein](#)

20.6. [Funciones de dispersión para diccionarios](#)

20.7. [Funciones de dispersión para diccionarios](#)

21. [Diccionarios](#)

21.1. [El arreglo del diccionario](#)

21.2. [Implementación en Java](#)

21.3. [Algoritmos para diccionarios](#)

21.4. [Complejidades en tiempo y en espacio](#)

21.5. [Implementaciones alternas de diccionarios](#)

21.6. [Implementaciones alternas de diccionarios](#)

22. [Conjuntos](#)

22.1. [Implementación en Java](#)

22.2. [Algoritmos para conjuntos](#)

22.3. [Otros usos de conjuntos](#)

22.4. [Otros usos de conjuntos](#)

23. [Mejorando gráficas](#)

23.1. [Modificaciones al código](#)

23.2. [Modificaciones al código](#)

24. [Conclusiones](#)

Bibliografía

Índice alfabético

Índice de figuras

- [1.1](#) La memoria utilizada por un estudiante en FORTRAN 56, suponiendo una arquitectura *big-endian*. El entero de 64 bits 0x00000000009A7010 es la dirección de memoria de la cadena con el nombre.
- [2.1](#) Advertencia del compilador de Java al tratar de compilar el código del listado [2.12](#) con la lista declarada como en el listado [2.9](#).
- [5.1](#) Lista del supermercado.
- [5.2](#) Visualización de una lista con múltiples elementos.
- [5.3](#) Los cambios al agregar un elemento al final de una lista no vacía. Las referencias punteadas son reemplazadas por las dos referencias oscuras.
- [5.4](#) Los cambios al eliminar la cabeza de una lista.
- [5.5](#) Los cambios al eliminar el rabo de una lista.
- [5.6](#) Los cambios al eliminar un nodo intermedio de una lista.
- [5.7](#) Lista de enteros.
- [5.8](#) Los cambios al agregar un elemento al inicio de una lista no vacía. Las referencias punteadas son reemplazadas por las dos referencias oscuras.
- [5.9](#) Los cambios al insertar un elemento en la i -ésima posición de la lista. Las referencias punteadas son reemplazadas por las dos referencias oscuras.
- [5.10](#) Visualización de un iterador i al inicio una lista.
- [5.11](#) El iterador i de la figura [5.10](#) después de que llama a su método `next()`.
- [5.12](#) El iterador i de la figura [5.11](#) al final de la lista.
- [5.13](#) El iterador i de la figura [5.12](#) después de mandar llamar el método `previous()`.
- [5.14](#) El iterador i antes (en gris y con líneas punteadas) y después (en negro y líneas continuas) de ejecutar `next()` cuando sí hay

un elemento siguiente.

[5.15](#) El iterador i antes (en gris y con líneas punteadas) y después (en negro y líneas continuas) de ejecutar `previous()` cuando sí hay un elemento anterior.

[6.1](#) La función $2n^3$ acota por arriba a $3n^3 + 40n^2 + n$ si $n \geq 1$.

[7.1](#) Un arreglo bidimensional, conceptualmente.

[7.2](#) Un arreglo bidimensional en memoria.

[7.3](#) El arreglo A de $3 \times 2 \times 4$, conceptualmente.

[7.4](#) El arreglo A de $3 \times 2 \times 4$, como realmente está en memoria.

[7.5](#) La (posible) memoria de un arreglo escalonado.

[8.1](#) Una cola después de agregar a, b, c, d, e, f, g, h , sacar dos elementos y agregar i, j, k, l ; el primero que entra es el primero que sale.

[8.2](#) Una pila después de agregar a, b, c, d, e , sacar dos elementos y agregar f, g, h ; el primero que entra es el último que sale.

[8.3](#) Los cambios al sacar.

[8.4](#) Los cambios al meter un elemento cuando la pila no es vacía.

[8.5](#) Los cambios al meter un elemento cuando la cola no es vacía.

[9.1](#) Orden parcial definido para $\wp(\{1, 2, 3\})$.

[10.1](#) En el i -ésimo paso, el algoritmo `SELECTIONSORT` comienza con el subarreglo $A[0 : i - 1]$ ordenado y con todos sus elementos siendo menores o iguales que los elementos en el subarreglo $A[i : n - 1]$; se busca el mínimo en $A[i : n - 1]$ (usando un índice j) e intercambia el mínimo con el i -ésimo.

[10.2](#) El algoritmo `QUICKSORT` elige un pivote p en el arreglo y deja todos los elementos en $A[0 : i - 1]$ menores o iguales a p ; todos los elementos en $A[i + 1 : n - 1]$ mayores a p ; y a p lo deja en $A[i]$ (el índice i varía dependiendo del valor del pivote). Por último continúa recursivamente en los subarreglos $A[0 : i - 1]$ y $A[i + 1 : n - 1]$.

[10.3](#) Análisis de movimientos de los índices i y j bajo la suposición de que los subarreglos se dividen siempre a la

mitad. El total de $n \log_2 n$ justifica una complejidad en tiempo de $O(n \log n)$.

[10.4](#) Análisis de movimientos de los índices i y j bajo la suposición de que el pivote siempre termina en el índice a . El total de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ justifica una complejidad en tiempo de $O(n^2)$.

[10.5](#) El algoritmo MEZCLA, como su nombre indica, mezcla dos listas ordenadas en una lista ordenada. En este paso del algoritmo, el elemento 4 está a punto de agregarse a la lista mezclada porque $4 \leq 5$; inmediatamente después, el iterador i se moverá a su siguiente.

[10.6](#) El algoritmo MERGESORT, divide las listas hasta tener listas de longitud 0 o 1 y por lo tanto ordenadas. Después las va mezclando par a par para reconstruir la lista ordenada.

[11.1](#) Los subarreglos a los que QUICKSORT entra recursivamente al buscar el elemento 17. El subarreglo $A[9 : 11]$ es el último, porque en este paso recursivo $A[m]=17$.

[12.1](#) Árbol binario.

[12.2](#) Árbol binario lleno.

[12.3](#) Coordenadas en un árbol binario. Este árbol tiene hoyos: los vértices en las coordenadas $(2, 2)$, $(2, 3)$, $(4, 3)$ y $(5, 3)$ no existen, pero sí hay otros vértices en los niveles 2 y 3.

[12.4](#) Árbol binario para representar en cadena.

[12.5](#) Representación en cadena del árbol de la figura [12.4](#).

[12.6](#) Árbol binario que no desperdicia las referencias de sus hojas.

[13.1](#) Árbol binario completo: todos los niveles están llenos excepto el último y en éste todos los hoyos están juntos al extremo derecho.

[13.2](#) Recorriendo un árbol binario completo por BFS.

[14.1](#) Árbol binario ordenado: todo vértice es mayor o igual que los vértices de su subárbol izquierdo y menor o igual que los vértices de su subárbol derecho.

Recorriendo un árbol binario ordenado en orden DFS *pre*-

[14.2](#) *order.* Los vértices se recorren en el orden 8, 4, 2, 1, 3, 6, 5, 7, 12, 10, 9, 11, 14, 13, 15.

[14.3](#) Recorriendo un árbol binario ordenado por DFS *post-order.* Los vértices se recorren en el orden 1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8.

[14.4](#) Recorriendo un árbol binario ordenado por DFS *in-order.* Los vértices se recorren en orden.

[14.5](#) Eliminamos la hoja con el elemento 5; esto consiste en desconectarla de su padre.

[14.6](#) Eliminamos el vértice que tiene el elemento 6; como tiene un único hijo esto consiste en “subir” su al mismo para que lo reemplace.

[14.7](#) Para eliminar el elemento 4 del árbol primero buscamos el vértice máximo de su subárbol izquierdo, que es el vértice que contiene a 3. Intercambiamos los elementos y como el vértice que era máximo no tiene hijo derecho podemos eliminarlo fácilmente.

[14.8](#) Giro sobre el vértice q a la derecha y sobre el vértice p a la izquierda.

[14.9](#) Resultado de agregar los elementos 1, 2, 3 y 4 (en ese orden) a un árbol binario ordenado.

[14.10](#) Si giramos el árbol a la derecha sobre su vértice 2, reducimos su altura y el número de hoyos que tiene.

[14.11](#) Árbol binario balanceado con $c = 2$.

[14.12](#) Árbol binario ordenado.

[15.1](#) Las dos ramas mostradas del vértice 3 tienen el mismo número de vértices **negros**, incluyendo los vértices \emptyset .

[15.2](#) Árbol rojinegro; los vértices 5 y 7 cumplen la propiedad 4 por las dos “hojas” \emptyset que cada uno tiene.

[15.3](#) **Caso 3:** coloreamos p y t de **negro**, a de **rojo** y hacemos recursión sobre a .

[15.4](#) **Caso 4:** giramos sobre p en su dirección y actualizamos v a p y p a v .

[15.5](#) **Caso 5:** coloreamos p de **negro**, a de **rojo** y giramos sobre a

en la dirección contraria de v .

15.6 **Caso 2:** coloreamos p de **rojo**, h de **negro** y giramos sobre p en la dirección de v .

15.7 **Caso 3:** coloreamos h de **rojo** y hacemos recursión sobre p .

15.8 **Caso 4:** coloreamos h de **rojo** y p de **negro**.

15.9 **Caso 5:** coloreamos h de **rojo**, a h_d de **negro** y giramos sobre h a la izquierda. No sabemos el color de p .

15.10 **Caso 6:** coloreamos h con el mismo color de p , a p de **negro**, a h_i de **negro** y giramos sobre p a la derecha. No sabemos el color de p ni de h_d .

15.11 Árbol rojinegro.

16.1 Árbol AVL. Cada vértice muestra h/b , donde h es la altura del vértice y b es su balance.

16.2 Al girar sobre q a la derecha o sobre p a la izquierda, los únicos vértices que pueden cambiar altura o balance son justamente p y q (y sus antecesores); los vértices en los subárboles A , B y C (que pueden ser vacíos) no cambian ni su altura ni su balance.

16.3 Si v tiene un balance de -2 y q tiene un balance de 1 , entonces giramos sobre q a la derecha, lo que deja al vértice x con un balance de -2 o -1 .

16.4 Si v tiene un balance de -2 y q tiene un balance de -2 , -1 o 0 , entonces giramos sobre v a la izquierda, lo que deja al subárbol $T(q)$ balanceado.

16.5 Si v tiene un balance de 2 y p tiene un balance de -1 , entonces giramos sobre p a la izquierda, lo que deja al vértice y con un balance de 0 , 1 o 2 .

16.6 Si v tiene un balance de 2 y p tiene un balance de 0 , 1 o 2 , entonces giramos sobre v a la derecha, lo que deja al subárbol $T(p)$ balanceado.

16.7 Árbol AVL.

17.1 Gráfica $G = (V = \{a, b, c, d, e, f, g, h, i, j\}, E = \{(a, c), (a, d), (a, e), (a, g), (a, i), (a, j), (b, c), (b, d), (b, e), (b, f), (b, g), (b, h), (b, i), (b, j), (c, d), (c, e), (c, g), (c, h), (c, j), (d, g), (e, f), (f, g), (g, h), (h, i), (i, j)\})$

$(d, h), (d, j), (e, f), (e, g), (e, i), (e, j), (f, g), (f, h), (f, i), (f, j), (g, i), (g, j), (h, i), (h, j)\}$.

17.2 El camino h, d, b, j, g, a, i, e en G , que también es trayectoria.

17.3 La subgráfica $H = (\{a, b, c, d, e, f, g, h, i\}, \{(a, e), (a, g), (a, i), (a, j), (b, c), (b, d), (b, f), (b, h), (b, j), (c, d), (c, j), (d, h), (e, i), (f, g), (f, h), (f, i), (f, j), (g, i), (g, j), (h, i)\})$ es plana.

17.4 Recorrido BFS comenzando por h .

17.5 Árbol generado por el recorrido BFS comenzando por h ; curveamos la arista (d, a) para que no cruzara otra arista del árbol.

17.6 Recorrido DFS comenzando por h .

17.7 Árbol generado por el recorrido DFS comenzando por h .

17.8 Gráfica K_3 .

17.9 Representación en cadena de una gráfica.

18.1 Montículo mínimo; cada vértice es menor o igual que sus vértices izquierdo y derecho.

18.2 El arreglo correspondiente al montículo mínimo de la figura 18.1.

18.3 El vértice que tenía al 13 cambió su valor a 0; tenemos que acomodarlo hacia arriba (*heapify-up*).

18.4 Intercambiamos el valor 0 con el de su padre mientras hay padre y sea mayor que 0; en este caso terminamos cuando 0 es la raíz.

18.5 Reacomodando hacia arriba el valor 0 en el arreglo.

18.6 El vértice que tenía al vértice 1 cambió su valor a 14; tenemos que acomodarlo hacia abajo (*heapify-down*).

18.7 Reacomodando hacia abajo el valor 14 en el arreglo.

18.8 Reacomodando hacia abajo el valor 14 en el arreglo.

18.9 Montículo mínimo.

19.1 Gráfica con pesos en las aristas.

19.2 Dos trayectorias distintas entre los

vértices h y e ; las trayectorias tienen pesos diferentes.

[19.3](#) La trayectoria mínima de una gráfica; en este caso es la única.

[19.4](#) Trayectoria de peso mínimo de una gráfica.

[20.1](#) Juntando 4 bytes en un entero de 32 bits usando *big-endian* y *little-endian*.

[20.2](#) La función de dispersión XOR.

[20.3](#) Realizando la operación $a \leftarrow b$.

[20.4](#) Realizando la operación $b \leftarrow (a \ll 11)$.

[20.5](#) La operación de mezcla de la función de dispersión de Bob Jenkins.

[20.6](#) Función de dispersión de Bob Jenkins: mientras haya 12 bytes disponibles en el arreglo, se crean 3 enteros con ellos y se suman a los valores a , b y c ; después se mezclan.

[20.7](#) Función de dispersión de Bob Jenkins: cuando haya menos de 12 bytes, aumentamos c en n , creamos tanto como podamos de 3 enteros con los bytes que queden y se suman a los valores a , b y c ; después se mezclan.

[21.1](#) Arreglo de listas de entradas en un diccionario.

[21.2](#) Representación en cadena de un diccionario.

[22.1](#) Representación en cadena de un conjunto.

[24.1](#) Tiempos para ordenar hasta 1,000,000 de elementos aleatorios.

[24.2](#) Tiempos para buscar hasta en 1,000,000 enteros aleatorios.

[24.3](#) Tiempos para llenar hasta con hasta 1,000,000 enteros en tablas de dispersión y conjuntos.

Índice de listados

- [1.1](#) Estructura para estudiantes en C.
- [1.2](#) Usando la estructura `Estudiante`.
 - [1.3](#) Funciones de la estructura `Estudiante`.
 - [1.4](#) Usando la estructura `Estudiante` con funciones.
 - [2.1](#) Esqueleto de una clase para listas.
 - [2.2](#) Agregando elementos a una lista.
 - [2.3](#) Asignación inválida; el método `getPrimero()` regresa una referencia de tipo `Object`, no `String`.
- [2.4](#) Asignación válida; hacemos una audición al método.
- [2.5](#) Confirmamos el tipo del objeto antes de hacer la audición.
 - [2.6](#) La clase `Lista` del listado [2.1](#) permite la creación de listas heterogéneas.
- [2.7](#) Asignación inválida en tiempo de ejecución; el primer elemento es una referencia de tipo `Lista`, pero el compilador no lo sabe y entonces lo permite.
- [2.8](#) Esqueleto de una clase para listas genéricas.
- [2.9](#) Instanciando un objeto de una clase genérica.
 - [2.10](#) Instanciando un objeto de una clase genérica con el tipo genérico siendo inferido.
- [2.11](#) Agregando elementos a una lista genérica.
- [2.12](#) Código inválido: el compilador no permite agregar a `lista` referencias que no sean del tipo `String`, porque así está declarada en el listado [2.9](#).
- [2.13](#) No es necesario usar una audición porque el compilador sabe que el método `getPrimero()` de `lista` regresa una instancia de `String`, porque así está declarada en el listado [2.9](#).
- [2.14](#) Método genérico.
- [2.15](#) Código genérico.

- [2.16](#) Código genérico después de la borradura de tipos.
- [2.17](#) El método `f()` recibe una lista de objetos.
- [2.18](#) Código inválido: tratamos de pasar una lista de cadenas al método `f()` del listado [2.17](#).
- [2.19](#) Si `lista` es una lista de objetos, entonces es válido agregarle instancias de `Integer`, porque cualquier instancia de `Integer` es también instancia de `Object`.
- [2.20](#) Método `f()` genérico.
- [2.21](#) Si obtenemos una instancia de `T` (de alguna manera), podemos agregarlo a `lista`.
- [2.22](#) Código inválido: nunca se puede instanciar un tipo genérico.
- [2.23](#) El método `promedio()` es genérico acotado a `Number`.
- [2.24](#) Clase genérica acotada a la interfaz `Comparable<T>`.
- [2.25](#) El método `promedio()` no está acotado, pero el parámetro `lista` está acotado a `Number` con un comodín.
- [2.26](#) Ejemplo de cómo sumar (ineficientemente) una lista de enteros.
- [2.27](#) Ejemplo de cómo sumar (ineficientemente) una lista de enteros usando empacamiento y desempacamiento.
- [3.1](#) Método que calcula (ineficientemente) el promedio de una lista de enteros.
- [3.2](#) Esqueleto de la clase `Lista`.
- [3.3](#) Interfaz `Iterable`.
- [3.4](#) Interfaz `Iterator` simplificada.
- [3.5](#) Clase `Lista` iterable.
- [3.6](#) Método que calcula *eficientemente* el promedio de una lista de enteros, usando el iterador de la clase `Lista`.
- [3.7](#) Método que calcula el promedio de una lista de enteros usando *for-each*.
- [4.1](#) La interfaz `colección`.
- [5.1](#) Declaración de la clase `Lista`.

[5.2](#) Firma del método `agregaInicio()`.

[5.3](#) Firma del método `agregaFinal()`.

[5.4](#) Firma del método `inserta()`.

[5.5](#) Firma del método `getLongitud()`.

[5.6](#) Firma de los métodos `eliminaPrimero()` y `eliminaUltimo()`.

[5.7](#) Firma de los métodos `getPrimero()` y `getUltimo()`.

[5.8](#) Firma del método `copia()`.

[5.9](#) Firma del método `reversa()`.

[5.10](#) Firma del método `get()`.

[5.11](#) Firma del método `indiceDe()`.

[5.12](#) Firma de los métodos `equals()` y `toString()`.

[5.13](#) Comportamiento de la clase `Lista`.

[5.14](#) Clase `Nodo`.

[5.15](#) Esqueleto de la clase `Lista` casi completo.

[5.16](#) Interfaz `IteradorLista`.

[5.17](#) Firma del método `iteradorLista()`.

[5.18](#) Esqueleto de la clase `Lista`.

[5.19](#) Código para imprimir los elementos de una lista, dentro de la clase `Lista`.

[5.20](#) Código para imprimir recursivamente los elementos de una lista, dentro de la clase `Lista`.

[5.21](#) Método `contiene()`, primera versión.

[5.22](#) Método `contiene()`, versión más concisa.

[5.23](#) Método `iterator()`.

[5.24](#) El inicio del método `equals()`, suprimiendo la advertencia relacionada con genéricos.

[5.25](#) Método `iteradorLista()`.

- 7.1 El arreglo de cadenas `A` se crea con `new` y se llena explícitamente; el arreglo `B` se crea y se llena implícitamente.
- 7.2 El arreglo bidimensional `A` se crea con `new` y se llena explícitamente; el arreglo `B` se crea y se llena implícitamente.
- 7.3 Arreglo de enteros `A` de $3 \times 2 \times 4$.
- 7.4 El arreglo de cadenas `A` se crea con `new` y se llena explícitamente; el arreglo `B` se crea y se llena implícitamente.
- 7.5 No podemos instanciar `T`, porque podría ser una interfaz o clase abstracta.
- 7.6 Tampoco podemos instanciar un arreglo de `T`, por razones históricas.
- 8.1 Comportamiento de la clase `Pila`.
- 8.2 Comportamiento de la clase `MeteSaca`.
- 8.3 Esqueleto de la clase `MeteSaca`.
- 8.4 Esqueleto de la clase `Pila`.
- 8.5 Esqueleto de la clase `cola`.
- 8.6 El inicio del método `equals()`, verificando la igualdad de clases y suprimiendo la advertencia relacionada con genéricos.
- 9.1 Firma para un método destructivo `ordena()` de la clase `Lista`.
- 9.2 Firma para un método `ordena()` no destructivo de la clase `Lista`.
- 9.3 Interfaz `Comparable` simplificada.
- 9.4 Cómo implementa la clase `Integer` la interfaz `Comparable`.
- 9.5 Método `ordena()` estático genérico acotado a `Comparable<T>`.
- 9.6 Invocación de `ordena()` si no fuera estático.
- 9.7 Invocación de `ordena()` estático.
- 9.8 Método inválido: no podemos pasar un método como argumento de otro método.
- 9.9 En Python sí podemos pasar una función como argumento de otra función o método. Este ejemplo sólo funciona en Python

- 2 por cambios en las listas en la versión 3 del lenguaje.
- [9.10](#) Otro ejemplo de pasar una función como argumento de otra función.
- [9.11](#) No definimos una función de comparación para números de cuenta; sencillamente pasamos el *código* al método `sort()` de las listas de Python.
- [9.12](#) Función que recibe como parámetro un apuntador a funciones.
- [9.13](#) Definiendo el tipo para funciones de comparación.
- [9.14](#) Usamos nuestra función de ordenamiento con una función de comparación.
- [9.15](#) Interfaz `Comparator` simplificada.
- [9.16](#) Método `ordena()` no genérico, que utiliza un `Comparator`.
- [9.17](#) Clase comparadora que compara estudiantes por número de cuenta.
- [9.18](#) Usando la clase `ComparaEstudiantesPorNúmeroDeCuenta` para ordenar estudiantes.
- [9.19](#) Con una clase interna y anónima podemos ahorrarnos la declaración de la clase `ComparaEstudiantesPorNúmeroDeCuenta`.
- [9.20](#) Usamos el método `ordena()` con una lambda.
- [9.21](#) Interfaz `X`.
- [9.22](#) Interfaz `Y`.
- [9.23](#) El polimorfismo de Java evita que podamos usar lambdas porque hay dos métodos `ejemplo()` que reciben argumentos no diferenciables a partir de sus parámetros.
- [9.24](#) Eliminamos la ambigüedad al definir el tipo de la lambda explícitamente.
- [9.25](#) Usamos una lambda para imprimir un mensaje al hacer click en un botón.
- [9.26](#) No podemos incrementar `cont` porque deja de existir cuando el método termina.
- [9.27](#) Sí podemos usar la variable `m` en la lambda, porque sólo la leemos.

- [9.28](#) La variable local `e` se puede usar en la lambda porque no se modifica, aunque el objeto al que refiere sí. Noten que cambiamos el nombre del parámetro de la lambda para que no tapara a la variable local.
- [9.29](#) Código válido: la variable `cont` es de clase y su alcance se extiende automáticamente a la lambda.
- [10.1](#) Firmas de los métodos para ordenar listas.
- [10.2](#) Método `mergeSort()` no estático.
- [10.3](#) Esqueleto de la clase `Arreglos` con la implementación de los métodos acotados.
- [11.1](#) Firmas de los métodos para realizar búsquedas binarias en arreglos.
- [11.2](#) Firmas de los métodos para realizar búsquedas lineales en listas.
- [12.1](#) La interfaz `VerticeArbolBinario`.
- [12.2](#) Esqueleto de la clase `ArbolBinario`.
- [12.3](#) Inicio del método `equals()` en `Vertice`.
- [12.4](#) El inicio del método `equals()` de `ArbolBinario`.
- [13.1](#) Esqueleto de la clase `ArbolBinarioCompleto`.
- [13.2](#) La interfaz `AccionVerticeArbolBinario`.
- [13.3](#) Imprimiendo los elementos de un árbol binario completo en BFS.
- [13.4](#) Imprimiendo los elementos de un árbol binario completo con sus iteradores.
- [13.5](#) Imprimiendo los elementos de un árbol binario completo en orden BFS.
- [14.1](#) Esqueleto de la clase `ArbolBinarioOrdenado`.
- [15.1](#) Esqueleto de la clase `ArbolRojoNegro`. Mostramos los métodos `giraIzquierda()` y `giraDerecha()` ya invalidados.
- [15.2](#) Método `equals()` de `VerticeRojoNegro`.
- [15.3](#) El método `nuevoVertice()` de `ArbolRojoNegro`.
- [15.4](#) Método auxiliar seguro para verificar si un vértice

es rojo.

16.1 Esqueleto de la clase `ArbolAVL`. Mostramos los métodos `giraIzquierda()` y `giraDerecha()` ya invalidados.

16.2 Método `equals()` de `VerticeAVL`.

16.3 El método `nuevoVertice()` de `ArbolAVL`.

17.1 Esqueleto de la clase `Grafica`, con el método `iterator()` ya escrito.

17.2 La firma del método `vecinos()`.

17.3 Inicio del método `equals()` en `Grafica`.

18.1 La interfaz `comparableIndexable`.

18.2 La clase `MonticuloMinimo` acotada a `Comparable` e `Indexable`.

18.3 El método `nuevoArreglo()`.

18.4 La interfaz `MonticuloDijkstra`.

18.5 Esqueleto de la clase `MonticuloMinimo` con los métodos `nuevoArreglo()` y `iterator()` ya implementados.

18.6 Inicio del método `equals()` en `MonticuloMinimo`.

18.7 Esqueleto de la clase `MonticuloArreglo`.

19.1 Método `get()` de la clase `Vecino`.

19.2 Método `setColor()` antes de tener gráficas ponderadas.

19.3 Método `setColor()` para gráficas ponderadas.

19.4 Recorriendo la lista de vecinos cuando son instancias de `Vertice`.

19.5 Recorriendo la lista de vecinos cuando son instancias de `Vecino`.

19.6 Esqueleto de la clase `Grafica` con pesos en las aristas.

20.1 Esqueleto de la clase `Dispersores`, donde estarán las implementaciones de bajo nivel de nuestros dispersores.

20.2 La interfaz `Dispersor`.

20.3 La clase `FabricaDispersores`.

Combinamos cuatro bytes en un entero de 32 bits en el

[20.4](#) esquema *big-endian* (suponiendo que `a` es el primer byte leído).

[20.5](#) Combinamos cuatro bytes en un entero de 32 bits en el esquema *big-endian*.

[20.6](#) Función de dispersión DJB2 en C.

[21.1](#) Un arreglo mapea enteros a objetos.

[21.2](#) El método auxiliar `nuevoArreglo()`.

[21.3](#) Creando el dispersor por omisión con una lambda.

[21.4](#) Esqueleto de la clase `diccionario`.

[21.5](#) Inicio del método `equals()` en `Diccionario`.

[22.1](#) Esqueleto de la clase `conjunto`.

Agradecimientos

Este libro es el resultado de varios años consecutivos impartiendo el curso de Estructuras de Datos en la carrera de Ciencias de la Computación en la Facultad de Ciencias de la Universidad Nacional Autónoma de México. El código, orden de los temas y presentación de los mismos evolucionaron a través de estos años, retroalimentándose de mi propia experiencia como estudiante de la carrera y programador profesional, así como de los (muchas veces muy atinados) comentarios de alumnos, ayudantes, profesores y colegas que me hicieron el favor de revisar distintas partes del manuscrito. Es por ello que, aunque el único autor del libro sea yo, utilizamos el plural de la primera persona a lo largo del mismo.

Como casi cualquier tarea no trivial, este libro está construido con aportaciones directas e indirectas de muchas personas, tantas que ni siquiera trataré de enumerarlas a todas. Sin embargo sí quiero mencionar en particular al Comité Editorial de la Facultad de Ciencias, así como a los árbitros que designó; al Programa de Apoyo para la Innovación y Mejoramiento de la Enseñanza (proyecto PAPIME PE100418) por apoyar financieramente la publicación del libro; y al doctor Omar Antolín por ofrecer la lectura más detallada del manuscrito por parte de alguien que no tenía ninguna obligación de leerlo.

Por supuesto, todo error u omisión en el texto es responsabilidad mía exclusivamente.

Ciudad de México, septiembre de 2018.

Requisitos para este libro

En casi todos los programas de Ciencias de la Computación y similares en distintas universidades del mundo, el curso de Estructuras de Datos se imparte después de un curso inicial de programación: siguiendo esa idea, este libro supone que el lector maneja los preceptos básicos de programar. Qué es un algoritmo, representación interna (complemento a 2, números de punto flotante, UTF-8), estructuras de control, recursión, la pila (*stack*) de ejecución, el espacio de memoria (*heap*), paso de parámetros por valor y por referencia, alcance de variables, etcétera, serán conceptos que el libro no cubre y que da por hecho que el lector maneja.

En particular el libro supone que el lector está familiarizado con el lenguaje de programación Java, por lo que la sintaxis del lenguaje y funcionamiento del compilador se considerarán también vistos. Las bases de la Orientación a Objetos (qué son clases, objetos, métodos, propiedades, herencia, polimorfismo, despacho dinámico, excepciones) también se espera que el lector las maneje, al menos de manera básica.

El manejar genéricos, iteradores y lambdas es conveniente, pero no necesario; el libro cubrirá estos temas con cierto detalle.

Por último y aunque no fundamental, será de ayuda para el lector que al menos conozca las definiciones básicas o haya oído hablar de conceptos como son las máquinas de Turing, el cálculo- λ , el principio de sustitución de Liskov y el problema del paro.

Acerca del idioma

La lengua universal de la computación (y en general del mundo científico) es el inglés, nos guste o no este hecho. Es fundamental que un computólogo (y de hecho cualquier programador) entienda y sea fluido en el idioma inglés; y lo más común es que el código que se escribe en la industria pública y privada en todo el mundo esté en inglés, en el sentido de que el nombre de casi cualquier identificador estará en ese idioma.

Habiendo dicho eso, este libro utilizará español para todos sus identificadores; clases, métodos, variables y demás estarán en el idioma español. Las razones son varias, pero ninguna es realmente importante desde el punto de vista técnico o pedagógico; es más bien que nos parece simpático y que hace al código resultante ciertamente poco común.

Java desde sus primeras versiones soporta identificadores con acentos, diéresis y eñes; incluso los soporta en alfabetos distintos al latino (árabe, griego y cirílico, por ejemplo) y también con ideogramas chinos y japoneses. Pero esto puede presentar problemas con ciertos editores para programar y en particular con el paquete de listados de código que empleamos en \LaTeX .

Por este motivo no utilizaremos acentos, diéresis o eñes en nuestros identificadores; el nombre de un método será `getultimo()`, no `getÚltimo()`. Esto resultará en nombres de variables y métodos que podrían parecer extraños; por ejemplo, nunca tendremos una variable de clase llamada `año` y no reemplazaremos la eñe con una ene, así que habrá que usar la imaginación para encontrar un nombre alternativo.

También podrá resultar algo extraño leer cosas como `public class Lista`, que mezclan el español y el inglés; pero como decimos arriba, ciertamente hará casi único al código.

Fuera del código el libro utilizará exclusivamente español para discutir los temas cubiertos en el mismo, aunque se mencionará al menos una vez (generalmente entre paréntesis) el término en inglés correspondiente más comúnmente usado. Hay una única excepción a esto: los términos *big-endian* y *little-endian* (revisados en el capítulo 20) son básicamente intraducibles (nos negamos a utilizar *ancho-extremista* y *estrecho-extremista*) y son utilizados tal cual en casi toda la literatura en español. Probablemente, como ocurrió con *scanner* y *escáner*, los términos deban sencillamente ser incorporados al idioma.

Otro término que realmente no cuenta con una traducción satisfactoria es

casting. Cuando existe una *conversión* de tipos se puede utilizar este término, por ejemplo al hacer `int a = (int)3.14`; sin embargo en varios lenguajes de programación (y en Java en particular) hacer *casting* en referencias es algo conceptual y prácticamente distinto: cuando uno hace `Integer n = (Integer)o` uno no realiza ninguna conversión o transformación, sólo cambia cómo el programa *interpreta* al objeto. El objeto sigue siendo el mismo.

Podríamos advocar entonces por integrar *casting* al español, como con *little-endian* y *big-endian*, pero para *casting* preferimos proponer un término que no es la traducción literal, pero que está relacionado y que nos parece es adecuado.

El término *casting* se utiliza para describir el proceso de *moldar* o *enyesar* algo; darle *forma* a un objeto físico. Probablemente es la acepción que se quería expresar cuando originalmente se empezó a usar para la conversión de tipos; podemos *moldar* un número de punto flotante a un entero, le damos *forma* de entero.

Pero el término también se utiliza para describir el proceso de selección del elenco de una obra o película. Esta acepción (intencional o no) encaja perfectamente con la Orientación a Objetos, donde los objetos podemos verlos como actores que interpretan distintos papeles a lo largo de la ejecución de un programa (la “obra”). En el ejemplo de arriba, el mismo objeto puede interpretar el papel de un `Object` y el de un `Integer`.

Siguiendo esta segunda acepción proponemos el término ***audición*** como traducción no literal del término *casting*; utilizando una vez más el ejemplo de arriba, le hacemos una *audición* a la referencia `o` para ver si puede interpretar el papel de `Integer`. E incluso la analogía sirve cuando el proceso falla; si la excepción `ClassCastException` es lanzada, podemos decir que el objeto falló o no pasó la audición.

Nos gustaría ver utilizado este término en la literatura, en lugar de seguir viendo las ocurrencias de *casting* o el uso de *conversión* incluso cuando no hay tal.

1. Introducción

Las Ciencias de la Computación lidian con todo aquello relacionado con algoritmos; su diseño, análisis, mejoramiento, implementación y evolución. Todo algoritmo puede ser diseñado utilizando una máquina de Turing o cualquier otro modelo matemático de cómputo; de manera similar, todo algoritmo puede ser implementado en el lenguaje de máquina de alguna arquitectura, ya sea que ésta exista en hardware, esté implementada virtualmente en software o incluso si es ficticia. Siempre y cuando dicho lenguaje de máquina sea Turing-completo (en otras palabras, que pueda computar lo mismo que una máquina de Turing), por supuesto.

Al inicio de la programación práctica (cuando de hecho ya existían computadoras), los programadores trabajaban así, porque no había otra manera de hacerlo. Rápidamente se hizo aparente que trabajar de esta manera es lento, propenso a errores y (probablemente lo más importante) dificulta la reutilización de código.

Para mejorar esto el primer paso fue la creación de los lenguajes de ensamblador, que son un mapeo directo del lenguaje de máquina a algo que se asemeja más al lenguaje natural. Pero aunque esto ayuda, realmente trabajar en ensamblador es casi igual de lento, propenso a errores y con dificultades para reutilizar código que trabajar en lenguaje de máquina directamente, dado que en los hechos son el mismo lenguaje, nada más que el segundo se escribe exclusivamente con unos y ceros (o perforando tarjetas o cerrando circuitos en un tablero de conmutadores o *switchboard*).

El primer lenguaje de programación de alto nivel que fue usado masivamente, FORTRAN, fue otro paso en la dirección correcta. Como su nombre indica (FORTRAN viene de *FOR*mula *TRAN*slator) no hacía mucho más que traducir fórmulas; esto es, podía tomar la expresión $a+b*c$ y traducirla en las instrucciones del lenguaje de máquina que multiplicaban y sumaban los valores de las variables, tomando en cuenta además la precedencia de operadores para que la multiplicación se ejecutara primero. No había procedimientos, ni mucho menos funciones y el manejo de memoria era muy similar al que el mismo lenguaje de máquina (o ensamblador) proveía; básicamente se podía leer y escribir en localidades arbitrarias de memoria. Las únicas estructuras de control que existían eran **IF** y **GOTO**.

FORTRAN ayudó a disminuir el tiempo que se tardaba en escribir código, a reducir los errores en el mismo y a su reutilización; pero fue sólo un paso más en el largo camino que llevó el desarrollar lenguajes de programación

modernos.

El lenguaje de programación FORTRAN (y los subsecuentes que aparecieron después) alcanzaron dos hitos muy importantes en los años siguientes: la introducción de funciones (y el concepto de la pila de ejecución para preservar el estado local del programa al momento de mandar a llamarlas) y la capacidad de declarar estructuras. Hay que hacer énfasis en el hecho de que las estructuras de datos (de lo que trata este libro) y los lenguajes de programación estructurados están obviamente muy relacionados, pero *no son lo mismo*. Las estructuras en los lenguajes de programación estructurados nacieron sencillamente de la necesidad de (como su nombre indica) estructurar la memoria de los programas; las estructuras de datos como implementación de tipos de datos abstractos son un concepto que se vuelve natural al tener memoria estructurada, pero que nace después de que ésta ha sido implementada en lenguajes de programación.

Antes de tener memoria estructurada, como FORTRAN al inicio, lo único que se podía hacer era acceder localidades arbitrarias de memoria para lectura y escritura. Que es por supuesto la base para tener arreglos; si uno tiene la localidad de memoria A y uno sabe que un entero ocupa 4 bytes de memoria (como de hecho así ocurre en casi todas las arquitecturas modernas), entonces uno puede inferir que en la localidad de memoria A hay un entero, en la localidad de memoria $A + 4$ hay un segundo entero y que en la localidad de memoria $A + (4 * i)$ hay un i -ésimo entero; a esto se le conoce como un *desplazamiento*: podemos pensar que “brincamos” a la dirección A y luego nos *desplazamos* $4 * i$ localidades.

Hacer que un compilador traduzca $A[i]$ a la expresión $A + (k * i)$ (donde k es el tamaño en bytes del tipo del arreglo A) es muy sencillo. Lo único que hay que tener en cuenta es que un arreglo debe ser una región *contigua* de memoria para que esta fórmula funcione siempre.

Los arreglos son, por su simpleza, muy limitados; pero es relativamente sencillo extender el concepto a cosas un poco más complejas. Por ejemplo (y suponiendo que aún estamos usando un lenguaje como FORTRAN a sus inicios) vamos a suponer que trabajamos en un sistema para manejar estudiantes en una escuela. Los datos que queremos manejar de los estudiantes son su nombre, su número de cuenta, su promedio y su edad. El nombre estará en una cadena, así que necesitamos un apuntador a caracteres a la dirección de memoria de la misma (un apuntador es similar, pero mucho más primitivo, que una referencia en Java). Los apuntadores en arquitecturas modernas suelen utilizar 64 bits, 8 bytes. El número de cuenta lo pondremos en un entero, entonces de nuevo son 4 bytes. El promedio lo pondremos en un número de punto flotante de doble precisión, entonces esos son otros 8 bytes.

Y la edad la pondremos en un entero de 16 bits (un `short` en Java), entonces esos son otros 2 bytes.

Por lo tanto, si nos dicen en nuestro programa que en la localidad de memoria A está un estudiante, sabemos entonces que en la dirección A (o $A + 0$) está el apuntador a caracteres con la dirección de memoria del nombre; que en la dirección $A + 8$ está el entero con el número de cuenta; que en la dirección $A + 12$ está el doble con el promedio; y que en la dirección $A + 20$ está el entero de 16 bits con la edad (figura 1.1). Es lento y propenso a errores, pero así podemos trabajar en nuestro sistema de estudiantes.

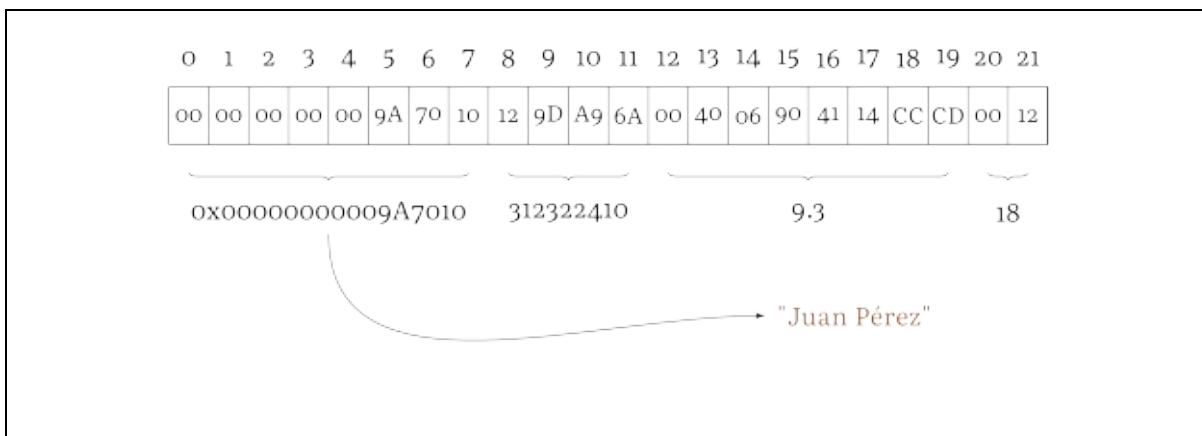


Figura 1.1: La memoria utilizada por un estudiante en FORTRAN 56, suponiendo una arquitectura *big-endian*. El entero de 64 bits 0x000000000009A7010 es la dirección de memoria de la cadena con el nombre.

Lo único que hicieron los lenguajes de programación estructurados es simplificar para el programador este concepto; en lugar de estar pensando en localidades de memoria y desplazamientos, uno puede pensar en un nivel más alto y considerar una estructura y sus miembros; por ejemplo en el lenguaje de programación C declararíamos una estructura `Estudiante` como se ve en el listado 1.1.

```
struct _Estudiante {
    char* nombre;
    int cuenta;
    double promedio;
    int16_t edad;
};
typedef struct _Estudiante Estudiante;
```

Listado 1.1: Estructura para estudiantes en C.

El `typedef` es únicamente para poder usar `Estudiante` directamente como tipo (como su nombre indica, `typedef` significa “definición de tipo”), en lugar de `struct _Estudiante`. Para utilizar una instancia de esta estructura, uno puede simplemente declararla y acceder a sus propiedades con el operador punto,

como se ve en el listado [1.2](#).

```
int
main(int argc, char* argv[])
{
    Estudiante e;
    e.nombre = "Juan Pérez";
    e.cuenta = 312322410;
    e.promedio = 9.3;
    e.edad = 18;
    printf("Nombre: %s\n", e.nombre);
    printf("Cuenta: %d\n", e.cuenta);
    printf("Promedio: %.2f\n", e.promedio);
    printf("Edad: %d\n", e.edad);
    return 0;
}
```

Listado 1.2: Usando la estructura `Estudiante`.

Para este código (que compila usando `gcc`, si incluimos los encabezados `stdint.h` para `int16_t` y `stdio.h` para `printf()`) lo que hace el compilador es traducir `e.promedio` por la dirección de memoria de la instancia de la estructura (en este caso en la pila de ejecución o *stack*), desplazada 12 bytes, porque `nombre` es un apuntador a caracteres y eso ocupa 8 bytes, `cuenta` es un entero y eso ocupa 4 bytes: $8 + 4 = 12$; y `nombre` y `cuenta` están declaradas antes que `promedio` en la estructura. Por supuesto el compilador hace otras cosas; no permite que asignemos a `e.cuenta` algo distinto a un entero si no le hacemos una conversión de tipos, por ejemplo.

Con esto ya podemos trabajar a un nivel bastante alto en comparación a lo que teníamos anteriormente; pero todo se hace mucho más elegante si además utilizamos funciones. Por ejemplo, podríamos declarar las funciones del listado [1.3](#).

```
Estudiante*
estudiante_new(const char* nombre, int cuenta,
               double promedio, int16_t edad)
{
    Estudiante* e =
        (Estudiante*)malloc(sizeof(Estudiante));
    e->nombre = strdup(nombre);
    e->cuenta = cuenta;
    e->promedio = promedio;
    e->edad = edad;
    return e;
}
```

```

char*  

estudiante_get_nombre(Estudiante* e)  

{  

    return e->nombre;  

}  

int  

estudiante_get_cuenta(Estudiante* e)  

{  

    return e->cuenta;  

}

```

Listado 1.3: Funciones de la estructura **Estudiante**.

Y funciones similares `estudiante_get_promedio()` y `estudiante_get_edad()`. La función `estudiante_new()` asigna memoria para un estudiante (`sizeof` nos permite obtener el tamaño en bytes de una estructura) y sencillamente inicializa las propiedades de la estructura con los parámetros que recibe (la cadena del nombre se duplica). Ahora `e` es un apuntador a `Estudiante` y al usar `malloc()` hacemos que su dirección esté en el espacio de memoria (*heap*) del programa, contrario a la pila de ejecución como antes. Como `e` es un apuntador, utilizamos el operador flecha (`->`) en lugar de punto para acceder a sus propiedades.

Nuestro programita podría ahora reescribirse como en el listado [1.4](#).

```

int  

main(int argc, char* argv[])
{
    Estudiante* e = estudiante_new("Juan Pérez",
                                    312322410,
                                    9.3, 18);

    printf("Nombre: %s\n",
            estudiante_get_nombre(e));
    printf("Cuenta: %d\n",
            estudiante_get_cuenta(e));
    printf("Promedio: %2.2f\n",
            estudiante_get_promedio(e));
    printf("Edad: %d\n",
            estudiante_get_edad(e));
    return 0;
}

```

Listado 1.4: Usando la estructura **Estudiante** con funciones.

Tener nuestra estructura y funciones asociadas a ella nos permite agrupar fácilmente todas sus operaciones relacionadas en un API (*Application Programming Interface*, interfaz de programación de aplicaciones), que

podemos poner en una biblioteca y entonces reutilizar el código de `Estudiante` en muchos otros lugares.

Esto por supuesto es lo que dio pie a la Orientación a Objetos a finales de los años setentas del siglo pasado; `e = estudiante_new(...)` se convirtió en `e = new Estudiante(...)` y `estudiante_get_nombre(e)` se convirtió en `e.get_nombre()`. Lo único que faltaba era una manera de proteger (encapsular) los miembros de las estructuras, para que sólo las funciones relacionadas pudieran tocarlos y (mucho más importante y complicado) agregar herencia, que es la técnica por excelencia para reutilización masiva de código.

Sin embargo esto fue más adelante, como decimos arriba a finales de la década de los setenta en el siglo XX. Antes de eso, con lenguajes de programación estructurados ya disponibles, la gente que se especializaba en diseñar, analizar e implementar algoritmos se dieron cuenta de que podían definir entes matemáticos que denominaron tipos de datos abstractos (*abstract data types* o ADTs en inglés), que tenían ciertas propiedades y algoritmos asociados con comportamientos y complejidades computacionales bien establecidos. Estos tipos de datos abstractos se podían (y pueden) implementar fácilmente en un lenguaje de programación estructurado (como C, Pascal, versiones de Fortran a partir de 1966) en lo que se denominó como *estructuras de datos* o *data structures* en inglés. Los tipos abstractos de datos son un modelo teórico (propuesto por primera vez por Barbara Liskov en 1974 [\[24\]](#)) y las estructuras de datos son su implementación concreta.

En diseño y análisis de algoritmos teórico, utilizar tipos de datos abstractos permite la creación de nuevos algoritmos de manera mucho más sencilla, porque uno puede reutilizar un ADT del cual ya se conocen sus complejidades computacionales y entonces (al igual que en implementación práctica de algoritmos) un investigador ya no tiene por qué estar “inventando la rueda” para subproblemas que ya están resueltos.

En el lado práctico, el tener una biblioteca de estructuras de datos, una vez más con complejidades computacionales ya bien conocidas y estudiadas, permite a los programadores el reutilizar código y concentrarse en resolver otras partes de un problema, sin tener que preocuparse por los bloques básicos del manejo de datos.

En 1976 Niklaus Wirth publicó un libro seminal en la enseñanza de estructuras de datos, *Algorithms + Data Structures = Programs* [\[33\]](#). Esto traducido a conceptos modernos de lenguajes de programación podría escribirse como *Comportamiento + Objetos = Programas*; o (si dejamos de intentar ser sutiles) *Uno Debería Escribir Programas Utilizando Orientación a Objetos*.

Y lo que ocurre es los tipos de datos abstractos son realmente clases y sus instancias son realmente objetos; sólo que cuando los inventaron los computólogos de entonces no tenían disponible este concepto todavía.

De manera intencional estamos dejando de lado el paradigma funcional y todos los lenguajes descendientes de IPL y LISP, así como la historia de los mismos; pero esto no es porque no los consideremos importantes. Todo lo contrario: son fundamentales en la formación de cualquier buen programador y varios de los conceptos de los mismos han permeado a todos los lenguajes de programación modernos (incluyendo Java, que a partir de la versión 8 del lenguaje incluye soporte para lambdas —o algo que se les asemeja mucho— y que se cubrirán en este libro). Es nada más que obviamente en un libro de Estructuras de Datos, el paradigma estructurado y la Orientación a Objetos (su descendiente directo) juegan un papel mucho más importante.

El libro de Wirth de 1976 fijó un estándar en cómo enseñar un curso de Estructuras de Datos que ha tomado mucho tiempo en evolucionar; en varias universidades del mundo el curso de Estructuras de Datos se sigue enseñando básicamente con el paradigma estructurado, incluso si el lenguaje utilizado en el curso es Orientado a Objetos. En los últimos años esto ha venido cambiando y los conceptos de Orientación a Objetos se han ido incorporando a la enseñanza de las Estructuras de Datos (un buen ejemplo de consulta es [\[17\]](#)).

Este libro sigue esta tendencia y trata de tomar un enfoque moderno a la enseñanza de Estructuras de Datos, utilizando las características modernas del lenguaje de programación Java a partir de su versión 8. No sólo se cubren todas las estructuras de datos clásicas que suelen cubrirse en este tipo de cursos, junto con todos los algoritmos relacionados y respetando sus complejidades computacionales en tiempo y en espacio; se hace esto siguiendo un diseño Orientado a Objetos (incluyendo encapsulamiento de datos, herencia, interfaces, etcétera) y utilizando todas las características modernas de Java 8 (como son genéricos, iteradores, lambdas, etcétera).

También hacemos énfasis en el hecho de que este libro está pensado para un curso de Estructuras de Datos *práctico*; el material relacionado a las estructuras de datos cubiertas no está únicamente presentado para que el alumno las estudie: se espera que el estudiante escriba la implementación final de las estructuras y pueda comprobar por sí mismo las diferencias en complejidades en tiempo y espacio de los distintos algoritmos implementados, incluyendo las diferencias reales que existen cuando se toma en cuenta la constante oculta por la notación de O grandota.

Relacionado al punto anterior, recalcamos que el libro es para un curso de Estructuras de Datos, no de Análisis de Algoritmos. Por lo tanto y aunque se

cubrirán las complejidades computacionales en tiempo y en espacio de todos y cada uno de los algoritmos vistos en el libro (exceptuando aquellos muy triviales), dicha cobertura será más bien informal; y aunque siempre habrá una justificación detallada de por qué un algoritmo tiene tal o cual complejidad en tiempo y espacio, la demostración matemática formal correspondiente será generalmente omitida. Nos parece más importante para un curso de Estructuras de Datos que los alumnos las implementen, a que vean las demostraciones de sus algoritmos asociados: para eso existen los cursos de Análisis de Algoritmos.

El capítulo [2](#) ofrece una introducción a los tipos genéricos en Java, ya que todas las estructuras de datos en el libro serán genéricas. Se hace lo mismo con la implementación de iteradores que ofrece el lenguaje en el capítulo [3](#), así como la estructura de control *for-each*. La interfaz [colección](#), que es utilizada por casi todas las estructuras de datos del libro, se ve en el capítulo [4](#).

En el capítulo [5](#) se cubren listas doblemente ligadas. El capítulo [6](#) explica los fundamentos de la complejidad computacional y de la notación de O grandota. Arreglos se ven en el capítulo [7](#) y en el capítulo [8](#) se cubren pilas y colas.

La implementación de Java para lambdas se explica en el capítulo [9](#) y serán utilizadas en el capítulo [10](#) para ordenar listas de objetos que no implementen la interfaz [comparable](#), además de ver varios algoritmos de ordenamiento. Las búsquedas en listas y arreglos serán cubiertas en el capítulo [11](#).

Los capítulos [12](#), [13](#) y [14](#) cubren árboles binarios, árboles binarios completos y árboles binarios ordenados respectivamente; árboles rojinegros y AVL se cubren en los capítulos [15](#) y [16](#) respectivamente.

Gráficas implementadas con listas de adyacencias se cubren en el capítulo [17](#). En el capítulo [18](#) se ven montículos mínimos, que son utilizados en el capítulo [19](#) para implementar el algoritmo de Dijkstra en las gráficas del capítulo [17](#) después de agregar pesos a las aristas en su implementación.

En el capítulo [20](#) se ven funciones de dispersión, que se utilizan para implementar diccionarios en el capítulo [21](#). Con diccionarios se implementan conjuntos en el capítulo [22](#) y en el capítulo [23](#) se mejora la implementación de gráficas de los capítulos [17](#) y [19](#), cambiando listas de adyacencias por diccionarios de adyacencias.

Por último en el capítulo [24](#) se presentan algunas conclusiones de todo el material cubierto por el libro.

2. Genéricos

Casi todas las estructuras de datos que se cubren en este libro son colecciones; con esto queremos decir que son agrupaciones de elementos donde están permitidas las repeticiones y por lo tanto no necesariamente se comportan como conjuntos.

Muchas veces ocurrirá que los algoritmos y el comportamiento de las estructuras de datos vistas no dependerá de características particulares de los elementos contenidos en ellas, sino únicamente de propiedades generales; por ejemplo, varias de nuestras estructuras de datos requerirán que los elementos que contengan sean comparables entre ellos, pero esto no quiere decir que las estructuras sepan cómo exactamente se realizan esas comparaciones.

Esto implica que, exceptuando algunas características muy generales, a nuestras estructuras de datos no les importará exactamente qué tipo tengan los elementos contenidos en ellas. En Java, que tiene una jerarquía de herencia con una única raíz (`Object`, de la cual todas las otras clases de Java heredan directa o indirectamente), esto suele implementarse utilizando a `Object` como el tipo con el que trabaja una colección.

Por ejemplo, una clase `Lista` podría estar definida de la siguiente manera (listado 2.1). Esto funciona (y de hecho, hasta la versión 5 del lenguaje, era la única manera de tener este tipo de colecciones en Java), pero tiene dos grandes desventajas. La primera es que fuerza al programador a utilizar audições (*castings*) al momento de obtener elementos de la colección; y la segunda es que permite que nuestras colecciones sean heterogéneas.

```
public class Lista {  
    /* ... */  
    public int getLongitud() { /* ... */ }  
    public void agregaFinal(Object elemento) { /* ... */ }  
    public void agregaInicio(Object elemento) { /* ... */ }  
    public void elimina(Object elemento) { /* ... */ }  
    public Object getPrimero() { /* ... */ }  
    public Object getUltimo() { /* ... */ }  
    public Object get(int i) { /* ... */ }  
    /* ... */  
}
```

Listado 2.1: Esqueleto de una clase para listas.

Para exemplificar el primer problema; podemos agregar elementos a instancias

de nuestra clase `Lista` sin problemas (listado 2.2).

```
Lista lista = new Lista();
lista.agregaFinal("una cadena");
lista.agregaFinal("otra cadena");
lista.agregaInicio("cadena");
```

Listado 2.2: Agregando elementos a una lista.

Esto es porque cualquier objeto en Java es instancia de la clase `Object` y como los métodos que reciben nuevos elementos en la colección reciben referencias a objetos de tipo `Object`, cualquier variable que le pasemos (siempre y cuando no sea de un tipo básico) será válida.

El obtener elementos de la colección en cambio es problemático; el código en el listado 2.3 no compila, suponiendo que siguiera al código del listado 2.2.

```
String s = lista.getPrimero();
```

Listado 2.3: Asignación inválida; el método `getPrimero()` regresa una referencia de tipo `Object`, no `String`.

Aunque nosotros sabemos que el primer elemento de la lista es una cadena, *el compilador no lo sabe*. Más aún, en general *no puede* saberlo; el problema de tratar de determinar en tiempo de compilación cuál será el tipo de un objeto arbitrario en tiempo de ejecución se puede reducir al problema del paro¹.

Entonces para poder obtener la primera cadena de nuestra lista, tenemos que ayudarle al compilador siendo explícitos al decir que la referencia que estamos recibiendo es de hecho a un objeto de la clase `String`; el código en el listado 2.4 sí compila:

```
String s = (String)lista.getPrimero();
```

Listado 2.4: Asignación válida; hacemos una audición al método.

Si quisiéramos ser precavidos o no estuviéramos 100% seguros de que el primer elemento en la lista es una cadena, podemos preguntar por el tipo del objeto antes de intentar hacer la audición, para garantizar que en tiempo de ejecución el programa no falle (listado 2.5).

```
Object o = lista.getPrimero();
if (o instanceof String)
    String s = (String)o;
```

Listado 2.5: Confirmamos el tipo del objeto antes de hacer la audición.

Lo cual por supuesto a largo plazo se vuelve engorroso y propenso a errores, además de que es redundante porque en general nos gustaría que nuestras listas fueran siempre *homogéneas*; es decir, que todos sus elementos fueran del mismo tipo. O (más apegado a la Orientación a Objetos y el principio de sustitución de Liskov²) que nos pudieran garantizar el tener todos el mismo comportamiento.

Como mencionábamos arriba, así declaradas nuestras colecciones pueden ser *heterogéneas*; si cualquier objeto en Java es instancia de `Object`, entonces podemos agregar a una misma lista lo que sea que queramos (listado 2.6).

```
lista.agregaFinal(new Integer(23));
lista.agregaFinal(new Double(3.14));
lista.agregaInicio(new Lista());
```

Listado 2.6: La clase `Lista` del listado 2.1 permite la creación de listas heterogéneas.

Con la clase `Lista` declarada como lo hicimos en el listado 2.1, el código del listado 2.6 compila y se ejecuta correctamente. Eso en sí mismo ya es malo; pero es peor aún que el código del listado 2.7 también compila correctamente.

```
String s = (String)lista.getPrimero();
```

Listado 2.7: Asignación inválida en tiempo de ejecución; el primer elemento es una referencia de tipo `Lista`, pero el compilador no lo sabe y entonces lo permite.

Aunque el código del listado 2.7 compila correctamente, va a lanzar una excepción cuando tratemos de ejecutarlo (`ClassCastException`), porque el primer elemento en la lista es otra lista. Una vez más, el compilador *no puede* determinar el tipo de una referencia arbitraria en tiempo de compilación.

En algunas situaciones muy específicas puede ser justificable el utilizar una colección heterogénea; sin embargo (como podemos ver en el ejemplo de arriba), en casi todos los casos lo que se aconseja es tener siempre colecciones homogéneas donde todos los elementos de las mismas tengan el mismo comportamiento.

Estos son los dos problemas principales que tiene el utilizar directamente a `Object` como el tipo de los elementos en nuestras colecciones y ambos problemas son los que primordialmente resuelven los genéricos en Java.

2.1. Tipos genéricos

En Java una clase se define genérica (*generic* en inglés) especificando un parámetro genérico entre los símbolos < y >; por ejemplo, nuestra clase `Lista`

de arriba la podríamos hacer genérica como se ve en el listado 2.8.

```
public class Lista<T> {
    /* ... */
    public int getLongitud() { /* ... */ }
    public void agregaFinal(T elemento) { /* ... */ }
    public void agregaInicio(T elemento) { /* ... */ }
    public void elimina(T elemento) { /* ... */ }
    public T getPrimero() { /* ... */ }
    public T getUltimo() { /* ... */ }
    public T get(int i) { /* ... */ }
    /* ... */
}
```

Listado 2.8: Esqueleto de una clase para listas genéricas.

El parámetro `T` puede llamarse como nosotros queramos, pero se acostumbra utilizar una única letra mayúscula, siendo `T` lo más comúnmente usado. Así declarada la clase `Lista` es genérica con `T` siendo su parámetro genérico. Cada instancia concreta de la clase `Lista` debe especificar (al momento de declaración e instanciación) el tipo de los elementos con los que trabajará, reemplazando `T` con dicho tipo, que puede ser cualquier tipo no básico de Java (clases e interfaces, básicamente). Para crear una lista ahora tenemos que especificar el tipo de sus elementos (listado 2.9).

```
Lista<String> lista = new Lista<String>();
```

Listado 2.9: Instanciando un objeto de una clase genérica.

El compilador de Java tiene una inferencia de tipos limitada, que podemos utilizar como se ve en el listado 2.10.

```
Lista<String> lista = new Lista<>();
```

Listado 2.10: Instanciando un objeto de una clase genérica con el tipo genérico siendo inferido.

Como la referencia `lista` está declarada como instancia de `Lista` de `String`, al instanciarla no es necesario especificar `String` de nuevo; el compilador lo *infiere* automáticamente. A `<>` (sin ningún tipo especificado dentro) se le conoce como el operador *diamante*. Por motivos didácticos y de claridad, en este libro no se usará esta inferencia de tipos, pero es perfectamente válido utilizarla.

Agregar cadenas a nuestra lista genérica no cambia en lo más mínimo (listado 2.11).

```
lista.agregaFinal("una cadena");
lista.agregaFinal("otra cadena");
lista.agregaInicio("cadena");
```

Listado 2.11: Agregando elementos a una lista genérica.

Sin embargo, ahora el compilador no nos va a permitir agregar a la lista elementos que no cumplan el comportamiento de la clase `String` (listado 2.12).

```
lista.agregaFinal(new Integer(23));
lista.agregaFinal(new Double(3.14));
lista.agregaInicio(new Lista());
```

Listado 2.12: Código inválido: el compilador no permite agregar a `lista` referencias que no sean del tipo `String`, porque así está declarada en el listado 2.9.

Ninguna de las líneas en listado 2.12 son válidas para el compilador, porque no estamos agregando elementos instancias de la clase `String` en ninguna de ellas. Esto nos permite mantener la homogeneidad de nuestras colecciones.

Más aún, nos evita el uso de audiciones porque ahora el compilador sí sabe el tipo de los elementos de la lista, dado que explícitamente la declaramos así (listado 2.13).

```
String s = lista.getPrimero();
```

Listado 2.13: No es necesario usar una audición porque el compilador sabe que el método `getPrimero()` de `lista` regresa una instancia de `String`, porque así está declarada en el listado 2.9.

El listado 2.13, contrario al listado 2.4, ahora sí es válido para el compilador y se ejecuta sin ningún problema.

La clase `Lista` así declarada se convierte en un tipo *genérico* de Java y el compilador espera que sea utilizada genéricamente todo el tiempo; en otras palabras, que siempre que la utilicemos especifiquemos explícitamente el tipo correspondiente a `T`.

Podemos utilizar la clase de forma no genérica, como en el listado 2.2; sin embargo el compilador nos dará una advertencia de que estamos utilizando un tipo genérico de manera no genérica (figura 2.1). La única razón por la que se permite es para que el código escrito con Java antes de la versión 5 pueda seguir compilándose con versiones modernas del compilador, pero ningún código nuevo debe utilizar nunca tipos genéricos de manera no genérica.

```
# javac UsoLista.java Lista.java
Note: UsoLista.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
# javac -Xlint:unchecked UsoLista.java Lista.java
```

```

UsoLista.java:18: warning: [unchecked] unchecked call to agregaFinal(T) as a
member of the raw type Lista
    lista.agregaFinal("una cadena");
           ^
  where T is a type-variable:
    T extends Object declared in class Lista
UsoLista.java:19: warning: [unchecked] unchecked call to agregaFinal(T) as a
member of the raw type Lista
    lista.agregaFinal("otra cadena");
           ^
  where T is a type-variable:
    T extends Object declared in class Lista
UsoLista.java:20: warning: [unchecked] unchecked call to agregaInicio(T) as a
member of the raw type Lista
    lista.agregaInicio("cadena");
           ^
  where T is a type-variable:
    T extends Object declared in class Lista
3 warnings

```

Figura 2.1: Advertencia del compilador de Java al tratar de compilar el código del listado [2.12](#) con la lista declarada como en el listado [2.9](#).

Además de permitir clases genéricas, Java también permite el tener métodos genéricos, como se ve en el listado [2.14](#).

```

public class UsoLista {
    public <T> T getIEsimo(Lista<T> lista, int i) {
        T t = lista.get(i);
        return t;
    }
}

```

Listado 2.14: Método genérico.

Este método genérico por ejemplo nos permitiría obtener el *i*-ésimo elemento de una lista genérica. La clase `UsoLista` *no* es genérica; únicamente su método `getIEsimo()` lo es; el parámetro genérico `T` sólo es válido dentro del método, no en toda la clase.

El compilador utiliza el parámetro genérico de la instancia de `Lista` recibida para determinar el parámetro genérico del método y dentro del mismo incluso podemos declarar variables (como `t`) con el mismo.

Este ejemplo es muy artificial; veremos más adelante un ejemplo más realista.

2.2. Borradura de tipos

Los genéricos son una característica del compilador de Java, no de la máquina virtual, utilizando una técnica conocida como *borradura de tipos* (*type erasure* en inglés). Veamos el código del listado [2.15](#).

```
Lista<String> lista = new Lista<String>();
lista.agregaFinal("una cadena");
lista.agregaFinal("otra cadena");
String s = lista.getPrimero();
```

Listado 2.15: Código genérico.

Lo que hace el compilador es eliminar (borrar) el tipo `String` y agregar adiciones donde sea necesario; el resultado es algo similar al listado [2.16](#).

```
Lista lista = new Lista();
lista.agregaFinal("una cadena");
lista.agregaFinal("otra cadena");
String s = (String)lista.getPrimero();
```

Listado 2.16: Código genérico después de la borrado de tipos.

Debe quedar claro que el compilador no hace una substitución textual; no convierte al código del listado [2.15](#) en el código del listado [2.16](#). Lo que hace es modificar lo que se conoce en compiladores como la representación interna del código del primer listado en el segundo; pero es más sencillo visualizarlo como si realmente convirtiera el código directamente.

El compilador de Java hace varias substituciones de representaciones internas; por simpleza a lo largo del libro diremos que el compilador traduce, substituye o convierte código de una forma a otra, pero debe quedar claro que es únicamente la representación interna del compilador lo que se cambia, no el código mismo.

De igual forma, el tipo `T` es visto como `Object` dentro de la clase `Lista` y en tiempo de ejecución la misma es tratada como un tipo no genérico; de hecho, en tiempo de ejecución *no existen los genéricos*. Los genéricos son únicamente una ayuda para que el compilador garantice la integridad de los tipos de nuestros datos.

Esto da pie a cosas que a primera vista pueden resultar anti intuitivas; por ejemplo, consideremos un método que recibe una lista de objetos (listado [2.17](#)).

```
public class Ejemplo {
    public void f(Lista<Object> lista) {
        /* ... */
    }
}
```

Listado 2.17: El método `f()` recibe una lista de objetos.

Uno podría considerar que utilizar el código del listado [2.18](#) es válido.

```
Lista<String> lista = new Lista<String>();
lista.agregaFinal("una cadena");
Ejemplo e = new Ejemplo();
e.f(lista);
```

Listado 2.18: Código inválido: tratamos de pasar una lista de cadenas al método `f()` del listado [2.17](#).

Dado que el método `f()` recibe una lista de objetos y `String` hereda de `Object`, se podría pensar que esto es válido; pero no lo es: el código del listado [2.18](#) no compila.

El “tipo” `Lista<String>` no “hereda” del “tipo” `Lista<Object>`, por el simple motivo de que no son dos tipos distintos: el único tipo es `Lista`, que es genérico para el compilador, *pero no para la máquina virtual*.

Si el último ejemplo fuera válido para el compilador, la implementación del método `f()` del listado [2.19](#) nos causaría problemas en tiempo de ejecución. El método `f()` es completamente válido; lo que es inválido es pasarle una lista de cualquier cosa que no sea `Object`.

```
public void f(Lista<Object> lista) {
    lista.agregaFinal(new Integer(5));
}
```

Listado 2.19: Si `lista` es una lista de objetos, entonces es válido agregarle instancias de `Integer`, porque cualquier instancia de `Integer` es también instancia de `Object`.

Si, dentro del método `f()`, `lista` es una `Lista` de instancias de `Object`, entonces es válido agregar un `Integer` a la misma, porque toda instancia de `Integer` es una instancia de `Object`. Es por este motivo que no podemos pasar una `Lista` de instancias de `String` al método; la lista podría dejar de ser homogénea.

Podemos reescribir el método para que sea genérico como se ve en el listado [2.20](#), si es que queremos poder pasarle una lista con cualquier tipo de elementos.

```
public <T> void f(Lista<T> lista) {
    T t = lista.getPrimero();
}
```

Listado 2.20: Método `f()` genérico.

Más aún, si podemos obtener un objeto de tipo `T`, entonces incluso podemos agregarlo a la lista (listado [2.21](#)).

```

public <T> void f(Lista<T> lista) {
    T t = /* Obtenemos un objeto de tipo T... */
    lista.agregaFinal(t);
}

```

Listado 2.21: Si obtenemos una instancia de `T` (de alguna manera), podemos agregarlo a `lista`.

Debe ser claro que no podemos crear (instanciar) objetos con un parámetro genérico; dado `T` un tipo genérico, el listado [2.22](#) nunca es válido y no puede compilarse.

```

T t = new T();

```

Listado 2.22: Código inválido: nunca se puede instanciar un tipo genérico.

Esto es porque `T` es reemplazada en tiempo de compilación por un tipo no básico de Java; esto incluye interfaces (que no son instanciables), clases abstractas (que no pueden instanciar objetos) y enumeraciones (cuyas únicas instancias son los elementos de la enumeración).

2.3. Acotamiento de tipos

Supongamos que queremos escribir un método que calcule el promedio de una lista de números, pero no queremos preocuparnos si la lista es exactamente de dobles, enteros o cualquier otro tipo numérico de Java. Para poder hacer esto utilizaremos acotamiento de tipos (listado [2.23](#)).

```

public <T extends Number> double
promedio(Lista<T> lista) {
    double r = 0.0;
    for (int i = 0; i < lista.getLongitud(); i++)
        /* Esto es muy ineficiente, pero no tenemos
        otra alternativa por el momento. */
        r += lista.get(i).doubleValue();
    return r / lista.getLongitud();
}

```

Listado 2.23: El método `promedio()` es genérico acotado a `Number`.

El método `promedio()` es genérico, pero contrario al ejemplo que se muestra en el listado [2.14](#), el parámetro genérico `T` está acotado a la clase `Number`, de la cual heredan todas las clase numéricas de Java, incluyendo `Integer` y `Double`. Por lo tanto el método puede recibir listas de cualquiera de esas clases.

El compilador verifica que sólo pasemos listas genéricas de esos tipos. Dentro

del método `promedio()` en tiempo de ejecución el tipo `T` es tratado como `Number`, de la misma manera que si `T` no estuviera acotado sería tratado como `Object`. Esto nos permite por ejemplo utilizar el método `doubleValue()`, de la clase `Number`.

Los problemas que mencionábamos respecto al listado [2.18](#) se siguen cumpliendo; mientras que en el método `promedio()` podemos *obtener* objetos del tipo `T` y tratarlos como instancias de `Number`, no podemos *pasar* objetos instancias de `Number` a métodos de la clase `Lista` y esperar que se comporten como `T`, por las mismas razones: para evitar agregar un `Double` a una lista de `Integer`. Sin embargo, sí podemos pasarle objetos instancia de `T`, si los conseguimos de alguna manera. El tipo `T` está acotado (puede ser cualquier tipo que extienda `Number`) en la *declaración* del método `promedio()`, pero queda fijo una vez que *usamos* el método. Ese tipo fijo sí lo podemos usar para pasar objetos a los métodos de `Lista`.

El acotamiento de tipos no sólo funciona para métodos genéricos; también funciona para clases genéricas; en el capítulo [14](#), la clase `ArbolBinarioOrdenado` será declarada como se ve en el listado [2.24](#).

```
public class ArbolBinarioOrdenado<T extends Comparable<T>>
    extends ArbolBinario<T> { /* ... */ }
```

Listado 2.24: Clase genérica acotada a la interfaz `Comparable<T>`.

A veces no nos interesa el tipo exacto de un parámetro; para estos casos podemos utilizar comodines (*wildcards* en inglés). Por ejemplo, podríamos reescribir el método `promedio()` como en el listado [2.25](#).

```
public double promedio(Lista<? extends Number> lista) {
    double r = 0.0;
    for (int i = 0; i < lista.getLongitud(); i++)
        /* De nuevo, esto es muy ineficiente. */
        r += lista.get(i).doubleValue();
    return r / lista.getLongitud();
}
```

Listado 2.25: El método `promedio()` no está acotado, pero el parámetro `lista` está acotado a `Number` con un comodín.

Esto funciona de forma muy similar a un método acotado; pero como no tenemos el tipo acotado, entonces no podemos declarar variables de ese tipo.

2.4. Empacamiento y desempacamiento

En este capítulo hemos explicado cómo usaríamos una lista genérica de

enteros de forma similar al del listado [2.26](#), por ejemplo para calcular su suma.

```
Lista<Integer> lista = new Lista<Integer>();
lista.agregaFinal(new Integer(1));
lista.agregaFinal(new Integer(2));
lista.agregaFinal(new Integer(3));
lista.agregaFinal(new Integer(4));
int suma = 0;
for (int i = 0; i < lista.getLongitud(); i++)
    /* De verdad, esto es muy ineficiente. */
    suma += lista.get(i).intValue();
```

Listado 2.26: Ejemplo de cómo sumar (ineficientemente) una lista de enteros.

Esto funciona, pero es muy engorroso: por suerte el compilador de Java ofrece una característica relacionada tangencialmente con genéricos que se conoce como empacamiento y desempacamiento automático o *automatic boxing and unboxing* en inglés (listado [2.27](#)).

```
Lista<Integer> lista = new Lista<Integer>();
lista.agregaFinal(1);
lista.agregaFinal(2);
lista.agregaFinal(3);
lista.agregaFinal(4);
int suma = 0;
for (int i = 0; i < lista.getLongitud(); i++)
    /* ¡Muy ineficiente! */
    suma += lista.get(i);
```

Listado 2.27: Ejemplo de cómo sumar (ineficientemente) una lista de enteros usando empacamiento y desempacamiento.

Básicamente como nuestra lista está declarada como una lista de enteros, lo que hace el compilador al ver que le pasamos un entero del tipo básico (`int`) al método `agregaFinal()`, es automáticamente envolver el tipo básico con un `new Integer()`. Análogamente, al ver que utilizamos el resultado del método `get(int)` como operando de una operación aritmética, automáticamente le agrega `.intValue()`.

En otras palabras, para el código del listado [2.26](#), el compilador lo convierte a la representación interna equivalente del código en el listado [2.27](#). Esto ocurre para todas las ocasiones en que utilizemos las clases que envuelven a los tipos básicos de Java: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` y `Boolean`; podemos utilizar literales y variables de los tipos básicos correspondientes y el compilador empacará (envolverá con `new Integer()`, por ejemplo) y desempacará (agregará `.intValue()`, por ejemplo) de forma automática.

Esto permite, en los hechos y en la gran mayoría de los casos, el utilizar los tipos básicos de Java como si fueran objetos. No siempre funciona; habrá ocasiones en que el compilador no podrá determinar qué empacamiento o desempacamiento debe utilizar y al igual que los genéricos es una característica del compilador, no de la máquina virtual. Las instrucciones para empacar y desempacar terminan siendo ejecutadas de cualquier manera, sólo el programador ya no tiene que escribirlas explícitamente.

2.5. Los genéricos en este libro

Los genéricos de Java son relativamente nuevos, habiendo sido integrados al lenguaje en su versión 1.5 [\[37\]](#) (o 5). Contrario a la mayor parte de los lenguajes de programación que ofrecen algo similar, en Java los genéricos sólo existen como una herramienta del compilador para ayudar a mantener la consistencia de tipos; en casi todos los demás lenguajes de programación con genéricos, los mismos ofrecen capacidades que se pueden utilizar (de maneras muy útiles en ciertos casos) en tiempo de ejecución.

Muchos programadores critican la implementación de genéricos en Java por esta y otras razones; por ejemplo que son Turing completos [\[18\]](#). Sin embargo también tienen sus ventajas (varias cubiertas en este capítulo) y son sin duda útiles. Además de que permitieron al lenguaje mantener compatibilidad con versiones anteriores del mismo.

Los genéricos son un tema muy extenso de Java y aquí sólo hemos cubierto los puntos más básicos del mismo. Sin embargo, hemos visto en general lo necesario para lo que utilizaremos en el resto del libro. Algunos temas de genéricos serán revisitados o profundizados cuando los utilicemos en capítulos más adelante.

Probablemente lo más importante de los genéricos, como programadores prácticos, es entender que son una herramienta que sólo existe en *tiempo de compilación*; al momento de ejecutar nuestro programa, los genéricos básicamente han dejado de existir.

Ejercicios

1. ¿Dónde se implementan los genéricos en Java; en el compilador o en la máquina virtual?
2. Menciona dos ventajas de utilizar genéricos para colecciones.
3. En Java un método que declare recibir una instancia de `Lista<Number>` no puede recibir una instancia declarada de `Lista<Integer>`. Explica por qué.

4. ¿Qué ventaja tiene que nuestras colecciones sean homogéneas, en lugar de heterogéneas?
 5. En una clase genérica cuyo parámetro genérico no acotado se llame `T`, ¿qué tipo sustituye a `T` dentro de la clase cuando el compilador realice la borradura de tipos?
-

1. El problema del paro (*Halting problem*, en inglés) es determinar, dado un programa p y su entrada x , si ejecutar el programa $p(x)$ termina (para). Este problema es en el caso general imposible de resolver; no es que sea difícil o complicado, sencillamente *no se puede resolver*.[←](#)
2. El principio de sustitución de Liskov [\[25\]](#) dice: sea $\phi(x)$ una propiedad demostrable de los objetos x de tipo T ; entonces $\phi(y)$ debe ser verdadera para los objetos y de tipo S donde S es un subtipo de T .[←](#)

3. Iteradores

En el capítulo anterior vimos que podríamos calcular el promedio de una lista de enteros como se ve en el listado 3.1.

```
public double promedio(Lista<Integer> lista) {
    double r = 0.0;
    for (int i = 0; i < lista.getLongitud(); i++)
        /* Terriblemente ineficiente. */
        r += lista.get(i);
    return r / lista.getLongitud();
}
```

Listado 3.1: Método que calcula (ineficientemente) el promedio de una lista de enteros.

En todos los ejemplos del estilo dejamos un comentario que hace énfasis en que es muy ineficiente recorrer una lista de esta manera. Esto es porque, en general y sin duda en particular en la implementación que veremos de listas, el acceder el i -ésimo elemento de una lista tomará i pasos, dado que siempre tendremos que empezar en el primer elemento e irnos moviendo por la lista elemento a elemento hasta llegar al i -ésimo.

Esto causa que acceder al elemento con índice 0 tome 0 pasos, que acceder al elemento con índice 1 tome 1 paso, etcétera, lo que resulta que recorrer la lista de esta manera tome el siguiente número de pasos:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}.$$

Para ponerlo en perspectiva, una lista con 1,000 elementos tomaría cerca de 500,000 pasos el recorrerla, lo cual es extremadamente ineficiente. Lo que lleva por supuesto a la pregunta, ¿cuál es la manera correcta de recorrer una lista?

La respuesta es que nos debe tomar un número constante de pasos el pasar de un elemento al siguiente de la lista. Para conseguir esto existen múltiples estrategias, pero la mayoría requieren tener acceso a las propiedades privadas de nuestra futura clase `Lista`; siguiendo los preceptos de la Orientación a Objetos esto se considera un mal diseño, además de que hay múltiples consecuencias negativas potenciales si lo permitimos.

Esto básicamente nos deja con dos posibles alternativas; una es el tener un comportamiento (en Java, un método) que nos permita decirle a la lista qué

acción o acciones queremos que se le realicen a cada elemento de la lista. Esto es factible y es de hecho lo preferido por lenguajes funcionales o con una componente funcional muy importante. Sin embargo presenta varias dificultades; a veces queremos realizar acciones sólo sobre ciertos elementos de la lista sin que se recorran todos o realizar acciones sobre grupos arbitrariamente grandes de elementos. Existen mecanismos para poder lidiar con estos casos al utilizar un comportamiento que aplique acciones a los elementos de una lista, pero se puede argumentar que es engorroso y ciertamente no es lo que han implementado la mayor parte de los lenguajes de programación primordialmente orientados a objetos.

En su lugar el consenso en Orientación a Objetos que se ha ido formando (desde finales del siglo XX) es que debe ser un actor *distinto* a la lista el que la recorre. Que es la segunda alternativa para recorrer una lista y la que nosotros utilizaremos no nada más para listas, sino para todas las colecciones en el libro. En el capítulo [17](#) veremos un ejemplo de cómo recorrer una colección usando un comportamiento que reciba una acción a realizar sobre los elementos de la misma.

En los ejemplos que hemos visto en estos primeros capítulos se ha mostrado el esqueleto de una clase `Lista` sin entrar en detalles de cómo realmente estaría implementada la misma ([listado 3.2](#)).

```
public class Lista<T> {
    /* ... */
    public int getLongitud() { /* ... */ }
    public void agregaFinal(T elemento) { /* ... */ }
    public void agregaInicio(T elemento) { /* ... */ }
    public void elimina(T elemento) { /* ... */ }
    public T getPrimero() { /* ... */ }
    public T getUltimo() { /* ... */ }
    public T get(int i) { /* ... */ }
    /* ... */
}
```

Listado 3.2: Esqueleto de la clase `Lista`.

En este ejemplo (que es como dejamos a la clase `Lista` en el capítulo anterior), vemos la declaración de una clase `Lista` genérica y las firmas de algunos de sus métodos (comportamiento), pero no se elabora cuáles serían sus variables de clase (propiedades) ni la implementación concreta de sus métodos (los algoritmos correspondientes).

Únicamente con esta información disponible podría parecer imposible el especificar cómo es que vamos a recorrer nuestras listas, pero es justamente donde entran los iteradores.

Siguiendo los preceptos de la Orientación a Objetos, nuestra clase `Lista` deberá respetar el encapsulamiento de datos y por lo tanto no importa que no sepamos exactamente cuáles serán sus variables de clase, porque de cualquier forma no vamos a poder utilizarlas fuera de la misma. Y como ya dijimos que la lista será recorrida por un actor distinto a ella, la única opción que nos queda (si no queremos violar el encapsulamiento de datos) es que la misma lista nos proporcione dicho actor.

Esto es exactamente lo que proporciona el patrón de diseño *Iterador* (*Iterator*): un actor que nos da la misma colección y que se encarga de iterarla. En Java esto se implementa con dos interfaces genéricas, la primera de las cuales es `Iterable` y que nos servirá para obtener una instancia de la segunda. Simplificando, la interfaz `Iterable` se define como en el listado 3.3.

```
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

Listado 3.3: Interfaz `Iterable`.

En otras palabras, todas las clases que implementan la interfaz `Iterable` lo que hacen es proporcionarnos una instancia de `Iterator`, que será el iterador para iterar los elementos de la clase iterable. La interfaz genérica `Iterator` está definida (de nuevo, simplificando) como se ve en el listado 3.4.

```
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
}
```

Listado 3.4: Interfaz `Iterator` simplificada.

Lo que haremos entonces con nuestra clase `Lista` es hacerla iterable al implementar la interfaz correspondiente (listado 3.5).

```
public class Lista<T> implements Iterable<T> {
    /* ... */
    public int getLongitud() { /* ... */ }
    public void agregaFinal(T elemento) { /* ... */ }
    public void agregaInicio(T elemento) { /* ... */ }
    public void elimina(T elemento) { /* ... */ }
    public T getPrimero() { /* ... */ }
    public T getUltimo() { /* ... */ }
    public T get(int i) { /* ... */ }
    public Iterator<T> iterator() { /* ... */ }
    /* ... */
```

```
}
```

Listado 3.5: Clase `Lista` iterable.

Los iteradores así definidos funcionarán como embajadores o intermediarios entre el estado interno (y privado o protegido) de nuestras colecciones, permitiéndonos iterarlas sin violar el encapsulamiento de datos. También es importante entender que los iteradores son objetos “desechables”; cuando uno crea un iterador, en principio se espera utilizarlo una única vez. Como la interfaz `Iterator` sólo tiene método `next()`, entonces cuando un iterador ha recorrido todo su objeto iterable, no tiene manera de regresarse y el objeto queda inútil.

Hay maneras de evitar eso (veremos una en el capítulo 5), pero en general es algo *bueno*; los iteradores sólo deben utilizarse para iterar un iterable y entonces deben reflejar el estado del iterable únicamente mientras lo recorre, no necesariamente después. La especificación de iteradores de Java por omisión considera un error modificar un iterable mientras esté siendo iterado y en ese sentido los iteradores no son objetos con una vida larga; en general sólo nos son útiles para iterar el iterable una única vez.

Cómo estará implementado el método `iterator()` de la clase `Lista` lo veremos hasta el siguiente capítulo; pero si suponemos que ya está escrito, entonces el iterar nuestras listas será muy sencillo.

El método `hasNext()` nos dirá si existe al menos un elemento por iterar en nuestro objeto iterable y el método `next()` nos permitirá obtener el siguiente elemento a iterar, además de mover el iterador para poder repetir ambas operaciones si existen más elementos. La interfaz también define un método `remove()`, pero Java ofrece una implementación por omisión que lanza la excepción `UnsupportedOperationException`. Las estructuras de datos iterables que este libro cubrirá se quedarán con esta implementación (o sea, no ofrecerán una implementación de `remove()`), lo cual está permitido por la especificación de la interfaz en Java.

Por lo tanto y de nuevo sin saber exactamente cómo será la implementación de nuestras listas, el recorrerlas (o iterarlas) se realizará como se ve en el listado 3.6, por ejemplo para obtener el promedio de los elementos en una lista de enteros.

```
public double promedio(Lista<Integer> lista) {
    double r = 0.0;
    Iterator<Integer> i = lista.iterator();
    while (i.hasNext())
        r += i.next();
    return r / lista.getElementos();
```

```
}
```

Listado 3.6: Método que calcula *eficientemente* el promedio de una lista de enteros, usando el iterador de la clase `Lista`.

El método `next()` no sólo nos dará el siguiente elemento a iterar en el objeto iterable; también realizará la iteración: conceptualmente, moverá el iterador al siguiente elemento en la lista. Es como hacer dos pasos con una sola operación; regresar el siguiente elemento y mover el iterador un lugar. La primera vez que llamemos al método `next()`, éste nos regresará el primer elemento en el objeto iterable, a menos que la colección esté vacía, en cuyo caso se lanzará la excepción `NoSuchElementException`.

En general la implementación de los métodos `hasNext()` y `next()` tomarán cada uno un número constante de instrucciones; entonces recorrer así una lista de 1,000 elementos tomará unas 1,000 instrucciones, multiplicadas por una constante pequeña. Lo cual es una mejora considerable en comparación con 500,000 instrucciones, que es lo que tomaba al usar el método `get()`.

3.1. El operador *for-each*

El utilizar iteradores al hacer iterables nuestras clases tendrá otra ventaja; podremos utilizar el operador *for-each*. Por ejemplo, nuestro método `promedio()` lo podríamos reescribir utilizando *for-each* como se ve en el listado 3.7.

```
public double promedio(Lista<Integer> lista) {
    double r = 0.0;
    for (Integer n : lista)
        r += n;
    return r / lista.getElementos();
}
```

Listado 3.7: Método que calcula el promedio de una lista de enteros usando *for-each*.

El *for-each*, que en el ejemplo del listado 3.7 se escribe `for (Integer n : lista)`, se lee “para cada entero *n* en la lista” y básicamente el compilador convierte automáticamente este código al código del listado 3.6, como si declarara el iterador y utilizara sus métodos `hasNext()` y `next()`. Nada más que lo hace por nosotros ahorrándonos trabajo y el *for-each* termina siendo mucho más legible.

Los iteradores son uno de los 23 patrones más conocidos del libro [16] de la “Banda de Cuatro” (*Gang of Four* o *GoF* en inglés). Casi todos los lenguajes de programación modernos orientados a objetos siguen el mismo patrón o uno muy similar.

En este capítulo sólo hemos cubierto cómo utilizaremos los iteradores de Java; no hemos especificado cómo los implementaremos concretamente. Todas las estructuras de datos que sean colecciones en este libro serán iterables, lo que implicará que habrá que implementar sus iteradores; veremos a detalle eso en sus capítulos correspondientes.

Lo que sí podemos adelantar es que, como los iteradores son embajadores que interactuarán con las propiedades privadas (o protegidas) de nuestras clases, esto forzará a que tendrán que estar definidos dentro de las mismas. En otras palabras, la clase que definirá los iteradores de una colección siempre será una clase interna y privada o protegida.

Ejercicios

1. Explica una ventaja de utilizar el patrón *Iterador* en el contexto de la Orientación a Objetos.
2. ¿Qué comportamiento provee la interfaz `Iterable`?
3. ¿Qué comportamiento provee la interfaz `Iterator`?
4. Menciona una alternativa a iteradores que nos permitan recorrer los elementos de una colección sin violar el encapsulamiento de datos.
5. Explica qué es lo que hace el compilador de Java al encontrar el operador *for-each*.

4. Colecciones

Como ya hemos mencionado varias veces en los capítulos [2](#) y [3](#), casi todas las estructuras de datos que veremos en el libro serán colecciones genéricas iterables.

Con colecciones (*collections* en inglés) queremos decir que, a diferencia de los conjuntos, permiten elementos repetidos; con genéricas queremos decir que tendrán un parámetro genérico (normalmente llamado `T`); y con iterables queremos decir que implementarán la interfaz `Iterable` de Java.

Todas las estructuras que sean colecciones tendrán una serie de operaciones en común; a todas podremos agregarles elementos; o eliminarles elementos; o preguntar si un elemento está contenido en ellas. Siguiendo los preceptos del diseño Orientado a Objetos, todas estas operaciones las vamos a definir en una interfaz que imaginativamente llamaremos `colección` y que será genérica e iterable. Esto nos permitirá el poder garantizar cierto comportamiento de cada instancia de nuestras estructuras de datos que sean colecciones.

La interfaz `colección` la podemos ver en el listado [4.1](#).

```
public interface Colección<T> extends Iterable<T> {
    public void agrega(T elemento);
    public void elimina(T elemento);
    public boolean contiene(T elemento);
    public boolean esVacia();
    public int getElementos();
    public void limpia();
}
```

Listado 4.1: La interfaz `colección`.

En general el nombre de cada uno de los métodos es en sí mismo bastante descriptivo, pero vamos a revisarlos uno por uno para entender su semántica:

- `agrega()`

El método no regresa ningún valor y recibe la referencia a un objeto de tipo `T`, que debe ser agregado a la colección. El método lanzará la excepción `IllegalArgumentException` si el elemento recibido es `null`; no permitiremos elementos `null` en ninguna de nuestras colecciones.

- `elimina()`

El método no regresa ningún valor y recibe la referencia a un objeto de

tipo `T`, que debe ser eliminado de la colección. En general, si el objeto recibido no está contenido en la colección, esto no se considerará un error, sencillamente el método no modificará el estado de la colección.

- `contiene()`

El método regresa un valor booleano y recibe la referencia a un objeto de tipo `T`. Si el objeto está contenido en la colección, el método regresa `true`; si no, regresa `false`.

- `esVacia()`

El método regresa un valor booleano y no recibe ningún argumento. Regresa `false` si la colección tiene al menos un elemento y `true` si no, en cuyo caso diremos que está vacía.

- `getElementos()`

El método regresa un valor entero y no recibe ningún argumento. Regresa el número de elementos contenidos en la colección; debería llamarse `getTamaño()`, pero como decidimos no utilizar acentos o caracteres especiales en nuestros identificadores, `getElementos()` fue el nombre que elegimos.

Al discutir las estructuras cubiertas en este libro, en particular cuando veamos los algoritmos asociados a ellas, por omisión n siempre será el número de elementos en la colección. En otras palabras, el método `getElementos()` siempre regresará n .

- `limpia()`

El método no regresa nada y no recibe ningún argumento. Limpia la colección de todos sus elementos, dejándola vacía.

- `iterator()`

Recordemos que la interfaz `colección` extiende la interfaz `Iterable` y por lo tanto el método `iterator()` es también parte del comportamiento de las colecciones. Regresa una instancia de la interfaz `Iterator`, que nos permitirá recorrer o iterar la colección.

Además de todos estos métodos, todas nuestras colecciones tendrán una implementación propia de los métodos `equals()` y `toString()` definidos en `Object`.

Ejercicios

1. Explica la diferencia entre un conjunto y una colección.

2. ¿Por qué definimos a **colección** como una interfaz en Java?

5. Listas

Las listas (*lists* en inglés) son la estructura de datos probablemente más utilizada en casi cualquier programa, junto con los arreglos (si el lenguaje de programación soporta arreglos, que algunos no lo hacen).

Conceptualmente son una estructura muy sencilla, que podemos ejemplificar con la lista del supermercado, como se ve en la figura [5.1](#).



Figura 5.1: Lista del supermercado.

Las listas fueron creadas en 1957 por Allen Newell, Cliff Shaw y Herbert Simon para su lenguaje de programación IPL (*Information Processing Language*), que es un lenguaje ensamblador con instrucciones y registros especiales para manipulación de listas que los autores diseñaron para lidiar con problemas de inteligencia artificial [\[28\]](#).

Los elementos que conforman una lista siempre tienen un orden (que no es lo mismo a que estén ordenados) y siempre que la lista no esté vacía podemos identificar un primer elemento (Jabón, en el ejemplo de arriba) y un último elemento (Servilletas).

Desde el capítulo [5](#) hemos mencionado una clase `Lista` y varias de sus posibles operaciones. En general el diseño orientado a objetos funciona de esta manera; el programador piensa en qué operaciones le gustaría que una familia de objetos tuviera y entonces con eso puede definir la clase correspondiente.

Vamos a ver qué operaciones nos gustaría que tuviera una lista y a determinar las firmas de los métodos asociados; pero antes veamos la declaración de la clase misma. Por lo visto en el capítulo [4](#), nos debe de quedar claro que nuestra clase `Lista` será genérica e implementará la interfaz `colección` y entonces la clase lista estaría declarada como en el listado [5.1](#).

```
public class Lista<T> implements Colección<T> { /* ... */ }
```

Listado 5.1: Declaración de la clase `Lista`.

Como la clase `Lista` implementa `colección`, todos los métodos que se discutieron en el capítulo 4 deben ser implementados por la misma; no volveremos a discutir aquí sus firmas.

Las operaciones que le pediremos a nuestras listas en muchos casos será obvio por qué nos interesa tenerlas; en otros casos las requeriremos por cómo continuaremos utilizando las listas en las estructuras de datos que se cubrirán más adelante en el libro.

- Agregar un elemento a la lista por omisión será al final de la misma; pero nos gustaría poder agregar elementos al inicio también. Vamos entonces a necesitar un método para agregar elementos al inicio de la lista que complementa al método `agrega()` de la interfaz `colección`. El método tendrá la misma firma que `agrega()`, como podemos ver en el listado 5.2.

```
public void agregaInicio(T elemento) { /* ... */ }
```

Listado 5.2: Firma del método `agregaInicio()`.

Por simetría vamos a tener también un método para agregar al final, que hará exactamente lo mismo que el método `agrega()`; lo más sencillo para implementarlos es escribir el algoritmo en uno y que el otro lo mande llamar o viceversa. La firma será también igual, como se ve en el listado 5.3.

```
public void agregaFinal(T elemento) { /* ... */ }
```

Listado 5.3: Firma del método `agregaFinal()`.

Por último en la sección de agregar elementos a la lista, vamos a tener un método para insertar un elemento en un lugar arbitrario de la lista, como se ve en el listado 5.4.

```
public void inserta(int i, T elemento) { /* ... */ }
```

Listado 5.4: Firma del método `inserta()`.

Si $i \leq 0$, agregaremos el elemento al inicio; si $i \geq n$ (donde n es el número de elementos en la lista), agregaremos el elemento al final. Si $0 < i < n$, después de insertar el elemento éste tendrá índice i en la lista (el primer elemento tiene índice 0 y el último $n - 1$).

Los tres métodos, al igual que el método `agrega()` de `colección`, lanzarán la

excepción `IllegalArgumentException` si el elemento recibido es `null`.

- Tradicionalmente, el número de elementos que tiene una lista se conoce como la longitud de la misma. La interfaz `colección` ya nos hace implementar un método `getElementos()`, que nos da esta información; pero para cumplir la tradición, también tendremos un método `getLongitud()` que hará lo mismo que `getElementos()`. Igual que en el caso anterior, hacer que uno de ellos mande llamar al otro es lo más sencillo para implementarlos. Su firma la podemos ver en el listado [5.5](#).

```
public int getLongitud() { /* ... */ }
```

Listado 5.5: Firma del método `getLongitud()`.

- Los elementos primero y último de la lista son especiales; vamos a tener métodos para eliminar cada uno. Y ya que estamos eliminando el primero o el último, vamos a aprovechar para que el método correspondiente nos regrese el elemento eliminado en el proceso (listado [5.6](#)).

```
public T eliminaPrimero() { /* ... */ }
public T eliminaUltimo() { /* ... */ }
```

Listado 5.6: Firma de los métodos `eliminaPrimero()` y `eliminaUltimo()`.

Sin embargo, a veces también querremos el primer o último elemento de una lista sin modificarla, entonces vamos a tener métodos para poder obtenerlos específicamente (listado [5.7](#)).

```
public T getPrimero() { /* ... */ }
public T getUltimo() { /* ... */ }
```

Listado 5.7: Firma de los métodos `getPrimero()` y `getUltimo()`.

Los cuatro métodos lanzarán la excepción `NoSuchElementException` si los mandamos llamar con una lista vacía.

- Vamos a necesitar un método para copiar la lista (listado [5.8](#)).

```
public Lista<T> copia() { /* ... */ }
```

Listado 5.8: Firma del método `copia()`.

- Y parecida a la operación anterior, vamos a necesitar un método para obtener una copia de la lista, pero en el orden inverso (listado [5.9](#)).

```
public Lista<T> reversa() { /* ... */ }
```

Listado 5.9: Firma del método `reversa()`.

- Como decimos al inicio del capítulo, los elementos de la lista están en orden en el sentido de que hay un primero, un segundo, un tercero, etcétera. Vamos a querer un método que nos permita obtener el i -ésimo elemento de la lista, donde (como siempre en computación) el primer elemento tendrá el índice 0 (listado 5.10).

```
public T get(int i) { /* ... */ }
```

Listado 5.10: Firma del método `get()`.

El método debe lanzar `ExpcionIndiceInvalido` si pedimos el i -ésimo elemento, con $i < 0$ o $i \geq n$, donde n es la longitud de la lista. Esta excepción es creación nuestra y extiende a `ArrayIndexOutOfBoundsException`.

- Inversamente, dado un elemento de la lista querremos saber su índice en la misma; si el elemento recibido no está en la lista, regresaremos -1 (listado 5.11).

```
public int indiceDe(T elemento) { /* ... */ }
```

Listado 5.11: Firma del método `indiceDe()`.

- Por último y como será el caso con todas las estructuras de datos en este libro, vamos a sobrecargar el método `equals()` (que nos permite ver si dos listas son iguales) y `toString()` (que nos da una representación en cadena de la lista) de la clase `Object` (listado 5.12).

```
@Override public boolean equals(T elemento) {  
    /* ... */  
}  
@Override public boolean toString(T elemento) {  
    /* ... */  
}
```

Listado 5.12: Firma de los métodos `equals()` y `toString()`.

Esto nos permite declarar la clase `Lista` en términos únicamente de su comportamiento:

```
public class Lista<T> implements Coleccion<T> {  
    @Override public void agrega(T elemento) { /* ... */ }}
```

```

@Override public void elimina(T elemento) { /* ... */ }
@Override public boolean
contiene(T elemento) { /* ... */ }
@Override public boolean esVacia() { /* ... */ }
@Override public int getElementos() { /* ... */ }
@Override public void limpia() { /* ... */ }
@Override public Iterator<T> iterator() { /* ... */ }
@Override public String toString() { /* ... */ }
@Override public boolean equals(Object o) { /* ... */ }

public int getLongitud() { /* ... */ }
public void agregaFinal(T elemento) { /* ... */ }
public void agregaInicio(T elemento) { /* ... */ }
public void inserta(int i, T elemento) { /* ... */ }
public T eliminaPrimero() { /* ... */ }
public T eliminaUltimo() { /* ... */ }
public T getPrimero() { /* ... */ }
public T getUltimo() { /* ... */ }
public T get(int i) { /* ... */ }
public Lista<T> copia() { /* ... */ }
public Lista<T> reversa() { /* ... */ }
public int indiceDe(T elemento) { /* ... */ }
}

```

Listado 5.13: Comportamiento de la clase `Lista`.

Los métodos que tenemos que implementar por la interfaz `colección` y los que sobrecargamos de la clase `Object` (`toString()` y `equals()`) tienen la anotación `@override`. Esto no es obligatorio, pero se considera una buena práctica de programación porque hace explícito qué comportamiento *no* es original de la clase.

Desde un punto de vista de *uso* de la clase, con esto sería suficiente; pero obviamente no nos interesa nada más utilizar la clase: queremos escribirla y eso significa escribir el cuerpo de esos métodos, que es implementar los algoritmos asociados a la estructura de datos. Para poder hacer eso, debemos saber cómo son las propiedades o miembros de la listas; en términos de Java, cuáles son las variables de clase que determinan los posibles estados que tendrá y que dicho estado será el que modificarán o reportarán los métodos de los cuales escribimos sus firmas arriba.

Y esto nada más podemos hacerlo si tenemos una definición precisa de qué es una lista, cosa que no hemos visto, pero que cubriremos en la siguiente sección.

5.1. Definición de listas

Desde un punto de vista matemático o teórico, existen múltiples formas de definir listas. Una muy utilizada es la siguiente.

Definicion 5.1. (Listas, primer intento) *Una lista ℓ es:*

- *la lista vacía (\emptyset) o*
- *un elemento seguido de una lista.*

Es una definición recursiva; define un caso base y luego hace uso de lo que está definiendo. Esta definición es muy concisa y elegante; lamentablemente, implementarla utilizando Orientación a Objetos resulta en situaciones incómodas. Como una lista sin elementos es una lista válida y ésta está definida como \emptyset , el único equivalente que tenemos en Java sería utilizar `null` y entonces cada vez que quisiéramos utilizar una lista lo primero que tendríamos que hacer sería preguntar si es distinta de `null`, para evitar un `NullPointerException` si queremos mandar llamar uno de sus métodos, por ejemplo.

Más grave desde un punto de vista conceptual, es que recorrer este tipo de lista con iteradores (los de Java, al menos) se vuelve básicamente imposible; a menos que la misma clase `Lista` implementara la interfaz `Iterator`, pero como vimos en el capítulo 3, esto sería un error, porque siguiendo el patrón *Iterador* una lista no debe recorrerse a sí misma.

Y el problema de fondo es que la definición 5.1 no define realmente listas, en términos de Orientación a Objetos en general y de comportamiento en particular; define lo que denominaremos como *nodos*. Las listas realmente serán conjuntos de nodos.

Otro problema es que la definición 5.1 resultaría en el equivalente de lo que se conoce como *listas simplemente ligadas*, donde a partir de un nodo sólo podemos movernos al nodo siguiente. Nos va a interesar poder movernos en ambas direcciones (hacia adelante y hacia atrás), usando lo que se conoce como *listas doblemente ligadas*.

Vamos a intentarlo de nuevo, pero esta vez primero tendremos una definición auxiliar para nodo.

Definicion 5.2. (Nodos) *Un nodo es:*

- *el nodo vacío (\emptyset) o*
- *un elemento, un nodo anterior y un nodo siguiente.*

Además, para todo par de nodos a y b se cumple que a es el nodo anterior de b si y sólo si b es el nodo siguiente de a .

La definición 5.2 es también recursiva, pero es menos elegante y más rebuscada que la definición 5.1. Sin embargo nos permite implementar nodos con Java de forma casi trivial; podemos definir la clase `Nodo` como se ve en el listado 5.14.

```
public class Nodo<T> {
    private T elemento;
    private Nodo<T> anterior;
    private Nodo<T> siguiente;
}
```

Listado 5.14: Clase `Nodo`.

La clase únicamente implementa la parte *estructural* de la definición 5.2; el nodo vacío será implementado con `null` y la clase `Nodo` es justo un elemento, un nodo anterior y un nodo siguiente. La parte de *comportamiento* de la definición (que dos nodos están mutuamente relacionados o no lo están) le correspondería justamente al comportamiento de la clase, a sus métodos; sin embargo, eso podemos dejarlo fuera de las manos de la clase `Nodo`.

Como dijimos justo antes de la definición 5.2, la definición es auxiliar; lo que queremos hacer es definir listas, así que hagamos eso.

Definicion 5.3. (Listas) *Una lista es un nodo inicial, que llamaremos cabeza y un nodo final, que llamaremos rabo¹. Siempre se cumplirá que:*

- *la lista no tiene elementos si y sólo si cabeza y rabo son \emptyset ,*
- *la lista tiene exactamente un elemento si y sólo si cabeza y rabo son el mismo nodo distinto de \emptyset ; y*
- *la lista tiene más de un elemento si y sólo si, comenzando en cabeza, podemos llegar a rabo siguiendo los elementos siguientes de cada nodo; y comenzando en rabo, podemos llegar a cabeza siguiendo los elementos anteriores de cada nodo.*

Si cabeza es distinto de \emptyset , entonces el nodo anterior a cabeza siempre es \emptyset ; si rabo es distinto de \emptyset , entonces el nodo siguiente a rabo siempre es \emptyset .

Esta definición de listas nos permite implementar la clase `Lista` de manera muy sencilla; y como los nodos únicamente son utilizados por la clase `Lista` (y de hecho no queremos que nadie más ni siquiera pueda verlos, para preservar el encapsulamiento de datos), podemos hacer la clase `Nodo` interna y privada a `Lista`. Más aún, como `Lista` es genérica, la clase `Nodo` ya no tiene que serlo

explícitamente; al estar dentro de `Lista` puede utilizar el parámetro genérico `T` directamente, lo que la hace genérica implícitamente.

```
public class Lista<T> implements Coleccion<T> {

    private class Nodo {
        public T elemento;
        public Nodo anterior;
        public Nodo siguiente;
        public Nodo(T elemento) { /* ... */ }
    }

    private Nodo cabeza;
    private Nodo rabo;

    /* Aquí siguen los métodos... */
}
```

Listado 5.15: Esqueleto de la clase `Lista` casi completo.

Le damos un constructor a la clase `Nodo`, para facilitar su uso; pero sus propiedades son públicas en lugar de tener comportamiento (métodos). Esto no viola el encapsulamiento de datos, porque la clase misma es privada y es únicamente utilizada de forma auxiliar por la clase `Lista`.

Con esto ya casi tenemos finalizado el esqueleto de nuestra clase `Lista`. Vamos a agregar una propiedad más, por razones de eficiencia; la clase `Lista` tendrá una variable de clase de tipo entero llamada `longitud`. Esto será únicamente para no tener que contar todos los elementos de la lista cada vez que nos pidan su longitud; sencillamente incrementaremos `longitud` cada vez que agreguen un elemento y lo decrementaremos cada vez que eliminan un elemento. También simplifica mucho los métodos `getLongitud()` y `getElementos()`, porque ambos lo único que tienen que hacer es regresar la variable de clase `longitud`.

Lo único que falta son los iteradores. Como dijimos al final del capítulo [3](#), la clase que implemente la interfaz `Iterator` debe ser interna y en este caso privada (en otros será protegida).

Ahora, la interfaz `Iterator` sólo permite iterar colecciones en una dirección (recuerden que sólo tiene métodos `hasNext()` y `next()`), entonces con ellos sólo podríamos recorrer nuestras listas en una dirección, de la cabeza hacia el rabo. Esto no es óptimo, ya que nos estamos tomando la molestia de implementar listas doblemente ligadas, así que para resolverlo vamos a crear una nueva interfaz que llamaremos `IteradorLista` y que extenderá a `Iterator` (listado [5.16](#)).

```

public interface IteradorLista<T> extends Iterator<T> {
    public boolean hasPrevious();
    public T previous();
    public void start();
    public void end();
}

```

Listado 5.16: Interfaz `IteradorLista`.

Los métodos `hasPrevious()` y `previous()` funcionan igual que `hasNext()` y `next()`, sólo en la dirección contraria. El método `start()` hace que el iterador regrese a la posición inicial, donde llamar `next()` regresa el primer elemento; el método `end()` hace lo inverso, después de llamarlo, invocar el método `previous()` nos regresa el último elemento.

La clase `Iterador` (en español), interna y privada a `Lista`, implementará la interfaz `IteradorLista` (y por lo tanto también `Iterator`). La vamos a agregar, pero discutiremos más adelante en el capítulo cómo implementar sus métodos.

También agregaremos un método más a la clase `Lista`; el método `iterador()` nos regresa una instancia de `Iterator` y lo que haremos será regresar una instancia de `IteradorLista`. Pero no podemos cambiar la firma del método `iterator()`, así que aunque lo que regresa sea una instancia de `IteradorLista`, el compilador sólo verá que recibimos una instancia de `Iterator`. Podríamos hacer una audición, pero mejor vamos a agregar un método `iteradorLista()`, con la firma en el listado [5.17](#). Esta es una solución más elegante.

```

public IteradorLista<T> iteradorLista() { /* ... */ }

```

Listado 5.17: Firma del método `iteradorLista()`.

El esqueleto completo de la clase `Lista` se puede ver en el listado [5.18](#).

```

public class Lista<T> implements Coleccion<T> {

    private class Nodo {
        public T elemento;
        public Nodo anterior;
        public Nodo siguiente;
        public Nodo(T elemento) { /* ... */ }
    }

    private class Iterador implements IteradorLista<T> {
        public Nodo anterior;
        public Nodo siguiente;
        public Iterador() { /* ... */ }
    }
}

```

```

@Override public boolean hasNext() { /* ... */ }
@Override public T next() { /* ... */ }
@Override public boolean hasPrevious() { /* ... */ }
@Override public T previous() { /* ... */ }
@Override public void start() { /* ... */ }
@Override public void end() { /* ... */ }

}

private Nodo cabeza;
private Nodo rabo;
private int longitud;

@Override public void agrega(T elemento) { /* ... */ }
@Override public void elimina(T elemento) { /* ... */ }
@Override public boolean
    contiene(T elemento) { /* ... */ }
@Override public boolean esVacia() { /* ... */ }
@Override public int getElementos() { /* ... */ }
@Override public void limpia() { /* ... */ }
@Override public Iterator<T> iterator() { /* ... */ }
@Override public String toString() { /* ... */ }
@Override public boolean equals(Object o) { /* ... */ }

public int getLongitud() { /* ... */ }
public void agregaFinal(T elemento) { /* ... */ }
public void agregaInicio(T elemento) { /* ... */ }
public T eliminaPrimero() { /* ... */ }
public T eliminaUltimo() { /* ... */ }
public T getPrimero() { /* ... */ }
public T getUltimo() { /* ... */ }
public T get(int i) { /* ... */ }
public Lista<T> copia() { /* ... */ }
public Lista<T> reversa() { /* ... */ }
public int indiceDe(T elemento) { /* ... */ }
public IteradorLista<T> iteradorLista() { /* ... */ }
}

```

Listado 5.18: Esqueleto de la clase [Lista](#).

Con esto terminamos con el diseño de la clase [Lista](#). Ahora sigue la otra mitad: implementar los métodos de la clase con los algoritmos correspondientes, que será lo que veremos en la sección siguiente.

5.2. Algoritmos para listas

Vamos a ver los distintos algoritmos que necesitan implementar los métodos de la clase [Lista](#). Los métodos de una clase son la implementación concreta de uno o varios algoritmos; un algoritmo existe de manera independiente a su

implementación concreta (podemos implementar el mismo algoritmo en muchos lenguajes de programación y existen muchos algoritmos que fueron diseñados cientos de años antes de que existieran las computadoras), pero la implementación depende totalmente del algoritmo.

En general describiremos los algoritmos usando lenguaje escrito y diagramas para explicarlos; no veremos el código o pseudocódigo directamente (la idea es que ustedes lo escriban), excepto para ver ejemplos y patrones que nos parecen importantes o que demuestran buenas prácticas de programación. En particular, la descripción de los algoritmos sólo mencionará los casos que consideraremos errores, sin entrar a detalles concretos de la implementación en Java (dícese, qué excepción se lanzaría).

Siguiendo la definición 5.3, podemos visualizar a una lista con múltiples elementos como en la figura 5.2. Cada caja representa un nodo y la flecha a la izquierda de una caja es el nodo anterior y la flecha a la derecha el nodo siguiente. Esta manera de representar listas se utiliza desde [28], si bien las listas de IPL eran simplemente ligadas y de hecho definidas como en la definición 5.1. Utilizamos c para representar la cabeza de la lista y r para representar el rabo. Cumpliendo con la definición, el nodo anterior a la cabeza es \emptyset , al igual que el nodo siguiente al rabo. Además, podemos recorrer la lista de la cabeza al rabo yéndonos por los nodos siguientes y podemos recorrer la lista del rabo a la cabeza yéndonos por los nodos anteriores.

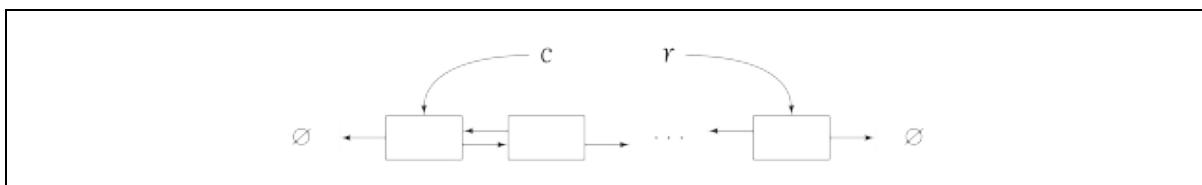


Figura 5.2: Visualización de una lista con múltiples elementos.

Los algoritmos que implementarán los métodos de la clase **Lista** los podremos dividir en dos grandes grupos: aquellos que necesitan recorrer los nodos de la lista y aquellos que no. En términos de los algoritmos abstractos, el patrón para recorrer una lista será siempre el mismo y podemos ver un ejemplo en el algoritmo 5.1 para imprimir todos los elementos de la lista. El algoritmo recorre la lista utilizando un nodo auxiliar n que comienza siendo la cabeza de la lista y que se actualiza a todos los nodos de la misma hasta llegar al \emptyset que es el nodo siguiente del rabo. Si la lista es vacía, ni siquiera se entra al **while**.

```

procedure IMPRIMEELEMENTOS( $L$ )
   $n \leftarrow \text{GETCABEZA}(L)$ 
  while ( $n \neq \emptyset$ )
    IMPRIME(GETELEMENTO( $n$ ))
     $nodo \leftarrow \text{GETSIGUIENTE}(n)$ 
  
```

Algoritmo 5.1: Algoritmo para imprimir todos los elementos de una lista L .

Una versión recursiva que hace lo mismo lo podemos ver en el algoritmo 5.2; recuerden que todo algoritmo recursivo puede plantearse iterativamente y viceversa. También noten que nuestro lenguaje para pseudocódigo soporta polimorfismo y entonces podemos tener funciones con el mismo nombre siempre y cuando reciban argumentos de tipos diferentes; en el caso del algoritmo 5.2, hay dos funciones IMPRIMEELEMENTOS(): una recibe una lista y otra un nodo.

```
procedure IMPRIMEELEMENTOS( $L$ )
    IMPRIMEELEMENTOS(GETCABEZA( $L$ ))

procedure IMPRIMEELEMENTOS( $n$ )
    if ( $n = \emptyset$ )
        return
    IMPRIME(GETELEMENTO( $n$ ))
    IMPRIMEELEMENTOS(GETSIGUIENTE( $n$ ))
```

Algoritmo 5.2: Algoritmo para imprimir recursivamente todos los elementos de una lista L .

Por supuesto, en general en Java no podremos recorrer nuestras listas así por el encapsulamiento de datos; la lista no debe proporcionarnos la cabeza y de hecho el mundo exterior ni siquiera debe saber de la existencia de los nodos. Cuando más adelante escribamos los métodos de la clase **Iterador** podremos utilizar iteradores para poder recorrer la lista en general.

Pero *dentro* de la misma clase **Lista**, para implementar los algoritmos correspondientes a cada uno de sus métodos sí podemos usar los nodos y el patrón que sigue el algoritmo 5.1 se puede traducir casi enunciado por enunciado a instrucciones de Java, como se muestra en el listado 5.19.

```
public void imprimeElementos() {
    Nodo nodo = cabeza;
    while (nodo != null) {
        System.out.println(nodo.elemento);
        nodo = nodo.siguiente;
    }
}
```

Listado 5.19: Código para imprimir los elementos de una lista, dentro de la clase **Lista**.

De la misma manera, el algoritmo 5.2 se traduciría en Java a algo como lo que se muestra en el listado 5.20.

```

public void imprimeElementos() {
    imprimeElementos(cabeza);
}
private void imprimeElementos(Nodo nodo) {
    if (nodo == null)
        return;
    System.out.println(nodo.elemento);
    imprimeElementos(nodo.siguiente);
}

```

Listado 5.20: Código para imprimir recursivamente los elementos de una lista, dentro de la clase `Lista`.

Hacemos énfasis en el hecho de que esto únicamente lo podremos hacer *dentro* de la clase `Lista`; afuera necesitaremos los iteradores para poder recorrer nuestras listas porque no tendremos disponibles a los nodos.

Veremos más adelante que en Java la versión recursiva de un algoritmo puede ser un poco más costosa en memoria que la iterativa; pero esto no es muy grave y todos los algoritmos que necesiten recorrer la lista los podríamos implementar recursivamente. La decisión quedará en el programador (o sea ustedes).

Veremos los algoritmos en el orden dado por los métodos del listado [5.18](#).

- `agrega()`

El algoritmo para agregar elementos a una lista, que como dijimos al inicio del capítulo será por omisión al final, tiene dos casos diferentes: cuando la lista es vacía y cuando al menos contiene un elemento.

En cualquiera de los dos casos lo primero que tenemos que hacer es construir un nuevo nodo n e incrementar el contador de elementos en la lista. Vamos a suponer que al crear un nodo, sus nodos siguiente y anterior son por omisión \emptyset ; en particular esto se cumple en Java: las variables de clase por omisión se inicializan en `null`. Sin embargo, esto no es cierto para todos los lenguajes de programación.

Si la lista es vacía, que podemos verificar sencillamente comprobando si el rabo r es \emptyset , entonces la cabeza c y el rabo r se actualizan a n y terminamos.

Si la lista no es vacía, entonces el rabo r es distinto de \emptyset ; hacemos que el siguiente de r sea n , que el anterior de n sea r (para cumplir la definición [5.2](#)) y por último actualizamos el rabo r a n (figura [5.3](#)). Conceptualmente es como si moviéramos el rabo un lugar a la derecha, porque tiene que apuntar (en terminología de Java, referir) al nuevo último elemento.

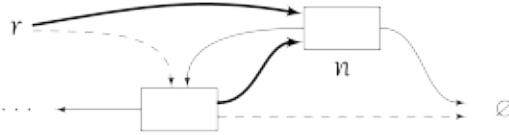


Figura 5.3: Los cambios al agregar un elemento al final de una lista no vacía. Las referencias punteadas son reemplazadas por las dos referencias oscuras.

- `elimina()`

El algoritmo para eliminar un elemento se puede dividir en dos algoritmos auxiliares; un algoritmo que regrese el primer nodo de la lista donde esté algún elemento (o \emptyset si el elemento no está en la lista) y otro algoritmo para eliminar un nodo de la lista.

El algoritmo para buscar el nodo que contenga un elemento sigue básicamente el patrón que mostramos en el algoritmo 5.1, sólo que en lugar de imprimir el elemento en la consola, verifica si es igual al que recibe como parámetro y si así es regresa el nodo. Si la iteración termina sin que ningún nodo contenga al elemento, el algoritmo regresa \emptyset .

Si el algoritmo auxiliar para buscar el nodo nos regresa \emptyset , terminamos. Si nos regresa un nodo distinto a \emptyset , usamos el algoritmo auxiliar para eliminar un nodo n de la lista, que hace lo siguiente: primero decrementa el contador de elementos en la lista. Después, tenemos cuatro casos distintos a tomar en cuenta:

1. La cabeza y el rabo son el mismo nodo; entonces lista contiene un único elemento y $c = r = n$. Sencillamente actualizamos r y c a \emptyset y terminamos.
2. El nodo n es la cabeza de la lista; sea s el nodo siguiente de n . El nodo s es distinto de \emptyset , de otra forma hubiera ocurrido el caso 1; actualizamos el nodo anterior de s a \emptyset , actualizamos c a s y terminamos (figura 5.4).

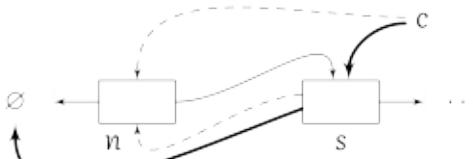


Figura 5.4: Los cambios al eliminar la cabeza de una lista.

3. El nodo n es el rabo de la lista; este caso es básicamente idéntico al

anterior, sólo intercambiando r y c e invirtiendo anterior y siguiente. Sea a el nodo anterior de n , que es distinto de \emptyset porque si no se hubiera dado el caso 1; actualizamos el nodo siguiente de a a \emptyset , actualizamos r a a y terminamos figura 5.5).

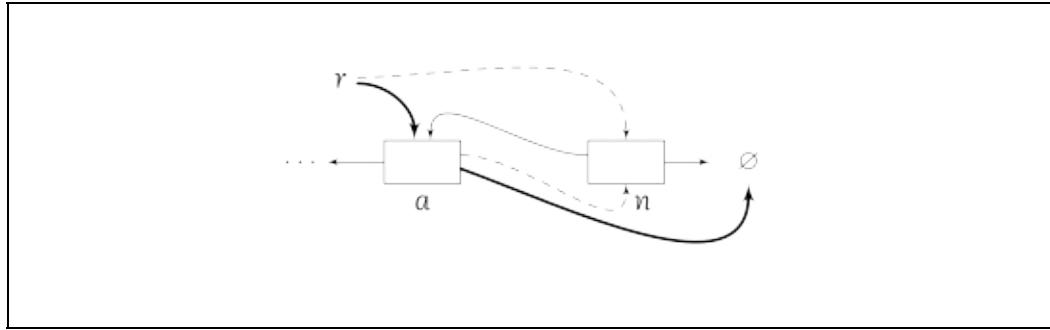


Figura 5.5: Los cambios al eliminar el rabo de una lista.

4. El nodo n no es ni la cabeza ni el rabo. Sean a y s los nodos anterior y siguiente a n , respectivamente; como no se dieron los casos 1, 2 o 3, entonces $a \neq \emptyset$ y $s \neq \emptyset$. Actualizamos el siguiente de a a que sea s y el anterior de s a que sea a y terminamos.

En español: actualizamos el nodo siguiente del anterior de n al siguiente de n y al nodo anterior del siguiente de n al anterior de n (figura 5.6).

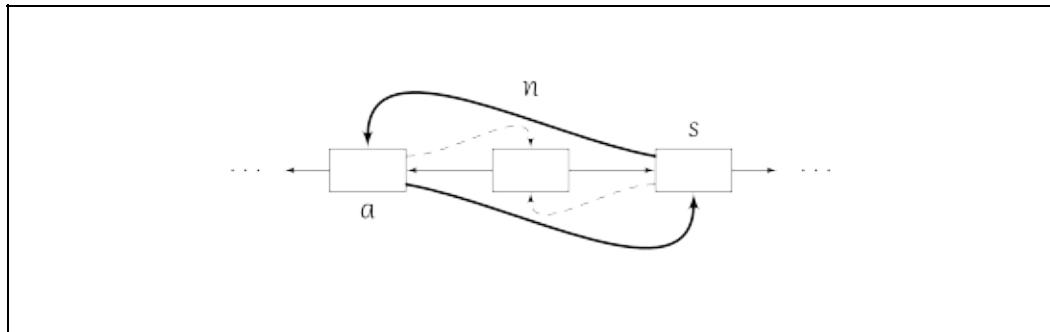


Figura 5.6: Los cambios al eliminar un nodo intermedio de una lista.

En cualquiera de los cuatro casos, hay que destruir (en el sentido contrario de construir) el nodo del elemento eliminado para liberar su memoria. Sin embargo, como en Java de esto se encarga el recolector de basura (porque ya no hay referencias vivas hacia el nodo del elemento eliminado), vamos a omitir este paso en este y todos los demás algoritmos que liberen memoria en el libro.

- `contiene()`

Lo único que hay que hacer es reutilizar el algoritmo auxiliar que usamos para buscar el nodo de un elemento en el método `elimina()`; si el nodo que regresa es vacío, entonces el algoritmo regresa falso, de otra

forma regresa verdadero.

Suponiendo que el algoritmo auxiliar lo implementamos en un método auxiliar (y por lo tanto privado) llamado `buscaNodo()`, el algoritmo es tan sencillo que lo vamos a mostrar. Una primera versión la podemos ver en el listado 5.21.

```
public boolean contiene(T elemento) {
    Nodo n = buscaNodo(elemento);
    if (n == null)
        return false;
    else
        return true;
}
```

Listado 5.21: Método `contiene()`, primera versión.

Esta primera versión es técnicamente correcta; pero estamos haciendo demasiado trabajo para algo que puede ser mucho más sencillo: no hay necesidad de declarar una variable (podríamos hacer el `if` con la llamada a `buscaNodo()` en la condicional directamente); si el cuerpo de un `if` termina en `return` no hay necesidad de hacer un `else`; pero más importante aún, ni siquiera hay necesidad de hacer un `if`. Todo el método se puede escribir en un enunciado (listado 5.22).

```
public boolean contiene(T elemento) {
    return buscaNodo(elemento) != null;
}
```

Listado 5.22: Método `contiene()`, versión más concisa.

Si un método regresa un booleano, en general uno puede ahorrarse por lo menos un `if`.

- `esVacia()`

Nada más hay que ver si la cabeza es vacía. Podemos verificar el rabo también, pero si mantenemos el estado de nuestras listas consistente en los métodos de la clase, entonces no debe ser posible que la cabeza sea vacía si el rabo no lo es y viceversa.

También podemos verificar si la longitud es cero; nosotros preferimos verificar los nodos, porque la variable de clase `longitud` es realmente un contador que se agregó por eficiencia. Pero el método funciona igual de bien si verifican la longitud en lugar de los nodos cabeza o rabo.

- `getElementos()`

Gracias a que agregamos la variable de clase `longitud`, lo único que hay que hacer es regresárla.

- `limpia()`

El algoritmo sólo regresa la lista a su estado inicial cuando era vacía. Lo único que hay que hacer es actualizar la cabeza y el rabo a vacío y la longitud de la lista a cero.

- `iterator()`

Estamos dejando para el final los métodos de la clase interna privada `Iterador`; pero si suponemos que ya está escrita, escribir `iterator()` es tan sencillo que lo mostramos en el listado 5.23.

```
public Iterator<T> iterator() {  
    return new Iterador();  
}
```

Listado 5.23: Método `iterator()`.

Sólo construimos un iterador y lo regresamos.

- `toString()`

La representación en cadena de una lista será utilizando corchetes y cada par de elementos separados por una coma y un espacio. Cada elemento se representará en cadena usando su propio método `toString()`.

La cadena correspondiente a la lista vacía siempre es “[]”, mientras que la cadena correspondiente a la lista de la figura 5.7 sería “[1, 2, 3, 4]”.



Figura 5.7: Lista de enteros.

El algoritmo para construir esta cadena funciona de la siguiente manera; si la lista es vacía, sencillamente se regresa la cadena “[]”.

Si la lista no es vacía, se inicializa una cadena *r* con el corchete izquierdo y el elemento del primer nodo. Después se recorre el resto de los nodos siguiendo el patrón del algoritmo 5.1, nada más que comenzando en el nodo siguiente a la cabeza, no en la cabeza (porque ese elemento ya está en la cadena). Para cada nodo que no sea \emptyset , se concatena a la cadena *r* una coma, un espacio y el elemento del nodo.

Terminando la iteración, se regresa *r* concatenada a un corchete derecho.

- `equals()`

El algoritmo para comprobar que dos listas son iguales es como sigue: si las longitudes de las dos listas son distintas, se regresa falso. Si son iguales, se utiliza una patrón similar al del algoritmo [5.1](#), nada más que con dos nodos temporales, uno por lista. Se comparan los elementos de cada nodo, si son distintos se regresa falso y si son iguales se mueve cada nodo a la derecha; si la iteración termina, se regresa verdadero.

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    Lista<T> lista = (Lista<T>)o;
    // Aquí va el resto del método.
}
```

Listado 5.24: El inicio del método `equals()`, suprimiendo la advertencia relacionada con genéricos.

Hay un detalle técnico con los genéricos de Java; el método `equals()` recibe un `Object` y necesitamos una `Lista<T>`. No existe de otra sino hacer una audición, pero como estamos haciendo una audición de un tipo no genérico (`Object`) a uno genérico (`Lista<T>`), el compilador de Java va a lanzar una advertencia.

Como sabemos que estamos haciendo lo correcto (de hecho, lo único que podemos hacer), vamos a suprimir la advertencia con `@SuppressWarnings`. Esta anotación debe ser utilizada con mucho cuidado, únicamente cuando estemos *completamente seguros* de que lo que estemos haciendo es correcto a pesar de las advertencias del compilador. El inicio del método `equals()` se puede ver en el listado [5.24](#).

Hacemos notar que no utilizamos `instanceof`; en su lugar comparamos las clases de los objetos con el método `getClass()` de la clase `Object`. En la clase `Lista` realmente no es necesario; pero lo será en varias de las clases más adelante en el libro. Lo explicaremos en el capítulo [8](#).

- `getLongitud()`

El algoritmo es idéntico al de `getElementos()` y de hecho lo más sencillo es que `getLongitud()` invoque a `getElementos()` o viceversa. El código repetido en general trataremos de evitarlo; si hubiera que modificar el algoritmo de `getLongitud()` (es improbable, pero nunca se sabe), sólo hay que modificarlo una vez si únicamente está implementado en un lugar.

- `agregaFinal()`

El algoritmo es idéntico al de `agrega()`; se aplica lo mismo del caso

anterior.

- `agregaInicio()`

El algoritmo es muy similar al de `agregaFinal()`, pero intercambiando la cabeza por el rabo e invirtiendo anterior y siguiente.

Creamos un nuevo nodo n e incrementamos la longitud de la lista. Si la lista es vacía actualizamos la cabeza c y el rabo r a n y terminamos.

Si la lista no es vacía la cabeza c es distinta de \emptyset ; hacemos que el anterior de c sea n y que el siguiente de n sea c . Por último, actualizamos c a n (figura 5.8).

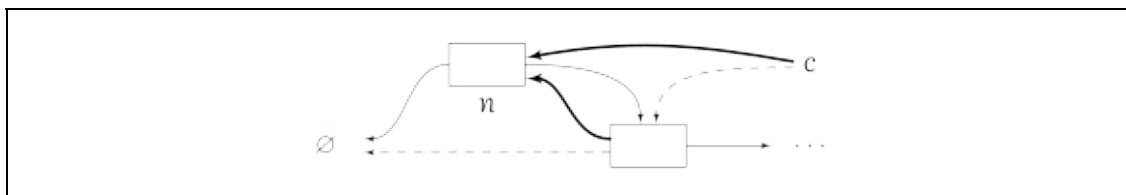


Figura 5.8: Los cambios al agregar un elemento al inicio de una lista no vacía. Las referencias punteadas son reemplazadas por las dos referencias oscuras.

- `inserta()`

Si $i < 1$ reutilizamos el algoritmo para agregar al inicio el elemento (en Java, mandamos llamar `agregaInicio()`). Si $i > n - 1$, reutilizamos el algoritmo para agregar al final el elemento. Noten que estos dos casos cubren si la lista tiene 0 o 1 elementos.

Vamos a crear un algoritmo auxiliar que regrese el i -ésimo nodo de la lista, si $0 \leq i \leq n - 1$; recorre los nodos de la lista i veces (hay que usar un contador auxiliar) siguiendo el patrón del algoritmo 5.1.

Si $0 < i \leq n - 1$, incrementamos la longitud de la lista y creamos un nodo n con el elemento a insertar. Sea s el i -ésimo nodo de la lista, obtenido con nuestro algoritmo auxiliar; como $0 < i$, el nodo anterior de s es distinto de \emptyset . Sea a el nodo anterior de s ; tenemos que insertar n a la derecha de a y a la izquierda de s .

Hacemos que el anterior de n sea a y que el siguiente de a sea n ; y hacemos que el siguiente de n sea s y el anterior de s sea n (figura 5.9).

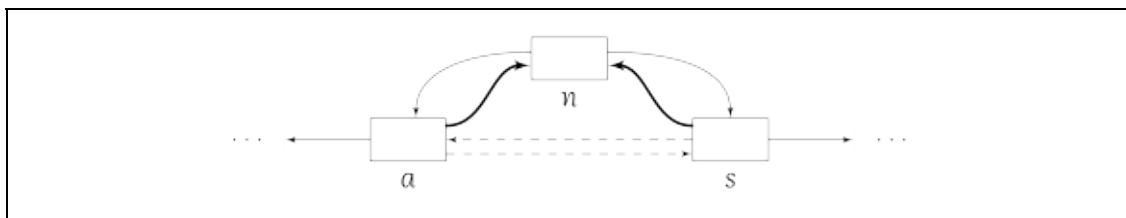


Figura 5.9: Los cambios al insertar un elemento en la i -ésima posición de la lista. Las referencias punteadas son reemplazadas por las dos referencias oscuras.

- `eliminaPrimero()`

Podemos reutilizar el algoritmo auxiliar del algoritmo para eliminar que elimina un nodo. Primero vemos si la lista es vacía y si lo es ocurre un error. Si no, eliminamos la cabeza usando el algoritmo auxiliar.

- `eliminaUltimo()`

Es idéntico al caso anterior, sólo que eliminamos el rabo en lugar de la cabeza.

- `getPrimero()`

Si la cabeza no es vacía, regresamos el elemento en ella. Si es vacía, ocurre un error.

- `getUltimo()`

Si el rabo no es vacío, regresamos el elemento en él. Si es vacío, ocurre un error.

- `get()`

Si la lista tiene n elementos y $i < 0$ o $i \geq n$, se da un error.

Si no, reutilizamos el algoritmo auxiliar de `inserta()` y obtenemos el i -ésimo nodo de la lista; sólo regresamos su elemento.

- `copia()`

Construimos una nueva lista y recorremos de cabeza a rabo los elementos de la lista a copiar, agregándolos al final de la nueva lista. Al terminar, regresamos la nueva lista.

- `reversa()`

Construimos una nueva lista y recorremos de cabeza a rabo los elementos de la lista a copiar, agregándolos al inicio de la nueva lista. Al terminar, regresamos la nueva lista.

También podemos recorrer la lista de rabo a cabeza y agregar a la nueva lista al final; es lo mismo.

- `indiceDe()`

Este método es el inverso de `get()`; inicializamos un contador en cero y recorremos la lista siguiendo el patrón del algoritmo 5.19, incrementando el contador cada vez que actualizamos el nodo auxiliar. Para cada elemento de la lista, verificamos si es igual al que recibimos; si así es regresamos el contador. Si la iteración termina, quiere decir que el elemento no está en la lista y regresamos -1 .

- `iteradorLista()`

De nuevo, estamos dejando para más adelante a los algoritmos de la clase `Iterator`; pero al igual que con `iterator()`, el método es tan sencillo que lo mostramos en el listado [5.25](#).

```
public IteratorLista<T> iteradorLista() {
    return new Iterador();
}
```

Listado 5.25: Método `iteradorLista()`.

Sólo construimos un iterador y lo regresamos. Como `Iterador` implementará la interfaz `IteratorLista<T>` (que a su vez extiende a `Iterator <T>`), las instancias de la clase funcionan para ambos métodos.

Con esto hemos terminado con todos los métodos públicos de la clase `Lista`; para finalizar la clase, lo único que necesitamos es completar la clase `Iterador` interna y privada de `Lista`.

5.3. Iteradores para listas

En abstracto y por lo visto en el capítulo [3](#), ya sabemos cómo funcionan los iteradores y el esqueleto de la clase `Iterador` se puede ver en el listado [5.18](#). Para completarla, primero veamos cómo funcionarán conceptualmente los iteradores para listas.

Podremos visualizar a los iteradores de una lista como que siempre están entre dos nodos a y b ; como los nodos, los iteradores tendrán un nodo siguiente y un nodo anterior, pero a diferencia de los nodos no tendrán un elemento. Esto se traduce a que en los distintos estados de un iterador, su nodo siguiente y anterior serán:

- Si el iterador está al inicio de la lista, entonces está a la izquierda de la cabeza c . El nodo anterior del iterador es \emptyset y el nodo siguiente del iterador es c .
- Si el iterador está al final de la lista, entonces está a la derecha del rabo r . El nodo anterior del iterador es r y el nodo siguiente del iterador es \emptyset .
- Si el iterador está entre dos nodos a y b , entonces su nodo anterior es a y su nodo siguiente es b .

Al crear un iterador, por omisión estará a la izquierda de la cabeza, como se puede ver en la figura [5.10](#).

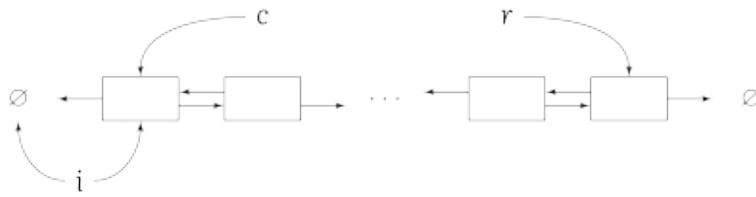


Figura 5.10: Visualización de un iterador i al inicio una lista.

Al mandar llamar el método `next()` del iterador y si la lista no es vacía, el iterador se mueve un lugar a la derecha y regresa el elemento en la cabeza (figura 5.11).

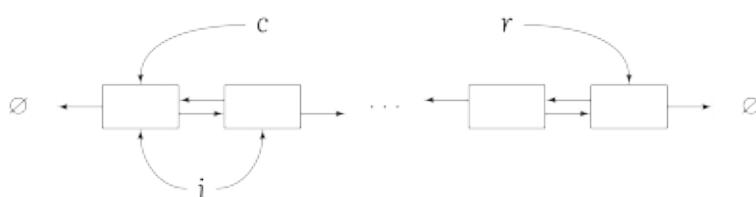


Figura 5.11: El iterador i de la figura 5.10 después de que llama a su método `next()`.

Podemos repetir esto hasta que el iterador esté a la derecha del rabo, como se ve en la figura 5.12.

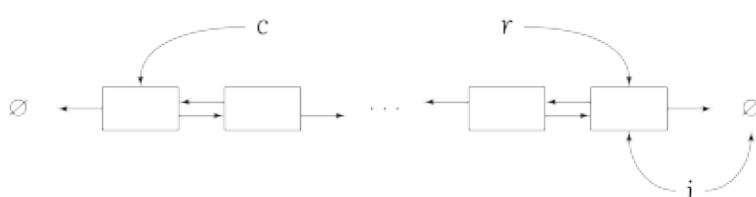


Figura 5.12: El iterador i de la figura 5.11 al final de la lista.

Si volvemos a llamar el método `next()` del iterador, una excepción debe ocurrir porque ya no existe un elemento siguiente. Pero podemos llamar al método `previous()` y entonces el iterador se moverá un lugar a la izquierda y regresará el elemento en el rabo (figura 5.13).

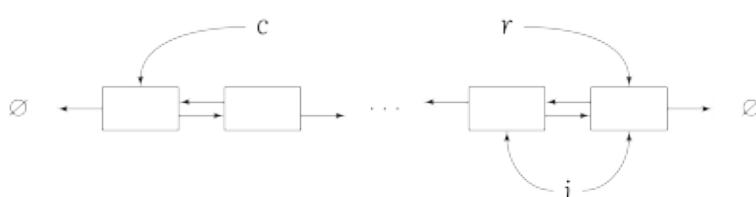


Figura 5.13: El iterador i de la figura 5.12 después de mandar llamar el método `previous()`.

Los iteradores de la clase `Lista` serán los únicos de este libro que no serán

desechables; al tener la interfaz `IteradorLista` un método `previous()`, podemos utilizar un mismo iterador para iterar la lista múltiples veces en distintas direcciones. Sin embargo, lo que mencionamos en el capítulo [3](#) se mantiene: el modificar la lista mientras la recorremos puede dar lugar a errores y en general trataremos de evitarlo.

Los iteradores son muy similares a los nodos y de hecho podríamos escribir la clase `Nodo` para que implementara la interfaz `IteradorLista`. Esto nos parece incorrecto, dado que los iteradores (como todas nuestras figuras muestran) existen aparte de los nodos y conceptualmente entre ellos.

Por lo tanto escribiremos la clase `Iterador` como se muestra en el listado [5.18](#); las únicas variables de clase que tiene son dos nodos, justo el anterior y el siguiente. Después de ver cómo se ven los iteradores de las listas conceptualmente, los algoritmos correspondientes se simplifican bastante. En el orden en que son declarados:

- `Iterador()`

El constructor; como dijimos arriba al crear un iterador por omisión estará a la izquierda de la cabeza; entonces sencillamente hacemos al anterior del iterador vacío y a su siguiente lo definimos como *c*. También podemos invocar el método `start()`, si queremos reutilizar el método.

- `hasNext()`

Sólo hay que comprobar que el siguiente del iterador sea distinto de vacío.

- `next()`

Si el siguiente del iterador es vacío ocurre un error. Si no, sea *s* el nodo siguiente del iterador *i*. Actualizamos al anterior de *i* como *s*, al nodo siguiente de *s* como el siguiente de *i* y regresamos el elemento del anterior de *i*, que para este momento es *s* (figura [5.14](#)). Noten que únicamente sabemos que el nodo *s* existe; es posible que sea el único en la lista.

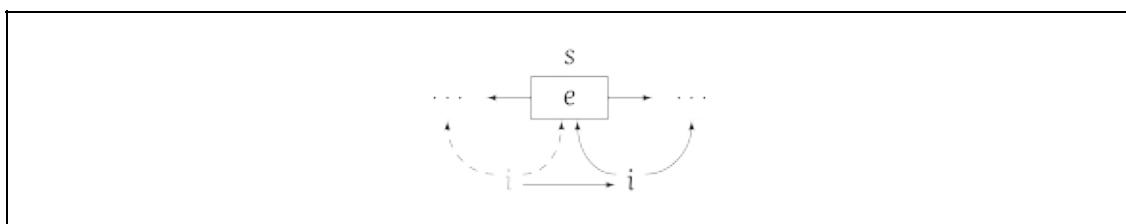


Figura 5.14: El iterador *i* antes (en gris y con líneas punteadas) y después (en negro y líneas continuas) de ejecutar `next()` cuando sí hay un elemento siguiente.

- `hasPrevious()`

Hay que comprobar que el anterior del iterador sea distinto del vacío.

- `previous()`

Es igual a `next()`, nada más que intercambiando anterior con siguiente: si el anterior del iterador es vacío ocurre un error.

Si no, sea a el nodo anterior del iterador i . Actualizamos al siguiente de i como a , al nodo anterior de a como el anterior de i y regresamos el elemento del siguiente de i , que para este momento es a (figura 5.15).

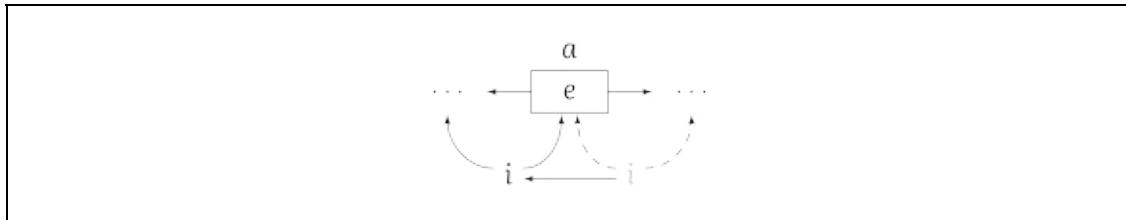


Figura 5.15: El iterador i antes (en gris y con líneas punteadas) y después (en negro y líneas continuas) de ejecutar `previous()` cuando sí hay un elemento anterior.

- `start()`

Esto mueve el iterador al inicio de la lista o en otras palabras a la izquierda de la cabeza; hacemos el anterior del iterador vacío y el siguiente la cabeza. Esto funciona siempre, incluso si la lista es vacía.

- `end()`

Esto mueve el iterador al final de la lista o en otras palabras a la derecha del rabo; hacemos el siguiente del iterador vacío y el anterior el rabo. Esto funciona siempre, incluso si la lista es vacía.

Con esto terminamos con la clase `Lista`, aunque regresaremos a ella en los capítulos [10](#) y [11](#) para agregarle cierta funcionalidades, para las cuales necesitamos ver antes otros temas.

Una vez escrita esa funcionalidad faltante, nuestras listas serán muy similares en su implementación a la clase `LinkedList` de Java. Las listas ligadas de Java ofrecen algo más de funcionalidad, pero nada realmente muy difícil de implementar; sólo no nos parece imprescindible para un curso de Estructuras de Datos.

Ejercicios

1. Implementa los métodos faltantes de la clase `Lista`.
2. ¿Qué tienen en común los métodos `inserta()`, `elimina()` y `copia()`?
3. ¿Qué tienen en común los métodos `agrega()`, `eliminaUltimo()` y `getPrimero()`?

4. Explica por qué es necesaria la interfaz `IteradorLista`.
 5. Explica por qué es necesario el método `iteradorLista()`.
-

1. En inglés los términos son *head* y *tail* (como al lanzar un volado); *tail* puede traducirse como **cola**, pero una estructura de datos muy importante (que veremos en el capítulo 8) se llama justamente cola, así que utilizamos el término **rabo**.[←](#)

6. Complejidad computacional

Los algoritmos que vimos en el capítulo 5 se pueden dividir en dos grupos principales: aquellos que no recorren la lista y aquellos que sí la recorren.

En el primer grupo se incluyen `getLongitud()`, `getPrimero()`, `agregaFinal()`, etcétera. En el segundo se incluyen `copia()`, `reversa()`, `elimina()`, etcétera.

Los primeros siempre tardan lo mismo (o casi lo mismo) no importa cuántos elementos contenga la lista; los segundos tardan cada vez más conforme la longitud de nuestra lista crece.

En complejidad computacional, los primeros se dice que tienen una complejidad $O(1)$ en tiempo, mientras que los segundos se dice que tienen una complejidad $O(n)$ en tiempo. En lenguaje hablado, esto se suele expresar como “*tienen complejidad constante en tiempo*” y “*tienen complejidad lineal en tiempo*”, respectivamente.

La notación de O grandota (*big O* en inglés) nos permite considerar únicamente el consumo más significativo de un recurso (tiempo, en estos primeros ejemplos) por parte del algoritmo, sin preocuparnos de detalles que, en la mayor parte de los casos, no aportan mucho al momento de analizarlo.

Por ejemplo, nuestro algoritmo para agregar un elemento al final de una lista se puede ver (en pseudocódigo en lugar de español, como en el capítulo pasado) en el algoritmo 6.1.

```
procedure AGREGAFINAL( $L, e$ )
   $n \leftarrow \text{NODO}(e)$ 
  if ( $\text{GETRABO}(L) = \emptyset$ )
     $\text{SETCABEZA}(L, n)$ 
     $\text{SETRABO}(L, n)$ 
  else
     $\text{SETANTERIOR}(n, \text{GETRABO}(L))$ 
     $\text{SETSIGUIENTE}(\text{GETRABO}(L), n)$ 
     $\text{SETRABO}(L, n)$ 
   $\text{SETELEMENTOS}(L, \text{GETELEMENTOS}(L) + 1)$ 
```

Algoritmo 6.1: Algoritmo para agregar un elemento e al final de una lista L .

El algoritmo siempre crea un nodo, que (simplificando) toma 4 instrucciones del procesador: una para asignar memoria y tres para inicializar las propiedades del nodo (elemento y nodos anterior y siguiente). También

siempre incrementa el número de elementos en la lista, que son otras 2 instrucciones (la suma y la asignación de la variable). Por último, siempre se revisa el rabo de la lista para saber si es vacío o no, que es 1 instrucción más.

El resto de las instrucciones ejecutadas depende del estado de la lista al momento de agregar el elemento. Si la lista es vacía (en otras palabras, si el rabo es vacío), se realizan 2 asignaciones, una al rabo y otra a la cabeza de la lista. Si la lista no es vacía, se realizan 3 asignaciones: una al anterior del nuevo nodo, otra al siguiente del rabo y por último se actualiza el rabo mismo.

Por lo tanto (y haciendo énfasis de nuevo en que estamos simplificando lo que ocurre), el algoritmo ejecuta 9 o 10 instrucciones, dependiendo del estado de la lista al momento de ejecutar el algoritmo. Lo primero que haremos será ocuparnos únicamente del peor de los casos; esto sirve dos propósitos. El primero es que necesitamos poder definir una función que dado el tamaño de la entrada del algoritmo nos regrese el número de instrucciones que ejecuta. Una función no puede regresar más de un valor, así que elegimos el valor máximo porque cubre todos los demás en lo que queremos medir, que es el número de instrucciones ejecutadas. El segundo propósito es que en general, sin saber exactamente cómo será la entrada de nuestros algoritmos, nos conviene ser pesimistas y suponer que el peor de los casos será el que ocurra. Si el algoritmo se comporta de manera respetable en el peor de los casos, entonces no tenemos de qué preocuparnos en el resto. Así que vamos a decir que el algoritmo [6.1](#) ejecuta 10 instrucciones, porque ese es el peor de los casos.

En otro ejemplo similar tenemos el algoritmo para obtener la longitud de la lista, que en nuestra implementación es una única instrucción; el regresar el contador interno de elementos (ni siquiera mostraremos el algoritmo).

La notación de O grandota lo que hace es considerar que estos dos algoritmos pertenecen a la misma clase de complejidad en tiempo, $O(1)$; o, siendo más específicos, que ambos algoritmos tienen *la misma* complejidad en tiempo. Esto a pesar de que uno es (técnicamente) diez veces más lento que el otro (ya que ejecuta diez veces más instrucciones). Lo que nos dice que ambos algoritmos tengan complejidad en tiempo de $O(1)$ es justamente que el número de instrucciones que los algoritmos ejecutan (en el peor de los casos) siempre es el mismo, sin importar el tamaño de la lista y que además no nos interesa exactamente *cuántas* instrucciones sean, siempre y cuando el número sea constante (independiente del tamaño de la lista).

En cambio los algoritmos para copiar una lista y eliminar un elemento de ella tendrán complejidad en tiempo de $O(n)$. Obviamente copiar una lista será más lento que eliminar un elemento de ella, incluso si el elemento no se encuentra

en la lista (lo que nos fuerza a recorrerla toda), dado que tenemos que asignar memoria para la nueva lista y sus nuevos nodos; pero eso no nos debe importar demasiado, porque al fin y al cabo tardan en tiempo comparativamente lo mismo: ambos algoritmos (en el peor de los casos en el caso de eliminar) recorren toda la lista y por lo tanto ejecutan cn instrucciones, donde c es una constante fija distinta para cada algoritmo. Desde un punto de vista teórico cuál sea esa c exactamente no es importante¹.

6.1. La notación de O grandota

Vamos a definir formalmente a la notación $O(f(n))$, donde $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, pero primero vamos a hacer hincapié en que $O(f(n))$ es de hecho un conjunto de funciones de los reales positivos en los reales positivos. Cuando digamos que un algoritmo X es $O(f(n))$ en tiempo o que tiene complejidad $O(f(n))$ en tiempo (para alguna $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$), lo que esto querrá decir es: si el número de instrucciones que toma ejecutar X en el peor de sus casos está dado por la función $g(n)$, $g : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, donde n es el tamaño de la entrada de X , entonces $g(n) \in O(f(n))$. Obviamente el número de instrucciones que un algoritmo ejecuta siempre es un entero positivo y el tamaño de la entrada de un algoritmo será también siempre un entero positivo porque es espacio en memoria y ésta es discreta, pero la definición de $O(f(n))$ es equivalente si utilizamos reales positivos, así que de una vez la hacemos general.

Habiendo aclarado esto, el conjunto $O(f(n))$ se define de la siguiente manera:

Definicion 6.1. (O grandota) Sean $f, g : \mathbb{R}^+ \rightarrow \mathbb{R}^+$; $g(n) \in O(f(n))$ si y sólo si existen $c, n_0 \in \mathbb{R}^+$ tales que:

$$g(n) \leq cf(n) \quad \forall n \geq n_0.$$

En otras palabras, si podemos encontrar constantes fijas c y n_0 en los reales positivos, tales que $g(n)$ siempre es menor o igual que $f(n)$ multiplicado por la constante c , a partir de que n sea mayor o igual a n_0 , entonces $g(n) \in O(f(n))$. O, simplificando, con una constante c adecuada siempre podemos hacer de $cf(n)$ una cota superior de $g(n)$, para valores de n suficientemente grandes.

Esto último es importante; en complejidad computacional, el comportamiento que nos interesa de una función $f(n)$ es el asintótico cuando n tiende a infinito. Cómo se comporte al inicio (para valores “pequeños”) no le damos tanta importancia; nos interesa el comportamiento cuando la n es no acotada.

Por ejemplo, demostremos que $3n^3 + 40n^2 + n + 7 \in O(n^3)$. Esto significa

que debemos encontrar $c > 0$ y $n_0 > 0$ tales que $3n^3 + 40n^2 + n + 7 \leq cn^3$, $\forall n \geq n_0$.

Primero observamos que las siguientes desigualdades son trivialmente ciertas para $n \geq 1$:

$$3n^3 \leq 3n^3, \quad 40n^2 \leq 40n^3, \quad n \leq n^3, \quad 7 \leq 7n^3$$

Por lo tanto si sumamos por ambos lados las cuatro desigualdades obtenemos:

$$3n^3 + 40n^2 + n + 7 \leq 3n^3 + 40n^3 + n^3 + 7n^3 = 51n^3 \quad \forall n \geq 1.$$

Por lo que $c = 51$ y $n_0 = 1$ sirven para que se cumpla la definición de que $3n^3 + 40n^2 + n + 7 \in O(n^3)$. No importa que para valores pequeños de n ocurra $3n^3 + 40n^2 + n + 7 > 51n^3$; lo importante es que a partir de un valor fijo (1 en este caso), $3n^3 + 40n^2 + n + 7$ **siempre** es menor o igual que $51n^3$ (figura 6.1).

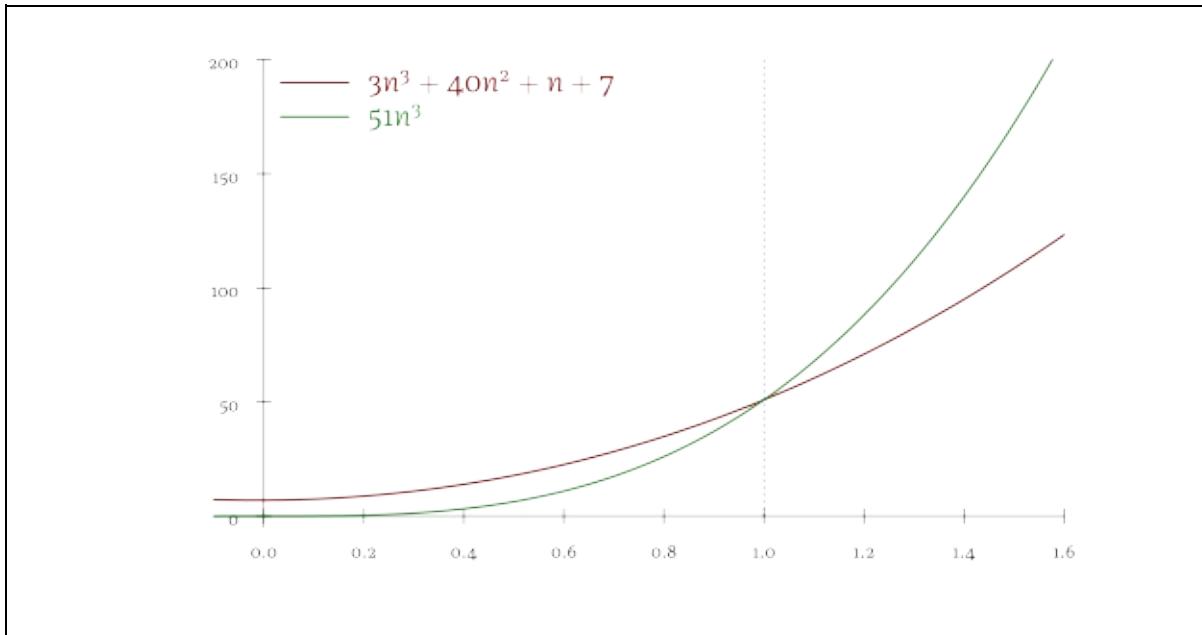


Figura 6.1: La función $2n^3$ acota por arriba a $3n^3 + 40n^2 + n + 7$ si $n \geq 1$.

De la definición se sigue que $O(1)=O(2)=O(\pi)=O(2^e)$ etcétera, para cualquier valor constante; por convención se toma 1. También se sigue que

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \dots$$

y por lo tanto si $g(n) \in O(1)$, entonces $g(n) \in O(n \log n)$. Lo interesante por supuesto es el conjunto $O(f(n))$ con la $f(n)$ menor (porque estamos buscando una cota superior), así que por convención decimos que $g(n)$ es $O(1)$, aunque

técnicamente también sea $O(n \log n)$.

Si $g(n) \in O(f(n))$, diremos que $g(n)$ es del orden de $O(f(n))$; o más informalmente que $g(n)$ es $O(f(n))$; o incluso (si el contexto es claro) que $g(n)$ es $f(n)$; en este libro trataremos de no usar esta última forma, pero es muy común en la literatura.

La notación de la O grandota es de las más abusadas en Ciencias de la Computación y no es raro encontrar libros y artículos donde técnicamente está mal utilizada; por ejemplo, es común hallar un artículo donde dicen que una función $g(n)$ es:

$$O(1) \leq g(n) \leq O(\log n) \quad \leftarrow \text{no hagan esto, por favor.}$$

Lo que los autores quieren decir es que la función $g(n)$ es en algunos casos constante y en otros logarítmica, pero esa notación de arriba no tiene ningún sentido bajo la definición de $O(f(n))$, particularmente cuando O grandota *siempre* habla del peor de los casos por omisión. Sin embargo, si el algoritmo X tiene una complejidad $O(f_1(n))$ en tiempo, el algoritmo Y tiene una complejidad $O(f_2(n))$ en tiempo y $f_1(n) < f_2(n)$ a partir de cierta n_c constante, entonces sí será muy común decir que el algoritmo X tiene una complejidad en tiempo *menor* que el algoritmo Y. Sólo nunca debe denotarse como que $O(f_1(n)) < O(f_2(n))$, porque bajo la definición de O grandota eso no tiene sentido.

Lo importante es que aunque puede no ser obvio por cómo se utiliza en lenguaje hablado y escrito, $O(f(n))$ *siempre* es un conjunto de funciones. Por último (y aunque tal vez redundante), debe quedar claro lo siguiente:

$$f(n) \in O(n) \neq f(n) \in O(1).$$

En otras palabras aunque un lado de la contención sí es verdadera ($f(n) \in O(1)$ implica que $f(n) \in O(n)$), el otro no lo es.

La notación de O grandota es miembro de una familia de notaciones que incluye a *o* chiquita ($o(f(n))$), *theta* ($\Theta(f(n))$), *omega* ($\Omega(f(n))$) y otras, donde todas son conjuntos de funciones. La familia fue inventada por Paul Bachmann [2], Edmund Landau [23] y otros en lo que colectivamente se denomina la notación Bachmann-Landau o (más comúnmente) notación asintótica.

6.2. Complejidades en tiempo y en espacio

Formalmente definiremos como sigue a la complejidad en tiempo de un algoritmo:

Definicion 6.2. (Complejidad en tiempo de un algoritmo) *Sea F un algoritmo. Si el número de instrucciones que F ejecuta en el peor de los casos, para una entrada de tamaño n , está dado por la función $g(n)$ y $g(n) \in O(f(n))$, entonces diremos que F tiene complejidad en tiempo $O(f(n))$.*

Cuando un algoritmo tiene complejidad en tiempo $O(f(n))$ también lo podremos decir como que el algoritmo es $O(f(n))$ en tiempo. La función $g(n)$ de la definición normalmente no será ni siquiera mencionada y quedará implícita.

La complejidad en espacio es básicamente lo mismo, excepto por un detalle para nosotros: para las estructuras de datos que se cubren en este libro, que por convención el número de elementos que todas contienen siempre será n , la complejidad en espacio estará dada por la memoria que utiliza el algoritmo *además* de la memoria utilizada por los n elementos en la estructura: hay que recordar que en Orientación a Objetos, el objeto que manda llamar un método se considera parte de la entrada del algoritmo o algoritmos que implementa dicho método; en Java en particular el objeto que ejecuta un método siempre está disponible *dentro* del mismo con la referencia [this](#).

Formalmente:

Definicion 6.3. (Complejidad en espacio de un algoritmo) *Sea F un algoritmo. Si el número de localidades de memoria que F utiliza, en el peor de los casos, para una entrada de tamaño n (además de la entrada misma), está dado por la función $g(n)$ y $g(n) \in O(f(n))$, entonces diremos que F tiene complejidad en espacio $O(f(n))$.*

Siguiendo esta definición, el algoritmo AGREGAFINAL al inicio del capítulo es $O(1)$ en espacio (porque crea únicamente un nuevo nodo), pero el algoritmo REVERSA (que regresa una copia de la lista con los elementos en orden invertido) es $O(n)$ en espacio (porque crea n nuevos nodos).

En general, cuando también consideraremos la memoria utilizada por la estructura de datos, como se necesita al menos una instrucción para leer o escribir en una localidad de memoria, la complejidad en espacio *siempre* será menor o igual a la complejidad en tiempo; o más formalmente, si el algoritmo X es $O(f(n))$ en tiempo y $O(g(n))$ en espacio, entonces siempre ocurrirá que $g(n) \leq f(n)$ a partir de cierta n_c constante. Haciendo hincapié en el abuso que suele sufrir la notación de O grandota, *nunca* diremos que $O(f(n)) \leq O(g(n))$, porque ambos son conjuntos.

Con nuestra definición, los algoritmos `ELIMINA` y `CONTIENE` son $O(1)$ en memoria porque no utilizan memoria proporcional al tamaño de la lista, sólo la recorren. Esto es cierto para las versiones iterativas y recursivas del algoritmo. Sin embargo, los métodos `elimina()` y `contiene()` de nuestra clase `Lista` consumen $O(1)$ de memoria en sus versiones iterativas, pero $O(n)$ en sus versiones recursivas. Esto es por la pila de ejecución; si los métodos son recursivos, entonces el recorrer toda la lista causará que se carguen n registros de activación en la misma.

Como mencionábamos en el capítulo anterior, esto no es muy grave, porque de cualquier manera la lista consume $O(n)$ de memoria, sólo no la contamos en el análisis del consumo de memoria de nuestros algoritmos para poder diferenciar los que consumen más *además* de nuestras estructuras de datos. Pero sí es necesario tenerlo en cuenta y habrá algoritmos en los que no tendrá sentido implementarlos recursivamente por el costo en memoria (ciertamente pequeño, en general) que implicaría.

También hay que entender que la regla no es universal; no es cierto que *siempre* consuma más memoria la versión recursiva de un algoritmo comparada con la iterativa: en muchos casos, la única manera de emular la recursión de forma iterativa obligará a que utilicemos una estructura de datos auxiliar (y, como debe ser obvio, esa estructura muchas veces será una pila) que consumirá $O(n)$ de memoria.

A lo largo de este libro explicaremos las complejidades en tiempo y en espacio de los algoritmos asociados a las estructuras de datos que se vayan cubriendo; sin embargo hacemos notar que dichas explicaciones en general serán informales, sin cubrir las demostraciones matemáticas formales correspondientes. Las demostraciones matemáticas formales deben ser cubiertas en un curso de Análisis de Algoritmos y quedan (en su mayoría) fuera del alcance de un curso de Estructuras de Datos.

Ejercicios

1. Demuestra que $7n^3 - 5n^2 + 3n - 1 \in O(n^3)$.
 2. Demuestra que $n^{1 + \varepsilon} \notin O(n)$, $\varepsilon > 0$.
 3. ¿Cuál es la complejidad en tiempo del método `next()` de la clase `Lista`. `Iterador`?
 4. ¿Cuál es la complejidad en tiempo del método `inserta()` de la clase `Lista`?
 5. ¿Cuál es la complejidad en espacio del método `inserta()` de la clase `Lista`?
-

1. Desde un punto de vista *práctico* sí puede llegar a ser importante, pero incluso en la práctica es muy extraño que la constante oculta en la notación de O grandota juegue un papel fundamental en el diseño e implementación de un sistema para producción.←

7. Arreglos

Como mencionamos en el capítulo 1, los arreglos (*arrays* en inglés) son sencillamente segmentos continuos de memoria; si un arreglo A es de n elementos de tipo T y el tipo T utiliza k bytes por instancia, entonces el segmento continuo de memoria utilizado por el arreglo A mide nk bytes. Para simplificar la notación vamos a suponer, sin pérdida de generalidad, que $k = 1$; sólo debemos multiplicar por k en otro caso.

Básicamente todos los lenguajes de programación con soporte para arreglos¹ utilizan una sintaxis de corchetes (“[” y “]”) para acceder elementos del arreglo (una excepción importante es Fortran 77, que utiliza paréntesis).

El lenguaje de programación Java (como todos los lenguajes descendientes de C) también utiliza corchetes; pero además les agrega mejoras en tiempo de compilación y de ejecución. En Java los arreglos son objetos con comportamiento y variables de clase (una de las cuales, `length`, nos dice la longitud o capacidad del arreglo) y en tiempo de ejecución pueden detectar si tratamos de acceder a un índice fuera de rango del arreglo.

Fuera de estas mejoras, los arreglos en Java son igual de simples que en todos los lenguajes de programación que los soportan. Lo cual por supuesto es, al mismo tiempo, su mayor ventaja y limitación.

Los arreglos fueron la primera estructura de datos utilizada por programadores, por la sencilla razón de que segmentos continuos de memoria eran realmente lo único que tenían disponible. Como estructura de datos los arreglos funcionan y se pueden utilizar de manera muy similar a las listas, con unas cuantas diferencias muy importantes.

Todas las diferencias entre arreglos y listas se derivan del hecho de que los arreglos, por definición, son una estructura de datos estática; una vez asignado el segmento de memoria, éste no se puede cambiar: no se puede reducir ni agrandar; algunos lenguajes ofrecen facilidades para reducir o agrandar la capacidad de un arreglo, pero en el fondo siempre es necesario (al menos en algunos casos) asignar memoria de nuevo y copiar el arreglo original al recién creado. Las listas en cambio son dinámicas; crecen tanto como queramos (y la memoria disponible nos lo permita) y podemos reducir la memoria que utilizan al eliminar elementos, aunque en el caso particular de Java el recolector de basura decide de manera independiente al programador cuándo de hecho esa memoria vuelve a estar disponible.

La naturaleza estática de los arreglos nos impide poder agregarles un número

(potencialmente) arbitrario de elementos; pero también nos permite acceder al i -ésimo elemento del arreglo en tiempo constante: como decíamos en la introducción del libro, si A es la dirección de memoria del arreglo, entonces el i -ésimo elemento del mismo está en la dirección de memoria $A + i$. Recordemos que estamos suponiendo $k = 1$, donde k es el tamaño del tipo del arreglo; si $k > 1$, entonces la dirección es $A + ik$.

```
String[] A = new String[3];
A[0] = "hola";
A[1] = "mundo";
A[2] = "cruel";
String[] B = { "hola", "mundo", "cruel" };
```

Listado 7.1: El arreglo de cadenas A se crea con `new` y se llena explícitamente; el arreglo B se crea y se llena implícitamente.

Para crear un arreglo en Java podemos utilizar el operador `new` y agregarle elementos explícitamente uno por uno, como se hace con el arreglo A en el listado [7.1](#); pero también podemos crearlos y poblarlos implícitamente utilizando llaves, como se ve con el arreglo B en el listado [7.1](#).

Los arreglos multidimensionales en Java utilizan un par de corchetes para cada dimensión y también pueden ser declarados y llenados explícita e implícitamente, como se ve en el listado [7.2](#).

```
String[][] A = new String[2][2];
A[0][0] = "a";
A[0][1] = "b";
A[1][0] = "c";
A[1][1] = "d";
String[][] B = { { "a", "b" }, { "c", "d" } };
```

Listado 7.2: El arreglo bidimensional A se crea con `new` y se llena explícitamente; el arreglo B se crea y se llena implícitamente.

7.1. El polinomio de redirecciónamiento

Conceptualmente los arreglos A y B del listado [7.2](#) se verían como en la figura [7.1](#); pero realmente en la memoria estarían como en la figura [7.2](#).

"a"	"b"
"c"	"d"

Figura 7.1: Un arreglo bidimensional, conceptualmente.

"a"	"b"	"c"	"d"
-----	-----	-----	-----

Figura 7.2: Un arreglo bidimensional en memoria.

Esto es una generalización de los arreglos unidimensionales; si A es un arreglo bidimensional de $n_1 \times n_2$, entonces ocupa $n_1 n_2$ bytes en memoria (suponiendo un tamaño de tipo $k = 1$; de otra forma es $n_1 n_2 k$). Y si A es la dirección en memoria del arreglo, en A (o $A + 0$ o $A + 0n_2$) comienza el primer renglón del mismo, en $A + n_2$ (o $A + 1n_2$) comienza el segundo, en $A + 2n_2$ comienza el tercero y así hasta llegar a $A + (n_1 - 1)n_2$. Un elemento con índices i y j , $0 \leq i < n_1$ y $0 \leq j < n_2$, ($A[i][j]$ en notación de Java) se encuentra en la dirección de memoria $A + in_2 + j$. Si el tamaño en bytes del tipo k del arreglo es mayor que 1, entonces la dirección es $A + k(in_2 + j)$.

Todo esto sigue siendo válido para dimensiones mayores a dos; en un arreglo A de d dimensiones, donde la dimensión s ($1 \leq s \leq d$) es de tamaño n_s (en notación de Java, $A = \text{new } T[n_1][n_2] \dots [n_d]$), el arreglo mide $n_1 \times \dots \times n_d = \prod_{s=1}^d n_s$ bytes (o $k \prod_{s=1}^d n_s$ si $k > 1$). Si la dirección en memoria del arreglo es A , la dirección de la entrada con índices i_1, \dots, i_d , $0 \leq i_s < n_s$, (en notación de Java, $A[i_1][i_2] \dots [i_d]$), es

$$A + i_1 \prod_{s=2}^d n_s + i_2 \prod_{s=3}^d n_s + \dots + i_{d-2} \prod_{s=d-1}^d n_s + i_{d-1} n_d + i_d. \quad (7.1)$$

El polinomio que se le suma a A (su desplazamiento) en la ecuación 7.1 se le conoce como el *polinomio de redirecccionamiento* de un arreglo y nos permite, en tiempo constante, acceder a cualquier elemento de un arreglo multidimensional lleno. Si definimos a $\prod_{s=d+1}^d n_s$ como 1, podemos simplificar el polinomio de redirecccionamiento como se muestra en la ecuación 7.2.

$$\sum_{t=1}^d i_t \prod_{s=t+1}^d n_s. \quad (7.2)$$

Recordemos que estamos suponiendo un tamaño de tipo k igual a 1; si no, hay que multiplicar la ecuación 7.2 por k para obtener el desplazamiento de la dirección de la entrada. Hacemos notar que si las n_s son fijas, que es el caso con todos los arreglos de un programa de computadora, entonces el polinomio es lineal.

Vamos a ver un ejemplo concreto; supongamos que creamos un arreglo de

$3 \times 2 \times 4$ en Java, como se muestra en el listado 7.3.

```
int[][][] A =
{
    {
        {
            { 1, 2, 3, 4},
            { 5, 6, 7, 8},
        },
        {
            { 9, 10, 11, 12 },
            { 13, 14, 15, 16 },
        },
        {
            { 17, 18, 19, 20 },
            { 21, 22, 23, 24 },
        }
    };
}
```

Listado 7.3: Arreglo de enteros A de $3 \times 2 \times 4$.

Conceptualmente el arreglo A son 3 arreglos bidimensionales de tamaño 2×4 y los arreglos $A[0]$, $A[1]$ y $A[2]$ cada uno son dos arreglos de tamaño 4. Conceptualmente el arreglo A se ve como en la figura 7.3

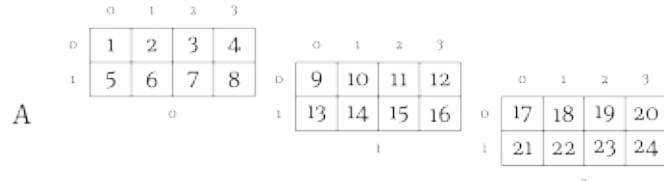


Figura 7.3: El arreglo A de $3 \times 2 \times 4$, conceptualmente.

El punto de explicar el polinomio de redireccionamiento es que, aunque visualmente útil, la figura 7.3 no representa cómo es el arreglo A en memoria realmente. En realidad el arreglo A se ve en memoria como muestra la figura 7.4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Figura 7.4: El arreglo A de $3 \times 2 \times 4$, como realmente está en memoria.

El polinomio de redireccionamiento nos dice que si queremos acceder el elemento $A[2][1][3]$, entonces debemos ir a la dirección de memoria A y sumar (o desplazarnos por) el polinomio; en otras palabras resolver la ecuación 7.2 donde $d = 3$, $n_1 = 3$, $n_2 = 2$, $n_3 = 4$, $i_1 = 2$, $i_2 = 1$ e $i_3 = 3$:

$$\sum_{t=1}^d i_t \prod_{s=t+1}^d n_s = 2(2 \times 4) + 1(4) + 3 = 16 + 4 + 3 = 23.$$

En la figura 7.3 podemos ver que $A[2][1][3]$ corresponde al tercer arreglo de 2×4 (el que tiene índice 2), al segundo renglón (el que tiene índice 1) y de éste la cuarta entrada (la que tiene índice 3), que corresponde a 24. Lo que ocurre en realidad es que al acceder el elemento $A[2][1][3]$, lo que se hace es brincar a la dirección A y desplazarse 23 lugares para acceder al mismo elemento 24, como se ve en la figura 7.4.

El polinomio de redireccionamiento únicamente funciona en arreglos multidimensionales que sean *llenos dimensionalmente* o sencillamente *llenos*. Un arreglo multidimensional de d dimensiones, donde la dimensión s es de tamaño n_s , es lleno si y sólo si es un hiperrectángulo de $n = \prod_{s=1}^d n_s$ elementos.

Existen lenguajes de programación que únicamente soportan arreglos multidimensionales llenos (los arreglos unidimensionales son trivialmente llenos siempre). A los arreglos multidimensionales que no son llenos se les suele llamar escalonados (*jagged* en inglés) y Java es de los lenguajes que los soporta. Un ejemplo de un arreglo bidimensional escalonado en Java se ve en el listado 7.4; como los llenos, Java puede crear arreglos escalonados de manera explícita e implícita.

```
String[][] A = new String[3][];
A[0] = new String[1];
A[0][0] = "a";
A[1] = new String[3];
A[1][0] = "b";
A[1][1] = "c";
A[1][2] = "d";
A[2] = new String[2];
A[2][0] = "e";
A[2][1] = "f";
String[][] B = { { "a" },
                 { "b", "c", "d" },
                 { "e", "f" } };
```

Listado 7.4: El arreglo de cadenas A se crea con `new` y se llena explícitamente; el arreglo B se crea y se llena implícitamente.

En los arreglos escalonados la memoria total del arreglo no es *necesariamente* contigua; por ejemplo, el arreglo del listado 7.4 podría tener su memoria organizada de una manera similar a como se muestra en la figura 7.5.

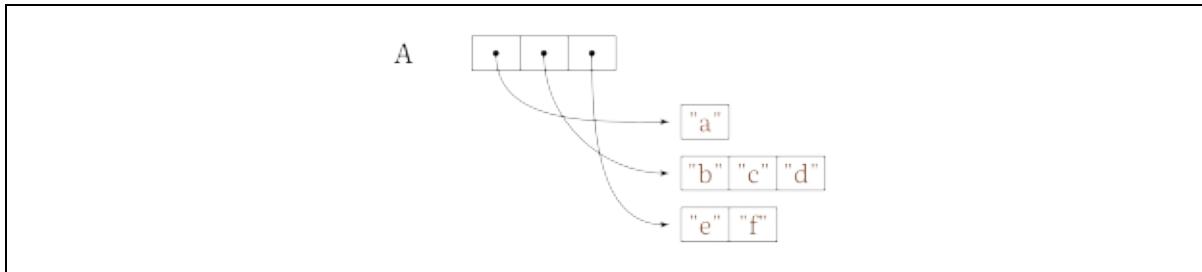


Figura 7.5: La (posible) memoria de un arreglo escalonado.

La complejidad en tiempo de acceder a un elemento de un arreglo escalonado es $O(1)$, igual que en un arreglo lleno, pero es un poco más lento; en un arreglo escalonado nos movemos a la dirección del arreglo y nos desplazamos el índice de la primera dimensión; nos movemos a esa dirección y nos desplazamos el índice de la segunda dimensión; etcétera. Esto se hace d veces, una vez por cada dimensión. En un arreglo lleno resolver el polinomio de redirecccionamiento toma $2d$ multiplicaciones y d sumas; pero nos movemos una única vez y nos desplazamos una única vez. Sumas y restas en casi todas las arquitecturas modernas son más rápidas que moverse a una dirección de memoria y desplazarse, por lo que es más rápido acceder a una entrada de un arreglo lleno que de un arreglo escalonado. Pero ambos toman tiempo $O(1)$, porque d siempre es un número constante conocido en tiempo de compilación (el número de *dimensiones* del arreglo es conocido en tiempo de compilación; no necesariamente qué tamaño n_s , $1 \leq s \leq d$, tendrá cada dimensión).

Los arreglos son muy útiles para cosas muy sencillas; pero si el problema a resolver requiere un manejo de datos no trivial, lo más probable es que a la larga las estructuras globales del sistema tengan que ser más poderosas que arreglos. Pero para cosas locales y pequeñas en general funcionan muy bien.

En este libro utilizaremos arreglos para ver algoritmos de ordenamiento y de búsqueda en los capítulos [10](#) y [11](#) y como estructura auxiliar en dos estructuras de datos en los capítulos [18](#) y [21](#). Fuera de esos capítulos, los arreglos no jugarán un papel muy importante en ninguna estructura, pero mencionaremos cuándo pudiera tener sentido que lo jugaran.

También debe quedar claro que todos y cada uno de los algoritmos que hemos visto y que veremos en este libro pueden ser implementados utilizando únicamente arreglos. Nada más suele ser lento y propenso errores, además de que definitivamente no ayuda a la reutilización de código.

7.2. Arreglos y genéricos

Para terminar con el tema de arreglos, debemos mencionar que los arreglos y

los genéricos en Java no se llevan muy bien, por razones principalmente históricas en lugar de técnicas.

Recordemos que si tenemos una clase genérica cuyo parámetro genérico es `T`, nunca podemos hacer dentro de ella lo que se muestra en el listado 7.5.

```
T t = new T();
```

Listado 7.5: No podemos instanciar `T`, porque podría ser una interfaz o clase abstracta.

El tipo `T` podría ser una interfaz o una clase abstracta; e incluso si fuera una clase concreta, podría ser que no tuviera un constructor sin parámetros. No podemos saberlo cuando *compilemos* la clase genérica, sólo cuando compilemos *otras clases* que la usen.

Sin embargo, en principio el código del listado 7.6 sí debería poder compilarse.

```
T[] a = new T[5];
```

Listado 7.6: Tampoco podemos instanciar un arreglo de `T`, por razones históricas.

Aunque no sepamos el tipo `T`, sabemos que es una referencia y todas las referencias en Java utilizan el mismo tamaño en memoria. Entonces asignar el segmento continuo de memoria para el arreglo genérico debería ser posible.

Pero no lo es. Por razones de eficiencia, los arreglos en Java por omisión no son arreglos de referencias; son arreglos de los *objetos* mismos. Con esto la máquina virtual se ahorra un brinco de memoria: si los arreglos fueran de referencias, tendríamos que brincar al inicio del arreglo, desplazarnos a la dirección del índice y después brincar a la dirección marcada en esa referencia. Si los arreglos son de los objetos mismos, ese último brinco puede omitirse; y suena poco pero en el momento de crear el lenguaje a inicios de la década de los noventa del siglo pasado sonaba como un ahorro razonable y sencillo de hacer.

Hay forma de darle la vuelta a este problema; al fin y al cabo Java soporta arreglos de tipo de alguna interfaz o clase abstracta y con esos tipos tampoco sabemos el tamaño en memoria de cada instancia hasta que la interfaz sea implementada o la clase abstracta extendida concretamente. En estos casos los arreglos sí son de referencias y podemos forzar al compilador a hacer algo de este tipo con nuestros arreglos genéricos, como lo veremos en los capítulos 18 y 21.

Pero *por omisión* los arreglos son de objetos, no de referencias a objetos; y como hay millones de líneas de código de Java en producción y una porción

muy significativa viene de la época en que no había genéricos, el compilador sigue funcionando igual para poder soportar ese código viejo.

Esto causa que utilizar arreglos con tipos genéricos se vuelva ligeramente engorroso. Que es otra de las razones por las cuales utilizaremos poco los arreglos en este libro.

Para terminar con este capítulo, mencionamos que es posible escribir una clase `Lista` que utilice arreglos como estructura auxiliar subyacente, en lugar de implementarlas como nodos doblemente ligados. En Java de hecho el tipo `List` es una interfaz, la cual es implementada por las clases `LinkedList` y `ArrayList` entre otras. La clase `LinkedList` (como mencionábamos en el capítulo anterior) es muy similar a nuestra clase `Lista`; y la clase `ArrayList` ofrece el mismo comportamiento pero usando un arreglo como estructura auxiliar subyacente.

Las listas implementadas con arreglos son favorecidas por muchos programadores. Sólo hay que tener cuidado al usarlas que a veces un incremento en el número de los elementos de la lista puede tener complejidad en tiempo $O(n)$. Las listas doblemente ligadas no tienen este problema; pero incurren en una (muy pequeña) penalización en tiempo al tener que seguir referencias al recorrerlas. Y por supuesto no se puede obtener el i -ésimo elemento de una lista doblemente ligada en tiempo $O(1)$.

Ejercicios

1. Menciona una ventaja que tengan los arreglos sobre las listas.
2. Menciona una ventaja que tengan las listas sobre los arreglos.
3. ¿En qué difieren los arreglos multidimensionales llenos y los escalonados?
4. Dado el arreglo en Java creado con la instrucción `int[][]a = new int[7][3][5][2]`, desarrolla el polinomio de redirecciónamiento para la entrada `a[5][2][3][1]`.
5. ¿Por qué Java no permite crear fácilmente arreglos de tipos genéricos?

1. Es relativamente común que los lenguajes funcionales y lógicos no soporten arreglos o que le hagan difícil al programador el usarlos directamente.[←](#)

8. Pilas y colas

Las pilas y las colas (*stacks* y *queues* en inglés) son dos estructuras de datos relativamente sencillas que se utilizan para priorizar información. Usando un par de analogías simples, una cola es literalmente como la cola de las tortillas: la gente llega por su kilo de tortillas y sale de ahí en el mismo orden en que llegó; por lo tanto el primer elemento que entra en la cola es el primero que sale. Las colas suelen representarse como se muestra en la figura 8.1.



Figura 8.1: Una cola después de agregar a, b, c, d, e, f, g, h , sacar dos elementos y agregar i, j, k, l ; el primero que entra es el primero que sale.

Una pila en cambio es como una pila de libros: se van apilando los libros de abajo hacia arriba y si quitamos alguno siempre es el que está en el tope; entonces el primer elemento de la pila que entra siempre es el último que sale. Las pilas suelen representarse como se muestra en la figura 8.2.

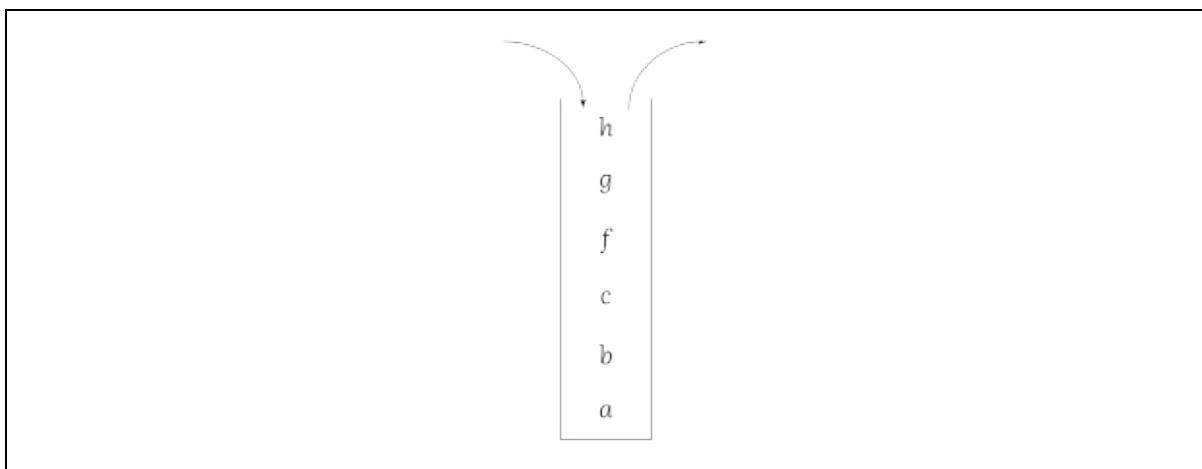


Figura 8.2: Una pila después de agregar a, b, c, d, e , sacar dos elementos y agregar f, g, h ; el primero que entra es el último que sale.

En inglés a estas reglas se le conocen como FIFO y FILO, respectivamente; *First In First Out* y *First In Last Out*: el primero que entra es el primero que sale y el primero que entra es el último que sale.

Las pilas y colas son utilizadas cuando queremos darle cierta prioridad a datos. Con pilas el ejemplo con el que deben estar más familiarizados es con la pila de ejecución: un programa de Java siempre tiene al método `main()` en el fondo de la pila de ejecución¹ y cada llamada a un método introduce un nuevo registro de activación en la misma. El método ejecutándose siempre está en el

tope de la pila y cuando termina se descarga y regresa al que lo mandó llamar. Esto permite guardar en orden el estado local de cada método y cosas más sofisticadas como saber cómo atrapar excepciones.

Las colas suelen ser utilizadas por programas proveedores de servicios; cada petición de servicio se agrega a una cola (o se encola) y el servidor va atendiendo peticiones una por una sacándolas de la cola. Así las peticiones son atendidas en el orden de llegada.

En un sentido pragmático, ya tenemos pilas y colas con nuestras listas; para usar una cola, agregamos siempre al final y eliminamos al inicio (o al revés); para usar una pila, agregamos siempre al final y eliminamos al final (o ambas al inicio).

Sin embargo vamos a implementar clases específicas para pilas y colas, no sólo por completez sino porque sí existen diferencias que nos interesa enfatizar entre pilas, colas y listas. La primera es que las pilas y las colas no son colecciones; no queremos poder eliminar elementos arbitrarios de nuestras pilas y colas porque eso afectaría el orden de sus elementos (que es probablemente su propiedad más importante). La segunda es que tampoco queremos que sean iterables; tradicionalmente pilas y colas únicamente muestran al mundo un elemento: el que está al frente de la cola y el que está en el tope de la pila.

Podríamos implementar estas restricciones utilizando una interfaz que no fuera iterable y que sólo ofreciera operaciones para agregar, eliminar y mirar el elemento al frente o en el tope; pero eso sólo podría funcionar para una de las dos estructuras y la otra habría que programarla de cualquier manera.

En Java esto es lo que termina ocurriendo: la clase `LinkedList` implementa la interfaz `queue` y la clase `Stack` existe aparte, extendiendo `vector`, que a su vez implementa `List`.

Nosotros implementaremos clases `Pila` y `cola` específicas; y como hicimos con las listas en el capítulo [5](#) vamos primero a definir el comportamiento que debe de tener cada una.

Aunque estas estructuras no serán colecciones sí queremos que sean genéricas. Además necesitamos operaciones para agregar y eliminar elementos; en inglés en colecciones estas operaciones son *add* y *remove*, pero para pilas y colas se utilizan otros nombres. Los nombres en inglés para las operaciones en pilas son *push* (para agregar elementos) y *pop* (para eliminar del frente o del tope); en colas son *enqueue* (literalmente *encolar* para agregar al final) y *dequeue* para eliminar al frente. Nosotros vamos a utilizar los mismos nombres para ambas estructuras (por razones que en un momento se harán obvias); a falta de mejores términos, nosotros utilizaremos “mete” y

“saca”. Además queremos poder ver el elemento en el frente o tope de la estructura sin eliminarlo; la operación en inglés suele denominarse *look* o *peek*, así que nosotros usaremos “mira”. Por último vamos a necesitar un método para determinar si la estructura es vacía.

Todo esto nos lleva a un comportamiento, por ejemplo para **Pila**, como se muestra en el listado [8.1](#).

```
public class Pila<T> {  
    public void mete(T elemento) { /* ... */ }  
    public T saca() { /* ... */ }  
    public T mira() { /* ... */ }  
    public boolean esVacia() { /* ... */ }  
    @Override public String toString() { /* ... */ }  
    @Override public boolean equals(Object o) { /* ... */ }  
}
```

Listado 8.1: Comportamiento de la clase **Pila**.

La definición de la clase **Pila** en el listado [8.1](#) está bien, sólo que *exactamente* la misma definición nos sirve para la clase **cola**. Y de hecho lo que ocurre es que ambas clases no sólo tienen el mismo comportamiento, sino que realmente son *idénticas* excepto por dos operaciones. O bien ambas clases agregan en un extremo de nuestra estructura y eliminan en extremos distintos; o bien agregan en extremos distintos y eliminan en el mismo. Y además la representación en cadena de pilas será diferente a la de colas.

La Orientación a Objetos en general nos da una mecánica que en esta situación se puede aplicar de manera perfecta: vamos a escribir una clase abstracta que implemente todas las operaciones de pilas y colas, pero que deje la operación de meter elementos abstracta. Las clases **Pila** y **Cola** entonces extenderán a la clase abstracta y los únicos métodos que implementarán serán **mete()** y **toString()**.

A la clase abstracta la llamaremos **MeteSaca**, porque en ambas estructuras esas serán las dos principales operaciones y su comportamiento sería entonces como en el listado [8.2](#).

```
public abstract class MeteSaca<T> {  
    abstract public void mete(T elemento);  
    public T saca() { /* ... */ }  
    public T mira() { /* ... */ }  
    public boolean esVacia() { /* ... */ }  
    @Override public boolean equals(Object o) { /* ... */ }  
}
```

Listado 8.2: Comportamiento de la clase `MeteSaca`.

Estructuralmente podríamos hacer que la clase `MeteSaca` tuviera una variable de clase protegida de tipo `Lista<T>` y entonces todos sus métodos serían triviales de implementar (así como los de `Pila` y `Cola`). Vamos a hacerlo entonces un poco más interesante y utilizaremos una implementación de listas simplemente ligadas; vamos a definir nodos para nuestras pilas y colas, nada más que estos nodos no tendrán la liga de regreso porque no la necesitamos. Una vez más tendremos un nodo cabeza y un nodo rabo y ambas estructuras sacarán eliminando la cabeza. Las pilas meterán por la cabeza y las colas por el rabo. Esta implementación utiliza un poco menos de memoria que usar una lista doblemente ligada (por la referencia hacia el nodo anterior que nos ahorramos); pero honestamente esa no es la razón para usarla: es únicamente que de verdad sería trivial la implementación utilizando una instancia de `Lista` dentro de cada pila y cola.

El esqueleto completo de la clase `MeteSaca` quedaría entonces como en el listado 8.3.

```
public abstract class MeteSaca<T> {
    protected class Nodo {
        public T elemento;
        public Nodo siguiente;
        public Nodo(T elemento) { /* ... */ }
    }

    protected Nodo cabeza;
    protected Nodo rabo;

    abstract public void mete(T elemento);
    public T saca() { /* ... */ }
    public T mira() { /* ... */ }
    public boolean esVacia() { /* ... */ }
    @Override public boolean equals(Object o) { /* ... */ }
}
```

Listado 8.3: Esqueleto de la clase `MeteSaca`.

Como las clases `Pila` y `Cola` necesitan acceso a los nodos, hacemos a la clase `Nodo` protegida, así como a las variables de clase `cabeza` y `rabo`.

Con la clase `MeteSaca` definida como en el listado 8.3, las clases `Pila` y `Cola` se definen como en los listados 8.4 y 8.5, respectivamente.

```
public class Pila<T> extends MeteSaca<T> {
    @Override public void mete(T elemento) { /* ... */ }
    @Override public String toString() { /* ... */ }
```

```
}
```

Listado 8.4: Esqueleto de la clase **Pila**.

```
public class Cola<T> extends MeteSaca<T> {  
    @Override public void mete(T elemento) { /* ... */ }  
    @Override public String toString() { /* ... */ }  
}
```

Listado 8.5: Esqueleto de la clase **cola**.

En la siguientes secciones veremos los algoritmos de todos los métodos, pero en general serán versiones simplificadas de los algoritmos correspondientes a la clase **Lista** que se vieron en el capítulo [5](#).

8.1. Algoritmos de la clase abstracta

La clase **MeteSaca** tiene únicamente cuatro métodos por implementar. Los cubriremos en el orden dado por el listado [8.3](#). Exceptuando el algoritmo para comparar (`equals()`), todos los algoritmos correspondientes tienen complejidad en tiempo constante y por lo tanto también complejidad en espacio constante.

- `saca()`

El nodo que vamos a sacar siempre es la cabeza; no podemos sacar el rabo porque no tenemos una referencia a su anterior, entonces no podemos recorrer el rabo a la izquierda.

Si la cabeza es \emptyset ocurre un error; si no, guardamos el elemento en la cabeza y movemos la cabeza a su siguiente. Si la cabeza se vuelve \emptyset actualizamos el rabo a \emptyset también y por último regresamos el elemento guardado (figura [8.3](#)). Esto incluye el caso cuando queda un único elemento en la estructura.

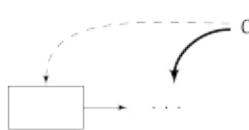


Figura 8.3: Los cambios al sacar.

- `mira()`

Si la cabeza es \emptyset ocurre un error; si no, regresamos el elemento en la cabeza.

- `esVacia()`

Regresamos el resultado de comparar la cabeza y \emptyset .

- `equals()`

El algoritmo es casi idéntico al del método `equals()` de la clase `Lista`, nada más que recorremos los nodos inmediatamente porque no tenemos la longitud de la estructura.

También aplica la misma nota técnica de la clase `Lista`; tenemos que suprimir las advertencias de genéricos porque hay que hacer una audición al objeto recibido.

En la clase `MeteSaca` sí importa que utilicemos el método `getClass()` en lugar de `instanceof`: implementamos el método `equals()` en la clase abstracta y entonces el algoritmo es el mismo para pilas que para colas. Por lo tanto no podemos sencillamente usar `instanceof`, ya que queremos que las pilas sólo se comparan con pilas y las colas con colas (todas las instancias de `Pila` y `Cola` son instancias de `MeteSaca` y por lo tanto el operador `instanceof` regresa `true` con ellas).

Si la clase del objeto recibido no es igual a la clase del objeto que invoca el método, regresaremos `false`.

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    MeteSaca<T> m = (MeteSaca<T>)o;
    // Aquí va el resto del método.
}
```

Listado 8.6: El inicio del método `equals()`, verificando la igualdad de clases y suprimiendo la advertencia relacionada con genéricos.

Con esto terminamos con la clase `MeteSaca`.

8.2. Algoritmos para pilas

Como decíamos arriba, las clases `Pila` y `Cola` únicamente necesitan implementar los métodos `mete()` (que es requerimiento de la clase abstracta `MeteSaca`) y `toString()` (porque todas nuestras estructuras lo implementarán).

- `mete()`

Las pilas meten del mismo extremo donde sacan, entonces en pilas hay que meter por la cabeza.

Como siempre con nuestras estructuras de datos, si el elemento a meter es \emptyset entonces ocurre un error.

Si el elemento no es vacío creamos un nodo n con él y tenemos dos casos:

1. La cabeza es \emptyset . En este caso actualizamos la cabeza y el rabo a n y terminamos.
2. La cabeza no es \emptyset . Hacemos que el siguiente de n sea la cabeza y actualizamos la cabeza a n (figura 8.4).

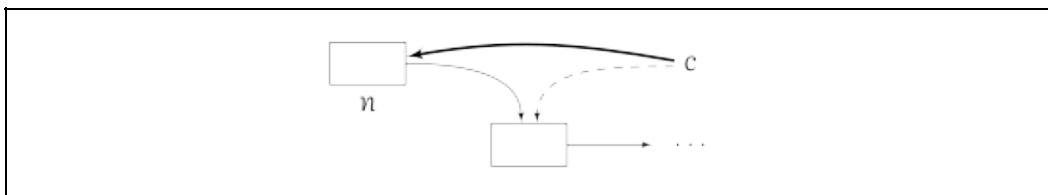


Figura 8.4: Los cambios al meter un elemento cuando la pila no es vacía.

El algoritmo tiene complejidad en tiempo y en espacio constante.

- `toString()`

Las cadenas para representar una pila las haremos muy simples: sencillamente será cada elemento de la pila en una línea distinta, con el fondo de la pila hasta abajo y el tope hasta arriba.

El algoritmo entonces consistirá de recorrer los nodos de la pila, concatenando en una cadena el elemento de cada nodo, seguido de un salto de línea. Esto es lineal en tiempo y en espacio.

Con esto terminamos con la clase `Pila`.

8.3. Algoritmos para colas

Los algoritmos para la clase `cola` serían:

- `mete()`

Las colas meten en el extremo opuesto de donde sacan, entonces en colas hay que meter por el rabo.

Si el elemento a meter es \emptyset entonces ocurre un error. Si el elemento no es vacío creamos un nodo n con él y tenemos dos casos:

1. El rabo es \emptyset . En este caso actualizamos la cabeza y el rabo a n y terminamos.
2. El rabo no es \emptyset . Hacemos que el siguiente del rabo sea n y

actualizamos el rabo a n (figura 8.5).

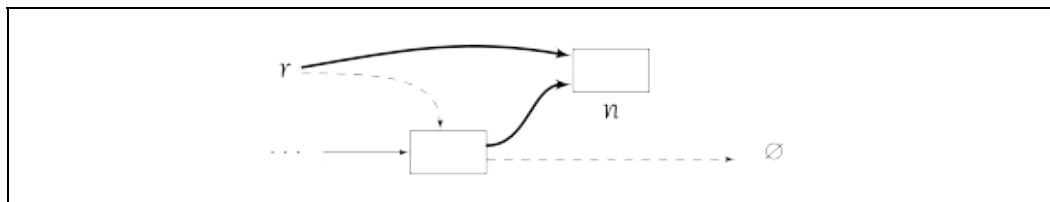


Figura 8.5: Los cambios al meter un elemento cuando la cola no es vacía.

Al igual que con pilas, el algoritmo tiene complejidad en tiempo y en espacio constante.

- `toString()`

Las cadenas para representar una cola las haremos también muy simples: sencillamente serán los elementos en una línea separados por espacios, comenzando con el elemento al final de la cola y terminando con el elemento al frente.

El algoritmo entonces consistirá de recorrer los nodos de la cola, concatenando en una cadena el elemento de cada nodo, seguido de un espacio. Lo único es que como la cabeza es el *frente* de la cola, tenemos que concatenar por la izquierda. Esto es también constante en tiempo y en espacio.

Con esto terminamos con la clase `cola`.

8.4. Pilas y colas en el resto del libro

Las pilas y colas son estructuras tan sencillas que, al igual que los arreglos, al parecer no existe un individuo a quien se le atribuya la autoría de las mismas. La pila de ejecución sin embargo es muy importante (se puede incluso conjeturar que es la aplicación más importante de pilas que existe en computación). Turing en 1945 propuso [6] lo que es básicamente la idea de una pila de ejecución en un reporte para el Comité Ejecutivo del Laboratorio Nacional de Física de Gran Bretaña; en lugar de *pop* y *push*, Turing utilizó *bury* y *unbury* como el nombre de las operaciones fundamentales. Este reporte de Turing no es muy conocido y desarrolla de manera independiente muchos de los mismos conceptos que John von Neumann propondría en la arquitectura que lleva su nombre. En 1957 Fritz Bauer y Klaus Samelson llenaron una patente en Alemania [3] y en 1988 Bauer recibió el Computer Pioneer Award por el invento del principio de la pila de ejecución (aunque Turing propusiera básicamente lo mismo más de diez años antes).

Las pilas y las colas son estructuras muy sencillas de implementar, incluso sin

tener una implementación de listas; con una implementación de listas es trivial. Nuestra implementación será utilizada por varias estructuras más adelante en el libro, particularmente árboles y gráficas. Además, varios de los algoritmos recursivos que veremos la única manera que tendremos de convertirlos a iterativos será usando una pila explícitamente, en lugar de usar la de ejecución implícitamente.

Ejercicios

1. Implementa los métodos faltantes de las clases `MeteSaca`, `Pila` y `Cola`.
2. Mete los elementos 1, 2, 3, 4 a una pila; saca dos veces el elemento al tope de la pila y mete los elementos 5, 6 y 7. Si sacas todos los elementos de la pila, ¿en qué orden salen?
3. Mete los elementos 1, 2, 3, 4 a una cola; saca dos veces el elemento al frente de la cola y mete los elementos 5, 6 y 7. Si sacas todos los elementos de la cola, ¿en qué orden salen?

-
1. A menos que usemos varios hilos de ejecución, en cuyo caso el método `run()` del hilo correspondiente estará al fondo de la pila correspondiente.[←](#)

9. Lambdas

A nuestra clase `Lista` le falta un comportamiento que probablemente se les haya ocurrido cuando la estaban escribiendo: la capacidad de pedirle que se ordene.

Nos gustaría tener un método en la clase `Lista` con la firma que se ve en el listado 9.1.

```
public void ordena() {  
    // Ordenamos la lista...  
}
```

Listado 9.1: Firma para un método destructivo `ordena()` de la clase `Lista`.

O, si por alguna razón no queremos perder el orden original de la lista, podríamos hacer el método no destructivo; que el método `ordena()` nos regrese una copia de la lista, pero con sus elementos ordenados (listado 9.2).

```
public Lista<T> ordena() {  
    // Regresamos una copia ordenada de la lista...  
}
```

Listado 9.2: Firma para un método `ordena()` no destructivo de la clase `Lista`.

Tenemos un pequeño problema con este método `ordena()`; contrario a `copia()` y `reversa()`, no podemos escribir el método para cualquier tipo de lista: únicamente se pueden ordenar listas cuyos elementos sean comparables entre sí.

Si los elementos de un conjunto no son comparables entre sí, no podemos ordenar una colección de los mismos. Por ejemplo, si tuviéramos una lista de colores, ¿qué orden sería el correcto para ordenarlos? Una idea es usar el orden lexicográfico de sus nombres; pero entonces no estamos ordenando colores, estamos ordenando cadenas.

Otro ejemplo: dado un conjunto de elementos S , digamos $S = \{1, 2, 3\}$, podemos definir un *orden parcial* dado su conjunto potencia $\wp(S)$. Si usamos la relación \subset como relación de comparación, esta relación define un orden parcial entre todos los elementos de $\wp(S)$; el elemento $\{1\}$ es “menor que” el elemento $\{1, 2\}$ porque $\{1\} \subset \{1, 2\}$, que a su vez es “menor que” el elemento $\{1, 2, 3\}$ porque $\{1, 2\} \subset \{1, 2, 3\}$ (figura 9.1).

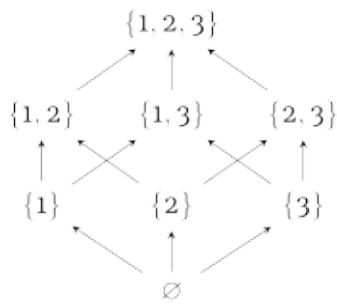


Figura 9.1: Orden parcial definido para $\wp(\{1, 2, 3\})$.

Sin embargo, el orden de los elementos $\wp(S)$ no es un *orden total*, porque no podemos **comparar** a todos los elementos entre sí utilizando la relación \subset , dado que $\{1\} \not\subset \{2\}$ y $\{2\} \not\subset \{1\}$.

Para poder ordenar bien una colección de elementos, el conjunto al que pertenecen debe tener un orden total, lo que implica que todos deben ser comparables entre todos. En Java sabremos que las instancias de una clase son comparables si la misma implementa la interfaz **Comparable**, que se muestra en el listado 9.3 de forma simplificada.

```
public interface Comparable<T> {
    public int compareTo(T e);
}
```

Listado 9.3: Interfaz **Comparable** simplificada.

La interfaz es por supuesto genérica, porque garantiza que las instancias de una clase son comparables entre ellas, no necesariamente con instancias de otras clases. La clase **Integer** por ejemplo está declarada como en el listado 9.4.

```
public class Integer extends Number
    implements Comparable<Integer>, ... {
    ...
    public int compareTo(Integer n) {
        return this.intValue() - n.intValue();
    }
}
```

Listado 9.4: Cómo implementa la clase **Integer** la interfaz **Comparable**.

La semántica del método `compareTo()` es la siguiente:

- `a.compareTo(b)` regresa un entero menor que cero si y sólo si $a < b$.

- `a.compareTo(b)` regresa un entero igual a cero si y sólo si $a = b$.
- `a.compareTo(b)` regresa un entero mayor que cero si y sólo si $a > b$.

Además y aunque no es obligatorio, se recomienda que si `a.compareTo(b)` es igual a cero, entonces `a.equals(b)` regrese `true`. De ser así, se dice que el orden de la clase dado por `compareTo()` es *consistente* con `equals()`.

El problema con nuestra clase `Lista`, es que la queremos utilizar para listas de cualquier tipo, no nada más de elementos comparables. No podemos escribir un método de la clase que funcione únicamente para ciertas instancias de la misma; sería una violación del comportamiento con el que se compromete a cumplir la clase.

Pero podemos esquivar el problema si utilizamos un método estático genérico y acotado, como se ve en el listado 9.5.

```
public static <T extends Comparable<T>>
Lista<T> ordena(Lista<T> lista) { /* ... */ }
```

Listado 9.5: Método `ordena()` estático genérico acotado a `Comparable<T>`.

Como vimos en el capítulo 2, el parámetro `T` de un método genérico está determinado por alguno de sus parámetros. Entonces el método deberá recibir la lista a ser ordenada, porque así podemos garantizar que sólo acepte listas de elementos que sean comparables.

El método podría ser no estático, pero no tendría mucho sentido; de cualquier manera necesitamos el parámetro con el genérico acotado y no podríamos utilizar `this` en ningún momento. Y se vería extraño hacer algo como el listado 9.6.

```
Lista<Integer> ordenada = lista.ordena(lista);
```

Listado 9.6: Invocación de `ordena()` si no fuera estático.

Entonces mejor hacemos al método estático y utilizaremos la clase para invocarlo (listado 9.7).

```
Lista<Integer> ordenada = Lista.ordena(lista);
```

Listado 9.7: Invocación de `ordena()` estático.

Esto funciona y será una de las cosas que haremos en el capítulo 10. Pero podemos resolver el problema de manera más generalizada.

Regresando al ejemplo de manejar estudiantes que vimos en el capítulo 1,

vamos a suponer que tenemos una clase `Estudiante` y una lista de instancias de la misma. Tiene sentido ordenar a los estudiantes por nombre... pero también por número de cuenta; o por edad; o por promedio. No tiene mucho sentido entonces que la clase `Estudiante` implemente `Comparable`, porque los estudiantes tienen múltiples formas de compararse.

Lo que nos gustaría poder hacer es tomar a nuestra lista de estudiantes (o de cualquier otro tipo de objetos, aunque no sean comparables) y mandar llamar un método que reciba como parámetro el código necesario para comparar dos elementos de la lista. Lamentablemente Java no permite esto, porque en Java no podemos pasar métodos como argumentos de otros métodos y (exceptuando constructores, la inicialización de variables de clase y el bloque `static`) los métodos son el único lugar donde podemos escribir código en Java.

Conceptualmente nos gustaría poder hacer lo que se muestra en el listado 9.8.

```
public class Ejemplo {
    public static int comparaPromedios(Estudiante e1,
                                         Estudiante e2) {
        if (e1.getPromedio() < e2.getPromedio())
            return -1;
        if (e1.getPromedio() > e2.getPromedio())
            return 1;
        return 0;
    }
    public static int main(String[] args) {
        Lista<Estudiante> lista = /* Creamos y llenamos
                                         la lista... */
        /* ¡La siguiente línea no es válida! */
        Lista<Estudiante> ordenada =
            lista.ordena(comparaPromedios);
    }
}
```

Listado 9.8: Método inválido: no podemos pasar un método como argumento de otro método.

Volvemos a hacer énfasis de que en Java esto no se puede; el código del listado 9.8 no compila y de hecho no existe una sintaxis para la firma del método `ordena()` de ese estilo (¿pueden imaginar cómo sería?): en Java no podemos pasar un método como argumento de otro método.

Cuando un lenguaje de programación permite pasar métodos como argumentos de otros métodos o más generalmente funciones como argumentos de otras funciones (Java no tiene funciones realmente), se dice que el lenguaje tiene funciones de *primera clase*.

9.1. Lambdas y funciones de primera clase

Vamos a tomar una pequeña desviación para ver cómo funciona un lenguaje que sí tenga funciones de primera clase, en este caso Python, continuando con el mismo ejemplo de una lista de estudiantes.

Como Python sí tiene funciones de primera clase, el ejemplo equivalente de arriba sí funciona aquí (listado 9.9).

```
def compara_promedios(e1, e2):
    if e1.get_promedio() < e2.get_promedio():
        return -1
    if e1.get_promedio() > e2.get_promedio():
        return 1
    return 0

lista = # ... creamos y llenamos una lista de estudiantes
lista.sort(compara_promedios)
```

Listado 9.9: En Python sí podemos pasar una función como argumento de otra función o método. Este ejemplo sólo funciona en Python 2 por cambios en las listas en la versión 3 del lenguaje.

El ejemplo de arriba funciona en Python 2, pero no en Python 3 porque el método `sort()` de las listas dejó de aceptar como argumento una función de comparación, al ser substituido por una mejor manera de hacer ordenamiento de listas; no cubriremos eso aquí.

Python sí tiene funciones, en el ejemplo anterior `compara_promedios()` es una función. Java no tiene funciones; tiene métodos, que por definición siempre pertenecen a una clase.

Pero además de tener funciones, en Python son de primera clase; esto quiere decir que pueden ser tratadas como cualquier otro valor en el lenguaje: pueden ser asignadas a variables, pasadas como argumentos de una función y regresadas como el resultado de ejecutar una función.

En ese sentido y como las funciones son de primera clase, las mismas tienen la capacidad de ser literales o anónimas. Para explicar por qué esto es conveniente supongamos que ahora queremos ordenar a nuestra lista de estudiantes por número de cuenta en lugar de por promedio. Como la función de comparación tiene la misma semántica a la del método `compareTo()`, ésta debe regresar un número menor que cero si el primer argumento es menor que el segundo; cero si los dos argumentos son iguales; y un número mayor que cero si el primer argumento es mayor que el segundo. Y dado que el número de cuenta es un entero, la función de comparación entre números de cuenta puede implementarse sencillamente restándolos (listado 9.10).

```

def compara_cuentas(e1, e2):
    return e1.get_cuenta() - e2.get_cuenta()

lista = # ... creamos y llenamos una lista de estudiantes
lista.sort(compara_cuentas)

```

Listado 9.10: Otro ejemplo de pasar una función como argumento de otra función.

El ejemplo del listado 9.10 muestra que es ligeramente redundante el tener que definir la función `compara_cuentas()`, cuando ésta sólo tiene una línea con un único enunciado.

Nos gustaría poder pasar el *código* de la función directamente, sin necesidad de tener que declarar la función por completo. Imaginen, haciendo una analogía, que para poder pasar un argumento a un método que recibe un entero no pudiéramos pasar el valor 5, sino que tuviéramos que hacer `int n = 5` y entonces pasar `n` como argumento. Sería muy incómodo escribir programas así; si extrapolamos esto a funciones de primera clase, debemos tener un mecanismo para no tener que declarar una función, sino pasar el código de la misma directamente.

Y esto es básicamente lo que son las lambdas; literales de funciones o funciones anónimas. Regresando a ordenar listas de estudiantes por número de cuenta, podríamos usar lambdas en Python como en el listado 9.11. Como el cuerpo de la lambda es únicamente un enunciado, ni siquiera necesitamos utilizar `return`.

```

lista = # ... creamos y llenamos una lista de estudiantes
lista.sort(lambda a, b: a.get_cuenta() - b.get_cuenta())

```

Listado 9.11: No definimos una función de comparación para números de cuenta; sencillamente pasamos el *código* al método `sort()` de las listas de Python.

Las lambdas como conceptualización de funciones anónimas vienen del modelo matemático de cómputo conocido como cálculo-λ, desarrollado por Alonzo Church en 1936 [7] de forma paralela a las máquinas de Turing [30]. Ambos modelos de cómputo son equivalentes: todo lo que se puede computar con una máquina de Turing se puede computar con cálculo-λ y viceversa.

En el cálculo-λ no hay estructuras de control, sólo hay funciones. Por poner un ejemplo, no hay una estructura de control `if`; hay una *función if()*, que tiene como parámetros un valor booleano, la función que ejecuta si el valor booleano es verdadero y la función que ejecuta si el valor booleano es falso. Tampoco hay nada similar a `while` o `for`; si un algoritmo requiere repetir instrucciones, hay que utilizar recursión.

Como funciones es lo único que hay en cálculo- λ , estar teniendo que nombrar cada función independientemente se vuelve muy tedioso y Church introdujo el concepto de lambda para no tener que definir una función en el sentido de darle un nombre: sólo se escribe λ y su cuerpo (simplificando *mucho* la idea). El cómputo que se realiza se puede inferir sin necesidad de saber el nombre de esa función; exactamente de la manera en que el código del listado 9.11 no necesita en lo más mínimo el nombre de la función que compara estudiantes por número de cuenta, únicamente necesita el código para hacer la comparación.

Los lenguajes de programación funcionales (que están muy relacionados con el cálculo- λ) siempre ofrecen funciones como objetos de primera clase (de ahí su nombre: *funcionales*) y generalmente ofrecen lambdas.

Los lenguajes imperativos, que abarcan casi todos los lenguajes estructurados y orientados a objetos, hasta hace algunos años lo más normal era que no ofrecieran lambdas y generalmente no ofrecían funciones como objetos de primera clase.

El lenguaje de programación C no tiene lambdas ni funciones como objetos de primera clase; pero permite una manera de hacer algo similar. Si en C tenemos una estructura `Lista` equivalente a nuestra clase, podríamos hacer una función que ordenara como se ve en el listado 9.12.

```
Lista*
lista_ordena(Lista* lista, int (*cmp) (void* a, void* b)) {
    // ...
    /* Usamos cmp. */
    if (cmp(x, y) < 0) { /* x < y */ }
    // ...
}
```

Listado 9.12: Función que recibe como parámetro un apuntador a funciones.

En el listado 9.12 la variable `cmp` es un apuntador a una función que regresa un entero y recibe dos apuntadores anónimos (sin tipo establecido), que es lo que significa `void*`. El tipo de un apuntador a función está dado por el tipo de regreso y los tipos de sus parámetros, por eso hay que especificarlo todo. Para hacerlo más legible, se suele utilizar un `typedef` como en el listado 9.13.

```
typedef int (*FuncionCompara) (const void* a,
                                const void* b);
Lista*
lista_ordena(Lista* lista, FuncionCompara cmp) {
    // ...
    /* Usamos cmp. */
```

```

    if (cmp(x, y) < 0) { /* x < y */ }
    // ...
}

```

Listado 9.13: Definiendo el tipo para funciones de comparación.

Y entonces podríamos utilizar una función de comparación para ordenar nuestras listas, como en el listado [9.14](#).

```

int
estudiante_compara_cuentas(const void* a, const void* b) {
    Estudiante* e1 = (Estudiante*)a;
    Estudiante* e2 = (Estudiante*)b;
    return estudiante_get_cuenta(e1) -
        estudiante_get_cuenta(e2);
}
int
main(int argc, char* argv[]) {
    Lista* lista = /* Creamos y llenamos nuestra
                    lista de estudiantes. */
    Lista* ordenada =
        lista_ordena(lista, estudiante_compara_cuentas);
    return 0;
}

```

Listado 9.14: Usamos nuestra función de ordenamiento con una función de comparación.

En otras palabras podemos definir funciones como parámetros y pasárselas como argumentos (en C el identificador de una función es en sí mismo un apuntador a la función), pero no hay funciones anónimas.

Hasta antes de su versión 8, el lenguaje de programación Java ni siquiera podía pasar métodos como argumentos, mucho menos métodos anónimos. Lo único que tenía (y de hecho aún tiene) como parámetros son tipos básicos y referencias a objetos. Pero con esto se puede hacer algo similar a lambdas, si bien el resultado es mucho más feo. Que es lo que veremos en la siguiente sección.

9.2. Clases internas anónimas

Como únicamente podemos pasar objetos como argumentos a nuestros métodos, vamos a hacer eso exactamente para emular funciones de primera clase. En lugar de pasar una función que compare dos elementos de la lista, vamos a pasar un objeto instancia de una clase que tenga un método que permita comparar dos elementos de la lista.

En otras palabras queremos un objeto nada más como mecanismo para pasar

realmente el método comparador. Y para usar este método comparador dentro de nuestro método de ordenamiento, lo único que necesitamos es su firma; así que la mejor manera de definir el método comparador es utilizar una interfaz.

Java ya lo hace por nosotros; la interfaz se llama `Comparator` y se muestra (muy simplificada) en el listado 9.15.

```
public interface Comparator<T> {
    public int compare(T a, T b);
}
```

Listado 9.15: Interfaz `Comparator` simplificada.

La semántica del método `compare()` de `Comparator` es idéntica a la del método `compareTo()` de `Comparable`:

- `compare(a, b)` regresa un entero menor que cero si y sólo si $a < b$.
- `compare(a, b)` regresa un entero igual a cero si y sólo si $a = b$.
- `compare(a, b)` regresa un entero mayor que cero si y sólo si $a > b$.

Usando la interfaz `Comparator` podemos definir un método `ordena()` en la clase `Lista`, que no sea estático ni genérico (listado 9.16).

```
public class Lista<T> implements Coleccion<T> {
    // ...
    public Lista<T> ordena(Comparator<T> compara) {
        /* ... */
    }
}
```

Listado 9.16: Método `ordena()` no genérico, que utiliza un `Comparator`.

Ahora podemos escribir una clase que implemente `Comparator` para comparar estudiantes por número de cuenta, como en el listado 9.17.

```
public class ComparaEstudiantesPorNumeroDeCuenta
    implements Comparator<Estudiante> {
    @Override public int compare(Estudiante e1,
                                Estudiante e2) {
        return e1.getNumeroDeCuenta() - e2.getNumeroDeCuenta();
    }
}
```

Listado 9.17: Clase comparadora que compara estudiantes por número de cuenta.

Usando la clase `ComparaEstudiantesPorNumeroDeCuenta` nos permite ordenar listas de

estudiantes por número de cuenta (listado 9.18).

```
Lista<Estudiante> lista = new Lista<Estudiante>();
// Llenamos la lista.
Comparator<Estudiante> comparador;
comparador = new ComparaEstudiantesPorNumeroDeCuenta();
Lista<Estudiante> ordenada = lista.ordena(comparador);
```

Listado 9.18: Usando la clase `ComparaEstudiantesPorNumeroDeCuenta` para ordenar estudiantes.

Con esto ya podemos ordenar listas de estudiantes utilizando múltiples métodos (en el sentido de maneras o formas y de métodos de clase) para compararlos. El problema es que tenemos que escribir una clase completa para ese método `compare()`, porque los métodos están asociados a clases.

Java no ofrece funciones o métodos como argumentos, pero sí ofrece (desde las primeras versiones del lenguaje) clases internas anónimas, lo que nos permite al menos ahorrarnos la declarar la clase `ComparaEstudiantesPorNumeroDeCuenta` (listado 9.19).

```
Lista<Estudiante> lista = new Lista<Estudiante>();
// Llenamos la lista.
Lista<Estudiante> ordenada;
ordenada = lista.ordena(new Comparator<Estudiante>() {
    @Override public int compara(Estudiante e1,
                                  Estudiante e2) {
        return e1.getNumeroDeCuenta() - e2.getNumeroDeCuenta();
    }
});
```

Listado 9.19: Con una clase interna y anónima podemos ahorrarnos la declaración de la clase `ComparaEstudiantesPorNumeroDeCuenta`.

La sintaxis de una clase interna anónima utiliza el operador `new` para instanciar una clase sin nombre (de ahí que sea anónima). Esto permite utilizar interfaces (como `Comparator` en el listado 9.19) o clases abstractas; el bloque entre las llaves se convierte en el cuerpo de la clase interna anónima y la misma implementa o extiende al tipo que sigue a `new` (de nuevo, `Comparator` en el listado 9.19).

Nuestra clase `Lista` se compila al *bytecode* en el archivo `Lista.class`; su clase interna `Nodo` se compila al *bytecode* en el archivo `Lista$Nodo.class`. Si la clase `Lista` tuviera varias clases internas anónimas, éstas se compilarían al *bytecode* en los archivos `Lista$1.class`, `Lista$2.class`, etcétera. Si todas las clases internas anónimas implementan interfaces distintas, el compilador con casi toda certeza utilizará una única clase interna anónima que implementará todas esas interfaces y por lo tanto sólo existirá el archivo `Lista$1.class`. Esto sólo

es para explicar por qué el número de archivos relacionados con clases internas anónimas no necesariamente corresponderá al número de clases internas anónimas en una clase.

El uso de clases internas anónimas no sólo nos permite ahorrarnos la declaración explícita de una clase de la cual únicamente nos interesa un método; como es interna, la clase anónima puede utilizar las variables de clase de la clase envolvente e incluso ciertas variables locales. Y en práctica funciona como si enviáramos código a un método; este código (o realmente el objeto que lo invoca) es utilizable únicamente en el método que lo recibe, porque como la clase interna es anónima, no podemos instanciar más objetos de la misma.

El único problema es que se ve *espantoso*. Funciona, pero se ve espantoso. Y básicamente eso es lo que resuelve la implementación de lambdas en Java.

9.3. Lambdas en Java con interfaces funcionales

Lo que vimos en la sección anterior básicamente ya permite el funcionamiento de lambdas en Java; de hecho la manera como declaramos el método `ordena()` en el listado 9.16 es como lo vamos a utilizar con lambdas. Para usar una lambda en Java, lo hacemos como en el listado 9.20.

```
Lista<Estudiante> lista = new Lista<Estudiante>();
// Llenamos la lista.
Lista<Estudiante> ordenada;
ordenada = lista.ordena((e1, e2) -> e1.getCuenta() -
                           e2.getCuenta());
```

Listado 9.20: Usamos el método `ordena()` con una lambda.

Para que esto funcione, para poder utilizar una lambda (la sintaxis `() -> {}`), el método que recibe la lambda debe recibir una interfaz que defina un único método. Porque lo que ocurre es que el compilador de Java convierte el código en el listado 9.20 en el código del listado 9.19.

Una vez más, como con genéricos y la estructura de control *for-each*, las lambdas son una mecánica del compilador de Java. El compilador toma el código a la derecha de `->` (puede ser un bloque enmarcado por llaves o un único enunciado como en el listado 9.20) y lo convierte en el cuerpo del único método de la interfaz que la clase interna anónima implementa. El compilador sabe qué interfaz utilizar porque la firma del método se lo dice: por ejemplo en el listado 9.20 la interfaz utilizada es `comparator` porque nuestro método `ordena()` es lo que recibe. De la misma manera, los parámetros `e1` y `e2` el

compilador sabe que son de tipo `Estudiante` porque la lista que manda llamar al método `es de estudiantes` y el método `ordena()` recibe como parámetro un `Comparador<T>`, donde la `T` es la misma de la clase `Lista`.

Como las lambdas de Java están relacionadas a interfaces, estas interfaces son especiales: se les llamará *interfaces funcionales*. En Java una interfaz es funcional si todos los métodos de la interfaz, excepto uno, tienen una implementación por omisión (`default`). En particular, toda interfaz que defina un único método sin implementación por omisión es funcional.

Si la interfaz cumple con esto, puede tener explícitamente la anotación `@FunctionalInterface` y el compilador verificará que exactamente uno de sus métodos no tenga implementación por omisión. Por lo mismo, se recomienda que cualquier interfaz funcional utilice la anotación, aunque no es obligatorio para seguir soportando código escrito antes de Java 8.

Los métodos por omisión en interfaces son una característica nueva de Java a partir de la versión 8 del lenguaje. La idea es que las interfaces funcionales sólo les “falte” implementar un método, que es el que puede utilizar como cuerpo de una lambda.

Aunque en general funcionan como uno esperaría, el que las lambdas de Java sean realmente clases internas y anónimas puede generar problemas. Por ejemplo, supongamos que tenemos dos interfaces funcionales `X` y `Y` definidas como en los listados [9.21](#) y [9.22](#) respectivamente.

```
@FunctionalInterface
public interface X {
    public void f();
}
```

Listado 9.21: Interfaz `X`.

```
@FunctionalInterface
public interface Y {
    public void g();
}
```

Listado 9.22: Interfaz `Y`.

Ahora definamos una clase `Ejemplo` como en el listado [9.23](#).

```
public class Ejemplo {
    public void ejemplo(X x) { x.f(); }
    public void ejemplo(Y y) { y.g(); }
    public static void main(String[] args) {
```

```

        Ejemplo e = new Ejemplo();
        e.ejemplo(() -> System.out.println("No funciona"));
    }
}

```

Listado 9.23: El polimorfismo de Java evita que podamos usar lambdas porque hay dos métodos `ejemplo()` que reciben argumentos no diferenciables a partir de sus parámetros.

La clase `Ejemplo` del listado 9.23 no compila; el compilador no sabe qué interfaz implementar porque ambas interfaces definen métodos que, aunque se llamen distintos, tienen la misma firma. Como hay métodos `ejemplo()` que definen como parámetros tanto instancias de `x` como de `y`, el compilador no sabe cuál utilizar para la clase interna anónima.

Podemos resolver esto al definir explícitamente el tipo de la clase interna anónima, como en el listado 9.24.

Las lambdas en Java son objetos que son instancia de una interfaz funcional, implementada por una clase interna anónima. Esto significa que estos objetos para motivos prácticos no tienen estado interno (excepto el proporcionado por `Object`) y entonces realmente sólo nos dan el método definido por la interfaz funcional. No son una función o método (como en casi todos los otros lenguajes de programación que ofrecen lambdas), sino un objeto con el cual podemos mandar llamar el código de un método.

```

public class Ejemplo {
    public void ejemplo(X x) { x.f(); }
    public void ejemplo(Y y) { y.g(); }
    public static void main(String[] args) {
        Ejemplo e = new Ejemplo();
        X x = () -> System.out.println("Sí funciona");
        e.ejemplo(x);
    }
}

```

Listado 9.24: Eliminamos la ambigüedad al definir el tipo de la lambda explícitamente.

Si nos ponemos estrictos entonces, las lambdas de Java no son realmente lambdas; nunca enviamos código a un método, seguimos enviando referencias a objetos, como siempre. Lo único que cambia es que podemos escribir de manera más sucinta y agradable cómo enviamos esta referencia a un objeto.

Sin embargo, en la práctica funcionan básicamente igual de bien que las lambdas “de verdad” de otros lenguajes de programación. Sólo sí debe quedar claro que estamos enviando la referencia a un objeto, que es la única instancia de una clase interna anónima.

En la práctica la diferencia no será muy significativa generalmente.

9.4. Alcance de variables

Las lambdas en Java (y casi cualquier otro lenguaje de programación) son particularmente útiles en interfaces gráficas de usuario; cuando se tiene un botón en una interfaz gráfica, se espera que cuando el usuario haga click en él cierto código se ejecute: una lambda es la mejor manera de decirle al botón “cuando te hagan click, ejecuta este código”.

Por ejemplo, en Java 8 con JavaFX se puede escribir el código del listado 9.25 para escribir un mensaje cada vez que hagan click en un botón.

```
public void agregaAccion(Button boton) {
    boton.setOnAction(e -> System.out.println("Click"));
}
```

Listado 9.25: Usamos una lambda para imprimir un mensaje al hacer click en un botón.

Como la lambda tiene un único parámetro, podemos ahorrarnos el paréntesis del mismo. Mostramos el ejemplo de un botón en una interfaz gráfica porque hace fácil explicar las limitaciones del alcance de variables locales en lambdas. Cuando agregamos código a un botón para que se ejecute cuando le hagan click, *este código se ejecuta después de ser agregado*. Y no inmediatamente después, sino potencialmente *mucho* después: si el botón está presente desde que la aplicación inicia, el código que ejecuta cuando le hagan click no correrá hasta que el usuario haga click, obviamente. Esto puede ser minutos (u horas) después.

Esto afecta el alcance de las variables locales porque el método donde la lambda es agregada es casi seguro que haya sido descargado de la pila de ejecución cuando la lambda se ejecute. Y todas las variables locales del método dejan de existir cuando el método se descarga.

Veamos un ejemplo: si quisiéramos imprimir un mensaje con el número de veces que se le ha hecho click al botón, lo primero que se le puede ocurrir a un programador es algo como el código en el listado 9.26.

```
public void agregaAccion(Button boton) {
    int cont = 0;
    boton.setOnAction(e -> {
        cont++;
        System.out.println("Clicks: " + cont);
    })
}
```

Listado 9.26: No podemos incrementar cont porque deja de existir cuando el método termina.

El código del listado 9.26 no compila: el compilador no puede utilizar la variable `cont` porque deja de existir cuando el método `agregaAccion()` termina y esto ocurre justamente después de que la lambda es agregada al botón. Cuando el usuario haga click la variable `cont` ya no existe y no podemos incrementarla.

Esto no significa que no podamos usar variables locales en lambdas; el alcance de las primeras sí se extiende a las segundas, pero únicamente si se utilizan únicamente para leerse, no para escribirse. Por ejemplo el código en el listado 9.27 es válido.

```
public void agregaAccion(Button boton) {
    String m = "Botón activado";
    boton.setOnAction(e -> System.out.println(m));
}
```

Listado 9.27: Sí podemos usar la variable `m` en la lambda, porque sólo la leemos.

Estamos hablando de la *variable*, no del objeto al que hace referencia (si es una referencia). Por ejemplo, el código del listado 9.28 es válido, porque la variable local `e` no es modificada, aunque el objeto al que refiere sí.

```
public void agregaAccion(Button boton) {
    Estudiante e = new Estudiante("Juan Pérez",
                                  312322410,
                                  9.3, 18);
    boton.setOnAction(v -> e.setPromedio(9.4));
}
```

Listado 9.28: La variable local `e` se puede usar en la lambda porque no se modifica, aunque el objeto al que refiere sí. Noten que cambiamos el nombre del parámetro de la lambda para que no tapara a la variable local.

Si la variable no es modificada, entonces sólo necesitamos el *valor* de la variable y el compilador puede guardar eso en una variable local **final** (`final`) dentro de la lambda.

El término técnico para las variables locales que se pueden utilizar dentro de una lambda es *variables realmente finales* (*effectively final variables* en inglés), porque es lo que son, variables que se comportan como si fueran finales. En las versiones de Java 7 y anteriores, las únicas variables locales que se pueden usar dentro de una clase interna anónima (que es lo que son las lambdas), son finales. En Java 8 esta restricción ha sido reducida y ahora sólo se tienen que *comportar* como variables finales, no tienen que ser declaradas como tales.

Resumiendo, las variables locales extienden su alcance a dentro de las

lambdas, pero únicamente si son realmente finales. Entonces, ¿cómo podemos hacer lo que queríamos al inicio de la sección, imprimir el número de clicks en el botón? Simple: usamos una variable de clase (listado 9.29).

```
private int cont;
public void agregaAccion(Button boton) {
    cont = 0;
    boton.setOnAction(e -> {
        cont++;
        System.out.println("Clicks: " + cont);
    });
}
```

Listado 9.29: Código válido: la variable cont es de clase y su alcance se extiende automáticamente a la lambda.

Las variables de clase son visibles dentro de toda la clase, entonces no hay problema de alcance con ellas en las lambdas. Que es otra ventaja de utilizar lambdas: nos dan acceso a todo el estado interno del objeto en la clase de la que la lambda es interna.

Las lambdas son otra característica del compilador de Java; realmente son instancias de clases internas anónimas, sólo se ven mucho mejor. De cualquier manera funcionan razonablemente bien y definitivamente nos permitirán ordenar nuestras listas, aunque sean de elementos cuya clase no implemente **Comparable**.

Ejercicios

1. Conceptualmente, ¿qué son las lambdas?
2. ¿Qué quiere decir que una interfaz en Java sea funcional?
3. ¿Cómo se implementan las lambdas en Java 8 y posteriores?
4. ¿Cómo debe ser una variable local para que pueda ser utilizada en el cuerpo de una lambda?
5. Cuando pasamos una lambda como argumento a un método, ¿qué se está pasando realmente?

10. Ordenamientos

Existen computólogos que sostienen que lo más importante de la computación (y, por extensión, de las computadoras) es ordenar datos y buscar rápidamente en ellos [22]. Ciertamente será un tema importante de este libro; la mayor parte de (si no es que todas) las estructuras y algoritmos cubiertos por el mismo estarán relacionados con ordenar y buscar datos.

Comencemos con ordenar las estructuras que ya tenemos; vamos (por fin) a ordenar listas y arreglos. Pero primero vamos a repasar las firmas de la clase `Lista` y dónde vivirá el código para poder ordenar arreglos.

Por lo visto en los capítulos 2 y 9, ya sabemos cómo son las firmas de los métodos que escribiremos para ordenar nuestras listas; serán como se ve en el listado 10.1.

```
public Lista<T> mergeSort(Comparator<T> comparador) {  
    /* ... */  
}  
public static <T extends Comparable<T>>  
Lista<T> mergeSort(Lista<T> lista) { /* ... */ }
```

Listado 10.1: Firmas de los métodos para ordenar listas.

Primero notemos que cambiamos el nombre de los métodos de `ordena()` a `mergeSort()`. Siguiendo los preceptos de la Orientación a Objetos, el método *debería* llamarse `ordena()`; pero veremos varios algoritmos para hacer ordenamientos y además hay ciertos algoritmos tan importantes en computación que nos parece importante recalcar su importancia al usar el nombre de los mismos en los métodos que los implementen. El algoritmo que usaremos para ordenar listas es MERGESORT y así se llamarán los métodos correspondientes.

Vamos a tener un método no estático que recibe un comparador para poder pasarle una lambda que haga la comparación de dos elementos de la lista; y un método estático acotado a listas de elementos comparables. Por supuesto, implementaremos el algoritmo una única vez en el método no estático; si lo hacemos así, el método estático se vuelve trivial de implementar. Tan trivial, que lo mostramos en el listado 10.2.

```
public static <T extends Comparable<T>>  
Lista<T> mergeSort(Lista<T> lista) {  
    return lista.mergeSort((a, b) -> a.compareTo(b));
```

```
}
```

Listado 10.2: Método `mergeSort()` no estático.

Como el método estático recibe una lista de elementos comparables, podemos crear un comparador usando el método `compareTo()` de sus elementos, como se ve en el listado 10.2. De esta forma sólo hay que implementar el algoritmo MERGESORT una vez.

Para ordenar arreglos tenemos el problema de que no hay manera de modificar la clase a la que pertenece un arreglo. Entonces necesitamos crear una clase especial para poder poner ahí los métodos que ordenen nuestros arreglos; recuerden, de nuevo, que en Java los métodos siempre pertenecen a una clase.

Siendo absolutamente poco imaginativos, a esta clase le llamaremos `Arreglos`; esto siguiendo la pauta dada por Java que tiene una clase `Arrays` en el paquete `java.util` para implementar algoritmos relacionados con arreglos.

Con arreglos implementaremos dos algoritmos para ordenarlos: SELECTIONSORT y QUICKSORT. Consecuentemente tendremos métodos `selectionSort()` y `quickSort()`; además tendremos también versiones genéricas acotadas a comparables y versiones genéricas sin cota que recibirán un comparador. Todas las versiones serán estáticas, porque no tiene sentido crear una instancia de `Arreglos`; la clase existe únicamente para poner ahí los métodos. Y de hecho, para evitar la instanciación de objetos usando `Arreglos`, le definiremos un único constructor privado sin parámetros.

El esqueleto de la clase `Arreglos` quedaría como en el listado 10.3; una vez más, las versiones acotadas son tan triviales que incluimos el código de ellas.

```
public class Arreglos {  
  
    private Arreglos() {}  
  
    public static <T> void  
    quickSort(T[] arreglo, Comparator<T> comparador) {  
        /* ... */  
    }  
  
    public static <T extends Comparable<T>> void  
    quickSort(T[] arreglo) {  
        quickSort(arreglo, (a, b) -> a.compareTo(b));  
    }  
  
    public static <T> void  
    selectionSort(T[] arreglo, Comparator<T> comparador) {  
        /* ... */  
    }  
}
```

```

    }

    public static <T extends Comparable<T>> void
    selectionSort(T[] arreglo) {
        selectionSort(arreglo, (a, b) -> a.compareTo(b));
    }
}

```

Listado 10.3: Esqueleto de la clase `Arreglos` con la implementación de los métodos acotados.

10.1. Ordenamientos en arreglos

Como mencionábamos, los algoritmos que veremos para ordenar arreglos son dos: `SELECTIONSORT` y `QUICKSORT`.

10.1.1. Algoritmo `SELECTIONSORT`

`SELECTIONSORT` es un algoritmo algo inocente, pero muy sencillo de entender e implementar. La idea es colocarnos al inicio del arreglo (índice cero), *seleccionar* de los n elementos el más pequeño e intercambiar el más pequeño por el que esté en el índice cero.

Después nos colocamos en el siguiente elemento (índice uno) y seleccionamos de los $n - 1$ elementos (a partir del 1-ésimo) el más pequeño; intercambiamos el 1-ésimo elemento por el más pequeño.

En el i -ésimo paso ($i < n$) recorremos los $n - i$ elementos comenzando en el i -ésimo, seleccionamos el más pequeño y lo intercambiamos con el i -ésimo.

Como al inicio del i -ésimo paso los elementos $0, \dots, i - 1$ ya están en orden y son menores o iguales a los elementos $i, \dots, n - 1$; y al final queda en el índice i el elemento más pequeño de los elementos $i, \dots, n - 1$, cuando termine el algoritmo el arreglo estará ordenado.

Lo más sencillo es implementar el algoritmo con dos iteraciones anidadas: la primera recorre el arreglo con un índice $i = 0, \dots, n - 1$, se guarda el valor de i en una variable temporal m (de menor) y la segunda recorre el arreglo con un índice $j = i + 1, \dots, n - 1$. Si el j -ésimo elemento es menor que el m -ésimo elemento, actualizamos m a j . Cuando acabe la iteración anidada (la del índice j), intercambiamos el i -ésimo elemento por el m -ésimo (figura 10.1).

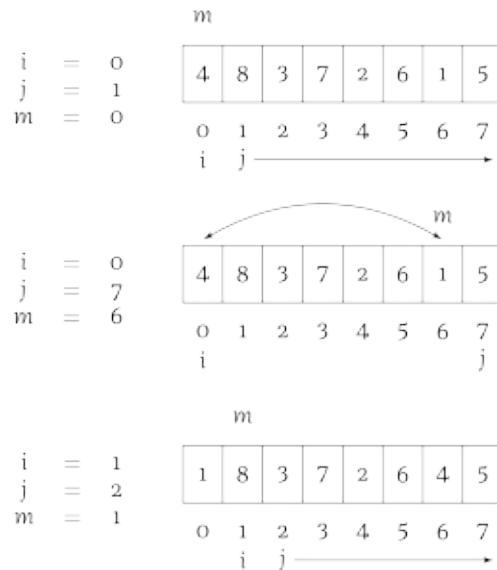


Figura 10.1: En el i -ésimo paso, el algoritmo SELECTIONSORT comienza con el subarreglo $A[0 : i - 1]$ ordenado y con todos sus elementos siendo menores o iguales que los elementos en el subarreglo $A[i : n - 1]$; se busca el mínimo en $A[i : n - 1]$ (usando un índice j) e intercambia el mínimo con el i -ésimo.

En general en este libro omitiremos las demostraciones formales de que los algoritmos vistos son correctos, pero SELECTIONSORT es muy sencillo demostrar que funciona correctamente y entonces lo utilizaremos para ejemplificar cómo se hace. Sólo enfatizamos que esto es realmente material de un curso de Análisis de Algoritmos.

Para demostrar que SELECTIONSORT funciona, vamos a definirlo más formalmente utilizando pseudocódigo (algoritmo 10.1).

```

procedure SELECTIONSORT( $A, n$ )
   $\triangleright$  Arreglo  $A$  de  $n$  elementos
  for ( $i \leftarrow 0$  to  $n - 1$ )
     $m \leftarrow i$ 
    for ( $j \leftarrow i + 1$  to  $n - 1$ )
      if ( $A[j] < A[m]$ )
         $m \leftarrow j$ 
    INTERCAMBIA( $A, i, m$ )
   $\triangleright$  Intercambiamos el  $i$ -ésimo y  $m$ -ésimo elemento en  $A$ 

```

Algoritmo 10.1: Algoritmo SELECTIONSORT.

El algoritmo SELECTIONSORT es iterativo (dblemente iterativo, de hecho); para demostrar que una iteración es correcta se demuestra que una invariante es verdadera. Una invariante, como su nombre nos dice, es un enunciado lógico que no varía durante toda la iteración: la invariante debe ser verdadera antes

de entrar al primer ciclo (o vuelta) de la iteración (*inicialización*); si es verdadera antes de un ciclo, entonces se mantiene verdadera antes del siguiente ciclo (*mantenimiento*); y al terminar (esto es importante) el enunciado debe ayudarnos a demostrar lo que queremos (*terminación*). Las demostraciones usando invariantes son muy similares a las pruebas por inducción, donde la inicialización reemplaza al caso base y el mantenimiento al uso de la hipótesis por inducción; la diferencia es que en inducción la hipótesis se usa *ad infinitum*, mientras que un ciclo siempre tiene que terminar y al hacerlo debe haber hecho lo que queremos que haga. Por eso es importante la terminación y que el enunciado nos ayude a demostrar que el algoritmo es correcto: en las dos iteraciones del algoritmo [10.1](#) el enunciado “*la variable i es de tipo entero*” es verdadero, pero no nos ayuda en lo más mínimo a demostrar que `SELECTIONSORT` es correcto.

Vamos a demostrar un lema auxiliar para la iteración anidada y en un teorema al final haremos la demostración de que `SELECTIONSORT` es correcto. Utilizaremos la notación de “rebanada” (*slice* en inglés) para subarreglos: $A[a : b]$ denotará el subarreglo de A con índice inicial a e índice final b ; si $b < a$ consideraremos al subarreglo vacío. Además, supondremos que el algoritmo `INTERCAMBIA` es correcto; pero es tan trivial que no debería causar ningún problema suponerlo (e implementarlo).

Lema 10.1. (La iteración anidada de `SELECTIONSORT` es correcta). Al terminar la iteración en las líneas 4-6 del algoritmo [10.1](#), el índice m es el de un elemento mínimo del subarreglo $A[i : n - 1]$.

Demostración. Probaremos la siguiente invariante:

Invariante: Para toda $j = i + 1, \dots, n - 1$, el índice m es el de un mínimo del subarreglo $A[i : j - 1]$.

- *Inicialización:* Antes de entrar a la iteración, $m = i$ (línea 3 en el algoritmo [10.1](#)). Por lo tanto antes del primer ciclo de la iteración, cuando $j = i + 1$, se tiene que $A[i : j - 1] = A[i : i] = A[m : m]$, por lo que trivialmente m es el índice de un mínimo (de hecho *el* mínimo) de $A[i : j - 1]$.
- *Mantenimiento:* Si m es el índice de un mínimo del subarreglo $A[i : k - 1]$, antes del ciclo cuando $j = k$, entonces m será un mínimo del subarreglo $A[i : k]$ antes del siguiente ciclo cuando $j = k + 1$.

Tenemos que $j = k$ y que m es un mínimo del subarreglo $A[i : k - 1]$. Por la línea 5 del algoritmo tenemos dos opciones: o bien $A[j] < A[m]$ o bien $A[j] \geq A[m]$.

En el primer caso como m era el índice un mínimo de $A[i : j - 1]$ y $A[j] < A[m]$, se sigue que j es el índice un mínimo de $A[i : j]$. Como se cumple la condición se entra al cuerpo del **if** en la línea 6 y actualizamos m al valor de j . Por lo tanto m es el índice de un mínimo de $A[i : j]$.

En el segundo caso m sigue siendo el índice de un mínimo de $A[i : j]$ porque $A[j] \geq A[m]$. No se entra al cuerpo del **if** y m se mantiene igual.

En ambos casos al final del ciclo cuando $j = k$ se cumple que m es el índice de un mínimo de $A[i : j] = A[i : k]$; por lo tanto antes del siguiente ciclo, cuando $j = k + 1$, la invariante se mantiene.

- **Terminación:** Como la invariante se mantiene cuando $j = n - 1$, al término del último ciclo $A[m]$ es un mínimo de $A[i : n - 1]$.

Por lo tanto al término de la iteración en las líneas 4-6 del algoritmo [10.1](#), el índice m es el de un elemento mínimo del subarreglo $A[i : n - 1]$. \square

Resaltamos que siempre decimos **un mínimo**, no **el mínimo**, porque puede haber elementos repetidos en el arreglo y es posible que haya varios mínimos entonces. El índice m se actualiza nada más al primer mínimo que se encuentra.

Con este lema podemos demostrar que **SELECTIONSORT** es correcto.

Teorema 10.1. (**SELECTIONSORT** es correcto). Al terminar el algoritmo **SELECTIONSORT** el arreglo de entrada A está ordenado.

Demostración. Demostraremos la siguiente invariante:

Invariante: Para toda $i = 0, \dots, n - 1$, el subarreglo $A[0 : i - 1]$ está ordenado y los elementos del mismo son menores o iguales que los elementos de $A[i : n - 1]$.

- **Inicialización:** Antes del primer ciclo de la iteración cuando $i = 0$, trivialmente tenemos que $A[0 : -1]$ por vacuidad está ordenado y que los elementos del mismo son menores o iguales que los elementos de $A[0 : n - 1]$.
- **Mantenimiento:** Si el subarreglo $A[0 : k - 1]$ está ordenado y los elementos del mismo son menores o iguales que los elementos de $A[k : n - 1]$, antes del ciclo cuando $i = k$, entonces el subarreglo $A[0 : k]$ estará ordenado y los elementos del mismo serán menores o iguales que los elementos de $A[k + 1 : n - 1]$ antes del siguiente ciclo cuando $i = k + 1$.

Tenemos que $i = k$ y $A[0 : k - 1]$ con sus elementos menores o iguales

que los de $A[k : n - 1]$. Por el lema 10.1, al final del ciclo en las líneas 4-6 del algoritmo $A[m]$ será un mínimo del subarreglo $A[k : n - 1]$ y mayor o igual que los elementos de $A[0 : k - 1]$.

En la línea 7 intercambiamos $A[k]$ y $A[m]$; y entonces al final del ciclo cuando $i = k$ se cumple que $A[0 : k]$ está ordenado y todos sus elementos son menores o iguales que los de $A[k + 1 : n - 1]$; por lo tanto antes del siguiente ciclo cuando $i = k + 1$, la invariante se mantiene.

- *Terminación:* Como la invariante se mantiene cuando $i = n - 1$, al término del último ciclo $A[0 : n - 1]$ está ordenado.

Por lo tanto, al terminar el algoritmo SELECTIONSORT el arreglo A está ordenado. \square

Hacemos notar que también ocurre que todos los elementos de $A[0 : n - 1]$ son menores o iguales que todos los elementos de $A[n - 1 : n - 1]$, pero esto es únicamente una consecuencia de que A esté ordenado.

Antes de continuar, queremos resaltar dos hechos que nos parecen importantes. Primero: el algoritmo 10.1 se puede enunciar con 7 líneas de pseudocódigo, pero la demostración correspondiente de que es correcto nos llevó más de 2 páginas. Segundo: la demostración formal de que el algoritmo es correcto de ninguna manera nos garantiza que su implementación concreta vaya a funcionar. Como dijo desde 1970 Edsger Dijkstra [11], esto probablemente sea señal de que programar sea más difícil de lo que comúnmente se supone.

Por el teorema 10.1 el algoritmo SELECTIONSORT funciona; el único problema es que lo hace de una manera muy poco eficiente.

Vamos a contar el número de comparaciones que realiza el algoritmo SELECTIONSORT. Cuando $i = 0$, el algoritmo realiza la comparación $A[j] < A[m]$ un total de $n - 1$ veces, desde que $j = 1$ hasta que $j = n - 1$, inclusive; cuando $i = 1$, lo hace un total de $n - 2$ veces; etcétera. Esto nos lleva a una suma idéntica a la que vimos al inicio del capítulo 3; esta expresión es también la definición del número de combinaciones de n en 2:

$$\binom{n}{2} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Es una fórmula importante, sería provechoso que se la aprendieran de memoria como las tablas de multiplicar.

La expresión que representa las combinaciones de n en 2 nos dice el número de posibles pares no ordenados que se pueden hacer con n elementos. En el

contexto de `SELECTIONSORT` esto es bastante malo, porque quiere decir que estamos realizando *todas* las posibles comparaciones entre los elementos del arreglo A . Es malo porque estamos haciendo comparaciones de más: si sabemos que $a < b$ y que $b < c$, es absolutamente redundante el preguntar si $a < c$; lo podemos inferir de lo que ya sabemos.

El algoritmo `SELECTIONSORT` tiene complejidad en tiempo $O(n^2)$; por lo que acabamos de discutir, es de esperarse que se pueda ordenar un arreglo con una complejidad en tiempo menor. No lo veremos aquí, pero está demostrado [9] que lo mejor que se puede hacer, el óptimo en tiempo para hacer ordenamientos (no sólo en arreglos: cualquier tipo de ordenamientos), es $O(n \log n)$. *Si utilizamos comparaciones.*

Si no usamos comparaciones a veces lo podemos hacer más rápido: si nos piden ordenar un arreglo A con los n primeros números naturales, sencillamente llenamos el arreglo de tal forma que $A[i]=i$ y acabamos en tiempo $O(n)$. No usamos comparaciones porque teníamos información extra acerca de nuestra entrada; que el arreglo tenía nada más los n primeros números naturales.

Pero si no sabemos nada de antemano de la entrada, esto implicará que tendremos que comparar los elementos de la misma y entonces ordenar nos tomará al menos $O(n \log n)$ en tiempo.

Lo que hace a `SELECTIONSORT` y su complejidad en tiempo de $O(n^2)$ bastante malo. En memoria en cambio es bastante eficiente; `SELECTIONSORT` es un algoritmo de ordenamiento en lugar (*in situ* en latín, *in place* en inglés), lo que significa que el arreglo original se modifica, no que creamos una copia ordenada. Esto le da una complejidad en espacio de $O(1)$ a `SELECTIONSORT`, si no contamos la memoria utilizada por el arreglo de entrada.

Vamos a ver ahora un algoritmo para ordenar arreglos que sea más eficiente.

10.1.2. Algoritmo `QUICKSORT`

El algoritmo `QUICKSORT` fue ideado por Charles Antony Hoare en 1959 y publicado en 1961 [20]. `QUICKSORT`, como su nombre indica, es uno de los algoritmos más rápidos que existen para hacer ordenamientos en arreglos. Excepto cuando es muy lento.

La idea del algoritmo es seguir la estrategia divide y vencerás; se elige un elemento del arreglo como pivote p y se acomodan los otros elementos en el arreglo de tal forma que en el subarreglo $A[0 : i]$ (para alguna $0 \leq i < n$) quedan todos los elementos menores o iguales a p y en el subarreglo $A[i + 1 : n - 1]$ quedan todos los elementos mayores a p . Luego se

intercambian el elemento p y el que esté en $A[i]$, lo que es equivalente a poner a p en el lugar que le corresponde en el arreglo ordenado: si todos los elementos de $A[0 : i]$ son menores o iguales a p , esto hace a p un máximo de ese subarreglo y entonces su lugar es justamente $A[i]$. El algoritmo continúa recursivamente en los subarreglos $A[0 : i - 1]$ y $A[i + 1 : n - 1]$; la cláusula de escape es cuando entremos al algoritmo con un subarreglo vacío o de longitud 1 (figura 10.2).

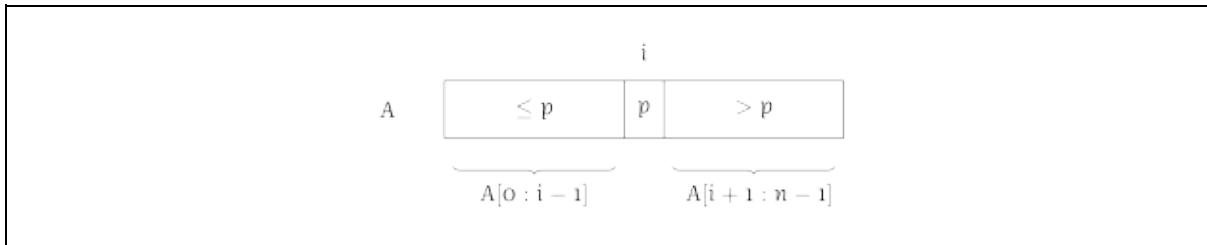


Figura 10.2: El algoritmo QUICKSORT elige un pivote p en el arreglo y deja todos los elementos en $A[0 : i - 1]$ menores o iguales a p ; todos los elementos en $A[i + 1 : n - 1]$ mayores a p ; y a p lo deja en $A[i]$ (el índice i varía dependiendo del valor del pivote). Por último continúa recursivamente en los subarreglos $A[0 : i - 1]$ y $A[i + 1 : n - 1]$.

El desempeño de QUICKSORT depende en gran medida del pivote que elijamos en cada llamada recursiva. Veremos más adelante que lo que nos conviene es que p quede lo más centrado posible en el subarreglo; lamentablemente no podemos garantizar eso de antemano o al menos no sin perder la complejidad en tiempo de $O(n \log n)$.

Lo más común es elegir un pivote aleatorio en el subarreglo, pero nosotros vamos sencillamente a tomar como pivote el primer elemento en el mismo. Si el arreglo es aleatorio (y en el caso promedio eso será lo que ocurra), elegir el primer elemento es equivalente a elegirlo aleatoriamente.

Vamos a describir el algoritmo paso por paso, primero en español: sean a y b los índices extremos de nuestro subarreglo, en otras palabras el algoritmo trabaja sobre el subarreglo $A[a : b]$. Si $b \leq a$ terminamos, pues esa es nuestra cláusula de escape (el subarreglo $A[a : b]$ es vacío o de tamaño 1 y trivialmente ordenado).

Si $a < b$, el pivote es $A[a]$ y vamos a llevar dos índices i y j , con $i = a + 1$ y $j = b$ inicialmente. Estos dos índices recorrerán el arreglo: i de izquierda a derecha y j de derecha a izquierda.

Este recorrido lo haremos iterativamente mientras $i < j$. Si $A[i] \leq A[a]$, el elemento $A[i]$ ya está en el subarreglo que le corresponde, así que incrementamos i . De la misma manera, si $A[j] > A[a]$ el elemento $A[j]$ ya está en el subarreglo que le corresponde y decrementamos j .

Si $A[i] > A[a]$ y además $A[j] \leq A[a]$, entonces intercambiamos $A[i]$ y $A[j]$, incrementamos i y decrementamos j .

Una vez terminada la iteración, existe la posibilidad de que $A[i] > A[a]$ si i y j se cruzaron en el último ciclo. Si ese es el caso, decrementamos i para que $A[i] \leq A[a]$, con lo que ya podemos intercambiar $A[a]$ y $A[i]$ para que $A[i]$ sea el pivote y esté en el lugar que le corresponde si el arreglo estuviera ordenado. Para terminar, hacemos recursión en $A[a : i - 1]$ y en $A[i + 1 : b]$.

La idea del algoritmo es que en cada llamada recursiva cada elemento de $A[a : b]$ queda en el subarreglo que le corresponde respecto a p y este subarreglo es cada vez más pequeño hasta que llega a tamaño 0 o 1. En ese momento el único elemento del subarreglo (si no es vacío) es el pivote y ya está en su lugar, por lo que al terminar el algoritmo el arreglo está completamente ordenado.

Con SELECTIONSORT vimos el pseudocódigo del algoritmo porque queríamos usarlo para hacer la demostración de que era correcto. Con QUICKSORT lo veremos sólo por razones didácticas; podemos ver QUICKSORT en el algoritmo [10.2](#).

```

procedure QUICKSORT( $A, a, b$ )
     $\triangleright$  Subarreglo  $A[a : b]$ 
    if ( $b \leq a$ )
        return
     $i \leftarrow a + 1$ 
     $j \leftarrow b$ 
    while ( $i < j$ )
        if ( $A[i] > A[a]$  and  $A[j] \leq A[a]$ )
            INTERCAMBIA( $A, i, j$ )
             $i \leftarrow i + 1$ 
             $j \leftarrow j - 1$ 
        elsif ( $A[i] \leq A[a]$ )
             $i \leftarrow i + 1$ 
        else
             $\triangleright$  Tiene que ocurrir que  $A[j] > A[a]$ 
             $j \leftarrow j - 1$ 
        if ( $A[i] > A[a]$ )
             $i \leftarrow i - 1$ 
        INTERCAMBIA( $A, a, i$ )
        QUICKSORT( $A, a, i - 1$ )
        QUICKSORT( $A, i + 1, b$ )
    
```

Algoritmo 10.2: Algoritmo QUICKSORT.

Volvemos a utilizar el algoritmo INTERCAMBIA que usamos en SELECTIONSORT; esto se traducirá a que nuestra clase **Arreglos** probablemente tendrá un método

auxiliar privado llamado `intercambia()`.

No vamos a hacer la demostración formal de que `QUICKSORT` funciona, pero es relativamente sencilla. Primero se demuestra el ciclo del **while** con la siguiente invariante: los elementos en $A[a : i - 1]$ son menores o iguales que $A[a]$ y los elementos en $A[j + 1 : b]$ son mayores que $A[a]$.

Después sólo hay que manejar el caso cuando i y j se cruzan (el último **if** se encarga de eso) y por inducción directa se puede demostrar que la doble recursión termina ordenando todo el arreglo (los algoritmos recursivos generalmente se pueden demostrar por inducción directa).

La idea de cubrir `QUICKSORT` era mostrar un algoritmo de ordenamiento de arreglos con mejor complejidad en tiempo que `SELECTIONSORT`; y en general ese es el caso. Veamos cómo es la complejidad en tiempo de `QUICKSORT`.

Para hacer más simple el análisis, supongamos que en cada llamada recursiva el pivote siempre termina en el centro del subarreglo. Por supuesto de hecho esto casi nunca ocurre; pero luego veremos que esta suposición no afecta tanto el resultado.

Si el pivote siempre está en el centro del subarreglo, cada subarreglo hará llamadas recursivas en dos subarreglos de tamaño aproximado a la mitad del arreglo original. Una vez más para simplificar el análisis, supongamos que cada arreglo se divide en dos arreglos exactamente del mismo tamaño e ignoraremos el hecho de que el pivote deja de ser parte de los subarreglos que se visitan recursivamente.

Comenzando con un arreglo de tamaño n , esto resulta en que se divide en dos subarreglos de tamaño $\frac{n}{2}$, que a su vez se dividen en dos subarreglos (cuatro en total) de tamaño $\frac{n}{4}$, etcétera. Esto se repite hasta llegar a n subarreglos de tamaño 1 y ya no hay más recursión porque se cumple la cláusula de escape. El número de veces que podemos dividir n entre 2 hasta llegar a 1 es justamente la definición de $\log_2 n$.

En cada uno de todos esos subarreglos los índices i y j recorren la longitud completa del subarreglo hasta que se cruzan; en el arreglo de tamaño n los índices i y j hacen n movimientos, en los arreglos de tamaño $\frac{n}{2}$ los índices i y j hacen $\frac{n}{2}$ movimientos, etcétera. Como hay 2 subarreglos de tamaño $\frac{n}{2}$, 4 de tamaño $\frac{n}{4}$ y en general 2^k de tamaño $\frac{n}{2^k}$ ($1 \leq k \leq \lfloor \log_2 n \rfloor$), en cada nivel de recursión se realizan en total n movimientos.

Esto nos permite visualizar el número de movimientos de los índices como un árbol binario de altura $\log_2 n$ y en donde en cada nivel se realizan un total n movimientos (figura 10.3).

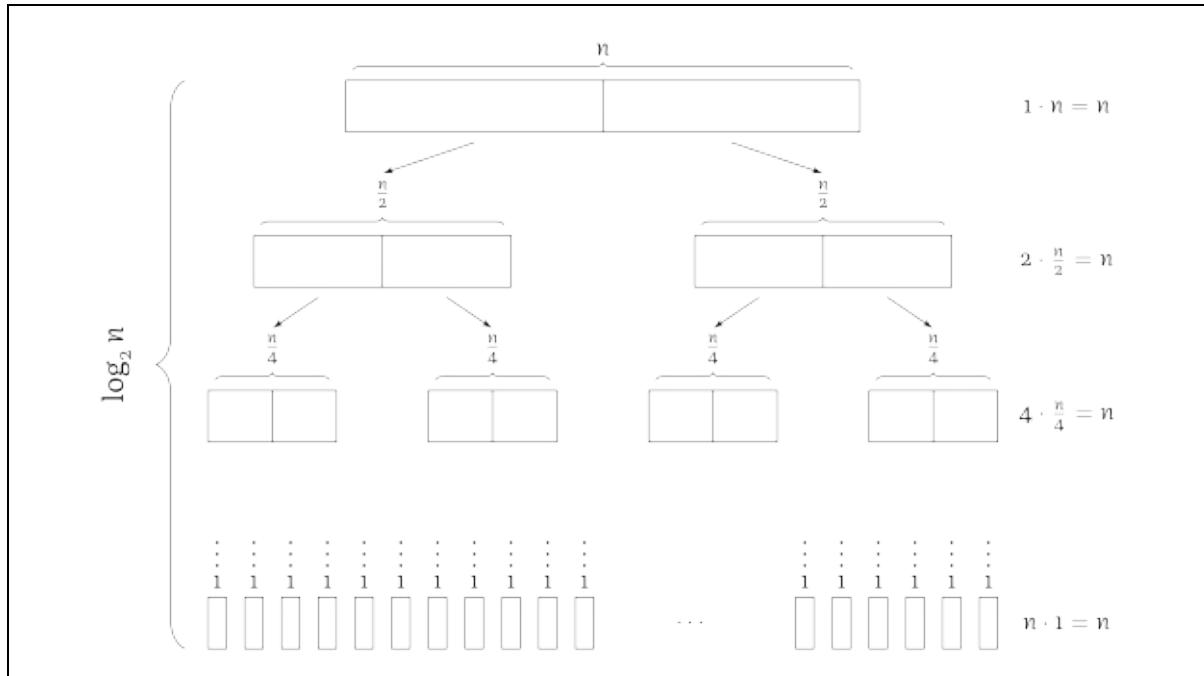


Figura 10.3: Análisis de movimientos de los índices i y j bajo la suposición de que los subarreglos se dividen siempre a la mitad. El total de $n \log_2 n$ justifica una complejidad en tiempo de $O(n \log n)$.

Esto nos da un total de $n \log_2 n$ movimientos, que se puede usar para justificar una complejidad en tiempo de $O(n \log n)$; recordemos que

$$\log_2 n = \frac{\log n}{\log 2} = \left(\frac{1}{\log 2} \right) \log n.$$

La razón por la cual **QUICKSORT** es mucho mejor que **SELECTIONSORT** es porque realiza muchas menos comparaciones. **SELECTIONSORT** compara todos los elementos contra todos, básicamente; **QUICKSORT** compara todos contra un pivote, lo que divide el arreglo de entrada en dos subarreglos, donde sabemos que todos los elementos del subarreglo $A[a : i - 1]$ son menores que todos los elementos del subarreglo $A[i + 1 : b]$, sin *jamás* comparar ningún elemento de $A[a : i - 1]$ con algún elemento de $A[i + 1 : b]$.

Esta complejidad de $O(n \log n)$ sólo se justifica bajo las suposiciones que hicimos, que son bastante fuertes. Sin embargo se puede demostrar que, siempre y cuando el pivote no se vaya *casi siempre* a los extremos del subarreglo, esta complejidad se mantiene. El pivote puede incluso caer en los extremos *algunas veces* y que la complejidad siga manteniéndose, el problema es cuando cae *casi siempre* en los extremos.

Estamos siendo ambiguos a propósito porque la explicación formal es más bien compleja. Para simplificarlo, veamos el otro extremo: vamos a suponer ahora que el pivote *siempre* cae al inicio del subarreglo. Recordemos que al elegir el pivote *no sabemos* dónde terminará en el subarreglo, eso sólo lo

determinamos después de comparar todos los elementos del subarreglo contra él.

Si el pivote termina siempre en el índice a , entonces la llamada recursiva se hace con un arreglo vacío ($A[a : a - 1]$) y un arreglo de $b - a$ elementos ($A[a + 1 : b]$). Entonces en la primera llamada a `QUICKSORT` se ejecutan $n - 1$ comparaciones, en la segunda $n - 2$, etcétera (figura 10.4).

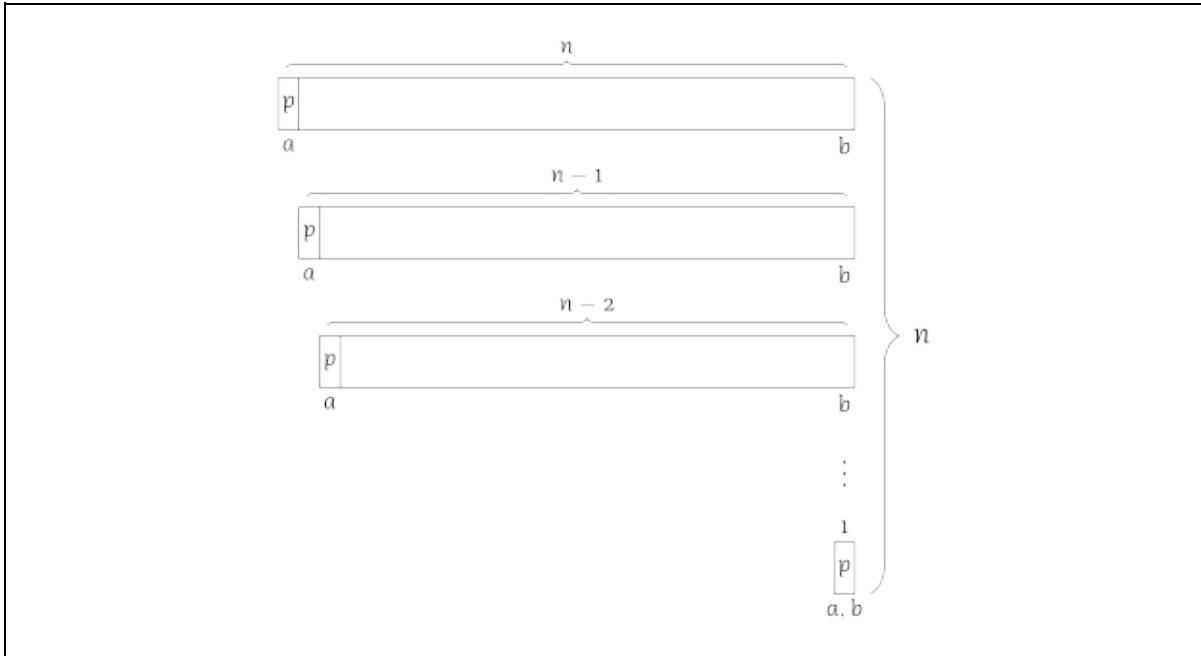


Figura 10.4: Análisis de movimientos de los índices i y j bajo la suposición de que el pivote siempre termina en el índice a . El total de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ justifica una complejidad en tiempo de $O(n^2)$.

Que nos lleva de regreso a nuestra vieja fórmula:

$$\binom{n}{2} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Tenemos entonces un caso extremo donde la complejidad en tiempo del algoritmo es $O(n^2)$ y otro caso extremo donde la complejidad en tiempo es $O(n \log n)$.

También hay que explicar que no hay forma de escoger el pivote de tal manera que siempre termine en el centro del subarreglo; si el algoritmo recibe como entrada un arreglo con n entradas idénticas, entonces el pivote siempre va a terminar al final del subarreglo, no importa cómo lo escojamos. Y no podemos verificar primero que las entradas del arreglo sean distintas, porque si recibimos un arreglo de tamaño n con $n - c$ entradas idénticas (con c una constante relativamente pequeña), ya no podemos hacer la verificación y de nuevo el pivote caerá *casi siempre* al final del subarreglo.

Se puede hacer el análisis detallado (que no haremos aquí, pero lo pueden consultar en [9]) y ver que *en promedio* la complejidad de QUICKSORT será de $O(n \log n)$. Esto quiere decir que, suponiendo que todas las permutaciones de los elementos son igualmente probables, el algoritmo en general ejecutará una constante por $n \log n$ instrucciones. Pero *a veces* (y realmente no se puede determinar el 100% de las veces cuándo será), el algoritmo se irá a $O(n^2)$ y ejecutará una constante por n^2 instrucciones.

La diferencia es *muy* significativa, como veíamos en el capítulo 3: con un arreglo de 1,000 entradas la diferencia entre $O(n^2)$ y $O(n \log n)$ es la diferencia entre 500,000 y 10,000, casi dos órdenes de magnitud.

Sin embargo, cuando es $O(n \log n)$ el algoritmo QUICKSORT es de los más rápidos que existen, con una constante oculta por la O grandota muy pequeña. Y como *en promedio* implica que en la gran mayoría de los casos el algoritmo se comporta como queremos, es un algoritmo muy utilizado y favorecido por muchos programadores.

Además de que en algunos casos tiene comportamiento cuadrático, QUICKSORT tiene otra desventaja: no es estable. Esto quiere decir que si dos elementos x y y en el arreglo se comparan iguales y x tiene un índice menor que el de y en el arreglo antes de ejecutar QUICKSORT, ese orden no necesariamente se preservará después de ejecutar el algoritmo; depende de cuál de los dos fuera elegido como pivote primero. Esto no es muy importante al estar ordenando enteros, pero en otras situaciones puede hacer al algoritmo básicamente inservible.

Supongamos que tenemos un arreglo de estudiantes y queremos ordenarlo por número de cuenta y después por nombre. Si ordenáramos primero por número de cuenta y luego por nombre, podría ocurrir que dos estudiantes homónimos terminarán primero el que tenga número de cuenta mayor, seguido del que tiene número de cuenta menor. Por supuesto esto se puede resolver creando una lambda que pregunte primero número de cuenta y después nombre, pero el punto es que la no estabilidad de QUICKSORT sí es una desventaja que tiene que considerarse.

En espacio el equivalente del comportamiento cuadrático en tiempo también se presenta: si ordenamos un arreglo con n elementos repetidos, esto causa que la pila de ejecución crezca en n registros de activación. En este caso la complejidad en espacio es $O(n)$. Pero incluso en el caso promedio la complejidad es $O(\log n)$, que es el número de llamadas recursivas que le lleva al algoritmo en llegar a un arreglo de longitud 0 o 1. Así que en ambos casos el algoritmo es peor en memoria que SELECTIONSORT, aunque también sea en lugar.

Y no se resuelve convirtiendo el algoritmo a iterativo: para hacer eso necesitamos crear una pila auxiliar dónde poner los índices de los subarreglos que hay que seguir ordenando y el uso de memoria es idéntico: podemos ver la versión iterativa de `QUICKSORT` en el algoritmo [10.3](#).

```

procedure QUICKSORT( $A, n$ )
     $\triangleright$  Arreglo  $A$  de  $n$  entradas,  $n > 0$ 
     $P \leftarrow \text{PILA}()$ 
     $\triangleright$  Creamos una pila  $P$ 
    METE( $P, 0$ )
    METE( $P, n - 1$ )
    while (not EsVACÍA( $P$ ))
         $b \leftarrow \text{SACA}(P)$ 
         $a \leftarrow \text{SACA}(P)$ 
         $i \leftarrow a + 1$ 
         $j \leftarrow b$ 
        while ( $i < j$ )
            if ( $A[i] > A[a]$  and  $A[j] \leq A[a]$ )
                INTERCAMBIA( $A, i, j$ )
                 $i \leftarrow i + 1$ 
                 $j \leftarrow j - 1$ 
            elsif ( $A[i] < A[a]$ )
                 $i \leftarrow i + 1$ 
            else
                 $\triangleright$  Tiene que ocurrir que  $A[j] \geq A[a]$ 
                 $j \leftarrow j - 1$ 
            if ( $A[i] > A[a]$ )
                 $i \leftarrow i - 1$ 
            INTERCAMBIA( $A, a, i$ )
            METE( $P, a$ )
            METE( $P, i - 1$ )
            METE( $P, i + 1$ )
            METE( $P, b$ )
    
```

Algoritmo 10.3: Algoritmo `QUICKSORT` iterativo. Utilizamos una pila auxiliar P para replicar el comportamiento del algoritmo recursivo.

El algoritmo `QUICKSORT` es de hecho peor que `SELECTIONSORT` en el caso cuadrático: aunque ambos sean $O(n^2)$, la constante de `QUICKSORT` es mucho mayor porque es costoso computacionalmente el estar invocando métodos y porque el uso de memoria es también mayor.

De todas maneras, en el caso promedio `QUICKSORT` es de los algoritmos de

ordenamiento más rápidos que existen.

10.1.3. Manteniendo arreglos ordenados

Ahora que ya podemos ordenar arreglos, las limitaciones de los mismos vuelven a hacerse notar. No podemos eliminar elementos de un arreglo manteniéndolo ordenado en tiempo menor a lineal, a menos que permitamos “hoyos” en el mismo, lo cual generalmente termina siendo demasiado problemático como para que valga la pena. Mucho menos podemos agregar elementos a un arreglo manteniéndolo ordenado en tiempo menor a lineal, cuando tampoco podemos hacerlo si no nos fijamos en el orden.

Si nuestra colección de datos es extremadamente dinámica (se agregan y eliminan varios elementos en distintos puntos durante la ejecución del programa), no tiene mucho sentido usar un arreglo para ella, mucho menos copiarla a un arreglo y ordenarla ahí. Vamos a necesitar estructuras más adecuadas para estos casos.

Pero si los datos no cambian después de adquirirlos (o no cambian mucho), usar un arreglo y ordenarlo con `QUICKSORT` será, en promedio, la mejor opción.

10.2. Ordenamientos en listas

Antes de ver el único algoritmo para ordenamientos en listas que cubriremos, hay que hacer notar un hecho obvio: podemos usar `SELECTIONSORT` y `QUICKSORT` para ordenar listas. No nada más de la forma trivial, copiando la lista a un arreglo, ordenándolo y regresándolo a otra lista (o a la misma); también podemos implementar los mismos algoritmos para listas, porque realmente nunca usamos la propiedad de acceder el i -ésimo elemento de un arreglo en tiempo $O(1)$.

Cuando accedemos a un elemento del arreglo con índice i en `SELECTIONSORT` y `QUICKSORT`; o bien $i = 0$ o $i = n - 1$; o justo antes habíamos accedido $i - 1$ o $i + 1$. Por lo tanto podríamos implementar ambos algoritmos en listas; necesitaríamos un método `intercambia()` que intercambiara los elementos de dos nodos sin modificar sus ligas siguiente y anterior y otras cosas del estilo, pero es factible.

Sólo no es muy interesante; en su lugar veremos un algoritmo especial para ordenar listas: `MERGESORT`. Aunque, irónicamente, el mismo fue implementado originalmente para ordenar arreglos.

10.2.1. Algoritmo MERGESORT

MERGESORT fue inventado por John von Neumann en 1945 [31] para ordenar arreglos, pero disponiendo de listas ligadas rápidamente se hizo evidente que funcionaba muy bien para ellas. El núcleo del algoritmo MERGESORT es un algoritmo auxiliar que llamaremos MEZCLA, que es la traducción de *merge*.

El algoritmo MEZCLA recibe dos listas ya ordenadas y regresa una lista ordenada que tiene todos los elementos de las listas de entrada. Para hacer esto se crea una nueva lista L y dos iteradores i y j (nodos temporales en nuestra implementación en Java) que se ponen al inicio de cada lista de entrada. Mientras ninguno de los iteradores llegue al final de su lista, se comparan los elementos en cada uno: si el elemento de i es menor o igual que el elemento de j , agregamos el elemento de i a L y movemos i a su siguiente; si no, agregamos el elemento de j a L y movemos j a su siguiente. Cuando alguno de los iteradores llegue al final de su lista, tomamos el resto de los elementos de la otra lista y los agregamos a L (figura 10.5).

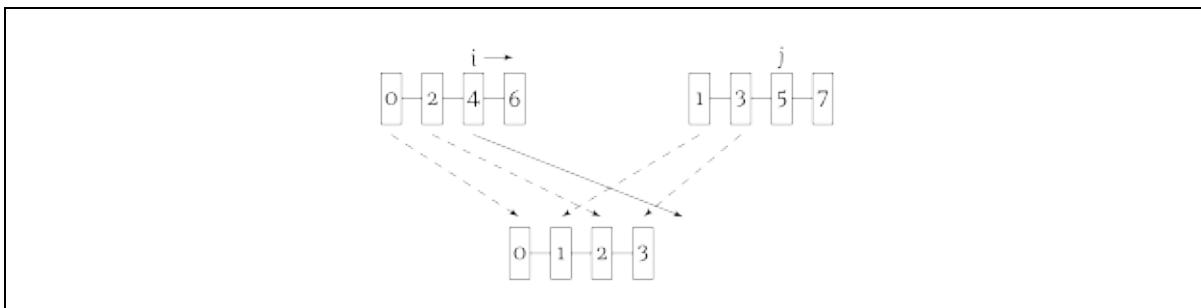


Figura 10.5: El algoritmo MEZCLA, como su nombre indica, mezcla dos listas ordenadas en una lista ordenada. En este paso del algoritmo, el elemento 4 está a punto de agregarse a la lista mezclada porque $4 \leq 5$; inmediatamente después, el iterador i se moverá a su siguiente.

Con el algoritmo MEZCLA podemos proceder con el algoritmo MERGESORT, que recibe una lista L de n elementos. Se parte L en dos listas L_1 y L_2 , cada una con $\frac{n}{2}$ elementos (si n es impar, L_1 tendrá $\frac{n-1}{2}$ elementos y L_2 tendrá $\frac{n-1}{2}+1$ elementos). Hacemos recursión con L_1 y L_2 , guardando el resultado de cada llamada recursiva en L_1 y L_2 de nuevo, lo que resulta en que las dos listas estén ordenadas. Después las mezclamos usando MEZCLA y regresamos la lista mezclada, que ya está ordenada. La cláusula de escape es cuando la lista L es de longitud 0 o 1, en cuyo caso sólo regresamos una copia de la misma.

Si suponemos que el algoritmo MEZCLA funciona, la demostración de que MERGESORT es correcto es relativamente simple.

Teorema 10.2. (MERGESORT es correcto). Al terminar el algoritmo MERGESORT la lista de entrada L está ordenada.

Demostración. Por inducción sobre la longitud de la lista: para el caso base cuando $n < 2$, el algoritmo funciona porque la cláusula de escape se ejecuta de inmediato y nos regresa una copia de la lista. Como la lista regresada tiene cero o un elementos, está trivialmente ordenada.

Suponemos que el algoritmo funciona cuando $n < k$ para $k \geq 2$. Sea L una lista de longitud k , sin pérdida de generalidad supondremos que k es par. Lo primero que hace el algoritmo es dividir L en dos listas L_1 y L_2 de longitud $\frac{k}{2}$ y llamar el algoritmo recursivamente sobre ellas. Como $\frac{k}{2}$ por hipótesis de inducción las listas actualizadas L_1 y L_2 están ordenadas y procedemos a mezclarlas con MEZCLA. Como suponemos que el algoritmo MEZCLA funciona, la lista mezclada está ordenada y como es lo que regresa el algoritmo, el mismo es correcto. \square

Demostrar que MEZCLA es correcto no es muy complicado, pero lo omitiremos. La invariante es: la lista L está ordenada y tiene los elementos de la primera lista a la izquierda de i y los elementos de la segunda lista a la izquierda de j .

Lo que hace MERGESORT es dividir la lista recursivamente en listas cada vez más pequeñas hasta tener listas de longitud 0 o 1. Regresando de la recursión entonces *siempre* tenemos una lista ordenada, primero de longitud 0 o 1 y después utilizamos mezcla para ir juntando las listas de par en par hasta obtener la lista completa ya ordenada (figura 10.6)

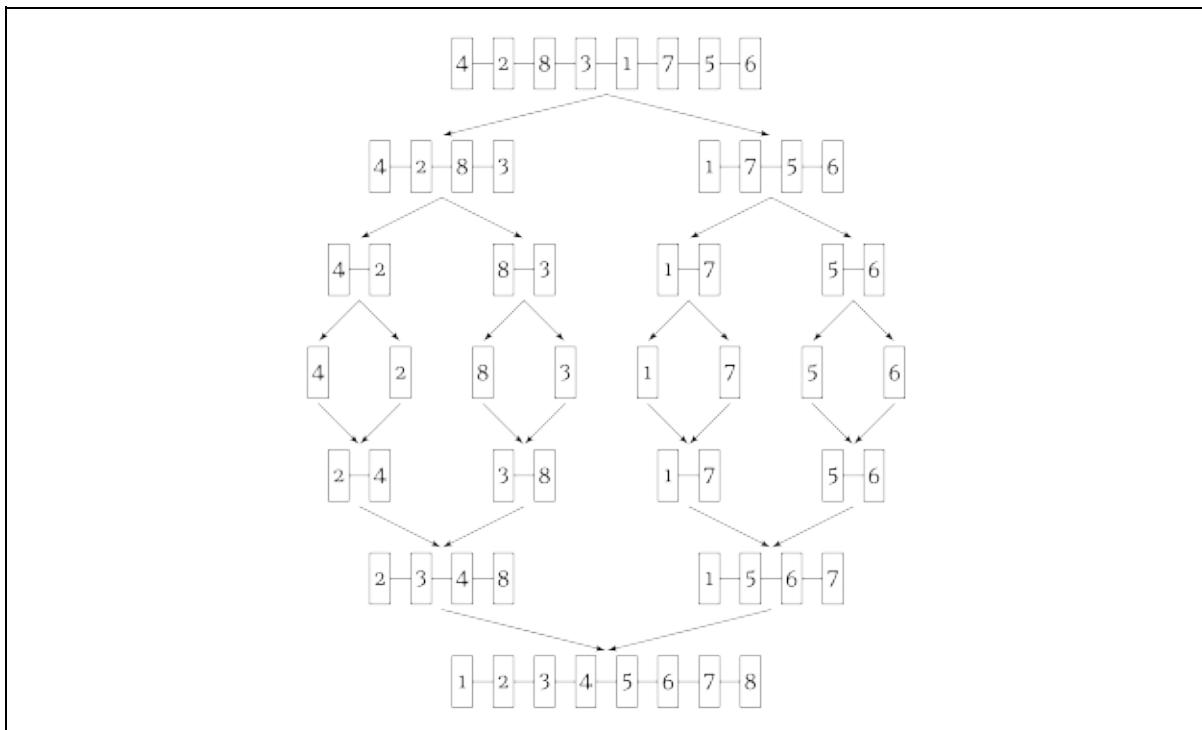


Figura 10.6: El algoritmo MERGESORT, divide las listas hasta tener listas de longitud 0 o 1 y por lo tanto ordenadas. Después las va mezclando par a par para reconstruir la lista ordenada.

El algoritmo MERGESORT se comporta de manera similar a QUICKSORT cuando el pivote termina en el centro del arreglo; la diferencia es que MERGESORT *siempre* divide a la lista L en dos listas cuya longitud difiere a lo más en 1.

Esto garantiza una complejidad en tiempo de $O(n \log n)$ *siempre*, no importa cómo esté formada la lista de entrada. Entonces, contrario a QUICKSORT, no existe ningún caso cuando la complejidad en tiempo del método sea cuadrática. Además MERGESORT es estable; dos elementos en la lista que se evalúen iguales preservarán en la lista ordenada el orden que tenían en la lista original, porque si el elemento en el iterador i es igual al elemento en el iterador j , agregamos el elemento en i primero.

Lamentablemente, la constante oculta por la notación de O grandota será mucho más grande para MERGESORT que para QUICKSORT cuando ambos sean $O(n \log n)$: la creación de tantas listas tiene su costo (se crean dos listas por cada llamada recursiva que no termina en la cláusula de escape).

En espacio MERGESORT es peor o igual que QUICKSORT: al inicio el algoritmo tiene la lista de tamaño n (que como siempre no consideramos). Esta se divide en dos listas de tamaño $\frac{n}{2}$ y se manda llamar el algoritmo recursivamente con la primera. También se hace recursión sobre la segunda lista, pero para ese momento ya acabó la recursión de la primera y no es irracional suponer que la memoria utilizada por toda esa rama recursiva ya ha sido liberada.

Entonces podemos suponer que en cada llamada recursiva el algoritmo asigna la misma cantidad de memoria que la que usa la lista de entrada. Por lo tanto se asigna n al inicio, $\frac{n}{2}$ en la segunda llamada, $\frac{n}{4}$ en la tercera, etcétera. Esto resulta en un consumo de memoria equivalente a

$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Por lo tanto el consumo de memoria está entre n (por la primera copia en dos listas que siempre se realiza) y $2n$; y entonces la complejidad en espacio de MERGESORT es $O(n)$. Al igual que con la complejidad en tiempo, la complejidad en espacio siempre es la misma, independientemente de cómo sea lista de entrada. Por último, no podemos dejar de lado el hecho que, en Java en particular, el recolector de basura es el que decide cuándo se libera la memoria de las listas temporales creadas durante la recursión del algoritmo. Esto implica que, en el peor de los casos, hasta $O(n \log n)$ de memoria puede estar siendo usada por el algoritmo; literalmente todas las listas de la figura [10.6](#).

QUICKSORT puede llegar a ser un orden de magnitud más rápido que

MERGESORT; esto puede ser suficiente para justificar el riesgo de que en algunos casos QUICKSORT tome tiempo cuadrático. Dependerá del problema.

El algoritmo MERGESORT es la opción conservadora; si es completamente inaceptable el que nuestro ordenamiento tome más de $O(n \log n)$, por bajo que sea el riesgo, será el algoritmo que utilizaremos.

10.2.2. Manteniendo listas ordenadas

Podría parecer que vamos a tener más suerte manteniendo listas ordenadas que con arreglos, dado que son una estructura dinámica. Dada una lista ordenada podemos eliminar un elemento y la lista se mantiene ordenada; lamentablemente eliminar tiene una complejidad en tiempo lineal, entonces no mejora la situación con arreglos.

Y podemos agregar un elemento a una lista ordenada manteniéndola ordenada, sencillamente recorriendo la lista y buscando el lugar del nuevo elemento; lamentablemente esto también tiene complejidad lineal en tiempo.

Así que al final de cuentas las listas no mejoran significativamente la situación que teníamos con arreglos, al menos respecto a mantenerlas ordenadas. Como decíamos en la sección [10.1.3](#), vamos a necesitar estructuras más adecuadas para mantener ordenadas colecciones altamente dinámicas.

10.3. La razón para ordenar colecciones

Teniendo disponibles algoritmos de ordenamiento para arreglos y listas, una pregunta sensata es: ¿para qué las queremos ordenadas? En muchos casos la respuesta es sencillamente para tener los datos ordenados, por ejemplo para presentarlos al usuario en una interfaz gráfica.

Sin embargo una razón importante para ordenar nuestras colecciones será el poder buscar en sus elementos rápidamente. Esto lo podremos hacer, en algunos casos: lo veremos en el capítulo [11](#).

Hay muchos más algoritmos de ordenamiento para listas y arreglos; algunos son incluso lineales en tiempo al aprovechar cierta información extra del arreglo o lista de entrada. No cubriremos estos algoritmos por no considerarlos fundamentales para un curso de Estructuras de Datos, pero sí veremos en subsecuentes capítulos cómo poder ordenar datos en estructuras no lineales y (probablemente más interesante) cómo mantener colecciones de datos ordenadas utilizando este tipo de estructuras.

Ejercicios

1. Implementa el método `mergesort()` de la clase `Lista`, y los métodos faltantes de la clase `Arreglos`.
2. ¿Cuál es la complejidad en tiempo del algoritmo `QUICKSORT`?
3. ¿Cuál es la complejidad en espacio del algoritmo `QUICKSORT`?
4. ¿Cuál es la complejidad en tiempo del algoritmo `MERGESORT`?
5. ¿Cuál es la complejidad en espacio del algoritmo `MERGESORT`?

11. Búsquedas

Como decíamos al final del capítulo [10](#), una razón posible para ordenar nuestras estructuras lineales (listas y arreglos) es para ser capaces de realizar búsquedas en ellas con una complejidad en tiempo mejor a la que tenemos ahora.

Si una lista o arreglo no está ordenado, la única manera de realizar búsquedas es revisando los elementos de la estructura uno por uno. Si el elemento que estamos buscando no está en la colección, entonces revisaremos *todos* los elementos antes de poder decir con certeza que ese es el caso.

Esto se traduce en una complejidad en tiempo lineal para realizar búsquedas cuando nuestra estructura lineal no está ordenada. Vamos a ver qué tanto podemos mejorar nuestros algoritmos de búsqueda cuando sí esté ordenada.

11.1. Búsquedas en arreglos

Para realizar búsquedas en arreglos ordenados vamos a utilizar un algoritmo que se llama **BÚSQUEDABINARIA**. El algoritmo recibe un arreglo *A*, los índices *a* y *b* que determinan el subarreglo en el cual trabajar y un elemento *e* para buscar. El algoritmo regresa el índice del elemento en el arreglo o -1 si el elemento no está en el arreglo.

El método correspondiente estará también en nuestra clase **Arreglos** y como fue el caso en el capítulo [10](#), tendremos versiones genérica acotada a comparables y genérica sin cotas; una vez más el método acotado a **Comparable** mandará llamar el método no acotado utilizando una lambda (listado [11.1](#)).

```
public static <T> int
busquedaBinaria(T[] arreglo, T elemento,
                Comparator<T> comparador) {
    /* ... */
}
public static <T extends Comparable<T>> int
busquedaBinaria(T[] arreglo, T elemento) {
    return busquedaBinaria(arreglo, elemento,
                           (a,b) -> a.compareTo(b));
}
```

Listado 11.1: Firmas de los métodos para realizar búsquedas binarias en arreglos.

Como con **QUICKSORT**, el algoritmo **BÚSQUEDABINARIA** trabajará con un

subarreglo $A[a : b]$ y será recursivo, pero la cláusula de escape será únicamente cuando el arreglo sea vacío (o en otras palabras si $b < a$). En ese caso regresará -1 porque ya no habremos encontrado elemento.

Si el subarreglo $A[a : b]$ no es vacío, se toma el índice medio del mismo $m = \left\lfloor \frac{a+b}{2} \right\rfloor$. Si el elemento e que estamos buscando es igual a $A[m]$, regresamos m . Si no, tenemos dos casos:

- $e < A[m]$: si el elemento está en $A[a : b]$, entonces estará en el subarreglo a la izquierda de m ; hacemos recursión sobre $A[a : m - 1]$.
- $e > A[m]$: si el elemento está en $A[a : b]$, entonces estará en el subarreglo a la derecha de m ; hacemos recursión sobre $A[m + 1 : b]$.

La demostración de que el algoritmo es correcto se les deja como ejercicio: es muy sencilla y se puede hacer por recursión directa.

El algoritmo BÚSQUEDA BINARIA es *mucho* mejor en tiempo que recorrer los elementos del arreglo uno por uno. Al igual que QSORT el algoritmo va haciendo recursión sobre subarreglos cada vez más pequeños; pero no recorre cada subarreglo y hace recursión nada más sobre el subarreglo de un lado. Además, como MERGESORT, *siempre* divide a la mitad cada arreglo para crear los subarreglos (figura 11.1).

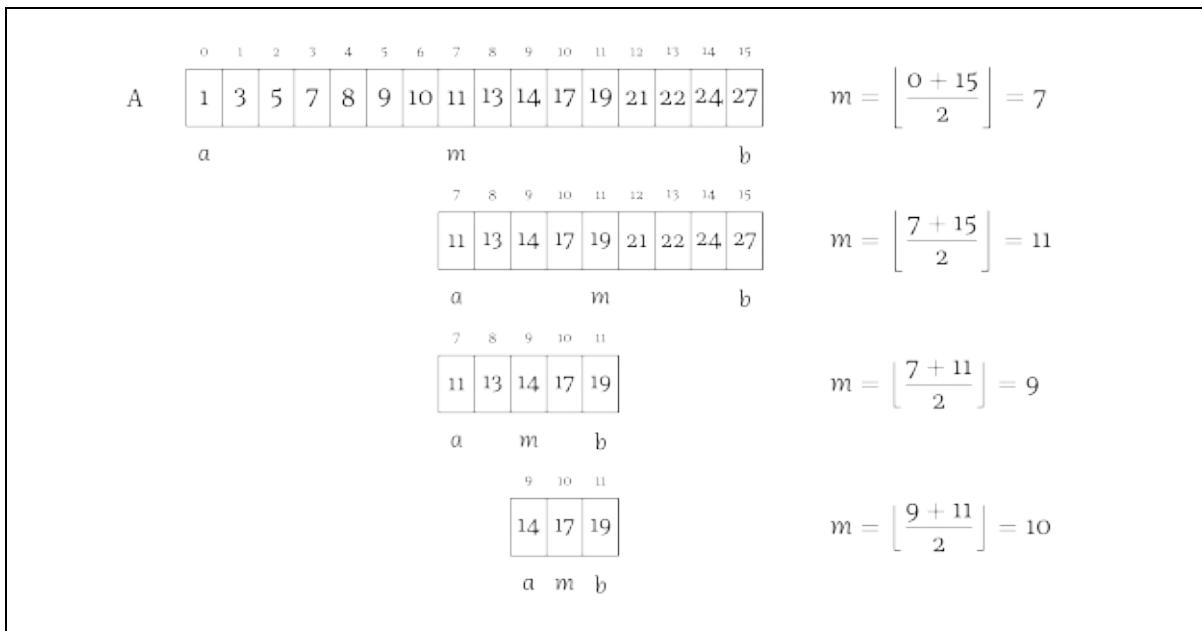


Figura 11.1: Los subarreglos a los que QSORT entra recursivamente al buscar el elemento 17. El subarreglo $A[9 : 11]$ es el último, porque en este paso recursivo $A[m]=17$.

Como en cada paso recursivo sólo se realiza un número de operaciones constante, la complejidad total del algoritmo está determinada por cuántas veces se hace recursión. Dado que siempre se divide al arreglo a la mitad, esto

es a lo más $\log_2 n$, lo que nos da una complejidad en tiempo de $O(\log n)$.

En espacio, el algoritmo es $O(\log n)$ si descontamos la memoria utilizada por el arreglo; al ser recursivo, es el número de registros de activación que se cargan en la pila de ejecución y por lo tanto es el número de llamadas recursivas.

Para mostrar lo significativo que es esto, imaginen que hacemos una búsqueda binaria en un arreglo con 1,000,000 elementos. El número de recursiones que se realizan es básicamente $\lfloor \log_2(1,000,000) \rfloor = 19$, por lo que el algoritmo ejecuta un número de instrucciones que es *varios órdenes* de magnitud menor que el tamaño del arreglo.

La primera referencia que existe del algoritmo de búsqueda binaria la dio John Mauchly en 1946 (mencionado en [\[22\]](#)) y aunque el algoritmo es en general sencillo de entender, muchos detalles suelen escapársele a programadores novatos e incluso experimentados. En particular, muchas implementaciones sufren de un error que varias bibliotecas estándares tienen (incluyendo la de Java durante 9 años): para calcular el punto medio del subarreglo usan el promedio tal cual está descrito en el algoritmo: $\left\lfloor \frac{a+b}{2} \right\rfloor$. Esto puede resultar en un desborde con números enteros con signo en complemento a 2; si $a + b$ rebasa el tamaño del entero más grande dada la magnitud en bits disponible. En la implementación se debe utilizar $m = a + \left\lfloor \frac{b-a}{2} \right\rfloor$: si así calculamos el promedio nunca ocurrirá un desborde.

Buscar un elemento en un arreglo ordenado es de las cosas más rápidas que podemos hacer; aunque técnicamente existen complejidades entre $O(1)$ y $O(\log n)$ (por ejemplo $O(\log^{1/2} n)$), al analizar algoritmos lo que no toma tiempo (o memoria) constante suele considerarse equivalente a $O(\log n)$; y en general tiempo constante sólo será alcanzable para cosas no muy interesantes, como fórmulas.

Lamentablemente, con listas no seremos tan afortunados.

11.2. Búsquedas en listas

Tener las listas ordenadas no nos ayuda a realizar las búsquedas más rápido, al contrario de lo que pasa con arreglos. Y el problema está en que no podemos acceder el i -ésimo elemento de la lista en tiempo $O(1)$.

La complejidad en tiempo de buscar un elemento en una lista ordenada será $O(n)$; podemos hacerlo un poco más rápido que en una lista desordenada (si un elemento de la lista es mayor que el elemento que estamos buscando, ya no

va a estar si la lista es ordenada), pero será lineal en el peor de los casos.

De todas formas y por completez vamos a implementar BÚSQUEDALINEAL en la clase `Lista`; aunque no mejora la complejidad en tiempo, sí podemos realizar la búsqueda un poco más rápido en varios casos, en particular cuando el elemento a buscar es menor que el primer elemento de la lista ordenada.

Las firmas de los métodos `busquedaLineal()` serán iguales a las de los métodos `busquedaBinaria()`, excepto por la versión que recibe un comparador, que no será estática (listado 11.2).

```
public boolean busquedaLineal(T elemento,
                               Comparator<T> comparador) {
    /* ... */
}

public static <T extends Comparable<T>>
boolean busquedaLineal(Lista<T> lista, T elemento) {
    return lista.busquedaLineal(elemento,
                               (a, b) -> a.compareTo(b));
}
```

Listado 11.2: Firmas de los métodos para realizar búsquedas lineales en listas.

Al igual que con ordenamientos, casi todas las subsecuentes estructuras de datos que cubramos en el libro involucrarán algoritmos para realizar búsquedas en las mismas; todas las colecciones al fin y al cabo implementan el método `contiene()`.

Varias de las técnicas y conceptos visitados en estos dos últimos capítulos (10 y 11) serán parecidos o nos resultarán útiles al revisar los algoritmos para ordenar y buscar elementos en el resto del libro.

Ejercicios

1. Implementa los métodos `busquedaLineal()` y `busquedaBinaria()` de las clases `Lista` y `Arreglos`.
2. ¿Cuál es la complejidad en tiempo del algoritmo BÚSQUEDABINARIA?
3. ¿Cuál es la complejidad en espacio del algoritmo BÚSQUEDABINARIA?

12. Árboles binarios

Los árboles binarios (*binary trees* en inglés) serán la primera estructura de datos no lineal que veremos en este libro; contrario a las estructuras anteriores que hemos visto, primero veremos la definición del concepto de árbol binario y hasta el final del capítulo veremos el comportamiento de la clase.

12.1. Definición de árboles binarios

Si suponemos conocimiento de Teoría de Gráficas (que veremos un poco en el capítulo [17](#)), una definición posible de árboles binarios es la siguiente.

Definición 12.1. (Árboles binarios, primer intento) *Los árboles binarios son gráficas conexas sin ciclos donde todos los vértices tienen a lo más grado 3.*

La definición [12.1](#) es absolutamente correcta, nada más no nos ayuda mucho a programarlos. Así que haremos algo similar a las listas; primero definiremos vértices de árbol binario y luego a los árboles binarios.

Definición 12.2. (Vértices de árbol binario) *Un vértice de árbol binario (o vértice si el contexto no es ambiguo) es:*

- *el vértice vacío (\emptyset) o*
- *un elemento, un vértice padre, un vértice hijo izquierdo y un vértice hijo derecho.*

Además, para todo par de vértices a y b , a es el vértice padre de b si y sólo si b es el vértice hijo izquierdo o vértice hijo derecho de a .

A los vértices hijos izquierdo y derecho en general les diremos sencillamente izquierdo y derecho o los hijos cuando hablamos de ambos. Con la definición [12.2](#) podemos definir los árboles binarios:

Definición 12.3. (Árboles binarios) *Los árboles binarios son un vértice especial que llamaremos raíz. Siempre se cumplirá que:*

- *el árbol no tiene elementos si y sólo si la raíz es \emptyset ,*
- *si la raíz es distinta de \emptyset entonces su padre es \emptyset ,*
- *todo elemento del árbol está en un vértice v tal que existe una secuencia única de vértices $v_1, \dots, v_k = v$, donde v_1 es la raíz del*

árbol y v_i es el vértice padre de v_{i+1} , $i = 1, \dots, k-1$.

La figura 12.1 muestra la forma en que normalmente se representan los árboles binarios, con la raíz hasta arriba y los vértices creciendo hacia abajo cuando nos vamos por el vértice izquierdo o derecho.

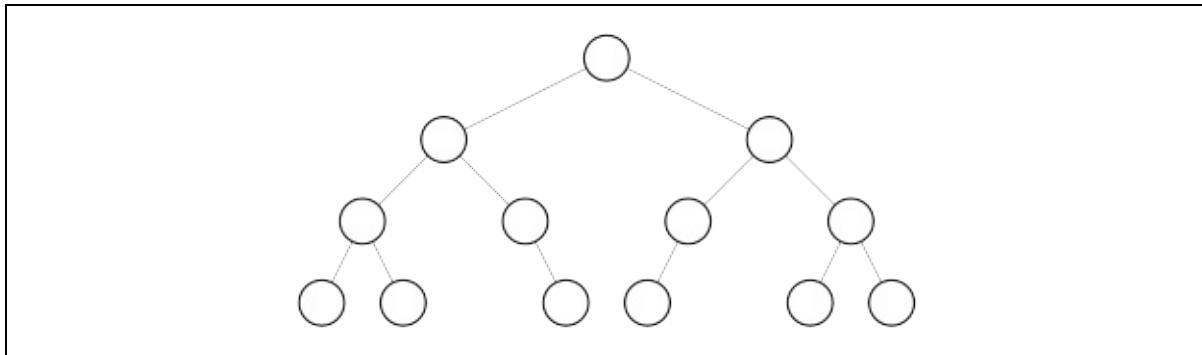


Figura 12.1: Árbol binario.

Antes de comenzar a ver la definición en Java de la clase `ArbolBinario`, su comportamiento y los algoritmos que implementaremos, vamos a definir ciertas propiedades de los árboles binarios. Tendremos cuatro clases que heredarán a `ArbolBinario` (en los capítulos 13, 14, 15 y 16, respectivamente) así que estas definiciones se aplicarán a todas ellas.

12.2. Propiedades de árboles binarios

Como veremos más adelante la eficiencia de muchos algoritmos para nuestros árboles binarios dependerá de la altura del mismo; lo cual puede ser sorprendentemente no trivial de definir.

Para simplificarnos la tarea, en ésta y múltiples otras definiciones, realizaremos primero la definición para vértices y después trivialmente usaremos esto para definir la propiedad en nuestro árboles.

Definicion 12.4. (Altura de vértices) *La altura de un vértice v , denotada por $h(v)$ es:*

- -1 , si el vértice es \emptyset ,
- 1 más el máximo de la altura de su vértice izquierdo y su vértice derecho, en otro caso.

Definir la altura del vértice vacío como -1 puede resultar contra intuitivo, pero es válido y sin duda alguna lo más sencillo (además de que nos permitirá programar de manera muy simple el algoritmo que calcule la altura de un árbol). Con la definición 12.4 podemos definir trivialmente la altura de un

árbol binario.

Definicion 12.5. (Altura de árboles binarios) *La altura de un árbol binario T , denotada por $h(T)$, es la altura de su vértice raíz.*

Pensar que el árbol con un único elemento (con nada más la raíz) tiene altura cero sí es bastante razonable. El árbol de la figura 12.1 tiene altura 3 bajo la definición 12.5.

Un concepto relacionado será la profundidad de un vértice en un árbol.

Definicion 12.6. (Profundidad de vértices) *Se v un vértice de un árbol binario T . La profundidad del vértice es:*

- 0, si el vértice es la raíz.
- 1 más la profundidad del padre en otro caso.

Sería equivalente definir la profundidad de un vértice v como el número de elementos en la secuencia única $v_1, \dots, v_k = v$ de vértices de T tal que v_1 es la raíz de T y v_i es el padre de v_{i+1} , $i = 1, \dots, k-1$.

Podríamos definir la “profundidad” de un árbol como el máximo de la profundidad de todos sus vértices, pero esto siempre es igual a la altura. Un concepto más interesante que se extrae de la profundidad de los vértices es el de nivel.

Definicion 12.7. (Niveles de árboles binarios) *El i -ésimo nivel de un árbol binario T , $0 \leq i \leq h(T)$, son todos los vértices en el mismo tales que su profundidad es i .*

De la definición 12.7 se sigue que un árbol binario T tiene $h(T)+1$ niveles. Los niveles de un árbol binario van a ser importantes por varios motivos; en particular nos va a interesar cuándo un nivel está lleno.

Definicion 12.8. (Niveles llenos) *Sea un árbol binario T ; diremos que el i -ésimo nivel de T está lleno si y sólo si el número de vértices en el nivel es 2^i .*

Si el árbol no es vacío entonces su nivel 0 siempre está lleno. En la figura 12.2 los cuatro niveles 0, 1, 2 y 3 están llenos. Cuando esto ocurra (que un árbol binario tenga todos sus niveles llenos), diremos que el árbol binario mismo está lleno.

Definicion 12.9. (Árboles binarios llenos) *Diremos que un árbol binario T está lleno si y sólo si todos sus niveles $0, \dots, h(T)$ están llenos.*

Podemos ver un árbol binario lleno en la figura [12.2](#).

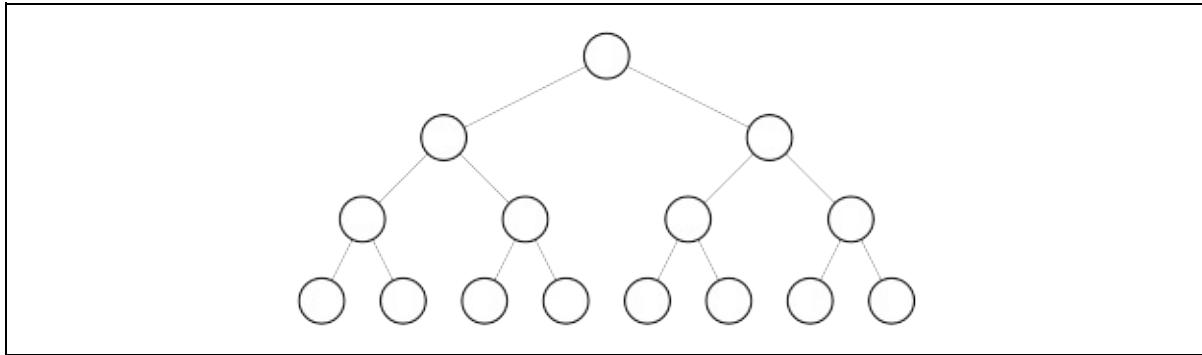


Figura 12.2: Árbol binario lleno.

Notemos que no hay manera de agregar un nuevo elemento a un árbol binario lleno sin aumentar su altura en uno y que deje de ser lleno (excepto por el caso trivial cuando es vacío). Tampoco podemos eliminarle ningún elemento sin que deje de ser lleno (excepto en el caso trivial cuando tiene un único elemento).

Los niveles de un árbol binario lleno crecen duplicando el número de vértices del nivel anterior, porque cada vértice del nivel $i - 1$ tiene dos hijos: el izquierdo y el derecho. Por lo tanto los vértices en el último nivel son más de la mitad de los elementos del árbol; por ejemplo, la figura [12.1](#) tiene 8 vértices en el último nivel y 7 vértices en el resto del árbol.

Los árboles binarios llenos son importantes pero muy restrictivos: combinatoriamente (si ignoramos los elementos en los vértices) sólo existe un árbol binario lleno de altura k , para cualquier $k \geq 0$. Además no podemos agregarles o eliminarles elementos de uno en uno sin que dejen de ser llenos; vamos a necesitar aceptar árboles no llenos si queremos trabajar con algo que tenga algunas de sus características.

Si un nivel no está lleno entonces podemos imaginarlo como si conceptualmente tuviera “hoyos”. Para el tipo de árboles que veremos en el capítulo [13](#), dónde están estos hoyos es importante; vamos entonces a definir una manera de poder explicar dónde está un vértice y consecuentemente un hoyo.

Definicion 12.10. (Coordenadas de vértices) *Sea T un árbol binario no vacío y v un vértice en el mismo. Las coordenadas (x, y) del vértice v , denotadas por $T_x(v)$ y $T_y(v)$, se definen como siguen:*

- *La raíz de T tiene coordenadas $(0, 0)$.*
- *Sea p el parent de v . Si v es el hijo izquierdo de p entonces $T_x(v)=2T_x(p)$; si v es el hijo derecho de p entonces*

$$T_x(v)=2T_x(p)+1. \text{ Además } T_y(v)=T_y(p)+1.$$

La coordenada- y de un vértice siempre es su profundidad y por lo tanto el nivel al que pertenece. Esto nos permite ordenar los vértices en un nivel por su coordenada- x y así podemos detectar hoyos en un árbol: si el vértice (i, j) no existe, $0 \leq i < 2^j$, pero sí hay otros vértices en el nivel j , entonces el árbol tiene un hoyo en la coordenada (i, j) , como se ve en la figura [12.3](#).

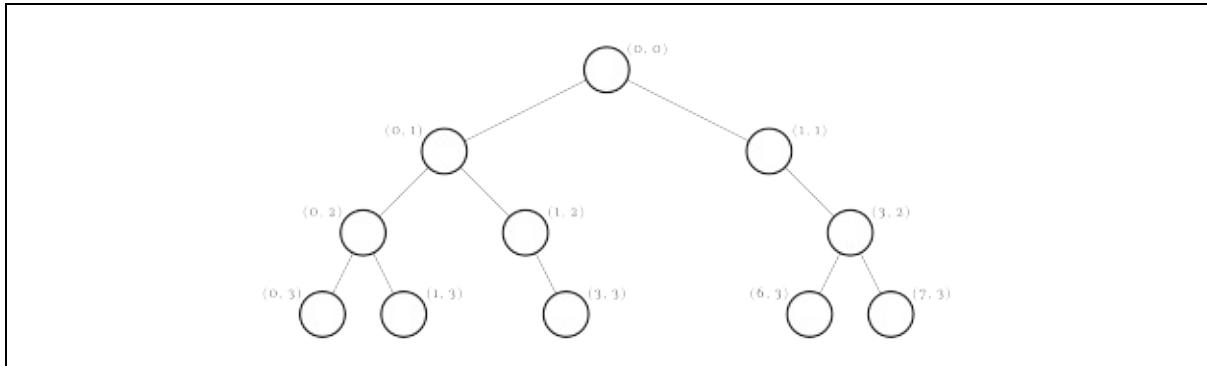


Figura 12.3: Coordenadas en un árbol binario. Este árbol tiene hoyos: los vértices en las coordenadas $(2, 2)$, $(2, 3)$, $(4, 3)$ y $(5, 3)$ no existen, pero sí hay otros vértices en los niveles 2 y 3.

También vamos a definir una manera de distinguir a los vértices que no tengan hijos de los demás.

Definicion 12.11. (Vertices hojas e internos de árboles binarios.) *Sea un árbol binario T . A los vértices de T cuyos vértices izquierdo y derecho sean \emptyset los denominaremos vértices hojas de T . Al resto los denominaremos vértices internos.*

Normalmente nos referiremos a un vértice hoja sencillamente como una hoja. Para terminar esta sección definiremos una propiedad sencilla de los vértices que utilizaremos en capítulos subsecuentes.

Definicion 12.12. (Dirección de vértices) *Sea un árbol binario T . Diremos de un vértice v de T que su dirección es izquierda si v es el hijo izquierdo de su padre o derecha si v es el hijo derecho de su padre. El vértice raíz no tiene dirección y es el único vértice con esta característica.*

Si dos vértices tienen dirección diferente normalmente diremos que están *cruzados*. Para simplificar, si un vértice tiene dirección izquierda diremos que el vértice es izquierdo y que es derecho si su dirección es derecha.

12.3. Implementación en Java

Como debió quedar claro desde que dimos la definición [12.3](#), la clase

`ArbolBinario` tendrá una clase interna `Vertice` para los vértices de árboles binarios. Esta clase interna será similar a la clase `Nodo` de la clase `Lista`, pero tendrá ciertas diferencias muy significativas que veremos en un momento.

Nuestra clase `ArbolBinario` será genérica e implementará a `colección`, como casi todas nuestras estructuras, porque un árbol binario cumple con todos los requisitos de una colección. Sin embargo, no podemos definir de manera concreta las operaciones para agregar y eliminar elementos, ni cómo deben actuar los iteradores, porque los árboles tendrán distintas formas de agregar y eliminar elementos, así como de iterarse, dependiendo de qué tipo sean.

Esto resultará en que la clase `ArbolBinario` será abstracta y no definirá los métodos `agrega()`, `elimina()` ni `iterator()`, así como tampoco la clase interna para iteradores. Lo que implicará que la clase `Vertice` deberá ser protegida; para preservar el encapsulamiento de datos pero permitir a las clases que extiendan a `ArbolBinario` usar los vértices.

Los iteradores presentan otro problema; son muy restrictivos. Sólo nos permiten iterar el árbol “en orden”, donde ese orden dependerá del tipo de árbol. Nos gustaría poder recorrer nuestro árbol como nosotros quisiéramos; dado un vértice, nos gustaría poder obtener el padre, el izquierdo o el derecho.

No podemos hacer esto, porque violaría el encapsulamiento de datos, así que sencillamente generalizaremos la idea detrás de los iteradores de Java y definiremos otro tipo de embajadores o intermediarios entre el estado no público de los árboles y el resto del mundo.

De hecho hicimos algo muy similar con las listas, usando la interfaz `IteradorLista`; pero siendo los árboles binarios una estructura no lineal, no nos bastará con extender `Iterator`: vamos a definir una nueva interfaz para estos embajadores para árboles binarios.

Esta interfaz de hecho será una manera de ver a los vértices verdaderos del árbol; no como los iteradores de las listas, que son objetos distintos a los nodos de las mismas. Pero esta interfaz será de sólo lectura, lo que nos permitirá preservar el encapsulamiento de datos.

La interfaz `VerticeArbolBinario`, que implementará la clase interna `Vertice`, nos dará operaciones para saber si el vértice tiene padre, izquierdo o derecho, que serán equivalentes al método `hasNext()`. También tendrá operaciones para obtener cualquiera de esos tres vértices, equivalentes al método `next()`, sólo que funcionarán en tres “direcciones” distintas. Como no se mezclarán las opciones de obtener un elemento y moverse (porque no se “moverán” los vértices, sólo podemos obtener los vértices vecinos inmediatos a partir de otro), tendremos también un método que nos dará el elemento en un vértice. Por último implementaremos las dos operaciones que definimos para vértices

en las definiciones [12.4](#) y [12.6](#), para obtener la altura y profundidad del vértice.

Esto nos permite definir la interfaz `VerticeArbolBinario`, que mostramos en el listado [12.1](#). Hacemos énfasis en que la interfaz nos permite consultar el estado de un vértice, pero no *modificarlo*; la consistencia del estado del objeto queda entonces exclusivamente en manos de la clase (y sus descendientes, porque es protegida).

```
public interface VerticeArbolBinario<T> {
    public boolean hayPadre();
    public boolean hayIzquierdo();
    public boolean hayDerecho();
    public VerticeArbolBinario<T> padre();
    public VerticeArbolBinario<T> izquierdo();
    public VerticeArbolBinario<T> derecho();
    public int altura();
    public int profundidad();
    public T get();
}
```

Listado 12.1: La interfaz `VerticeArbolBinario`.

La clase abstracta `ArbolBinario`, además de los métodos definidos en la interfaz `colección`, tendrá unas cuantas operaciones extra; varias existirán únicamente pensando a futuro en las clases que extenderán `ArbolBinario` de manera concreta:

- `raiz()`

Regresa un `VerticeArbolBinario` con la raíz del árbol, pero lanzará `NoSuchElementException` si el árbol es vacío.

- `altura()`

Nos regresa la altura del árbol.

- `busca()`

Recibe un elemento de tipo `T` y regresa un `VerticeArbolBinario` con el vértice que contiene al elemento o `null` si el elemento no está contenido en el árbol.

- `nuevoVertice()`

Un método protegido auxiliar para las clases que extiendan a `ArbolBinario`; algunas de estas clases tendrán una clase especial para vértices que extenderá `Vertice`: si utilizamos `new Vertice<T>()` para crear vértices dentro de `ArbolBinario`, en todas las clases que la extiendan *también* estaríamos

instanciando con `Vertice`, cuando querríamos utilizar la clase que la extienda. Al definir una operación (protegida) para crear vértices, le permitimos a las clases que extiendan el sobrecargarlo para que creen el tipo de vértice que necesiten.

- `vertice()`

Otro método protegido auxiliar para las clases que extiendan a `ArbolBinario`; tendremos varios lugares donde tendremos un `VerticeArbolBinario`, pero querremos un `Vertice`. Podemos hacer audiciones en cada uno de estos casos, pero mejor creamos un método auxiliar para que sea el único lugar donde se necesiten hacer.

Con esto tenemos el comportamiento de nuestros árboles. Estructuralmente y como dice su definición, la única variable de clase que es obligatoria es un vértice con la raíz del árbol. Sin embargo usaremos la misma técnica que en la clase `Lista` y tendremos un contador que incrementaremos al agregar elementos y decrementaremos al eliminarlos, para que el método `getElementos()` sea $O(1)$ en tiempo. El esqueleto de `ArbolBinario` se puede ver en el listado 12.2.

```
public abstract class ArbolBinario<T>
    implements Coleccion<T> {

    protected class Vertice
        implements VerticeArbolBinario<T> {

            public T elemento;
            public Vertice padre;
            public Vertice izquierdo;
            public Vertice derecho;
            public Vertice(T elemento) { /* ... */ }
            @Override public boolean
            hayPadre() { /* ... */ }
            @Override public boolean
            hayIzquierdo() { /* ... */ }
            @Override public boolean
            hayDerecho() { /* ... */ }
            @Override public VerticeArbolBinario<T>
            padre() { /* ... */ }
            @Override public VerticeArbolBinario<T>
            izquierdo() { /* ... */ }
            @Override public VerticeArbolBinario<T>
            derecho() { /* ... */ }
            @Override public int altura() { /* ... */ }
            @Override public int profundidad() { /* ... */ }
            @Override public T get() { /* ... */ }
            @Override public String toString() { /* ... */ }
        }
}
```

```

@Override public boolean
equals(Object o) { /* ... */ }
}

protected Vertice raiz;
protected int elementos;

public ArbolBinario() { /* ... */ }
public ArbolBinario(Coleccion<T> colección) {
/* ... */
}

@Override public boolean
contiene(T elemento) { /* ... */ }
@Override public boolean esVacia() { /* ... */ }
@Override public int getElementos() { /* ... */ }
@Override public void limpia() { /* ... */ }
@Override public boolean equals(Object o) { /* ... */ }
@Override public String toString() { /* ... */ }
protected Vertice
nuevoVertice(T elemento) { /* ... */ }
protected Vertice
vertice(VerticeArbolBinario<T> vertice) { /* ... */ }
public VerticeArbolBinario<T>
busca(T elemento) { /* ... */ }
public VerticeArbolBinario<T>
raiz() { /* ... */ }
public int altura() { /* ... */ }
}

```

Listado 12.2: Esqueleto de la clase `ArbolBinario`.

Como mencionábamos al inicio de la sección la clase no implementará los métodos `agrega()` ni `elimina()` de `Colección`, ni tampoco definirá sus iteradores: esto será tarea de las clases concretas.

Lo único que falta entonces por ver son los algoritmos de cada uno de estos métodos; sin embargo no podremos utilizar la clase hasta que hayamos implementado una clase concreta que la extienda.

12.4. Algoritmos para árboles binarios

Veremos los algoritmos en el orden de los métodos correspondientes en el listado 12.2. Casi todos los métodos de la clase `Vertice` son muy sencillos; en particular con complejidad en tiempo y en espacio constante. Sólo hay que recordar que los métodos `padre()`, `izquierdo()` y `derecho()` terminarán con un error cuando el vértice requerido no existe.

Las excepciones son `altura()`, `profundidad()`, `toString()` y `equals()`, que tienen algoritmos sencillos pero interesantes; los tres tienen complejidad en tiempo $O(n)$ y en espacio también, en el peor caso por la pila de ejecución, excepto por `toString()`. Sólo veremos estos métodos de `Vertice`.

- `altura()`

El algoritmo es implementar la definición [12.4](#); la altura del vértice es 1 más el máximo de la altura del izquierdo y la altura del derecho.

En la implementación en Java nada más hay que tener cuidado de no tratar de invocar un método con una referencia `null`; sencillamente hay que verificar que el izquierdo o derecho no sean `null` antes de llamar el método `altura()` recursivamente; sigue siendo recursión, porque recuerden que en Orientación a Objetos el objeto que manda llamar un método siempre se considera entrada de dicho método.

- `profundidad()`

De igual manera el algoritmo es implementar la definición [12.6](#); si el vértice es la raíz (en otras palabras si su padre es \emptyset), su profundidad es 0. Si no es 1 más la recursión sobre el padre del vértice.

- `toString()`

Sencillamente regresamos la representación en cadena del elemento en el vértice. Algunas de las clases que extenderán `Vertice` sobrecargarán el método.

- `equals()`

Implementamos `equals()` en `Vertice` para hacer más la implementación de `equals()` en `ArbolBinario`. Una vez más tenemos que hacer una audición de una clase no genérica a una clase genérica (la clase `Vertice` es genérica implícitamente porque usa el tipo `T`) y suprimir la advertencia resultante (listado [12.3](#)).

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    Vertice vertice = (Vertice)o;
    // ...
}
```

Listado 12.3: Inicio del método `equals()` en `Vertice`.

Como en los capítulos [15](#) y [16](#) extenderemos la clase `Vertice`, utilizar el

método `getClass()` en lugar de usar `instanceof` vuelve a ser obligatorio.

El algoritmo es sencillamente comprobar que los elementos del vértice que llama el método y del vértice recibido sean iguales; y que recursivamente los vértices izquierdos sean iguales y los derechos también. Una vez más en Java hay que verificar que los hijos no sean `null` antes de hacer la recursión.

Esto termina los métodos de la clase `vertice`. Ahora veremos los de `ArbolBinario` propiamente.

- `ArbolBinario()` y `ArbolBinario(Coleccion<T>)`

El primer constructor no hará nada; nada más tenemos que declararlo porque de otra forma lo perderíamos si declaramos nada más `ArbolBinario(Coleccion<T>)`. Tiene complejidad $O(1)$ en tiempo y en espacio.

El constructor que recibe una colección sencillamente la recorre y agrega los elementos de la misma al árbol usando el algoritmo para agregar. No importa que no esté implementado en esta clase, las clases concretas *tienen* que implementarlo y por lo tanto podemos usarlo aquí. La complejidad en tiempo y en espacio dependerá del algoritmo que utilicemos para agregar vértices y del estado del árbol.

- `contiene()`

Utilizamos el mismo algoritmo que usa `busca()` (y por lo tanto comparte sus complejidades en tiempo y en espacio) y comprobamos que el resultado no sea \emptyset .

- `esVacia()`

Comprobamos si la raíz es \emptyset ; es constante en tiempo y en espacio.

- `getElementos()`

Regresamos nuestro contador auxiliar, así que también es constante en tiempo y en espacio.

- `limpia()`

Hacemos \emptyset la raíz y reiniciamos el contador de elementos en cero. Constante en tiempo y espacio.

- `equals()`

```
    @Override public boolean equals(Object o) {  
        if (o == null || getClass() != o.getClass())  
            return false;
```

```

@SuppressWarnings("unchecked")
ArbolBinario<T> arbol = (ArbolBinario<T>)o;
// ...
}

```

Listado 12.4: El inicio del método `equals()` de `ArbolBinario`.

Tenemos de nuevo que hacer una audición y suprimir advertencias (listado [12.4](#)).

Como implementamos `equals()` en `Vertice`, el algoritmo es sencillamente comprobar que la raíz del árbol que llama al método sea igual que la raíz del árbol recibido; por lo tanto también es $O(n)$ en tiempo y en espacio la complejidad dependerá de la altura del árbol, pues eso determina cuánto crece la pila de ejecución.

- `toString()`

La representación en cadenas de nuestros árboles binarios será con los niveles verticales en lugar de horizontales; podríamos hacerlos horizontales, pero es mucho más sencillo de la otra manera. Por ejemplo, tomemos el árbol en la figura [12.4](#).

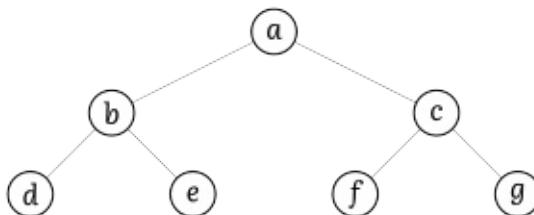


Figura 12.4: Árbol binario para representar en cadena.

Este árbol representado como cadena se verá como en la figura [12.5](#).

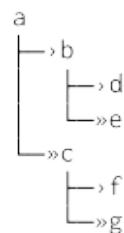


Figura 12.5: Representación en cadena del árbol de la figura [12.4](#).

Utilizamos `>` para distinguir a los vértices que sean izquierdos de los derechos, con los que usaremos `>>`. La explicación detallada de cómo vamos a recorrer el árbol para construir esta cadena la veremos en el capítulo [14](#); por esto y como el algoritmo es el más complejo que veremos para representar una de nuestras estructuras como cadena,

vamos a mostrar el pseudocódigo para hacerlo.

En cada línea de la cadena habrá el elemento de un vértice en el extremo derecho y a su izquierda el símbolo para representar si es izquierdo o derecho (”>” o ”>>”). Antes de esto habrá símbolos para representar las ramas (”L”, ”|”, ”|”, ”-”) o espacios (”U”). Vamos a necesitar un algoritmo auxiliar para determinar cuándo agregar una rama y cuándo agregar espacios. El algoritmo recibirá un arreglo binario con un 1 para representar una rama vertical y un cero para representar un espacio. El algoritmo auxiliar lo podemos ver en el algoritmo [12.1](#).

```
procedure DIBUJAESPACIOS(l, A)
    ▷ Nivel l, arreglo binario A
    s  $\leftarrow$  ””
    ▷ Cadena vacía
    for (i  $\leftarrow$  0 to l - 1)
        if (A[i]=1)
            s  $\leftarrow$  s + ”|UU”
            ▷ Agregamos una barra vertical y dos espacios a la cadena
        else
            s  $\leftarrow$  s + ”UUU”
            ▷ Agregamos tres espacios a la cadena
    return s
```

Algoritmo 12.1: Algoritmo para crear la cadena antes de un vértice.

Con el algoritmo [12.1](#) podemos definir el algoritmo recursivo que nos dará la representación en cadena del árbol. El algoritmo crea la cadena con la línea correspondiente a cada vértice utilizando el algoritmo auxiliar [12.1](#), concatenándole los símbolos correspondientes dependiendo de si esa rama seguirá si tiene un hermano o no. Podemos verlo en el algoritmo [12.2](#).

```
procedure TOSTRING(v, l, A)
    ▷ Vértice v, nivel l y arreglo binario A
    s  $\leftarrow$  TOSTRING(v) +  $\leftarrow$ 
    ▷  $\leftarrow$  representa un salto de línea
    A[l]  $\leftarrow$  1
    if (IZQUIERDO(v)  $\neq$   $\emptyset$  and DERECHO(v)  $\neq$   $\emptyset$ )
        s  $\leftarrow$  s + DIBUJAESPACIOS(l, A)
        s  $\leftarrow$  s + ”|>”
        s  $\leftarrow$  s + TOSTRING(IZQUIERDO(v), l + 1, A)
        s  $\leftarrow$  s + DIBUJAESPACIOS(l, A)
```

```

 $s \leftarrow s + \text{“} \text{“}$ 
 $A[l] \leftarrow 0$ 
     $\triangleright$  Dejamos de dibujar la rama correspondiente al vértice
     $s \leftarrow s + \text{TOSTRING}(\text{DERECHO}(v), l + 1, A)$ 
elsif ( $\text{IZQUIERDO}(v) \neq \emptyset$ )
     $s \leftarrow s + \text{DIBUJAESPACIOS}(l, A)$ 
     $s \leftarrow s + \text{“} \text{“}$ 
     $A[l] \leftarrow 0$ 
     $\triangleright$  Dejamos de dibujar la rama correspondiente al vértice
     $s \leftarrow s + \text{TOSTRING}(\text{IZQUIERDO}(v), l + 1, A)$ 
elsif ( $\text{DERECHO}(v) \neq \emptyset$ )
     $s \leftarrow s + \text{DIBUJAESPACIOS}(l, A)$ 
     $s \leftarrow s + \text{“} \text{“}$ 
     $A[l] \leftarrow 0$ 
     $\triangleright$  Dejamos de dibujar la rama correspondiente al vértice
     $s \leftarrow s + \text{TOSTRING}(\text{DERECHO}(v), l + 1, A)$ 

```

Algoritmo 12.2: Algoritmo recursivo para crear la cadena que representa la línea de vértices.

Lo único que falta es el algoritmo principal, el que recibe el árbol y manda llamar el recursivo con la raíz. Lo podemos ver en el algoritmo [12.3](#).

```

procedure  $\text{TOSTRING}(T)$ 
     $\triangleright$  Árbol binario  $T$ 
    if ( $\text{RAÍZ}(T) = \emptyset$ )
        return “”
     $A \leftarrow \text{ARREGLO}(\text{ALTURA}(T) + 1)$ 
    for ( $i \leftarrow 0$  to  $\text{ALTURA}(T) + 1$ )
         $A[i] \leftarrow 0$ 
    return  $\text{TOSTRING}(\text{RAÍZ}(T), 0, A)$ 

```

Algoritmo 12.3: Algoritmo para crear una representación en cadena de un árbol.

El algoritmo para representar árboles binarios como cadenas es el más complicado del estilo en todo el libro; tiene complejidad $O(n)$ en tiempo y en espacio. El resto de nuestras estructuras tendrán representaciones en cadena muy simples o las simplificaremos porque una cadena no será capaz de representar de la mejor manera la estructura.

- `nuevoVertice()`

El algoritmo sencillamente regresa una nueva instancia de `Vertice`. Como mencionábamos anteriormente, la idea es que las clases que extienden `ArbolBinario` y a su vez declaran una clase interna que extienda a `Vertice`

sobrecarguen este método para que pueda regresar instancias de esta nueva clase. Dentro de la clase `ArbolBinario` (y en todas sus herederas) no debe utilizarse `new Vertice()` nunca; los vértices deben crearse usando este método. Es constante en tiempo y espacio.

- `vertice()`

Como se decía arriba, lo único que hace este método es hacer la audición de `VerticeArbolBinario` a `Vertice`, para que no se tenga que hacer en otro lado. Hacemos notar que no es necesario suprimir advertencias en esta audición: como `Vertice` implementa `VerticeArbolBinario` y ambas son genéricas, el compilador permite la audición sin mostrar advertencia.

El método es constante en tiempo y espacio.

- `busca()`

El algoritmo hace recursión comenzando con la raíz; si el vértice es \emptyset se regresa \emptyset ; si no y el elemento del vértice es igual al recibido, se regresa el vértice; si no, se busca recursivamente en el izquierdo y el derecho, si alguno es distinto de \emptyset se regresa ese, si no se regresa \emptyset .

Este algoritmo tiene complejidad en tiempo lineal (recorre todos los vértices del árbol) y en espacio puede ser logarítmico o lineal, dependiendo de la altura del árbol. Veremos con más detalle esto en las clases concretas que extiendan `ArbolBinario`.

- `raiz()`

Se regresa la raíz del árbol utilizando tiempo y espacio constantes. Si el árbol es vacío ocurre un error.

- `altura()`

Se regresa -1 , si la raíz es \emptyset ; si no se regresa la altura de la raíz y por lo tanto tiene la misma complejidad en tiempo y espacio que calcular la altura de un vértice.

12.5. Aprovechando referencias no utilizadas

Como mencionábamos anteriormente, el número de hojas puede ser mayor que el número de vértices internos en un árbol binario. Y aunque éste no sea el caso, el número de referencias a \emptyset (o `null` en nuestra implementación en Java) siempre es mayor que el número de referencias a un vértice existente.

Hace algunas décadas las restricciones de memoria en las computadoras eran tales que el hecho de que la mayor parte de las referencias en una estructura

de datos no se utilizaran se consideraba un desperdicio *enorme* de recursos. Esto resultó en que se buscara cómo poder hacer un uso útil de las múltiples referencias a \emptyset en árboles binarios.

Una de las opciones que se llegaron a usar se puede visualizar en la figura 12.6; cada referencia que en nuestra definición apunta a \emptyset se utiliza para apuntar a otro vértice en el árbol binario: excepto por dos referencias que quedan “colgando”.

Por qué se optó por esta configuración lo podremos entender mejor en el capítulo 14; nosotros nada más mencionamos que existe la posibilidad de programar los árboles binarios así, pero resulta en un código más complejo y menos elegante. Por lo mismo nosotros no la usaremos.

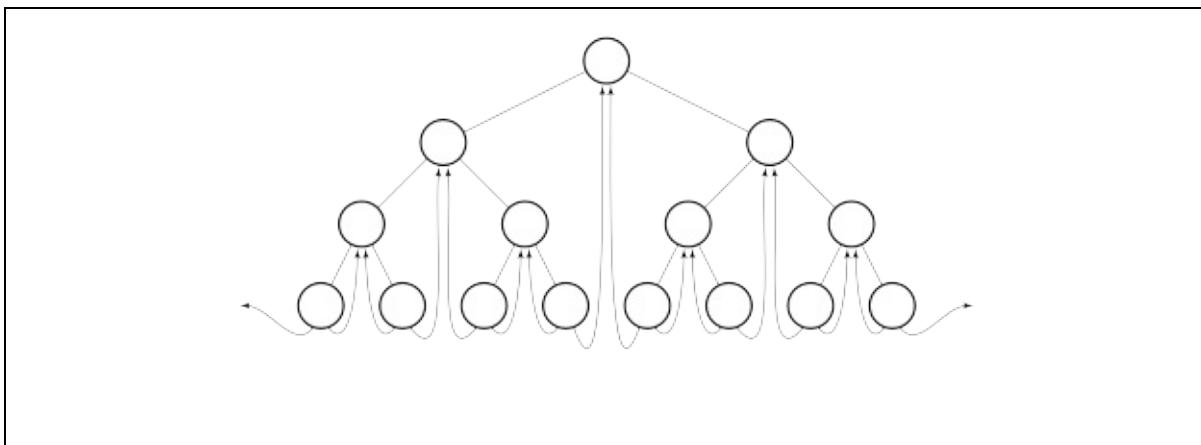


Figura 12.6: Árbol binario que no desperdicia las referencias de sus hojas.

Los árboles binarios son generalmente usados con elementos comparables; la definición e implementación vistos en este capítulo son resultado de querer aprovechar las características modernas del lenguaje de programación Java. En ese sentido, árboles binarios como los vimos aquí casi no aparecen en la literatura. Esto no quiere decir que no serán útiles; todo lo contrario, pero su utilidad se hará evidente cuando los extendamos a partir del capítulo 13.

Ejercicios

1. Implementa los métodos faltantes de la clase abstracta `ArbolBinario`.
2. Dado un árbol binario con n vértices; ¿cuál es su altura mínima?
3. Dado un árbol binario con n vértices; ¿cuál es el mayor número de hojas que puede tener?
4. Dado un árbol binario de altura h ; ¿cuál es el menor número de vértices que puede tener?
5. ¿Cuál es la diferencia entre la altura y la profundidad de un vértice?

13. Árboles binarios completos

Como vimos en el capítulo [12](#), los árboles binarios llenos son muy restrictivos. Para poder tener una estructura que se asemeje a ellos vamos a utilizar los árboles binarios completos, que como veremos son árboles binarios *casi* llenos.

13.1. Definición de árboles binarios completos

Los árboles binarios completos (*complete binary trees*) serán árboles binarios donde los hoyos que pudieran tener siempre estarán localizados en una zona.

Definición 13.1. (Árboles binarios completos) *Un árbol binario T será un árbol binario completo si y sólo todos los niveles de T están llenos excepto tal vez el último (el nivel $h(T)$). Además, si el vértice con coordenada $(j, h(T))$ está en el árbol, entonces el vértice con coordenada $(j - 1, h(T))$ también estará, para toda $0 < j < 2^{h(T)}$.*

Si el árbol binario T es completo todos los hoyos que puede tener siempre estarán en el último nivel y siempre todos juntos al extremo derecho del nivel (figura [13.1](#)). Debe ser claro que un árbol binario lleno, por definición, es completo; lo opuesto no es en general verdadero.

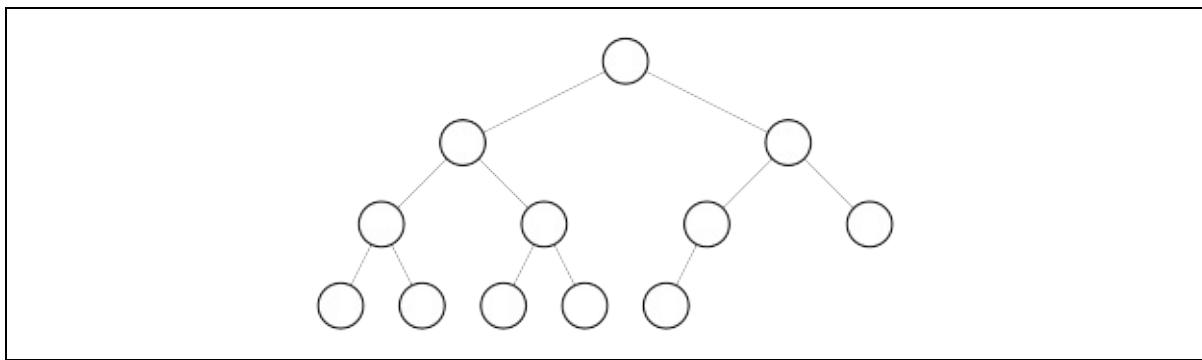


Figura 13.1: Árbol binario completo: todos los niveles están llenos excepto el último y en éste todos los hoyos están juntos al extremo derecho.

Los árboles binarios completos no nos serán de mucha utilidad realmente; pero son una manera sencilla de implementar concretamente la clase `ArbolBinario` y además son muy buenos para explicar los recorridos BFS para árboles binarios. Una versión mucho más natural y mucho más útil de árboles binarios completos la veremos en el capítulo [18](#).

Como hemos mencionado desde el capítulo [12](#) aprovecharemos la herencia dada por la Orientación a Objetos y por lo tanto la clase `ArbolBinarioCompleto`

extenderá a la clase abstracta `ArbolBinario`. Como será concreta deberemos implementar todo el comportamiento que la clase `ArbolBinario` no implementa; pero de cualquier forma el código pertinente a `ArbolBinarioCompleto` se verá sustancialmente reducido gracias a que la clase `ArbolBinario` ya habrá implementado gran parte.

Habrá un único método existente de `ArbolBinario` que la clase `ArbolBinarioCompleto` sobrecargará; como podemos calcular la altura del árbol sin recorrerlo, nada más a través del número de elementos que tiene, aprovecharemos esa característica del mismo para poder ejecutar el método más rápido. También tendremos un método `bfs()`, que explicaremos en la sección [13.2](#).

El esqueleto de la clase `ArbolBinarioCompleto` lo podemos ver en el listado [13.1](#).

```
public class ArbolBinarioCompleto<T>
    extends ArbolBinario<T> {

    private class Iterador implements Iterator<T> {
        private Cola<Vertice> cola;
        public Iterador() { /* ... */ }
        @Override public boolean hasNext() { /* ... */ }
        @Override public T next() { /* ... */ }
    }

    public ArbolBinarioCompleto() { /* ... */ }
    public ArbolBinarioCompleto(Coleccion<T> colección) {
        /* ... */
    }
    @Override public void agrega(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public int altura() { /* ... */ }
    @Override public Iterator<T> iterator() {
        return new Iterador();
    }

    public void bfs(AccionVerticeArbolBinario<T> accion) {
        /* ... */
    }
}
```

Listado 13.1: Esqueleto de la clase `ArbolBinarioCompleto`.

Como suele ser el caso, el método `iterator()` es tan sencillo que lo mostramos. Podemos observar que los iteradores para nuestros árboles binarios completos tienen una cola. Vamos a explicar eso en la siguiente sección.

13.2. Recorriendo árboles por amplitud

Existen dos principales formas de recorrer un árbol binario; por amplitud y por profundidad. *Cualquier* árbol binario se puede recorrer por ambas, pero en algunos tipos de árboles una forma suele tener más sentido que la otra.

En árboles binarios completos el recorrerlos por amplitud suele ser lo adecuado; es de la misma forma en que el árbol crece. Cuando agregamos un elemento a un árbol binario completo, siempre utilizamos el hueco más a la izquierda del último nivel, si no está lleno; o como el primer elemento de un nuevo nivel: así es como implementaremos el método `agrega()` de la clase `ArbolBinarioCompleto` más adelante. Para obtener el árbol de la figura 13.2 (suponiendo que no eliminamos ningún elemento) tendríamos que agregar los elementos en el orden *a, b, c, d, e, f, g, h, i, j, k, l, m, n*. Si recorremos el árbol por amplitud, los elementos del mismo serían visitados en ese mismo orden.

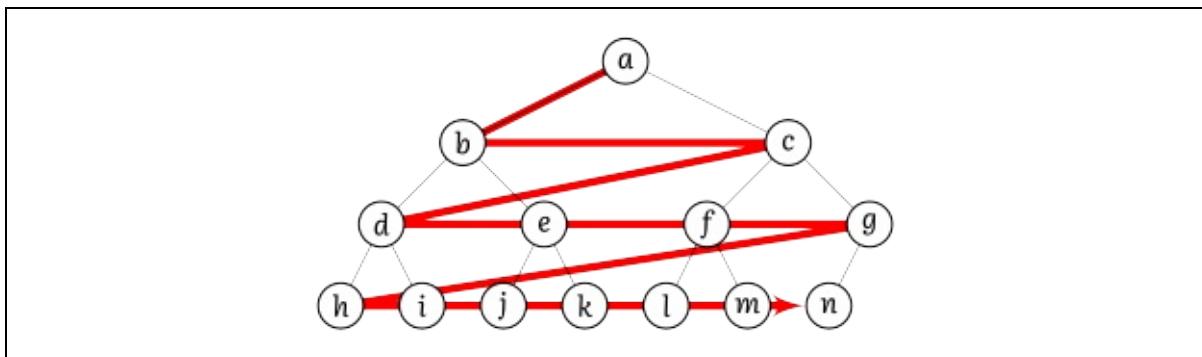


Figura 13.2: Recorriendo un árbol binario completo por BFS.

El recorrido por amplitud (BFS por sus siglas en inglés, *Breadth-First Search*), como su nombre indica, recorre el árbol primero por lo ancho en lugar de hacerlo por lo profundo. Para lograr esto lo que se hace es asegurar que cuando se visita un vértice *v*, los hijos de *v* serán visitados hasta después de que todos los vértices en el mismo nivel de *v* sean visitados.

Aunque se puede hacer un recorrido por amplitud en cualquier dirección, izquierda o derecha (en el sentido del orden en que se procesan los hijos de un vértice), lo común es hacerlo de izquierda a derecha. Para hacer un recorrido BFS vamos a poner la raíz del árbol en una cola y mientras la cola no sea vacía vamos a sacar el vértice *v* al frente de la misma, procesar ese elemento y después agregar los hijos izquierdo y el derecho a la cola, en ese orden.

Como en una cola el primer elemento en entrar es el primero en salir, esto quiere decir que todos los vértices de un nivel son procesados antes que cualquier vértice del siguiente; y como agregamos primero al hijo izquierdo y después al derecho, se garantiza que el nivel se recorre de izquierda a derecha, como en la figura 13.2.

Dado que BFS es el recorrido que más sentido tiene para árboles binarios completos, será el que los iteradores de la clase `ArbolBinarioCompleto`

implementarán; y es por eso que la clase interna `Iterador` tiene una cola como variable de clase. Además, la clase tendrá un método `bfs()` para hacer recorridos BFS *de los vértices*; para hacer esto de manera sencilla necesitamos acciones para los vértices, que explicaremos en la siguiente sección.

13.2.1. Acciones para vértices de árboles binarios

El recorrido BFS estará implementado en el método `bfs()`; este método recibe una instancia de la interfaz `AccionVerticeArbolBinario`, que es una interfaz funcional de la cual podemos ver su definición en el listado 13.2.

```
@FunctionalInterface
public interface AccionVerticeArbolBinario<T> {
    public void actua(VerticeArbolBinario<T> vertice);
}
```

Listado 13.2: La interfaz `AccionVerticeArbolBinario`.

La idea por supuesto es utilizar lambdas con este método; por ejemplo, para imprimir todos los elementos del árbol en orden BFS, podríamos hacer lo que se muestra en el listado 13.3.

```
ArbolBinarioCompleto<Integer> arbol =
    new ArbolBinarioCompleto<Integer>();
// Llenamos el árbol...
arbol.bfs((v) -> System.out.printf("%n, ", v.get()));
System.out.println();
```

Listado 13.3: Imprimiendo los elementos de un árbol binario completo en BFS.

Por supuesto, dado que nuestros iteradores iterarán el árbol en el orden BFS, el mismo resultado lo podríamos obtener con el código en el listado 13.4.

```
for (Integer n : arbol)
    System.out.printf("%n, ", n);
System.out.println();
```

Listado 13.4: Imprimiendo los elementos de un árbol binario completo con sus iteradores.

La diferencia radica en que el método `bfs()` tiene acceso a los vértices del árbol binario completo, mientras que los iteradores únicamente tienen acceso a los *elementos*. La diferencia es sutil, pero importante.

Como mencionamos en el capítulo 3, esta manera de recorrer los árboles es una alternativa a los iteradores; teniendo un comportamiento que recibe una

acción a aplicar a cada elemento de la colección. Algunos lenguajes orientados a objetos es la que siguen, en particular SmallTalk; pero en general los lenguajes de programación orientados a objetos, especialmente los modernos, han optado por utilizar iteradores para poder recorrer colecciones.

El método `bfs()` se ofrece por completez y conveniencia; la interfaz `VerticeArbolBinario` es lo suficientemente flexible como para permitírnoslo. Lo podemos ver en el listado 13.5 para imprimir los elementos del árbol en orden BFS.

```
Cola<VerticeArbolBinario<Integer>> cola;
cola = new Cola<VerticeArbolBinario<Integer>>();
cola.mete(arbol.raiz());
while (!cola.esVacia()) {
    VerticeArbolBinario<Integer> v = cola.saca();
    System.out.printf("%n ", v.get());
    if (v.hayIzquierdo())
        cola.mete(v.izquierdo());
    if (v.hayDerecho())
        cola.mete(v.derecho());
}
System.out.println();
```

Listado 13.5: Imprimiendo los elementos de un árbol binario completo en orden BFS.

Con esto ya podemos ver los algoritmos para nuestros árboles binarios completos.

13.3. Algoritmos para árboles binarios completos

Como hemos hecho hasta ahora, revisaremos los algoritmos de nuestra estructura en el orden especificado en el listado 13.1, así que comenzaremos con la clase interna `Iterador`

- `Iterador()`

El constructor; lo único que hace es crear la cola y agregarle la raíz si no es \emptyset .

- `hasNext()`

Regresa verdadero si la cola no es \emptyset , falso en otro caso.

- `next()`

Si la cola es \emptyset ocurre un error; si no saca el siguiente vértice. Si los hijos del vértice son distintos de \emptyset los agrega a la cola en orden izquierdo–

derecho. Por último regresa el elemento del vértice.

Veamos los algoritmos de la clase misma por método. Contrario a la clase **Lista** y sus nodos, no entraremos en los detalles de cómo manejar las referencias entre vértices; debe quedar claro que al “conectar” dos vértices a y b , esto implica que a define su izquierdo (o derecho) como b y b define a su padre como a (o viceversa).

- **ArbolBinarioCompleto()** y **ArbolBinarioCompleto(Colección)**

Sencillamente encadena (manda llamar) el constructor correspondiente de la clase padre. Recordemos que en Java esto se hace con **super()**.

- **agrega()**

El algoritmo para agregar en un árbol binario completo se puede implementar con una complejidad en tiempo de $O(\log n)$. Esto es relativamente complicado y por lo tanto permitiremos el implementar el algoritmo en tiempo lineal; pero explicaremos las dos versiones.

1. Para implementar el algoritmo de agregación en tiempo $O(n)$ creamos un nuevo vértice con el elemento a agregar e incrementamos nuestro contador de elementos. Después verificamos si la raíz es \emptyset , en cuyo caso el nuevo vértice es la raíz y terminamos.

Si la raíz es distinta de \emptyset recorremos los vértices usando BFS hasta encontrar un vértice que no tenga hijo izquierdo o derecho (*en ese orden*); para el primer vértice que ocurra esto (que es buscar el primer hoyo del árbol), agregamos ahí el nuevo vértice y terminamos.

2. Para implementar el algoritmo en $O(\log n)$ en tiempo, calculamos en tiempo logarítmico la coordenada del nuevo vértice utilizando nada más el número de elementos y la altura del árbol. Con esto podemos reconstruir la ruta de la raíz al padre del nuevo vértice en tiempo $O(\log n)$; si la coordenada x del padre es impar entonces es un hijo derecho, si no es izquierdo. Haciendo reconstruido la ruta la seguimos y agregamos el nuevo elemento en su lugar.

El algoritmo no sólo se puede implementar en tiempo $O(\log n)$; se puede hacer esto sin utilizar los métodos `power()` o `log()` de la clase **Math** y sin realizar multiplicaciones ni divisiones. Lo cual es un ejercicio interesante.

- **elimina()**

Hacemos notar que los árboles binarios completos no están ordenados; la única característica que mantienen es su estructura completa: todos sus niveles son llenos o sólo tienen hoyos en el extremo derecho del último nivel.

Por lo tanto al eliminar elementos lo único que cuidaremos será mantener el árbol como completo. El algoritmo buscará el vértice del elemento a eliminar (podemos reutilizar el método `busca()` de la clase `ArbolBinario`); si no se encuentra en el árbol terminaremos.

Si sí está decrementaremos nuestro contador; si éste llega a cero, hacemos la raíz \emptyset y terminamos. Si no buscaremos el último vértice del árbol; por definición este elemento será el último que queda en la cola al recorrer el árbol por BFS. Intercambiaremos los elementos de estos dos vértices, de tal forma que el elemento a eliminar quede en el último vértice, el cual procederemos a eliminar.

Como el último vértice por definición no tiene hijos, eliminarlo consiste en eliminar la referencia que tiene su padre a él mismo; hay que verificar si es izquierdo o derecho para poder hacer esto.

El algoritmo es $O(n)$ en tiempo y no hay forma de mejorarlo; hay que buscar el elemento a eliminar y esto siempre será $O(n)$ en tiempo porque hay que recorrer los elementos del árbol uno por uno. Podríamos encontrar el último elemento en $O(\log n)$; pero dado que la búsqueda del elemento a eliminar es lineal, no tiene mucho sentido tomarnos la molestia.

- `altura()`

La altura de un árbol binario completo siempre es $\lfloor \log_2 n \rfloor$, donde n es el número de elementos en el árbol. Podemos usar el método `log()` de la clase `Math`, pero se puede dividir la altura entre 2 hasta llegar a 1. También se puede hacer lo mismo sin usar la división de enteros; se deja como ejercicio para el lector.

- `bfs()`

Si el árbol es vacío terminamos. Si no, creamos una cola y agregamos la raíz del árbol a la misma; como el árbol no es vacío la raíz tampoco lo será.

Mientras la cola no sea vacía sacamos el vértice al frente de la misma, le aplicamos la acción que recibe el método y si sus hijos son distintos del vacío, los agregamos a la cola.

Con esto terminamos la clase `ArbolBinarioCompleto`. Aunque relativamente

sencilla, los árboles binarios completos nos sirven para ejemplificar los recorridos por BFS y son un buen primer ejemplo para escribir una clase concreta que extienda a `ArbolBinario`. Sin embargo, en general *nadie* implementa árboles binarios completos utilizando vértices y referencias (o apuntadores) a los mismo; como dijimos arriba, los árboles binarios completos se implementan como lo veremos en el capítulo [18](#). En ese sentido y al igual que en el capítulo [12](#), los árboles binarios completos como los cubrimos en este capítulo casi no aparecen en la literatura.

Los siguientes árboles binarios que veremos serán en cambio muy comunes en la literatura. Sus algoritmos también serán más complicados y por lo tanto las estructuras mismas más interesantes.

Ejercicios

1. Implementa los métodos faltantes de la clase `ArbolBinarioCompleto`.
2. ¿En qué difieren los árboles binarios llenos y los completos?
3. ¿Cuál es la altura de un árbol binario completo con n elementos?
4. Dado un árbol binario completo con n elementos, ¿cuál es la coordenada de su último elemento?

14. Árboles binarios ordenados

El uso más importante de árboles binarios es posible que sea como árboles binarios ordenados o árboles binarios de búsqueda (BST por sus siglas en inglés: *binary search tree*). Exceptuando los árboles binarios completos del capítulo 13, todos los árboles concretos que veremos en el libro serán árboles binarios ordenados.

La primera referencia a árboles binarios de búsqueda la dio Sandy Douglas en [12] en 1959; sin embargo, con casi toda certeza la idea se le ocurrió de manera simultánea a varios computólogos. Como veremos en un momento, los árboles binarios ordenados se comportan de manera muy similar al algoritmo QUICKSORT.

Para poder definir los árboles binarios ordenados, debemos primero definir lo que es un subárbol de un árbol.

Definicion 14.1. (Subárboles) *Sea T un árbol binario y u un vértice de T . El subárbol $T(u)$ de T inducido por u es el conjunto de todos los vértices v de T tales que existe una secuencia de vértices $u = v_1, \dots, v_k = v$ tales que v_i es padre de v_{i+1} , $i = 1, \dots, k-1$, junto con el vértice u .*

Si $u = \emptyset$ entonces $T(u)$ es un árbol vacío.

En otras palabras es como si tomáramos a u como “raíz” de un árbol, aunque su padre no sea \emptyset . Exceptuando eso, un subárbol tiene las mismas características de un árbol.

Para hacer la notación más simple también definiremos lo que son los subárboles izquierdo y derecho de un vértice.

Definicion 14.2. (Subárboles izquierdo y derecho) *Sea T un árbol binario y v un vértice de T . Sean v_i y v_d los vértices izquierdo y derecho de v respectivamente: los subárboles izquierdo y derecho de v , denotados por $T_i(v)$ y $T_d(v)$, serán $T(v_i)$ y $T(v_d)$ respectivamente.*

Como su nombre indica, los árboles binarios ordenados están ordenados, lo cual inmediatamente implica que los elementos dentro de los vértices del mismo son comparables.

Para no tener que estarnos refiriendo todo el tiempo a los elementos de los vértices, dados dos vértices u y v diremos que:

- $u < v$ si y sólo si el elemento de u es menor que el elemento de v ,
- $u > v$ si y sólo si el elemento de u es mayor que el elemento de v ; y
- $u = v$ si y sólo si el elemento de u es igual que el elemento de v .

Con estas definiciones auxiliares podemos proceder a definir árboles binarios ordenados.

14.1. Definición de árboles binarios ordenados

Definición 14.3. (Árboles binarios ordenados) *Sea T un árbol binario. T será un árbol binario ordenado si y sólo si para todo vértice u de $T_i(v)$ y para todo vértice w de $T_d(v)$ se cumple que $u \leq v$ y $v \leq w$.*

Es importante que las desigualdades no sean estrictas por una operación de árboles binarios ordenados que veremos más adelante. La propiedad definitoria de los árboles binarios ordenados será una invariante de la estructura de datos; un árbol binario ordenado *siempre* estará ordenado. Un ejemplo se puede ver en la figura 14.1.

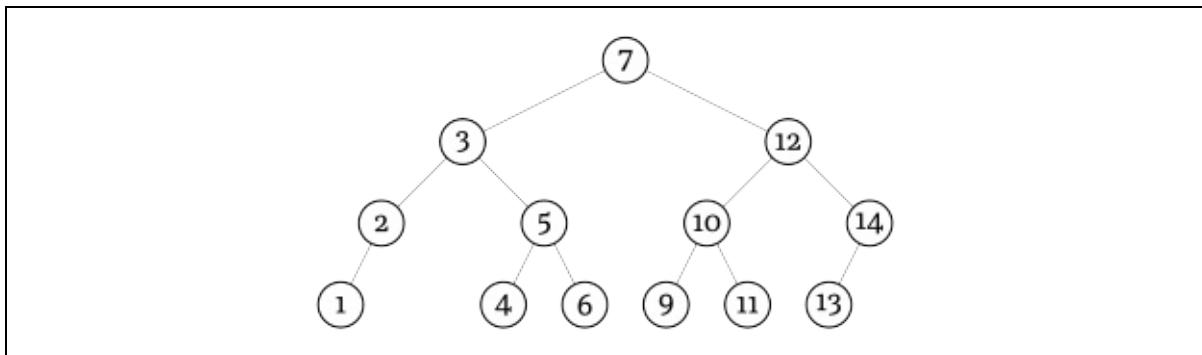


Figura 14.1: Árbol binario ordenado: todo vértice es mayor o igual que los vértices de su subárbol izquierdo y menor o igual que los vértices de su subárbol derecho.

Al igual que con la clase `ArbolBinarioCompleto`, la clase `ArbolBinarioOrdenado` extenderá `ArbolBinario` y por lo tanto deberá implementar los métodos que ésta no concreta; una vez más esto incluirá los métodos `agrega()` y `elimina()`, así como los iteradores para árboles binarios ordenados.

Contrario a `ArbolBinarioCompleto`, en `ArbolBinarioOrdenado` no sobrecargaremos `altura()`; pero sí vamos a sobrecargar `busca()` porque podemos hacer que se ejecute en $O(\log n)$ en promedio, aunque el peor de los casos continuará siendo $O(n)$.

La clase `ArbolBinarioOrdenado` está pensada para ser extendida: los últimos dos árboles binarios que veremos en los capítulos 15 y 16 heredarán de `ArbolBinarioOrdenado`.

El algoritmo para eliminar vértices en árboles binarios ordenados puede diseñarse utilizando dos algoritmos auxiliares, en cuyo caso podemos reutilizarlos en los algoritmos de eliminación de las clases que extiendan **ArbolBinarioOrdenado**. Es por esto que tendremos los métodos protegidos `intercambiaEliminable()` y `eliminaVertice()`, donde implementaremos estos algoritmos auxiliares.

De manera similar el algoritmo para agregar elementos puede reutilizarse tal cual en las clases que extienden; pero después debemos tener disponible el vértice del elemento agregado. Por esto tendremos como variable de clase un vértice llamado `ultimoAgregado`, en el que guardaremos el vértice del último elemento que agreguemos; pero el valor de esta variable sólo estará garantizado inmediatamente después de haber agregado un elemento.

El tener disponible el vértice del último elemento agregado es en general bastante útil, entonces también tendremos un método `getUltimoVerticeAgregado()`, pero tampoco podremos garantizar que el vértice que nos regrese sea el último agregado, a menos que sea llamado *inmediatamente después* de haber agregado un vértice.

Adicionalmente tendremos métodos para girar al árbol sobre un vértice dado, `giraDerecha()` y `giraIzquierda()`, que son operaciones muy importantes de los árboles ordenados, particularmente para las clases que extiendan a **ArbolBinarioOrdenado**.

Una vez más el método `iterator()` es tan sencillo que lo mostraremos ya implementado. Los iteradores para nuestros árboles binarios ordenados tendrán una pila, en lugar de una cola como en **ArbolBinarioCompleto**; explicaremos eso en la siguiente sección, donde veremos los recorridos por profundidad. Ahí también veremos los algoritmos correspondientes a los métodos `dfsPreOrder()`, `dfsInOrder()` y `dfsPostOrder()`, donde implementaremos las tres versiones distintas de recorridos por profundidad más comunes.

Podemos ver el esqueleto de la clase **ArbolBinarioOrdenado** en el listado 14.1.

```
public class ArbolBinarioOrdenado<T extends Comparable<T>>
    extends ArbolBinario<T> {
    private class Iterador implements Iterator<T> {
        private Pila<Vertice> pila;
        public Iterador() { /* ... */ }
        @Override public boolean hasNext() { /* ... */ }
        @Override public T next() { /* ... */ }
    }
    protected Vertice ultimoAgregado;
    public ArbolBinarioOrdenado() { super(); }
    public ArbolBinarioOrdenado(Coleccion<T> colección) {
```

```

        /* ... */
    }

    @Override public void agrega(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public
    VerticeArbolBinario<T> busca(T elemento) { /* ... */ }
    @Override public Iterator<T> iterator() {
        return new Iterador();
    }

    protected Vertice
    intercambiaEliminable(Vertice vertice) { /* ... */ }

    protected void
    eliminaVertice(Vertice vertice) { /* ... */ }

    public VerticeArbolBinario<T>
    getUltimoVerticeAgregado() { /* ... */ }

    public void
    giraDerecha(VerticeArbolBinario<T> vertice) {
        /* ... */
    }

    public void
    giraIzquierda(VerticeArbolBinario<T> vertice) {
        /* ... */
    }

    public void
    dfsPreOrden(AccionVerticeArbolBinario<T> accion) {
        /* ... */
    }

    public void
    dfsInOrden(AccionVerticeArbolBinario<T> accion) {
        /* ... */
    }

    public void
    dfsPostOrden(AccionVerticeArbolBinario<T> accion) {
        /* ... */
    }
}

```

Listado 14.1: Esqueleto de la clase **ArbolBinarioOrdenado**.

14.2. Recorriendo árboles por profundidad

En el capítulo 13 vimos los recorridos de árboles binarios por amplitud; aquí revisaremos los recorridos por profundidad (o DFS, por sus siglas en inglés, *Depth-First Search*).

Como su nombre indica la idea de los recorridos por profundidad es procesar primero los descendientes de un vértice antes que sus hermanos, lo que resulta en que el recorrido siempre procesa el primer vértice del último nivel del

árbol antes de procesar el hijo derecho de la raíz. Esto es por supuesto si procedemos de izquierda a derecha; igual que con los recorridos por amplitud podemos proceder de derecha a izquierda, pero esto casi nunca se hace.

De manera análoga a BFS, DFS puede utilizar una pila para esto, agregando los descendientes del vértice al ser procesado. Es lo que nosotros utilizaremos en nuestros iteradores y la razón de tener una pila de vértices en la clase **Iterador**.

Sin embargo también podemos utilizar implícitamente la pila de ejecución, en lugar de usar una explícita, al hacer el recorrido recursivamente. Debe ser claro que al recorrer un árbol binario recursivamente el algoritmo recursivo recibe como parámetro un vértice (comenzando siempre por la raíz) donde la cláusula de escape será que el vértice sea vacío. El algoritmo hará recursión primero sobre el hijo izquierdo y después sobre el derecho (porque procedemos de izquierda a derecha); pero al momento de procesar el vértice actual, el que recibimos como parámetro, tendremos la opción de hacerlo antes de las llamadas recursivas, al final de las mismas o en medio de ambas.

Esto resulta en los recorridos DFS pre-orden, post-orden y en-orden; pero como ya utilizamos las siglas en inglés de los recorridos en profundidad, usaremos los términos en inglés: *pre-order*, *post-order* e *in-order*. Tendremos un método para cada uno de los tres.

Los tres recorridos DFS son muy parecidos al momento de implementarlos, pero el orden en que recorren los vértices del árbol son completamente distintos. Vamos a ver a detalle los tres algoritmos, porque son muy sencillos de implementar recursivamente.

14.2.1. DFS *pre-order*

Como mencionábamos arriba, en el recorrido DFS *pre-order* procesamos el vértice actual antes de hacer las llamadas recursivas. Por ejemplo, para imprimir los elementos de un árbol recorriéndolos en orden DFS *pre-order*, usaríamos el algoritmo [14.1](#).

```
procedure IMPRIMEDFSPREORDER(v)
    ▷ Vértice v
    if (v = ø)
        return
    IMPRIME(GETELEMENTO(v))
    IMPRIMEDFSPREORDER(IZQUIERDO(v))
    IMPRIMEDFSPREORDER(DERECHO(v))
```

Algoritmo 14.1: Algoritmo para imprimir los elementos del árbol en orden DFS *pre-order*.

Obviamente el algoritmo debe ser ejecutado con la raíz del árbol inicialmente. Como se procesa el vértice actual antes de hacer la recursión, la raíz de cada subárbol se imprime (o en general procesa) antes que todo el resto del subárbol.

El orden en que es recorrido el árbol puede verse en la figura [14.2](#).

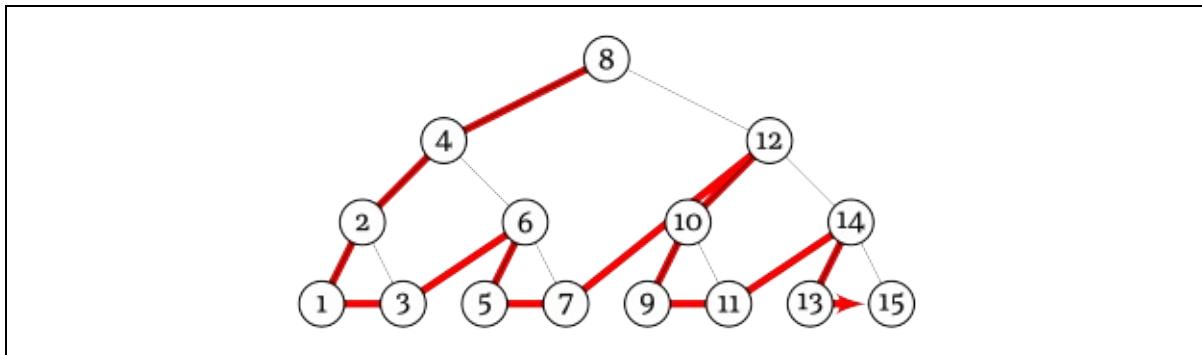


Figura 14.2: Recorriendo un árbol binario ordenado en orden DFS *pre-order*. Los vértices se recorren en el orden 8, 4, 2, 1, 3, 6, 5, 7, 12, 10, 9, 11, 14, 13, 15.

14.2.2. DFS *post-order*

En el recorrido DFS *post-order* hacemos las llamadas recursivas *antes* de procesar el vértice actual; esto podría parecer como el inverso de DFS *pre-order* y en un sentido estructural así es: el recorrido *post-order* es la reflexión vertical del recorrido *pre-order*, pero esto no es equivalente a sencillamente invertir el orden de un recorrido para obtener el otro. Podemos ver en el algoritmo [14.2](#) el pseudocódigo para recorrer un árbol usando DFS *post-order*.

```
procedure IMPRIMEDFSPOSTORDER(v)
    ▷ Vértice v
    if (v = ø)
        return
    IMPRIMEDFSPOSTORDER(IZQUIERDO(v))
    IMPRIMEDFSPOSTORDER(DERECHO(v))
    IMPRIME(GETELEMENTO(v))
```

Algoritmo 14.2: Algoritmo para imprimir los elementos del árbol en orden DFS *post-order*.

Como el vértice actual es procesado *después* de las llamadas recursivas, esto resulta en que la raíz de cada subárbol siempre es lo último en imprimirse (o en general procesarse) del mismo.

El orden en que es recorrido el árbol puede verse en la figura [14.3](#).

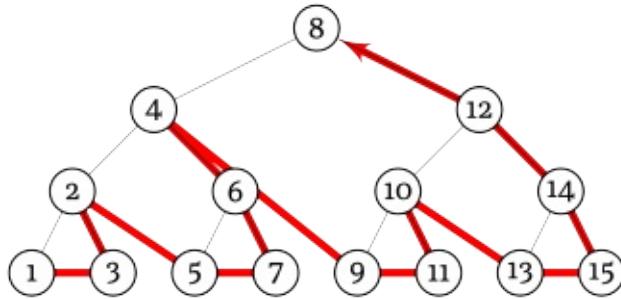


Figura 14.3: Recorriendo un árbol binario ordenado por DFS *post-order*. Los vértices se recorren en el orden 1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8.

14.2.3. DFS *in-order*

Al recorrer un árbol binario usando DFS *in-order*, el vértice actual es procesado entre las llamadas recursivas. Como un árbol binario ordenado es tal que, por definición, un vértice es mayor o igual que todos los vértices en su subárbol izquierdo y menor o igual que todos los vértices en su subárbol derecho, el recorrido *in-order* resulta en que los elementos son visitados (como el nombre del recorrido implica) *en orden*.

El pseudocódigo en el algoritmo [14.3](#) muestra cómo recorrer un árbol binario usando DFS *in-order* para imprimir sus elementos.

```

procedure IMPRIMEDFSINORDER(v)
    > Vértice v
    if (v =  $\emptyset$ )
        return
    IMPRIMEDFSINORDER(IZQUIERDO(v))
    IMPRIME(GETELEMENTO(v))
    IMPRIMEDFSINORDER(DERECHO(v))

```

Algoritmo 14.3: Algoritmo para imprimir los elementos del árbol en orden DFS *in-order*.

El equivalente iterativo del algoritmo [14.3](#) es el que utilizaremos en la clase **Iterator** dentro de **ArbolBinarioOrdenado**; pero al igual que con los árboles binarios completos, el iterador recorre los *elementos* del árbol, mientras que el método `dfsInOrder()` recorrerá *vértices*. La diferencia es sutil, pero importante.

Podemos ver cómo se recorre un árbol con DFS *in-order* en la figura [14.4](#). Esta figura también explica el por qué se reutilizan referencias “colgantes” como mostramos en la figura [12.6](#) del capítulo [12](#); si el árbol binario es ordenado, cada vértice hoja puede acceder a su siguiente vértice (en orden) en tiempo $O(1)$.

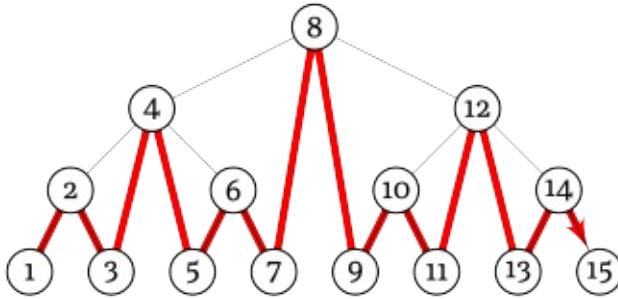


Figura 14.4: Recorriendo un árbol binario ordenado por DFS *in-order*. Los vértices se recorren en orden.

14.3. Algoritmos para árboles binarios ordenados

Una vez más revisaremos los algoritmos de nuestra estructura de datos en el orden dado por el listado [14.1](#), comenzando con la clase interna `Iterador`.

- `Iterador()`

El constructor; además de crear la pila que utilizaremos, vamos a agregarla *toda* la primer rama izquierda del árbol (la raíz, su hijo izquierdo, el hijo izquierdo del hijo izquierdo, etcétera). Esto es por cómo funciona DFS *in-order* recursivo (y que queremos replicar aquí): los vértices izquierdos se apilan en la pila de ejecución hasta llegar al primer vértice que no tenga hijo izquierdo (figura [14.4](#)). De esta manera el método

`next()` siempre regresará el elemento del vértice en el tope de la pila.

- `hasNext()`

Sencillamente comprobamos que la pila no sea vacía.

- `next()`

Si la pila es vacía ocurre un error; en nuestra implementación con Java ni siquiera tenemos que comprobar este error: sencillamente invocamos el método `saca()` de nuestra pila. Si la pila es vacía, el método lanzará la excepción `NoSuchElementException`, que es la que el método `next()` debe lanzar cuando no hay siguiente elemento.

Sacamos el siguiente vértice de la pila, guardamos su elemento y vemos si tiene hijo derecho. Si lo tiene, lo agregamos a la pila así como a toda su primer rama izquierda.

Esto, junto con el constructor del iterador, nos permite replicar el comportamiento del método recursivo `dfsInOrder()`.

Continuamos con los métodos de la clase `ArbolBinarioOrdenado` propiamente; al igual que con la clase `ArbolBinarioCompleto`, ya no entraremos en los detalles de cómo manejar las referencias entre vértices; debe quedar claro que al “conectar” dos vértices a y b , esto implica que a define su izquierdo (o derecho) como b y b define a su padre como a (o viceversa).

- `ArbolBinarioOrdenado()` y `ArbolBinarioOrdenado(Coleccion<T>)`.

Sencillamente encadenamos (mandamos llamar) el constructor correspondiente de la clase padre.

- `agrega()`

Creamos un nuevo vértice con el elemento a agregar e incrementamos nuestro contador de elementos.

Si la raíz es \emptyset , la actualizamos al nuevo vértice y terminamos. Si no, invocamos con la raíz un algoritmo auxiliar recursivo que recibe un vértice actual distinto de \emptyset y el nuevo vértice.

En el algoritmo auxiliar comparamos el elemento del vértice actual con el elemento del nuevo vértice; uno de los siguientes casos se sigue:

1. El elemento del nuevo vértice es menor o igual que el elemento del vértice actual. Si el izquierdo del vértice actual es \emptyset , hacemos al nuevo vértice el izquierdo del vértice actual y terminamos. Si no, hacemos recursión sobre el izquierdo del vértice actual.
2. El elemento del nuevo vértice es mayor que el elemento del vértice actual. Si el derecho del vértice actual es \emptyset , hacemos al nuevo vértice el derecho del vértice actual y terminamos. Si no, hacemos recursión sobre el derecho del vértice actual.

- `elimina()`

Si el vértice del elemento que vamos a eliminar es una hoja, es sencillo eliminarlo: sólo lo desconectamos de su padre (figure 14.5).

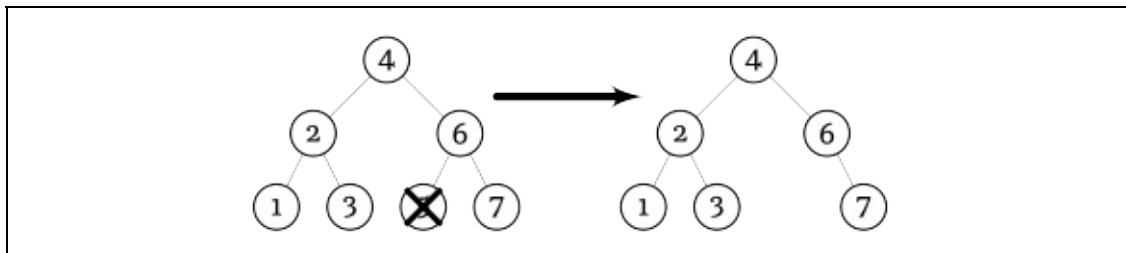


Figura 14.5: Eliminamos la hoja con el elemento 5; esto consiste en desconectarla de su padre.

Si el vértice del elemento que vamos a eliminar tiene un único hijo, también es fácil eliminarlo; “subimos” su único hijo para que lo

reemplace (figure 14.6).

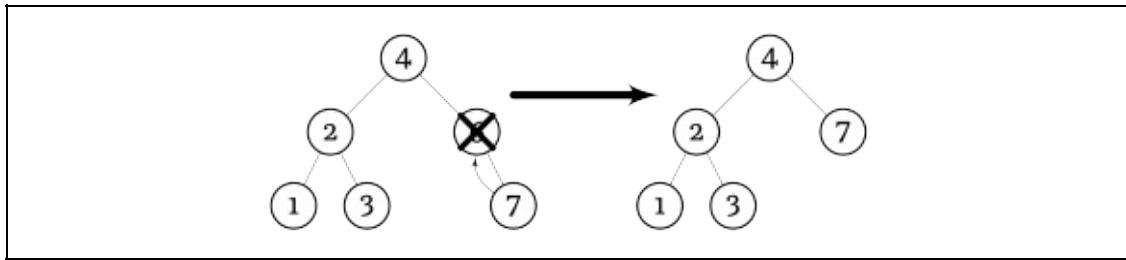


Figura 14.6: Eliminamos el vértice que tiene el elemento 6; como tiene un único hijo esto consiste en “subir” su al mismo para que lo reemplace.

Si el vértice del elemento que vamos a eliminar es tal que sus dos hijos son distintos de \emptyset , entonces es más complicado eliminarlo. No podemos eliminar el vértice directamente; por lo tanto vamos a buscar un segundo vértice que podamos eliminar (o sea, que tenga a lo más un hijo) e intercambiaremos los elementos de los vértices. Debemos hacer este intercambio y que el árbol binario continúe siendo ordenado al eliminar el vértice: que todo vértice siga siendo mayor o igual que los vértices de su subárbol izquierdo y menor o igual que los vértices de su subárbol derecho.

Como v tiene dos hijos, entonces $T_i(v)$ (su subárbol izquierdo) es distinto de \emptyset . Vamos a buscar en $T_i(v)$ el primer vértice cuyo hijo derecho sea \emptyset ; podemos ver la versión recursiva de esto en el algoritmo 14.4.

```

procedure MÁXIMOEnSUBÁRBOL( $v$ )
   $\triangleright$  Vértice  $v$ 
  if (DERECHO( $v$ )= $\emptyset$ )
    return  $v$ 
  return MÁXIMOEnSUBÁRBOL(DERECHO( $v$ ))

```

Algoritmo 14.4: Algoritmo recursivo para encontrar el máximo en un subárbol no vacío.

Sea u el vértice que $\text{MÁXIMOEnSUBÁRBOL}(\text{DERECHO}(v))$ nos regresa. Por definición u es menor o igual que v ; además (y aunque no lo demostraremos formalmente) es mayor o igual que todos los vértices de $T_i(v)$: para todo vértice w en $T_i(v)$ se cumple que u está en $T_d(w)$ o existe un vértice w' tal que w está en $T_i(w')$ y u está en $T_d(w')$.

Por esto es que el algoritmo 14.4 se llama MÁXIMOEnSUBÁRBOL : nos regresa un elemento que es mayor o igual que todos los elementos del subárbol definido por el vértice que recibe como parámetro.

Como u es un elemento maximal en $T_i(v)$, si intercambiamos los elementos de u y v el árbol binario continuará siendo ordenado cuando

eliminemos u : el elemento que originalmente estaba en u es mayor o igual que todos los elementos de $T_i(v)$ y por definición es menor o igual que todos los elementos de $T_d(v)$; por lo tanto puede tomar el lugar del elemento originalmente en v sin romper el orden de T . Además, como u no tiene hijo derecho (así lo buscamos explícitamente en el algoritmo 14.4), entonces tiene a lo más un hijo y podemos eliminarlo sin mayores complicaciones (figura 14.7).

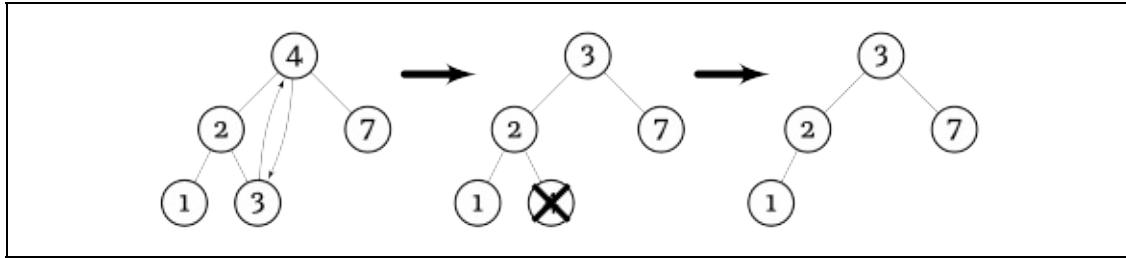


Figura 14.7: Para eliminar el elemento 4 del árbol primero buscamos el vértice máximo de su subárbol izquierdo, que es el vértice que contiene a 3. Intercambiamos los elementos y como el vértice que era máximo no tiene hijo derecho podemos eliminarlo fácilmente.

Como mencionábamos arriba el algoritmo para eliminar un elemento utilizará dos algoritmos auxiliares; el primero de ellos buscará el máximo en el subárbol izquierdo y hará el intercambio de los elementos. Este algoritmo auxiliar sólo será utilizado si el vértice a eliminar tiene sus dos hijos distintos de \emptyset . El otro algoritmo auxiliar eliminará un vértice que tenga a lo más un hijo.

Por lo tanto el algoritmo para eliminar consiste en buscar el vértice que contenga el elemento a eliminar; si el elemento no está en el árbol, terminamos. Para hacer esto usamos el algoritmo del método `busca()`.

Si el elemento sí está en el árbol decrementamos nuestro contador de elementos y verificamos si el vértice que lo contiene tiene sus hijos distintos de \emptyset . Si así es usamos nuestro primer algoritmo auxiliar y obtenemos el vértice a eliminar que tenga a lo más un hijo. Si no, el vértice original ya tiene a lo más un hijo.

Por último usamos nuestro segundo algoritmo auxiliar para eliminar el vértice que tiene a lo más un hijo y terminamos.

- `busca()`

Vamos a sobrecargar el método `busca()` de `ArbolBinario` porque podemos mejorar el algoritmo explotando el hecho de que el árbol está ordenado.

El algoritmo será recursivo sobre un vértice v y comenzaremos (como suele ser el caso) con la raíz. La cláusula de escape es que $v = \emptyset$ en cuyo caso regresamos \emptyset .

Si $v \neq \emptyset$ comparamos el elemento a buscar con el elemento de v ; si son iguales regresamos v . Si el elemento a buscar es menor, regresamos el resultado de hacer recursión sobre el izquierdo de v ; si no regresamos el resultado de hacer recursión sobre el derecho de v .

Que algoritmo es correcto es muy fácil de demostrar por inducción; se les deja de ejercicio.

- `intercambiaEliminable()`

Este algoritmo auxiliar recibe un vértice con dos hijos distintos de \emptyset y regresa un vértice con un único hijo con el que se habrá intercambiado el elemento del primero.

El algoritmo utiliza a su vez el algoritmo [14.4](#) para obtener el vértice máximo del subárbol izquierdo del vértice, intercambia los elementos de los dos vértices y regresa el vértice maximal para que sea eliminado.

- `eliminaVertice()`

Este algoritmo auxiliar elimina un vértice que tiene a lo más un hijo; pero no sabemos qué hijo será.

Sea v el vértice a eliminar y u su hijo distinto de \emptyset , si existe (es posible que *los dos* hijos de v sean \emptyset). Sea p el vértice padre de v (que también puede ser \emptyset).

Si $p \neq \emptyset$ determinamos si v es izquierdo o derecho; sin pérdida de generalidad supongamos lo primero. Hacemos que el izquierdo de p sea u . Si $p = \emptyset$ hacemos que la raíz del árbol sea u .

Si $u \neq \emptyset$ hacemos que el padre de u sea p y terminamos.

- `getUltimoVerticeAgregado()`

Esta operación sólo está definida inmediatamente después de que se haya llamado el método `agrega()`; la vamos a necesitar en algunos de los algoritmos de las clases que extienden `ArbolBinarioOrdenado`, porque no podemos modificar la firma de `agrega()` para que nos regrese el vértice del elemento agregado, ya que es una operación general de colecciones y no particular de árboles binarios ordenados.

Si el árbol es modificado de cualquier manera después de llamar el método `agrega()`, el comportamiento de este método no está definido; en otras palabras, en ese caso no podremos garantizar que regrese el último método agregado.

La implementación es sencillamente regresar el último vértice agregado.

- `giraDerecha()` Y `giraIzquierda()`

Las operaciones de giro son muy importantes en los árboles binarios ordenados, entre otras cosas porque nos permiten cambiar la altura del árbol preservando el orden de los mismos.

Los algoritmos correspondientes son relativamente sencillos y se pueden ver gráficamente en la figura [14.8](#).

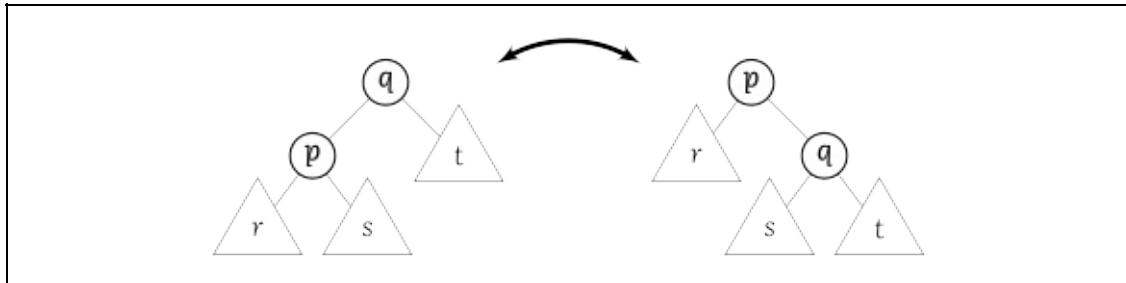


Figura 14.8: Giro sobre el vértice q a la derecha y sobre el vértice p a la izquierda.

Para que un vértice q pueda girar a la derecha éste tiene que tener un hijo izquierdo p ; e inversamente para que un vértice p pueda girar a la izquierda debe tener un hijo derecho q . En la figura [14.8](#) r , s y t representan subárboles; cada uno de ellos puede ser \emptyset .

Si giramos q a la derecha y por lo tanto su hijo izquierdo p existe, entonces p se vuelve el padre de q y q el hijo derecho de p . El hijo izquierdo de p (si existe) continúa siéndolo; el hijo derecho de q (si existe) continúa siéndolo; y el hijo derecho de p (si existe) se convierte en el hijo izquierdo de q .

Si giramos p a la izquierda y por lo tanto su hijo derecho q existe, entonces q se vuelve el padre de p y p el hijo izquierdo de q . El hijo izquierdo de p (si existe) continúa siéndolo; el hijo derecho de q (si existe) continúa siéndolo; y el hijo izquierdo de q (si existe) se convierte en el hijo derecho de p .

En ambos casos hay que tomar en cuenta que el vértice sobre el que se gira no tiene que ser la raíz y por lo tanto puede tener parentesco; hay que reconectar dicho parentesco con el vértice derecho o izquierdo, dependiendo de en qué dirección se gire.

La operación no es autoinvertible; girar primero a la derecha y luego a la izquierda sobre el mismo vértice no nos regresa al árbol original.

- `dfsPreOrder()`, `dfsPostOrder()` y `dfsInOrder()`

Los algoritmos correspondientes son idénticos a los algoritmos [14.1](#), [14.2](#) y [14.3](#), excepto que en lugar de imprimir el elemento del vértice, actuamos la acción recibida sobre el mismo.

14.4. Complejidades en tiempo y en espacio

Dejamos el análisis de las complejidades en tiempo y espacio para su propia sección porque en general van a depender de la altura del árbol.

Como un árbol binario ordenado está siempre ordenado, el hecho de agregarle n elementos implica automáticamente ordenarlos. Por lo que vimos en el capítulo [10](#) esto a su vez implica que no podemos hacerlo mejor que $O(n \log n)$. Y como son n elementos, esto significa que tendremos que ser capaces de agregar un elemento a un árbol binario ordenado en tiempo $O(\log n)$.

En promedio este será el caso; pero es dependiente de la altura del árbol. Si agregamos a un árbol binario ordenado n elementos, del más pequeño al más grande, además de que al final el árbol tendrá altura $n - 1$, la complejidad en tiempo de hacerlo será $O(n^2)$ (figura [14.9](#)).

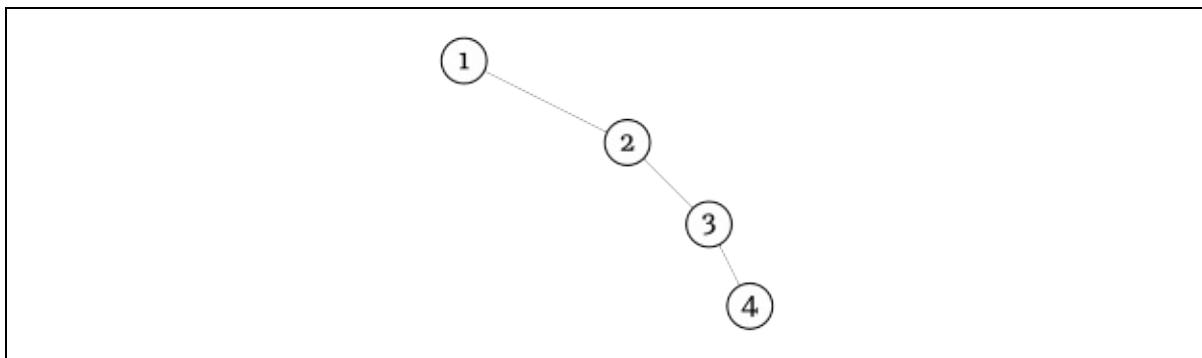


Figura 14.9: Resultado de agregar los elementos 1, 2, 3 y 4 (en ese orden) a un árbol binario ordenado.

De la misma manera, buscar un elemento tomará, en el peor de los casos, la altura del árbol; si el árbol tiene una altura lineal, la complejidad será $O(n)$; si la altura es logarítmica, la complejidad en tiempo será $O(\log n)$. Y como para eliminar un elemento primero debemos buscarlo, esto también se aplica al algoritmo para eliminar.

Los árboles binarios ordenados tienen un comportamiento similar al del algoritmo `QUICKSORT`; si agregamos los elementos ya ordenados, la complejidad en tiempo de agregar n elementos será $O(n^2)$. Pero si los elementos se agregan de manera aleatoria, la complejidad en tiempo de agregar n elementos será $O(n \log n)$.

Siempre será mejor para nosotros que el árbol binario ordenado no sea muy alto; al contrario, el árbol debe ser lo más corto en altura posible. En otras palabras, querremos que el número de hoyos en el árbol sea lo más pequeño posible. Es aquí donde entran en juego las operaciones de giro: con ellas podremos tomar un árbol alto y hacerlo más corto (figura [14.10](#)).

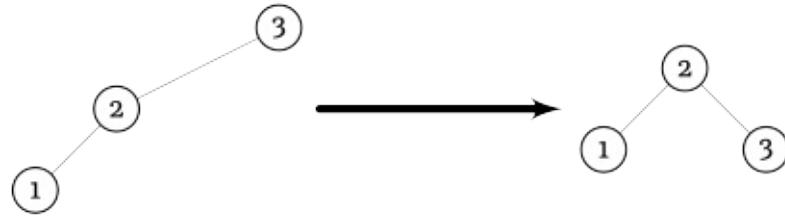


Figura 14.10: Si giramos el árbol a la derecha sobre su vértice 2, reducimos su altura y el número de hoyos que tiene.

Siendo más formales, lo que queremos no es nada más reducir la altura del árbol; queremos *balancearlo*. Definiremos un árbol binario balanceado como sigue.

Definición 14.4. (Árboles binarios balanceados) *Un árbol binario T es balanceado si y sólo si para todo vértice de T se cumple que la distancia máxima del vértice a alguna de sus hojas es a lo más c veces la distancia mínima del vértice a alguna de sus hojas, donde c es una constante fija.*

En particular un árbol binario lleno es trivialmente balanceado, así como un árbol binario completo. Sin embargo la definición 14.4 es bastante flexible: el árbol de la figura 14.11 es balanceado, a pesar de verse muy denso del lado derecho y muy ligero del izquierdo. La distancia del vértice 4 al vértice 15 (que es la máxima) es 4; la distancia del vértice 4 al vértice 1 (que es la mínima) es 2. Por lo tanto la distancia máxima es 2 veces la distancia mínima y esta es cota superior para la diferencia entre las distancias mínima y máxima de todos los vértices a alguna de sus hojas en el árbol, por lo que es balanceado con $c = 2$.

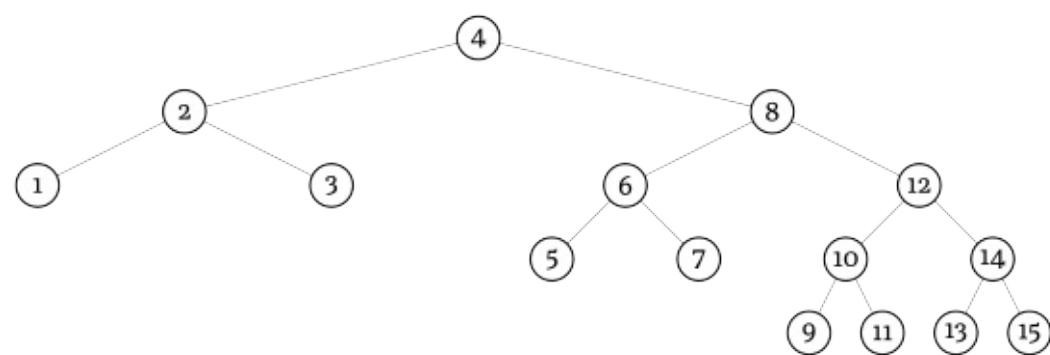


Figura 14.11: Árbol binario balanceado con $c = 2$.

Si al trabajar con nuestros árboles binarios ordenados éstos se mantienen balanceados entonces todas nuestras operaciones sobre ellos tendrán, en el peor de los casos, una complejidad en tiempo de $O(\log n)$. Y en promedio de hecho esto es lo que ocurre.

Sin embargo hay ocasiones en que las operaciones se vuelven lineales. Para evitar esto, nos gustaría poder mantener balanceados nuestros árboles binarios; y esto es justo lo que harán las últimas estructuras de árboles binarios ordenados que veremos en los capítulos [15](#) y [16](#).

Ejercicios

1. Implementa los métodos faltantes de la clase [ArbolBinarioOrdenado](#).
2. ¿Cuál es la complejidad en tiempo de agregar un elemento a un árbol binario ordenado?
3. ¿Cuál es la complejidad en espacio de agregar un elemento a un árbol binario ordenado?
4. Dado el árbol binario ordenado en la figura [14.12](#), gíralo a la derecha sobre el vértice que contiene a 3.

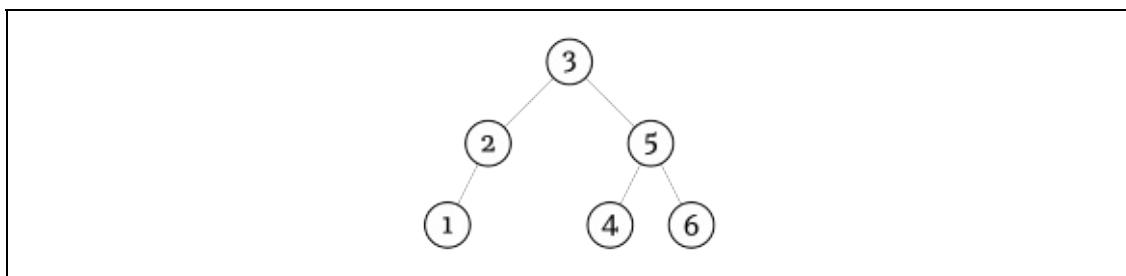


Figura 14.12: Árbol binario ordenado.

5. Dado el árbol binario ordenado resultante de la pregunta anterior, gíralo a la izquierda sobre el vértice que contiene a 3.

15. Árboles rojinegros

Como mencionábamos en el capítulo [14](#), muchas de las operaciones en árboles binarios ordenados dependen de la altura del árbol; en el mejor de los casos esta altura será logarítmica respecto al número de elementos, pero (de forma similar a lo que ocurre con `QUICKSORT`) dependiendo de cómo agreguemos y eliminemos elementos, la altura del árbol se puede disparar a lineal.

Si pudiéramos garantizar que el árbol está balanceado (bajo la definición [14.4](#)), entonces la altura del mismo será siempre logarítmica y todas sus operaciones tendrán complejidad $O(\log n)$ en tiempo.

Esto es algo muy fuerte; como mencionamos en el capítulo [11](#), complejidad en tiempo logarítmica es en general lo mejor que podremos esperar de algoritmos interesantes: si nuestros árboles binarios ordenados se mantienen balanceados, esto va a significar que podremos eliminar elementos de ellos en tiempo $O(\log n)$. Podremos entonces eliminar un elemento de un árbol con 1,000,000 de elementos en aproximadamente 20 operaciones (multiplicado por una constante pequeña). *Y manteniendo ordenado el árbol.*

Los primeros árboles binarios ordenados autobalanceables que veremos serán los árboles rojinegros.

15.1. Definición de árboles rojinegros

Definiremos los árboles rojinegros (*red-black trees*) como sigue:

Definición 15.1. (*Árboles rojinegros*) *Un árbol rojinegro T es un árbol binario ordenado que cumple las siguientes propiedades:*

1. *Todos los vértices son **negros** o **rojos**.*
2. *La raíz es **negra**.*
3. *Todas las hojas (\emptyset) son **negras**, como la raíz.*
4. *Un vértice **rojo** siempre tiene dos hijos **negros**.*
5. *Para todo vértice v de T , todas las trayectorias de v a alguna de sus hojas descendientes tiene el mismo número de vértices **negros**.*

Las propiedades 4 y 5 implican que un árbol rojinegro es, por definición,

balanceado: como todo vértice **rojo** tiene dos hijos **negros** y el número de vértices **negros** es el mismo para cualquier trayectoria de un vértice a alguna de sus hojas descendientes, esto significa que una rama tiene en el peor de los casos dos veces el número de vértices que otra rama (figura 15.1).

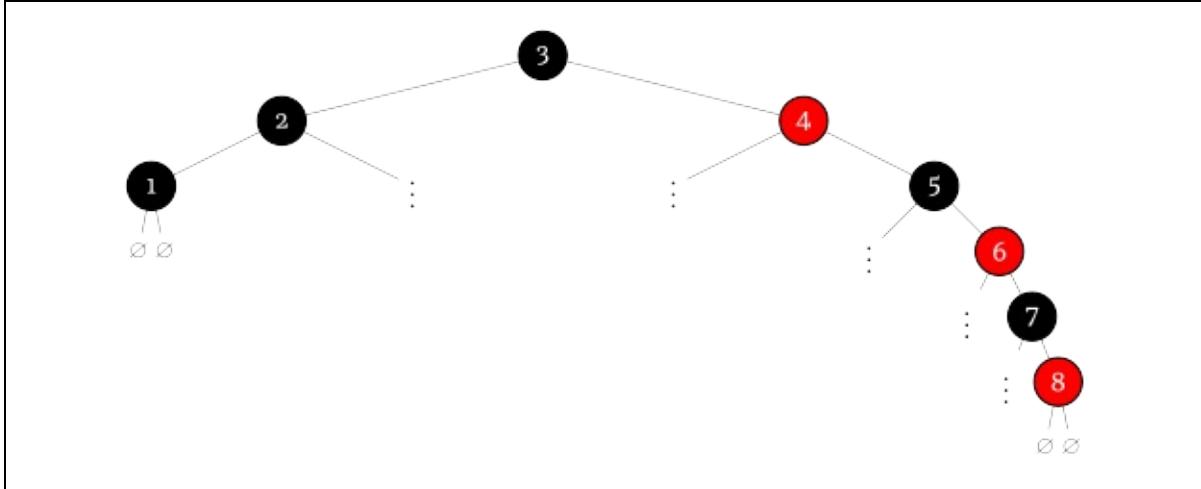


Figura 15.1: Las dos ramas mostradas del vértice 3 tienen el mismo número de vértices **negros**, incluyendo los vértices \emptyset .

En todas estas cuentas los vértices vacíos que tienen como hijos las hojas serán considerados como las “hojas” del árbol y serán **negros**. Contrario a la figura 15.1, normalmente no mostraremos las “hojas” vacías en nuestras figuras; un árbol rojinegro será mostrado como lo hacemos en la figura 15.2.

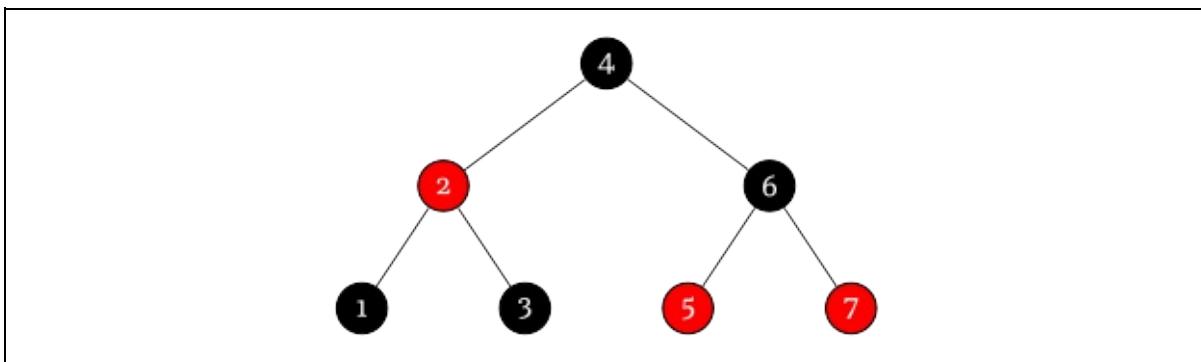


Figura 15.2: Árbol rojinegro; los vértices 5 y 7 cumplen la propiedad 4 por las dos “hojas” \emptyset que cada uno tiene.

Rudolf Bayer en 1972 introdujo lo que él llamó árboles-B simétricos [5], que fueron una evolución de los árboles-B que había inventado con Edward McCreight [4].

En 1978 Leonidas Guibas y Robert Sedgewick derivaron los árboles rojinegros de los árboles-B simétricos [19]; sin embargo son un caso particular de ellos, así que Bayer es al que se le da el crédito.

Lo que los árboles rojinegros hacen es, básicamente, rebalancear un árbol binario ordenado con giros cada vez que se le agregan o eliminan elementos

para que se sigan cumpliendo sus propiedades, utilizando los colores de los vértices para determinar qué debe hacerse en cada caso.

Nuestra implementación de árboles rojinegros, la clase `ArbolRojinegro`, extenderá a la clase `ArbolBinarioOrdenado` y por lo tanto podremos reutilizar mucho de la misma. El cambio más importante es que extenderemos la clase interna protegida `Vertice` con `VerticeRojinegro`, para poder agregarle una variable `color` (para esto usaremos, como suele ser el caso, una enumeración `color`). Además del constructor de esta clase interna, tendremos que sobrecargar los métodos `toString()` y `equals()` de `ArbolBinario.Vertice`.

En la clase `ArbolRojinegro` propiamente deberemos sobrecargar el método protegido `nuevoVertice()`, para que los vértices creados en `ArbolBinarioOrdenado` y `ArbolBinario` sean instancias de `VerticeRojinegro` y no de `Vertice`.

Obviamente tendremos que sobrecargar los métodos `agrega()` y `elimina()`, porque debemos rebalancear el árbol después de ambas operaciones.

También sobrecargaremos `giraDerecha()` y `giraIzquierda()`, para invalidar las operaciones en árboles rojinegros (siempre lanzarán una excepción). Esto es porque los usuarios de `ArbolRojinegro` no pueden girar los árboles sin desbalancearlos, así que sencillamente haremos imposible usar los métodos fuera de la clase (dentro usaremos la implementación de `ArbolBinarioOrdenado` usando `super`).

Hacemos notar que esto (el invalidar las operaciones de giro en los árboles rojinegros) es, técnicamente, una violación de la Orientación a Objetos; en particular al principio de sustitución de Liskov [25]: no podemos tomar cualquier código que reciba un árbol binario ordenado, pasarle un árbol rojinegro y garantizar que funcione si funcionaba con árboles binarios ordenados, porque el código fallará si intentamos realizar un giro. Si un árbol rojinegro no tiene todo el comportamiento de un árbol binario ordenado, se puede discutir que las clases correspondientes no deberían heredar una de la otra; sin embargo nos ahorramos mucho trabajo si heredamos los árboles rojinegros de los ordenados y, exceptuando por los giros, tienen el mismo comportamiento así que seremos pragmáticos y violaremos (técticamente) un diseño orientado a objetos “puro” con la meta de ahorrarnos trabajo.

Por último tendremos un método `getColor()` para poder preguntarle a una instancia de `VerticeArbolBinario` su color. Este método no es realmente necesario para el funcionamiento de la clase, pero sí para poder verificar que el árbol rojinegro es válido fuera de la misma.

Podemos ver el esqueleto de la clase `ArbolRojinegro` en el listado [15.1](#).

```
public class ArbolRojinegro<T extends Comparable<T>>
```

```

    extends ArbolBinarioOrdenado<T> {
    protected class VerticeRojinegro extends Vertice {
        public Color color;
        public VerticeRojinegro(T elemento) { /* ... */ }
        @Override public String toString() { /* ... */ }
        @Override public boolean
        equals(Object o) { /* ... */ }
    }
    public ArbolRojinegro() { /* ... */ }
    public ArbolRojinegro(Coleccion<T> colección) {
        /* ... */
    }
    public Color
    getColor(VerticeArbolBinario<T> vertice) { /* ... */ }
    @Override protected Vertice
    nuevoVertice(T elemento) { /* ... */ }
    @Override public void agrega(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public void
    giraIzquierda(VerticeArbolBinario<T> vertice) {
        throw new UnsupportedOperationException();
    }
    @Override public void
    giraDerecha(VerticeArbolBinario<T> vertice) {
        throw new UnsupportedOperationException();
    }
}
}

```

Listado 15.1: Esqueleto de la clase **ArbolRojinegro**. Mostramos los métodos `giraIzquierda()` y `giraDerecha()` ya invalidados.

15.2. Algoritmos de los árboles rojinegros

Como hemos venido haciendo en otros capítulos, veremos los algoritmos de los métodos de **ArbolRojinegro** en el orden dado por el listado 15.1; *excepto* por los métodos `agrega()` y `elimina()`.

Los algoritmos para rebalancear el árbol después de agregar y eliminar elementos serán probablemente los más complicados que veamos en todo el libro; vamos a dejarlos para el final del capítulo.

Comenzaremos con la clase interna **VerticeRojinegro**.

- **VerticeRojinegro()**

El constructor se encadena con el constructor de la clase padre (**Vertice**) y deja al color indefinido; en Java esto se traducirá en asignarle el valor `Color.NINGUNO` de la enumeración.

- `toString()`

En lugar de nada más regresar la representación en cadena del elemento del vértice, vamos a envolver la misma en las cadenas "`R{"` y `"}`", si el vértice es **rojo**; y "`N{"y "}`", si el vértice es **negro**.

Con esto podremos distinguir el color de nuestros vértices en las cadenas que representen nuestros árboles.

- `equals()`

Podemos reutilizar el método `equals()` de `Vertice` (usando `super`), pero antes nada más hay que comparar que los colores de ambos vértices sean el mismo.

Como hemos hecho hasta ahora, debemos hacer una audición y suprimir advertencias al hacerlo. El método es tan sencillo que lo mostramos en el listado [15.2](#).

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    VerticeRojinegro vertice =
        (VerticeRojinegro)o;
    return (color == vertice.color &&
            super.equals(o));
}
```

Listado 15.2: Método `equals()` de `VerticeRojinegro`.

Ahora podemos ver los métodos de la clase propia.

- `ArbolRojinegro()` y `ArbolRojinegro(Coleccion<T>)`.

Sencillamente encadenamos (mandamos llamar) el constructor correspondiente de la clase padre.

- `getColor()`

Sólo hay que convertir la instancia de `VerticeArbolBinario` a una instancia de `VerticeRojinegro` y regresar el color del vértice.

- `nuevoVertice()`

Creamos un vértice rojinegro y lo regresamos. Es tan sencillo que lo mostramos en el listado [15.3](#)

```
@Override protected Vertice
```

```

nuevoVertice(T elemento) {
    return new VerticeRojinegro(elemento);
}

```

Listado 15.3: El método nuevoVertice() de `ArbolRojinegro`.

Y esto termina los métodos más sencillos de la clase `ArbolRojinegro`. Nada más nos faltan ver los algoritmos para agregar y eliminar.

15.2.1. Algoritmo para agregar

Para agregar elementos a un árbol rojinegro reutilizaremos el algoritmo de árboles binarios ordenados; en Java esto se traducirá en invocar el método `agrega()` de `ArbolBinarioOrdenado` utilizando `super`.

Una vez hecho esto tomamos el último vértice agregado que tenemos en la variable de clase `ultimoAgregado`; como justo acabamos de agregar un vértice, sabemos que el valor de la variable es válido.

Convertimos este vértice en un vértice rojinegro haciéndole una audición y le ponemos el color **rojo**; todos los vértices agregados originalmente tendrán color **rojo**: al rebalancear el árbol el color del vértice agregado podrá cambiar a **negro**.

Vamos a utilizar un algoritmo auxiliar recursivo para rebalancear el árbol rojinegro. El algoritmo *siempre* siempre recibe un vértice rojinegro v de color **rojo** (y por lo tanto distinto de \emptyset) y funciona por casos.

- **Caso 1.** El vértice v tiene padre \emptyset : coloreamos v de **negro** y terminamos.

Si el **Caso 1** no ocurre, entonces el vértice v tiene padre distinto de \emptyset ; sea p el vértice padre de v .

- **Caso 2.** El vértice p es **negro**: como el vértice v es **rojo**, el árbol no se ha desbalanceado y terminamos.

Si el **Caso 2** no ocurre, entonces el vértice p es **rojo** y por lo tanto el abuelo de v (el padre de p) es distinto de \emptyset ; sea a el abuelo de v (el padre de p). Además, sea t el tío de v (el hermano del padre; el hijo de a distinto de p); el vértice t *puede* ser \emptyset .

- **Caso 3.** El vértice t es **rojo**: como p también es **rojo**, coloreamos t y p de **negro**, coloreamos a de **rojo** y hacemos recursión sobre a (figura 15.3). Regresando de la recursión terminamos.

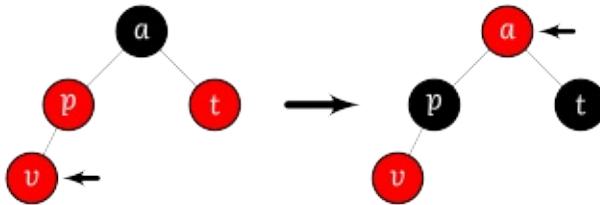


Figura 15.3: Caso 3: coloreamos p y t de **negro**, a de **rojo** y hacemos recursión sobre a .

Este caso lo que hace es balancear el subárbol $T(a)$ y recursivamente lidiar con el caso de un “nuevo” vértice **rojo** potencialmente desbalanceando el resto del árbol.

Una nota respecto a la figura 15.3: en este ejemplo v es izquierdo de p , p es izquierdo de a y t es derecho de a . El caso debe lidiar con todas las posibilidades, como que v sea derecho o t izquierdo, etcétera. Lo mismo aplicará en las distintas figuras que usaremos para el resto de los casos.

Si el **Caso 3** no ocurre, entonces el vértice t es **negro**. El vértice t será distinto de \emptyset únicamente dentro de una llamada recursiva al algoritmo; en otras palabras, t será siempre \emptyset si los dos hijos de v son \emptyset . A partir de ahora ignoraremos a t .

- **Caso 4.** Los vértices v y p están *cruzados*; en otras palabras o bien p es derecho y v es izquierdo o bien p es izquierdo y v derecho. Los “enderezamos”: giramos sobre p en su dirección. En otras palabras, si p es izquierdo giramos sobre p a la izquierda; si p es derecho, giramos sobre p a la derecha. Actualizamos v para que sea p y p para que sea v ; en otras palabras el vértice v (el vértice “actual”) se mantiene en el mismo nivel que antes de ejecutar el caso y p continúa siendo su padre (figura 15.4).

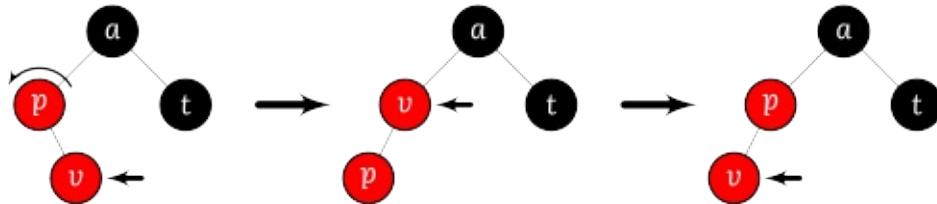


Figura 15.4: Caso 4: giramos sobre p en su dirección y actualizamos v a p y p a v .

Debe quedar claro que no reemplazamos los elementos de p y v ; sólo renombramos los vértice para que v siga siendo el vértice en el nivel original y p su parent. En Java será equivalente a actualizar las referencias.

Este caso *no termina*; continúa al **Caso 5**.

- **Caso 5.** Los vértices v y p no están cruzados; en otras palabras o bien v y p son izquierdos o bien v y p son derechos. No hay necesidad de verificar nada; si llegamos a este punto en el algoritmo es lo único que puede ocurrir.

Coloreamos al vértice p de **negro**, al vértice a de **rojo** y giramos sobre a en la dirección *contraria* a la de v : en otras palabras, si v es izquierdo giramos sobre a a la derecha; si v es derecho, giramos sobre a a la izquierda (figura 15.5).

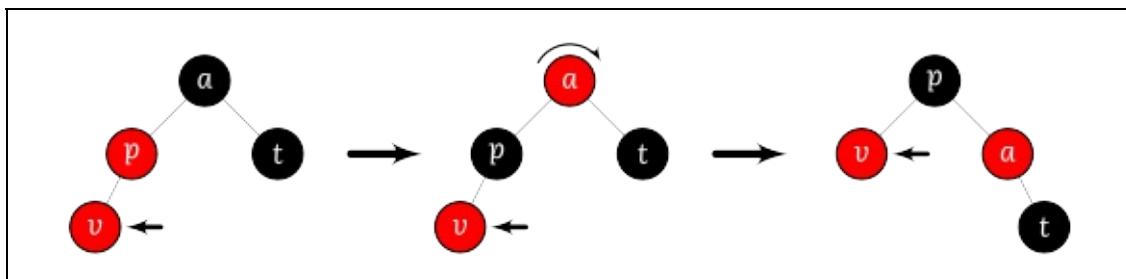


Figura 15.5: Caso 5: coloreamos p de **negro**, a de **rojo** y giramos sobre a en la dirección contraria de v .

Con esto balanceamos el subárbol $T(p)$ y terminamos.

La demostración de que el algoritmo para agregar en árboles rojinegros mantiene las propiedades de los mismos no es trivial y no la cubriremos. Sin embargo debe quedar claro que su complejidad en tiempo es $O(\log n)$; todos los casos toman tiempo $O(1)$ (recordemos que girar sobre un vértice toma tiempo $O(1)$) excepto el **Caso 3**, que hace recursión sobre el abuelo del vértice. Como comenzamos con un vértice hoja del árbol y el **Caso 1** hace que el algoritmo termine al llegar al vértice raíz, en el peor de los casos el algoritmo toma un número de pasos proporcional a la altura del árbol (de hecho la mitad, porque nos brincamos al vértice padre al hacer recursión). Y al ser árboles rojinegros, por definición balanceados, el algoritmo toma a lo más tiempo $O(\log n)$. La complejidad en espacio es la misma por la pila de ejecución.

Para implementar este algoritmo en Java nos conviene escribir métodos auxiliares que nos permitan verificar ciertas propiedades de manera segura. Por ejemplo se puede implementar un método auxiliar para verificar si un vértice es rojo como se muestra en el listado 15.4.

```
private boolean esRojo(VerticeRojinegro vertice) {
    return (vertice != null &&
            vertice.color == Color.ROJO);
}
```

Listado 15.4: Método auxiliar seguro para verificar si un vértice es rojo.

El método es seguro porque no es posible que falle con una excepción; podemos pasarle como argumento cualquier vértice sin tener que preocuparnos de si es o no `null`. Tener algoritmos similares para obtener el padre, el tío, etcétera de un vértice hace el código más fácil de leer y mantener.

15.2.2. Algoritmo para eliminar

Para eliminar un vértice reutilizaremos los algoritmos auxiliares para eliminar de los árboles binarios ordenados. Recordemos que nos aseguramos de que el vértice a eliminar tenga a lo más un hijo distinto de \emptyset ; si éste no es el caso, buscamos el máximo del subárbol izquierdo del vértice e intercambiamos los elementos. Tenemos un algoritmo auxiliar que hace esto y otro que se encarga de eliminar el vértice.

Para árboles rojinegros vamos a necesitar que el vértice eliminado tenga exactamente un hijo distinto de \emptyset ; si ambos hijos del vértice son \emptyset , vamos a crearle un vértice que llamaremos *fantasma*. El punto del vértice fantasma es que podamos girar nuestros vértices sin tener que preocuparnos de un hoyo en el árbol. Al final del algoritmo eliminaremos el vértice fantasma (si es que lo creamos). En nuestra implementación con Java un vértice fantasma tendrá como elemento `null`.

El rebalanceo se hará también recursivamente, como en el algoritmo para agregar; el vértice de entrada será el vértice hijo del vértice eliminado (por lo que puede ser fantasma).

Primero buscamos el vértice que contenga el elemento a eliminar; si no existe, terminamos. Sea v el vértice a eliminar si existe.

Si v tiene dos hijos distintos de \emptyset utilizamos el algoritmo auxiliar que hace el intercambio con el máximo de su subárbol izquierdo. Después, si los dos hijos de v son \emptyset , le agregamos un vértice fantasma. Independientemente de qué ocurre, para este momento v tiene exactamente un hijo distinto de \emptyset . Sea h este hijo.

Ahora utilizamos el algoritmo auxiliar que desconecta el vértice a eliminar del resto del árbol; esto deja al vértice h en el lugar que ocupaba el vértice v . Tenemos entonces 3 posibilidades respecto a los colores de v y h :

- h es **rojo** y por lo tanto v es **negro**. Sencillamente coloreamos h de **negro** y terminamos.
- v es **rojo** y por lo tanto h es **negro**. No tenemos que hacer nada.
- v y h son los dos **negros**. Rebalanceamos sobre h .

Regresando del algoritmo que rebalancea eliminamos el vértice fantasma (si existe) y terminamos. Evidentemente no hay que preocuparse del caso cuando v y h son **rojos**, porque esto no es posible dada la propiedad 4 de los árboles rojinegros.

El algoritmo auxiliar recursivo de rebalanceo *siempre* recibe un vértice v **negro** distinto de \emptyset (aunque puede ser fantasma) y también funciona por casos.

- **Caso 1.** El vértice v tiene padre \emptyset ; terminamos.

Si el **Caso 1** no se cumple, el padre de v es distinto de \emptyset ; sea p el vértice padre de v . Como el vértice eliminado (el que v está reemplazando) era **negro** y distinto de \emptyset (lo eliminamos al fin y al cabo), entonces el hermano de v tiene que ser distinto de \emptyset . Sea h el hermano de v .

- **Caso 2.** El vértice h es **rojo** y por lo tanto p es **negro**; coloreamos a p de **rojo**, a h de **negro** y giramos sobre p en la dirección de v . En otras palabras, si v es izquierdo giramos sobre p a la izquierda; y si v es derecho giramos sobre p a la derecha (figura 15.6).

Este caso *no termina*; continúa al **Caso 3**. Si ocurre el caso, hay que actualizar h para que vuelva a ser el hermano de v después de girar y siempre será distinto de \emptyset ; de otra forma el árbol hubiera estado desbalanceado antes de eliminar. A partir de este momento, h_i y h_d denotarán los hijos izquierdo y derecho de h , respectivamente; ambos pueden ser \emptyset .

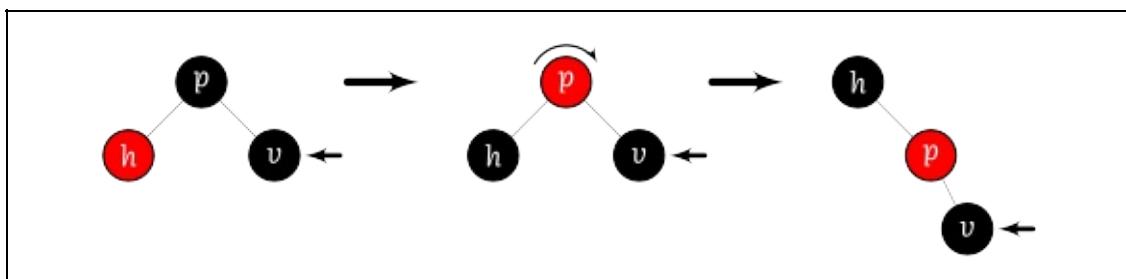


Figura 15.6: **Caso 2:** coloreamos p de **rojo**, h de **negro** y giramos sobre p en la dirección de v .

- **Caso 3.** Los vértices p , h , h_i y h_d son **negros** (h_i y h_d pueden ser \emptyset); coloreamos h de **rojo**, hacemos recursión sobre p y terminamos al regresar de la recursión (figura 15.7).

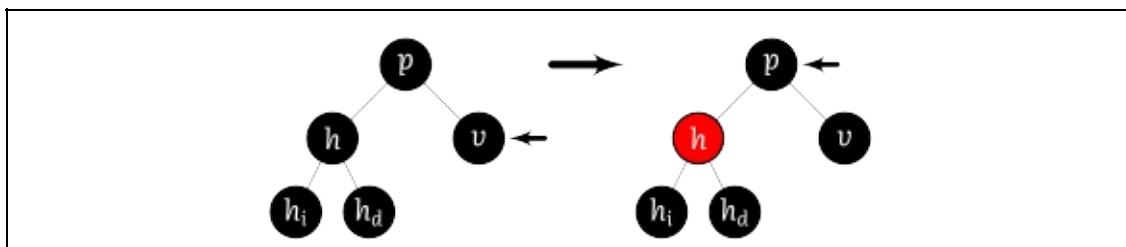


Figura 15.7: Caso 3: coloreamos h de **rojo** y hacemos recursión sobre p .

El desbalanceo causado por el que era padre de v se balancea al colorear h de **rojo**, lo que deja al subárbol $T(p)$ balanceado, así que continuamos el algoritmo sobre p .

- **Caso 4.** Los vértices h , h_i y h_d son **negros** y p es **rojo**; coloreamos h de **rojo**, a p de **negro** y terminamos (figura 15.8).

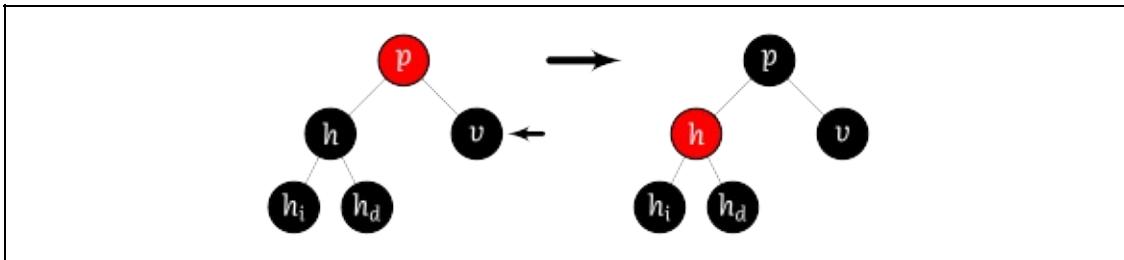


Figura 15.8: Caso 4: coloreamos h de **rojo** y p de **negro**.

Al colorear h de **rojo** y p de **negro**, el subárbol $T(p)$ queda balanceado, pero además se recupera el vértice **negro** que se había perdido al eliminar al que era padre de v para cumplir la propiedad 5 de los árboles rojinegros. Por lo tanto terminamos.

- **Caso 5.** El vértice v es izquierdo, h_i es **rojo** y h_d es **negro**; o el vértice v es derecho, h_i es **negro** y h_d es **rojo**; coloreamos h de **rojo**, al hijo **rojo** de h de **negro** y giramos sobre h en la dirección contraria a v (figura 15.9).

Este caso *no termina*; continúa al **Caso 6**. Si ocurre el caso, hay que actualizar h para que vuelva a ser el hermano de v después de girar y será distinto de \emptyset porque era **rojo** antes de que lo coloreáramos **negro**.

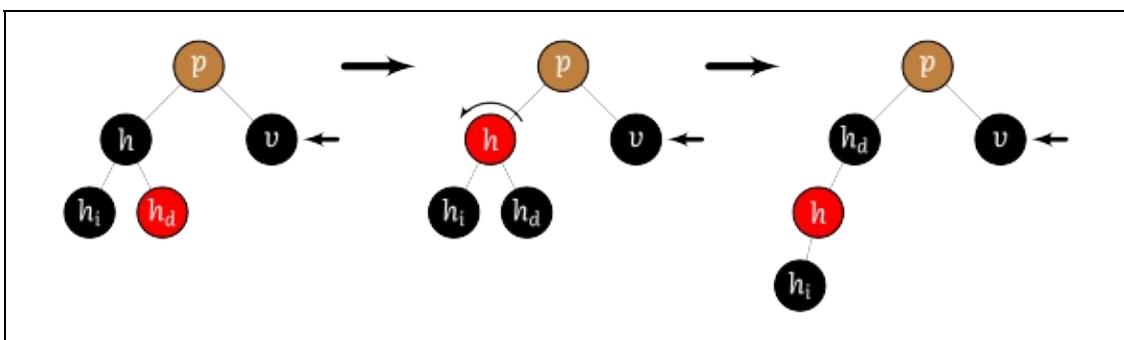


Figura 15.9: Caso 5: coloreamos h de **rojo**, a h_d de **negro** y giramos sobre h a la izquierda. No sabemos el color de p .

- **Caso 6.** El vértice v es izquierdo y h_d es **rojo** o el vértice v es derecho y h_i es **rojo**. No hay que verificar nada, si llegamos a este punto en el algoritmo es lo que tiene que ocurrir.

Coloreamos h con el mismo color de p , a p de **negro**, al hijo de h con dirección contraria a v de **negro** y giramos sobre p en la dirección de v (figura 15.10).

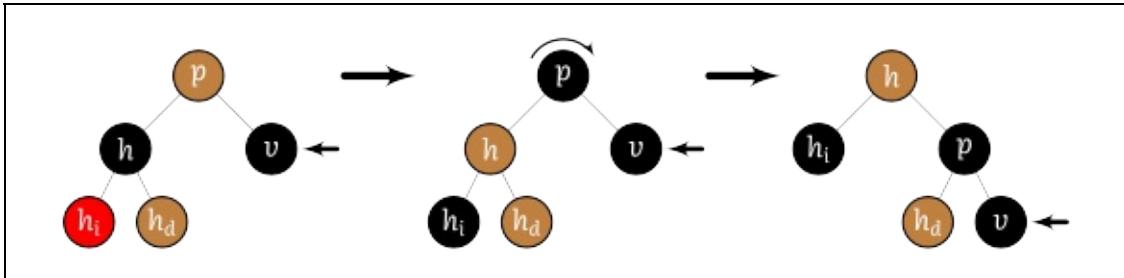


Figura 15.10: Caso 6: coloreamos h con el mismo color de p , a p de **negro**, a h_i de **negro** y giramos sobre p a la derecha. No sabemos el color de p ni de h_d .

Independientemente del color original de p (que se traslada a h) y el de h_d , esto balancea el subárbol $T(h)$ y además rebalancea $T(p)$ reponiendo el vértice negro que eliminamos.

Tampoco veremos la prueba de que el algoritmo de eliminación es correcto. Su complejidad en tiempo es $O(\log n)$; todos los casos toman tiempo $O(1)$, excepto el **Caso 3** donde hacemos recursión. La recursión es sobre el padre del vértice actual y comenzamos (en el peor de los casos) en una hoja, así que siempre terminamos en la raíz. Por lo tanto la complejidad en tiempo está acotada por la altura del árbol y al estar éste balanceado la complejidad será $O(\log n)$.

15.3. Usos de árboles rojinegros

Los árboles rojinegros son utilizados extensivamente en producción en muchos sistemas; la biblioteca estándar de Java los utiliza como motor para su implementación de la clase `TreeMap` [38]; libgee, la biblioteca estándar de colecciones del lenguaje de programación Vala los utiliza como motor para la implementación de su clase `TreeSet` [40]; la biblioteca estándar de C del proyecto GNU (GNU glibc) los utiliza por omisión para manejar árboles binarios en la familia de funciones `tsearch()` [34]; el kernel de Linux tiene su propia implementación [39]; y en general es la implementación por omisión utilizada para estructuras ordenadas altamente dinámicas donde las operaciones de agregar, eliminar y buscar requieren complejidad en tiempo $O(\log n)$.

Es difícil encontrar una estructura más óptima que los árboles rojinegros; dados los requerimientos que cubren (que mantienen ordenada la colección de elementos) todas sus operaciones son óptimas en tiempo y además utilizan

muy poca memoria extra a la que ocupan sus elementos; algunas implementaciones incluso codifican el color de los vértices utilizando un único bit para ahorrar consumo de memoria.

Para hacer una comparación con los árboles rojinegros veremos otra versión de árboles binarios ordenados autobalanceables; pero realmente con árboles rojinegros nos basta para cualquier tipo de colección que necesite mantenerse ordenada.

Ejercicios

1. Implementa los métodos faltantes de la clase [ArbolRojinegro](#).
2. ¿Qué propiedad de los árboles rojinegros garantiza que estén balanceados?
3. ¿Cuál es la complejidad en tiempo de agregar, eliminar y buscar un elemento en un árbol rojinegro?
4. Dado el árbol rojinegro en la figura [15.11](#), agrégale el elemento 8.

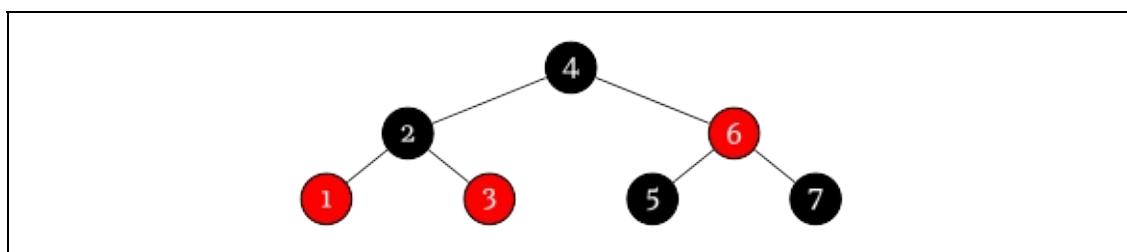


Figura 15.11: Árbol rojinegro.

5. Dado el árbol rojinegro resultante de la pregunta anterior, elimínale el elemento 4.

16. Árboles AVL

La segunda estructura de datos que veremos que implementa árboles binarios ordenados autobalanceables serán los árboles AVL. Al igual que los rojinegros, los árboles AVL mantienen una altura de orden $O(\log n)$, lo que nos permitirá realizar las operaciones de agregar, eliminar y buscar con complejidad en tiempo de $O(\log n)$.

16.1. Definición de árboles AVL

Para definir a los árboles AVL (*AVL trees*) nos conviene primero definir el balance de un vértice.

Definicion 16.1. (Balance de vértices) *Sea T un árbol binario y v un vértice de T . El balance del vértice v , denotado por $b(v)$, estará definido como $h(T_i(v)) - h(T_d(v))$.*

El balance de un vértice es la diferencia entre la altura de su subárbol izquierdo y la altura de su subárbol derecho. Definido el balance, podemos definir los árboles AVL de la siguiente manera:

Definicion 16.2. (Árboles AVL) *un árbol AVL T es un árbol binario ordenado donde para todo vértice v de T se cumple que $-1 \leq b(v) \leq 1$.*

En otras palabras, en un árbol AVL la diferencia de las alturas de los subárboles izquierdo y derecho de un vértice será -1 , 0 o 1 (figura 16.1). Un árbol AVL por definición estará balanceado y de hecho en promedio siempre tendrá menor altura que un árbol rojinegro con la misma cantidad de elementos.

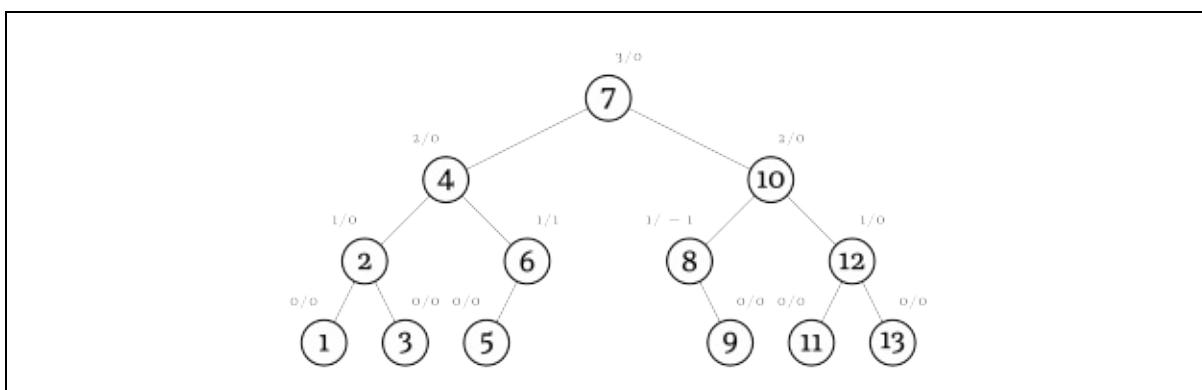


Figura 16.1: Árbol AVL. Cada vértice muestra h/b , donde h es la altura del vértice y b es su balance.

Los árboles AVL son así llamados por sus dos inventores soviéticos: Georgy

Adelson-Velsky y Evgenii Landis [1]. La creación de los mismos en 1962 antecede a la de los árboles rojinegros por diez años y los árboles AVL son más fáciles de implementar que éstos: como la restricción de altura de los árboles AVL es más fuerte que con los rojinegros, esto se traduce en un algoritmo de rebalanceo mucho más sencillo. Si utilizamos algoritmos distintos para rebalancear los árboles AVL después de agregar o eliminar elementos, esto nos permite hacer un poco más rápido el algoritmo de agregación; sin embargo ambos algoritmos de cualquier manera tendrán complejidad en tiempo $O(\log n)$ y podemos utilizar un único algoritmo de rebalanceo para ambas operaciones, así que haremos eso para simplificar nuestra implementación.

En nuestra implementación en Java de los árboles AVL, la clase `ArbolAVL` extenderá a `ArbolBinarioOrdenado` y al igual que `ArbolRojinegro` también extenderá la clase interna `Vertice` con `VerticeAVL`, para poder agregarle una propiedad `altura` a nuestros vértices y así poder calcular el balance de cada uno. La clase interna `VerticeAVL` obviamente sobrecargará el método `altura()`, porque ahora consistirá en únicamente regresar la altura guardada del vértice y también sobrecargará `equals()` y `toString()`.

```

public class ArbolAVL<T extends Comparable<T>>
    extends ArbolBinarioOrdenado<T> {

    protected class VerticeAVL extends Vertice {
        public int altura;
        public VerticeAVL(T elemento) { /* ... */ }
        @Override public int altura() { /* ... */ }
        @Override public String toString() { /* ... */ }
        @Override public boolean
        equals(Object object) { /* ... */ }
    }

    public ArbolAVL() { /* ... */ }
    public ArbolAVL(Coleccion<T> colección) { /* ... */ }
    @Override protected Vertice
    nuevoVertice(T elemento) { /* ... */ }
    @Override public void agrega(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public void
    giraDerecha(VerticeArbolBinario<T> vertice) {
        throw new UnsupportedOperationException();
    }
    @Override public void
    giraIzquierda(VerticeArbolBinario<T> vertice) {
        throw new UnsupportedOperationException();
    }
}

```

Listado 16.1: Esqueleto de la clase **ArbolAVL**. Mostramos los métodos `giraIzquierda()` y `giraDerecha()` ya invalidados.

La clase **ArbolAVL** sobrecargará `nuevovertice()`, `agrega()` y `elimina()`; y como hace **ArbolRojinegro** también invalidará `giraDerecha()` y `giraIzquierda()`. No es necesario sobrecargar `altura()`; el sobrecargar el método correspondientes de los vértices se propaga al método de la clase **ArbolBinario**, que únicamente regresa la altura del vértice raíz (a menos que el árbol sea \emptyset , en cuyo caso regresa -1).

Podemos ver el esqueleto de la clase **ArbolAVL** en el listado [16.1](#).

16.2. Algoritmos de los árboles AVL

Como hemos hecho a lo largo del libro, veremos los algoritmos de la clase **ArbolAVL** en el orden dado en el listado [16.1](#); comenzaremos con la clase interna **VerticeAVL**.

- **VerticeAVL()**

El constructor se encadenará con el de la clase padre; no hay necesidad de hacer nada más, ya que un vértice recién construido es por definición una hoja y por lo tanto su altura es 0.

- `altura()`

Sencillamente regresamos la altura del vértice.

- `toString()`

Además de la representación en cadena del elemento en el vértice, también le concatenaremos la altura del vértice, una diagonal y el balance de vértice, que será la diferencia de las alturas de sus hijos.

- `equals()`

Reutilizaremos el método `equals()` de **Vertice** después de comprobar que las alturas de los dos vértices sean la misma. El método es tan sencillo que lo mostramos en el listado [16.2](#).

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    VerticeAVL vertice = (VerticeAVL)o;
    return (altura == vertice.altura && super.equals(o));
}
```

Listado 16.2: Método `equals()` de `VerticeAVL`.

Ahora podemos proceder a los métodos de la clase `ArbolAVL`.

- `ArbolAVL()` y `ArbolAVL(Coleccion<T>)`.

Sencillamente encadenamos (manda llamar) el constructor correspondiente de la clase padre.

- `nuevoVertice()`

Creamos un vértice AVL y lo regresamos. Es tan sencillo que lo mostramos en el listado 16.3

```
@Override protected Vertice
nuevoVertice(T elemento) {
    return new VerticeAVL(elemento);
}
```

Listado 16.3: El método `nuevoVertice()` de `ArbolAVL`.

- `agrega()`

Agregamos el elemento usando el algoritmo de árboles binarios ordenados y luego rebalanceamos sobre el padre del vértice agregado. El algoritmo de rebalanceo lo veremos al final del capítulo.

- `elimina()`

Buscamos el elemento usando el algoritmo de búsqueda de árboles binarios ordenados; si el elemento no está terminamos.

Si el elemento está utilizamos el algoritmo auxiliar que tenemos para asegurarnos de que el vértice que lo contiene tenga a lo más un hijo. Despues utilizamos el algoritmo auxiliar para eliminar el vértice y rebalanceamos sobre el padre del vértice eliminado. El algoritmo de rebalanceo lo veremos al final del capítulo.

El algoritmo de rebalanceo es relativamente sencillo, en particular en comparación con los utilizados por los árboles rojinegros. De cualquier forma lo cubriremos en su propia sección.

16.2.1. Algoritmo de rebalanceo

Como mencionábamos arriba, el algoritmo de rebalanceo funcionará para las operaciones de agregar y eliminar un elemento. La restricción que tienen los árboles AVL (que todo vértice tiene subárboles de casi la misma altura) es tan

fuerte que implementar el algoritmo es relativamente sencillo: sólo hay que recorrer desde el padre del vértice agregado o eliminado hacia la raíz y girar uno o dos vértices dependiendo del balance que tenga el vértice actual.

Para entender el algoritmo de rebalanceo es necesario entender cómo los giros afectan el balance de los vértices involucrados. Si un vértice q tiene un hijo izquierdo p y giramos q a la derecha, todos los vértices en los subárboles izquierdo y derecho de p no modifican ni su altura ni su balance. Tampoco les pasa nada a los vértices en el subárbol derecho de q . Los únicos vértices que *pueden* cambiar su altura y balance (y es posible que no ocurra) son justamente p y q (y obviamente sus antecesores); lo mismo ocurre al girar sobre un vértice a la izquierda (figura 16.2).

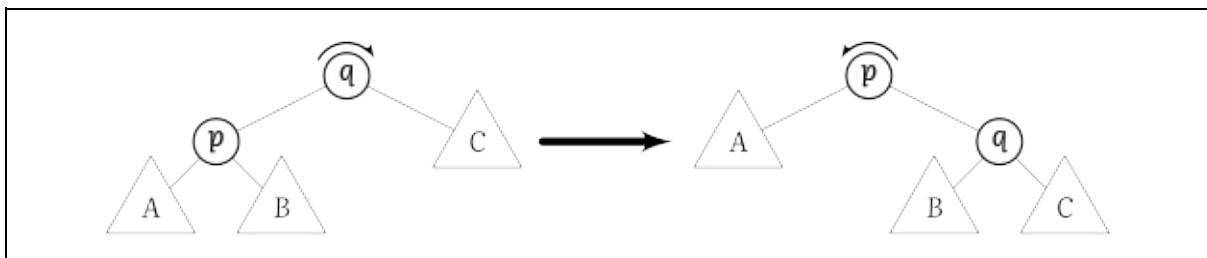


Figura 16.2: Al girar sobre q a la derecha o sobre p a la izquierda, los únicos vértices que pueden cambiar altura o balance son justamente p y q (y sus antecesores); los vértices en los subárboles A , B y C (que pueden ser vacíos) no cambian ni su altura ni su balance.

Al realizar giros en árboles AVL sólo será necesario recalcular las alturas de los dos vértices involucrados en el giro; aunque los antecesores de los mismos podrían ver su altura modificada, ésta será calculada automáticamente al hacer recursión.

El algoritmo de rebalanceo siempre recibirá el vértice padre v del vértice agregado o del vértice eliminado y será recursivo.

Si v es \emptyset el algoritmo termina; esta es la cláusula de escape. Si v no es \emptyset se actualiza la altura de v como el máximo entre las alturas de sus hijos más 1. Además se calcula el balance de v como la diferencia entre las alturas de sus hijos.

Lo único que puede ocurrir para que el vértice v esté desbalanceado es que su balance sea -2 o 2 ; como el árbol cumplía el requisito de los árboles AVL antes de agregar o eliminar y como estas operaciones modifican la altura de los vértices en la rama afectada a lo más en uno, entonces el balance de un vértice no puede cambiar en más de uno y por lo tanto en el peor de los casos será -2 o 2 . Los dos casos son simétricos, pero los veremos individualmente.

- Se tiene $b(v) = -2$. El subárbol izquierdo de v tiene una altura menor en 2 a la del subárbol derecho de v ; por lo tanto el vértice derecho de v tiene que ser distinto de \emptyset y además tener un hijo distinto de \emptyset : sean p y q los

hijos izquierdo y derecho de v , respectivamente. Como q es distinto de \emptyset , sean x y y sus hijos izquierdo y derecho respectivamente.

El caso más extremo para que v tenga balance -2 es que p sea \emptyset y por lo tanto tenga altura -1 y que q tenga altura 1 . Esto resultaría en que el balance de v fuera $(-1)-1 = -2$.

Sea $H = h(v)$, la altura de v al comenzar el caso. Como el balance de v es -2 , entonces $h(p)=H-3$ y $h(q)=H-1$ respectivamente. Para balancear el subárbol $T(v)$ hacemos lo siguiente: si q tiene balance 1 , por definición $h(x)-h(y)=1 \Rightarrow h(x)=h(y)+1$ y por lo tanto $h(x)=h(q)-1 = H-2$ y $h(y)=H-3$. Giramos sobre q a la derecha.

Como mencionábamos al inicio de la sección, el giro únicamente afecta la altura de q y x (y sus antecesores, en este caso v), dejando al resto de los vértices en sus subárboles con la misma altura.

Sean x_i y x_d los vértices izquierdo y derecho de x respectivamente (pueden ser \emptyset); x_i y x_d tienen dos posibilidades para su altura antes del giro; ambos pueden tener altura $H-3$ o $H-4$, pero al menos uno debe tener altura $H-3$. Esto es porque antes del giro $h(x)=H-2$ y x estaba balanceado. Las distintas posibilidades para las alturas de x_i y x_d se pueden ver en el cuadro [16.1](#).

$h(x_i)$	$h(x_d)$	$b(x)$
$H-4$	$H-3$	-1
$H-3$	$H-3$	0
$H-3$	$H-4$	1

Cuadro 16.1: Las posibles alturas de los vértices x_i y x_d y el balance del vértice x antes de girar sobre q a la derecha.

Como al girar sobre x a la derecha las alturas de x_i , x_d y y no cambian, entonces $h(x)=H-1$ y $h(q)=H-2$. El balance de x será entonces $(H-3)-(H-2)=-1$, si $h(x_i)=H-3$; o bien $(H-4)-(H-2)=-2$, si $h(x_i)=H-4$ (figura [16.3](#)).

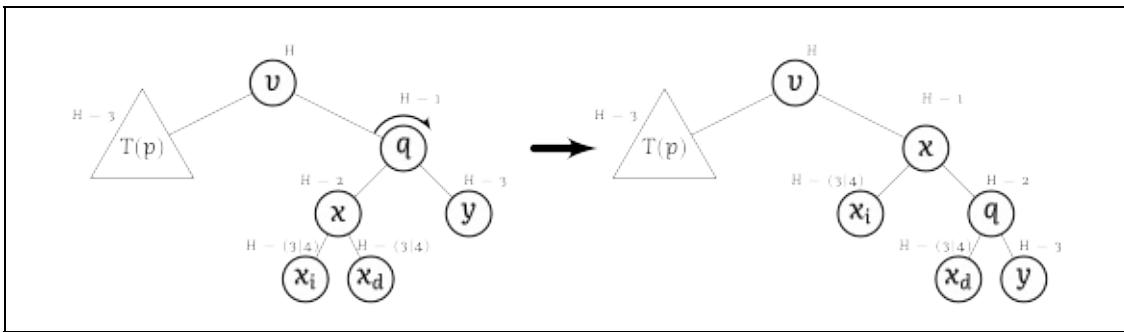


Figura 16.3: Si v tiene un balance de -2 y q tiene un balance de 1 , entonces giramos sobre q a la derecha, lo que deja al vértice x con un balance de -2 o -1 .

No hay de qué preocuparse si $b(x) = -2$; la siguiente parte del algoritmo se encargará de eso. Si giramos sobre q actualizamos nuestros nombres para que de nuevo q sea el hijo derecho de v , x el hijo izquierdo de q y y el hijo derecho de q .

Sea $H = h(v)$, la altura del vértice v en este punto del algoritmo. En este punto del caso cuando el balance de v es -2 , su hijo derecho q tiene balance 0 , -1 o -2 y su altura es $H - 1$, hayamos o no girado sobre q y renombrado.

Como la altura de q es $H - 1$ y su balance es 0 , -1 o -2 , la altura del vértice y sólo puede ser $H - 2$ como se muestra en el cuadro 16.2. La altura del vértice x puede ser $H - 2$, $H - 3$ o $H - 4$.

$h(x)$	$h(y)$	$b(q)$
$H - 2$	$H - 2$	0
$H - 3$	$H - 2$	-1
$H - 4$	$H - 2$	-2

Cuadro 16.2: Las posibles alturas de los vértices x y y y el balance del vértice q antes de girar sobre v .

Giramos sobre el vértice v a la izquierda; los únicos vértices que cambian de altura son v y q en el subárbol $T(q)$. La altura de v dependerá del vértice x : si $h(x) = H - 2$, la altura de v será $H - 1$; si $h(x) = H - 3$ o $h(x) = H - 4$, la altura de v será $H - 2$. El vértice p se mantiene con altura $H - 3$ y la altura de v no depende de él.

La altura del vértice q dependerá de la altura de v ; si $h(v) = H - 1$, la altura de q será H . Si $h(v) = H - 2$, como $h(y) = H - 2$ también, la altura de q será $H - 1$.

Cualquiera que sea el caso, el balance del vértice q será

$(H - 2) - (H - 2) = 0$ si $h(x) = H - 2$; $(H - 3) - (H - 2) = -1$ si $h(x) = H - 3$; o $h(x) = H - 4$; esto ocurre incluso si el balance de q era -2 . Por lo tanto el subárbol $T(q)$ queda balanceado (figura 16.4).

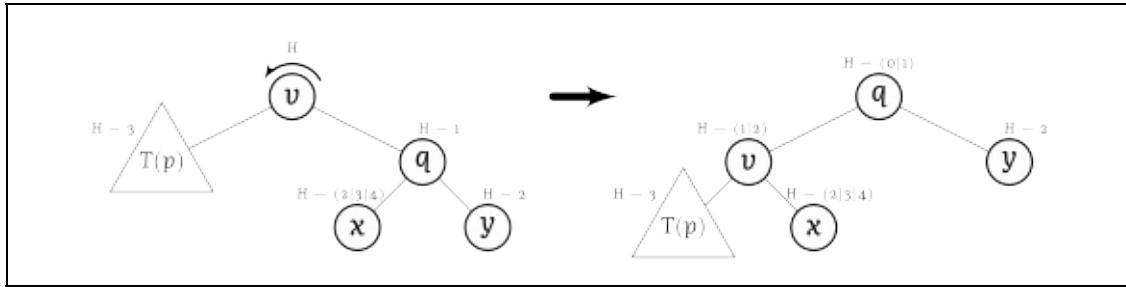


Figura 16.4: Si v tiene un balance de -2 y q tiene un balance de -2 , -1 o 0 , entonces giramos sobre v a la izquierda, lo que deja al subárbol $T(q)$ balanceado.

- Se tiene $b(v)=2$. Este caso es el inverso vertical del caso cuando $b(v)=-2$, pero lo vamos a repasar de cualquier forma, aunque omitiendo ciertas explicaciones que hicimos explícitas en el caso anterior.

Sean p y q los vértices izquierdo y derecho de v , respectivamente. Sea $H = h(v)$; como el balance de v es 2 entonces $h(p)=H-1$ y $h(q)=H-3$.

Si el balance de p es -1 , sean x y y los vértices izquierdo y derecho de p respectivamente; un balance de -1 para p implica que $h(x)-h(y)=-1 \Rightarrow h(y)=h(x)+1$ y por lo tanto $h(y)=h(p)-1=H-2$. Giramos sobre p a la izquierda.

El vértice y cambia su altura a $H-1$ y el vértice p cambia su altura a $H-2$; ningún otro vértice cambia de altura. Sean y_i y y_d los vértices izquierdo y derecho de y respectivamente (pueden ser \emptyset); y_i y y_d pueden tener dos alturas antes del giro; pueden tener altura $H-3$ o $H-4$ (pero alguna debe tener altura $H-3$), porque antes del giro $h(y)=H-2$ y y era balanceado. Las posibles alturas de x y y antes del giro se pueden ver en el cuadro 16.3.

$h(y_i)$	$h(y_d)$	$b(y)$
$H-4$	$H-3$	-1
$H-3$	$H-3$	0
$H-3$	$H-4$	1

Cuadro 16.3: Las posibles alturas de los vértices y_i y y_d y el balance del vértice y antes de girar sobre p a la izquierda.

Como las alturas de y_i , y_d y x no se ven afectadas por el giro sobre p a la

izquierda, la altura de y_d es $H - 4$ o $H - 3$. De esto se sigue que el balance de y después de girar sobre q sea 0, 1 o 2 (figura 16.5).

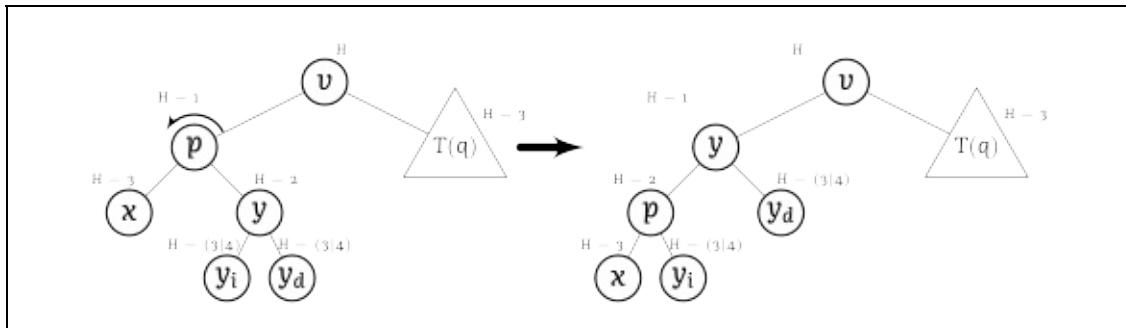


Figura 16.5: Si v tiene un balance de 2 y p tiene un balance de -1, entonces giramos sobre p a la izquierda, lo que deja al vértice y con un balance de 0, 1 o 2.

Si realizamos el giro sobre p renombramos nuestros vértices para que de nuevo p sea el hijo izquierdo de v , x el hijo izquierdo de p y y el hijo derecho de p .

Sea $H = h(v)$, la altura del vértice v en este punto del algoritmo. En este punto del caso cuando el balance de v es 2, su hijo izquierdo p tiene balance 0, 1 o 2 y su altura es $H - 1$, hayamos o no girado sobre p y renombrado.

Como la altura de p es $H - 1$ y su balance es 0, 1 o 2, la altura del vértice x sólo puede ser $H - 2$ como se muestra en el cuadro 16.4. La altura del vértice y puede ser $H - 2$, $H - 3$ o $H - 4$.

$h(x)$	$h(y)$	$b(p)$
$H - 2$	$H - 2$	0
$H - 2$	$H - 3$	1
$H - 2$	$H - 4$	2

Cuadro 16.4: Las posibles alturas de los vértices x y y y el balance del vértice p antes de girar sobre v a la derecha.

Giramos sobre el vértice v a la derecha; los únicos vértices que cambian de altura son v y p en el subárbol $T(p)$. La altura de v dependerá del vértice y : si $h(y)=H - 2$, la altura de v será $H - 1$; si $h(y)=H - 3$, la altura de v será $H - 2$. El vértice q se mantiene con altura $H - 3$ y no depende de él la altura de v .

La altura del vértice p dependerá de la altura de v ; si $h(v)=H - 1$, la altura de p será H . Si $h(v)=H - 2$, como $h(x)=H - 2$ también, la altura de p será $H - 1$.

Cualquiera que sea el caso, el balance del vértice p será $(H - 2) - (H - 2) = 0$ si $h(y) = H - 2$ o $(H - 2) - (H - 3) = 1$ si $h(y) = H - 3$. Por lo tanto el subárbol $T(p)$ queda balanceado (figura 16.6).

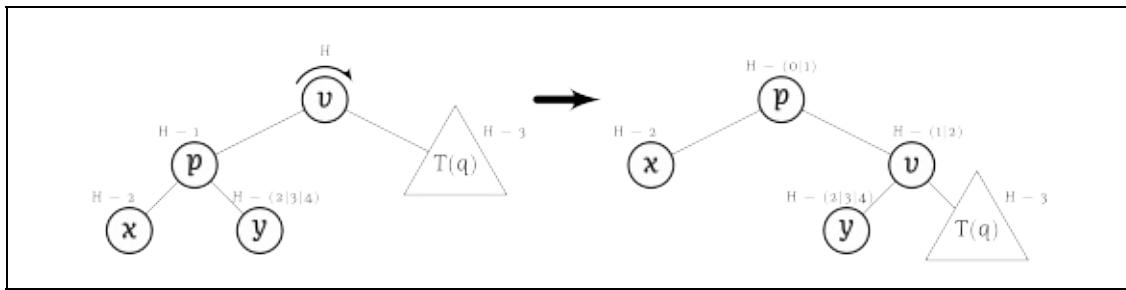


Figura 16.6: Si v tiene un balance de 2 y p tiene un balance de 0, 1 o 2, entonces giramos sobre v a la derecha, lo que deja al subárbol $T(p)$ balanceado.

Si nada más estuviéramos agregando un elemento, podríamos terminar al momento de llegar a un vértice con balance 0 (no sería necesario llegar a la raíz). Sin embargo, como vamos a usar el mismo algoritmo de rebalanceo para los algoritmos de agregación y de eliminación, tenemos que continuar haciendo recursión sobre el padre de v . Esto es porque al eliminar, aunque el subárbol local ya esté balanceado, es posible que la altura de los antecesores haya sido modificada y por lo tanto también su balance.

Los árboles AVL no son tan utilizados como los rojinegros, a pesar de que fueron inventados antes y sean más sencillos de implementar. Pero es un buen ejemplo de una alternativa para mantener nuestros árboles binarios ordenados balanceados.

Con esto terminamos árboles binarios, al menos los del estilo que extienden a la clase `ArbolBinario`; veremos una implementación alterna de árboles binarios en el capítulo 18.

Ejercicios

1. Implementa los métodos faltantes de la clase `ArbolAVL`.
2. Dados un árbol rojinegro y un árbol AVL, ambos con n elementos, ¿cuál tiene mayor altura?
3. ¿Cuál es la complejidad en tiempo de agregar, eliminar y buscar un elemento en un árbol AVL?
4. Dado el árbol AVL en la figura 16.7, agrégale el elemento 5.

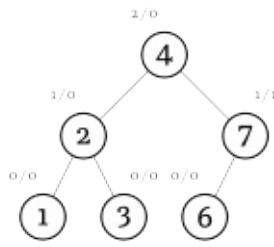


Figura 16.7: Árbol AVL.

5. Dado el árbol AVL resultante de la pregunta anterior, elimínale el elemento 4.

17. Gráficas

Las gráficas (*graphs* en inglés¹) nos permiten modelar relaciones entre pares de elementos en un conjunto. En ese sentido todas las estructuras de datos que hemos visto a lo largo de este libro (con la posible excepción de arreglos, lo cual es discutible) son gráficas. En una lista doblemente ligada dos nodos están relacionados si y sólo si uno es el anterior del otro y el otro es siguiente del uno; en un árbol binario dos vértices están relacionados si y sólo si un vértice es padre del otro y el otro es hijo izquierdo o derecho del uno, etcétera.

Las gráficas liberan todas las restricciones estructurales que le hemos impuesto a nuestras estructuras, dejando únicamente la relación de vecino en sí misma y además permitiendo que un vértice pueda ser vecino de cuantos vértices queramos, incluyendo ninguno si así se desea (aunque no permitiremos que un vértice sea vecino de sí mismo).

Otra diferencia importante es que los elementos de una gráfica serán un conjunto, no una colección. No vamos a permitir elementos repetidos en nuestras gráficas; ni tampoco multiaristas (que dos elementos estén relacionados múltiples veces). Las gráficas de cualquier modo serán una colección, porque un conjunto es una colección trivialmente; pero aunque nuestra clase `grafica` implemente la interfaz `colección`, no será como colecciones la utilización primaria que les daremos.

Desde un punto de vista técnico, implementar la clase `grafica` será relativamente sencillo; casi todos sus algoritmos serán bastante directos y los más complicados (los recorridos) serán de cualquier manera conceptualmente simples. Especialmente si reutilizamos las estructuras de datos que hemos definido con anterioridad.

Antes de definir las gráficas debemos recordar que el que se reconoce como el primer artículo en teoría de gráficas (de Euler sobre los puentes de Königsberg [13]) fue publicado en 1736; el material y resultados que se han acumulado en el área bastan y sobran para cubrir varios cursos de nivel licenciatura. En este capítulo cubriremos apenas lo indispensable para poder implementar gráficas y los recorridos BFS y DFS, además de sentar las bases para poder implementar el algoritmo de Dijkstra en el capítulo 19. Un estudio de teoría de gráficas más profundo corresponde a un cuso especializado, que en casi cualquier programa de Ciencias de la Computación del mundo será llamado Teoría de Gráficas, Gráficas y Juegos o algún nombre similar.

17.1. Definición de gráficas

La definición clásica y estándar de gráficas será la que utilizaremos:

Definición 17.1. (Gráficas) Una gráfica $G = (V, E)$ es un conjunto V de vértices y un conjunto $E \subseteq V \times V$ de aristas, donde para toda $(u, v) \in E$ se cumple que $(v, u) \in E$ y $u \neq v$.

Dado que $(u, v) \in E$ si y sólo si $(v, u) \in E$, en general sólo mencionaremos una combinación del par. Por lo tanto si definimos $E = \{(u, v)\}$ será implícito que $E = \{(u, v), (v, u)\}$.

Como no hay realmente una restricción estructural en los conjuntos de vértices y aristas, más allá de que son conjuntos y que no puede haber una arista de un vértice a sí mismo, no nos ayuda demasiado para implementar las gráficas el definir primero los vértices. Además, técnicamente, en las gráficas los elementos *son* los vértices; pero esto es relativamente complicado de implementar, así que en nuestra implementación los vértices de las gráficas tendrán un elemento y un conjunto de vértices que serán sus vecinos.

Las gráficas suelen representarse como se muestra en la figura [17.1](#).

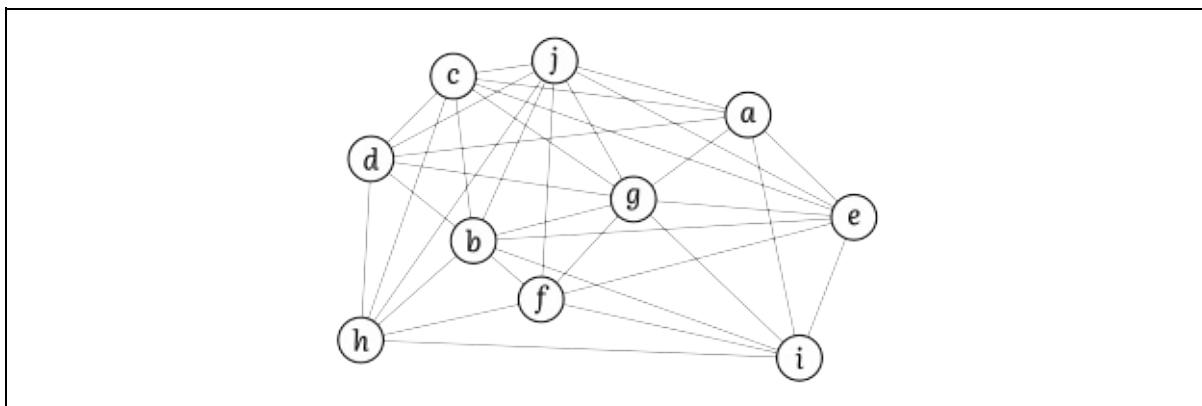


Figura 17.1: Gráfica $G = (V = \{a, b, c, d, e, f, g, h, i, j\}, E = \{(a, c), (a, d), (a, e), (a, g), (a, i), (a, j), (b, c), (b, d), (b, e), (b, f), (b, g), (b, h), (b, i), (b, j), (c, d), (c, e), (c, g), (c, h), (c, j), (d, g), (d, h), (d, j), (e, f), (e, g), (e, i), (e, j), (f, g), (f, h), (f, i), (f, j), (g, i), (g, j), (h, i), (h, j)\})$.

Como hicimos con los árboles binarios en el capítulo [12](#), antes de ver cómo definiremos en Java a nuestra clase, vamos a ver algunas propiedades de las gráficas.

17.2. Propiedades de gráficas

Como las gráficas no tienen ninguna restricción respecto a las relaciones entre sus elementos, esto en particular permite que $E = \emptyset$ o que $E = (V \times V) \setminus \{(v, v) : v \in V\}$, con todas las posibilidades intermedias. Esto

resultará en que a partir de un vértice u en una gráfica, no siempre podemos llegar a un vértice v moviéndonos a través de las aristas en E .

Esta propiedad de las gráficas será fundamental, pero para definirla vamos a necesitar un par de definiciones auxiliares.

Definicion 17.2. (Caminos en gráficas) *Sea $G = (V, E)$ una gráfica.*

Un camino en G será una secuencia v_1, v_2, \dots, v_k tal que $v_i \in V$, $1 \leq i \leq k$ y donde $(v_{i-1}, v_i) \in E$ para $1 < i \leq k$.

Por ejemplo, la gráfica de la figura 17.1 tiene el camino h, d, b, j, g, a, i, e porque $\{(h, d), (d, b), (b, j), (j, g), (g, a), (a, i), (i, e)\} \subset E$ (figura 17.2).

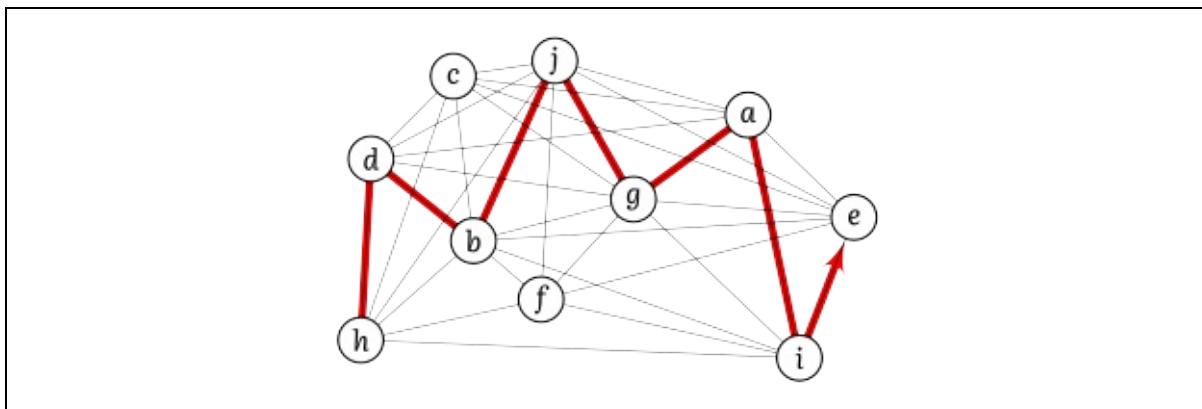


Figura 17.2: El camino h, d, b, j, g, a, i, e en G , que también es trayectoria.

En principio un camino puede tener vértices repetidos, siempre y cuando el vértice repetido no sea consecutivo dado que (v, v) nunca está en E . En general nos interesarán los caminos que no tengan vértices repetidos, a los que llamaremos trayectorias.

Definicion 17.3. (Trayectorias en gráficas) *Sean $G = (V, E)$ una gráfica y $T = v_1, v_2, \dots, v_k$ un camino en G . Diremos que T es una trayectoria en G si y sólo si $v_i \neq v_j$ para todos los índices $i \neq j$.*

Podemos definir la longitud de una trayectoria de la manera obvia.

Definicion 17.4. (Longitud de trayectorias) *Sea $G = (V, E)$ una gráfica. Sea $T = v_1, \dots, v_k$ una trayectoria en G ; la longitud de T , denotada por $\ell(T)$, es $k - 1$, el número de vértices en la trayectoria menos uno.*

Tanto los caminos como las trayectorias pueden tener longitud 0, en cuyo caso tienen únicamente un vértice de la gráfica; siendo consistentes entonces, una trayectoria vacía tendrá longitud -1. Los caminos y trayectorias de longitud 0 o -1 no son terriblemente útiles, pero los vamos a permitir.

Con estas definiciones podemos continuar con la definición de conexidad.

Definicion 17.5. (Conexidad en gráficas) *Sea $G = (V, E)$ una gráfica.*

Diremos que G es conexa si y sólo si para todo par de vértices $u \neq w$ de G , existe una trayectoria $u = v_1, v_2, \dots, v_k = w$ en G . Si una gráfica no es conexa, diremos que es inconexa.

Si una gráfica es inconexa nos va a interesar en cuántas partes conexas se divide. Para eso primero necesitaremos definir lo que es una subgráfica.

Definicion 17.6. (Subgráficas) *Sea $G = (V, E)$ una gráfica. Una subgráfica $H = (V_H, E_H)$ de G será una gráfica tal que $V_H \subseteq V$ y $E_H \subseteq E \cap (V_H \times V_H)$.*

En particular y aunque no sea particularmente útil, una gráfica G siempre es subgráfica de sí misma. Con subgráficas podemos definir las componentes conexas de una gráfica.

Definicion 17.7. (Componentes conexas de una gráfica) *Sea $G = (V, E)$ una gráfica. Una componente conexa de G será una subgráfica $H = (V_H, E_H)$ tal que H es conexa y además, para todo par de vértices u, v en V tales que $u \in V \setminus V_H$ y $v \in V_H$, se cumple que $(u, v) \notin E$ y $(v, u) \notin E$.*

En otras palabras, no sólo H debe ser conexa, sino que no debe haber en G aristas que conecten vértices de H con vértices que no estén en H . Dada la definición [17.7](#) toda gráfica conexa G tiene una única componente conexa, que es ella misma.

Como mencionábamos al inicio de la sección, el conjunto E de aristas puede variar desde ser \emptyset hasta ser $(V \times V) \setminus \{(v, v) : v \in V\}$. Esto quiere decir que las gráficas pueden ocupar $O(1)$, $O(n)$ u $O(n^2)$ de memoria (si $|V|=n$) y esto evidentemente podrá afectar la complejidad de nuestros algoritmos.

Hay una familia de gráficas muy importante que siempre tienen un número lineal de aristas, que será nuestra siguiente definición.

Definicion 17.8. (Gráficas planas) *Sea $G = (V, E)$ una gráfica. Diremos que G es una gráfica plana (o que es planar) si y sólo si podemos dibujar la gráfica en el plano de tal forma que no haya aristas que se crucen.*

Las aristas no tienen por qué dibujarse con líneas rectas; se pueden utilizar curvas, aunque por el teorema de Fáry [\[14\]](#) si la gráfica es plana siempre se puede hacer un dibujo de la misma con líneas rectas. Una gráfica plana tiene a lo más $3n - 6$ aristas ($n > 2$); esto será importante porque entonces toda gráfica plana utilizará $O(n)$ de memoria. En la figura [17.3](#) mostramos una subgráfica plana.

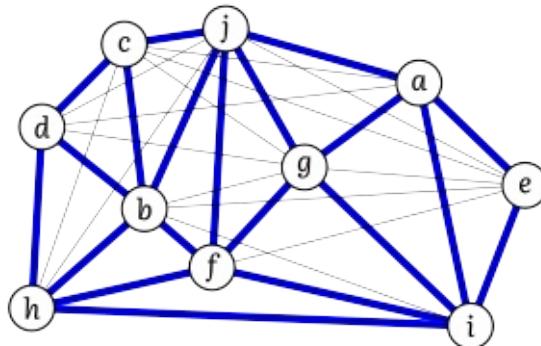


Figura 17.3: La subgráfica $H = (\{a, b, c, d, e, f, g, h, i\}, \{(a, e), (a, g), (a, i), (a, j), (b, c), (b, d), (b, f), (b, h), (b, j), (c, d), (c, j), (d, h), (e, i), (f, g), (f, h), (f, i), (f, j), (g, i), (g, j), (h, i)\})$ es plana.

Los vértices en general no tendrán muchas propiedades por sí mismos; básicamente podremos colorearlos (vamos a necesitar esto para poder recorrer las gráficas) y recorrer sus vecinos. Relacionado con eso será el grado de un vértice.

Definicion 17.9. (Grado de un vértice) *Sea $G = (V, E)$ una gráfica. Dado un vértice $v \in V$ definiremos el grado de un vértice como la cardinalidad del conjunto $\{(v, w) : w \in V \text{ y } (v, w) \in E\}$.*

17.3. Implementación en Java

Para implementar de manera más natural una gráfica necesitaríamos una implementación de conjuntos (y de hecho, más práctico, de diccionarios); pero estas estructuras de datos no las veremos hasta los capítulos 22 y 21, respectivamente. A falta de ellas utilizaremos nuestra colección más sencilla: nuestros conjuntos estarán modelados por una lista.

Esto afecta negativamente las complejidades en tiempo de varios de los algoritmos de la clase `Grafica`; buscar un vértice dado su elemento nos tomará tiempo $O(n)$ y lo mismo ocurrirá para saber si dos vértices están relacionados. Peor aún, construir una gráfica (incluso con un número lineal de aristas) *siempre* tomará tiempo $O(n^2)$, porque al agregar un elemento hay que revisar que el elemento no esté ya contenido en la gráfica (los vértices son un conjunto) y entonces agregar un elemento tomará tiempo $O(n)$. Por este motivo, en cuanto tengamos las estructuras de datos necesarias regresaremos a la clase `Grafica` para que sus algoritmos tengan las complejidades en tiempo y espacio que les corresponde.

De forma análoga a los árboles binarios, tendremos una interfaz `VerticeGrafica` que implementará la clase `Vertice` interna y privada a `Grafica`. Lo más importante de la interfaz `VerticeGrafica` es que nos permitirá obtener un iterable

para los vecinos del vértice; además tendrá las operaciones básicas para obtener el elemento del vértice, su grado (el número de vecinos) y su color.

La clase `Vertice` interna y privada a `Grafica` tendrá un elemento genérico, un color y una lista de vértices vecinos (a falta de algo mejor, como conjuntos). La clase `Grafica` a su vez tendrá una lista de vértices (a falta de conjuntos) y un contador interno para el número de aristas en la misma. Además y como `Grafica` implementará `colección`, tendremos una clase interna y privada `Iterador`, que será probablemente la más sencilla que hayamos escrito hasta el momento.

Además de las operaciones obligadas por implementar la interfaz `colección`, la gráfica tendrá operaciones para obtener el número de aristas, para conectar y desconectar dos elementos de la misma, para consultar si dos elementos están conectados, para obtener una instancia de `VerticeGrafica` a partir de un elemento, para cambiar el color de un vértice y para recorrer los vértices de la gráfica en el orden en que fueron agregados los elementos, en orden BFS y en orden DFS.

Por último, la clase tendrá un contador interno para contabilizar el número de aristas en la gráfica. El esqueleto de la clase `Grafica` se puede ver en el listado 17.1.

```
public class Grafica<T> implements Colección<T> {

    private class Iterador implements Iterator<T> {
        private Iterator<Vertice> iterador;
        public Iterador() { /* ... */ }
        @Override public boolean hasNext() { /* ... */ }
        @Override public T next() { /* ... */ }
    }

    private class Vertice implements VerticeGrafica<T> {
        public T elemento;
        public Color color;
        public Lista<Vertice> vecinos;
        public Vertice(T elemento) { /* ... */ }
        @Override public T get() { /* ... */ }
        @Override public int getGrado() { /* ... */ }
        @Override public Color getColor() { /* ... */ }
        @Override public
        Iterable<? extends VerticeGrafica<T>> vecinos() {
            /* ... */
        }
    }

    private Lista<Vertice> vertices;
    private int aristas;
```

```

public Grafica() { /* ... */ }
public int getAristas() { /* ... */ }
public void conecta(T a, T b) { /* ... */ }
public void desconecta(T a, T b) { /* ... */ }
public boolean sonVecinos(T a, T b) { /* ... */ }
public VerticeGrafica<T>
vertice(T elemento) { /* ... */ }
public void setColor(VerticeGrafica<T> vertice,
                     Color color) { /* ... */ }
public boolean esConexa() { /* ... */ }
public void
paraCadaVertice(AccionVerticeGrafica<T> accion) {
    /* ... */
}
public void
bfs(T elemento, AccionVerticeGrafica<T> accion) {
    /* ... */
}
public void
dfs(T elemento, AccionVerticeGrafica<T> accion) {
    /* ... */
}

@Override public int getElementos() { /* ... */ }
@Override public void agrega(T elemento) { /* ... */ }
@Override public boolean
contiene(T elemento) { /* ... */ }
@Override public void elimina(T elemento) { /* ... */ }
@Override public boolean esVacia() { /* ... */ }
@Override public void limpia() { /* ... */ }
@Override public String toString() { /* ... */ }
@Override public boolean equals(Object o) { /* ... */ }
@Override public Iterator<T> iterator() {
    return new Iterador();
}
}
}

```

Listado 17.1: Esqueleto de la clase **Grafica**, con el método `iterator()` ya escrito.

Antes de ver los algoritmos correspondientes a los métodos de la clase **Grafica**, vamos revisar las maneras clásicas en que podemos recorrer una gráfica.

17.4. Recorridos en gráficas

Para recorrer nuestras gráficas tenemos tres cosas que hay que tomar en cuenta: en primer lugar, no tenemos un punto de partida natural. Contrario a las listas (donde teníamos la cabeza) o los árboles binarios (donde teníamos la

raíz), en una gráfica no hay un elemento de la misma que obviamente sirva como primer elemento para recorrerla. Siempre tendremos que recibir como entrada del algoritmo el primer elemento para recorrer la gráfica.

En segundo lugar, tendremos que marcar los vértices mientras recorremos la gráfica. Como no hay una jerarquía u orden en las relaciones entre pares de elementos, si no marcamos un vértice que ya recorrimos no hay manera de saber si ya pasamos por él; no es como en las listas donde si las recorremos siguiendo los nodos siguientes nunca podremos repetirlos; o los árboles binarios donde si siempre bajamos (ya sea por el hijo izquierdo o derecho) o siempre subimos por el padre, entonces es imposible repetir vértices. Es por esto que nuestros vértices podrán ser coloreables; usaremos colores para determinar si ya visitamos un vértice. Relacionado a esto, también debe quedar claro que aunque no visitemos múltiples veces un vértice, todas las aristas serán visitadas (dos veces, de hecho). Veremos esto más adelante.

En tercer y último lugar, si la gráfica es inconexa nuestros algoritmos de recorrido no podrán recorrerla toda. Usaremos las aristas de la gráfica para desplazarnos de un vértice a otro, así que únicamente podremos recorrer la componente conexa a la que pertenezca el primer elemento del recorrido.

También existe un detalle de implementación; dado el vértice actual del algoritmo, comenzaremos a recorrer sus vértices vecinos. El orden en que el vértice nos presente sus vecinos determinará el orden en que el algoritmo recorra los vértices. En nuestra implementación en Java esto estará determinado por el orden en que *conectemos* los vértices, porque en ese orden se agregarán los vértices a la lista de vecinos de un vértice en particular.

En el libro vamos a suponer que los vecinos de un vértice están ordenados lexicográficamente. Esto es únicamente para tener una manera de poder ver ejemplos concretos de manera que podamos analizar cómo se comporta nuestros algoritmos, pero debe quedar claro que nuestra implementación en Java *no* se comportará necesariamente de esta manera.

Los dos recorridos que veremos en este capítulo son una generalización de los que vimos con árboles binarios; búsqueda por amplitud y búsqueda por profundidad o BFS y DFS como son más conocidos.

17.4.1. BFS en gráficas

Igual que con los árboles binarios, BFS en gráficas recorrerá la gráfica por amplitud: primero se visitarán los vecinos del vértice inicial; después se visitarán los vecinos de los vecinos exceptuando los ya visitados; después los vecinos de los vecinos de los vecinos, etcétera.

Y también como con los árboles binarios, una cola nos servirá para guardar los vecinos del vértice actual y así asegurar que sus vecinos sean visitados antes que los vecinos de los vecinos. Lo único distinto será que usaremos un color para marcar los vértices como visitados; cada vez que metamos un vértice a la cola lo colorearemos antes de **negro** y nunca volveremos a agregar un vértice **negro** a la misma. En otras palabras el color **negro** servirá para marcar a los vértices como visitados. Para garantizar que esto funcione, todos los vértices se colorearán de **rojo** como primer paso del algoritmo.

El algoritmo recibe un elemento, así que lo primero que haremos será buscar el vértice correspondiente al elemento y un error ocurrirá si el elemento no está en la gráfica. Teniendo el vértice w correspondiente al elemento, coloreamos todos los vértices de la gráfica de color **rojo**, creamos una cola Q , coloreamos w de **negro** y lo metemos en Q . Mientras Q no sea vacía sacamos el elemento al frente de la misma y lo guardamos en u ; hacemos con u lo que tengamos planeado en el recorrido (por ejemplo imprimir el elemento en el vértice). Para todo vecino v de u , si v tiene color **rojo** lo coloreamos de **negro** y lo metemos a Q .

Veamos un ejemplo con la gráfica de la figura [17.1](#), comenzando con h . Coloreamos todos los vértices de **rojo**, a h de **negro** y lo metemos en nuestra recién creada cola Q . Sacamos el vértice al frente de Q (h) y recorremos en orden lexicográfico (como dijimos arriba) sus vecinos: b, d, f, i ; en este punto todos son **rojos**, por lo que metemos los vértices en Q en ese orden después de colorearlos de **negro**. Sacamos b de Q y recorremos sus vecinos: c, d, e, f, g, h, i, j en ese orden; los **negros** (d, f, h, i) los ignoramos; los **rojos** (c, e, g, j) los coloreamos de **negro** y los metemos a Q en ese orden. La cola Q en este momento se ve de la siguiente manera:

j	g	e	c	i	f	d
-----	-----	-----	-----	-----	-----	-----

Sacamos el siguiente elemento de Q (d) y recorremos sus vecinos: a, b, c, g, h, j en ese orden; a los **negros** (b, c, g, h, j) los ignoramos y al único **rojo** (a) lo coloreamos de **negro** y lo metemos a Q .

Para este momento *todos* los vértices de la gráfica son **negros**, por lo que ya ningún elemento entrará a Q y recorreremos el resto de los vértices en el orden dado por la cola. El recorrido se puede ver en la figura [17.4](#).

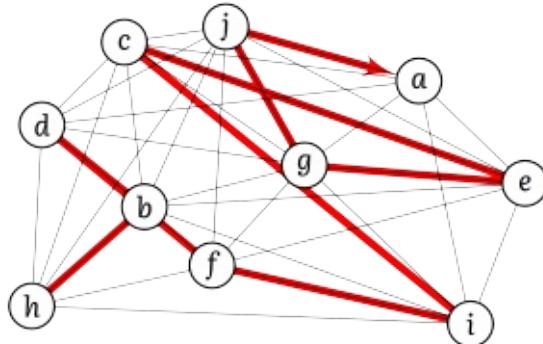


Figura 17.4: Recorrido BFS comenzando por h .

Aunque el orden de la figura 17.4 es el dado por BFS comenzando por h (y suponiendo orden lexicográfico en todas las listas de vecinos), la figura no es de mucha ayuda. Es más útil pensar que BFS genera un árbol, donde un vértice u es padre de un vértice v si v fue agregado a Q mientras recorriáramos los vecinos de u . El árbol correspondiente se puede ver en la figura 17.5.

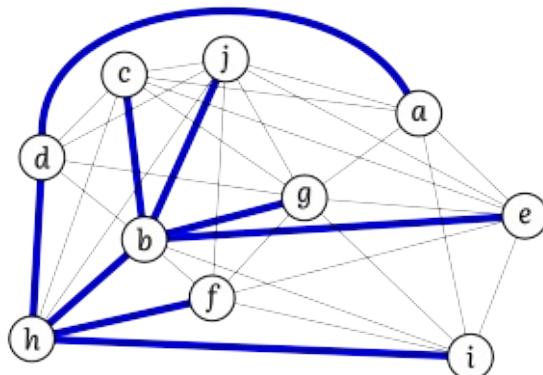


Figura 17.5: Árbol generado por el recorrido BFS comenzando por h ; curveamos la arista (d, a) para que no cruzara otra arista del árbol.

Como mencionábamos arriba, todo vértice entra a la cola Q a lo más una vez (si la gráfica es inconexa habrá vértices que no entran nunca); pero toda arista de la componente conexa es visitada *dos veces*. La arista (u, v) es visitada cuando u sale de la cola, revisa su vecino v , lo pinta de **negro** y lo mete a la cola; y es visitada de nuevo cuando v sale de la cola, revisa su vecino u , ve que es **rojo** y *no* lo mete a la cola.

Esto significa que la complejidad en tiempo de recorrer una gráfica con BFS es $O(|V|+|E|)$; técnicamente esto es cuadrático, porque $|E|$ es $O(n^2)$ en el peor de los casos, pero suele denotarse con $O(|V|+|E|)$ para hacer énfasis en que si la gráfica tiene un número de aristas de orden lineal (por ejemplo, si la gráfica es plana), entonces el algoritmo será también lineal.

En espacio obviamente la complejidad es $O(n)$; en el peor de los casos comenzamos a recorrer la gráfica por un vértice que esté conectado a todos los demás y la cola termina con $n - 1$ elementos.

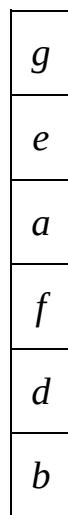
17.4.2. DFS en gráficas

El algoritmo DFS es exactamente igual a BFS, exceptuando por el hecho de que usaremos una pila en lugar de una cola. De todas maneras lo describiremos.

El algoritmo recibe un elemento que buscaremos su vértice correspondiente y habrá un error si no está en la gráfica. Coloreamos todos los vértices de la gráfica de color **rojo**, creamos una pila S , coloreamos el vértice w correspondiente al elemento de **negro** y lo metemos en S . Mientras S no sea vacía sacamos el elemento en el tope de la misma y lo guardamos en u ; hacemos con u lo que tengamos planeado en el recorrido. Para todo vecino v de u , si v tiene color **rojo** lo coloreamos de **negro** y lo metemos a S .

Repitamos el mismo ejemplo que hicimos con BFS; recorramos la gráfica de la figura [17.1](#) con DFS comenzando con h .

Coloreamos todos los vértices de **rojo**, a h de **negro** y lo metemos en nuestra pila S . Sacamos el vértice en el tope de S (h) y recorremos en orden lexicográfico (como dijimos arriba) sus vecinos: b, d, f, i ; en este punto todos son **rojos**, por lo que metemos los vértices en S en ese orden después de colorearlos de **negro**. Sacamos i de S y recorremos sus vecinos: a, b, e, f, g, h en ese orden; los **negros** (b, f, h) los ignoramos; los **rojos** (a, e, g) los coloreamos de **negro** y los metemos a S en ese orden. La pila S en este momento se ve de la siguiente manera:



Sacamos el siguiente elemento de S (g) y recorremos sus vecinos: a, b, c, d, e, f, i, j en ese orden; a los **negros** (a, b, d, e, f, i) los ignoramos y a los

rojos (c, j) los coloreamos de **negro** y los metemos a S .

Para este momento *todos* los vértices de la gráfica son **negros**, por lo que ya ningún elemento entrará a S y recorreremos el resto de los vértices en el orden dado por la pila. El recorrido se puede ver en la figura 17.6.

Una vez más la figura 17.6 no nos da mucha información; definiremos un árbol para el recorrido DFS de la misma manera a como hicimos con BFS, que podemos ver en la figura 17.7.

Como los algoritmos son idénticos excepto por la estructura auxiliar que utilizan (BFS una cola, DFS una pila), DFS también tiene complejidad en tiempo ($|V|+|E|$) y complejidad en espacio $O(n)$.

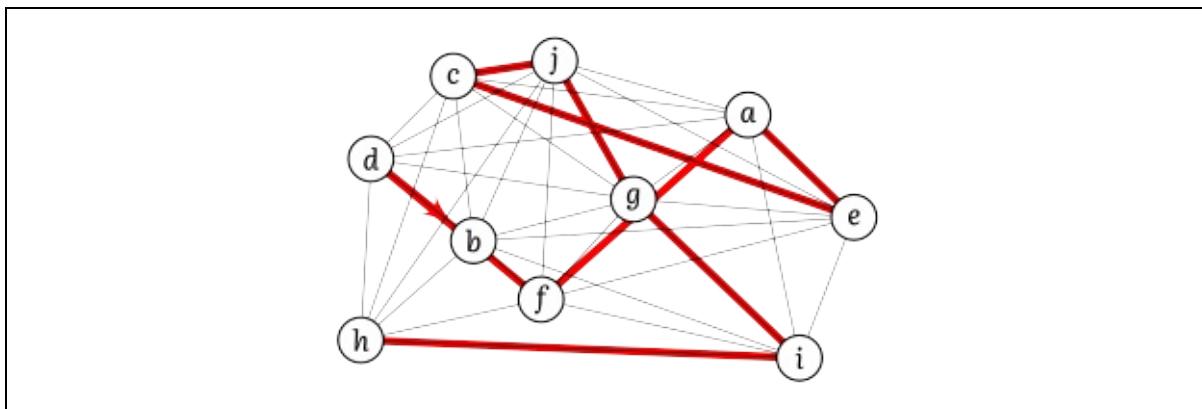


Figura 17.6: Recorrido DFS comenzando por h .

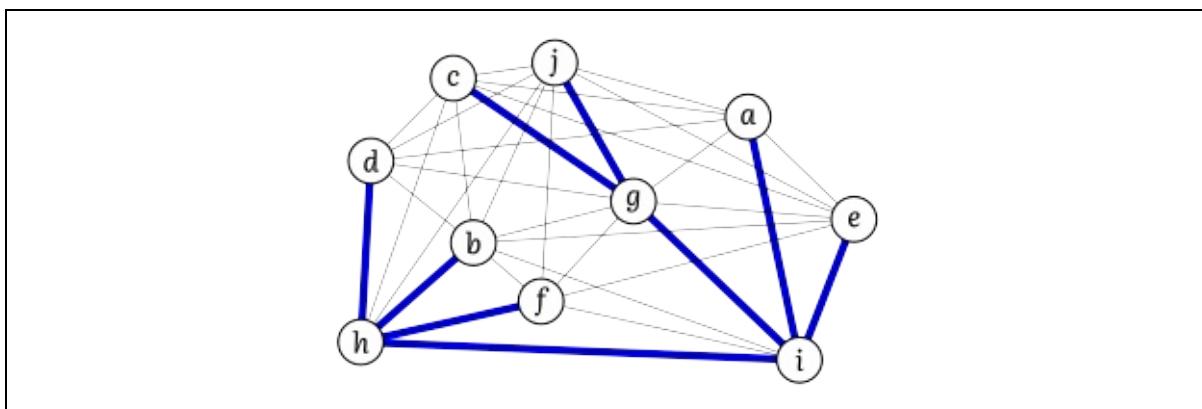


Figura 17.7: Árbol generado por el recorrido DFS comenzando por h .

17.5. Algoritmos para gráficas

Revisaremos los algoritmos para gráficas en el orden dado por el listado 17.1, comenzando por la clase interna privada **iterador**. Esta implementación de **Iterator** será, con casi toda certeza, la más sencilla de todo el libro; no podemos utilizar un recorrido BFS o DFS para iterar la gráfica, porque si ésta es inconexa entonces no se iterarían todos los vértices, además de que no

queda claro qué vértice debería utilizarse como inicial. Por lo tanto sencillamente iteraremos los elementos de la gráfica iterando la lista de vértices.

- `Iterador()`

El constructor crea un iterador de la lista de vértices.

- `hasNext()`

Se manda llamar el método `hasNext()` del iterador.

- `next()`

Se regresa el elemento de llamar `next()` con el iterador.

La clase interna privada `Vertice` implementa la interfaz `VerticeGrafica` y los únicos métodos que tendrá serán los definidos por la interfaz (y su constructor). En añadidura, todos los métodos únicamente regresan el estado del vértice, así que terminan siendo métodos de una línea donde se hace `return` de alguna propiedad del vértice.

- `Vertice()`

El constructor define el elemento del vértice como el elemento recibido, el color del vértice como `Color.NINGUNO` y se crea la lista de vértices vecinos.

- `get()`

Se regresa el elemento del vértice.

- `getGrado()`

Se regresa la longitud de la lista de vértices vecinos.

- `getColor()`

Se regresa el color del vértice.

- `vecinos()`

Se regresa la lista de vecinos. Un detalle técnico acerca de este método; su firma la podemos ver en el listado [17.2](#).

```
@Override public
Iterable<? extends VerticeGrafica<T>> vecinos() {
    /* ... */
}
```

Listado 17.2: La firma del método `vecinos()`.

En primer lugar regresamos una instancia de `Iterable`, no de `Lista`. Esto es para poder iterar los vecinos de forma sencilla; pero además de esta manera el objeto se vuelve de sólo lectura: no podemos modificar la lista de vecinos si la estamos viendo como un iterable.

En segundo lugar utilizamos un acotamiento del genérico con un comodín acotado a `VerticeGrafica`. Esto nos permite regresar la lista de vecinos, que es de tipo `Lista<Vertice>`, pero la misma se ve de tipo `Lista<VerticeGrafica>` fuera de la clase. Esto nos mantiene el encapsulamiento de datos (los vértices son una clase interna privada) sin necesidad de copiar la lista de vecinos. Como la interfaz `VerticeGrafica` es de sólo lectura, esto nos permite recorrer los vecinos del vértice de manera segura.

Habiendo revisado los algoritmos de sus clases internas y privadas, procedemos con los métodos de la clase `Grafica`.

- `Grafica()`

El constructor inicializa la lista de vértices de la gráfica.

- `getAristas()`

Regresa el contador interno de aristas.

- `conecta()`

Se buscan los vértices correspondientes a los elementos recibidos; si alguno no está en la gráfica un error ocurre. Si los dos vértices son iguales también ocurre un error porque no podemos conectar un vértice a sí mismo. Si los vértices ya son vecinos (se puede reutilizar `sonVecinos()`) también ocurrirá un error; no permitiremos tratar de conectar dos vértices ya conectados.

Si llegamos a este punto ya es seguro conectar los vértices; agregamos el primer vértice a la lista de vecinos del segundo y viceversa. Además incrementamos nuestro contador interno de aristas.

- `desconecta()`

Se buscan los vértices correspondientes a los elementos recibidos; si alguno no está en la gráfica un error ocurre. Si los vértices no están conectados (de nuevo podemos usar `sonVecinos()`) un error ocurre.

Eliminamos el primer vértice de la lista de vecinos del segundo y viceversa. Además decrementamos el contador interno de aristas.

- `sonVecinos()`

Se buscan los vértices correspondientes a los elementos recibidos; si alguno no está en la gráfica un error ocurre. Se revisa que el primer vértice esté en la lista de vecinos del segundo y viceversa.

- `vertice()`

El método regresa el vértice correspondiente a un elemento, como instancia de `VerticeGrafica`. Si el elemento no está en la gráfica ocurre un error.

- `setColor()`

El método recibe un vértice como instancia de `VerticeGrafica` y un color. Hacemos una audición para tener el vértice como instancia de `Vertice`; si el vértice no es instancia de `Vertice` (en Java usaremos el método `getclass()` para verificarlo, como en el método `equals()`) un error ocurre. Definimos el color del vértice como el color recibido por el método.

Se puede discutir si este método debería pertenecer a la interfaz `VerticeGrafica` y por lo tanto a la clase `Vertice`; la razón para moverlo a la clase `Grafica` es para mantener la interfaz `VerticeGrafica` como de sólo lectura.

- `esConexa()`

Lo único que tiene que hacer el algoritmo es recorrer la gráfica con BFS (o DFS, no importa); al terminar, si todos los vértices son **rojos**, entonces la gráfica es conexa. De otra manera la gráfica es inconexa.

- `paraCadaVertice()`

El método recibe una instancia de `AccionVerticeGrafica`, que al igual que `AccionVerticeArbolBinario` es una interfaz funcional para que podamos realizar una acción sobre todos los vértices de la gráfica.

El método únicamente recorre la lista de vértices y ejecuta la acción sobre cada uno de ellos.

- `bfs()`

El método recibe un elemento y una instancia de `AccionVerticeGrafica`. Se busca el vértice correspondiente al elemento recibido y se ejecuta el algoritmo visto en la sección [17.4.1](#); cada vez que sacamos un vértice de la cola ejecutamos la acción sobre el mismo.

Este método junto con `dfs()` pueden utilizar el mismo código si se escribe un método auxiliar que reciba una instancia de la clase `MeteSaca`; este método llamaría el método auxiliar con una instancia de `cola`.

- `dfs()`

El método recibe un elemento y una instancia de `AccionVerticeGrafica`. Se busca el vértice correspondiente al elemento recibido y se ejecuta el algoritmo visto en la sección [17.4.2](#); cada vez que sacamos un vértice de la pila ejecutamos la acción sobre el mismo.

Este método junto con `bfs()` pueden utilizar el mismo código si se escribe un método auxiliar que reciba una instancia de la clase `MeteSaca`; este método llamaría el método auxiliar con una instancia de `Pila`.

- `getElementos()`

Sencillamente regresamos la longitud de la lista de vértices.

- `agrega()`

Si el elemento recibido es \emptyset o ya está en la gráfica (se puede utilizar `contiene()`) un error ocurre.

Si no creamos un nuevo vértice con el elemento y lo agregamos a la lista de vértices.

- `contiene()`

Buscamos el vértice que contenga el elemento recibido; si está regresamos verdadero, si no regresamos falso.

- `elimina()`

Buscamos el vértice correspondiente al elemento recibido; si no existe ocurre un error. Si sí eliminamos el vértice de la lista de vértices.

Además recorremos los vecinos del vértice y lo eliminamos de las listas de vecinos de los vecinos; decrementamos el contador interno de aristas por cada vecino.

- `esVacia()`

Regresamos si la lista de vecinos es vacía o no.

- `limpia()`

Limpiamos la lista de vértices y actualizamos el contador interno para aristas a cero.

- `toString()`

Vamos a regresar la cadena con el conjunto de elementos y el de aristas como los hemos mostrado en las figuras del capítulo. Por ejemplo, tomemos la gráfica K_3 , que es la gráfica *completa* (con todas las posibles aristas) de tamaño 3 en la figura [17.8](#).

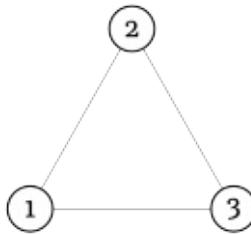


Figura 17.8: Gráfica K_3 .

La representación en cadena de esta gráfica será como se muestra en la figura [17.9](#).

"{1, 2, 3, }, {(1, 2), (1, 3), (2, 3), }"

Figura 17.9: Representación en cadena de una gráfica.

Como cada arista (u, v) la tenemos dos veces en nuestra implementación (una vez en el vértice u con v en su lista de vecinos; y otra vez en el vértice v con u en su lista de vecinos), no queremos que " (u, v) " y " (u, v) " aparezcan ambas en la cadena. Si u está antes que v en la lista de vértices, entonces nada más debe aparecer " (u, v) ".

Esta representación no es muy útil que digamos, cualquier gráfica mediana es ilegible si sólo vemos sus conjunto de vértices y aristas (como se puede ver en la figura [17.1](#)). Agregamos el método sólo por completez, dado que todas las estructuras de datos del libro lo tienen.

- `equals()`

Tenemos que hacer de nuevo una audición en una clase genérica y suprimir la advertencia resultante (listado [17.3](#)).

```

@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    Grafica<T> grafica = (Grafica<T>)o;
    // ...
}

```

Listado 17.3: Inicio del método `equals()` en `Grafica`.

Después verificamos que el número de elementos y aristas en ambas gráficas sea el mismo, que los mismos elementos estén en ambas gráficas (aunque no necesariamente en el mismo orden) y que si dos elementos están conectados en una gráfica, también lo estén en la otra.

Si todo esto se cumple regresamos verdadero, de otra forma regresamos falso. El método tiene una complejidad en tiempo *terrible*: $O(n^3)$, si lo hacemos con cuidado recorriendo la lista de vecinos de cada vértice; $O(n^4)$ si somos descuidados y sencillamente verificamos par a par cada posible arista de la gráfica. En el capítulo [23](#) veremos cómo mejorar la complejidad en tiempo.

17.6. Implementaciones alternativas de gráficas

Nuestra implementación de gráficas es conocida como lista de adyacencias, porque cada vértice tiene una lista de vértices vecinos que modelan las adyacencias del mismo.

Existe otra implementación alternativa para gráficas; matriz de adyacencias, donde utilizamos una matriz (un arreglo bidimensional) para modelar las adyacencias de los vértices. Por ejemplo, la gráfica de la figura [17.1](#) tendría la matriz de adyacencias que se ve en el cuadro [17.1](#).

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>
<i>a</i>	0	0	1	1	1	0	1	0	1	1
<i>b</i>	0	0	1	1	1	1	1	1	1	1
<i>c</i>	1	1	0	1	1	0	1	1	0	1
<i>d</i>	1	1	1	0	0	0	1	1	0	1
<i>e</i>	1	1	1	0	0	1	1	0	1	1
<i>f</i>	0	1	0	0	1	0	1	1	1	1
<i>g</i>	1	1	1	1	1	1	0	0	1	1
<i>h</i>	0	1	1	1	0	1	0	0	1	1
<i>i</i>	1	1	0	0	1	1	1	1	0	0
<i>j</i>	1	1	1	1	1	1	1	1	0	0

Cuadro 17.1: La matriz de adyacencias de la gráfica en la figura [17.1](#).

La matriz de adyacencias tiene como renglones y columnas los elementos de la gráfica y en cada entrada hay un 1 si los dos elementos correspondientes

están conectados; y 0 si no lo están.

Implementar gráficas con una matriz de adyacencias es más sencillo que con lista de adyacencias; pero potencialmente desperdicia mucha memoria si la gráfica tiene un número lineal de aristas y además afecta las complejidades en tiempo de nuestros algoritmos: para recorrer los vecinos de un vértice, tenemos que recorrer toda la columna o renglón correspondiente al vértice.

Además las matrices de adyacencias tienen los mismos problemas que todos los arreglos tienen; no podemos agregar un número arbitrario de vértices y la estructura se vuelve estática o tenemos que perder tiempo si queremos que sea dinámica.

Para gráficas muy densas (con muchas aristas) y no muy dinámicas tiene sentido utilizar una matriz de adyacencias; pero en el caso general una implementación con listas de adyacencias siempre será la mejor opción.

No hemos terminado con gráficas; en el capítulo [19](#) agregaremos pesos a sus aristas e implementaremos algoritmos para encontrar trayectorias entre pares de vértices y en el capítulo [21](#) utilizaremos diccionarios para poder encontrar el vértice asociado a un elemento en tiempo $O(1)$.

Ejercicios

1. Implementa los métodos faltantes de la clase `Grafica`.
2. Dada una gráfica de n vértices, ¿cuál es el número máximo de aristas que puede tener?
3. Dada una gráfica *plana* de n vértices, ¿cuál es el número máximo de aristas que puede tener?
4. Dada una gráfica conexa de n vértices, ¿cuál es el número *mínimo* de aristas que puede tener?
5. Al ejecutar el algoritmo BFS en una gráfica conexa, ¿cuántas veces es visitada cada arista?

1. Algunas personas insisten en llamar a las gráficas *grafos* en español. Nosotros ignoraremos esta obvia equivocación y utilizaremos el término preferido por la mayor parte de la gente en México que de hecho se dedica al estudio de las gráficas.[←](#)

18. Montículos mínimos

Los montículos mínimos (*min-heaps* en inglés) son árboles binarios completos, lo cual podría llevarnos a pensar que los árboles binarios completos que vimos en el capítulo [13](#) nos bastarían; pero los montículos mínimos serán árboles binarios completos implementados con arreglos. Por supuesto podríamos implementarlos extendiendo la clase `ArbolBinarioCompleto` (con ciertas modificaciones), pero entonces perderíamos la principal ventaja que nos dan los montículos: que podemos construir un montículo con n elementos en tiempo $O(n)$.

Dado lo anterior debe quedar claro que los montículos mínimos *no están ordenados*; tendrán elementos comparables entre ellos, pero no estarán ordenados. No pueden estarlo si los podemos construir en tiempo $O(n)$, porque ordenar una colección de elementos nos toma al menos tiempo $O(n \log n)$ usando comparaciones.

18.1. Definición de montículos mínimos

Definiremos a los montículos mínimos de la siguiente manera:

Definicion 18.1. (Montículos mínimos) *Un montículo mínimo (o simplemente montículo si el contexto no es ambiguo¹) es un árbol binario completo donde para todo vértice v , se cumple que si $v_i \neq \emptyset$ es el hijo izquierdo de v entonces $v \leq v_i$; y si $v_d \neq \emptyset$ es el hijo derecho de v entonces $v \leq v_d$.*

Contrario a los árboles binarios ordenados, cada vértice es menor o igual que sus dos vértices hijos; en particular, no hay (necesariamente) ninguna relación entre los vértices hijos de un vértice; el izquierdo puede ser mayor al derecho o viceversa. Podemos ver un montículo mínimo en la figura [18.1](#).

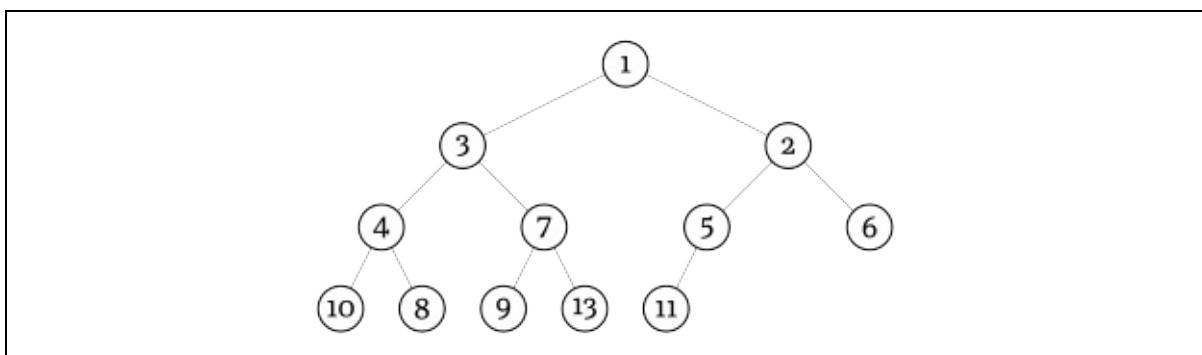


Figura 18.1: Montículo mínimo; cada vértice es menor o igual que sus vértices izquierdo y derecho.

Como habíamos mencionado los árboles binarios completos que son los montículos mínimos estarán implementados con un arreglo. Dado un montículo mínimo de n elementos tendremos un arreglo de tamaño n ; la raíz estará en el índice 0 del arreglo. El elemento en el índice i tiene como hijo izquierdo al elemento en el índice $2i + 1$ y como hijo derecho al elemento en el índice $2i + 2$; de la misma manera, su padre será el elemento en el índice $\left\lfloor \frac{i-1}{2} \right\rfloor$. Obviamente si el índice del elemento es 0 entonces no tiene padre y si el índice del elemento es i y $2i + 1$ y $2i + 2$ son mayores o iguales que el número de elementos, entonces ese elemento es una hoja.

El arreglo correspondiente al montículo mínimo de la figura 18.1 lo podemos ver en la figura 18.2.

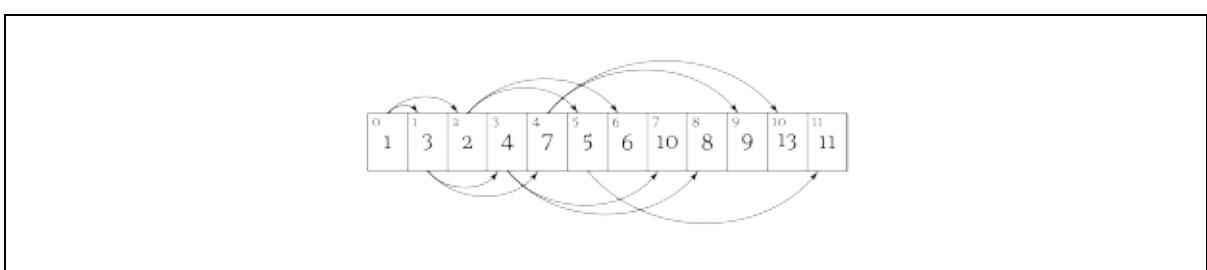


Figura 18.2: El arreglo correspondiente al montículo mínimo de la figura 18.1.

Los montículos mínimos estarán implementados con arreglos, pero en nuestras figuras siempre presentaremos el árbol binario, casi nunca el arreglo. El arreglo es una decisión de implementación; conceptualmente serán árboles binarios completos tal como los vimos en el capítulo 13. Sólo la implementación será más sencilla.

En 1964 John William Joseph Williams introdujo los montículos mínimos [32] como una estructura auxiliar para el algoritmo de ordenamiento HEAPSORT, que veremos más adelante en el capítulo.

Los montículos mínimos, por definición, tienen en su raíz a un elemento que es menor o igual que todos los otros elementos del montículo. Aunado al hecho de que podremos eliminar la raíz o actualizar los elementos en el mismo y después reacomodar el montículo para que siga siendo válido, todo en tiempo $O(\log n)$, esto hará a los montículos una estructura perfecta para implementar el algoritmo de Dijkstra en el capítulo 19.

Hacemos notar que podríamos utilizar un árbol rojinegro o AVL para implementar el algoritmo de Dijkstra (con ciertas modificaciones); pero los montículos mínimos con n elementos podemos construirlos en tiempo $O(n)$, contrario a $O(n \log n)$ con un árbol binario ordenado autobalanceable.

Cuando agreguemos, eliminemos o modifiquemos un elemento del montículo,

tendremos que reorganizarlo para que siga cumpliendo su propiedad definitoria. Dependiendo de la operación ocurrirá una de las siguientes cosas: o bien un vértice se va acomodando hacia arriba siguiendo a su padre; o bien un vértice se va acomodando hacia abajo siguiendo a su hijo más pequeño. Estas dos operaciones auxiliares no son parte del comportamiento público de la clase `MonticuloMinimo`, así que veremos los algoritmos correspondientes en la siguiente sección.

18.2. Acomodando hacia arriba y hacia abajo

Como mencionábamos antes vamos a necesitar algoritmos para reacomodar un vértice si el elemento que contiene cambia de valor o bien al agregar o eliminar elementos del montículo. En inglés estos algoritmos son conocidos por varios nombres, pero dos de los más comunes son *heapify-up* y *heapify-down*; como en español no acostumbramos usar sustantivos como verbos, nosotros usaremos *acomodando hacia arriba* y *acomodando hacia abajo*.

18.2.1. Acomodando hacia arriba

Tomemos el montículo de la figura 18.1 y modifiquemos el valor del elemento 13 a 0; como el vértice es menor que su padre, se viola la definición 18.1 (figura 18.3).

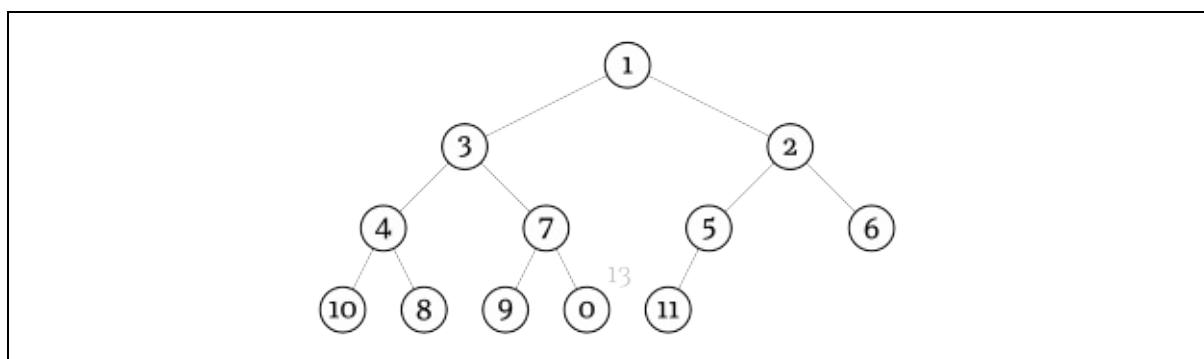


Figura 18.3: El vértice que tenía al 13 cambió su valor a 0; tenemos que acomodarlo hacia arriba (*heapify-up*).

El algoritmo es muy sencillo; sea v el vértice cuyo elemento cambió de valor *hacia abajo* (el valor fue decrementado por cierta magnitud). Mientras v tenga padre y sea menor que él, intercambiamos los valores de v y su padre y redefinimos v como su padre (figura 18.4).

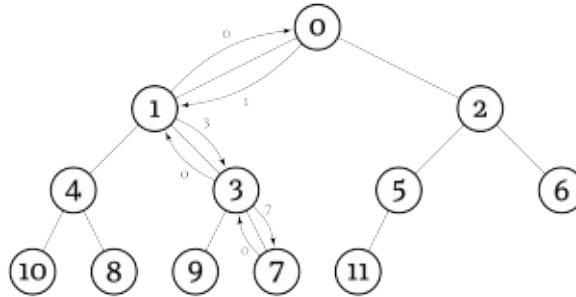


Figura 18.4: Intercambiamos el valor 0 con el de su padre mientras hay padre y sea mayor que 0; en este caso terminamos cuando 0 es la raíz.

Hacemos notar que no hay “vértices” realmente; los elementos del montículo están en un arreglo. Entonces intercambiar los elementos de dos vértices es intercambiar los elementos en dos índices en el arreglo (figura 18.5), algo que venimos haciendo desde el capítulo 10.

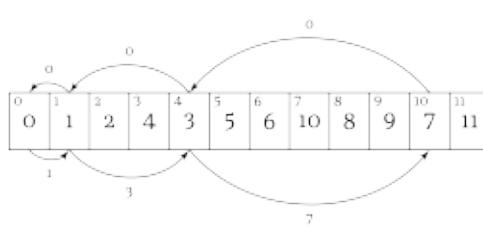


Figura 18.5: Reacomodando hacia arriba el valor 0 en el arreglo.

Sea v el vértice cuyo valor cambió de valor *hacia abajo*; evidentemente el subárbol $T(v)$ es un montículo mínimo válido: si v era menor que sus dos hijos *antes* de cambiar de valor hacia abajo, lo seguirá siendo *después* de cambiar. Y si al ser menor que su padre los intercambiamos, el subárbol $T(v)$ después del cambio volverá a ser un montículo mínimo válido.

El algoritmo tiene una complejidad en tiempo de $O(\log n)$; en el peor de los casos se recorre toda la altura del árbol, que al ser binario completo es por definición $\lfloor \log_2 n \rfloor$. En espacio puede ocupar $O(1)$, si lo hacemos iterativo; pero la versión recursiva es bastante elegante, aunque tome $O(\log n)$ por la pila de ejecución.

18.2.2. Acomodando hacia abajo

Dado el montículo de la figura 18.1 modifiquemos el valor del elemento 1 a 14. Como ahora la raíz es mayor que sus hijos el árbol deja de ser un montículo mínimo (figura 18.6).

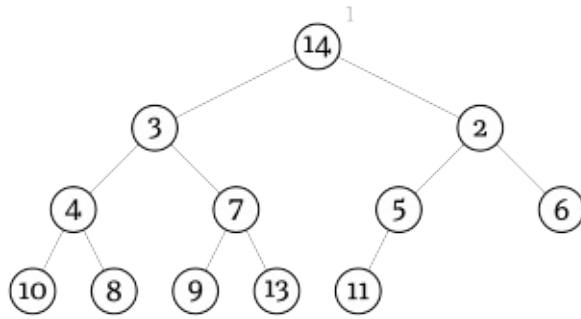


Figura 18.6: El vértice que tenía al vértice 1 cambió su valor a 14; tenemos que acomodarlo hacia abajo (*heapify-down*).

Sea v el vértice que cambió su valor *hacia arriba*. Mientras v sea mayor que alguno de sus hijos, intercambiamos v con el hijo menor y lo redefinimos como v (figura 18.7).

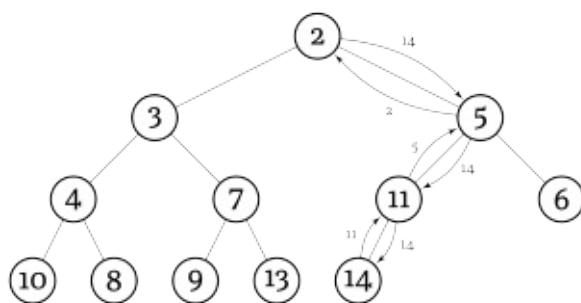


Figura 18.7: Reacomodando hacia abajo el valor 14 en el arreglo.

Cómo se ve el arreglo y los cambios correspondientes se puede ver en la figura 18.8.

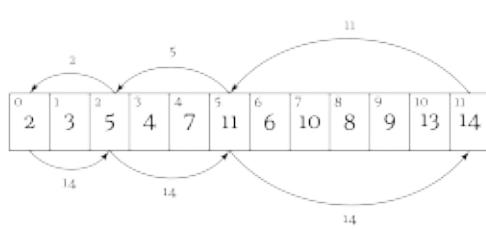


Figura 18.8: Reacomodando hacia abajo el valor 14 en el arreglo.

Si v es el vértice que cambió de valor *hacia arriba*, toda su rama de ascendientes hasta la raíz cumple con la definición de montículo mínimo. Si existe un hijo menor que v y reemplazamos v con su hijo de menor valor, este antes hijo volverá a estar bien con la definición.

La complejidad en tiempo del algoritmo es $O(\log n)$; una vez más en el peor de los casos recorremos la altura del árbol (sólo hacia abajo en este caso), que es $\lceil \log_2 n \rceil$. Al igual que al acomodar hacia abajo, la complejidad en espacio es

$O(1)$ si lo hacemos iterativamente o $O(\log n)$ si lo hacemos recursivamente.

18.3. Implementación en Java

Los montículos mínimos presentan varias dificultades para que los implementemos en Java. En primer lugar y como mencionamos en el capítulo 7, los arreglos no se llevan muy bien con los genéricos y obviamente queremos que nuestros montículos sean genéricos. Esto será fácil de solucionar, si bien la solución no es muy elegante; lo veremos en un momento.

En segundo lugar, para que la complejidad en tiempo de los montículos funcione, necesitamos poder acceder a los elementos en el montículo en tiempo constante; como hemos mencionado en los algoritmos para acomodar hacia arriba y hacia abajo, tenemos que poder modificar el valor de un elemento del montículo para después volver a acomodarlo. Si no sabemos dónde está en el montículo esto tomaría tiempo $O(n)$, porque al no estar ordenado no podemos buscar en tiempo $O(\log n)$. Por suerte localizar al elemento en el montículo es trivial si sabemos su índice; el problema es que una instancia de `Comparable` no tiene comportamiento para determinar (o actualizar) su índice.

Lo que haremos será definir una interfaz `ComparableIndexable` que extienda `Comparable` y agregue comportamiento para definir y modificar un índice; podemos ver la interfaz en el listado 18.1.

```
public interface ComparableIndexable<T>
    extends Comparable<T> {
    public int getIndice();
    public void setIndice(int indice);
}
```

Listado 18.1: La interfaz `ComparableIndexable`.

Esto restringirá el uso de nuestros montículos, porque no podremos usarlos para cualquier `Comparable`, necesitaremos envolver el comparable en una instancia de `ComparableIndexable`. Pero dado que implementaremos montículos mínimos básicamente para usarlos en el algoritmo de Dijkstra, esto no será tanto problema. También usaremos los montículos para implementar `HEAPSORT`, pero en ese caso podremos utilizar un truco para poder implementar el algoritmo de tal forma que funcione para cualquier instancia de `Comparable`.

Si piensan detenidamente el problema de que necesitamos acceder y

modificar el índice de cualquier elemento en el montículo, la interfaz `ComparableIndexable` es de hecho una mala idea desde el punto de vista del diseño; sería más elegante definir una interfaz `Indexable` (independiente de `Comparable`) con los métodos `getIndice()` y `setIndice()`; y acotar el genérico de los montículos a `Comparable` e `Indexable` (se vería como en el listado 18.2).

```
public class MonticuloMinimo<T extends Comparable<T> &
                           Indexable> {
    /* ... */
}
```

Listado 18.2: La clase `MonticuloMinimo` acotada a `Comparable` e `Indexable`.

Esta solución más elegante no la podemos utilizar, por culpa de la incompatibilidad que existe en Java entre arreglos y genéricos. Para nuestros montículos necesitamos un arreglo `T[]`, pero como ya hemos explicado no podemos hacer `T[] a = new T[n]` porque no podemos (directamente) crear un arreglo de genéricos.

Hay dos maneras de crear arreglos de genéricos y ambas necesitan utilizar una audición (y por lo tanto suprimir advertencias). Una de ellas es utilizar el método estático `newInstance()` de la clase `Array`; esto implica utilizar introspección (un tema cuyos detalles están fuera del alcance de este libro). Vamos a ver esta manera de crear un arreglo genérico en el capítulo 21.

Para nuestros montículos utilizaremos la segunda manera, que consiste en crear un arreglo de una interfaz (y con esto forzar un arreglo de referencias, porque con una interfaz no podemos saber cuánto espacio ocuparán en memoria) y hacerle una audición a `T[]`. El problema es que podemos crear un arreglo de *una* interfaz, no de varias, así que no podemos acotar `T` a `Comparable` e `Indexable`; tenemos que acotarlo a `ComparableIndexable`. No es la solución más elegante, pero funciona.

De forma muy anti intuitiva, también podríamos hacer que la clase *compilara* creando un arreglo de tipo `Object[]` y haciendo la audición a `T[]`; pero entonces el programa falla *en tiempo de ejecución* al tratar de hacer que un `Object` se comporte como el tipo que reemplace a `T`.

Para crear el arreglo utilizaremos un método privado auxiliar llamado `nuevoArreglo()`, que mostramos en el listado 18.3.

```
@SuppressWarnings("unchecked") private T[]
nuevoArreglo(int n) {
    return (T[])(new ComparableIndexable[n]);
}
```

Listado 18.3: El método nuevoArreglo().

Eso soluciona nuestro arreglo. Como el árbol binario es completo, los elementos del arreglo estarán todos al inicio del mismo; vamos a necesitar una variable de clase de tipo entero que nos diga el número de elementos que también servirá para saber cuál es el siguiente índice del arreglo donde podríamos agregar un elemento (el primer índice sin elemento).

Como con todas nuestras colecciones vamos a necesitar una clase interna para nuestros iteradores, en este caso privada. Para poder implementar HEAPSORT vamos a necesitar otra clase interna privada que nos permita envolver un `Comparable` en un `ComparableIndexable`; llamaremos esta clase `Adaptador`, porque es el patrón que utilizaremos, el patrón *Adaptador* (*Adapter*). Veremos esto a detalle en la sección [18.5](#).

Nuestros montículos implementarán la interfaz `Coleccion`, así que tendremos los métodos de la misma. Además tendremos un método para eliminar la raíz del árbol y otro para reordenar un elemento del montículo; esto métodos los necesita el algoritmo de Dijkstra que veremos el próximo capítulo. También tendremos otro método para obtener el i -ésimo elemento del arreglo, que nos servirá junto con los métodos `esVacia()` y `getElementos()` para poder verificar que el montículo es consistente. A estos métodos (incluyendo los dos que pertenecen a `Coleccion`) los pondremos en una interfaz llamada `MonticuloDijkstra` que `MonticuloMinimo` implementará; la razón de ser de esta interfaz la veremos en la sección [18.6](#).

La interfaz `MonticuloDijkstra` la podemos ver en el listado [18.4](#).

```
public interface MonticuloDijkstra<T extends ComparableIndexable<T>> {
    public T elimina();
    default public void reordena(T elemento) {}
    public T get(int i);
    public boolean esVacia();
    public int getElementos();
}
```

Listado 18.4: La interfaz `MonticuloDijkstra`.

Hacemos notar que la interfaz `MonticuloDijkstra` utiliza los métodos por omisión para interfaces (*default methods*) que Java 8 incorpora en el lenguaje. Veremos en la sección [18.6](#) por qué, pero mientras es interesante recalcar que el método `reordena()` por omisión no hace nada. Literalmente.

Por último, nuestra clase `MonticuloMinimo` tendrá un método estático para implementar HEAPSORT; veremos éste en la sección [18.5](#). La clase `MonticuloMinimo` la podemos ver en el listado [18.5](#).

```

public class MonticuloMinimo<T extends Comparable>
    implements Coleccion<T>, MonticuloDijkstra<T> {

    private class Iterador implements Iterator<T> {
        private int indice;
        @Override public boolean hasNext() { /* ... */ }
        @Override public T next() { /* ... */ }
    }

    private static class Adaptador<T extends Comparable>
        implements ComparableIndexable<Adaptador<T>> {

        private T elemento;
        private int indice;

        public Adaptador(T elemento) { /* ... */ }
        @Override public int getIndice() { /* ... */ }
        @Override public void setIndice(int indice) { /* ... */ }
        @Override public int compareTo(Adaptador<T> adaptador) { /* ... */ }
    }

    private int elementos;
    private T[] arbol;
    @SuppressWarnings("unchecked") private T[]
    nuevoArreglo(int n) {
        return (T[])(new ComparableIndexable[n]);
    }
    public MonticuloMinimo() { /* ... */ }
    public MonticuloMinimo(Coleccion<T> colección) {
        /* ... */
    }
    public MonticuloMinimo(Iterable<T> iterable,
                           int n) { /* ... */ }
    public T elimina() { /* ... */ }
    public T get(int i) { /* ... */ }
    public void reordena(T elemento) { /* ... */ }
    @Override public boolean
    contiene(T elemento) { /* ... */ }
    @Override public boolean esVacia() { /* ... */ }
    @Override public int getElementos() { /* ... */ }
    @Override public void agrega(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public void limpia() { /* ... */ }
    @Override public boolean equals(Object o) { /* ... */ }
    @Override public String toString() { /* ... */ }
    @Override public Iterator<T> iterator() {
        return new Iterador();
    }
}

```

```

    }

    public static <T extends Comparable<T>>
    Lista<T> heapSort(Coleccion<T> colección) { /* ... */ }
}

```

Listado 18.5: Esqueleto de la clase `Monticulominimo` con los métodos `nuevoArreglo()` y `iterator()` ya implementados.

Debe quedar claro que al agregar un elemento al arreglo hay que decirle al elemento en qué índice está y que al cambiar un elemento de lugar en el arreglo hay que actualizar su índice; ambas cosas las haremos con el método `setIndice()` de `ComparableIndexable`. Por convención, al eliminar un elemento del montículo definiremos su índice como `-1`.

18.4. Algoritmos para montículos mínimos

Como ha sido costumbre revisaremos los algoritmos para los montículos en el orden dado por el listado [18.5](#), comenzando por los métodos del iterador.

- La clase `Iterador` no tiene constructor porque no lo necesita; la variable `índice` se inicializa en 0.
- `hasNext()`

Sólo hay que verificar que el índice del iterador sea menor que el número de elementos en el montículo.

- `next()`

Si el índice del iterador es mayor o igual que el número de elementos en el montículo ocurre un error. Si no regresamos el elemento del arreglo en el índice e incrementamos el mismo.

La clase `Adaptador` la veremos cuando revisemos el algoritmo `HEAPSORT`. Vemos ahora los algoritmos de la clase.

- `Monticulominimo()`

El constructor sin parámetros únicamente crea el arreglo con un tamaño arbitrario; un tamaño alrededor de 100 es recomendado, pero cualquier tamaño funciona.

- `Monticulominimo(Colección<T> colección)` y `Monticulominimo(Iterable<T> iterable, int n)`

Estos dos constructores son equivalentes y lo más sencillo es que el

primero mande llamar al segundo; necesitaremos la versión que recibe un iterable por razones que veremos en el capítulo [23](#).

Estos constructores son importantes, porque nos permiten construir el montículo mínimo en tiempo $O(n)$. El algoritmo es como sigue: se crea el arreglo del mismo tamaño que el recibido y se agregan en orden los elementos del iterable al arreglo. La variable con el número de elementos se define como el tamaño recibido. Hasta aquí hemos utilizado tiempo $O(n)$.

Sea n el tamaño del arreglo; lo que hacemos ahora es recorrer el mismo desde el índice $\frac{n}{2}$ hasta 0, acomodando hacia abajo cada elemento usando el algoritmo de la sección [18.2.2](#). Dado que acomodar hacia abajo tiene una complejidad en tiempo de $O(\log n)$ y hacemos esto $\frac{n}{2}$ veces, podría parecer que el algoritmo toma tiempo $O(n \log n)$; pero se puede hacer la amortización de los movimientos para demostrar que esto toma $O(n)$. El punto fuerte de la amortización consiste en lo siguiente: al comenzar en el índice $\frac{n}{2}$, esto deja *al menos* el último nivel del árbol (y el que más elementos tiene) fuera del recorrido. Al inicio los elementos se acomodan hacia abajo a lo más un nivel; y cada vez que subimos de nivel el número de elementos se reduce a la mitad. En otras palabras en el peor de los casos (y viéndolo de arriba hacia abajo): en el nivel 0 tenemos 1 elemento que se mueve $\lfloor \log_2 n \rfloor - 1$ niveles; en el nivel 1 tenemos 2 elementos que se mueven $\lfloor \log_2 n \rfloor - 2$ niveles; en el nivel 2 tenemos 4 elementos que se mueven $\lfloor \log_2 n \rfloor - 3$ niveles, etcétera. Esto se traduce en

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} 2^i (\lfloor \log_2 n \rfloor - i - 1) < n.$$

La desigualdad se puede demostrar fácilmente por inducción; se le deja de ejercicio al lector.

- `elimina()`

El método elimina la raíz del árbol. Para hacer esto intercambiamos la raíz y el último elemento en el arreglo (el que esté en el índice igual al número de elementos en el montículo menos 1) y decrementamos el número de elementos en el montículo; como el último elemento era mayor que la raíz, acomodamos hacia abajo la nueva raíz. Al final regresamos el elemento que estaba en la raíz.

- `get()`

Si el índice recibido es menor que cero o mayor o igual que el número de elementos en el montículo, ocurre un error. Si no regresamos el i -ésimo elemento del arreglo.

- `reordena()`

El algoritmo supone que el elemento recibido ha cambiado de valor, así que lo reordenamos (lo accedemos en el arreglo con su índice). Si el elemento es menor que su padre lo acomodamos hacia arriba; si es mayor que alguno de sus hijos lo acomodamos hacia abajo. De hecho podemos ejecutar incondicionalmente los algoritmos para acomodar hacia arriba y hacia abajo; ambos se detienen de inmediato si el elemento ya está bien acomodado respecto a su padre o hijos, respectivamente.

- `contiene()`

Por cómo mantendremos el índice de los elementos del arreglo, si el índice del elemento recibido es menor que cero o mayor o igual que el número de elementos, regresamos falso. Si no, comparamos el elemento del arreglo en el índice del elemento recibido con el elemento recibido; si son iguales regresamos verdadero, si no regresamos falso.

- `esVacia()`

Regresamos si el número de elementos es mayor que 0.

- `getElementos()`

Regresamos el número de elementos.

- `agrega()`

Si el número de elementos en el montículo es igual al tamaño del arreglo, nos veremos forzados a incrementar el tamaño del mismo. Lo que faremos en este caso es crear un nuevo arreglo del doble del tamaño del original, copiar los elementos del viejo arreglo al nuevo y hacer al nuevo arreglo el arreglo del montículo.

Independientemente de si crecimos o no el arreglo, ponemos el nuevo elemento en el índice correspondiente al número de elementos, incrementamos el número de elementos y acomodamos hacia arriba al último elemento del arreglo.

- `elimina(T elemento)`

Si el índice del elemento recibido es menor que cero o mayor o igual que el número de elementos, no hacemos nada. Si no, intercambiamos el elemento recibido con el último, decrementamos el número de elementos y acomodamos el elemento que reemplazó al eliminado, ya sea hacia

arriba o hacia abajo dependiendo del que tenga el elemento que reemplaza.

- `limpia()`

Definimos el número de elementos en cero y anulamos todas las entradas del arreglo.

- `toString()`

Vamos a hacer sencillo el método; únicamente serán las cadenas de los elementos en el arreglo en el orden del mismo, separados por comas.

- `equals()`

Tenemos que hacer de nuevo una audición en una clase genérica y suprimir la advertencia resultante (listado 18.6).

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    MonticuloMinimo<T> m =
        (MonticuloMinimo<T>)o;
    // ...
}
```

Listado 18.6: Inicio del método `equals()` en `MonticuloMinimo`.

Vamos a considerar dos montículos mínimos iguales únicamente si tienen los mismos elementos en el mismo orden en el arreglo; pero el tamaño exacto de los arreglos puede diferir.

- `heapSort()`

Veremos el algoritmo en la siguiente sección.

18.5. Algoritmo HEAPSORT

Para poder implementar HEAPSORT que funcione con cualquier objeto instancia de `Comparable`, necesitamos *adaptar* ese comparable a un comparable que además sea indexable. Es un patrón de diseño bastante usado (*Adapter*) que nos permite adaptar (como su nombre indica) las instancias de una clase sin necesidad de modificarla o extenderla; en este caso es justo lo que necesitamos porque al ser `T` un genérico acotado a `Comparable<T>`, no podemos hacer ninguna de las dos.

Para eso tendremos la clase interna privada `Adaptador`, que lo único que hace es

envolver un elemento comparable y agregarle un índice. Los algoritmos de la clase **Adaptador** son todos muy sencillos:

- **Adaptador()**

El constructor define el elemento del adaptador como el recibido y define su índice como -1 .

- **getIndice()**

Regresamos el índice del adaptador.

- **setIndice()**

Definimos el índice del adaptador.

- **compareTo()**

Compara el elemento del adaptador con el elemento del adaptador recibido.

El algoritmo **HEAPSORT** es básicamente trivial; tomamos la colección recibida y creamos con ella una lista ℓ_1 de adaptadores, donde cada adaptador de la lista contiene el elemento correspondiente de la colección. Creamos también una lista ℓ_2 de elementos del tipo de la colección.

Con la lista ℓ_1 creamos un montículo mínimo en tiempo $O(n)$ y mientras el montículo tenga elementos, eliminamos su raíz y agregamos el elemento del adaptador eliminado en la lista ℓ_2 . Al final regresamos ℓ_2 ; como eliminar la raíz del montículo toma tiempo $O(\log n)$, la operación en total toma tiempo $O(n \log n)$.

Por supuesto casi cualquier otro de los algoritmos de ordenamiento (excepto **SELECTIONSORT**) o cualquiera de los árboles autabalancables que hemos visto es de hecho más sensato utilizarlos que **HEAPSORT**; pero como los montículos fueron creados justamente para definir el algoritmo, decidimos incluirlo en el capítulo.

18.6. Montículos de arreglos

En el próximo capítulo cubriremos el algoritmo de Dijkstra, que es la razón principal para haber visto montículos mínimos. Los montículos mínimos nos darán en promedio la mejor complejidad en tiempo para implementar el algoritmo de Dijkstra; sin embargo, en algunos casos los montículos mínimos nos dan una complejidad en tiempo no deseable.

Sorprendentemente, un arreglo normal funciona mejor que un montículo mínimo en estos casos del algoritmo de Dijkstra. Lo que hacemos entonces es encapsular las operaciones que necesita realizar el algoritmo de Dijkstra (junto con algunas más auxiliares) en la interfaz `MonticuloDijkstra` y vamos a escribir una segunda clase que la implemente, que sencillamente envolverá a un arreglo.

Esto nos permitirá implementar el algoritmo de Dijkstra de tal forma que use un montículo de Dijkstra (una instancia de `MonticuloDijkstra`) y dependiendo del tipo de gráfica que tengamos se usará un montículo mínimo o un montículo de arreglo.

La clase se llamará `MonticuloArreglo` y, además de los constructores, solamente tendrá los métodos públicos definidos en la interfaz `MonticuloDijkstra`. Podemos ver su esqueleto en el listado 18.7.

```
public class MonticuloArreglo<T> extends
    ComparableIndexable<T>>
    implements MonticuloDijkstra<T> {

    private int elementos;
    private T[] arreglo;

    @SuppressWarnings("unchecked") private T[]
    nuevoArreglo(int n) {
        return (T[])(new ComparableIndexable[n]);
    }

    public MonticuloArreglo(Coleccion<T> colección) {
        /* ... */
    }
    public MonticuloArreglo(Iterable<T> iterable,
                           int n) { /* ... */ }
    @Override public T elimina() { /* ... */ }
    @Override public T get(int i) { /* ... */ }
    @Override public boolean esVacia() { /* ... */ }
    @Override public int getElementos() { /* ... */ }
}
```

Listado 18.7: Esqueleto de la clase `MonticuloArreglo`.

La clase `MonticuloArreglo` *no* es una colección; sólo es auxiliar para usarla en el algoritmo de Dijkstra. Esto la simplifica mucho; no necesitamos preocuparnos por agregar elementos o eliminar elementos arbitrarios ni tampoco tenemos que hacerla iterable. El montículo sólo eliminará el elemento mínimo hasta que esté vacío, buscándolo en tiempo lineal cada vez.

También usamos el mismo truco de `MonticuloMinimo` para crear arreglos

genéricos y de hecho lo mostramos ya implementado en el esqueleto de la clase.

Por último hacemos notar que la clase `MonticuloArreglo` no sobrecarga el método `reordena()`, que no hace nada en la implementación por omisión de la interfaz y por lo tanto los montículos de arreglo no harán nada al pedirles que reordenen un elemento. Lo cual tiene sentido porque los elementos del arreglo no están ordenados de ninguna manera.

Los algoritmos correspondientes a los métodos de `MonticuloArreglo` son en general muy sencillos:

- `MonticuloArreglo(Coleccion<T> colección)` y `MonticuloArreglo(Iterable<T> iterable, int n)`

Al igual que con `MonticuloMinimo`, estos dos constructores son equivalentes y lo más sencillo es que el primero mande llamar al segundo; veremos la necesidad del segundo constructor en el capítulo [23](#).

Creamos un nuevo arreglo usando `nuevoArreglo()` con la n recibida y agregamos los elementos del iterable en el arreglo, definiendo los índices de los mismos. La variable `elementos` se inicializa con el tamaño del arreglo.

- `elimina()`

Si el número de elementos en el montículo es 0 (en otras palabras, la variable de clase `elementos` es 0), ocurre un error.

Si no recorremos el arreglo buscando el mínimo elemento en el mismo. Anulamos la entrada en el arreglo que le corresponde, le definimos su índice como -1 , decrementamos el número de elementos y regresamos el elemento mínimo.

Podríamos intercambiar el último elemento del arreglo (usando la variable `elementos`) con el eliminado, para no tener hoyos anulados; pero es realizar una operación más con la que no ganamos realmente nada.

La complejidad en tiempo de este método es lineal.

- `get()`

Si i es menor que cero o mayor o igual que la longitud del arreglo, ocurre un error. De otro modo regresamos la i -ésima entrada en el arreglo. La entrada regresada puede ser \emptyset .

- `esVacia()`

Regresamos verdadero si el número de elementos es cero, falso en otro

caso.

- `getElementos()`

Regresamos el número de elementos.

La clase `MonticuloArreglo` y la interfaz `MonticuloDijkstra` las utilizaremos únicamente para manejar algunos casos del algoritmo de Dijkstra. Veremos esto a detalle en el próximo capítulo.

Ejercicios

1. Implementa los métodos faltantes de las clases `MonticuloArreglo` y `MonticuloMinimo`.
2. ¿Cuál es la complejidad en tiempo de crear un montículo mínimo con n elementos?
3. ¿Cuál es la complejidad en tiempo de agregar un elemento a un montículo mínimo?
4. ¿Cuál es la complejidad en tiempo de eliminar un elemento de un montículo mínimo?
5. Dado el montículo mínimo en la figura 18.9, elimínale su menor elemento.

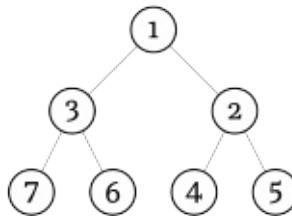


Figura 18.9: Montículo mínimo.

-
1. En el libro sólo veremos montículos mínimos; los montículos máximos son idénticos, sólo invirtiendo las desigualdades.[18.9](#)

19. Algoritmo de Dijkstra

En este capítulo regresaremos a nuestras gráficas para implementar un algoritmo muy importante en las mismas: cómo encontrar la trayectoria de peso mínimo entre dos vértices.

Para que este problema sea interesante necesitamos que nuestras gráficas tengan peso en las aristas; el peso generalmente representará el *costo* de movernos de un vértice a otro. Si los vértices de una gráfica representan ciudades y dos vértices están conectados si existe una autopista directa entre ellos, podríamos asociar la distancia entre las ciudades como el peso. O, si son autopistas de pago, el costo de las casetas. O el costo promedio del combustible que se gasta al ir de una ciudad a otra.

Con nuestras gráficas con pesos en las aristas podemos definir el peso de una trayectoria y con ello buscar una trayectoria de peso mínimo. Veremos esto en la siguiente sección.

19.1. Definición de trayectoria de peso mínimo

El peso de una arista en una gráfica lo definiremos como una función de las aristas de la gráfica a los reales positivos:

Definicion 19.1. (Gráficas ponderadas) *Una gráfica ponderada es una gráfica $G = (V, E)$ junto con una función $w : E \rightarrow \mathbb{R}^+$. El número $w(e)$ se llama el peso de la arista e .*

Para motivos de legibilidad, definiremos $w(u, v) = w((u, v))$, para toda $u, v \in V$. Por lo que discutimos en el capítulo 17, debe ser obvio que $w(u, v) = w(v, u)$. Podemos ver una gráfica con pesos en la figura 19.1.

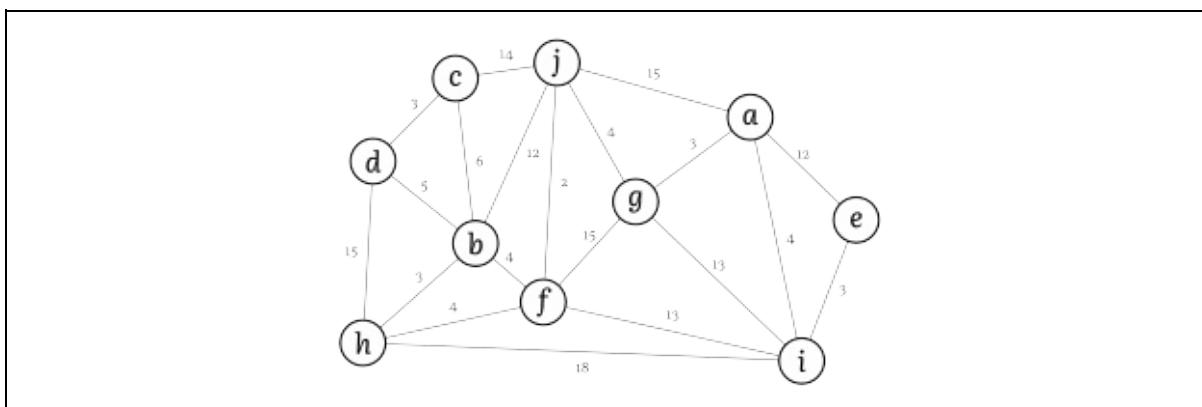


Figura 19.1: Gráfica con pesos en las aristas.

Una vez que nuestra gráfica tiene pesos en sus aristas, podemos definir el peso de una trayectoria como sigue:

Definicion 19.2. (Pesos de trayectorias) *Sea $G(E, V)$ una gráfica ponderada y w la función de costo de sus aristas. Sea $T = v_1, v_2, \dots, v_k$ una trayectoria en G ; el peso de T , denotado por $w(T)$, está definido por:*

$$w(T) = \sum_{i=2}^k w(v_{i-1}, v_i).$$

Con esto distintas trayectorias entre dos vértices tendrán en general distintos pesos (figura 19.2), por lo que tiene sentido preguntarnos qué trayectoria tendrá peso mínimo.

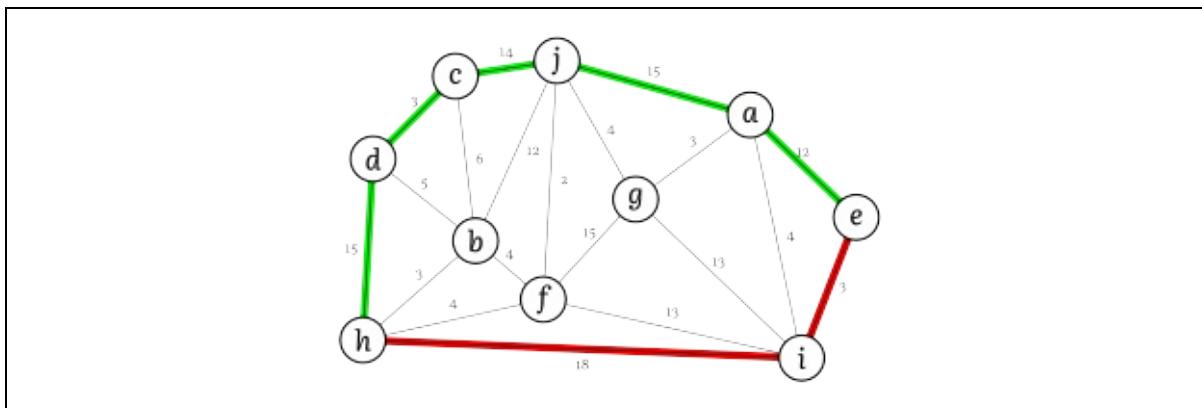


Figura 19.2: Dos trayectorias distintas entre los vértices h y e ; las trayectorias tienen pesos diferentes.

Es conveniente recordar que las trayectorias por definición no pueden tener vértices repetidos y que por lo tanto en una gráfica finita sólo hay un número finito de trayectorias entre cada dos vértices. Por lo tanto una trayectoria de peso mínimo entre dos vértices es sencillamente la que menor peso tenga entre esos dos vértices (puede haber varias trayectorias con peso mínimo).

Existen distintos algoritmos para encontrar una trayectoria de peso mínimo, ya sea entre dos vértices o entre *todos* los vértices de la gráfica. En este libro (y como el nombre del capítulo indica) veremos el algoritmo de Dijkstra en la sección 19.5.

De la misma manera que con la trayectoria de peso mínimo, tiene sentido preguntarnos cuál es la trayectoria de longitud mínima (o trayectoria mínima) entre dos vértices. El algoritmo para encontrar la trayectoria mínima lo veremos en la sección 19.4; en la siguiente sección veremos cómo implementar los pesos de las aristas en nuestra clase `Grafica`.

19.2. Implementación en Java

El problema de implementar los pesos en las aristas no es excesivamente complicado, pero tampoco es trivial. Tenemos que determinar dónde en la memoria estarán los pesos; obviamente un vértice no puede tener el peso de una arista, porque el número de aristas de las cuales el vértice forma parte es variable.

Así como un vértice tiene una lista de vértices vecinos, podríamos tener una lista de pesos para cada vecino. Esto es factible, pero el mantener sincronizadas las listas puede resultar engorroso, además de que se puede discutir de que no es un diseño elegante bajo la Orientación a Objetos.

Lo que haremos para resolver el problema es justamente utilizar Orientación a Objetos; crearemos una clase interna y privada llamada `vecino`, que implementará la interfaz `VerticeGrafica` y que tendrá como propiedades un vértice y un doble; un adaptador de `Vertice`. De esta manera cada vértice puede tener una lista de vecinos y cada vecino sabe el peso de la arista formada por el vértice y el vértice vecino. Como `vecino` será un adaptador de `vertice` (siguiendo el patrón *Adaptador* que vimos el capítulo pasado), todos sus métodos adaptan los métodos de su clase padre; por ejemplo, el método `get()` de `vecino` regresa el elemento del vértice vecino (listado 19.1).

```
@Override public T get() {  
    return vecino.elemento;  
}
```

Listado 19.1: Método `get()` de la clase `Vecino`.

Este cambio nos obliga a hacer algunas cosas distintas en nuestra clase `Grafica`; por ejemplo, el método `setColor()` sencillamente verificaba que la instancia de la interfaz `VerticeGrafica` fuera instancia de la clase interna `Vertice` y si así era el caso procedía a modificar el color del vértice (listado 19.2).

```
public void setColor(VerticeGrafica<T> vertice,  
                     Color color) {  
    if (vertice == null ||  
        vertice.getClass() != Vertice.class) {  
        m = "Vértice inválido";  
        throw new IllegalArgumentException(m);  
    }  
    Vertice v = (Vertice)vertice;  
    v.color = color;  
}
```

Listado 19.2: Método `setColor()` antes de tener gráficas ponderadas.

Ahora no será tan sencillo, porque la instancia de `VerticeGrafica` puede no sólo ser instancia de `Vertice`; también podrá ser instancia de `Vecino` y éstas últimas no tienen la propiedad `color`. Para resolver esto haremos lo obvio y verificamos que pueda ser instancia de `Vecino` además de instancia de `Vertice`. Si el vértice recibido es instancia de `Vertice`, actualizamos su color; si es instancia de `Vecino`, actualizamos el color del vértice vecino (listado 19.3).

```
public void setColor(VerticeGrafica<T> vertice,
                     Color color) {
    if (vertice == null ||
        (vertice.getClass() != Vertice.class &&
         vertice.getClass() != Vecino.class)) {
        m = "Vértice inválido";
        throw new IllegalArgumentException(m);
    }
    if (vertice.getClass() == Vertice.class) {
        Vertice v = (Vertice)vertice;
        v.color = color;
    }
    if (vertice.getClass() == Vecino.class) {
        Vecino v = (Vecino)vertice;
        v.vecino.color = color;
    }
}
```

Listado 19.3: Método `setColor()` para gráficas ponderadas.

Todo esto es necesario porque el método `setColor()` de la clase `Grafica` tiene que manejar tanto instancias de `Vertice` como de `Vecino` y al ser público no podemos en general saber exactamente cuál de las dos clases instanció nuestro objeto. En cambio dentro de la clase `Grafica` en general sí sabremos exactamente con qué tipo estamos lidiando y actuar como corresponda; por ejemplo, en nuestra implementación anterior cambiar los colores de los vecinos al recorrer la lista de vecinos lo hacíamos como en el listado 19.4.

```
for (Vertice v : u.vecinos) {
    if (v.color == Color.NINGUNO) {
        v.color = Color.ROJO;
        cola.mete(v);
    }
}
```

Listado 19.4: Recorriendo la lista de vecinos cuando son instancias de `Vertice`.

Ahora el recorrer la lista de vecinos requerirá de una redirección más, ya que el vecino no será una instancia de `Vertice`, sino una instancia de `Vecino`

(listado 19.5).

```
for (Vecino v : u.vecinos) {
    if (v.vecino.color == Color.NINGUNO) {
        v.vecino.color = Color.ROJO;
        cola.mete(v.vecino);
    }
}
```

Listado 19.5: Recorriendo la lista de vecinos cuando son instancias de `Vecino`.

Todos estos cambios únicamente afectan la implementación interna de la clase; todo el código que dependiera de nuestra antigua implementación puede seguir utilizando la clase sin ningún cambio, porque el comportamiento público de la misma no cambia excepto para agregar algunos métodos.

Vamos también a agregar un método `conecta()` que reciba un parámetro más, un doble para el peso de la arista; el antiguo método `conecta()` seguirá existiendo, pero definirá un peso con valor de 1 por omisión. De manera análoga, tendremos un método `getPeso()` que nos dé el peso de un par de vértices de la gráfica.

Como con `mergeSort()` y `quicksort()`, el método para encontrar la trayectoria de peso mínimo se llamará como el algoritmo que implementará: `dijkstra()`. El método para encontrar la trayectoria mínima se llamará, poco imaginativamente, `trayectoriaMinima()`. Ambos métodos regresarán listas de elementos de la gráfica, que representarán las trayectorias.

Para implementar `dijkstra()` vamos a utilizar los montículos mínimos que implementamos en el capítulo 18 (básicamente para esto los implementamos). Vamos a tener que poner los vértices de la gráfica en un montículo mínimo y por lo tanto los vértices tendrán que implementar la interfaz `ComparableIndexable`; esto implicará que nuestros vértices tendrán que agregar una propiedad entera `indice` y los métodos `getIndice()` y `setIndice()`. Además agregaremos una propiedad doble `distancia`, que necesitaremos al implementar los algoritmos para encontrar trayectorias y que usaremos para comparar con el método `compareTo()`.

Por último con la clase `Vertice`, podría parecer que tenemos un problema con el método `vecinos()`, porque tenemos que regresar un iterable de vértices de gráfica y ahora nuestra lista de vecinos no es de instancias de `Vertice`, sino de `Vecino`. Sin embargo lo que regresa el método está definido como `Iterable<? extends VerticeGrafica<T>>` (por eso podíamos regresar una lista de instancias de `Vertice`) y como la clase clase `Vecino` implementa a la interfaz `VerticeGrafica<T>` entonces no será problema.

Para terminar con esta nueva versión de la clase `Grafica`, tendremos una interfaz privada llamada `BuscadorCamino` que nos ayudará a implementar nuestros métodos `dijkstra()` y `trayectoriaMinima()`. Explicaremos cómo se utiliza más adelante.

Podemos ver el esqueleto de nuestra versión actualizada de la clase `Grafica` en el listado [19.6](#).

```
public class Grafica<T> implements Coleccion<T> {

    private class Iterador implements Iterator<T> {
        private Iterator<Vertice> iterador;
        public Iterador() { /* ... */ }
        @Override public boolean hasNext() { /* ... */ }
        @Override public T next() { /* ... */ }
    }

    private class Vertice
        implements VerticeGrafica<T>,
                   ComparableIndexable<Vertice> {
        public T elemento;
        public Color color;
        public double distancia;
        public int indice;
        public Lista<Vecino> vecinos;
        public Vertice(T elemento) { /* ... */ }
        @Override public T get() { /* ... */ }
        @Override public int getGrado() { /* ... */ }
        @Override public Color getColor() { /* ... */ }
        @Override public
        Iterable<? extends VerticeGrafica<T>> vecinos() {
            /* ... */
        }
        @Override public void
        setIndice(int indice) { /* ... */ }
        @Override public int getIndice() { /* ... */ }
        @Override public int
        compareTo(Vertice vertice) { /* ... */ }
    }

    private class Vecino implements VerticeGrafica<T> {
        public Vertice vecino;
        public double peso;
        public Vecino(Vertice vecino,
                     double peso) { /* ... */ }
        @Override public T get() { /* ... */ }
        @Override public int getGrado() { /* ... */ }
        @Override public Color getColor() { /* ... */ }
        @Override public
    }
}
```

```

        Iterable<? extends VerticeGrafica<T>> vecinos() {
            /* ... */
        }
    }

    @FunctionalInterface
    private interface BuscadorCamino {
        public boolean seSiguen(Grafica.Vertice v,
                               Grafica.Vecino a);
    }

    private Lista<Vertice> vertices;
    private int aristas;

    public Grafica() { /* ... */ }
    public int getAristas() { /* ... */ }
    public void conecta(T a, T b) { /* ... */ }
    public void conecta(T a, T b, double peso) { /* ... */ }
    public void desconecta(T a, T b) { /* ... */ }
    public boolean sonVecinos(T a, T b) { /* ... */ }
    public double getPeso(T a, T b) { /* ... */ }
    public VerticeGrafica<T>
    vertice(T elemento) { /* ... */ }
    public void setColor(VerticeGrafica<T> vertice,
                        Color color) { /* ... */ }
    public boolean esConexa() { /* ... */ }
    public void
    paraCadaVertice(AccionVerticeGrafica<T> accion) {
        /* ... */
    }
    public void
    bfs(T elemento, AccionVerticeGrafica<T> accion) {
        /* ... */
    }
    public void
    dfs(T elemento, AccionVerticeGrafica<T> accion) {
        /* ... */
    }

    @Override public int getElementos() { /* ... */ }
    @Override public void agrega(T elemento) { /* ... */ }
    @Override public boolean
    contiene(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public boolean esVacia() { /* ... */ }
    @Override public void limpia() { /* ... */ }
    @Override public String toString() { /* ... */ }
    @Override public boolean equals(Object o) { /* ... */ }
    @Override public Iterator<T> iterator() {
        return new Iterador();
    }
}

```

```

    }

    public Lista<VerticeGrafica<T>>
    trayectoriaMinima(T origen, T destino) {
        /* ... */
    }

    public Lista<VerticeGrafica<T>>
    dijkstra(T origen, T destino) {
        /* ... */
    }
}

```

Listado 19.6: Esqueleto de la clase **Grafica** con pesos en las aristas.

19.3. Algoritmos para gráficas ponderadas

La mayor parte de los algoritmos de nuestra clase **Grafica** actualizada ya los vimos en el capítulo [17](#) y en su mayor parte no cambian o cambian de formas triviales; por ejemplo, el constructor de **Vertice** ahora no crea una lista de vértices, sino una lista de vecinos. Vamos entonces a revisar únicamente los algoritmos nuevos.

La clase interna **Iterador** no cambia en absoluto. La clase **Vertice** agrega los métodos de la interfaz **ComparableIndexable** y además ahora tiene una lista de vecinos, no de vértices. Fuera de eso no cambia.

Los métodos `setIndice()` y `getIndice()` son triviales; el método `compareTo()` tampoco es difícil, pero tiene que manejar pesos *infinitos* (por cómo implementaremos nuestros algoritmos). Dado que los pesos tienen que ser mayores que cero (el algoritmo de Dijkstra no funciona con pesos negativos), podemos utilizar un valor negativo para representar el infinito; pero cualquier valor (como `Double.MAX_VALUE`) se puede justificar como válido. Por último el método `compareTo()` debe tomar el infinito como un valor mayor que cualquier otro (excepto infinito mismo).

La clase **Vecino** básicamente lo único que hace es envolver el comportamiento de **Vertice**, ya que es un adaptador que nada más agrega el vértice vecino y el peso de la arista.

- **Vecino()**

Define el vértice vecino y el peso.

- `get()`

Regresa el elemento del vértice vecino.

- `getGrado()`

Regresa el grado del vértice vecino.

- `getColor()`

Regresa el color del vértice vecino.

- `vecinos()`

Regresa la lista de vecinos del vértice vecino.

Para los métodos de la nueva versión de la clase `Grafica` repetimos que sólo veremos los que sean nuevos o que cambien significativamente.

- `conecta()`

El nuevo método `conecta()` recibe un nuevo parámetro con el peso de la arista, pero se comporta casi idéntico al método `conecta()` original. La única diferencia es que tiene que crear vecinos para agregar en las listas de vecinos en lugar de nada más agregar los vértices vecinos.

- `getPeso()`

Si los elementos no están en la gráfica o no son vecinos ocurre un error. Si están y son vecinos, regresamos el peso de la arista que los une.

- `setColor()`

Este método lo mostramos en el listado [19.3](#).

Estamos dejando de lado varios métodos auxiliares que valen la pena implementar; por ejemplo un método para buscar el vecino (la instancia de `vecino`) de un vértice. Los métodos que calculan las trayectorias los veremos en las siguientes sección.

19.4. Algoritmo de trayectoria mínima

Para calcular la trayectoria mínima entre dos vértices de una gráfica básicamente realizaremos un recorrido BFS en la misma. Sea s el vértice origen y t el vértice destino; si s y t son iguales regresamos una trayectoria con s como único elemento.

Si s y t son distintos, definimos la distancia de todos los vértices de la gráfica como infinito, excepto s cuya distancia será cero; denotaremos con $d(v)$ a la distancia del vértice v . Creamos una cola y metemos s en ella.

Mientras la cola no sea vacía sacamos el siguiente vértice u de la misma; para

todos los vecinos v de u vemos si v tiene distancia infinito. Si es así definimos la distancia de v como la distancia de u más 1 y metemos v a la cola.

Cuando la cola esté vacía todos los vértices tendrán la distancia de longitud mínima al vértice origen; si $d(t)=\infty$ entonces la gráfica es inconexa y los vértices s y t están en componentes conexas distintas: regresamos una trayectoria vacía.

Si t tiene distancia menor a infinito tenemos que reconstruir la trayectoria mínima a partir de él: para esto hacemos lo siguiente: sea u el vértice destino y L una trayectoria con únicamente u . Para todo vecino v de u , si la distancia de v es la distancia de u más 1 (en otras palabras, si $d(v)=d(u)+1$), entonces agregamos v a L y hacemos $u=v$, repitiendo esto hasta que v sea el vértice origen. Regresamos la trayectoria reversa de L .

Debe ser obvio por qué reconstruimos la trayectoria a partir del vértice destino; si tratamos de hacerlo a partir del vértice origen, como todos los vecinos son tales que los pesos de las aristas correspondientes son iguales, no podemos (en tiempo constante) determinar por cuál de ellos debe continuar la trayectoria. En cambio si empezamos por el vértice destino no importa cuál de los vecinos escojamos; siempre y cuando tenga una distancia menor por uno, nos llevará al vértice origen.

Podemos ver la trayectoria mínima de una gráfica en la figura [19.3](#).

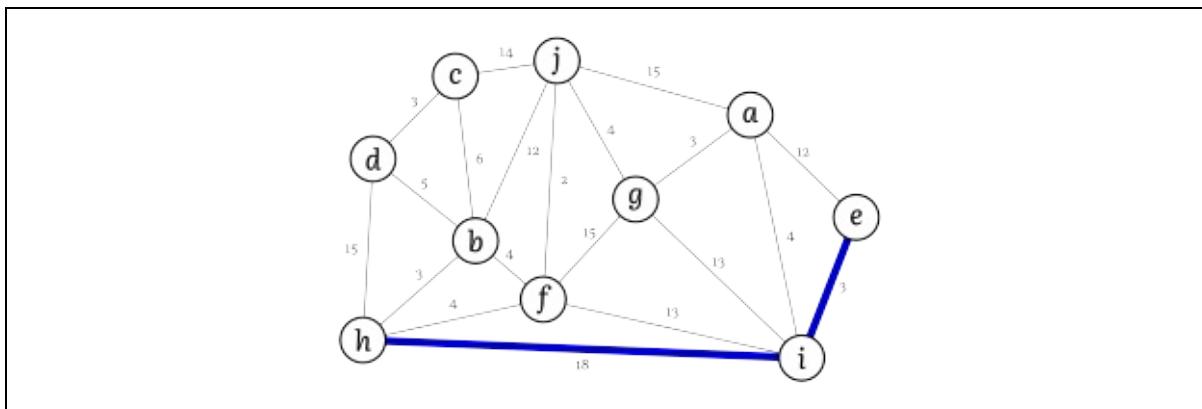


Figura 19.3: La trayectoria mínima de una gráfica; en este caso es la única.

Como el algoritmo es básicamente BFS (nada más utilizando las distancias en lugar de colores para no repetir vértices), la complejidad en tiempo del mismo es $O(|V|+|E|)$ y su complejidad en espacio es $O(n)$.

19.5. Algoritmo de Dijkstra

El algoritmo de Dijkstra fue propuesto por Edsger Dijkstra en 1956 [\[10\]](#) y es de los algoritmos más utilizados e implementados en existencia; o al menos

versiones derivadas del mismo.

El algoritmo recibe vértices origen y destino (denotados por s y t) y define la distancia de todos los vértices de la gráfica como infinito, excepto por s cuya distancia se define en cero. Todos los vértices (incluyendo s) se utilizan para crear un montículo en tiempo $O(n)$; podemos usar nuestros montículos mínimos o nuestros montículos de arreglos, de lo cual dependerá la complejidad en tiempo del algoritmo. Recordemos que los vértices de nuestra gráfica serán comparables (aunque los elementos de la gráfica no lo sean), utilizando como métrica de comparación su propiedad de distancia.

Mientras el montículo no sea vacío se elimina la raíz del mismo; sea u el vértice recién eliminado del montículo. Para cada vecino v de u verificamos que la distancia de v sea mayor que la distancia de u más $w(u, v)$; en otras palabras verificamos si $d(v) > d(u) + w(u, v)$. Si éste es el caso actualizamos la distancia de v a la distancia de u más $w(u, v)$ (en otras palabras, $d \leftarrow d(u) + w(u, v)$), sin olvidarnos de reordenar el montículo.

Cuando el montículo esté vacío todos los vértices de la gráfica tendrán su distancia de peso mínimo al vértice origen; si el vértice destino tiene distancia infinita entonces la gráfica es inconexa y los vértices origen y destino están en componentes conexas distintas. Regresamos una trayectoria vacía en este caso.

Si el $d(t) < \infty$ tenemos que reconstruir la trayectoria de peso mínimo a partir de él; para esto hacemos lo siguiente: sea u el vértice destino y L una trayectoria únicamente con u . Para todo vecino v de u , si la distancia de v es igual a la distancia de u más $w(u, v)$, entonces agregamos v a L y hacemos $u = v$, repitiendo esto hasta que v sea el vértice origen. Regresamos la trayectoria reversa de L .

Al igual que con el algoritmo para calcular una trayectoria mínima, debemos reconstruir la trayectoria de peso mínimo comenzando por el vértice destino, porque de otra forma puede ocurrir que no sepamos por qué vecino continuar construyéndola.

Podemos ver la trayectoria de peso mínimo de una gráfica en la figura [19.4](#).

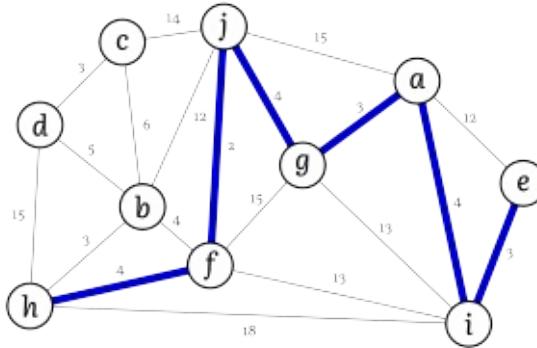


Figura 19.4: Trayectoria de peso mínimo de una gráfica.

La complejidad en tiempo del algoritmo está determinada por las operaciones de eliminar el mínimo y reordenar un elemento del montículo. Sea T_e el número de instrucciones que se necesitan para eliminar el elemento mínimo del montículo y T_r el número de instrucciones que se necesitan para reordenar un elemento en el montículo.

Todos los vértices de la gráfica son eliminados exactamente una vez del montículo, por lo que el total de instrucciones para eliminar los elementos son del orden de $O(|V|T_e)$. Para cada vértice, además, recorremos *todas* sus aristas y (en el peor de los casos) reordenamos *todos* los vecinos del vértice; por lo tanto se ejecutan T_r instrucciones por cada arista de la gráfica, lo que es del orden de $O(|E|T_r)$. Por lo tanto la complejidad en tiempo del algoritmo es $O(|V|T_e + |E|T_r)$.

Si usamos nuestros montículos de arreglos eliminar toma tiempo $O(n)$ y reordenar toma tiempo $O(1)$, por lo que la complejidad en tiempo del algoritmo es $O(|V|^2 + |E|) = O(n^2)$; así estaba descrito el algoritmo en su versión original de 1956 [10].

Si usamos nuestros montículos mínimos eliminar y reordenar toman ambas tiempo $O(\log n)$, por lo que la complejidad en tiempo del algoritmo es $O((|V|+|E|)\log n)$. En el peor de los casos, si la gráfica tiene un número cuadrático de aristas y además casi todas las arista reordenan un vértice al recorrerlas, esta complejidad es equivalente a $O(n^2 \log n)$; sin embargo, si la gráfica tiene un número lineal de aristas (por ejemplo, cuando la gráfica es plana) entonces la complejidad se reduce a $O(n \log n)$.

En 1987 Michael Fredman y Robert Tarjan introdujeron los montículos de Fibonacci (*Fibonacci heaps*) [15], que son de hecho un bosque o colección de árboles (no necesariamente binarios) que cumplen la propiedad de los montículos mínimos: cada vértice es menor o igual que todos sus hijos. Los montículos de Fibonacci pueden eliminar el elemento mínimo en tiempo

$O(\log n)$ y reordenar (hacia abajo, como lo requiere el algoritmo de Dijkstra) en tiempo $O(1)$ amortizado. Esto resulta en una complejidad en tiempo de $O(|E|+|V|\log|V|)$ para el algoritmo de Dijkstra, al menos en teoría [9].

En la práctica los montículos de Fibonacci son muy complicados de implementar y no se comportan tan rápido como sus complejidades en tiempo sugieren. Los montículos mínimos y de arreglos en cambio son relativamente sencillos de implementar, así que usaremos una combinación de ambos; si el número de aristas es mayor a $\frac{n(n-1)}{2} - n$ utilizaremos montículos de arreglos; en otro caso utilizaremos montículos mínimos. De esta manera existirán gráficas en las cuales el algoritmo de Dijkstra tomará tiempo $O(n^2 \log n)$ en ejecutarse, pero serán relativamente pocas y además ocurrirá sólo en casos particularmente malos de las mismas, donde el orden de los vecinos en cada vértice esté dado de tal forma que haya que ordenar muchos vértices en el montículo, casi tantos como aristas. Esto nos parece un compromiso razonable, especialmente si nos ahorra implementar otro tipo de montículos; además, de esta manera garantizamos una complejidad en tiempo $O(n \log n)$ para gráficas planas.

En espacio el algoritmo es obviamente lineal, por el espacio consumido por el montículo mínimo.

19.6. Reconstruyendo trayectorias

Para terminar con el capítulo mencionaremos un detalle técnico que nos parece interesante; debieron notar que reconstruir tanto la trayectoria mínima como la trayectoria de peso mínimo son de hecho el mismo algoritmo, excepto por la condición que se utiliza para elegir el vecino por el cuál continuar.

La trayectoria mínima elige al vecino cuya distancia sea la distancia del vértice actual menos uno; la trayectoria de peso mínimo elige al vecino cuya distancia sea la distancia del vértice actual menos el peso de la arista que los une.

Esta manera de elegir es justamente lo que modela la interfaz funcional **BuscadorCamino**: tiene un método que recibe dos vértices y regresa verdadero si se siguen en la trayectoria o falso en otro caso. Esto nos permite implementar la reconstrucción de los dos tipos de trayectorias utilizando un único método auxiliar que recibe como parámetro una lambda que determina cómo elegir al vecino por el cuál continuar.

No es necesario utilizar la interfaz, pero hace el código más conciso y nos parece también más elegante.

19.7. Pesos con matrices de adyacencias

Al final del capítulo [17](#) mencionábamos que se pueden implementar gráficas con matrices de adyacencias. Si este es el diseño que se decide utilizar, entonces se puede modificar trivialmente para agregarles pesos a las aristas; en lugar de tener una matriz de enteros, tendríamos una matriz de números de punto flotante (en Java probablemente de tipo `double`) con ceros en las entradas de las aristas no existentes y el peso de las mismas cuando sí existieran.

Por ejemplo, la matriz de adyacencias con pesos en las aristas correspondiente a la gráfica de la figura [19.1](#) se puede ver en el cuadro [19.1](#).

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>
<i>a</i>	0	0	0	0	12	0	3	0	4	15
<i>b</i>	0	0	6	5	0	4	0	3	0	12
<i>c</i>	0	6	0	3	0	0	0	0	0	14
<i>d</i>	0	5	3	0	0	0	0	15	0	0
<i>e</i>	12	0	0	0	0	0	0	0	0	3
<i>f</i>	0	4	0	0	0	0	15	4	13	2
<i>g</i>	3	0	0	0	0	15	0	0	13	4
<i>h</i>	0	3	0	15	0	4	0	0	18	0
<i>i</i>	4	0	0	0	0	13	13	18	0	0
<i>j</i>	15	12	14	0	3	2	4	0	0	0

Cuadro 19.1: La matriz de adyacencias de la gráfica en la figura [19.1](#).

Las mismas consideraciones que se vieron en el capítulo [17](#) aplican aquí, aunque sin duda acceder el peso de una matriz de adyacencias siempre será más rápido que de una lista de adyacencias, obviamente.

Ejercicios

1. Actualiza tu implementación de la clase `Grafica` para que soporte gráficas

ponderadas y agrégale los métodos faltantes cubiertos en este capítulo.

2. ¿Cuál es la complejidad en tiempo del algoritmo de Dijkstra si utilizamos montículos mínimos?
3. ¿Cuál es la complejidad en tiempo del algoritmo de Dijkstra si utilizamos arreglos?
4. ¿Cuál es la complejidad en tiempo del algoritmo de Dijkstra si utilizamos montículos de Fibonacci?

20. Funciones de dispersión

En esencia una función de dispersión (*hash function* en inglés) es una función que recibe un número arbitrario de bits y regresa un número fijo de bits. Una función de dispersión que se considere buena en general cumplirá que dos entradas distintas generarán salidas distintas (aunque es imposible garantizar esto siempre, como veremos más adelante) y que si dos entradas son muy similares (por ejemplo, que difieran únicamente en un bit) entonces las salidas correspondientes serán significativamente diferentes. En otras palabras es como si *dispersaran* su entrada y de ahí su nombre en español.

Como en general las computadoras trabajan con al menos un byte (ocho bits), en la práctica las funciones de dispersión se implementan recibiendo un número arbitrario de bytes (generalmente en un arreglo) y regresando un número fijo de bytes (generalmente también en un arreglo, pero nosotros usaremos un entero de 32 bits, 4 bytes).

Las funciones de dispersión son la base sobre la cual se sientan muchos conceptos de criptografía y seguridad en cómputo; en ese contexto pueden ser extremadamente complejas [29]. Nosotros las utilizaremos de una manera mucho más sencilla; para poder acomodar entradas en nuestros diccionarios del capítulo 21.

20.1. Colisiones en funciones de dispersión

Definiremos formalmente una función de dispersión de la siguiente manera:

Definicion 20.1. (Funciones de dispersión) *Una función de dispersión estará definida como una función $f: \mathbb{N} \rightarrow \mathbb{N}_{2^k}$, donde $\mathbb{N}_{2^k} = \{n \in \mathbb{N} | n < 2^k\}$, $k > 0$ fija. Diremos que f tiene tamaño k .*

Dada una entrada a describiremos como *dispersar a* el aplicarle f y a $f(a)$ como *la dispersión de a bajo f* o sencillamente *la dispersión de a* si f queda claro en el contexto.

El tamaño de la función es el número de bits que tiene su salida; como mencionábamos arriba, en la práctica 2^k será siempre un múltiplo de 8.

Las colisiones en las funciones de dispersión estarán definidas como sigue:

Definicion 20.2. (Colisiones en funciones de dispersión) *Sean f una función de dispersión y $a, b \in \mathbb{N}$. Si $a \neq b$ y $f(a) = f(b)$, diremos que a*

y b son una colisión en f.

En otras palabras, una colisión en f serán dos entradas distintas cuya dispersión sea idéntica. Como hicimos notar al inicio del capítulo, es *imposible* tener una función de dispersión para valores *arbitrarios* que no tenga colisiones. Para exemplificarlo tomemos un ejemplo extremo: sea f una función de dispersión de tamaño 1.

Esto quiere decir que los únicos valores que f puede generar son 0 y 1; por lo tanto, sólo podemos tener 2 valores de entrada que no produzcan colisiones en f . A partir de 3 valores siempre tendremos colisiones.

Como una función de dispersión *siempre* tiene tamaño fijo k , al momento de tener $2^k + 1$ valores ya es imposible garantizar que no habrá colisiones. Por supuesto 2^k comienza a ser un número enorme para valores de k no muy grandes; pero aún así las colisiones en las funciones de dispersión son sencillamente algo que ocurre: en general no podemos garantizar que no van a pasar y hay que planear entonces qué hacer cuando lo hagan.

Si el dominio de la función de dispersión es relativamente pequeño y se conoce de antemano, podemos definir una función de dispersión *perfecta*, donde no existan colisiones; sin embargo esto es una situación muy particular y no cambia el hecho de que una función de dispersión que tenga que trabajar con valores arbitrarios tendrá colisiones.

Como mencionamos al inicio del capítulo, las funciones de dispersión son la base de la criptografía y la seguridad en cómputo; por ejemplo, en los sistemas operativos basados en Unix las contraseñas de los usuarios nunca son guardadas tal cual las escriben sus dueños. En cambio, cuando el usuario escribe su contraseña la computadora guarda únicamente su dispersión; de esta manera al momento en que el usuario trate de acceder al sistema, la computadora le pedirá su contraseña, dispersará lo que el usuario escriba y lo comparará con la dispersión que haya guardado.

Por eso es que son importantes las colisiones en funciones de dispersión en un contexto criptográfico: si el sistema Unix utilizara una función de dispersión muy mala (con muchas colisiones), existiría la posibilidad de que dos usuarios con contraseñas distintas tuvieran la misma dispersión de las mismas y por lo tanto que uno pudiera acceder la cuenta del otro.

Por supuesto los sistemas Unix suelen utilizar muy buenas funciones de dispersión; pero además las contraseñas en general tienen un tamaño de unos cuantos bytes, así que la probabilidad de que ocurra una colisión con dos contraseñas diferentes es astronómicamente pequeña. Qué tan pequeña es esa probabilidad se puede calcular con bastante precisión; pero eso es parte del material de un curso de Criptografía y definitivamente fuera del alcance de

este libro.

En este capítulo cubriremos los algoritmos para tres funciones de dispersión relativamente sencillas y definitivamente no adecuadas para usos criptográficos, pero más que aceptables para nuestros diccionarios en el siguiente capítulo. Antes de ver estos algoritmos discutiremos el diseño que usaremos para nuestra implementación.

20.2. Implementación en Java

Las funciones de dispersión en Java presentan el ligero problema de que el lenguaje de programación no ofrece funciones. Vamos a eludir este problema de la manera obvia, implementando nuestras funciones de dispersión utilizando métodos estáticos en una clase auxiliar **Dispersores**. Podemos ver el esqueleto de esta clase en el listado 20.1; al igual que con la clase **Arreglos** del capítulo 7 la clase tendrá un único constructor privado para evitar que se instancien objetos de ella. El tener definido cada algoritmo para arreglos de bytes nos permitirá probar que funcionen de forma idéntica a las versiones correspondientes en el lenguaje de programación C, que son las originales.

```
public class Dispersores {  
  
    private Dispersores() {}  
  
    public static int  
    dispersaXOR(byte[] llave) { /* ... */ }  
  
    public static int  
    dispersaBJ(byte[] llave) { /* ... */ }  
  
    public static int  
    dispersaDJB(byte[] llave) { /* ... */ }  
}
```

Listado 20.1: Esqueleto de la clase **Dispersores**, donde estarán las implementaciones de bajo nivel de nuestros dispersores.

Sin embargo esto nada más nos deja con métodos (estáticos) de bajo nivel que reciben un arreglo de bytes y regresan un entero de 32 bits que son realmente 4 bytes combinados. Queremos poder evaluar nuestras funciones de dispersión en distintos objetos de Java utilizando métodos de alto nivel que reciban dichos objetos como parámetros.

Para esto vamos a utilizar una interfaz funcional **Dispersor** que nuestros dispersores de alto nivel implementarán. Esta interfaz es funcional para poder

utilizar lambdas al momento de crear nuestros dispersores; esto será útil en el siguiente capítulo. Podemos ver la interfaz en el listado 20.2.

```
@FunctionalInterface
public interface Dispersor<T> {
    public int dispersa(T objeto);
}
```

Listado 20.2: La interfaz `Dispersor`.

Para la implementación concreta de nuestros dispersores de alto nivel nos concentraremos únicamente en dispersores de cadenas (que tienen un conveniente método `getBytes()`) y utilizaremos el patrón *Método Fábrica* (*Factory Method*). El patrón *Método Fábrica*, como su nombre indica, utiliza un método estático que fabrica objetos, en este caso instancias de la interfaz `Dispersor`; hacemos esto porque nos permite tener distintas implementaciones de dispersores con un único método estático por tipo y además sin tener que preocuparnos de las clases concretas que las proveen. Tendremos una clase `FabricaDispersores` con un único método estático `dispersorCadena()`, que recibirá el elemento de una enumeración para determinar qué algoritmo implementará el dispersor. Podemos ver la clase `FabricaDispersores` en el listado 20.3. Usando el método `getBytes()` de la clase `String` el método `dispersorCadena()` se vuelve trivial, así que lo mostramos ya escrito; como todas las clases que no van a instanciar objetos, le agregamos un único constructor privado.

```
public class FabricaDispersores {

    private FabricaDispersores() {}

    public static Dispersor<String>
    dispersorCadena(AlgoritmoDispersor algoritmo) {
        switch (algoritmo) {
            case XOR_STRING:
                return c ->
                    Dispersores.dispersaXOR(c.getBytes());
            case BJ_STRING:
                return c ->
                    Dispersores.dispersaBJ(c.getBytes());
            case DJB_STRING:
                return c ->
                    Dispersores.dispersaDJB(c.getBytes());
            default: throw new IllegalArgumentException();
        }
    }
}
```

Listado 20.3: La clase `FabricaDispersores`.

20.2.1. Cascando huevos

En dos de las funciones de dispersión que implementaremos tenemos que combinar bytes en un entero. Existen dos esquemas canónicos para combinar 4 bytes en un entero de 32 bits: *big-endian* y *little-endian*. La diferencia radica en qué byte consideramos el más significativo al irlos leyendo ya sea de memoria, disco duro o una conexión en red; si el primer byte recibido es el más significativo, estamos usando *big-endian*; si el primer byte recibido es el *menos* significativo, estamos usando *little-endian* (figura 20.1). Como estamos trabajando con bytes, utilizaremos números hexadecimales en todos nuestros ejemplos, que es lo que se suele hacer en computación: un byte son siempre dos dígitos hexadecimales.

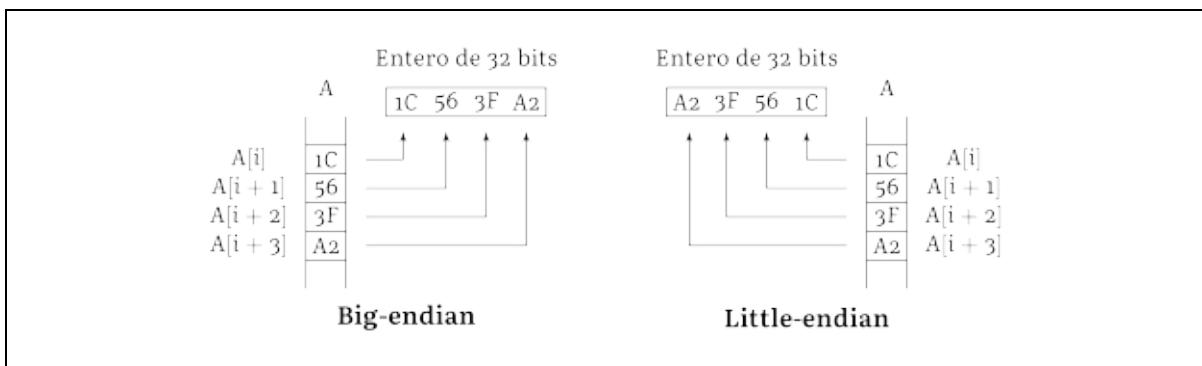


Figura 20.1: Juntando 4 bytes en un entero de 32 bits usando *big-endian* y *little-endian*.

Los términos *big-endian* y *little-endian*¹ vienen de la novela satírica *Los Viajes de Gulliver* de Jonathan Swift; en ella una guerra civil se desata porque algunos prefieren cascar sus huevos cocidos en el extremo pequeño (*little end*) y otros prefieren hacerlo en el extremo grande (*big end*). El término en computación lo introdujo Danny Cohen en 1980 (publicado en 1981 [8]), justamente para tratar de “evitar una guerra” entre facciones que discutían qué byte (o bit) debía ser el más significativo.

Por supuesto la guerra se dio de cualquier manera y hoy en día distintos sistemas utilizan tanto *little-endian* como *big-endian*. Los sistemas operativos basados en Unix (y por ende Internet también) funcionan en su mayoría utilizando *big-endian*; pero múltiples arquitecturas (destacadamente la arquitectura Intel y todas las derivadas de la misma) utilizan *little-endian*.

En general el sistema operativo se encarga de manejar automáticamente las conversiones necesarias, pero como nuestros algoritmos para funciones de dispersión trabajan con bytes directamente, los mismos tienen que tomar una decisión de qué esquema utilizar. Reflejando el estado actual (y probablemente permanente) de esta guerra de esquemas, uno de nuestros

algoritmos utiliza *big-endian* y otro *little-endian*.

En el lenguaje de programación C el combinar cuatro bytes en *big-endian* se puede realizar con el operador de desplazamiento de bits (`<<`) y el operador OR de bits (`|`): desplazamos el primer byte 24 bits (3 bytes), el segundo 16 bits (2 bytes), el tercero 8 bits (1 byte) y el cuarto no lo desplazamos. Hacemos OR de bits de los enteros resultantes para cada desplazamiento y con esto tenemos nuestro entero de 32 bits; podemos ver un ejemplo en el listado 20.4.

```
int
combina(uint8_t a, uint8_t b, uint8_t c, uint8_t d) {
    return (a << 24) | (b << 16) | (c << 8) | d;
}
```

Listado 20.4: Combinamos cuatro bytes en un entero de 32 bits en el esquema *big-endian* (suponiendo que a es el primer byte leído).

Esto es trivial en C porque el lenguaje de programación tiene tipos enteros *sin signo* (como `uint8_t` en el ejemplo del listado 20.4). Lamentablemente esto *no* es tan sencillo en Java, porque no tenemos enteros sin signo en el lenguaje. Esto quiere decir que el tipo `byte` tiene signo lo cual genera problemas al querer ver un byte únicamente como 8 bits.

Si el byte tiene prendido el bit más significativo, entonces Java lo considera negativo siguiendo las reglas de complemento a 2; si convertimos el `byte` a un `int` (lo cual ocurre con todos los operadores de bits del lenguaje), Java va a preservar el signo, lo cual deja prendido el bit más significativo y, como un `byte` es muy pequeño (en el rango $[-128, 127]$), por el complemento a 2 el entero de 32 bits equivalente tendrá prendidos casi todos los bits.

Por poner un ejemplo, el byte *F3* en Java *siempre* es el valor -13 por el complemento a 2 de 8 bits: *F3* es 11110011 en binario y $\neg(11110011 - 00000001) = \neg11110010 = 00001101$, que es *0D* en hexadecimal y 13 en decimal; la magnitud del valor negativo que representa *F3* en complemento a 2 (y sabemos que es negativo porque su bit más significativo está prendido).

Al convertir *F3* a un entero de 32 bits, Java por omisión mantiene el valor de -13 del mismo; pero -13 en complemento a 2 de 32 bits es *FFFFFFF3* por la misma razón que es *F3* en complemento a 2 de 8 bits. Como hacer OR de bits o hacer desplazamientos (`|` y `<<` respectivamente) convierte automáticamente un `byte` en un `int`, en Java no podemos nada más hacer como en el listado 20.4 porque se prenderían bits que no deben prenderse.

Para evitarnos este problema, lo que vamos a hacer es realizar un AND de bits

sobre el `byte` con el entero $000000FF = FF$, lo cual automáticamente convierte el resultado a un entero de 32 bits pero con los primeros 24 bits apagados. Teniendo así el valor ya podemos hacer otras operaciones sin problemas; el equivalente del listado [20.4](#) para Java lo podemos ver en el listado [20.5](#).

```
public int combina(byte a, byte b, byte c, byte d) {
    return ((a & 0xFF) << 24) | ((b & 0xFF) << 16) |
           ((c & 0xFF) << 8) | ((d & 0xFF));
}
```

Listado 20.5: Combinamos cuatro bytes en un entero de 32 bits en el esquema *big-endian*.

Para desplazar un entero a la derecha ocurre el mismo problema; el desplazamiento *aritmético* de bits a la derecha ($>>$) de Java *preserva* el bit más significativo para no perderlo. Por suerte también podemos desplazar el bit más significativo utilizando el desplazamiento *lógico* a la derecha ($>>>$).

20.3. Función de dispersión XOR

La primera función de dispersión que veremos es bastante mala, pero es muy sencilla de implementar y servirá para exemplificar cómo *no* debe ser una función de dispersión.

El algoritmo funciona como sigue: si el número total de bytes en el arreglo no es un múltiplo de 4, lo completaremos a un múltiplo de 4 agregando ceros al final del mismo.

Inicializaremos una variable entera r en 0 y para cada cuatro bytes del arreglo los combinaremos en un entero n de 32 bits siguiendo el esquema *big-endian*. Actualizamos r a $r \oplus n$.

Una vez recorrido el arreglo, regresamos r . Podemos ver una representación gráfica del algoritmo en la figura [20.2](#).

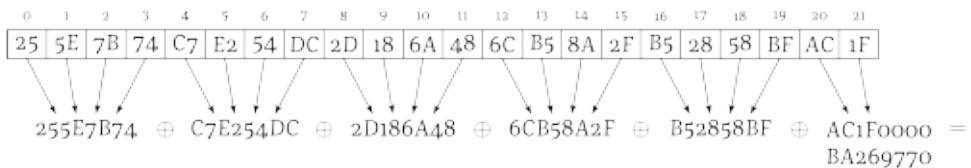


Figura 20.2: La función de dispersión XOR.

La función de dispersión XOR es bastante mala porque es trivial generar colisiones para ella; nada más hay que tomar 8 bytes y rotarlos por 4 bytes. Tomemos los siguientes dos arreglos de bytes:

$$\begin{aligned} A &= \{03, 1D, F8, 58, 02, 9D, 64, BB\} \\ B &= \{02, 9D, 64, BB, 03, 1D, F8, 58\} \end{aligned}$$

El primer arreglo se combinará en dos enteros, $031DF858$ y $029D64BB$. El segundo arreglo se combinará en los mismos enteros, nada más en el orden inverso: $029D64BB$ y $031DF858$. Como el operador de XOR (\oplus) es simétrico, entonces $031DF858 \oplus 029D64BB = 029D64BB \oplus 031DF858 = 25205987$ y tenemos una manera trivial de obtener colisiones.

No sólo es sencillo encontrar colisiones para el dispersor; además dos entradas que sean muy similares tendrán dispersiones también muy similares. En otras palabras el dispersor no dispersa realmente.

Las otras dos funciones de dispersión que veremos son mucho mejores que XOR.

20.4. Función de dispersión Bob Jenkins

La función de dispersión de Bob Jenkins [21] fue diseñada (contrario a lo que pudiera pensarse) por Bob Jenkins en 1997 utilizando experimentación empírica. Está optimizada para cadenas ASCII de hasta 200 caracteres y en la práctica funciona bastante bien.

El núcleo de la función de dispersión de Bob Jenkins es un algoritmo MEZCLA que (como su nombre indica) mezcla tres bytes realizando distintas operaciones sobre los mismos. La implementación de este algoritmo en el lenguaje de programación C es muy fácil de encontrar en Internet; nosotros lo describiremos usando diagramas.

Por ejemplo, en la figura 20.3 el valor de b se niega (se multiplica por -1) y se suma al de a , reemplazándolo. Esto es equivalente a hacer $a = a + (-b)$; o $a = a - b$; o (en Java en particular) $a -= b$. El valor de b no es modificado.

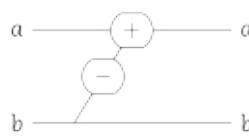


Figura 20.3: Realizando la operación $a -= b$.

De la misma manera, en la figura 20.4 el valor de a se desplaza por 11 bits y se hace XOR con el valor de b , reemplazándolo. Esto es equivalente a hacer $b = b \oplus (a \ll 11)$ o (en Java en particular) $b ^= (a \ll 11)$. El valor de a no es modificado.

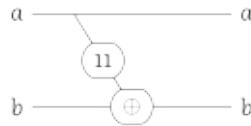


Figura 20.4: Realizando la operación $b \wedge= (a \ll 11)$.

En la figura 20.5 podemos ver el algoritmo MEZCLA dividido en tres partes. En la versión original en C, Bob Jenkins utilizó una macro del preprocesador de C para implementar el algoritmo MEZCLA; en Java no hay macros, así que se tiene que implementar el algoritmo de otra manera. Debe quedar claro que el algoritmo *modifica* su entrada; los bytes a , b y c son actualizados (o mezclados) entre ellos.

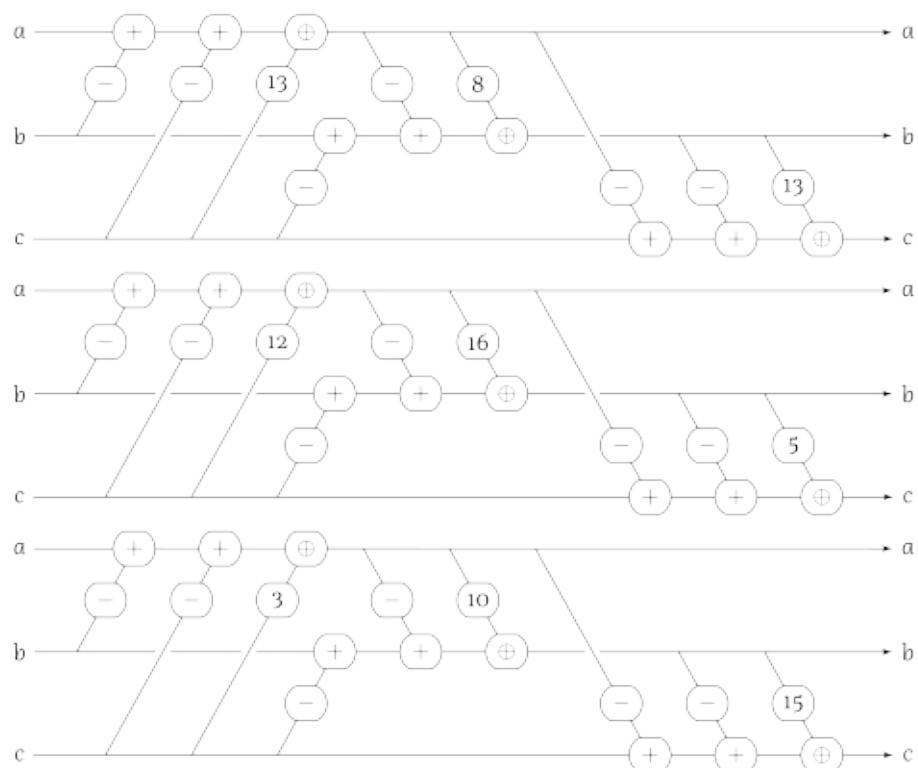


Figura 20.5: La operación de mezcla de la función de dispersión de Bob Jenkins.

También debe quedar claro que se aplican las advertencias que se hicieron arriba sobre el comportamiento de los bytes en Java; hay que tener cuidado de no extender el signo de bytes negativos al convertirlos a enteros de 32 bits.

La función de dispersión de Bob Jenkins funciona así: se inicializan variables a , b y c a los valores $9E3779B9$, $9E3779B9$ y $FFFFFFFF$ respectivamente. El valor $9E3779B9$ es la proporción áurea, pero realmente es un valor arbitrario, lo mismo que $FFFFFFFF$.

Mientras haya disponibles 12 bytes en el arreglo formamos 3 enteros de 32 bits usando el esquema *little-endian* y aumentamos a , b y c con dichos

enteros. Despues los mezclamos y repetimos (figura 20.6).

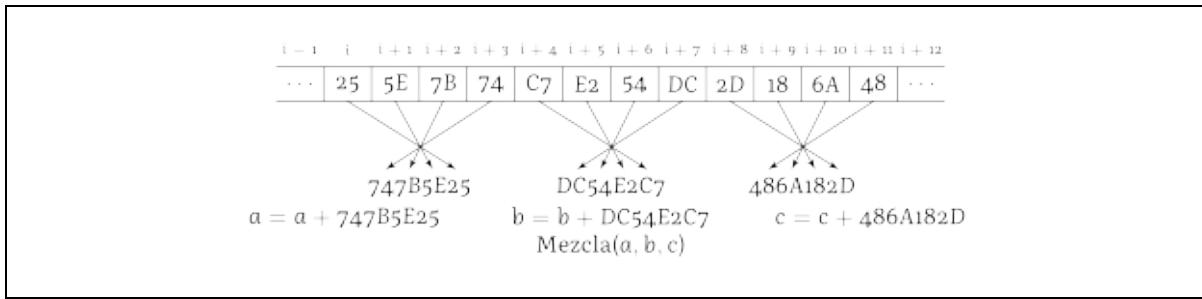


Figura 20.6: Función de dispersión de Bob Jenkins: mientras haya 12 bytes disponibles en el arreglo, se crean 3 enteros con ellos y se suman a los valores a , b y c ; después se mezclan.

Esto se repite mientras queden 12 bytes en el arreglo; cuando queden menos de 12 bytes creamos tanto como podamos de tres enteros con ellos: si queda un byte, digamos $9B$, creamos el entero $0000009B$ y lo sumamos a a ; si quedan dos bytes, digamos $9B, C1$, creamos el entero $0000C19B$ y lo sumamos a a ; si quedan seis bytes, digamos $9B, C1, A5, 07, 8C, D2$, creamos los enteros $07A5C19B$ y $0000D28C$ y sumamos el primero a a y el segundo a b . Como nunca habrá un byte 12, los bytes 9, 10 y 11 (si existen) ocupan los 3 bytes más significativos de c y para ocupar el lugar del byte menos significativo de c , aumentamos c en el tamaño del arreglo; como decíamos arriba, la función está optimizada para llaves menores a 200 bytes, así que el tamaño n del arreglo debería caber en un byte. Pero incluso si $n > 255$ de cualquier manera funciona. Esto siempre lo hacemos; c se incrementa por n siempre.

Por último volvemos a mezclar a , b y c . Podemos ver un ejemplo de cuando quedan exactamente 11 bytes en la figura 20.7.

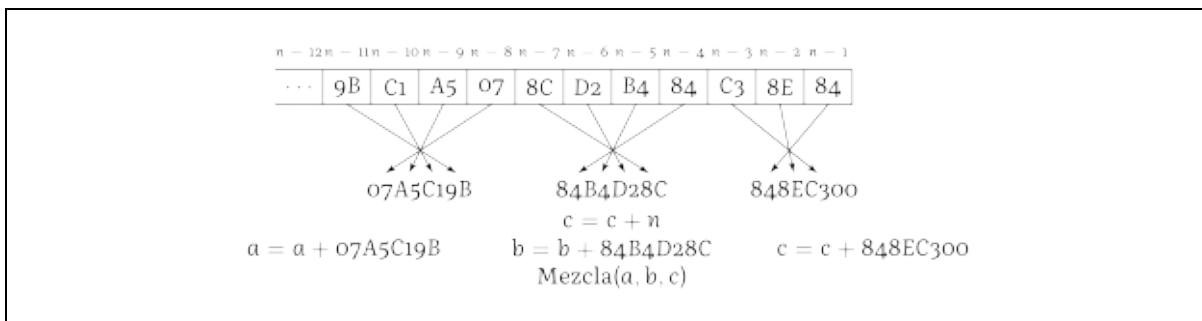


Figura 20.7: Función de dispersión de Bob Jenkins: cuando haya menos de 12 bytes, aumentamos c en n , creamos tanto como podamos de 3 enteros con los bytes que quedan y se suman a los valores a , b y c ; después se mezclan.

La función de dispersión de Bob Jenkins es algo complicada de implementar, pero funciona bastante bien. En la siguiente sección veremos una función de dispersión con un comportamiento similarmente bueno, pero que es mucho más sencilla de implementar.

20.5. Función de dispersión de Daniel J. Bernstein

La función de dispersión DJB2 fue creada por Daniel J. Bernstein a inicios de siglo y al igual que la función de dispersión de Bob Jenkins está optimizada para cadenas. La función utiliza la construcción Merkle–Damgård, sobre la cual está basada la función de dispersión MD5 (MD no viene de ahí, viene de *message digest*) y más recientemente las funciones SHA1 y SHA2. Existe un consenso en general de que DJB2 es de las mejores funciones de dispersión para cadenas que existen; es la que utiliza por omisión la biblioteca GLib para dispersar cadenas [\[35\]](#).

La función es tan sencilla en el lenguaje de programación C que la mostramos en el listado [20.6](#).

```
uint32_t
dispersa_djb(const uint8_t* k, int n)
{
    uint32_t h = 5381;
    for (int i = 0; i < n; i++)
        h += (h << 5) + k[i];
    return h;
}
```

Listado 20.6: Función de dispersión DJB2 en C.

El número mágico 33 ($h += (h << 5)$ es equivalente a $h *= 33$), el por qué funciona mejor que muchas otras constantes, siendo éstas números primos o no, hasta ahora no ha podido ser explicado de forma adecuada [\[36\]](#).

Hacemos notar que traducir la función a Java tiene los mismos problemas que hemos mencionado antes; DJB2 en C también utiliza enteros de 8 bits sin signo como bytes y por lo tanto no es trivial escribirlo en Java.

20.6. Funciones de dispersión para diccionarios

Como hemos dicho a lo largo del capítulo, las funciones de dispersión que hemos visto serán utilizadas para nuestros diccionarios. Para este tipo de usos se les suele denominar funciones de dispersión de búsqueda (*lookup hash function*); la razón será obvia en el siguiente capítulo.

Las funciones de dispersión de búsqueda suelen ser mucho más sencillas que las criptográficas; consecuentemente también suele ser menos importante si de repente se descubre una manera sencilla de generar colisiones para una función de dispersión. En una función de dispersión criptográfica, esto puede

ser catastrófico.

Ejercicios

1. Implementa los métodos faltantes de las clases `Dispersores` y `FabricaDispersores`.
2. ¿Cuál es la complejidad en tiempo de las tres funciones de dispersión vistas?
3. Presenta dos arreglos de bytes distintos que tengan la misma dispersión bajo la función de dispersión XOR.
4. Dado el siguiente arreglo de bytes:

1F	2E	3D	4C	5B	6A	79	00
----	----	----	----	----	----	----	----

combínalos para formar 2 enteros de 32 bits usando *little-endian* y *big-endian*.

5. Dado el siguiente arreglo de bytes:

0F	0E	0D	0C	10	20	30	40
----	----	----	----	----	----	----	----

genera su dispersión XOR.

-
1. Las traducciones al español de *Los Viajes de Gulliver* (dado que el libro es del dominio público desde hace más de un siglo existen decenas de traducciones) suelen ofrecer los términos *ancho-extremista* y *estrecho-extremista* para *big-endian* y *little-endian* respectivamente. Nos negamos a utilizar estos términos.[←](#)

21. Diccionarios

Como mencionamos en el capítulo anterior, las funciones de dispersión que implementamos están pensadas para ser usadas en nuestros diccionarios; son funciones de dispersión de búsqueda. Los diccionarios en inglés se conocen como *hash tables* y de ahí debe ser obvio por qué necesitábamos nuestras funciones de dispersión (*hash* en inglés). Los diccionarios también son conocidos como mapeos (*mappings* en inglés) y obviamente también como la traducción literal: *dictionaries*. El lenguaje de programación Python en particular los denomina de esta manera.

Conceptualmente los diccionarios son una generalización del *concepto* de arreglos; con los arreglos *mapeamos* enteros a objetos. Cuando tenemos un arreglo como el del listado 21.1, conceptualmente estamos mapeando el entero 0 a la cadena “*México*”; el entero 1 a la cadena “*Guadalajara*”; y el entero 2 a la cadena “*Monterrey*”.

```
String[] a = { "México", "Guadalajara", "Monterrey" };
```

Listado 21.1: Un arreglo mapea enteros a objetos.

Por supuesto los arreglos son muy restrictivos; no podemos mapear *cualquier* entero a un objeto: tienen que ser enteros consecutivos y además comenzar en cero. La idea de los diccionarios es generalizar esto no nada más a *cualquier* conjunto de enteros, sino a cualquier conjunto de *objetos*. Formalmente:

Definicion 21.1. (Diccionarios) *Un diccionario es un mapeo de un conjunto K a una colección V. A los elementos de K les llamaremos llaves y a los elementos de V les llamaremos valores.*

Los arreglos son entonces un caso particular de diccionarios, donde $K \subset \mathbb{N}$ (consecutivos comenzando en 0) y V es una colección de algún tipo. Y como los diccionarios son (conceptualmente) generalizaciones de los arreglos, no deberá extrañarnos que los vamos a implementar utilizando justamente arreglos.

Para hacer esto tendremos que (un poco paradójicamente) mapear cada llave de un diccionario a un índice del arreglo; que es justamente donde entrarán nuestras funciones de dispersión. Utilizando una función de dispersión podremos representar cualquier objeto como un entero, que usaremos para encontrar un índice en nuestro arreglo. Por supuesto no podemos utilizar el resultado de la función de dispersión directamente: como el resultado es un

entero de 32 bits en complemento a 2, el mismo varía de $-2,147,483,648$ a $2,147,483,647$ (-2^{31} a $2^{31} - 1$). Vamos a tener que ver cómo transformar el resultado de nuestra función de dispersión a un rango más manejable para poder usarlo como índice en un arreglo.

21.1. El arreglo del diccionario

Lo que nuestro diccionario hará será acomodar nuestros pares de llaves y valores en un arreglo; para esto dispersaremos la llave y esto nos dará un entero a partir del cual debemos ser capaces de encontrar el índice en el arreglo donde estará el valor asociado a dicha llave.

Cómo transformar la dispersión de la llave en el índice de nuestro arreglo tiene varias soluciones; generalmente lo que se hace es crear el arreglo de tal forma que su capacidad sea algún número primo y a la dispersión de la llave hacerle módulo ese número primo. Por qué es buena idea utilizar un número primo es material de un curso de Teoría de Números y definitivamente fuera del alcance de este libro.

Para no tener que estar buscando números primos nosotros utilizaremos una técnica mucho más sencilla y que funciona razonablemente bien: crearemos nuestros arreglos de tal forma que su capacidad sea una potencia de 2 y le aplicaremos una máscara a la dispersión de la llave.

Veamos un ejemplo de esto: supongamos que nuestro arreglo fuera de capacidad $64 = 2^6$. Entonces si usamos la máscara $3F$ (111111 en binario) y hacemos AND de bits de la dispersión de una llave y la máscara, el resultado es un número que tiene prendidos a lo más los 6 bits menos significativos. Como $3F$ es 63 en decimal, esto nos da el rango $[0, 63]$ para los posibles resultados de aplicarle la máscara a una dispersión *independientemente del signo de la misma*: el AND de bits apagará el bit más significativo porque la máscara *siempre* prenderá menos de 32 bits (justamente la idea es usar relativamente pocos bits, no todos).

Podemos ver en el cuadro [21.1](#) las máscaras correspondientes a las primeras 8 potencias de 2, primero en binario, luego en hexadecimal y por último en decimal.

Potencia	Valor	Máscara binaria	Hexadecimal	Decimal
2^1	2	1	1	1
2^2	4	11	3	3

2^3	8	111	7	7
2^4	16	1111	F	15
2^5	32	1 1111	$1F$	31
2^6	64	11 1111	$3F$	63
2^7	128	111 1111	$7F$	127
2^8	256	1111 1111	FF	255

Cuadro 21.1: Máscaras correspondientes a las primeras 8 potencias de 2.

Como se puede ver en el cuadro [21.1](#), la máscara siempre será la capacidad del arreglo menos uno, si la capacidad del arreglo es una potencia de 2. Esto además nos permite crecer fácilmente la capacidad del arreglo: sencillamente duplicamos su capacidad y la máscara será esta nueva capacidad menos uno.

Ya sabemos entonces cómo crearemos y actualizaremos nuestro arreglo; pero aún no hemos explicado de qué tipo será el mismo. Como mencionamos múltiples veces en el capítulo [20](#), las funciones de dispersión en general tendrán colisiones. Si encima tomamos únicamente algunos bits de las dispersiones, entonces el número de llaves que irán a parar a un mismo índice en el arreglo crecerá significativamente.

Por ejemplo: si nuestro arreglo es de capacidad $2^8 = 256$, la máscara correspondiente será FF . Dos valores distintos a y b cuyas dispersiones bajo la función de dispersión f sean $f(a)=AB03FF35$ y $f(b)=7624DE35$, por definición no son una colisión. Sin embargo $AB03FF35 \wedge FF = 35$ y $7624DE35 \wedge FF = 35$, por lo que ambas llaves estarán en el índice 35 (53 en decimal).

Cuando esto ocurra, que dos llaves *distintas* terminen en el mismo índice del arreglo, diremos que es una *colisión de diccionario*. Una colisión en la función de dispersión siempre causa una colisión de diccionario; al revés no es necesariamente cierto.

Hay múltiples maneras de lidiar con colisiones en el diccionario; podemos tratar de enviar cada elemento a entradas aledañas en el arreglo; podemos dispersar la dispersión original (*rehashing* en inglés; y de hecho podemos repetir esto múltiples veces); o podemos hacer que nuestro arreglo sea de estructuras que puedan tener varios elementos de nuestras entradas (a esta técnica se le suele llamar *encadenamiento* o *chaining*).

Nosotros tomaremos la última opción: nuestro arreglo será de listas y las listas a su vez serán de *entradas*, donde una entrada será una llave y su valor asociado. Siempre y cuando la longitud de todas las listas sea relativamente pequeña (siendo 1 el caso ideal), las operaciones de agregar, eliminar y buscar las podremos realizar con complejidad en tiempo $O(1)$.

Lo que tenemos que pagar por estas complejidades en tiempo (además de que son amortizadas; si tenemos que crecer el arreglo del diccionario necesitaremos tiempo $O(n)$) es básicamente espacio: los diccionarios ocuparán *siempre* más memoria de la que se necesitaría únicamente de tener los pares de llaves y valores. Sin embargo la complejidad en memoria continuará siendo $O(n)$, nada más que con una constante oculta mayor que con el resto de las estructuras que hemos cubierto.

Para cuantificar la discrepancia entre la capacidad del arreglo y el número de entradas en las listas del mismo usaremos el *factor de carga* (o carga) del diccionario, que será simplemente el número de entradas en el diccionario divididas entre la capacidad del arreglo.

Resultados experimentales [26] muestran que una carga arriba de 0.7 pero menor a 0.8 es en general óptima; una carga mayor resulta en un rápido deterioro de las complejidades en tiempo para buscar llaves, mientras que una carga menor no proporciona ninguna ventaja perceptible al sacrificar más memoria.

Dado que usaremos potencias de 2 para las capacidades de nuestros arreglos, nuestra carga en promedio será al menos de 0.5, pero en algunos casos será tan baja como 0.25. No vamos a permitir que la cargue llegue nunca a 0.72. Independientemente de la carga, la capacidad *mínima* de nuestro arreglo será $2^6 = 64$.

Un último detalle técnico del arreglo de listas: el mismo *never* tendrá una lista vacía, toda entrada del arreglo será \emptyset hasta que se necesite una lista en ella; hasta ese momento se creará. De la misma manera, si al eliminar una llave del diccionario la lista correspondiente se hace vacía, anularemos la entrada. Podemos ver una representación gráfica del arreglo en la figura 21.1.

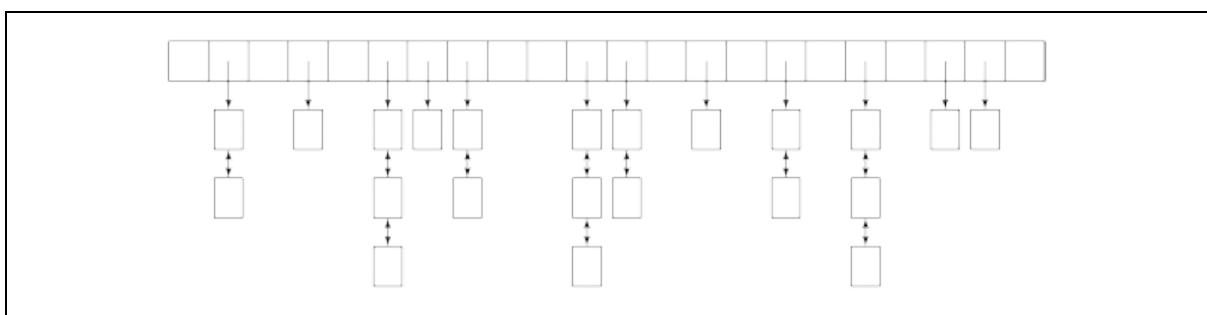


Figura 21.1: Arreglo de listas de entradas en un diccionario.

21.2. Implementación en Java

Necesitamos tipos potencialmente distintos para las llaves y los valores de nuestros diccionarios; esto resultará en que nuestra clase `Diccionario` será doblemente genérica: la primera (y única) de este estilo en el libro.

También tendremos una clase interna privada para las entradas del diccionario, que justamente tendrá una llave (del tipo de las llaves) y un valor (del tipo para los valores).

El que la clase sea doblemente genérica causará que no podamos implementar la interfaz `colección` para nuestros diccionarios; no hay forma de implementar `agrega()` porque recibe un único parámetro y al diccionario siempre debemos agregarles pares: una llave asociada a un valor. Y no podemos usar la llave como valor para sí misma, porque las llaves y los valores pueden ser de tipos distintos (por eso la clase será doblemente genérica).

De cualquier manera la clase `Diccionario` implementará `Iterable` para iterar sus valores; y también tendremos un iterador para llaves. Usaremos el mismo truco que en la clase `Lista`: el método `iterator()` de `Diccionario` regresará el iterador para valores y tendremos un método `iteradorLlaves()` que regresará el iterador para llaves. En este caso sin embargo los iteradores serán de clases internas y privadas distintas, porque serán de tipos (potencialmente) distintos. Como estas clases se comportarán de manera casi idéntica, tendremos una clase interna y privada que implementará el comportamiento en común de ambos iteradores, dejando cosas más específicas en las clases que la hereden. Esto resultará en que la clase `Diccionario` tendrá cuatro clases internas y privadas.

Tendremos dos variables de clase estáticas y finales; una para definir la carga máxima (0.72) y otra para la capacidad mínima ($2^6 = 64$). Cada diccionario tendrá una función de dispersión, su arreglo de listas de entradas y (como ha sido el caso hasta ahora) un contador de cuántos elementos se encuentran en la estructura.

Como nuestro arreglo será de listas y las listas son genéricas, tendremos que suprimir advertencias para crearlo. Contrario a los montículos del capítulo [18](#) no usaremos un arreglo de una interfaz; vamos a utilizar el método `newInstance()` de la clase `Array` del paquete `java.lang.reflect`, para hacerlo más interesante. Podemos ver nuestro método auxiliar `nuevoArreglo()` en el listado [21.2](#).

```
@SuppressWarnings("unchecked")
private Lista<Entrada>[] nuevoArreglo(int n) {
```

```

        return (Lista<Entrada>[])
            Array.newInstance(Lista.class, n);
    }
}

```

Listado 21.2: El método auxiliar `nuevoArreglo()`.

La clase `Diccionario` tendrá cuatro constructores: uno recibirá una capacidad inicial y un dispersor; otro recibirá nada más la capacidad inicial; otro nada más el dispersor; y por último uno no recibirá nada. La capacidad inicial por omisión será $2^6 = 64$ y el dispersor por omisión lo definiremos con una lambda, utilizando el método `hashCode()` de la clase `Object` (listado 21.3).

```

Dispersor<K> dispersor = (K llave) -> llave.hashCode();

```

Listado 21.3: Creando el dispersor por omisión con una lambda.

El método `hashCode()` de la clase `Object` define un dispersor para cada objeto de Java. Varias clases sobrecargan este método con algoritmos de dispersión diversos; algunos son relativamente complicados, como el de la clase `String` que trata a los caracteres de la cadena como entradas de un polinomio; y otros son muy sencillos, como el de la clase `Integer` que sólo regresa el valor del entero.

Sin embargo, la mayor parte de las clases de Java no sobrecargan el método y por lo tanto utilizan la implementación de `Object`, que básicamente regresa la dirección en memoria del objeto¹. Esto resulta en que un objeto con la misma información en memoria produzca una dispersión distinta en diferentes ejecuciones del programa. En general esto no necesariamente causará problemas, pero hay que tenerlo en cuenta.

Tendremos métodos para agregar un par de llave y valor; otro para obtener un valor a través de una llave; otro para verificar si una llave está contenida en el diccionario; y otro para eliminar una entrada a través de su llave. También habrá tres métodos de consulta para obtener el número de colisiones y la colisión máxima en el diccionario, así como la carga del mismo; y otros que son similares a los correspondientes de la interfaz `colección`, aunque los diccionarios no la implementen: nos dirán cuántos elementos hay en el diccionario; si es o no vacío y nos permitirán limpiarlo. Por último tendremos (como siempre) implementaciones para los métodos `equals()` y `toString()`.

Podemos ver el esqueleto de la clase `Diccionario` en el listado 21.4.

```

public class Diccionario<K, V> implements Iterable<V> {

    private class Entrada {
        public K llave;
        public V valor;
    }
}

```

```

        public V valor;
        public Entrada(K llave, V valor) { /* ... */ }
    }

    private class Iterador {
        private int indice;
        private Iterator<Entrada> iterador;
        public Iterador() { /* ... */ }
        public boolean hasNext() { /* ... */ }
        public Entrada siguiente() { /* ... */ }
    }

    private class IteradorLlaves extends Iterador
        implements Iterator<K> {
        public IteradorLlaves() { /* ... */ }
        @Override public K next() { /* ... */ }
    }

    private class IteradorValores extends Iterador
        implements Iterator<V> {
        public IteradorValores() { /* ... */ }
        @Override public V next() { /* ... */ }
    }

    public static final double MAXIMA_CARGA = 0.72;
    private static final int MINIMA_CAPACIDAD = 64;
    private Dispensor<K> dispensor;
    private Lista<Entrada>[] entradas;
    private int elementos;

    @SuppressWarnings("unchecked")
    private Lista<Entrada>[] nuevoArreglo(int n) {
        return (Lista<Entrada>[])
            Array.newInstance(new Lista().getClass(), n);
    }

    public Diccionario() {
        this(MINIMA_CAPACIDAD,
            (K llave) -> llave.hashCode());
    }
    public Diccionario(int capacidad) {
        this(capacidad, (K llave) -> llave.hashCode());
    }
    public Diccionario(Dispensor<K> dispensor) {
        this(MINIMA_CAPACIDAD, dispensor);
    }
    public Diccionario(int capacidad,
                      Dispensor<K> dispensor) {
        /* ... */
    }
}

```

```

public void agrega(K llave, V valor) { /* ... */ }
public V get(K llave) { /* ... */ }
public boolean contiene(K llave) { /* ... */ }
public void elimina(K llave) { /* ... */ }
public int colisiones() { /* ... */ }
public int colisionMaxima() { /* ... */ }
public double carga() { /* ... */ }
public int getElementos() { /* ... */ }
public boolean esVacia() { /* ... */ }
public void limpia() { /* ... */ }
@Override public String toString() { /* ... */ }
@Override public boolean equals(Object o) { /* ... */ }
public Iterator<K> iteradorLlaves() {
    return new IteradorLlaves();
}
@Override public Iterator<V> iterator() {
    return new IteradorValores();
}
}
}

```

Listado 21.4: Esqueleto de la clase **Diccionario**.

21.3. Algoritmos para diccionarios

El constructor de la clase **Entrada** es trivial, así que no lo veremos. La clase **Iterador** es más interesante:

- **Iterador()**

El constructor buscará en el arreglo la primera lista distinta de \emptyset e inicializará su índice a esa entrada en el arreglo y su iterador con el iterador de esa lista. Si ninguna lista es distinta de \emptyset el iterador también será \emptyset .

- **hasNext()**

El método verificará que el iterador no sea \emptyset .

- **siguiente()**

Si el iterador es \emptyset ocurrirá un error. En otro caso se guarda la entrada siguiente del iterador; si el iterador ya no tiene siguiente elemento se buscará la siguiente entrada en el arreglo con una lista distinta de \emptyset y se actualizará el iterador al iterador de esa lista. Si ya no hay entradas en el arreglo distintas de \emptyset , haremos el iterador \emptyset .

Las clases **IteradorLlaves** e **IteradorValores** son básicamente idénticas: ambas llaman al constructor de **Iterador** y ambas en su método **next()** llaman al

método `siguiente()`. En `IteradorLlaves` regresaremos la llave de la entrada regresada y en `IteradorValores` regresaremos el valor.

Veremos los métodos de la clase `Diccionario` en el orden del listado [21.4](#).

- `Diccionario(int capacidad, Dispensor<K> dispensor)`

El constructor inicializa el dispensor con el que se recibe. Si la capacidad recibida es menor que la capacidad mínima la incrementamos a ésta. Buscamos la primera potencia de 2 que sea mayor o igual al *doble* de la capacidad y ésta será la capacidad del arreglo. No tenemos una variable para la máscara porque es la capacidad del arreglo menos 1.

- `agrega()`

Si la llave o el valor recibidos son \emptyset ocurre un error.

Sea i la dispersión de la llave con la máscara aplicada; si el i -ésimo elemento del arreglo es \emptyset , creamos una lista de entradas y la ponemos en el índice i del arreglo. Agregamos una nueva entrada con la llave y el valor en esta lista e incrementamos el contador de elementos.

Si la lista en i sí existe, la recorremos para ver si existe una entrada con la misma llave. Si éste es el caso, reemplazamos el valor de la entrada con el nuevo valor. Si no es el caso, agregamos una nueva entrada con la llave y el valor a la lista e incrementamos el contador de elementos.

Si la carga del diccionario alcanza o excede la carga máxima, doblamos la capacidad del arreglo. Hay que volver a agregar todas las entradas; no funciona nada más copiar las listas del viejo arreglo al nuevo, porque al cambiar la capacidad también cambia la máscara y entonces cada llave queda en entradas potencialmente distintas del arreglo.

- `get()`

Si la llave es \emptyset ocurre un error.

Sea i la dispersión de la llave con la máscara aplicada; si el i -ésimo elemento del arreglo es \emptyset , ocurre un error. Si no buscamos la entrada en la lista con una llave igual a la recibida.

Si la encontramos regresamos el valor de la entrada; si no ocurre un error.

- `contiene()`

Si la llave es \emptyset regresamos falso.

Sea i la dispersión de la llave con la máscara aplicada; si el i -ésimo elemento del arreglo es \emptyset regresamos falso. Si no buscamos la entrada en

la lista con una llave igual a la recibida.

Si la encontramos regresamos verdadero; si no regresamos falso.

- `elimina()`

Si la llave es \emptyset ocurre un error.

Sea i la dispersión de la llave con la máscara aplicada; si el i -ésimo elemento del arreglo es \emptyset , ocurre un error. Si no buscamos la entrada en la lista con una llave igual a la recibida.

Si la encontramos eliminamos la entrada de la lista y decrementamos el contador de elementos; si no ocurre un error.

- `colisiones()`

Regresamos la suma de las longitudes de las listas menos 1 en el arreglo.

- `colisionMaxima()`

Regresamos la longitud de la lista más grande en el arreglo menos 1.

- `carga()`

Regresamos el número de elementos en el diccionario sobre la capacidad del arreglo.

- `getElementos()`

Regresamos el número de elementos en el diccionario.

- `esVacia()`

Verificamos si el número de elementos en el diccionario es 0.

- `limpia()`

Creamos un nuevo arreglo del mismo tamaño del actual y definimos el número de elementos en 0.

- `toString()`

Recorriendo el diccionario con los iteradores de entradas (no de llaves o valores), para cada entrada creamos una cadena con la representación en cadena de la llave entre comillas simples, dos puntos, un espacio, la representación en cadena del valor entre comillas simples, una coma y un espacio. Regresamos esta cadena entre llaves. Por ejemplo, la cadena correspondiente al diccionario que mapea las cadenas `"a"`, `"b"` y `"c"` a 1, 2 y 3 la podemos ver en la figura 21.2.

```
{"a": '1', 'b': '2', 'c': '3', }"
```

Figura 21.2: Representación en cadena de un diccionario.

- `equals()`

Como ha sido con todas las estructuras del curso, necesitamos hacer una audición y suprimir advertencias para convertir el objeto recibido en una instancia del diccionario; podemos ver el inicio del método `equals()` en el listado 21.5.

```
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass())
        return false;
    @SuppressWarnings("unchecked")
    Diccionario<K, V> d =
        (Diccionario<K, V>)o;
    // ...
}
```

Listado 21.5: Inicio del método `equals()` en `Diccionario`.

Verificamos que ambos diccionarios tienen el mismo número de elementos, que las llaves que tienen sean iguales y que el valor asociado a cada llave sea igual en cada diccionario.

21.4. Complejidades en tiempo y en espacio

Los métodos principales de la clase `Diccionario` (`agrega()`, `get()`, `contiene()` y `elimina()`) tienen todos una complejidad en tiempo $O(1)$; *amortizada* y si la función de dispersión es buena.

Lo primero es fácil de ver; como mencionábamos arriba, si al agregar hay que crecer el tamaño del arreglo, la complejidad en tiempo del método brinca de $O(1)$ a $O(n)$. Sin embargo no es descabellado pensar que esto ocurrirá un número constante de veces durante la ejecución de nuestros programas, así que la complejidad en tiempo es $O(1)$ amortizada.

Lo segundo es también fácil de ver con una función de dispersión extremadamente mala: sea f la función de dispersión que hace $f(a)=0$ para todo objeto a . Esto nos envía a todas nuestras entradas a la misma lista y todas nuestras complejidades en tiempo saltan a lineales en ese caso.

En general por supuesto no crecemos muchas veces el tamaño de nuestros arreglos y nuestras funciones de dispersión son relativamente buenas; no sólo no tienen muchas colisiones, sino que de hecho *dispersan* nuestras entradas con una distribución cercana a uniforme. Pero incluso usando una función de dispersión decididamente mala (como nuestra función XOR) y comenzando

con un arreglo con capacidad muy pequeña y agregando muchos elementos, el comportamiento de los diccionarios es razonablemente bueno. Usando una buena función de dispersión y calculando con cuidado de antemano la capacidad que deben tener sus arreglos, son probablemente la manera más eficiente de agregar, acceder y eliminar información.

Por supuesto no son muy eficientes en memoria, pero la misma sigue siendo (en general) lineal. Más grave que esto, no podemos garantizar ningún orden ni entre los valores ni entre las llaves de nuestras entradas. En muchos casos esto no es terriblemente grave, pero sí hay que tenerlo en cuenta.

21.5. Implementaciones alternas de diccionarios

La idea de mapear llaves a valores se puede implementar de múltiples maneras; en lugar de tener nuestras entradas en un arreglo, podríamos tenerlas en un árbol autbalanceable, haciendo las entradas comparables si las llaves a su vez lo son. Esto implicaría complejidades en tiempo de $O(\log n)$, pero nos daría orden en las llaves y además un uso más efectivo de la memoria.

La biblioteca estándar de Java es lo que hace; se tiene una interfaz `Map` (dblemente genérica, como `Diccionario`) y ésta es implementada por la clase `HashMap`, que (toda proporción guardada) funciona de manera muy similar a nuestra clase `Diccionario`. Además está la interfaz `SortedMap` que extiende a `Map`; la clase concreta `TreeMap` la implementa y usa un árbol autbalanceable. Un árbol rojinegro, de hecho.

Los diccionarios (o tablas de dispersión cuando utilizan funciones de dispersión) son utilizados de múltiples formas; en muchos lenguajes de programación interpretados, al hacer `a.p` para acceder la propiedad `p` del objeto `a`, el nombre de la propiedad se usa como una llave en un diccionario de propiedades (hacemos notar que Java *justo no* hace esto; Java hace algo similar a lo que se explicó en el capítulo 1 para acceder las propiedades de sus objetos). Lo mismo ocurre con los métodos: las clases suelen tener una tabla de métodos donde el nombre (junto con los tipos de los parámetros) son la llave para obtener su dirección en memoria. Muchos lenguajes de programación funcionales utilizan a las listas como estructura fundamental; el lenguaje de programación JavaScript hace básicamente lo mismo con diccionarios. Los lenguajes de programación Python y Perl los ofrecen como estructuras nativas, no en la biblioteca estándar. Muchas de las bases de datos graciosamente llamadas NoSQL son básicamente diccionarios que se almacenan en el disco duro.

Todos los lenguajes de programación modernos ofrecen distintas

implementaciones de diccionarios, al menos una que utiliza funciones de dispersión (como la nuestra) y otra ordenada para llaves comparables, generalmente utilizando árboles rojinegros.

Aunque al parecer Hans Peter Luhn escribió un memorando interno en IBM donde describe por primera vez un diccionario usando funciones de dispersión con encadenamiento (mencionado en [\[27\]](#)), lo cierto es que probablemente se dieron implementaciones equivalentes de manera simultánea en distintos lugares del mundo.

De cierta manera, este libro termina en este capítulo; el siguiente capítulo son conjuntos, que son un uso particular de los diccionarios que no quisimos dejar fuera del libro, pero es realmente muy sencillo; y el último capítulo únicamente mejora nuestra clase [Grafica](#), usando una vez más diccionarios.

Entonces realmente la última estructura de datos que vemos son diccionarios y en los últimos capítulos veremos aplicaciones de los mismos.

Ejercicios

1. Implementa los métodos faltantes de la clase [Diccionario](#).
2. ¿Cuál es la complejidad en tiempo de agregar un elemento a nuestra implementación de diccionarios?
3. ¿Cuál es la complejidad en tiempo de eliminar un elemento de nuestra implementación de diccionarios?
4. Para alcanzar sus complejidades en tiempo, ¿qué sacrifican nuestros diccionarios?

-
1. No es tan simple, pero podemos pensar que es lo que hace. [←](#)

22. Conjuntos

Una vez teniendo diccionarios implementar una clase para conjuntos es básicamente trivial. La clase `conjunto` es probablemente la más sencilla (no interna) que veremos en todo el libro.

Como las llaves en los diccionarios funcionan como un conjunto (sólo puede haber una copia de una llave en un diccionario), entonces la clase `conjunto` será únicamente un adaptador de la clase `Diccionario` para que podamos usarlos como una colección. Para esto, los valores asociados al conjunto de llaves serán las mismas llaves.

22.1. Implementación en Java

Como mencionamos arriba los conjuntos serán un adaptador de la clase `Diccionario` para que podamos implementar la interfaz `colección`. Por esto tendremos una variable de clase (la única) para tener una instancia de `Diccionario`.

Los métodos de la clase serán básicamente los de la interfaz `colección`; las únicas operaciones exclusivas de conjuntos que agregaremos serán métodos para unir e intersecar conjuntos. Contrario a casi todas las clases que vimos en el libro, no tendremos una clase interna para los iteradores; vamos a utilizar directamente el iterador de nuestro diccionario.

Podemos ver el esqueleto de la clase `conjunto` en el listado [22.1](#).

```
public class Conjunto<T> implements Colección<T> {
    private Diccionario<T, T> conjunto;

    public Conjunto() { /* ... */ }
    public Conjunto(int n) { /* ... */ }

    @Override public void agrega(T elemento) { /* ... */ }
    @Override public boolean
    contiene(T elemento) { /* ... */ }
    @Override public void elimina(T elemento) { /* ... */ }
    @Override public boolean esVacia() { /* ... */ }
    @Override public int getElementos() { /* ... */ }
    @Override public void limpia() { /* ... */ }
    @Override public String toString() { /* ... */ }
    @Override public boolean equals(Object o) { /* ... */ }
    @Override public Iterator<T> iterator() {
```

```

        return conjunto.iterator();
    }

    public Conjunto<T>
    interseccion(Conjunto<T> conjunto) { /* ... */ }
    public Conjunto<T>
    union(Conjunto<T> conjunto) { /* ... */ }
}

```

Listado 22.1: Esqueleto de la clase **Conjunto**.

22.2. Algoritmos para conjuntos

Como mencionamos antes, la clase **Conjunto** probablemente será la más sencilla de implementar en todo el libro. Todos sus algoritmos (excepto por los métodos `union()` e `interseccion()`) serán sencillamente utilizar el diccionario para realizar las operaciones.

- **Conjunto()** y **Conjunto(int n)**

Los constructores únicamente crean el diccionario, con y sin tamaño predeterminado dependiendo del constructor.

- `agrega()`

El método recibe un elemento e ; sencillamente agregamos la entrada (e, e) al diccionario.

- `contiene()`

Verificamos si el diccionario contiene al elemento recibido como llave.

- `elimina()`

Eliminamos la llave del diccionario.

- `esVacia()`

Verificamos si el diccionario está vacío.

- `getElementos()`

Regresamos el número de elementos del diccionario.

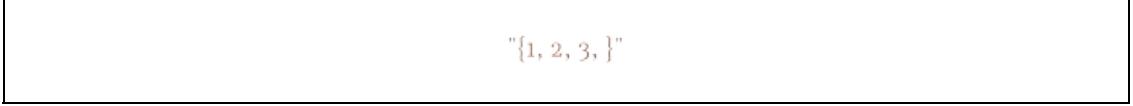
- `limpia()`

Limpiamos el diccionario.

- `toString()`

Regresamos los elementos del conjunto entre llaves y separados por

comas. Por ejemplo, el conjunto $\{1, 2, 3\}$ lo representaremos como se muestra en la figura 22.1.



"{1, 2, 3, }"

Figura 22.1: Representación en cadena de un conjunto.

- `equals()`

Verificamos que los diccionarios de ambos conjuntos sean iguales.

- `interseccion()`

Creamos un conjunto S y recorremos el conjunto que manda llamar el método; si el elemento actual está también en el conjunto recibido, lo agregamos a S . Al terminar regresamos S .

- `union()`

Creamos un conjunto S y recorremos el conjunto que manda llamar el método; agregamos cada elemento a S . Después recorremos el conjunto recibido por el método; agregamos cada elemento a S . Al terminar regresamos S .

Las operaciones de los conjuntos (excepto por `union()` e `interseccion()`) tienen la misma complejidad en tiempo y en espacio que las operaciones correspondientes en diccionarios. Los métodos `union()` e `interseccion()` tienen complejidad en tiempo y en espacio $O(n)$.

22.3. Otros usos de conjuntos

Los conjuntos obviamente se pueden utilizar para modelar conjuntos en nuestros programas, pero tienen otros usos que tal vez no sean tan obvios.

En nuestra clase `Grafica` la clase interna privada `Vertice` tiene colores; y de hecho estos colores se exportan a la interfaz pública `VerticeGrafica`. Y esto es necesario porque de otra manera no podemos implementar cosas como recorridos tipo BFS fuera de la clase.

Con conjuntos podemos ahorrarnos este tipo de propiedades; podemos definir un conjunto de vértices llamado `rojos` y sencillamente agregar a él los vértices que queramos que sean **rojos**. De la misma manera podríamos evitarnos el extender la clase interna protegida `Vertice` de los árboles binarios en nuestros árboles rojinegros.

Algo similar podemos hacer con los árboles AVL, pero en ese caso

necesitaríamos utilizar un diccionario directamente para mapear vértices a enteros y poder guardar así las alturas de los mismos.

Las clases como `Conjunto` existen básicamente por comodidad; podríamos utilizar directamente nuestros diccionarios, pero tienen la ventaja de que podemos usarlos como colecciones.

De la misma manera que podemos implementar diccionarios usando llaves comparables con un árbol rojinegro en lugar de un arreglo, podemos crear conjuntos ordenados usando árboles autabalancables. Una vez más, esto es lo que hace la biblioteca estándar de Java; se tiene una interfaz `Set`, la cual es implementada de forma concreta por `HashSet` que usa un `HashMap` de manera casi exactamente igual que nuestra clase `Conjunto` usa `Diccionario`. También se tiene la interfaz `SortedSet` que extiende la interfaz `Set` y que es implementada concretamente por la clase `TreeSet`, la cual usa la clase `TreeMap` de manera análoga.

Ejercicios

1. Implementa los métodos faltantes de la clase `Conjunto`.
2. Explica por qué nuestros diccionarios no son una colección

23. Mejorando gráficas

Lo último que haremos en el libro será mejorar nuestra clase `Grafica` una vez más. El cambio será muy sencillo: en lugar de tener una lista de vértices, cada uno con una lista de vecinos, tendremos un diccionario de vértices, cada uno con un diccionario de vecinos. Las llaves en ambos casos serán los elementos; los valores serán los vértices y los vecinos respectivamente.

Este cambio lo único que hace es que nuestra gráfica puede agregar, eliminar, conectar y desconectar elementos en tiempo $O(1)$ (amortizado), además de poder contestar si un elemento está o no en la gráfica en tiempo constante también. Hay una ligera penalización en memoria por cambiar nuestras listas por diccionarios, pero consideramos el intercambio ventajoso para nosotros. Y de cualquier forma la complejidad en espacio de la misma estructura se mantiene en $O(n^2)$ (por las aristas).

También se pierde cierto orden; las listas de vecinos estaban en el orden en que hicéramos las conexiones. Ahora el orden dependerá de la función de dispersión que implemente cada objeto en su método `hashCode()` y cómo se acomode en el arreglo interno. No consideramos esto muy grave.

Antes de ver los cambios en el código hacemos notar que, una vez más, esta modificación no altera el comportamiento de nuestra clase `Grafica`; todo el código que la usara antes puede seguirla utilizando de la misma manera (a menos que dependiera del orden de los vecinos en cada vértice). Ningún método se agrega o elimina; todos los cambios ocurren únicamente en la implementación interna. Los métodos siguen cumpliendo lo especificado en capítulos anteriores; únicamente ciertos órdenes son modificados y (por supuesto) ahora todo funciona más rápido.

23.1. Modificaciones al código

Los únicos cambios en nuestro código son que la variable de clase `vecinos` de la clase interna `Vertice` cambia su tipo a `Diccionario<T, Vecino>` y que la variable de clase `vertices` de `Grafica` cambia su tipo a `Diccionario <T, Vertice>`. *Todos* los métodos (ya sea de `Grafica` o alguna de sus clases internas) mantienen la misma firma. Por lo mismo no volvemos a mostrar el esqueleto de la clase.

De la misma manera, *todos* los algoritmos son idénticos, sólo cambia la técnica para buscar un vértice en el diccionario de vértices y cómo buscar un vecino en los diccionarios de vecinos. Esto resulta también en otros cambios

menores; por ejemplo ya no agregamos un vértice a la lista de vértices: ahora agregamos al diccionario de vértices un elemento y el vértice que lo contiene como llave y valor respectivamente. En el algoritmo de Dijkstra ahora creamos el montículo (ya sea `MonticuloMinimo` o `MonticuloArreglo`) usando el constructor que recibe un iterable y un entero (es la razón de ser de esos constructores), porque los vértices ya no son una colección. No repetiremos entonces ninguno de los algoritmos de la clase.

Dependiendo de cómo fuera la implementación original, el número de líneas cambiadas puede ser menor a dos docenas y ninguna representa una modificación conceptualmente importante a los algoritmos implementados.

Pero ahora podemos afirmar que todas las operaciones en nuestras gráficas tienen complejidad en tiempo igual a si hubiéramos utilizado una implementación de matriz de adyacencias (y en varios casos de hecho mejores), sin necesidad de utilizar una matriz.

Con estos cambios las complejidades en tiempo y en espacio de los métodos de `Grafica` son tan buenas como pueden llegar a ser, excepto por `dijkstra()` que podría tener una complejidad en tiempo garantizada de $O(|E| + |V|\log|V|)$ si usáramos montículos de Fibonacci. Sin embargo debemos hacer notar que aunque las complejidades son tan buenas como pueden ser, las constantes ocultas por la notación de la O grandota pueden reducirse todavía más (en algunos casos) usando matrices de adyacencias.

En particular si nuestro programa consiste de una gráfica estática y debemos consultar muchas veces el peso de las aristas, entonces una implementación con matrices de adyacencias probablemente sea lo más conveniente.

Ejercicios

1. Actualiza tu implementación de la clase `Grafica` para que utilice diccionarios de vértices en lugar de una lista de vértices; y para que cada vértice tenga un diccionario de vecinos en lugar de una lista de vecinos.
2. ¿Qué métodos en la clase `Grafica` cambian sus complejidades al cambiar de listas de adyacencias a diccionarios de adyacencias?

24. Conclusiones

En este libro se cubrieron las siguientes estructuras de datos:

- listas doblemente ligadas,
- pilas y colas,
- árboles binarios completos,
- árboles binarios ordenados,
- árboles rojinegros,
- árboles AVL,
- gráficas con listas y diccionarios de adyacencias,
- montículos mínimos,
- diccionarios y
- conjuntos.

Además de los algoritmos asociados a cada una de estas estructuras, se cubrieron algoritmos de ordenamiento y búsqueda en arreglos y listas, el algoritmo de Dijkstra, tres distintas funciones de dispersión y algunos más como el algoritmo de trayectoria mínima y HEAPSORT. También se cubrió, de una manera muy ligera, lo que es la complejidad computacional y la notación de O grandota.

Todas estas estructuras (con la excepción de gráficas) tienen implementaciones en la biblioteca estándar de Java; pero nos parece fundamental que los estudiantes de un curso práctico de Estructuras de Datos las implementen, especialmente bajo la restricción de que deben mantener las complejidades en tiempo y en espacio apropiadas.

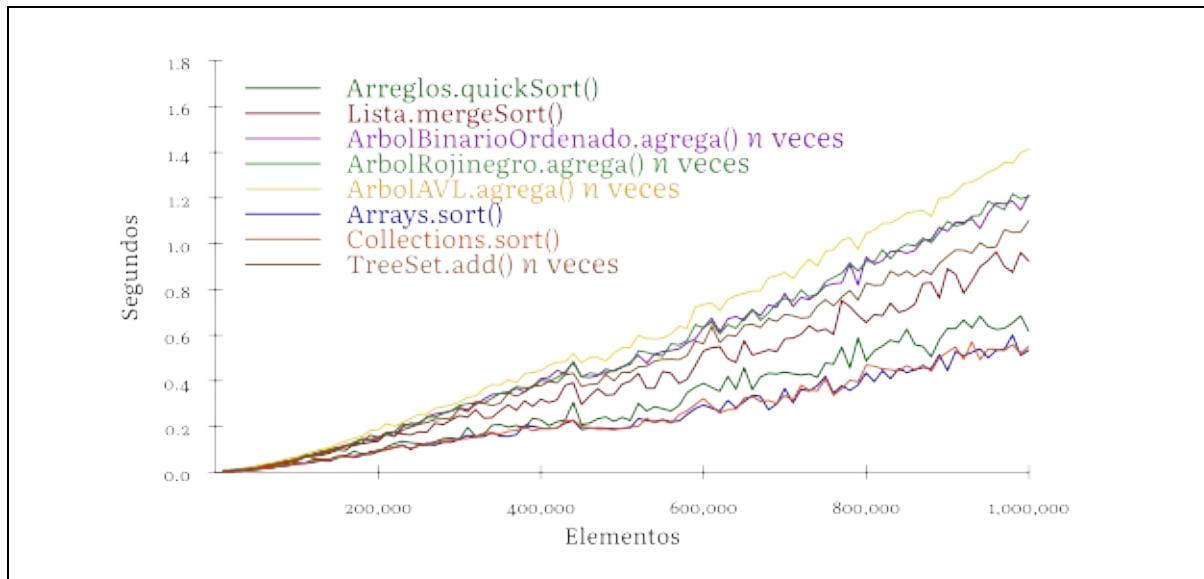


Figura 24.1: Tiempos para ordenar hasta 1,000,000 de elementos aleatorios.

También hacemos notar que las estructuras que los estudiantes deben escribir al seguir el libro, aunque sólo pensadas como ejercicios escolares y por lo tanto sin las optimizaciones con las que usualmente cuentan las implementaciones de una biblioteca estándar, tienen un comportamiento equiparable con éstas. Por ejemplo la figura 24.1 muestra los tiempos promedio de ordenar 10 veces colecciones de hasta 1,000,000 enteros aleatorios con los métodos `quicksort()` de `Arreglos`, `mergeSort()` de `Lista`, `sort()` de `Arrays`, `sort()` de `Collections`, así como usando los métodos `agrega()` de `ArbolBinarioOrdenado`, `ArbolRojinegro`, `ArbolAVL` y `add()` de `TreeSet` con los mismos elementos.

Las estructuras dinámicas todas tienen un comportamiento similar, con nuestra implementación de MERGESORT en la clase `Lista` siendo un poco más rápida en promedio y los árboles AVL del libro siendo un poco más lentos; en particular los conjuntos ordenados de Java (clase `TreeSet`) son un poco más lentos que nuestro método `mergesort()`. Los métodos `sort()` de las clases `Arrays` y `Collections` se comportan básicamente de la misma manera; lo cual no nos extraña porque el segundo método lo que hace es copiar la colección a un arreglo, ordenarlo con el método `sort()` de la clase `Arrays` y copiar los elementos de regreso a la colección. Nuestro método `quicksort()` es muy similar, pero *ligeramente* más lento. Todo esto es, repetimos, sin ningún tipo de optimización especializada en el código del libro.

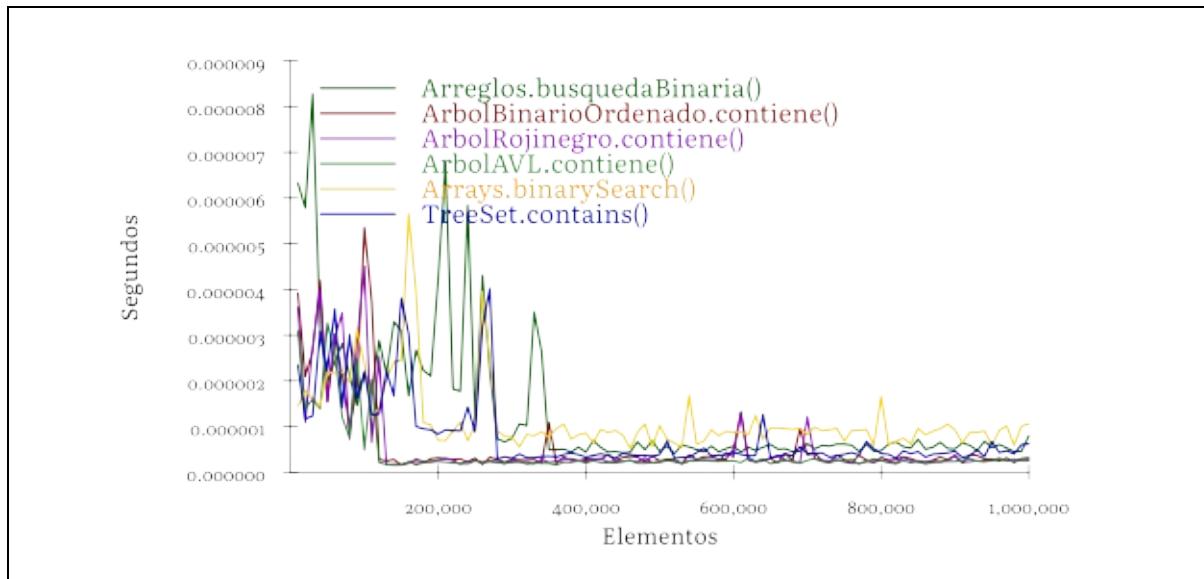


Figura 24.2: Tiempos para buscar hasta en 1,000,000 enteros aleatorios.

En la figura 24.2 se muestran los tiempos promedio de buscar 10 veces en colecciones de hasta 1,000,000 enteros aleatorios, usando los métodos `contains()` de `ArbolBinarioOrdenado`, `ArbolRojinegro` y `ArbolAVL`; y los métodos `busquedaBinaria()`, `binarySearch()` y `contains()` de las clases `Arreglos`, `Arrays` y `TreeSet` respectivamente. Realmente no hay mucha diferencia entre las distintas implementaciones (aunque nos sorprende que `binarySearch()` de `Arrays` sea el más lento para arreglos grandes); únicamente para arreglos “pequeños” hay un comportamiento notablemente distinto; pero lo que probablemente ocurra es que el cambio de contexto necesario para leer el reloj antes y después de realizar las búsquedas en arreglos pequeños sea mucho más tardado que las búsquedas mismas.

Sólo por completez mostramos en la figura 24.3 los tiempos promedio para llenar un diccionario o conjunto con 1,000,000 de enteros o pares de enteros. Sin ningún tipo de sorpresa, la clase `HashSet` se comporta básicamente igual a `HashMap`, de la misma manera que la clase `conjunto` se comporta básicamente igual a `Diccionario`. Las implementaciones del libro son, de nuevo, *ligeramente* más lentas que las de Java.

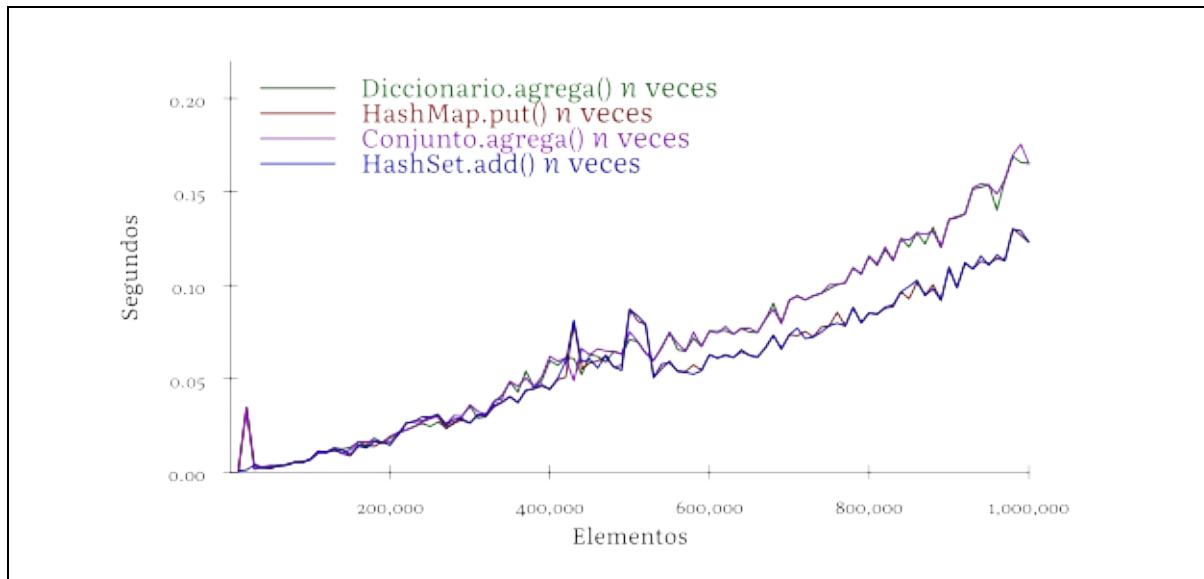


Figura 24.3: Tiempos para llenar hasta con hasta 1,000,000 enteros en tablas de dispersión y conjuntos.

La tabla correspondiente a consultas en diccionarios y conjuntos que usen tablas de dispersión la omitimos, porque los resultados toman un tiempo tan pequeño que son casi ilegibles y además probablemente se agregue un factor de error nada más por los cambios de contexto del procesador e incluso redondeo de los números de punto flotante de doble precisión.

Todos los experimentos se realizaron en una computadora Intel® Core™ i7-4790 a 3.60GHz y antes de ser realizados se ejecutó la operación a probar con una colección de elementos más grande o igual para descartar la intervención del compilador JIT (*Just-In-Time*) de Java.

Por último, pero definitivamente no menos importante, queremos resaltar que se cubrieron en el libro (aunque a veces de forma muy ligera) varias características de Java moderno como son genéricos, iteradores y lambdas; ésta última en particular agregada en la versión 8 del lenguaje, liberada en marzo de 2014. Todas las estructuras de datos cubiertas en el libro son genéricas, casi todas son iterables, la mayoría son colecciones y varias utilizan lambdas. Todas además cumplen en su diseño (tanto como nos pareció razonable) de los fundamentos de la Orientación Objetos; en particular pilas, colas y árboles binarios usan de manera fundamental la herencia; y todas las estructuras buscan no violar nunca el encapsulamiento de datos.

Es nuestra opinión que estas características no son sólo agregados interesantes del lenguaje; son fundamentales al impartir un curso moderno de Estructuras de Datos, porque las mismas (junto con las demás propiedades de la programación Orientada a Objetos) permiten diseñar e implementar las estructuras de datos clásicas del libro de Niklaus Wirth (y varias otras más modernas) de manera más elegante, concisa y reutilizable. Y preservando las

complejidades en tiempo y en espacio apropiadas.

Como mencionábamos en la introducción de este libro: se llaman estructuras de datos únicamente porque no se contaba con el concepto de objeto cuando fueron definidas. Son realmente objetos; y sus algoritmos asociados su comportamiento.

Bibliografía

- [1] Georgy Adelson-Velsky y Evgenii Landis: *An algorithm for the organization of information*. En *Proceedings of the USSR Academy of Sciences*, volumen 146, páginas 263—266, 1962 English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259—1263.
- [2] Paul Bachmann: *Die analytische Zahlentheorie*. Leipzig: B. G. Teubner, 1894.
- [3] Friedrich Ludwig Bauer y Klaus Samelson: *Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens*. En *Pioneers and Their Contributions to Software Engineering*, páginas 29—40. Springer, 2001.
- [4] Rudolf Bayer y Edward McCreight: *Organization and Maintenance of Large Ordered Indices*. En *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, páginas 107—141, 1970.
- [5] Rudolf Bayer: *Symmetric binary B-trees: Data structure and maintenance algorithms*. Acta informatica, 1(4):290—306, 1972.
- [6] Brian Edward Carpenter y Robert William Doran: *The other Turing machine*. The Computer Journal, 20(3):269—279, 1977.
- [7] Alonzo Church: *An unsolvable problem of elementary number theory*. American Journal of Mathematics, 58(2):345—363, 1936.
- [8] Danny Cohen: *On holy wars and a plea for peace*. Computer, 14(10):48—54, 1981.
- [9] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest y Clifford Stein: *Introduction to Algorithms, Third Edition*. MIT press Cambridge, 2009.
- [10] Edsger Wybe Dijkstra: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1(1):269—271, 1959.
- [11] Edsger Wybe Dijkstra: *Notes on structured programming*, 1970.
- [12] Alexander Shafto Douglas: *Techniques for the recording of, and reference to data in a computer*. The Computer Journal, 2(1):1—9, 1959.
- [13] Leonhard Euler: *Solutio problematis ad geometriam situs pertinensis*.

- Commentarii academiae scientiarum Petropolitanae, 8:128—140, 1736.
- [14] István Fáry: *On straight line representations of planar graphs*. Acta. Sci. Math. (Szeged), 11:229—233, 1948.
- [15] Michael Lawrence Fredman y Robert Endre Tarjan: *Fibonacci heaps and their uses in improved network optimization algorithms*. Journal of the ACM (JACM), 34(3):596—615, 1987.
- [16] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides: *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [17] Sally Goldman y Kenneth Goldman: *A practical guide to data structures and algorithms using Java*. CRC Press, 2007.
- [18] Radu Grigore: *Java Generics are Turing Complete*. CoRR, abs/1605.05274:1—13, 2016.
- [19] Leonidas John Guibas y Robert Sedgewick: *A dichromatic framework for balanced trees*. En *Foundations of Computer Science, 1978., 19th Annual Symposium on*, páginas 8—21, 1978.
- [20] Charles Antony Richard Hoare: *Quicksort*. The Computer Journal, 5(1):10—16, 1962.
- [21] Robert John Jenkins Junior: *Hash functions*. Dr Dobbs Journal, 22(9):107—+, 1997.
- [22] Donald Ervin Knuth: *The Art of Computer Programming: Sorting and Searching*. Pearson Education, 1998.
- [23] Edmund Landau: *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig: B. G. Teubner, 1909.
- [24] Barbara Huberman Liskov y Stephen Zilles: *Programming with abstract data types*. En *ACM Sigplan Notices*, volumen 9(4), páginas 50—59, 1974.
- [25] Barbara Huberman Liskov y Jeannette Marie Wing: *A behavioral notion of subtyping*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811—1841, 1994.
- [26] Witold Litwin: *Linear hashing: A new tool for file and table addressing*. En *VLDB*, volumen 80, páginas 1—3, 1980.
- [27] Dinesh Mehta: *Handbook of data structures and applications*. Chapman & Hall/CRC, 2005.
- [28] Allen Newell y John Clifford Shaw: *Programming the Logic Theory*

Machine. En *Western Joint Computer Conference: Techniques for Reliability*, páginas 230—240, 1957.

- [29] Phillip Rogaway y Thomas Shrimpton: *Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance*. En *International Workshop on Fast Software Encryption*, páginas 371—388, 2004.
- [30] Alan Mathison Turing: *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2(1):230—265, 1937.
- [31] John von Neumann y Abraham Haskel Taub: *John von Neumann Collected Works: Volume V-Design of Computers, Theory of Automata and Numerical Analysis*. Pergamon Press, 1963.
- [32] John William Joseph Williams: *Algorithm 232 — Heapsort*, 1964.
- [33] Niklaus Wirth: *Algorithms + Data Structures = Programs*. Prentice Hall PTR, 1978.
- [34] [GNU C Library: tsearch](#).
- [35] [GLib g_str_hash\(\) function](#).
- [36] [Hash Functions](#).
 - [37] [New Features and Enhancements J2SE 5.0](#).
 - [38] [Java Standard Library: Tree Maps](#).
 - [39] [Linux kernel repository: Red Black Trees](#).
 - [40] [Vala libgee: Tree Set](#).

Índice alfabético

Iterator

hasNext() [1](#), [2](#)

next() [1](#), [2](#), [3](#)

Adaptador

[1](#), [2](#), [3](#)

compareTo() [1](#)

getIndice() [1](#)

setIndice() [1](#)

constructor [1](#)

agrega()

ArbolAVL [1](#), [2](#)

ArbolBinarioCompleto [1](#), [2](#)

ArbolBinarioOrdenado [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)

ArbolRojinegro [1](#), [2](#), [3](#)

Coleccion [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)

Conjunto [1](#)

Diccionario [1](#), [2](#)

Grafica [1](#)

Lista [1](#), [2](#), [3](#), [4](#)

MonticuloMinimo [1](#)

agregaFinal()

Lista [1](#), [2](#), [3](#), [4](#), [5](#)

agregaInicio()

Lista [1](#), [2](#), [3](#)

altura()

ArbolAVL.VerticeAVL [1](#)

ArbolBinario.Vertice [1](#), [2](#), [3](#)

ArbolBinarioCompleto [1](#)

ArbolBinario [1](#), [2](#), [3](#), [4](#)

ArbolAVL [1](#)

agrega() [1](#), [2](#)

elimina() [1](#)

nuevoVertice() [1](#)

constructor [1](#), [2](#)

ArbolAVL.VerticeAVL

altura() [1](#)

equals() [1](#)

toString() [1](#)

ArbolBinario [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#)

altura() [1](#), [2](#), [3](#), [4](#)

busca() [1](#), [2](#), [3](#), [4](#), [5](#)

contiene() [1](#)

derecho() [1](#)

equals() [1](#)

esVacia() [1](#)

getElementos() [1](#), [2](#)

izquierdo() [1](#)

limpia() [1](#)

nuevoVertice() [1](#), [2](#), [3](#), [4](#)

padre() [1](#)

raiz() [1](#), [2](#)

toString() [1](#)

vertice() [1](#), [2](#)

constructor [1](#)

ArbolBinario.Vertice

altura() [1](#), [2](#), [3](#)

equals() [1](#), [2](#), [3](#), [4](#)

profundidad() [1](#), [2](#)

toString() [1](#), [2](#), [3](#), [4](#), [5](#)

ArbolBinarioCompleto [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

agrega() [1](#), [2](#)

altura() [1](#)

bfs() [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

elimina() [1](#)

iterator() [1](#)

constructor [1](#)

ArbolBinarioCompleto.Iterador [1](#), [2](#)

hasNext() [1](#)

next() [1](#)

constructor [1](#)

ArbolBinarioOrdenado [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)

agrega() [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)

busca() [1](#), [2](#)

dfsInOrder() [1](#), [2](#), [3](#), [4](#)

dfsPostOrder() [1](#), [2](#)

dfsPreOrder() [1](#), [2](#)

elimina() [1](#), [2](#), [3](#), [4](#)

eliminaVertice() [1](#), [2](#)

getUltimoVerticeAgregado() [1](#), [2](#)

giraDerecha() [1](#), [2](#), [3](#), [4](#)

giraIzquierda() [1](#), [2](#), [3](#), [4](#)

intercambiaEliminable() [1](#), [2](#)

iterator() [1](#)

constructor [1](#)

ArbolBinarioOrdenado.Iterador [1](#), [2](#)

hasNext() [1](#)

next() [1](#), [2](#)

constructor [1](#)

ArbolRojinegro [1](#), [2](#), [3](#)

agrega() [1](#), [2](#), [3](#)

elimina() [1](#), [2](#)

getColor() [1](#), [2](#)

nuevoVertice() [1](#)

constructor [1](#)

ArbolRojinegro.VerticeRojinegro

equals() [1](#)

toString() [1](#)

constructor [1](#)

árboles AVL [1](#), [2](#)

definición [1](#)

historia [1](#)

árboles binarios [1](#), [2](#)

altura [1](#)

balanceados [1](#)

definición [1](#)

llenos [1](#)

niveles [1](#)

niveles, llenos [1](#)

subárboles [1](#)

subárboles, izquierdo y derecho [1](#)

vértices, altura [1](#)

vértices, balance [1](#)

vértices, coordenadas [1](#)

vértices, definición [1](#)

vértices, dirección [1](#)

vértices, hojas e internos [1](#)

vértices, profundidad [1](#)

árboles binarios completos [1](#), [2](#)

definición [1](#)

árboles binarios ordenados [1](#), [2](#)

definición [1](#)

historia [1](#)

árboles rojinegros [1](#), [2](#)

definición [1](#)

historia [1](#)

Arreglos [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

quickSort() [1](#), [2](#), [3](#)

selectionSort() [1](#)

bfs()

ArbolBinarioCompleto [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

Grafica [1](#), [2](#)

big-endian [1](#)

historia [1](#)

busca()

ArbolBinarioOrdenado [1](#), [2](#)

ArbolBinario [1](#), [2](#), [3](#), [4](#), [5](#)

busquedaBinaria()

Arreglos [1](#)

cálculo- λ [1](#), [2](#)

historia [1](#)

carga()

Diccionario [1](#)

cola [1](#), [2](#)

mete() [1](#)
toString() [1](#)
colas [1](#), [2](#), [3](#)
colección [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#)
agrega() [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
contiene() [1](#), [2](#)
elimina() [1](#), [2](#)
esVacia() [1](#)
getElementos() [1](#), [2](#)
iterator() [1](#), [2](#)
limpia() [1](#)
colisionMaxima()

Diccionario [1](#)
colisiones()

Diccionario [1](#)

ComparaEstudiantesPorNúmeroDeCuenta [1](#)

Comparable

compareTo() [1](#), [2](#), [3](#), [4](#), [5](#)

ComparableIndexable

getIndice() [1](#)

setIndice() [1](#), [2](#)

Comparator

compare() [1](#), [2](#)
compare()

Comparador [1](#), [2](#)

compareTo()

Grafica.Vertice [1](#), [2](#), [3](#)

Adaptador [1](#)

Comparable [1](#), [2](#), [3](#), [4](#), [5](#)

complejidad en espacio [1](#), [2](#)

complejidad en tiempo [1](#), [2](#)

conecta()

Grafica [1](#), [2](#), [3](#), [4](#)

Conjunto [1](#)

agrega() [1](#)

contiene() [1](#)

elimina() [1](#)

equals() [1](#)

esVacia() [1](#)

getElementos() [1](#)

interseccion() [1](#), [2](#), [3](#)

limpia() [1](#)

toString() [1](#)

union() [1](#), [2](#), [3](#)

constructor [1](#)

conjuntos [1](#), [2](#)

constructor

ArbolAVL.VerticeAVL [1](#)

ArbolBinarioCompleto.Iterador [1](#)
ArbolBinarioOrdenado.Iterador [1](#)
ArbolRojinegro.VerticeRojinegro [1](#)
Diccionario.Iterador [1](#)
Grafica.Iterador [1](#)
Grafica.Vecino [1](#)
Grafica.Vertice [1](#)
Lista.Iterador [1](#)
Adaptador [1](#)
ArbolAVL [1](#)
ArbolBinarioCompleto [1](#)
ArbolBinarioOrdenado [1](#)
ArbolBinario [1](#)
ArbolRojinegro [1](#)
Conjunto [1](#)
Diccionario [1](#)
Grafica [1](#)
MonticuloArreglo [1](#)
MonticuloMinimo [1](#)
contiene()

ArbolAVL [1](#)
ArbolBinarioOrdenado [1](#)
ArbolBinario [1](#)
ArbolRojinegro [1](#)

Colección [1](#), [2](#)

Conjunto [1](#)

Diccionario [1](#), [2](#)

Grafica [1](#), [2](#)

Lista [1](#), [2](#)

MontículoMínimo [1](#)

copia()

Lista [1](#), [2](#), [3](#), [4](#)

derecho()

ÁrbolBinario [1](#)

desconecta()

Grafica [1](#)

dfs()

Grafica [1](#)

DFS PRE ORDER [1](#)

DFS PRE ORDER [1](#)

DFS PRE ORDER [1](#)

dfsInOrder()

ÁrbolBinarioOrdenado [1](#), [2](#), [3](#), [4](#)

dfsPostOrder()

ÁrbolBinarioOrdenado [1](#), [2](#)

dfsPreOrder()

ÁrbolBinarioOrdenado [1](#), [2](#)

Diccionario [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

agrega() [1](#), [2](#)
carga() [1](#)
colisionMaxima() [1](#)
colisiones() [1](#)
contiene() [1](#), [2](#)
elimina() [1](#), [2](#)
equals() [1](#)
esVacia() [1](#)
get() [1](#), [2](#)
getElementos() [1](#)
iteradorLlaves() [1](#)
iterator() [1](#)
limpia() [1](#)
nuevoArreglo() [1](#)
toString() [1](#)
constructor [1](#)
Diccionario.Iterador [1](#)
hasNext() [1](#)
next() [1](#)
siguiente() [1](#), [2](#)
constructor [1](#)
Diccionario.IteradorLlaves [1](#)
Diccionario.IteradorValores [1](#)
diccionarios [1](#), [2](#), [3](#)

definición [1](#)

historia [1](#)

Dijkstra

algoritmo de [1](#), [2](#)

historia del algoritmo de [1](#)

dispersorCadena()

FabricaDispersores [1](#), [2](#)

Dispersores [1](#), [2](#)

agrega()

Diccionario [1](#)

eliminaPrimero()

Lista [1](#), [2](#)

eliminaUltimo()

Lista [1](#), [2](#)

eliminaVertice()

ArbolBinarioOrdenado [1](#), [2](#)

end()

Lista.Iterador [1](#)

IteradorLista [1](#)

equals()

ArbolAVL.VerticeAVL [1](#)

ArbolBinario.Vertice [1](#), [2](#), [3](#), [4](#)

ArbolRojinegro.VerticeRojinegro [1](#)

ArbolBinario [1](#)

Conjunto [1](#)

Diccionario [1](#)

Grafica [1](#), [2](#)

Lista [1](#)

MeteSaca [1](#), [2](#)

MonticuloMinimo [1](#)

Object [1](#), [2](#), [3](#), [4](#), [5](#)

`esConexa()`

Grafica [1](#)

`esVacia()`

ArbolBinario [1](#)

Coleccion [1](#)

Conjunto [1](#)

Diccionario [1](#)

Grafica [1](#)

Lista [1](#)

MeteSaca [1](#)

MonticuloArreglo [1](#)

MonticuloMinimo [1](#), [2](#)

FabricaDispersores [1](#), [2](#)

`dispensorCadena()` [1](#), [2](#)

funciones de dispersión [1](#)

colisiones [1](#)

definición [1](#)

genéricos [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#)

arreglos y [1](#)

get()

Grafica.**Vecino** [1](#), [2](#)

Grafica.**Vertice** [1](#)

Diccionario [1](#), [2](#)

Lista [1](#), [2](#), [3](#), [4](#)

MonticuloArreglo [1](#)

MonticuloMinimo [1](#)

getAristas()

Grafica [1](#)

getColor()

Grafica.**Vecino** [1](#)

Grafica.**Vertice** [1](#)

ArbolRojoNegro [1](#), [2](#)

getElementos()

ArbolBinario [1](#), [2](#)

Coleccion [1](#), [2](#)

Conjunto [1](#)

Diccionario [1](#)

Grafica [1](#)

Lista [1](#), [2](#), [3](#), [4](#), [5](#)

MonticuloArreglo [1](#)

MonticuloMinimo [1](#), [2](#)

getGrado()

Grafica.**Vecino** [1](#)

Grafica.**Vertice** [1](#)

getIndice()

Adaptador [1](#)

ComparableIndexable [1](#)

getLongitud()

Lista [1](#), [2](#), [3](#), [4](#), [5](#)

getPeso()

Grafica [1](#), [2](#)

getPrimero()

Lista [1](#), [2](#), [3](#)

getUltimo()

Lista [1](#), [2](#)

getUltimoVerticeAgregado()

ArbolBinarioOrdenado [1](#), [2](#)

giraDerecha()

ArbolBinarioOrdenado [1](#), [2](#), [3](#), [4](#)

giraIzquierda()

ArbolBinarioOrdenado [1](#), [2](#), [3](#), [4](#)

Grafica [1](#), [2](#), [3](#), [4](#), [5](#)

agrega() [1](#)

bfs() [1](#), [2](#)

conecta() [1](#), [2](#), [3](#), [4](#)

contiene() [1](#), [2](#)
desconecta() [1](#)
dfs() [1](#), [2](#)
dijkstra() [1](#), [2](#), [3](#), [4](#)
elimina() [1](#)
equals() [1](#), [2](#)
esConexa() [1](#)
esVacia() [1](#)
getAristas() [1](#)
getElementos() [1](#)
getPeso() [1](#), [2](#)
limpia() [1](#)
paraCadaVertice() [1](#)
setColor() [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)
sonVecinos() [1](#), [2](#), [3](#)
toString() [1](#)
trayectoriaMinima() [1](#), [2](#)
vertice() [1](#)
constructor [1](#)
Grafica.Iterador [1](#), [2](#), [3](#)
hasNext() [1](#)
next() [1](#)
constructor [1](#)
Grafica.Vecino [1](#)

get() [1](#), [2](#)

getColor() [1](#)

getGrado() [1](#)

vecinos() [1](#)

constructor [1](#)

Grafica.Vertice

compareTo() [1](#), [2](#), [3](#)

get() [1](#)

getColor() [1](#)

getGrado() [1](#)

vecinos() [1](#), [2](#)

constructor [1](#)

gráficas [1](#), [2](#), [3](#)

caminos [1](#)

componentes conexas [1](#)

conexidad [1](#)

definición [1](#)

historia [1](#)

planaridad [1](#)

ponderadas [1](#)

subgráficas [1](#)

trayectorias [1](#)

trayectorias, longitud [1](#)

trayectorias, pesos [1](#)

vértices, grado [1](#)

hasNext()

ArbolBinarioCompleto.Iterador [1](#)

ArbolBinarioOrdenado.Iterador [1](#)

Diccionario.Iterador [1](#)

Grafica.Iterador [1](#)

MonticuloMinimo.Iterador [1](#)

IteradorLista [1](#)

Iterador [1](#), [2](#), [3](#), [4](#)

Iterator [1](#), [2](#)

hasPrevious()

Lista.Iterador [1](#)

IteradorLista [1](#)

HEAPSORT [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)

historia [1](#)

heapSort()

MonticuloMinimo [1](#), [2](#)

historia

Orientación a Objetos [1](#)

big-endian [1](#)

little-endian [1](#)

HEAPSORT [1](#)

MERGESORT [1](#)

QUICKSORT [1](#)

algoritmo de Dijkstra [1](#), [2](#), [3](#)

búsqueda binaria [1](#)

cálculo- λ [1](#)

diccionarios [1](#)

función de dispersión Bob Jenkins [1](#)

función de dispersión de Daniel J. Bernstein [1](#)

gráficas [1](#)

lenguajes de programación [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

listas [1](#)

montículos mínimos [1](#)

pilas [1](#)

árboles AVL [1](#)

árboles binarios ordenados [1](#)

árboles rojinegros [1](#)

indiceDe()

Lista [1](#), [2](#)

inserta()

Lista [1](#)

intercambiaEliminable()

ArbolBinarioOrdenado [1](#), [2](#)

interseccion()

Conjunto [1](#), [2](#), [3](#)

Iterador

hasNext() [1](#), [2](#), [3](#), [4](#)

next() [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)

IteradorLista

end() [1](#)

hasPrevious() [1](#)

previous() [1](#), [2](#)

start() [1](#)

iteradorLista()

Lista [1](#)

iteradorLlaves()

Diccionario [1](#)

iteradores [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#), [42](#)
iterador() [1](#)

ArbolBinarioCompleto [1](#)

ArbolBinarioOrdenado [1](#)

Coleccion [1](#), [2](#)

Diccionario [1](#)

Lista [1](#), [2](#), [3](#), [4](#), [5](#)

iteratorLista()

Lista [1](#), [2](#)

izquierdo()

ArbolBinario [1](#)

lambdas [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)

limpia()

ArbolBinario [1](#)

Colección [1](#)

Conjunto [1](#)

Diccionario [1](#)

Grafica [1](#)

Lista [1](#)

MontículoMínimo [1](#)

Liskov

Barbara [1](#)

principio de sustitución de [1](#), [2](#), [3](#)

Lista [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#)

agrega() [1](#), [2](#), [3](#), [4](#)

agregaFinal() [1](#), [2](#), [3](#), [4](#), [5](#)

agregaInicio() [1](#), [2](#), [3](#)

contiene() [1](#), [2](#), [3](#)

copia() [1](#), [2](#), [3](#), [4](#)

elimina() [1](#), [2](#), [3](#), [4](#)

eliminaPrimero() [1](#), [2](#), [3](#)

eliminaUltimo() [1](#)

equals() [1](#)

esVacia() [1](#)

get() [1](#), [2](#), [3](#), [4](#)

getElementos() [1](#), [2](#), [3](#), [4](#), [5](#)

getLongitud() [1](#), [2](#), [3](#), [4](#), [5](#)

getPrimero() [1](#), [2](#), [3](#)

getUltimo() [1](#), [2](#)
indiceDe() [1](#), [2](#)
inserta() [1](#)
iterator() [1](#), [2](#), [3](#), [4](#), [5](#)
iteratorLista() [1](#), [2](#), [3](#)
limpia() [1](#)
mergeSort() [1](#), [2](#), [3](#), [4](#)
reversa() [1](#), [2](#), [3](#), [4](#)
toString() [1](#)
Lista.Iterador [1](#), [2](#), [3](#), [4](#), [5](#)
end() [1](#)
hasNext() [1](#)
hasPrevious() [1](#)
next() [1](#), [2](#), [3](#), [4](#)
previous() [1](#), [2](#), [3](#)
start() [1](#), [2](#)
constructor [1](#)
listas [1](#), [2](#), [3](#), [4](#)
definición [1](#)
historia [1](#)
nodos [1](#)
little-endian [1](#)
historia [1](#)
MERGESORT [1](#), [2](#)

historia [1](#)

mergeSort()

Lista [1](#), [2](#), [3](#), [4](#), [5](#)

mete()

Cola [1](#)

MeteSaca [1](#)

Pila [1](#), [2](#)

MeteSaca [1](#), [2](#), [3](#)

equals() [1](#), [2](#)

esVacia() [1](#)

mete() [1](#)

mira() [1](#)

saca() [1](#)

toString() [1](#)

mira()

MeteSaca [1](#)

MonticuloArreglo [1](#)

elimina() [1](#)

esVacia() [1](#)

get() [1](#)

getElementos() [1](#)

constructor [1](#)

MonticuloDijkstra

reordena() [1](#)

MonticuloMinimo [1](#)

agrega() [1](#)

contiene() [1](#)

elimina() [1](#), [2](#)

equals() [1](#)

esVacia() [1](#), [2](#)

get() [1](#)

getElementos() [1](#), [2](#)

heapSort() [1](#)

limpia() [1](#)

nuevoArreglo() [1](#), [2](#)

reordena() [1](#), [2](#)

toString() [1](#)

constructor [1](#)

MonticuloMinimo.Iterador [1](#)

hasNext() [1](#)

next() [1](#)

montículos mínimos [1](#), [2](#)

definición [1](#)

historia [1](#)

hasNext()

ArbolBinarioCompleto.Iterador [1](#)

nuevoArreglo()

Diccionario [1](#)

Monticulominimo [1](#), [2](#)

nuevoVertice()

ArbolAVL [1](#)

ArbolBinario [1](#), [2](#), [3](#), [4](#)

ArbolRojinegro [1](#)

O grandota [1](#), [2](#)

definición [1](#)

Object

equals() [1](#), [2](#), [3](#), [4](#), [5](#)

toString() [1](#), [2](#)

Orientación a Objetos [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#)

padre()

ArbolBinario [1](#)

paraCadaVertice()

Grafica [1](#)

Pila [1](#), [2](#)

mete() [1](#), [2](#)

saca() [1](#)

toString() [1](#), [2](#)

pilas [1](#), [2](#), [3](#)

historia [1](#)

previous()

Lista.Iterador [1](#), [2](#), [3](#)

IteradorLista [1](#), [2](#)

problema del paro [1](#), [2](#)

profundidad()

ArbolBinario.Vertice [1](#), [2](#)

QUICKSORT [1](#)

historia [1](#)

quickSort()

Arreglos [1](#), [2](#), [3](#), [4](#)

raiz()

ArbolBinario [1](#), [2](#)

rebalanceo

árboles AVL [1](#)

árboles rojinegros al agregar [1](#)

árboles rojinegros al eliminar [1](#)

recursión [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#)
reordenar()

MonticuloDijkstra [1](#)

MonticuloMinimo [1](#), [2](#)

reversa()

Lista [1](#), [2](#), [3](#), [4](#)

saca()

MeteSaca [1](#)

Pila [1](#)

SELECTIONSORT [1](#)

selectionSort()

Arreglos [1](#)

setColor()

Grafica [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

setIndice()

Adaptador [1](#)

ComparableIndexable [1](#), [2](#)

siguiente()

Diccionario.Iterador [1](#), [2](#)

sonVecinos()

Grafica [1](#), [2](#), [3](#)

start()

Lista.Iterador [1](#), [2](#)

IteradorLista [1](#)

toString()

ArbolAVL.VerticeAVL [1](#)

ArbolBinario.Vertice [1](#), [2](#), [3](#), [4](#), [5](#)

ArbolRojinegro.VerticeRojinegro [1](#)

ArbolBinario [1](#)

Cola [1](#)

Conjunto [1](#)

Diccionario [1](#)

Grafica [1](#)

Lista [1](#)

MeteSaca [1](#)

Monticulominimo [1](#)

Object [1](#), [2](#)

Pila [1](#), [2](#)

trayectoriaminima()

Grafica [1](#), [2](#)

union()

Conjunto [1](#), [2](#), [3](#)

vecinos()

Grafica.Vecino [1](#)

Grafica.Vertice [1](#), [2](#)

vertice()

ArbolBinario [1](#), [2](#)

Grafica [1](#)