# Piensa en Estructuras de Datos

## Algoritmos y Recuperación de Información en Java

Versión 1.0.1

# Piensa en Estructuras de Datos

## Algoritmos y Recuperación de Información en Java

Versión 1.0.1

Allen B. Downey

Traducido por: Ernesto A. Aguilar

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham, MA 02492



El formato original de este libro es código fuente LaTeX . La compilación de este código tiene el efecto de generar una representación del libro independiente del dispositivo, que puede convertirse a otros formatos e imprimirse.

El fuente LaTeX de este libro está disponible en http://thinkdast.com/repo.

# Contenidos

# Prefacio

## La filosofía tras este libro

Las estructuras de datos y los algoritmos están entre las invenciones más importantes de los últimos 50 años, y son herramientas fundamentales que los ingenieros de software necesitan conocer. Pero en mi opinión, la mayoría de los libros sobre el tema son muy teóricos, muy grandes y muy "detallados":

**Muy teóricos** El análisis matemático de los algoritmos se base en simplificar supuestos, lo que limita su utilidad en la práctica. La mayoría de exposiciones sobre el tema apenas cubren las simplificaciones y se enfocan en la matemática. En este libro presento el subconjunto más práctico de este material y omito o enfatizo menos el resto.

**Muy grandes** La mayoría de libros sobre el tema tienen al menos 500 páginas, y algunos más de 1000. Al enfocarme en los temas que considero más útiles para los ingenieros de software, mantuve este libro por debajo de las 200 páginas.

**Muy "detallados"** Muchos libros de estructuras de datos se enfocan más en cómo funcionan las estructuras de datos (las implementaciones) y menos en cómo usarlas (las interfaces). En este libro, proveo una "visión general", al iniciar con las interfaces. Los lectores aprenderán a usar las estructuras del Java Collections Framework antes de adentrarse en los detalles de cómo funcionan.

Finalmente, algunos libros presentan este material fuera de contexto y sin motivación alguna: ¡es sólo una estructura de datos tras otra! Trato de volverlo más ameno al organizar los temas alrededor de una aplicación – la búsqueda

web – que usa estructuras de datos ampliamente, y es una tema interesante e importante por sus propios méritos.

Esta aplicación motiva algunos temas que generalmente no se cubren en una curso introductorio de estructuras de datos, incluyendo estructuras de datos persistentes con Redis.

He tomado algunas decisiones difíciles con relación a qué dejar fuera, pero he hecho algunas concesiones. Incluyo algunos temas que la mayoría de los lectores nunca usarán, pero que podría esperarse que conozcan, posiblemente en una entrevista técnica. Para estos temas, presento tanto la sabiduría convencional como mis razones para ser escéptico al respecto.

Este libro también presenta aspectos básicos de la ingeniería de software en la práctica, incluyendo control de versiones y pruebas unitarias. La mayoría de los capítulos incluyen un ejercicio que permite a los lectores aplicar lo que han aprendido. Cada ejercicio provee pruebas automáticas para comprobar la solución. Y para la mayoría de ejercicios, presento mi solución al principio del siguiente capítulo.

## 0.1 Prerrequisitos

Este libro está dirigido a estudiantes de ciencias de la computación y ramas afines, así como a ingenieros de software profesionales, personas capacitándose en ingeniería de software y personas personas preparándose para entrevistas técnicas.

Antes de comenzar este libro, deberías conocer Java bastante bien; en particular, deberías saber como definir una nueva clase que extienda una clase existente o implemente una `interface`. Si tu Java está oxidado, aquí están dos libros con los que podrías empezar:

- Downey and Mayfield, *Think Java* (O'Reilly Media, 2016, en inglés), pensado para personas que nunca antes han programado.
- Sierra and Bates, *Head First Java* (O'Reilly Media, 2005, en inglés), apropiado para personas que ya conocen otro lenguaje de programación.

Si no estás familiarizado con las interfaces en Java, podrías revisar el tutorial llamado "What Is an Interface?" en http://thinkdast.com/interface.

Una nota de vocabulario: la palabra "interfaz" (o "interface", en inglés) puede resultar confusa. En el contexto de una **interfaz de programación de aplicaciones** (API, de **application programming interface**), se refiere a un conjunto de clases y métodos que proveen ciertas funcionalidades.

En el caso de Java, también se refiere a una característica del lenguaje, similar a una clase, que especifica un conjunto de métodos. Para evitar confusiones, usaré "interfaz" en un tipo de fuente normal y en español, para la idea general de una interfaz, e `interface` en una fuente monoespaciada y en inglés, para la característica del lenguaje Java.

También deberías estar familiarizado con los parámetros de tipo y los tipos genéricos. Por ejemplo, deberías saber cómo crear un objeto con un parámetro de tipo, como `ArrayList<Integer>`. Si no, puedes leer sobre los parámetros de tipo en http://thinkdast.com/types.

Deberías estar familiarizado con el Java Collections Framework (JCF), sobre el que puedes leer en http://thinkdast.com/collections. En particular, deberías conocer la `interface List` y las clases `ArrayList` y `LinkedList`.

Idealmente deberías estar familiarizado con Apache Ant, una herramienta automatizada de de construcción para Java. Puedes leer más sobre Ant en http://thinkdast.com/anttut.

Y deberías estar familiarizado con `JUnit`, un *framework* para Java. Puedes leer más sobre él en http://thinkdast.com/junit.

## Trabajando con el código

El código para este libro está en un repositorio de Git en http://thinkdast.com/repo.

Git es un "sistema de control de versiones" que te permite dar seguimiento a los archivos que conforman un proyecto. Una colección de archivos bajo el control de Git es llamada un "repositorio".

GitHub es un servicio de alojamiento que provee almacenamiento para repositorios de Git y una interfaz web conveniente. Provee varias formas de trabajar con el código:

- Puedes crear una copia del repositorio en GitHub presionando el botón Fork. Si no tienes una cuenta en GitHub, necesitarás crear una. Tras crear el fork, tendrás tu propio repositorio en GitHub que puedes usar para dar seguimiento al código que escribes. Luego puedes "clonar" ("clone", en inglés) el repositorio, para descargar una copia de los archivos a tu computadora.

- Alternativamente, puedes clonar el repositorio sin crear un fork. Si eliges esta opción, no necesitas una cuenta de GitHub, pero no podrás guardar tus cambios en GitHub.

- Si no quieres usar Git en lo absoluto, puedes descargar el código en un archivo ZIP usando el botón Download en la página de GitHub, o este enlace: http://thinkdast.com/zip.

Después de clonar el repositorio o descomprimir el archivo ZIP, tendrás un directorio llamado `ThinkDataStructures`, que es el título del libro en inglés, con un subdirectorio `code`.

Los ejemplos en este libro fueron desarrollados y probados usando el Java SE Development Kit 7. Si estás usando una versión más antigua, algunos ejemplos no funcionarán. Si estás usando una versión más reciente, todos deberían funcionar.

## Colaboradores

Este libro es una adaptación de la currícula que escribí para la Flatiron School en New York City, que ofrece varias clases en línea relacionadas con programación y desarrollo web. Ellos ofrecen una clase basada en este material, que provee un entorno de desarrollo en línea, ayuda de instructores y otros estudiantes, así como un certificado de finalización. Puedes encontrar más información en http://flatironschool.com.

- En la Flatiron School, Joe Burgess, Ann John y Charles Pletcher brindaron orientación, sugerencias y correcciones desde la especificación inicial hasta la implementación y las pruebas. ¡Muchas gracias a todos ustedes!

- Estoy muy agradecido con mis revisores técnicos, Barry Whitman, Patrick White y Chris Mayfield, que dieron muchas sugerencias útiles y detectaron muchos errores. Por supuesto, ¡cualquier error restante es mi responsabilidad, no la de ellos!

- Gracias a los instructores y estudiantes de Estructuras de Datos y Algoritmos en Olin College, que leyeron este libro y brindaron realimentación útil.

- Charles Roumeliotis editó y corrigió el libro para O'Reilly Media y realizó muchas mejoras.

Si tienes comentarios o ideas sobre el texto, por favor envíalas a: `feedback@greenteapress.com`.

# Capítulo 1

# Interfaces

Este libro presenta tres temas:

- Estructuras de datos: Iniciando con las estructuras en el Java Collections Framework (JCF), aprenderás cómo usar estructuras de datos como listas y mapas y verás cómo trabajan.
- Análisis de algoritmos: Presento técnicas para analizar código y predecir qué tan rápido se ejecutará y cuánto espacio (memoria) requerirá.
- Recuperación de información: Para motivar el estudio de los primeros dos temas y hacer los ejercicios más interesantes, usaremos estructuras de datos y algoritmos para construir un motor simple de búsqueda en la web.

Aquí está una descripción general del orden de los temas:

- Iniciaremos con la interfaz `List` y escribirás clases que implementarán esta interfaz de dos formas diferentes. Luego, vamos a comparar tus implementaciones con las clases de Java `ArrayList` y `LinkedList`.
- A continuación, introduciré tres estructuras de datos en forma de árbol, y trabajarás en la primera aplicación: un programa que lea páginas de Wikipedia, interprete sus contenidos y navegue por el árbol resultante para encontrar enlaces y otras características. Usaremos estas herramientas para comprobar la conjetura "Llegar a la Filosofía" (puedes anticiparte leyendo http://thinkdast.com/getphil).

- Aprenderemos sobre la interfaz `Map` y la implementación `HashMap` de Java. Luego escribiremos clases que implementen esta interfaz usando una tabla hash y un árbol binario de búsqueda.
- Finalmente, usarás estas clases (y unas cuantas más que presentaré en el camino) para implementar un motor de búsqueda en la web, incluyendo: un *rastreador* que encuentra y lee páginas, un indexador que guarda los contenidos de las páginas Web de forma que puedan realizarse búsquedas en ellas eficientemente, y un recuperador que toma las consultas de un usuario y devuelve resultados relevantes.

Comencemos.

## 1.1 ¿Por qué hay dos tipos de `List`?

Cuando las personas comienzan a trabajar con el Java Collections Framework, a veces se confunden entre `ArrayList` y `LinkedList`. ¿Por qué Java provee dos implementaciones de la `interface List`? ¿Y cómo deberías elegir cuál usar? Responderemos estas preguntas en los siguientes capítulos.

Comenzaré por revisar las `interface`s y las clases que las implementan y presentará la idea de "programar para una interfaz".

En los primeros ejercicios a continuación, implementarás clases similares a `ArrayList` y `LinkedList`, para entender cómo funcionan y veremos que cada una de ellas tiene pros y contras. Algunas operaciones son más rápidas o usan menos espacio con una `ArrayList`; otras son más rápidas o más pequeñas con una `LinkedList`. Cuál de ellas es mejor para una aplicación particular depende de las operaciones que se realizan con más frecuencia.

## 1.2 Interfaces en Java

Una `interface` de Java especifica un conjunto de métodos; cualquier clase que implemente esta `interface` tiene que proveer estos métodos. Por ejemplo, aquí está el código fuente para `Comparable`, que es una `interface` definida en el paquete `java.lang`:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Esta definición de `interface` usa un parámetro de tipo, `T`, lo que hace a `Comparable` un **tipo genérico**. Para implementar esta `interface`, una clase tiene que

- Especificar el tipo al que `T` se refiere y
- Proveer un método llamado `compareTo` que tome un objeto como parámetro y devuelva un `int`.

Por ejemplo, aquí está el código fuente para `java.lang.Integer`:

```
public final class Integer extends Number implements Comparable<Integer> {

    public int compareTo(Integer anotherInteger) {
        int thisVal = this.value;
        int anotherVal = anotherInteger.value;
        return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
    }

    // otros métodos omitidos
}
```

Esta clase extiende `Number`, por lo que hereda los métodos y variables de instancia de `Number`; e implementa `Comparable<Integer>`, así que provee un método con el nombre `compareTo` que toma un `Integer` y devuelve un `int`.

Cuando una clase declara que implementa una `interface`, el compilador verifica que provea todos los métodos definidos por la `interface`.

Al margen, esta implementación de `compareTo` usa el "operador ternario", que a veces se escribe `?:`. Si no estás familiarizado con él, puedes leer al respecto en http://thinkdast.com/ternary.

## 1.3 La interfaz List

El Java Collections Framework (JCF) define una `interface` llamada `List` y provee dos implementaciones, `ArrayList` y `LinkedList`.

La `interface` define lo que implica ser una `List`; cualquier clase que implemente esta `interface` tiene que proveer un conjunto particular de métodos, que incluyen `add`, `get`, `remove` y alrededor de 20 más.

`ArrayList` y `LinkedList` proveen estos métodos, así que ambas pueden ser usadas de forma intercambiable. Un método escrito para funcionar con una `List` funcionará con una `ArrayList`, `LinkedList`, o cualquier otro objeto que implemente `List`.

Aquí está un ejemplo artificial para demostrar el punto:

```
public class ListClientExample {
    private List list;

    public ListClientExample() {
        list = new LinkedList();
    }

    private List getList() {
        return list;
    }

    public static void main(String[] args) {
        ListClientExample lce = new ListClientExample();
        List list = lce.getList();
        System.out.println(list);
    }
}
```

`ListClientExample` no hace nada útil, pero contiene los elementos esenciales de una clase que **encapsula** una `List`; es decir, contiene una `List` como una variable de instancia. Usaré esta clase para ilustrar un punto y luego trabajarás con ella en el primer ejercicio.

El constructor `ListClientExample` inicializa una `list` al **instanciar** (es decir, crear) una nueva `LinkedList`; el método getter `getList` devuelve una

referencia al objeto interno `List`; y `main` contiene unas pocas líneas de código para probar estos métodos.

Lo importante de este ejemplo es que usa `List` cuando es posible y evita especificar `LinkedList` o `ArrayList` a menos que sea necesario. Por ejemplo, la variable de instancia es declarada como una `List` y `getList` devuelve una `List`, pero ninguno especifica qué tipo de lista.

Si cambias de opinión y decides usar una `ArrayList`, solo tienes que cambiar el constructor; no tienes que hacer ninguna modificación adicional.

Este estilo es llamado **programación basada en interfaces**, o de forma más casual, "programar para una interfaz" (véase http://thinkdast.com/interbaseprog). De lo que hablamos es de la idea general de una interfaz, no de una `interface` de Java.

Cuando usas una biblioteca, tu código debería depender únicamente de la interfaz, como `List`. No debería depender de implementaciones específicas, como `ArrayList`. De esa forma, si la implementación cambia en el futuro, el código que la usa seguirá funcionando.

Por otro lado, si la interfaz cambia, el código que depende de ella también tiene que cambiar. Es por eso que los desarrolladores de bibliotecas evitan cambiar las interfaces a menos que sea absolutamente necesario.

## 1.4 Ejercicio 1

Dado que este es el primer ejercicio, lo mantendremos simple. Tomarás el código de la sección anterior e **intercambiarás la implementación**; es decir, reemplazarás la `LinkedList` con una `ArrayList`. Debido a que el código programa para una interfaz, serás capaz de intercambiar la implementación cambiando una sola línea de código y agregando una instrucción `import`.

Comienza configurando tu entorno de desarrollo. Para todos los ejercicios, necesitarás poder compilar y ejecutar código de Java. Desarrollé los ejemplo usando el Java SE Development Kit 7. Si estás usando una versión más reciente, todo debería funcionar. Si estás usando una versión más antigua, puede que encuentres algunas incompatibilidades.

Recomiendo usar un entorno de desarrollo integrado (IDE) que provea comprobación de sintaxis, auto-completado y caracterización de código fuente. Estas características te ayudan a prevenir errores o a encontrarlos rápidamente. Sin embargo, si estás preparándote para una entrevista técnica, recuerda que no tendrás a tu disposición estas herramientas durante la entrevista, por lo que podrías querer practicar escribiendo código sin ellas.

Si aun no has descargado el código para este libro, revisa las instrucciones en la Sección 0.1.

En el directorio llamado `code`, encontrarás estos archivos y directorios:

- `build.xml` es un archivo Ant que facilita compilar y ejecutar el código.
- `lib` contiene las bibliotecas que necesitarás (para este ejercicio, solo JUnit).
- `src` contiene el código fuente.

Si navegas a `src/com/allendowney/thinkdast`, encontrarás el código fuente para este ejercicio:

- `ListClientExample.java` contiene el código de la sección anterior.
- `ListClientExampleTest.java` contiene un test de JUnit para `ListClientExample`.

Revisa `ListClientExample` y asegúrate de entender lo que hace. Luego, compílalo y ejecútalo. Si usas Ant, puedes navegar al directorio `code` y ejecutar `ant ListClientExample`.

Puede ser que obtengas una advertencia como:

```
List is a raw type. References to generic type List<E>
should be parameterized.
```

Para mantener este ejemplo simple, no me preocupé por especificar el tipo de los elementos en la lista. Si esta advertencia te molesta, puedes arreglarla reemplazando `List` o `LinkedList` con `List<Integer>` o `LinkedList<Integer>`.

Revisa `ListClientExampleTest`. Ejecuta solo un prueba, que crea un `ListClientExample`, invoca `getList`, y luego comprueba si el resultado es una `ArrayList`. Inicialmente, este test fallará porque el resultado es una `LinkedList`, no una `ArrayList`. Ejecuta el test y confirma que falla.

NOTA: Este test tiene sentido para este ejercicio, pero no es un buen ejemplo de un test. Los buenos tests deberían comprobar si la clase que está siendo probada satisface los requerimientos de la *interfaz*; no deberían depender de los detalles de la *implementación*.

En la `ListClientExample`, reemplaza `LinkedList` con `ArrayList`. Podrías tener que agregar una instrucción `import`. Compila y ejecuta `ListClientExample`. Entonces ejecuta el test de nuevo. Con este cambio, el test debería pasar.

Para lograr que se pase el test, tuviste que cambiar `LinkedList` en el constructor; no tuviste que cambiar nada en el resto de lugares donde `List` aparece. ¿Qué sucedería si lo haces? Adelante, reemplaza una o más apariciones de `List` con `ArrayList`. El programa debería seguir funcionando correctamente, pero ahora está "sobre especificado". Si cambias de idea en el futuro y quieres intercambiar la interfaz de nuevo, tendrías que cambiar más código.

En el constructor de `ListClientExample`, ¿qué sucede si reemplazas `ArrayList` con `List`? ¿por qué no puedes instanciar una `List`?

# Capítulo 2

# Análisis de Algoritmos

Como vimos en el capítulo anterior, Java provee dos implementaciones de la interfaz `List`, `ArrayList` y `LinkedList`. Para algunas aplicaciones, `LinkedList` es más rápida; para otras, `ArrayList` es más rápida.

Para decidir cuál es mejor para una aplicación particular, una aproximación es probar ambas y ver cuánto se tardan. Esta aproximación, que se conoce como "perfilado", tiene algunos problemas:

1. Antes de comparar los algoritmos, tienes que implementar ambos.

2. Los resultados podrían depender de qué tipo de computadora uses. Un algoritmo podría ser mejor en una máquina; el otro podría ser mejor en una máquina diferente.

3. Los resultados podrían depender del tamaño del problema o de los datos que se proporcionaron como entrada.

Podemos superar algunos de estos problemas al realizar un **análisis de algoritmos**. Cuando funciona, el análisis de algoritmos hace posible comparar algoritmos sin tener que implementarlos. Pero tenemos que asumir algunos supuestos:

1. Para no lidiar con los detalles del hardware de computadoras, usualmente identificamos las operaciones básicas que conforman un algoritmo — como suma, multiplicación y comparación de números— y contamos el número de operaciones que cada algoritmo requiere.

2. Para no lidiar con los detalles de los datos de entrada, la mejor opción es analizar el desempeño promedio para las entradas que esperamos. Si eso no es posible, una alternativa común es analizar el peor escenario.

3. Finalmente, tenemos que lidiar con la posibilidad que un algoritmo funcione mejor para problemas pequeños y otro para los grandes. En ese caso, usualmente nos enfocamos en los grandes, porque para problemas pequeños la diferencia probablemente no importe, pero para problemas grandes la diferencia puede ser enorme.

Esta clase de análisis conduce a una clasificación simple de los algoritmos. Por ejemplo, si sabemos que el tiempo de ejecución del Algoritmo A tiende a ser proporcional al tamaño de las entradas, $n$, y el Algoritmo B tiende a ser proporcional a $n^2$, esperaríamos que A sea más rápido que B, al menos para valores grandes de $n$.

La mayoría de algoritmos simples pueden agruparse en unas pocas categorías.

- Tiempo constante: Un algoritmo es de "tiempo constante" si el tiempo de ejecución no depende del tamaño de las entradas. Por ejemplo, si tienes un arreglo de $n$ elementos y usas el operador de corchetes (`[]`) para acceder a uno de los elementos, esta operación requiere el mismo número de operaciones independientemente de qué tan grande sea el arreglo.

- Lineal: Un algoritmo es "lineal" si el tiempo de ejecución es proporcional al tamaño de la entrada. Por ejemplo, si sumas todos los elementos de un arreglo, tienes que acceder a $n$ elementos y realizar $n - 1$ sumas. El número total de operaciones (accesos a elementos y sumas) es $2n - 1$, que es proporcional a $n$.

- Cuadrático: Un algoritmo es "cuadrático" si el tiempo de ejecución es proporcional a $n^2$. Por ejemplo, supón que quieres comprobar si cualquiera de los elementos en una lista aparece más de una vez. Un algoritmo simple es comparar cada elemento con todos los demás. Si hay $n$ elementos y cada se compara a los $n-1$ restantes, el número total de comparaciones es $n^2 - n$, que es proporcional a $n^2$ conforme $n$ crece.

## 2.1 Ordenamiento por selección

Por ejemplo, aquí está una implementación de un algoritmo simple llamado **ordenamiento por selección** (véase http://thinkdast.com/selectsort):

```
public class SelectionSort {

    /**
     * Intercambia los elementos en los índices i y j.
     */
    public static void swapElements(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    /**
     * Encuentra el índice del valor más bajo
     * comenzando desde el índice al principio (inclusive)
     * y continuando hasta el final del arreglo.
     */
    public static int indexLowest(int[] array, int start) {
        int lowIndex = start;
        for (int i = start; i < array.length; i++) {
            if (array[i] < array[lowIndex]) {
                lowIndex = i;
            }
        }
        return lowIndex;
    }

    /**
     * Ordena los elementos (en sitio) por selección.
     */
    public static void selectionSort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            int j = indexLowest(array, i);
            swapElements(array, i, j);
        }
```

```
    }
}
```

El primer método, `swapElements`, intercambia (del inglés *swap*) dos elementos del arreglo. Leer y escribir elementos son operaciones de tiempo constante, porque si conocemos el número de elementos y la ubicación del primero, podemos calcular la ubicación de cualquier otro elemento con una multiplicación y una suma, y estas son operaciones de tiempo constante. Dado que todo en `swapElements` es de tiempo constante, el método completo es de tiempo constante.

El segundo método, `indexLowest`, encuentra el índice del elemento más pequeño del arreglo comenzando en un índice dado, `start`. Cada vez, en todas las repeticiones, accede a dos elementos del arreglo y realiza una comparación. Dado que estas son todas operaciones de tiempo constante, realmente no importa cuáles contemos. Para mantenerlo simple, contemos el número de comparaciones.

1. Si `start` es 0, `indexLowest` recorre el arreglo completo, y el número total de comparaciones es la longitud del arreglo, a la que llamaré $n$.

2. Si `start` es 1, el número de comparaciones es $n - 1$.

3. En general, el número de comparaciones es $n$ - `start`, así que `indexLowest` es lineal.

El tercer método, `selectionSort`, ordena el arreglo. Se repite desde 0 hasta $n - 1$, así que el bucle se ejecuta $n$ veces. Cada vez, llama a `indexLowest` y luego realiza una operación de tiempo constante, `swapElements`.

La primera vez que se llama a `indexLowest`, éste lleva a cabo $n$ comparaciones. La segunda vez, se llevan a cabo $n - 1$ comparaciones, y así sucesivamente. El número total de comparaciones es

$$n + n - 1 + n - 2 + ... + 1 + 0$$

La suma de esta serie es $n(n + 1)/2$, que es proporcional a $n^2$; eso implica que `selectionSort` es cuadrático.

Para obtener el mismo resultado de forma diferente, podemos pensar en `indexLowest` como un bucle anidado. Cada vez que llamamos a `indexLowest`, el número de operaciones es proporcional a $n$. Lo llamamos $n$ veces, así que el número total de operaciones es proporcional a $n^2$.

## 2.2 Notación Big O

Todos los algoritmos de tiempo constante pertenecen a un conjunto llamado $O(1)$. Así que otra manera de decir que un algoritmo es de tiempo constante es decir que está en $O(1)$. De forma similar, todos los algoritmos lineales pertenecen a $O(n)$, y todos los algoritmos cuadráticos pertenecen a $O(n^2)$. A esta forma de clasificar algoritmos se la conoce como "notación big O".

NOTA: Esta es un definición informal de la notación big O. Para un tratamiento más formal, véase http://thinkdast.com/bigo.

Esta notación provee una manera conveniente de escribir reglas generales acerca de cómo se comportan los algoritmos cuando los combinamos. Por ejemplo, si se lleva a cabo un algoritmo de tiempo lineal seguido de un algoritmo constante, el tiempo total de ejecución es lineal. Al usar $\in$ con el significado "pertenece a":

Si $f \in O(n)$ u $g \in O(1)$, $f + g \in O(n)$.

Si se llevan a cabo dos operaciones lineales, el total sigue siendo lineal.

Si $f \in O(n)$ y $g \in O(n)$, $f + g \in O(n)$.

De hecho, si se lleva a cabo una operación lineal cualquier número de veces, $k$, el total es lineal, siempre que $k$ sea una constante que no dependa de $n$.

Si $f \in O(n)$ y $k$ es una constante, $kf \in O(n)$.

Pero si llevas a cabo una operación lineal $n$ veces, el resultado es cuadrático:

Si $f \in O(n)$, $nf \in O(n^2)$.

En general, sólo nos interesa el exponente mayor de $n$. Así que si el número total de operaciones es $2n+1$, pertenece a $O(n)$. El coeficiente, 2, y el término

aditivo, 1, no son importantes para este tipo de análisis. De forma similar, $n^2 + 100n + 1000$ está en $O(n^2)$. ¡Que no te distraigan los números grandes!

"Orden de crecimiento" es otro nombre para la misma idea. Una orden de crecimiento es un conjunto de algoritmos cuyos tiempos de ejecución están en la misma categoría de big O; por ejemplo, todos los algoritmos lineales pertenecen a la misma orden de crecimiento porque sus tiempos de ejecución están en $O(n)$.

En este contexto, una"orden" es un grupo, como la *Orden de los Caballeros de la Mesa Redonda*, que es un grupo de caballeros, no una forma de alinearlos. Así que puedes imaginarte a la *Orden de los Algoritmos Lineales* como un conjunto de algoritmos valientes, galantes y particularmente eficientes.

## 2.3 Ejercicio 2

El ejercicio para este capítulo es implementar una `List` que use un arreglo de Java para guardar sus elementos.

En el repositorio de código para este libro (véase la Sección 0.1), encontrarás los archivos de código fuente que necesitarás:

- `MyArrayList.java` contiene una implementación parcial de la interfaz de `List`. Cuatro de los métodos están incompletos, tu trabajo es agregar las partes faltantes.
- `MyArrayListTest.java` contiene tests de JUnit que puedes usar para comprobar tu trabajo.

También encontrarás el archivo de construcción Ant `build.xml`. Desde el directorio `code`, deberías poder ejecutar `ant MyArrayList` para ejecutar `MyArrayList.java`, que contiene unos pocos tests simples. O puedes ejecutar `ant MyArrayListTest` para ejecutar los tests de JUnit.

Cuando ejecutes los tests, varios de ellos deberían fallar. Si examinas el código fuente, verás cuatro comentarios `TODO` (cosas por hacer) indicando los métodos que deberías completar.

Antes de comenzar a agregar el código en los métodos incompletos, revisemos una porción del código. Aquí esta la definición de la clase, variables de instancia y el constructor.

```
public class MyArrayList<E> implements List<E> {
    int size;                   // lleva un registro del número de elementos
    private E[] array;          // guarda los elementos

    public MyArrayList() {
        array = (E[]) new Object[10];
        size = 0;
    }
}
```

Como indican los comentarios, `size` registra cuántos elementos están en `MyArrayList`, y `array` es el arreglo que, de hecho, contiene los elementos.

El constructor crea un arreglo de 10 elementos, que inicialmente son `null`, y establece el `size` en 0. La mayor parte del tiempo, la longitud del arreglo es mayor que `size`, así que hay muchos espacios sin utilizar en el arreglo.

Un detalle sobre Java: no puedes instanciar un arreglo usando un parámetro de tipo; por ejemplo, lo siguiente no funcionará:

```
        array = new E[10];
```

Para superar esta limitante, tendrás que instanciar un arreglo de `Object` y después forzar la conversión de tipo. Puedes leer más al respecto en http://thinkdast.com/generics.

A continuación examinaremos el método que agrega elementos a la lista:

```
    public boolean add(E element) {
        if (size >= array.length) {
            // crear un arreglo más grande y copiar los elementos
            E[] bigger = (E[]) new Object[array.length * 2];
            System.arraycopy(array, 0, bigger, 0, array.length);
            array = bigger;
        }
        array[size] = element;
        size++;
        return true;
    }
```

Si no hay espacios sin utilizar en el arreglo, tenemos que crear un arreglo más grande y copiar los elementos a éste. Luego ya podemos guardar el elemento en el arreglo e incrementar el `size`.

Puede que no sea obvio por qué este método devuelve un booleano, dado que siempre devuelve `true`. Como siempre, puedes encontrar la respuesta en la documentación: http://thinkdast.com/colladd. Tampoco es obvio cómo analizar el desempeño de este método. En el caso normal, es de tiempo constante, pero si tenemos que redimensionar el arreglo, es lineal. Explicaré cómo manejar esto en la Sección 3.2.

Finalmente, demos una mirada a `get`; y luego ya puedes empezar con los ejercicios.

```
public T get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return array[index];
}
```

De hecho, `get` es bastante simple: si el índice está fuera de rango, lanza una excepción; de lo contrario lee y devuelve un elemento del arreglo. Nota que se comprueba si el índice es menor que `size`, no `array.length`, así que no es posible acceder a los elementos sin utilizar del arreglo.

En `MyArrayList.java`, encontrarás código de relleno para `set` que se ve como esto:

```
public T set(int index, T element) {
    // TODO: fill in this method.
    return null;
}
```

Lee la documentación de `set` en http://thinkdast.com/listset, luego llena el cuerpo de este método. Si ejecutas `MyArrayListTest` de nuevo, `testSet` debería pasar.

PISTA: Trata de no repetir el código que comprueba el índice.

Tu siguiente misión es llenar `indexOf`. Como es usual, debería leer la documentación en http://thinkdast.com/listindof así ya sabes lo que se supone que haga. En particular, nota cómo se supone que maneje `null`.

He provisto un método auxiliar llamado `equals` que compara un elementos del arreglo con un valor objetivo y devuelve `true` si son iguales (y que maneja los `null` correctamente). Nota que este método es privado porque solo se usa dentro de la clase; no es parte de la interfaz `List`.

Cuando termines, ejecuta `MyArrayListTest` de nuevo; `testIndexOf` debería pasar ahora, así como el otro test que depende de él.

Sólo dos métodos más para terminar, y habrás finalizado este ejercicio. El siguiente es una versión sobrecargada de `add` que toma un entero y guarda el nuevo valor en un índice dado, desplazando los otros elementos para hacer espacio, de ser necesario.

De nuevo, lee la documentación en http://thinkdast.com/listadd, escribe una implementación, y ejecuta los tests para confirmar.

PISTA: Evita repetir el código que hace al arreglo más grande.

El último: llena el cuerpo de `remove`. La documentación está en http://thinkdast.com/listrem. Cuando finalices este útlimo, todos los tests deberían pasar.

Una vez que tus implementaciones funcionen, compáralas con la mía, la cual puedes leer en http://thinkdast.com/myarraylist.

# Capítulo 3

# ArrayList

Este capítulo mata dos pájaros de un tiro: presento las soluciones al ejercicio anterior y demuestro una forma de clasificar algoritmos utilizando el **análisis amortizado**.

## 3.1 Clasificación de métodos de MyArrayList

Para muchos métodos, podemos identificar la orden de crecimiento al examinar el código. Por ejemplo, aquí está la implementación de `get` de `MyArrayList`:

```
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return array[index];
}
```

Todo en `get` es de tiempo constante, así que `get` es de tiempo constante. Sin problemas.

Ahora que hemos clasificado `get`, podemos clasificar `set`, que lo usa. Aquí está nuestra implementación de `set` del ejercicio anterior:

```
public E set(int index, E element) {
    E old = get(index);
    array[index] = element;
    return old;
}
```

Una parte ligeramente astuta de esta solución es que no comprueba los límites del arreglo de forma explícita, sino que se aprovecha de `get`, que lanza una excepción si el índice es inválido.

Todo en `set`, incluso la invocación de `get`, es de tiempo constante, así que `set` también es de tiempo constante.

Luego examinaremos algunos métodos lineales. Por ejemplo, aquí esta mi implementación de `indexOf`:

```
public int indexOf(Object target) {
    for (int i = 0; i<size; i++) {
        if (equals(target, array[i])) {
            return i;
        }
    }
    return -1;
}
```

En cada repetición del bucle, `indexOf` invoca a `equals`, así que tenemos que clasificar a `equals` primero. Aquí está:

```
private boolean equals(Object target, Object element) {
    if (target == null) {
        return element == null;
    } else {
        return target.equals(element);
    }
}
```

Este método invoca a `target.equals`; el tiempo de ejecución de este método podría depender del tamaño de `target` o de `element`, pero probablemente no depende del tamaño del arreglo, así que lo consideraremos de tiempo constante para el propósito de de analizar `indexOf`.

Regresando a `indexOf`, todo dentro del bucle es de tiempo constante, así que la siguiente pregunta que tenemos que considerar es: ¿cuántas veces se repite el bucle?

Si tenemos suerte, podríamos encontrar que el valor objetivo (target) al principio y devolverlo después de comprobar sólo un elemento. Si somos desafortunados, podríamos tener que comprobar todos los elementos. En promedio, esperaríamos comprobar la mitad de los elementos, así que este método es considerado lineal (excepto en el improbable caso que sepamos que el elemento objetivo está al principio del arreglo).

El análisis de `remove` es similar. Aquí está mi implementación:

```
public E remove(int index) {
    E element = get(index);
    for (int i=index; i<size-1; i++) {
        array[i] = array[i+1];
    }
    size--;
    return element;
}
```

Este método usa a `get`, que es de tiempo constante, y luego itera a lo largo del arreglo, comenzando en `index`. Si removemos el elemento al final de la lista, el bucle nunca se ejecuta y este método es de tiempo constante. Si removemos el primer elemento, recorreremos todos los elementos restantes, lo que es lineal. Así que, de nuevo, este método se considera lineal (excepto en el caso especial donde sabemos que elemento está al final o a una distancia constante del final).

## 3.2 Clasificación de `add`

Aquí está una versión de `add` que toma un índice y un elemento como parámetros:

```
public void add(int index, E element) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    // agregar el elemento para forzar el redimensionamiento
```

```
        add(element);

        // desplaza los otros elementos
        for (int i=size-1; i>index; i--) {
            array[i] = array[i-1];
        }
        // coloca el nuevo elemento en el lugar correcto
        array[index] = element;
    }
```

Esta versión con dos parámetros, llamada `add(int, E)`, usa a la versión con un parámetros, llamada `add(E)`, que agrega un nuevo elemento al final. Luego desplaza los otros elementos a la derecha, y coloca el nuevo elemento en el lugar correcto.

Antes de clasificar `add(int, E)` con dos parámetros, tenemos que clasificar `add(E)` con un parámetro:

```
    public boolean add(E element) {
        if (size >= array.length) {
            // make a bigger array and copy over the elements
            E[] bigger = (E[]) new Object[array.length * 2];
            System.arraycopy(array, 0, bigger, 0, array.length);
            array = bigger;
        }
        array[size] = element;
        size++;
        return true;
    }
```

La versión con un parámetro resulta difícil de analizar. Si hay un espacio sin utilizar en el arreglo, es de tiempo constante, pero si tenemos que redimensionar el arreglo, es lineal porque `System.arraycopy` toma tiempo proporcional al tamaño del arreglo.

¿Así que es `add` de tiempo constante o lineal? Podemos clasificar este método al pensar en el número promedio de operaciones por add en una serie de $n$ adds. Por simplicidad, asume que comenzamos con un arreglo que tiene espacio para 2 elementos.

- La primera vez que llamamos a add, encuentra un espacio libre en el arreglo, así que guarda un elemento.
- La segunda vez, encuentra un espacio libre en el arreglo, así que guarda 1 elemento.
- La tercera vez, tenemos que redimensionar el arreglo, copiar 2 elementos y guardar un elemento. Ahora el tamaño del arreglo es 4.
- La cuarta vez se guarda un elemento.
- La quinta vez redimensiona el arreglo, copia 4 elementos, y guarda 1 elemento. Ahora el tamaño del arreglo es 8.
- Los siguientes 3 llamados a add guardan 3 elementos.
- El siguiente add copia 8 y guarda 1. Ahora el tamaño es 16.
- Los siguientes 7 llamados a add guardan 7 elementos.

Y así sucesivamente. Juntando todo:

- Tras 4 llamados a add, hemos guardado 4 elementos y copiado 2.
- Tras 8 llamados a add, hemos guardado 8 elementos y copiado 6.
- Tras 16 llamados a add, hemos guardado 16 elementos y copiado 14.

Por ahora ya deberías ver el patrón: para llamar a add $n$ veces, tenemos que guardar $n$ elementos y copiar $n - 2$. Así que el número total de operaciones es $n + n - 2$, que es $2n - 2$.

Para obtener el número promedio de operaciones por add, dividimos el total por $n$; el resultado es $2 - 2/n$. Conforme $n$ se hace más grande, el segundo término, $2/n$, se hace más pequeño. Si tenemos en cuenta el principio que sólo nos interesa el exponente mayor de $n$, podemos pensar en `add` como de tiempo constante.

Podría parecer extraño que un algoritmo que a veces es lineal puede ser de tiempo constante en promedio. La clave es que duplicamos la longitud del arreglo cada vez que se redimensiona. Esto limita el número de veces que cada elemento es copiado. De otra manera — si agregáramos una cantidad fija a la

longitud del arreglo, en lugar de multiplicar por una cantidad fija — el análisis no funcionaría.

Esta forma de clasificar un algoritmo, al calcular el tiempo promedio en una serie de invocaciones, es llamada **análisis amortizado**. Puedes leer más al respecto en http://thinkdast.com/amort. La idea clave es que el costo extra de copiar el arreglo es distribuido, o "amortizado", entre una serie de invocaciones.

Ahora, si `add(E)` es de tiempo constante, ¿qué sucede con `add(int, E)`? Tras llamar a `add(E)`, itera a través de parte del arreglo y desplaza elementos. Este bucle es lineal, excepto en el caso especial donde estamos agregando al final de la lista. Así que `add(int, E)` es lineal.

## 3.3 Tamaño del problema

El último ejemplo que consideraremos es `removeAll`; aquí está la la implementación en `MyArrayList`:

```
public boolean removeAll(Collection<?> collection) {
    boolean flag = true;
    for (Object obj: collection) {
        flag &= remove(obj);
    }
    return flag;
}
```

En cada repetición, `removeAll` invoca a `remove`, que es lineal. Así que es tentador pensar que `removeAll` es cuadrático. Pero ese no es necesariamente el caso.

En este método, el bucle se ejecuta una vez para cada elemento en `collection`. Si `collection` contiene $m$ elementos y la lista de la que estamos removiendo contiene $n$ elementos, este método está en $O(nm)$. Si el tamaño de la `collection` puede ser considerado constante, `removeAll` es lineal con respecto a $n$. Pero si el tamaño de la colección es proporcional a $n$, `removeAll` es cuadrático. Por ejemplo, si `collection` siempre contiene 100 elementos o menos, `removeAll`

es lineal. Pero si `collection` generalmente contiene 1% de los elementos en la lista, `removeAll` es cuadrático.

Cuando hablamos del **tamaño del problema**, tenemos que ser cuidadosos con respecto a cuál tamaño, o tamaños, estamos hablando. Este ejemplo demuestra un error de bulto en el análisis de algoritmos: el tentador atajo de contar bucles. Si sólo hay un bucle, el algoritmo es *generalmente* linear. Si hay dos bucles (uno anidado dentro del otro), el algoritmo es *generalmente* cuadrático. ¡Pero ten cuidado! Tienes que pensar en cuántas veces se ejecuta cada bucle. Si el número de iteraciones proporcional a $n$ para todos los bucles, puedes salirte con la tuya limitándote a contar los bucles. Pero si, como en este ejemplo, el número de iteraciones no es siempre proporcional a $n$, tendrás que pensarlo con más detalle.

## 3.4 Estructuras de datos enlazadas

Para el siguiente ejercicio proveo una implementación parcial de la interfaz `List` que usa una lista enlazada para guardar los elementos. Si no estás familiarizado con las listas enlazadas, puedes leer sobre ellas en http://thinkdast.com/linkedlist, pero esta sección provee una breve introducción.

Una estructura de datos está "enlazada" si está formada por objetos, muchas veces llamados "nodos", que contienen referencias a los otros nodos. En una *lista* enlazada, cada nodo contiene una referencia al siguiente nodo en la lista. Otras estructuras enlazadas incluyen árboles y grafos, en los cuales los nodos pueden contener referencias a más de un nodo separado.

Aquí está una definición de clase para un nodo simple:

```
public class ListNode {

    public Object data;
    public ListNode next;

    public ListNode() {
        this.data = null;
        this.next = null;
    }
```

```
    public ListNode(Object data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(Object data, ListNode next) {
        this.data = data;
        this.next = next;
    }

    public String toString() {
        return "ListNode(" + data.toString() + ")";
    }
}
```

El objeto `ListNode` tiene dos variables de instancia: `data` es una referencia a algún tipo de `Object`, y `next` es una referencia al siguiente nodo en la lista. En el último nodo de la lista, por convención, `next` es `null`.

`ListNode` provee varios constructores, permitiéndote proveer valores para `data` y `next`, o inicializarlos con el valor por defecto, `null`.

Puedes pensar en cada `ListNode` como una lista con un único elemento, pero de forma más general, una lista puede contener cualquier número de nodos. Hay varias manera de crear una nueva lista. Una opción simple es crear un conjunto de objetos `ListNode`, así:

```
ListNode node1 = new ListNode(1);
ListNode node2 = new ListNode(2);
ListNode node3 = new ListNode(3);
```

Y luego enlazarlos, de esta manera:

```
node1.next = node2;
node2.next = node3;
node3.next = null;
```

Alternativamente, puedes crear un nodo y enlazarlo al mismo tiempo. Por ejemplo, si quieres agregar un nuevo nodo al principio de una lista, puedes hacerlo así::

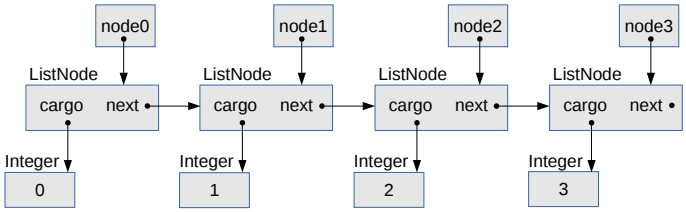


Figura 3.1: Diagrama de objeto de una lista enlazada.

```
ListNode node0 = new ListNode(0, node1);
```

Tras esta secuencia de instrucciones, tenemos cuatro nodos que contienen los `Integer`s 0, 1, 2, y 3 como datos, enlazados en orden ascendente. En el último nodo, el campo `next` es `null`.

La figura 3.1 es un diagrama de objeto que muestra estas variables y los objetos a los que se refieren. En un diagrama de objeto, las variables aparecen como nombres fuera de las cajas, con flechas que muestran a qué se refieren. Los objetos aparecen como cajas con su tipo en el exterior (como `ListNode` y `Integer`) y sus variables de instancia en el interior.

## 3.5 Ejercicio 3

En el repositorio para este libro, encontrarás los archivos de código fuente que necesitas para este ejercicio:

- `MyLinkedList.java` contiene una implementación parcial de la interfaz `List` que usa una lista enlazada para guardar los elementos.
- `MyLinkedListTest.java` contiene tests de JUnit para `MyLinkedList`.

Ejecute `ant MyLinkedList` para correr `MyLinkedList.java`, que contiene unos pocos tests simples.

Luego puedes ejecutar `ant MyLinkedListTest` para correr los tests de JUnit. Varios de ellos deberían fallar. Si examinas el código fuente, encontrarás tres comentarios `TODO` indicando los métodos que deberías rellenar.

Antes de iniciar, revisemos una porción del código. Aquí están las variables de instancia y el constructor para `MyLinkedList`:

```
public class MyLinkedList<E> implements List<E> {

    private int size;            // registra el número de elementos
    private Node head;           // referencia al primer nodo

    public MyLinkedList() {
        head = null;
        size = 0;
    }
}
```

Como lo indican los comentarios, `size` lleva el control de cuántos elementos están en `MyLinkedList`; `head` es una referencia al primer `Node` en la lista o `null` si la lista está vacía.

Guardar el número de elementos no es necesario y en general es riesgoso mantener información redundante, porque si no se actualiza correctamente, crea oportunidades para cometer errores. Además toma un poquito de espacio extra.

Pero si guardamos `size` explícitamente, podemos implementar el método `size` en tiempo constante; de otra forma, tendríamos que recorrer toda la lista y contar los elementos, lo que requiere tiempo lineal.

Porque guardamos `size` explícitamente, tenemos que actualizarlo cada vez que agregamos o removemos un elemento, así que hace un poco más lentos estos métodos, pero no cambia su orden de crecimiento, por lo que probablemente vale la pena.

El constructor establece `head` en `null`, lo que indica una lista vacía, y establece `size` en 0.

Esta clase usa un parámetro de tipo `E` para el tipo de elementos. Si no estás familiarizado con los parámetros de tipo, podrías querer leer este tutorial: http://thinkdast.com/types.

El parámetro de tipo también aparece en la definición de `Node`, que está anidada dentro de `MyLinkedList`:

```
    private class Node {
```

```
        public E data;
        public Node next;

        public Node(E data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
```

Aparte de eso, `Node` es similar a `ListNode`, presentada anteriormente.

Finalmente, aquí está mi implementación de `add`:

```
    public boolean add(E element) {
        if (head == null) {
            head = new Node(element);
        } else {
            Node node = head;
            // loop until the last node
            for ( ; node.next != null; node = node.next) {}
            node.next = new Node(element);
        }
        size++;
        return true;
    }
```

Este ejemplo demuestra dos patrones que necesitarás para tus soluciones:

1. Para muchos métodos, tenemos que tratar al primer elemento de la lista como un caso especial. En este ejemplo, si estamos agregando el primer elemento de una lista, tenemos que modificar `head`. De lo contrario, recorremos la lista, encontramos el final, y agregamos el nuevo nodo.

2. Este método muestra cómo usar un bucle `for` para recorrer los nodos en una lista. En tus soluciones, probablemente escribirás varias variantes de este bucle. Nota que tenemos que declarar `node` antes del bucle para poder acceder a él después del bucle.

Ahora es tu turno. Llena el cuerpo de `indexOf`. Como es lo usual, deberías leer la documentación, en http://thinkdast.com/listindof, para que sepas lo

que se supone que debe hacer. En particular, nota cómo se supone que maneje los `null`.

Como en el ejercicio anterior, proveo un método auxiliar llamado `equals` que compara un elemento de un arreglo a un valor objetivo y comprueba si son iguales — y que maneja los `null` correctamente. Este método es privado porque se usa dentro de esta clase pero no es parte de la interfaz `List`.

Cuando termines, ejecuta los tests de nuevo; `testIndexOf` debería pasar ahora, así como los otros tests que dependen de él.

Luego, deberías completar la versión con dos parámetros de `add`, que toma un índice y guarda el nuevo valor en el índice dado. De nuevo, lee la documentación en http://thinkdast.com/listadd, escriba una implementación y ejecuta los tests para confirmar.

El último: llena el cuerpo de `remove`. Aquí está la documentación: http://thinkdast.com/listrem. Cuando finalices este, todos los tests deberían pasar.

Una vez que tus implementaciones funcionen, compáralas a la versión en el directorio `solution` del repositorio.

## 3.6 Una nota sobre la recolección de basura

En `MyArrayList` del ejercicio anterior, el arreglo crece si es necesario, pero nunca se reduce. El arreglo nunca es recolectado y los elementos no son limpiados por el recolector de basura hasta que la lista como tal es destruida.

Una ventaja de la implementación de lista enlazada es que se reduce cuando los elementos son removidos y los nodos sin utilizar son limpiados por el recolector de basura inmediatamente.

Aquí está mi implementación del método `clear`:

```
public void clear() {
    head = null;
    size = 0;
}
```

Cuando establecemos el valor de `head` en `null`, removemos una referencia al primer `Node`. Si no hay otras referencias a ese `Node` (y no deberían haber), también es limpiado por el recolector de basura. Este proceso continúa hasta que todos los nodos son limpiados.

Así que, ¿cómo deberíamos clasificar `clear`? El método en si mismo contiene dos operaciones de tiempo constante, así que seguro parece como de tiempo constante. Pero cuando lo invocas, haces que el recolector de basura realice un trabajo que es proporcional al número de elementos. ¡Así que tal vez deberíamos considerarlo lineal!

Este es un ejemplo de lo que a veces se conoce como un **bug de rendimiento**: un programa que es correcto en el sentido que hace lo correcto, pero que no pertenece a la orden de crecimiento que esperábamos. En lenguajes como Java que realizan gran parte del trabajo, como la recolección de basura, tras bambalinas, esta clase de bug puede ser difícil de encontrar.

# Capítulo 4

# LinkedList

Este capítulo presenta soluciones al ejercicio previo y continúa la discusión sobre el análisis de algoritmos.

## 4.1 Clasificación de los métodos de `MyLinkedList`

Mi implementación de `indexOf` se encuentra a continuación. Léela y mira si puedes identificar su orden de crecimiento antes de leer la explicación.

```
public int indexOf(Object target) {
    Node node = head;
    for (int i=0; i<size; i++) {
        if (equals(target, node.data)) {
            return i;
        }
        node = node.next;
    }
    return -1;
}
```

Inicialmente `node` obtiene una copia de `head`, así que ambos se refieren al mismo `Node`. La variable de repetición, `i`, cuenta desde 0 hasta `size-1`. En cada repetición del bucle, usamos `equals` para ver si hemos encontrado el

objetivo (`target`). De ser así, devolvemos `i` inmediatamente. De lo contrario, avanzamos al siguiente `Node` en la lista.

Normalmente deberíamos asegurarnos que el siguiente `Node` no sea `null`, pero en este caso es seguro porque el bucle termina al llegar al final de la lista (asumiendo que `size` es consistente con el número exacto de nodos en la lista).

Si el bucle finaliza sin haber encontrado al objetivo, devolvemos `-1`.

Así que, ¿cuál es la orden de crecimiento para este método?

1. En cada repetición del bucle invocamos `equals`, que es de tiempo constante (lo que podría depender del tamaño de `target` o `data`, pero no depende del tamaño de la lista). Las otras operaciones en el bucle son también de tiempo constante.

2. El bucle podría ejecutarse $n$ veces, porque en el peor escenario, podríamos tener que recorrer la lista completa.

Así que el tiempo de ejecución de este método es proporcional a la longitud de la lista.

Para continuar, aquí está mi implementación del método `add` con dos parámetros. De nuevo, deberías tratar de clasificarlo antes de leer la explicación.

```
public void add(int index, E element) {
    if (index == 0) {
        head = new Node(element, head);
    } else {
        Node node = getNode(index-1);
        node.next = new Node(element, node.next);
    }
    size++;
}
```

Si `index==0`, agregamos el nuevo `Node` al principio, así que lo trataremos como un caso especial. De lo contrario, tendríamos que recorrer la lista para encontrar el elemento en `index-1`. Usamos el método auxiliar `getNode`:

```
private Node getNode(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    Node node = head;
    for (int i=0; i<index; i++) {
        node = node.next;
    }
    return node;
}
```

`getNode` comprueba si el `index` está fuera de rango; de ser así, lanza una excepción. De otra forma, recorre la lista y devuelve el Nodo solicitado.

Volviendo a `add`, una vez encontramos el `Node` apropiado, creamos el nuevo `Node` y lo colocamos entre `node` y `node.next`. Podrías encontrar útil dibujar un diagrama de esta operación para estar seguro que la entiendes.

Así, ¿cuál es la orden de crecimiento para `add`?

1. `getNode` es similar a `indexOf`, y es lineal por la misma razón.
2. En `add`, todo antes y después de `getNode` es de tiempo constante.

Al combinar todo, `add` es lineal.

Finalmente, examinemos `remove`:

```
public E remove(int index) {
    E element = get(index);
    if (index == 0) {
        head = head.next;
    } else {
        Node node = getNode(index-1);
        node.next = node.next.next;
    }
    size--;
    return element;
}
```

`remove` usa `get` para encontrar y guardar el elemento en `index`. Luego remueve el `Node` que lo contiene.

Si `index==0`, de nuevo lo tratamos como un caso especial. De lo contrario encontramos el nodo en `index-1` y lo modificamos para que se salte `node.next` y se enlace directamente con `node.next.next`. Esto efectivamente remueve `node.next` de la lista, y puede ser limpiado por el recolector de basura.

Finalmente, disminuimos `size` y devolvemos el elemento que recuperamos al principio.

Así, ¿cuál es la orden de crecimiento para `remove`? Todo en `remove` es de tiempo constante excepto `get` y `getNode`, que son lineales. Así que `remove` es lineal.

Cuando las personas ven dos operaciones lineales, a veces creen que el resultado es cuadrático, pero eso solo aplica si una operación está anidada dentro de la otra. Si invocas una operación después de la otra, los tiempos de ejecución se suman. Si ambas están en $O(n)$, la suma también está en $O(n)$.

## 4.2 Comparación entre `MyArrayList` y `MyLinkedList`

La siguiente tabla resume las diferencias entre `MyLinkedList` y `MyArrayList`, donde `1` significa $O(1)$ o tiempo constante y $n$ significa $O(n)$ o lineal.

| | MyArrayList | MyLinkedList |
|---|---|---|
| add (al final) | **1** | n |
| add (al principio) | n | **1** |
| add (en general) | n | n |
| get / set | **1** | n |
| indexOf / lastIndexOf | n | n |
| isEmpty / size | 1 | 1 |
| remove (del final) | **1** | n |
| remove (del principio) | n | **1** |
| remove (en general) | n | n |

Las operaciones en las `MyArrayList` destaca son agregar (add) al final, remover (remove) del final, obtener (get) y establecer (set).

Las operaciones en las que `MyLinkedList` destaca son agregar al principio y remover del principio.

Para otras operaciones, las dos implementaciones están en la misma orden de crecimiento.

¿Cuál implementación es mejor? Depende de cuáles operaciones es probable que uses más. Y es por eso que Java provee más de una implementación, porque esto depende.

## 4.3 Perfilado

Para el siguiente ejercicio proveo una clase llamada `Profiler` que contiene código que ejecuta un método para varios tamaños de problemas, mide el tiempo de ejecución y grafica los resultados.

Usarás un `Profiler` (Perfilador) para clasificar el desempeño de los métodos `add` para las implementaciones en Java de `ArrayList` y `LinkedList`.

Aquí está un ejemplo que muestra como usar el perfilador:

```
public static void profileArrayListAddEnd() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new ArrayList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
        }
    };

    String title = "ArrayList add end";
    Profiler profiler = new Profiler(title, timeable);

    int startN = 4000;
    int endMillis = 1000;
    XYSeries series = profiler.timingLoop(startN, endMillis);
    profiler.plotResults(series);
}
```

Este método mide el tiempo que toma ejecutar `add` en una `ArrayList`, el cual agrega el nuevo elemento al final. Explicaré el código y luego mostraré los resultados.

Para usar el `Profiler`, necesitamos crear un objeto `Timeable` que provee dos métodos: `setup` y `timeMe`. El método `setup` hace lo que sea que se necesite hacer antes de iniciar el cronómetro; en este caso crea una lista vacía. Luego `timeMe` realiza cualquier operación que estemos tratando de medir; en este caso agrega $n$ elementos a la lista.

El código que crea `timeable` es una **clase anónima** que define una nueva implementación de la interfaz `Timeable` y crea una instancia de la nueva clase al mismo tiempo. Si no estás familiarizado con las clases anónimas, puedes leer sobre ellas aquí: http://thinkdast.com/anonclass.

Pero no necesitas conocer mucho para el siguiente ejercicio; incluso si no te sientes cómodo con las clases anónimas, puedes copiar y modificar el código de ejemplo.

El siguiente paso es crear el objeto `Profiler` pasándole el objeto `Timeable` y un título como parámetros.

El `Profiler` provee `timingLoop` que usa el objeto `Timeable` guardado como una variable de instancia. Éste invoca al método `timeMe` en el objeto `Timeable` varias veces con un rango de valores de $n$. `timingLoop` toma dos parámetros:

- `startN` es el valor de $n$ con el que debería iniciar el temporizador.
- `endMillis` es un umbral en milisegundos. Conforme `timingLoop` incrementa el tamaño del problema, el tiempo de ejecución se incrementa; cuando el tiempo de ejecución sobrepasa este umbral, `timingLoop` se detiene.

Cuando realices los experimentos, podrías tener que ajustar estos parámetros. Si `startN` es muy bajo, el tiempo de ejecución podría ser muy corto para medirlo con exactitud. Si `endMillis` es muy bajo, puede que no obtengas datos suficientes para ver una relación clara entre el tamaño del problema y el tiempo de ejecución.

Este código está en `ProfileListAdd.java`, que ejecutarás en el siguiente ejercicio. Cuando yo lo ejecuté, obtuve esta salida:

```
4000, 3
8000, 0
16000, 1
32000, 2
64000, 3
128000, 6
256000, 18
512000, 30
1024000, 88
2048000, 185
4096000, 242
8192000, 544
16384000, 1325
```

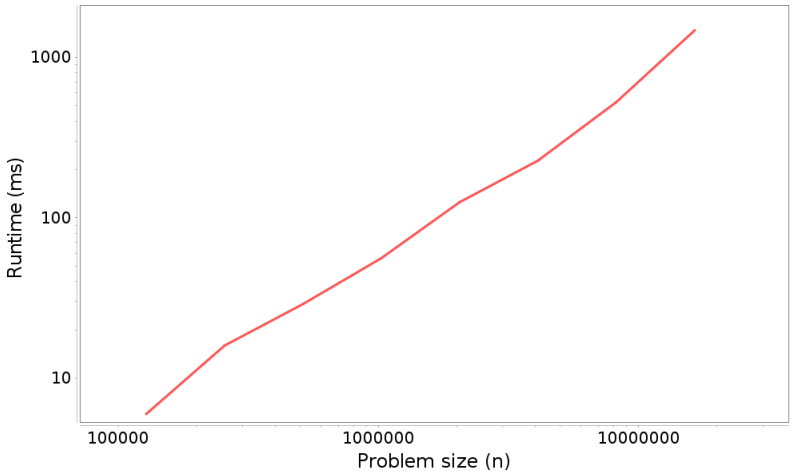


Figura 4.1: Resultados del perfilado: tiempo de ejecución versus tamaño del problema para agregar $n$ elementos al final de una `ArrayList`.

La primera columna es el tamaño del problema, $n$; la segunda columna es el tiempo de ejecución en milisegundos. Las primeras mediciones contienen bastante ruido, podría haber sido mejor establecer `startN` alrededor de 64000.

El resultado del `timingLoop` es una `XYSeries` que contiene estos datos. Si pasas esta serie a `plotResults`, genera una gráfica como la de Figura 4.1.

La siguiente sección explica cómo interpretarla.

## 4.4 Interpretación de los resultados

Con base en nuestro entendimiento sobre cómo funciona `ArrayList`, esperaríamos que el método `add` tome una cantidad de tiempo constante cuando agregamos elementos al final. Así que el tiempo total para agregar $n$ elementos debería ser lineal.

Para comprobar esta teoría, podríamos graficar el tiempo total de ejecución versus el tamaño del problema, y deberíamos ver una línea recta, al menos para tamaños de problemas lo suficientemente grandes para obtener mediciones precisas. Matemáticamente, podemos escribir la función para esa línea:

$$\text{tiempo de ejecución} = a + bn$$

donde $a$ es la intersección de la línea y $b$ es la pendiente.

Por otro lado, si `add` es lineal, el tiempo total para $n$ adds sería cuadrático. Si graficamos el tiempo de ejecución versus el tamaño del problema, esperaríamos ver una parábola. O matemáticamente, algo como:

$$\text{tiempo de ejecución} = a + bn + cn^2$$

Con datos perfectos, podríamos ser capaces de distinguir entre una línea recta y una parábola, pero si las mediciones contienen ruido, esto puede ser difícil. Una mejor manera de interpretar mediciones con ruido es graficar el tiempo de ejecución y el tamaño del problema en una escala **log-log**.

¿Por qué? Supongamos que el tiempo de ejecución es proporcional a $n^k$, pero no sabemos cuál es el exponente $k$. Podemos escribir la relación como sigue:

$$\text{tiempo de ejecución} = a + bn + \ldots + cn^k$$

Para valores grandes de $n$, el término con el exponente mayor es el más importante, así que:

$$\text{tiempo de ejecución} \approx cn^k$$

donde $\approx$ significa "aproximadamente igual a". Ahora, si calculamos el logaritmo de ambos lados de la ecuación:

$$\log(\text{tiempo de ejecución}) \approx \log(c) + k \log(n)$$

Esta ecuación implica que si graficamos tiempo de ejecución versus $n$ en una escala log-log, esperaríamos ver una línea recta con intersección $\log(c)$ y pendiente $k$. No nos interesa mucho la intersección, pero la pendiente indica la orden de crecimiento: si $k = 1$, el algoritmo es lineal; si $k = 2$, es cuadrático.

Al observar la figura de la sección previa, puedes estimar la pendiente visualmente. Pero cuando llamas a `plotResults` se calcula un ajuste de mínimos cuadrados y se imprime la pendiente estimada. En este ejemplo:

```
Estimated slope = 1.06194352346708
```

Que es cercano a 1; y eso sugiere que el tiempo total para $n$ llamadas a add es lineal, así que cada add es de tiempo constante, tal como esperábamos.

Un punto importante: si ves una línea recta en una gráfica como esta, eso **no** significa que el algoritmo es lineal. Si el tiempo de ejecución es proporcional a $n^k$ para cualquier exponente $k$, esperaríamos ver una línea recta con pendiente $k$. Si la pendiente es cercana a 1, eso sugiere que el algoritmo es lineal. Si es cercana a 2, probablemente sea cuadrático.

## 4.5 Ejercicio 4

En el repositorio para este libro encontrarás los archivos de código fuente que necesitarás para este ejercicio:

1. `Profiler.java` contiene la implementación de la clase `Profiler` descrita anteriormente. Usarás esta clase, aunque no tienes que saber cómo funciona. Pero siéntete libre de leer el código.

2. `ProfileListAdd.java` contiene un código inicial para este ejercicio, incluyendo el ejemplo anterior, que perfila `ArrayList.add`. Modificarás este archivo para perfilar algunos otros métodos.

También, en el directorio `code`, encontrarás el archivo de construcción Ant `build.xml`.

Ejecuta `ant ProfileListAdd` para correr `ProfileListAdd.java`. Deberías obtener resultados similares a la Figura 4.1, pero puede que tengas que ajustar `startN` o `endMillis`. La pendiente estimada debería ser cercana a 1, lo que indica que realizar $n$ operaciones add toma un tiempo proporcional a $n$ elevada al exponente 1; es decir, está en $O(n)$.

En `ProfileListAdd.java`, encontrarás un método vacío llamado `profileArrayListAddBeginning`. Llena el cuerpo de este método con el código que prueba `ArrayList.add`, siempre poniendo el nuevo elemento al principio. Si inicias con una copia de `profileArrayListAddEnd`, deberías tener que hacer unos cuantos cambios. Agrega una línea en `main` para invocar este método.

Ejecuta `ant ProfileListAdd` de nuevo e interpreta los resultados. Con base en nuestro entendimiento de cómo funciona una `ArrayList` esperaríamos que cada operación add fuese lineal, así que el tiempo total para $n$ llamadas a add debería ser cuadrático. De ser el caso, la pendiente estimada de la línea, en una escala log-log, debería ser cercana a 2. ¿Lo es?

Ahora comparemos estas mediciones con el desempeño de `LinkedList`. Llena el cuerpo de `profileLinkedListAddBeginning` y úsalo para clasificar `LinkedList.add` cuando colocamos el nuevo elemento al principio. ¿Qué desempeño esperaríamos? ¿Son los resultados consistentes con tus expectativas?

Finalmente, llena el cuerpo de `profileLinkedListAddEnd` y úsalo para clasificar `LinkedList.add` cuando colocamos el nuevo elemento al final. ¿Qué rendimiento esperarías? ¿Son los resultados consistentes con tus expectativas?

Presentaré los resultados y las respuestas a estas preguntas en el siguiente capítulo.

# Capítulo 5

# Listas de enlace doble

Este capítulo revisa los resultados de los ejercicios previos e introduce otra implementación de la interfaz `List`, la lista de enlace doble.

## 5.1 Resultados del perfilado de desempeño

En el ejercicio anterior, usamos `Profiler.java` para realizar varias operaciones con `ArrayList` y `LinkedList` para un rango de tamaños de problemas. Graficamos los tiempos de ejecución versus el tamaño del problema en una escala log-log y estimamos la pendiente de la curva resultante, que indica el exponente predominante de la relación entre tiempo de ejecución y tamaño del problema.

Por ejemplo, cuando usamos el método `add` para agregar elementos al final de una `ArrayList`, encontramos que el tiempo total para llevar a cabo $n$ llamadas a add era proporcional a $n$; es decir, la pendiente estimada era cercana a 1. Concluimos que completar $n$ llamadas a add está en $O(n)$, así que en promedio el tiempo para una llamada a add es de tiempo constante, o $O(1)$, que es lo que esperaríamos con base en el análisis de los algoritmos.

El ejercicio te pedía llenar el cuerpo de `profileArrayListAddBeginning`, que prueba el desempeño de agregar nuevos elementos al principio de una

`ArrayList`. Con base en nuestro análisis, esperábamos que add fuera lineal, porque tiene que desplazar los otros elementos a la derecha; así que esperaríamos que $n$ llamadas a add sean cuadráticas.

Aquí está una solución, que puedes encontrar en el directorio `solution` del repositorio.

```
public static void profileArrayListAddBeginning() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new ArrayList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
    int startN = 4000;
    int endMillis = 10000;
    runProfiler("ArrayList add beginning", timeable, startN, endMillis);
}
```

Este método es casi idéntico a `profileArrayListAddEnd`. La única diferencia está en `timeMe`, que usa la versión de dos parámetros de `add` para colocar el nuevo elemento en el índice 0. También, incrementos `endMillis` para obtener un punto de datos adicional.

Aquí están los resultados de las mediciones (tamaño del problema a la izquierda, tiempo de ejecución en milisegundos a la derecha):

```
4000, 14
8000, 35
16000, 150
32000, 604
64000, 2518
128000, 11555
```

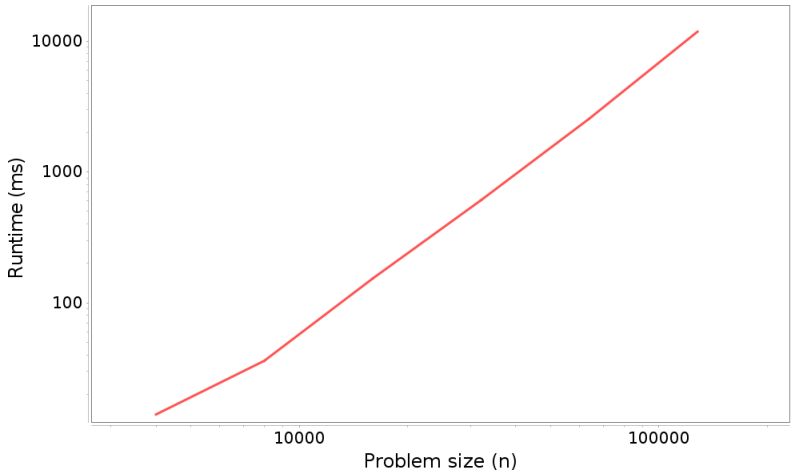


Figura 5.1: Resultados del perfilado: tiempo de ejecución versus tamaño del problema para agregar $n$ elementos al principio de una `ArrayList`.

La figura 5.1 muestra la gráfica de tiempo de ejecución versus tamaño del problema.

Recuerda que una línea recta en esta gráfica **no** significa que el algoritmo es lineal. En su lugar, si el tiempo de ejecución es proporcional a $n^k$ para cualquier exponente, $k$, esperaríamos ver una línea recta con pendiente $k$. En este caso, esperaríamos que el tiempo total para $n$ llamadas a add sea proporcional a $n^2$, así que esperaríamos una línea recta con pendiente 2. De hecho, la pendiente estimada es 1.992, que se acerca tanto que me daría temor falsificar datos tan buenos.

## 5.2 Perfilado de los métodos de `LinkedList`

En el ejercicio anterior también perfilaste el desempeño de agregar nuevos elementos al principio de una `LinkedList`. Con base en nuestro análisis, esperaríamos que cada llamada a `add` tomara tiempo constante, porque en una lista enlazada, no tenemos que desplazar los elementos existentes; podemos simplemente agregar un nuevo nodo al principio. Así que esperaríamos que el tiempo total para $n$ llamadas a add fuese lineal.

Aquí está una solución:

```
public static void profileLinkedListAddBeginning() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new LinkedList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
    int startN = 128000;
    int endMillis = 2000;
    runProfiler("LinkedList add beginning", timeable, startN, endMillis);
}
```

Tenemos que hacer unos cuantos cambios, reemplazando `ArrayList` con `LinkedList` y ajustando `startN` y `endMillis` para obtener un buen rango de datos. Las mediciones contenían más ruido que el anterior intento; aquí están los resultados:

```
128000, 16
256000, 19
512000, 28
1024000, 77
2048000, 330
4096000, 892
8192000, 1047
16384000, 4755
```

La figura 5.2 muestra la gráfica de estos resultados.

No es una línea muy recta, y la pendiente no es exactamente 1; la pendiente del ajuste de mínimos cuadrados es 1.23. Pero estos resultados indican que el tiempo total para $n$ llamadas a add es por lo menos aproximadamente $O(n)$, así que cada llamada a add es de tiempo constante.

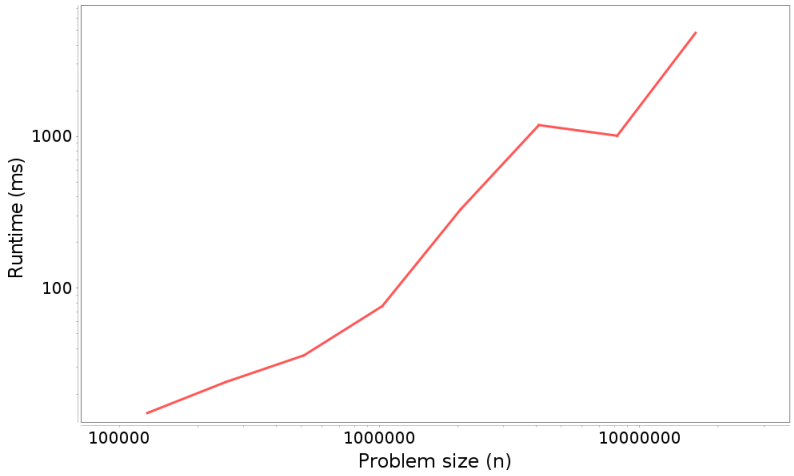


Figura 5.2: Resultados del perfilado: tiempo de ejecución versus tamaño del problema para agregar $n$ elementos al principio de una `LinkedList`.

## 5.3 Agregar al final de una `LinkedList`

Agregar elementos al principio es una de las operaciones donde esperamos que `LinkedList` sea más rápida que `ArrayList`. Pero para agregar elementos al final, esperamos que `LinkedList` sea más lenta. En mi implementación, tenemos que recorrer la lista completa para agregar un elemento al final, lo que es lineal. Así que esperamos que el tiempo total para $n$ llamadas a add sea cuadrático.

Bien, no lo es. Aquí está el código:

```
public static void profileLinkedListAddEnd() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new LinkedList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
```

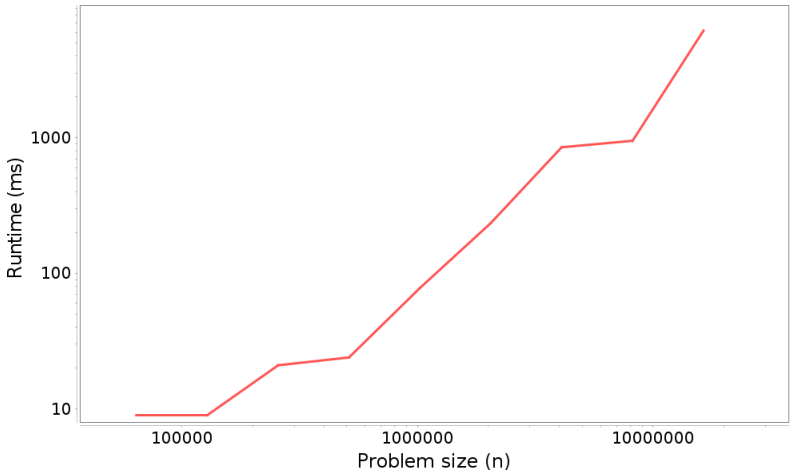


Figura 5.3: Resultados del perfilado: tiempo de ejecución versus tamaño del problema para agregar $n$ elementos al final de una `LinkedList`.

```
            }
        };
        int startN = 64000;
        int endMillis = 1000;
        runProfiler("LinkedList add end", timeable, startN, endMillis);
    }
```

Aquí están los resultados:

```
64000, 9
128000, 9
256000, 21
512000, 24
1024000, 78
2048000, 235
4096000, 851
8192000, 950
16384000, 6160
```

La figura 5.3 muestra la gráfica de estos resultados.

De nuevo, las medidas contienen mucho ruido y la línea no es perfectamente recta, pero la pendiente estimada es 1.19, que es cercana a la que obtuvimos al

agregar elementos al principio, y no muy cercana a 2, que es lo que esperábamos con base en nuestro análisis. De hecho, es más cercana a 1, lo que sugiere que agregar elementos al final es de tiempo constante. ¿Qué está sucediendo?

## 5.4 Lista de enlace doble

Mi implementación de una lista enlazada, `MyLinkedList`, usa una lista de enlace sencillo; es decir, cada elemento contiene un enlace al siguiente y el objeto `MyArrayList` en sí mismo tiene un enlace al primer nodo.

Pero si lees la documentación de `LinkedList` en http://thinkdast.com/linked, dice:

> Implementación de lista de enlace doble de las interfaces List y Deque. [...] Todas las operaciones se desempeñan como podría esperarse de una lista de enlace doble. Las operaciones provean un índice a la lista recorrerán la lista desde el principio o desde el final, dependiendo de cuál esté más cerca al índice especificado.

Si no estás familiarizado con las listas de enlace doble, puedes leer más sobre ellas en http://thinkdast.com/doublelist, pero la versión corta es:

- Cada nodo contiene un enlace al siguiente nodo y un enlace al nodo previo.
- El objeto `LinkedList` contiene enlaces al primer y último elementos de la lista.

Así que podemos empezar en cualquier extremo de la lista y recorrerla en cualquier dirección. Como resultado, ¡podemos agregar y remover elementos desde el principio y el final de la lista en tiempo constante!

La siguiente tabla resume el desempeño que esperaríamos de `ArrayList`, `MyLinkedList` (enlace simple), y `LinkedList` (enlace doble):

| | MyArrayList | MyLinkedList | LinkedList |
|---|---|---|---|
| add (al final) | **1** | n | **1** |
| add (al principio) | n | **1** | **1** |
| add (en general) | n | n | n |
| get / set | **1** | n | n |
| indexOf / lastIndexOf | n | n | n |
| isEmpty / size | 1 | 1 | 1 |
| remove (del final) | **1** | n | **1** |
| remove (del principio) | n | **1** | **1** |
| remove (en general) | n | n | n |

## 5.5 Elección de una estructura

La implementación de enlace doble es mejor que `ArrayList` para agregar y remover al principio e igual de buena que `ArrayList` para agregar y remover al final. Así que la única ventaja de `ArrayList` es para `get` y `set`, que requieren tiempo lineal en una lista enlazada, incluso si es de enlace doble.

Si sabes que el tiempo de ejecución de tu aplicación depende del tiempo que toma obtener (`get`) o modificar (`set`) los elementos, una `ArrayList` podría ser la mejor opción. Si el tiempo de ejecución depende de agregar y remover elementos cerca del principio o del final, `LinkedList` podría ser mejor.

Pero recuerda que estas recomendaciones se basan en la orden de crecimiento para problemas grandes. Hay otros factores a considerar:

- Si estas operaciones no requieren una fracción sustancial del tiempo de ejecución para tu aplicación — es decir, si tu aplicación pasa la mayor parte de su tiempo haciendo otras cosas — entonces tu elección de una implementación de `List` no importará mucho.
- Si las listas con las que estás trabajando no son muy grandes, podría ser que no obtengas el desempeño que esperas. Para problemas pequeños, un

algoritmo cuadrático podría ser más rápido que un algoritmo lineal, o un algoritmo lineal podría ser más rápido que uno de tiempo constante. Y para problemas pequeños, la diferencia probablemente no es importante.

- También, no olvides el espacio. Hasta el momento nos hemos enfocado en el tiempo de ejecución, pero diferentes implementaciones requieren diferentes cantidades de espacio. En una `ArrayList`, los elementos se guardan uno a la par de otro en un trozo de la memoria, así que hay poco espacio desperdiciado y el hardware de la computadora es más rápido con trozos contiguos. En una lista enlazada, cada elemento requiere un nodo con uno o dos enlaces. Los enlaces ocupan espacio (¡a veces más que los datos!), y con nodos dispersos por todas partes en la memoria, el hardware podría ser menos eficiente.

En resumen, el análisis de algoritmos provee algunas guías para elegir entre estructuras de datos, pero solo si

1. El tiempo de ejecución de tu aplicación es importante,
2. El tiempo de ejecución de tu aplicación depende de la elección de una estructura de datos y
3. El tamaño del problema es lo suficientemente grande para que el orden de crecimiento realmente prediga cuál estructura de datos es mejor.

Podrías tener una larga carrera como ingeniero de software y nunca encontrarte en esta situación.

# Capítulo 6

# Recorrido de árboles

Este capítulo introduce la aplicación que desarrollaremos durante el resto del libro, un motor de búsqueda web. Describo los elementos de un motor de búsqueda e introduzco la primera aplicación, un rastreador Web que descarga e interpreta páginas de Wikipedia. Este capítulo también presenta una implementación recursiva de la búsqueda en profundidad y una implementación iterativa que usa una `Deque` de java para implementar una pila "último en entrar, primero en salir".

## 6.1 Motores de búsqueda

Un **motor de búsqueda web**, como Google Search o Bing, toma un conjunto de "términos de búsqueda" y devuelve una lista de páginas web que son relevantes para esos términos (explicaré más tarde lo que significa "relevante"). Puedes leer más en http://thinkdast.com/searcheng, pero explicaré lo que necesites conforme que avancemos.

Los componentes esenciales de un motor de búsqueda son:

- Rastrear: Necesitaremos un programa que pueda descargar una página, interpretarla y extraer el texto y cualquier enlace a otras páginas.
- Indexar: Necesitaremos una estructura de datos que haga posible buscar un término de búsqueda y encontrar las páginas que lo contienen.

- Recuperar: Y necesitaremos una forma de recolectar los resultados del Índice e identificar las páginas que son más relevantes para los términos de búsqueda.

Comenzaremos con el rastreador. La meta de un rastreador es descubrir y descargar un conjunto de páginas web. Para motores de búsqueda como Google y Bing, la meta es encontrar *todas* las páginas web, pero muchas veces los rastreadores están limitados a un dominio más pequeño. En nuestro caso, sólo leeremos páginas de Wikipedia.

Como un primer paso, construiremos un rastreador que lea una página de Wikipedia, encuentre el primer enlace, siga un enlace a otra página y repita el proceso. Usaremos este rastreador para comprobar la conjetura "Llegar a la Filosofía", que establece:

> Al hacer clic en el primer enlace en minúsculas en el texto principal de un artículo de Wikipedia, y luego repetir el proceso para artículos subsiguientes, por lo general eventualmente te conducirá al artículo sobre Filosofía.

La conjetura se establece en `http://thinkdast.com/getphil`, y puedes leer su historia ahí.

Probar la conjetura nos permitirá construir las piezas básicas de un rastreador sin tener que rastrear toda la web, o incluso toda Wikipedia. ¡Y pienso que el ejercicio es algo divertido!

En unos cuantos capítulos, trabajaremos en el indexador, y luego en el recuperador.

## 6.2 Interpretación del HTML

Cuando descargas una página web, los contenidos se escriben en el Lenguaje de Marcas de Hipertexto, también conocido como HTML. Por ejemplo, aquí está un documento de HTML con los contenidos mínimos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

Las frases "This is a title" y "Hello world!" son el texto que de hecho aparece en la página; los otros elementos son **etiquetas** que indican cómo debería mostrarse el texto.

Cuando nuestro rastreador descarga una página, necesitará interpretar el HTML para extraer el texto y encontrar los enlaces. Para hacerlo, usaremos **jsoup**, que es una biblioteca de Java de código abierto que descarga e interpreta HTML.

El resultado de interpretar HTML es un árbol de Modelo de Objeto de Documento (DOM) o **árbol DOM**, que contiene los elementos de un documento, incluyendo texto y etiquetas. El árbol es una estructura de datos enlazada formada por nodos; los nodos representan texto, etiquetas y otros elementos del documento.

Las relaciones entre los nodos son determinadas por la estructura del documento. En el ejemplo anterior, el primer nodo, llamado la **raíz**, es la etiqueta `<html>`, la cual contiene enlaces a los dos nodos que contiene, `<head>` y `<body>`; estos nodos son los **hijos** del nodo raíz.

El nodo `<head>` tiene un hijo, `<title>`, y el nodo `<body>` tiene un hijo, `<p>` (que significa "párrafo"). La figura 6.1 representa este árbol gráficamente.

Cada nodo contiene enlaces a sus hijos; además, cada nodo contiene un enlace a su **padre**, así que desde cualquier nodo es posible navegar hacia arriba y hacia abajo en el árbol. El árbol DOM para páginas web reales usualmente es más complicado que este ejemplo.

La mayoría de los navegadores web proveen herramientas para inspeccionar el DOM de la página que estás viendo. En Chrome, puedes hacer clic derecho sobre cualquier parte de una página web y seleccionar "Inspeccionar"

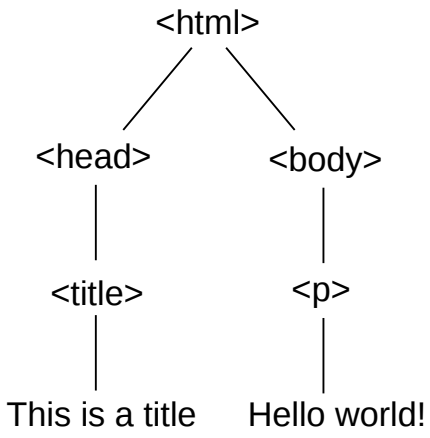


Figura 6.1: Árbol DOM para una página HTML simple.

desde el menú emergente. En Firefox, puedes hacer clic derecho y seleccionar "Inspeccionar Elemento" del menú. Safari provee una herramienta llamada Inspector Web, sobre la que puedes leer en http://thinkdast.com/safari. Para Internet Explorer, puedes leer las instrucciones en http://thinkdast.com/explorer.

La figura 6.2 muestra una captura de pantalla del DOM para la página de Wikipedia en Java, http://thinkdast.com/java. El elemento que está resaltado es el primer párrafo del texto principal del artículo, que está contenido dentro de un elemento `<div>` con `id="mw-content-text"`. Usaremos el id de este elemento para identificar el texto principal de cada artículo que descarguemos.

## 6.3 Uso de jsoup

jsoup facilita descargar e interpretar páginas web y para navegar el árbol DOM. Aquí está un ejemplo:

```
String url = "http://en.wikipedia.org/wiki/Java_(programming_language)";

// descarga e interpreta el documento
Connection conn = Jsoup.connect(url);
```

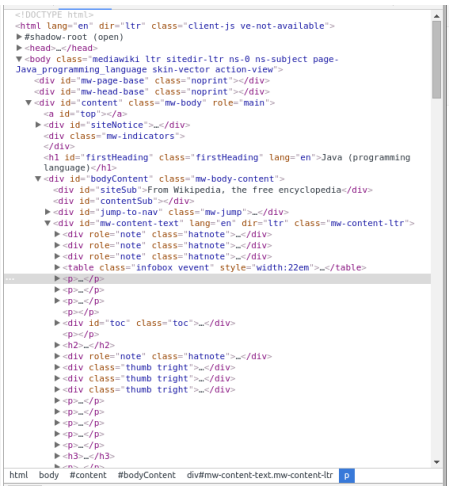


Figura 6.2: Captura de pantalla del Inspector DOM de Chrome.

```
Document doc = conn.get();

// selecciona el texto del contenido y extráelo de los párrafos.
Element content = doc.getElementById("mw-content-text");
Elements paragraphs = content.select("p");
```

`Jsoup.connect` toma una URL como una `String` y se conecta al servidor web; el método `get` descarga el HTML, lo interpreta, y devuelve un objeto `Document`, que representa el DOM.

`Document` provee métodos para navegar por el árbol y seleccionar nodos. De hecho, provee tantos métodos, que puede ser confuso. Este ejemplo demuestra dos formas de seleccionar nodos:

- `getElementById` toma una `String` y busca un elemento en el árbol que tenga un campo "id" que coincida. Aquí selecciona el nodo `<div id="mw-content-text"` que aparece en cada página de Wikipedia para identificar el elemento `<div>` que contiene el texto principal de la página, en oposición a la barra de navegación y otros elementos.

  El valor de retorno de `getElementById` es un objeto `Element` que representa este `<div>` y contiene los elementos en el `<div>` como hijos, nietos, etc.

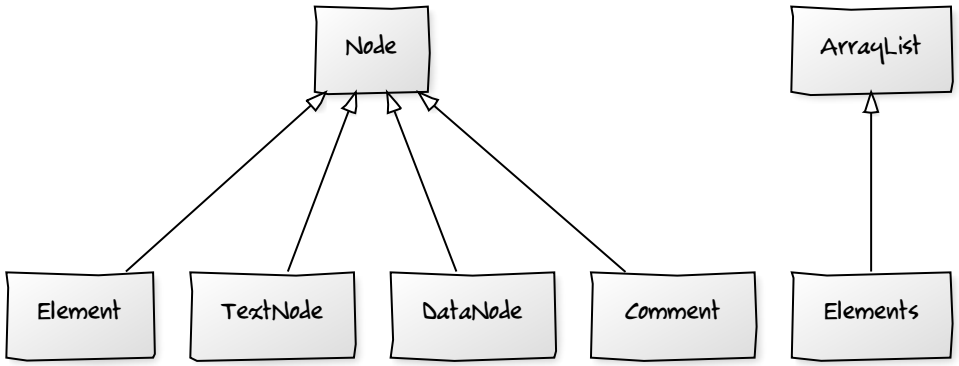


Figura 6.3: Diagrama UML para clases seleccionadas provistas por jsoup.

- `select` toma una `String`, recorre el árbol y devuelve todos los elementos con etiquetas que coincidan con la `String`. En este ejemplo, devuelve todos los párrafos que aparecen en `content`. El valor de retorno es un objeto Elements.

Antes de continuar, deberías dar una ojeada a la documentación de estas clases para que sepas lo que pueden hacer. Las clases más importantes son `Element`, `Elements`, y `Node`, sobre las que puedes leer en http://thinkdast.com/jsoupelt, http://thinkdast.com/jsoupelts y http://thinkdast.com/jsoupnode.

`Node` representa un nodo en el árbol DOM; hay varias subclases que extienden `Node`, incluyendo `Element`, `TextNode`, `DataNode` y `Comment`. `Elements` es una `Collection` de objetos `Element`.

La figura 6.3 es un diagrama UML que muestra las relaciones entre estas clases. En un diagrama de clases UML, una línea con una cabeza de flecha vacía indica que una clase extiende a la otra. Por ejemplo, este diagrama indica que `Elements` extiende `ArrayList`. Volveremos a los diagramas UML en la Sección 11.6.

## 6.4 Recorriendo el DOM

Para facilitarte la vida, proveo una clase llamada `WikiNodeIterable` que te permite recorrer los nodos en un árbol DOM. Aquí está un ejemplo que muestra cómo usarlo:

```
Elements paragraphs = content.select("p");
Element firstPara = paragraphs.get(0);

Iterable<Node> iter = new WikiNodeIterable(firstPara);
for (Node node: iter) {
    if (node instanceof TextNode) {
        System.out.print(node);
    }
}
```

Este ejemplo continúa donde finalizó el anterior. Selecciona el primer párrafo en `paragraphs` y luego crea un `WikiNodeIterable`, que implementa `Iterable<Node>`. `WikiNodeIterable` realiza una "búsqueda en profundidad", que produce los nodos en el orden que aparecerían en la página.

En este ejemplo, imprimimos un `Node` solo si es un `TextNode` e ignoramos otros tipos de `Node`, específicamente los objetos `Element` que representan etiquetas. El resultado es el texto plano del párrafo HTML sin ninguna marca. La salida es:

> Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented,[13] and specifically designed ...

## 6.5 Búsqueda en profundidad

Hay varias maneras en las que podrías razonablemente recorrer un árbol, cada una con diferentes aplicaciones. Comenzaremos con "búsqueda en profundidad", o DFS (por sus siglas en inglés, *depth-first search*). DFS comienza en la raíz del árbol y selecciona el primer hijo. Si el hijo tiene hijos, selecciona el primer hijo de nuevo. Cuando llega a un nodo sin hijos, se regresa, moviéndose hacia arriba en el árbol al nodo padre, donde selecciona al siguiente hijo si hay uno; de otra manera se regresa de nuevo. Cuando ha explorado el último hijo de la raíz, finaliza.

Hay dos formas comunes de implementar DFS, recursivamente e iterativamente. La implementación recursiva es simple y elegante:

```
private static void recursiveDFS(Node node) {
    if (node instanceof TextNode) {
        System.out.print(node);
    }
    for (Node child: node.childNodes()) {
        recursiveDFS(child);
    }
}
```

Este método se invoca en cada `Node` en el árbol, comenzando con la raíz. Si el `Node` que se devuelve es un `TextNode`, se imprimen sus contenidos. Si el `Node` tiene hijos, se invoca `recursiveDFS` en cada uno de ellos en orden.

En este ejemplo, imprimimos los contenidos de cada `TextNode` antes de recorrer los hijos, así que este es un ejemplo de un recorrido "pre orden". Puedes leer sobre los recorridos "pre orden", "post orden" y "en orden" en http://thinkdast.com/treetrav. Para esta aplicación el orden del recorrido no importa.

Al realizar llamadas recursivas, `recursiveDFS` usa la pila de llamadas (http://thinkdast.com/callstack) para llevar el control de los nodos hijos y procesarlos en el orden correcto. Como una alternativa, podemos usar una estructura de datos pila para llevar el control de los nodos nosotros mismos; si hacemos eso, podemos evitar la recursión y recorrer el árbol iterativamente.

## 6.6 Pilas (Stacks) en Java

Antes de explicar la versión iterativa de DFS, explicaré la estructura de datos pila. Comenzaremos con el concepto general de una pila, a la que llamaré una "pila" en minúsculas y en español. Luego hablaremos sobre dos `interfaces` de Java que definen los métodos de una pila: `Stack` y `Deque`.

Una pila es una estructura de datos que es similar a una lista: es una colección que mantiene el orden de los elementos. La principal diferencia entre una pila y una lista es que una pila provee menos métodos. En la convención usual, provee:

- `push`: que agrega un elemento a la parte superior de la pila.

- `pop`: que remueve y devuelve el elemento de la parte superior de la pila.
- `peek`: que devuelve el elemento de la parte superior sin modificar la pila.
- `isEmpty`: que indica si la pila está vacía.

Porque `pop` siempre devuelve el elemento de la parte superior, una pila también es conocida como una "UEPS", que significa "ultimo en entrar, primero en salir". Una alternativa a una pila es una "cola", que devuelve elementos en el mismo orden que son añadidos, es decir, "primero en entrar, primero en salir" o PEPS.

Podría no ser obvio por qué las pilas y las colas son útiles: no proveen ninguna operación que no provean las listas; de hecho proveen menos operaciones. Así que, ¿por qué no usar las listas para todo? Hay dos razones:

1. Si te limitas tú mismo a un pequeño conjunto de métodos — es decir, una API pequeña — tu código será más legible y menos propenso a errores. Por ejemplo, si usas una lista para representar una pila, podrías accidentalmente remover un elemento en el orden incorrecto. Con la API pila, esta clase de error es literalmente imposible. Y la mejor manera para evitar errores es hacerlos imposibles.

2. Si una estructura de datos provee una API pequeña, es más fácil de implementar eficientemente. Por ejemplo, una manera simple de implementar una pila es con una lista de enlace sencillo. Cuando agregamos (push) un elemento a la pila, lo agregamos al principio de la lista; cuando removemos (pop) un elemento, lo removemos del principio. Para una lista enlazada, agregar y remover desde el principio son operaciones de tiempo constante, así que esta implementación es eficiente. De forma similar, las APIs grandes son más difíciles de implementar eficientemente.

Para implementar una pila en Java, tienes tres opciones:

1. Adelante, usa `ArrayList` o `LinkedList`. Si usas `ArrayList`, asegúrate de agregar y remover desde el *final*, que es una operación de tiempo constante. Y ten el cuidado de no agregar elementos en el lugar equivocado o removerlos en el orden equivocado,

2. Java provee una clase llamada `Stack` que provee el conjunto estándar de métodos de una pila. Pero esta clase es una parte antigua de Java: no es consistente con el Java Collections Framework, que vino después.

3. Probablemente la mejor opción es usar uno de las implementaciones de la interfaz `Deque`, como `ArrayDeque`.

"Deque" significa "double-ended queue" ("cola con dos extremos"); se supone que se pronuncia "deck", pero algunos dicen "deek". En Java, la interfaz `Deque` provee `push`, `pop`, `peek`, y `isEmpty`, así que puedes usar una `Deque` como una pila. Provee otros métodos, sobre los que puedes leer en http://thinkdast.com/deque, pero no los usaremos por ahora.

## 6.7 DFS Iterativa

Aquí está una versión iterativa de DFS que usa una `ArrayDeque` para representar una pila de objetos `Node`:

```
private static void iterativeDFS(Node root) {
    Deque<Node> stack = new ArrayDeque<Node>();
    stack.push(root);

    while (!stack.isEmpty()) {
        Node node = stack.pop();
        if (node instanceof TextNode) {
            System.out.print(node);
        }

        List<Node> nodes = new ArrayList<Node>(node.childNodes());
        Collections.reverse(nodes);

        for (Node child: nodes) {
            stack.push(child);
        }
    }
}
```

El parámetro, `root`, es la raíz del árbol que queremos recorrer, así que comenzamos por crear la pila y agregar la raíz en ella.

El bucle continúa hasta que la pila está vacía. En cada repetición, se remueve un `Node` de la pila. Si se obtiene un `TextNode`, se imprimen los contenidos. Entonces se agregan a sus hijos en la pila. Para procesar a los hijos en el orden correcto, tenemos que agregarlos al stack en orden inverso; hacemos eso al copiar los hijos en una `ArrayList`, invertimos los elementos en sitio y luego iteramos a través de la `ArrayList` invertida.

Una ventaja de la versión iterativa de DFS es que es más fácil de implementar como un `Iterator` de Java; verás cómo en el siguiente capítulo.

Pero antes, un último comentario sobre la interfaz `Deque`: además de `ArrayDeque`, Java provee otra implementación de `Deque`, nuestra vieja amiga `LinkedList`. `LinkedList` implementa ambas interfaces, `List` y `Deque`. La interfaz que obtendrás depende de cómo la uses. Por ejemplo, si asignas un objeto `LinkedList` a una variable `Deque`, así:

```
Deqeue<Node> deque = new LinkedList<Node>();
```

puedes usar los métodos de la interfaz `Deque`, pero ningún método de la interfaz `List`. Si asignas una variable `List`, como esta:

```
List<Node> deque = new LinkedList<Node>();
```

puedes usar los métodos de `List` pero ninguno de los métodos de `Deque`. Y si la asignas así:

```
LinkedList<Node> deque = new LinkedList<Node>();
```

puedes usar *todos* los métodos. Pero si combinas métodos de interfaces diferentes, tu código será menos legible y más propenso a errores.

# Capítulo 7

# Llegar a la Filosofía

La meta de este capítulo es desarrollar un rastreador Web para probar la conjetura "Llegar a la Filosofía", que presentamos en la Sección 6.1.

## 7.1 Introducción

En el repositorio para este libro, encontrarás código base para ayudarte a empezar:

1. `WikiNodeExample.java` contiene el código del capítulo anterior, demostrando implementaciones recursivas e iterativas de la búsqueda en profundidad (DFS) en un árbol DOM.

2. `WikiNodeIterable.java` contiene una clase `Iterable` para recorrer un árbol DOM: Explicaré este código en la siguiente sección.

3. `WikiFetcher.java` contiene una clase utilitaria que usa jsoup para descargar páginas de Wikipedia. Para ayudarte a cumplir con los términos del servicio de Wikipedia, esta clase limita qué tan rápido puedes descargar páginas; si solicitas más de una página por segundo, se duerme antes de descargar la siguiente página.

4. `WikiPhilosophy.java` contiene un boceto del código que escribirás para este ejercicio. Lo revisaremos en breve.

También encontrarás el archivo de construcción Ant `build.xml`. Si escribes `ant WikiPhilosophy`, se ejecutará un fragmento de código inicial.

## 7.2 Iterables e Iterators

En el capítulo anterior, presenté una búsqueda en profundidad iterativa (DFS), y sugerí que una ventaja de la versión iterativa, comparada a la versión recursiva, es que es más fácil de envolver en un objeto `Iterator`. En esta sección, veremos cómo hacer eso.

Si no estás familiarizado con las interfaces `Iterator` y `Iterable`, puedes leer sobre ellas en http://thinkdast.com/iterator y http://thinkdast.com/iterable.

Observa los contenidos de `WikiNodeIterable.java`. La clase más externa, `WikiNodeIterable` implementa la interfaz `Iterable<Node>` para que podamos usarla en un bucle for de la siguiente forma:

```
Node root = ...
Iterable<Node> iter = new WikiNodeIterable(root);
for (Node node: iter) {
    visit(node);
}
```

donde `root` es la raíz del árbol que queremos recorrer y `visit` es un método que hace lo que queramos cuando "visitamos" un `Node`.

La implementación de `WikiNodeIterable` sigue una fórmula convencional:

1. El constructor toma y guarda una referencia al `Node` raíz.
2. El método `iterator` crea y devuelve un objeto `Iterator`.

Así es como se ve:

```
public class WikiNodeIterable implements Iterable<Node> {

    private Node root;

    public WikiNodeIterable(Node root) {
        this.root = root;
    }
```

```
    @Override
    public Iterator<Node> iterator() {
        return new WikiNodeIterator(root);
    }
}
```

La clase más interna, `WikiNodeIterator`, hace todo el trabajo:

```
    private class WikiNodeIterator implements Iterator<Node> {

        Deque<Node> stack;

        public WikiNodeIterator(Node node) {
            stack = new ArrayDeque<Node>();
            stack.push(root);
        }

        @Override
        public boolean hasNext() {
            return !stack.isEmpty();
        }

        @Override
        public Node next() {
            if (stack.isEmpty()) {
                throw new NoSuchElementException();
            }

            Node node = stack.pop();
            List<Node> nodes = new ArrayList<Node>(node.childNodes());
            Collections.reverse(nodes);
            for (Node child: nodes) {
                stack.push(child);
            }
            return node;
        }
    }
```

Este código es casi idéntico a la versión iterativa de DFS, pero ahora se divide en tres métodos:

1. El constructor inicializa la pila (que se implementa usando una `ArrayDeque`) y agrega el nodo raíz a dicha pila.

2. `isEmpty` comprueba si la pila está vacía.

3. `next` remueve (pop) el siguiente `Node` de la pila, agrega (push) sus hijos en orden inverso y devuelve el `Node` que removió. Si alguien invoca `next` en un `Iterator` vacío, lanza una excepción.

Podría no ser obvio que vale la pena reescribir un método perfectamente aceptable con dos clases y cinco métodos. Pero ahora que lo hemos hecho, podemos usar `WikiNodeIterable` en cualquier parte que se llame a un `Iterable`, lo que nos hace más fácil y es más limpio desde un punto de vista sintáctico, separar la lógica de iteración (DFS) de cualquier otro proceso que estemos haciendo en los nodos.

## 7.3 `WikiFetcher`

Cuando escribes un rastreador Web, es fácil descargar muchas páginas muy rápido, lo que podría violar los términos del servicio para el servidor del que estás realizando las descargas. Para evitar esto, proveo una clase llamada `WikiFetcher` que hace dos cosas:

1. Encapsula el código que mostramos en el capítulo previo para descargar páginas de Wikipedia, interpretar el HTML, y seleccionar el texto del contenido.

2. Mide el tiempo entre solicitudes y, si no hemos dejado pasar suficiente tiempo entre solicitudes, duerme hasta que ha pasado un intervalo razonable. Por defecto, el intervalo es un segundo.

Aquí está la definición de `WikiFetcher`:

```
public class WikiFetcher {
    private long lastRequestTime = -1;
    private long minInterval = 1000;

    /**
```

```
     * Recupera e interpreta una cadena URL,
     * devuelve una lista de elementos del párrafo.
     *
     * @param url
     * @return
     * @throws IOException
     */
    public Elements fetchWikipedia(String url) throws IOException {
        sleepIfNeeded();

        Connection conn = Jsoup.connect(url);
        Document doc = conn.get();
        Element content = doc.getElementById("mw-content-text");
        Elements paragraphs = content.select("p");
        return paragraphs;
    }

    private void sleepIfNeeded() {
        if (lastRequestTime != -1) {
            long currentTime = System.currentTimeMillis();
            long nextRequestTime = lastRequestTime + minInterval;
            if (currentTime < nextRequestTime) {
                try {
                    Thread.sleep(nextRequestTime - currentTime);
                } catch (InterruptedException e) {
                    System.err.println(
                        "Warning: sleep interrupted in fetchWikipedia.");
                }
            }
        }
        lastRequestTime = System.currentTimeMillis();
    }
}
```

El único método público es `fetchWikipedia`, que toma una URL como una `String` y devuelve una colección `Elements` que contiene un elemento del DOM para cada párrafo en el texto del contenido. Este código debería resultarle familiar.

El nuevo código está es `sleepIfNeeded`, que comprueba el tiempo desde la última solicitud y se duerme si el tiempo transcurrido es menor que `minInterval`, que está en milisegundos.

Eso es todo en `WikiFetcher`. Aquí está un ejemplo que demuestra su uso:

```
WikiFetcher wf = new WikiFetcher();

for (String url: urlList) {
    Elements paragraphs = wf.fetchWikipedia(url);
    processParagraphs(paragraphs);
}
```

En este ejemplo, asumimos que `urlList` es una colección de `String`s, y `processParagraphs` es un método que hace algo con el objeto `Elements` que devuelve `fetchWikipedia`.

Este ejemplo demuestra algo importante: deberías crear un objeto `WikiFetcher` y usarlo para manejar todas las solicitudes. Si tienes múltiples instancias de `WikiFetcher`, ellas no forzarán el intervalo mínimo entre solicitudes.

NOTA: Mi implementación de `WikiFetcher` es simple, pero sería fácil para alguien utilizarla incorrectamente al crear múltiples instancias. Puedes evitar este problema haciendo a `WikiFetcher` un "singleton", sobre el que puedes leer más en http://thinkdast.com/singleton.

## 7.4 Ejercicio 5

En `WikiPhilosophy.java` encontrarás un método `main` simple que muestra cómo usar algunas de estos fragmentos. Comenzando con este código, tu trabajo es escribir un rastreador que:

1. Tome una URL para una página de Wikipedia, la descargue y la interprete.

2. Debería recorrer el árbol resultante para encontrar el primer enlace *válido*. Explicaré lo que significa "válido" a continuación.

3. Si la página no tiene enlaces, o si el primer enlace es una página que ya hemos visto, el programa debería indicar que fracasó y salir.

4. Si el enlace coincide con la URL de la página de Wikipedia sobre filosofía, el programa debería indicar que tuvo éxito y salir.

5. De otra manera debería regresar al Paso 1.

El programa debería construir una `List` de las URL que visita y mostrar los resultados al final (independientemente de si es exitoso o fracasa).

Entonces, ¿qué deberíamos considerar un enlace"válido"? Tienes algunas opciones. Varias versiones de la conjetura "Llegar a la Filosofía" usan reglas ligeramente diferentes, pero aquí están algunas opciones:

1. El enlace debería estar en el texto del contenido de la página, no en una barra lateral o en un recuadro.

2. No debería estar en cursiva o entre paréntesis.

3. Deberías omitir enlaces externos, enlaces a la página actual y enlaces en rojo.

4. En algunas versiones, deberías omitir un enlace si el texto comienza con una letra mayúscula.

No tienes que forzar todas estas reglas, pero te recomendamos que al menos consideres los paréntesis, las cursivas y los enlaces a la página actual.

Si sientes que tienes información para empezar, adelante. O podrías querer leer estas pistas:

1. Conforme recorras el árbol, las dos clases de `Node` que necesitarás manejar son `TextNode` y `Element`. Si encuentras un `Element`, probablemente tendrás que forzar su conversión a otro tipo para acceder a la etiqueta y otra información.

2. Cuando encuentres un `Element` que contiene un enlace, puedes comprobar si está en cursiva siguiendo los enlaces hacia su padre en el árbol. Si hay una etiqueta `<i>` o `<em>` en la cadena de nodos padres, el enlace está en cursiva.

3. Para comprobar si un enlace está entre paréntesis, tendrás que escanear el texto conforme recorras el árbol y llevar el control de los paréntesis de apertura y de cierre (idealmente tu solución debería ser capaz de manejar paréntesis anidados (como estos)).

4. Si comienzas desde la página de Java, deberías llegar a la Filosofía después de seguir siete enlaces, a menos que algo haya cambiado desde que ejecuté el código.

Muy bien, esa es toda la ayuda que te daré. Ahora te toca a ti. ¡Que te diviertas!

# Capítulo 8

# Indexador

Llegados a este punto hemos construido un rastreador Web básico; la siguiente pieza en la que trabajaremos será el **índice**. En el contexto de una búsqueda web, un índice es una estructura de datos que hace posible explorar un término de búsqueda y encontrar las páginas donde este término aparece. Además, nos gustaría saber cuántas veces aparece el término de búsqueda en cada página, lo que nos ayudará a identificar las páginas más relevantes para el término.

Por ejemplo, si un usuario envía los términos de búsqueda "programación" y "Java", deberíamos buscar ambos términos y obtener dos conjuntos de páginas. Las páginas con la palabra "programación" incluirían páginas sobre diferentes lenguajes de programación, así como otros usos de la palabra. Las páginas con la palabra "Java" incluirían páginas sobre la isla de Java, el término en inglés para un café y el lenguaje de programación. Al seleccionar páginas con ambos términos, esperamos eliminar páginas irrelevantes y encontrar las que tratan sobre programación en Java.

Ahora que entendemos lo que es un índice y las operaciones que realiza, podemos diseñar una estructura de datos para representarlo.

## 8.1 Selección de una estructura de datos

La operación fundamental del índice es una **búsqueda**; específicamente, necesitamos la habilidad de buscar un término y encontrar todas las páginas que

lo contengan. La implementación más simple sería una colección de páginas. Dado un término de búsqueda, podríamos iterar a través de los contenidos de las páginas y seleccionar las que contengan el término de búsqueda Pero el tiempo de ejecución sería proporcional al número total de palabras en todas las páginas, lo que sería súper lento.

Una mejor alternativa es un **mapa**, que es una estructura de datos que representa una colección de **pares clave-valor** y provee una forma rápida de buscar una **clave** y encontrar el **valor** correspondiente. Por ejemplo, el primer mapa que construiremos es un `TermCounter` (contador de términos), que mapea cada término de búsqueda al número de veces que aparece en una página. Las claves son los términos de búsqueda y los valores son los conteos (también llamados "frecuencias").

Java provee una interfaz llamada `Map` que especifica los métodos que un mapa debe proveer; los más importantes son:

- `get(key)`: Este método busca una clave (key) y devuelve el valor (value) correspondiente.
- `put(key, value)`: Este método agrega un nuevo par clave-valor al `Map`, o si la clave ya está en el mapa, reemplaza el valor asociado con `key`.

Java provee varias implementaciones de `Map`, incluyendo las dos en la que nos enfocaremos, `HashMap` y `TreeMap`. En capítulos posteriores, examinaremos estas implementaciones y analizaremos su desempeño.

Además del `TermCounter`, que mapea términos de búsqueda a conteos, definiremos una clase llamada `Index`, que mapea un término de búsqueda con una colección de las páginas donde aparece. Y eso hace surgir la siguiente pregunta, que es cómo representar una colección de páginas. De nuevo, si pensamos en las operaciones que queremos realizar, eso guía nuestra decisión.

En este caso, necesitaremos combinar dos o más colecciones y encontrar las páginas que aparecen en todas ellas. Podrías reconocer esta operación como la **intersección de conjuntos**: la intersección de dos conjuntos es el conjunto de elementos que aparecen en ambos.

Como podrías esperar, Java provee una interfaz `Set` (Conjunto, en inglés) que define las operaciones que un conjunto debería realizar. No provee realmente

una intersección de conjuntos, pero provee métodos que hacen posible implementar la intersección y otras operaciones de conjuntos eficientemente. Los métodos principales de un `Set` son:

- `add(element)`: Este método añade un elemento al conjunto; si el elemento ya está en el conjunto, no tiene ningún efecto.
- `contains(element)`: Este método comprueba si el elemento dado está en el conjunto.

Java provee varias implementaciones de `Set`, incluyendo `HashSet` y `TreeSet`.

Ahora que hemos diseñado nuestras estructuras de datos a nivel general, las implementaremos desde lo más básico, comenzando con `TermCounter`.

## 8.2 TermCounter

`TermCounter` es una clase que representa un mapeo de términos de búsqueda con el número de veces que aparecen en una página. Aquí está la primera parte de la definición de la clase:

```
public class TermCounter {

    private Map<String, Integer> map;
    private String label;

    public TermCounter(String label) {
        this.label = label;
        this.map = new HashMap<String, Integer>();
    }
}
```

Las variables de instancia son `map`, que contiene el mapeo de términos con conteos, y `label`, que identifica el documento del que provienen los términos; la usaremos para guardar URLs.

Para implementar el mapeo, elegí `HashMap`, que es el `Map` más comúnmente usado. En unos cuantos capítulos, verás cómo funciona y por qué es una elección común.

`TermCounter` provee `put` y `get`, que se definen como sigue:

```
public void put(String term, int count) {
    map.put(term, count);
}

public Integer get(String term) {
    Integer count = map.get(term);
    return count == null ? 0 : count;
}
```

`put` es solo un **método envoltorio (wrapper)**; cuando llamas `put` en un `TermCounter`, éste llama a `put` en el mapa embebido.

Por otro lado, `get` de hecho hace un poco de trabajo. Cuando lo llamas a `get` en un `TermCounter`, llama a `get` en el mapa y comprueba el resultado. Si el término no aparece en el mapa,`TermCount.get` devuelve 0. Al definir `get` de esta forma es más fácil escribir `incrementTermCount`, que toma un término e incrementa en uno el contador asociado con ese término.

```
public void incrementTermCount(String term) {
    put(term, get(term) + 1);
}
```

Si el término no ha aparecido antes, `get` devuelve 0; le sumamos 1, y luego usamos `put` para agregar un nuevo par clave-valor al mapa. Si el término ya existe en el mapa, obtenemos el conteo anterior, le sumamos 1 y luego guardamos el nuevo conteo, que reemplaza al valor antiguo.

Además, `TermCounter` provee estos otros métodos para ayudar con la indexación de páginas Web:

```
public void processElements(Elements paragraphs) {
    for (Node node: paragraphs) {
        processTree(node);
    }
}

public void processTree(Node root) {
    for (Node node: new WikiNodeIterable(root)) {
        if (node instanceof TextNode) {
            processText(((TextNode) node).text());
```

```
        }
    }
}

public void processText(String text) {
    String[] array = text.replaceAll("\\pP", " ").
                          toLowerCase().
                          split("\\s+");

    for (int i=0; i<array.length; i++) {
        String term = array[i];
        incrementTermCount(term);
    }
}
```

- `processElements` toma un objeto `Elements`, que es una colección de objetos `Element` de jsoup. Itera por la colección y llama a `processTree` para cada uno.
- `processTree` toma un `Node` de jsoup que representa la raíz de un árbol DOM. Itera por el árbol para encontrar los nodos que contienen texto; luego extrae el texto y lo pasa a `processText`.
- `processText` toma una `String` que contiene palabras, espacios, signos de puntuación, etc. Remueve los signos de puntuación reemplazándolos con espacios, convierte las letras restantes a minúsculas, luego las separa en palabras. A continuación, itera por las palabras que encontró y llama a `incrementTermCount` con cada una. Los métodos `replaceAll` y `split` toman **expresiones regulares** como parámetros; puedes leer más sobre ellas en http://thinkdast.com/regex.

Finalmente, aquí está un ejemplo que demuestra cómo se usa `TermCounter`:

```
String url = "http://en.wikipedia.org/wiki/Java_(programming_language)";
WikiFetcher wf = new WikiFetcher();
Elements paragraphs = wf.fetchWikipedia(url);

TermCounter counter = new TermCounter(url);
counter.processElements(paragraphs);
counter.printCounts();
```

Este ejemplo usa un `WikiFetcher` para descargar una página de Wikipedia e interpretar el texto principal. Luego crea un `TermCounter` y lo usa para contar las palabras en la página.

En la siguiente sección, tendrás una oportunidad de ejecutar este código y probar tu comprensión al llenar un método faltante.

## 8.3 Ejercicio 6

En el repositorio para este libro, encontrarás los archivos de código fuente para este ejercicio:

- `TermCounter.java` contiene el código de la sección anterior.
- `TermCounterTest.java` contiene código de prueba para `TermCounter.java`.
- `Index.java` contiene la definición de la clase para la siguiente parte de este ejercicio.
- `WikiFetcher.java` contiene la clase que usamos en el ejercicio anterior para descargar e interpretar páginas Web.
- `WikiNodeIterable.java` contiene la clase que usamos para recorrer los nodos en un árbol DOM.

También encontrarás el archivo de construcción Ant `build.xml`.

Ejecuta `ant build` para compilar los archivos de código fuente. Luego ejecuta `ant TermCounter`; debería ejecutar el código de la sección anterior e imprimir una lista de términos y sus conteos. La salida debería verse algo así:

```
genericservlet, 2
configurations, 1
claimed, 1
servletresponse, 2
occur, 2
Total of all counts = -1
```

Cuando lo ejecutes, el orden de los términos podría ser diferente.

Se supone que la última línea imprima el total de los conteos de términos, pero devuelve `-1` porque el método `size` está incompleto. Llena este método y ejecuta `ant TermCounter` de nuevo. El resultado debería ser `4798`.

Ejecuta `ant TermCounterTest` para confirmar que esta parte del ejercicio está completa y correcta.

Para la segunda parte del ejercicio, presentaré una implementación de un objeto `Index` y llenarás un método faltante. Aquí está el principio de la definición de la clase:

```
public class Index {

    private Map<String, Set<TermCounter>> index =
        new HashMap<String, Set<TermCounter>>();

    public void add(String term, TermCounter tc) {
        Set<TermCounter> set = get(term);

        // si vemos un término por primera vez, creamos un nuevo Set
        if (set == null) {
            set = new HashSet<TermCounter>();
            index.put(term, set);
        }
        // de lo contrario modificamos un Set existente
        set.add(tc);
    }

    public Set<TermCounter> get(String term) {
        return index.get(term);
    }
```

La variable de instancia, `index`, es un mapa de cada término de búsqueda con un conjunto de objetos `TermCounter`. Cada `TermCounter` representa una página donde el término de búsqueda aparece.

El método `add` agrega un nuevo `TermCounter` al conjunto (set) asociado con un término. Cuando indexamos un término que no había aparecido antes,

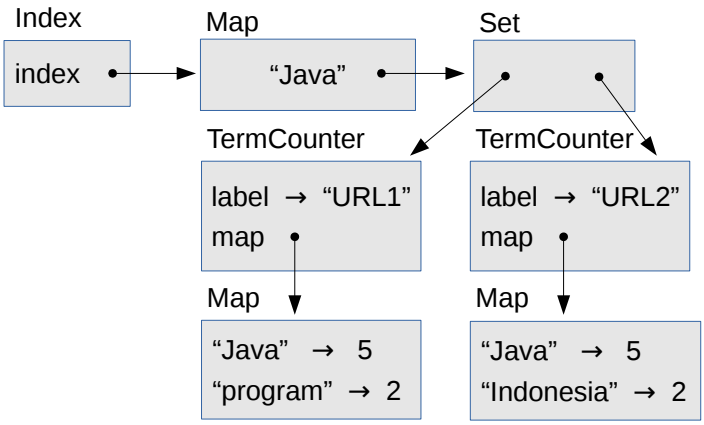


Figura 8.1: Diagrama de objeto de un `Index`.

tenemos que crear un nuevo conjunto. De lo contrario, podemos simplemente agregar un nuevo elemento a un set existente. En ese caso, `set.add` modifica un set que vive dentro de un `index`, pero no modifica el `index` en sí mismo. La única vez que modificamos `index` es cuando agregamos un nuevo término.

Finalmente, el método `get` toma un término de búsqueda y devuelve el set correspondiente de objetos `TermCounter`.

Esta estructura de datos es moderadamente más complicada. Para repasar, un `Index` contiene un `Map` de cada término de búsqueda con un `Set` de objetos `TermCounter`, y cada `TermCounter` es un mapa de términos de búsqueda a conteos.

La figura 8.1 es un diagrama de objeto que muestra estos objetos. El objeto `Index` tiene una variable de instancia llamada `index` que se refiere a un `Map`. Es este ejemplo el `Map` contiene sólo una cadena, `"Java"`, que se mapea a un `Set` que contiene dos objetos `TermCounter`, uno para cada página donde la palabra “Java” aparece.

Cada `TermCounter` contiene `label`, que es la URL de la página y `map`, que es un `Map` que contiene las palabras en la página y el número de veces que cada palabra aparece.

El método `printIndex` muestra cómo desempaquetar esta estructura de datos:

```
    public void printIndex() {
        // itera por los términos de búsqueda
        for (String term: keySet()) {
            System.out.println(term);

            // para cada término, imprime las páginas en
            // las que aparece y sus frecuencias
            Set<TermCounter> tcs = get(term);
            for (TermCounter tc: tcs) {
                Integer count = tc.get(term);
                System.out.println("    " + tc.getLabel() + " " + count);
            }
        }
    }
```

El bucle externo itera por los términos de búsqueda. El bucle interno itera por los objetos `TermCounter`.

Ejecuta `ant build` para asegurarte que tu código fuente está compilado y entonces ejecuta `ant Index`. Éste descarga dos páginas de Wikipedia, las index e imprime los resultados; pero cuando lo ejecutes no verás ninguna salida porque hemos dejado vacío uno de los métodos.

Tu trabajo es llenar `indexPage`, que toma una URL (como una `String`) y un objeto `Elements`, y actualiza el índice. Los comentarios siguientes dicen a grandes rasgos lo que debería hacer:

```
public void indexPage(String url, Elements paragraphs) {
    // crea un TermCounter y cuenta los términos en los párrafos

    // para cada término en el TermCounter, agrega el TermCounter al index
}
```

Cuando funcione, ejecuta `ant Index` de nuevo, y deberías ver una salida como esta:

```
...
configurations
    http://en.wikipedia.org/wiki/Programming_language 1
    http://en.wikipedia.org/wiki/Java_(programming_language) 1
```

```
claimed
    http://en.wikipedia.org/wiki/Java_(programming_language) 1
servletresponse
    http://en.wikipedia.org/wiki/Java_(programming_language) 2
occur
    http://en.wikipedia.org/wiki/Java_(programming_language) 2
```

El orden de los términos de búsqueda podría ser diferente cuando lo ejecutes.

También, ejecuta `ant TestIndex` para confirmar que esta parte del ejercicio está completa.

# Capítulo 9

# La interfaz Map

En los siguientes ejercicios, presento varias implementaciones de la interfaz `Map`. Una de ellas está basada en una **tabla hash (hash table)**, que posiblemente es la estructura de datos más mágica alguna vez inventada. Otra, que es similar a `TreeMap`, no es tan mágica, pero tiene la capacidad adicional de permitir iterar por los elementos en orden.

Tendrás una oportunidad de implementar estas estructuras de datos, y entonces analizaremos su desempeño.

Pero antes de que podamos explicar las tablas hash, empezaremos con una implementación simple de un `Map` usando una `List` de pares clave-valor.

## 9.1 Implementación de `MyLinearMap`

Como es usual, proveo un código para empezar y llenarás los métodos faltantes. Aquí está el principio de la definición de la clase `MyLinearMap`:

```
public class MyLinearMap<K, V> implements Map<K, V> {

    private List<Entry> entries = new ArrayList<Entry>();
```

Esta clase usa dos parámetros de tipo, `K`, que es el tipo de las claves, y `V`, que es el tipo de los valores. `MyLinearMap` implementa `Map`, lo que significa que tiene que proveer los métodos de la interfaz `Map`.

Un objeto `MyLinearMap` tiene una única variable de instancia, `entries`, que es una `ArrayList` de objetos `Entry`. Cada `Entry` contiene un par clave-valor. Aquí está la definición:

```
public class Entry implements Map.Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public K getKey() {
        return key;
    }
    @Override
    public V getValue() {
        return value;
    }
}
```

No hay mucho que decir; una `Entry` es solo un contenedor para una clave (key) y un valor (value). Esta definición esta anidada dentro de `MyLinearList`, así que usa los mismos parámetros de tipo, `K` y `V`.

Eso es todo lo que necesitas para hacer el ejercicio, así que comencemos.

## 9.2 Ejercicio 7

En el repositorio para este libro, encontrarás los archivos de código fuente para este ejercicio:

- `MyLinearMap.java` contiene código inicial para la primera parte del ejercicio.
- `MyLinearMapTest.java` contiene los tests unitarios para `MyLinearMap`.

También encontrarás el archivo de construcción Ant `build.xml`.

Ejecuta `ant build` para compilar los archivos fuente. Luego ejecuta `ant MyLinearMapTest`; varios tests fallarán, ¡porque aún tienes trabajo por hacer!

Primero, llena el cuerpo de `findEntry`. Este es un método auxiliar que no es parte de la interfaz `Map`, pero una vez consigas que funcione, puedes usarlo para varios métodos. Dada una clave objetivo, debería buscar a lo largo de las entradas (entries) y devolver la entrada que contenga el objetivo (como una clave, no un valor) o `null` si no está ahí. Nota que proveo un método `equals` que compara dos claves y maneja los `null` correctamente.

Puedes ejecutar `ant MyLinearMapTest` de nuevo, pero incluso si tu `findEntry` es correo, los tests no pasarán porque `put` no está completo.

Llena `put`. Deberías leer la documentación de `Map.put` en http://thinkdast.com/listput para que sepas lo que se supone que haga. Podrías querer iniciar con una versión de `put` que siempre agregue una nueva entrada y no modifique una entrada existente; de esa forma puedes probar el caso más simple primero. O si te sientes con más confianza, puedes escribir todo el método de una vez.

Una vez logres que `put` funcione, el test para `containsKey` debería pasar.

Lee la documentación de `Map.get` en http://thinkdast.com/listget y luego llena el método. Ejecuta los tests de nuevo.

Finalmente, lee la documentación de `Map.remove` en http://thinkdast.com/maprem y llena el método.

Llegados a este punto, todos los tests deberían pasar. ¡Felicitaciones!

## 9.3 Análisis de MyLinearMap

En esta sección presento una solución al ejercicio anterior y analizo el desempeño de los métodos principales. Aquí están `findEntry` y `equals`:

```
private Entry findEntry(Object target) {
    for (Entry entry: entries) {
        if (equals(target, entry.getKey())) {
```

```
            return entry;
        }
    }
    return null;
}

private boolean equals(Object target, Object obj) {
    if (target == null) {
        return obj == null;
    }
    return target.equals(obj);
}
```

El tiempo de ejecución de `equals` podría depender del tamaño del `target` (objetivo) y las claves, pero generalmente no depende del número de entradas, $n$. Así que `equals` es de tiempo constante.

En `findEntry`, podríamos tener suerte y encontrar la clave que estamos buscando al principio, pero no podemos contar con ello. En general, el número de entradas que tenemos que buscar es proporcional a $n$, así que `findEntry` es lineal.

La mayoría de los métodos principales de `MyLinearMap` usan `findEntry`, incluyendo `put`, `get`, y `remove`. Así es como se ven:

```
public V put(K key, V value) {
    Entry entry = findEntry(key);
    if (entry == null) {
        entries.add(new Entry(key, value));
        return null;
    } else {
        V oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    }
}

public V get(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
```

```
        return null;
    }
    return entry.getValue();
}

public V remove(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
        return null;
    } else {
        V value = entry.getValue();
        entries.remove(entry);
        return value;
    }
}
```

Después que `put` llama a `findEntry`, todo lo demás es de tiempo constante. Recuerda que `entries` es una `ArrayList`, así que agregar un elemento *al final* es de tiempo constante, en promedio. Si la clave ya está en el mapa, no tenemos que agregar una entrada, pero tenemos que llamar a `entry.getValue` y `entry.setValue`, y ambos son de tiempo constante. Al combinar todo, `put` es lineal.

Siguiendo el mismo razonamiento, `get` también es lineal.

`remove` es ligeramente más complicado porque `entries.remove` podría tener que remover un elemento del principio o de la mita de la `ArrayList`, y eso toma tiempo lineal. Pero está bien: dos operaciones lineales siguen siendo lineales.

En resumen, los métodos principales son todos lineales, y es por eso que llamamos a esta implementación `MyLinearMap` (¡ta-da!).

Si sabemos que el número de entradas será pequeño, esta implementación podría ser lo suficientemente buena, pero podemos mejorarla. De hecho, hay una implementación de `Map` donde todos los métodos principales son de tiempo constante. Cuando escuchas eso por primera vez, podría parecer que no es posible. Lo que estamos diciendo, en efecto, es que puedes encontrar una aguja en un pajar en tiempo constante, sin importar el tamaño del pajar. Es magia.

Explicaré cómo funciona en dos pasos:

1. En lugar de guardar las entradas en una gran `List`, las dividiremos en muchas listas pequeñas. Para cada clave, usaremos un **código hash** (se explica en la siguiente sección) para determinar qué lista usar.

2. Usar muchas listas pequeñas es más rápido que usar solo una, pero como explicaré, no cambia el orden de crecimiento, las operaciones principales aún son lineales. Pero hay un truco adicional: si incrementamos el número de listas para limitar el número de entradas por lista, el resultado es un mapa de tiempo constante. Verás los detalles en el siguiente ejercicio, pero primero, ¡veamos qué es hashing!

En el siguiente capítulo, presentaré una solución, analizaré el desempeño de los métodos principales de `Map`, e introduciré una implementación más eficiente.

# Capítulo 10

# Hashing

En este capítulo, defino `MyBetterMap`, una mejor implementación de la interfaz `Map` que `MyLinearMap`, e introduzco el concepto de **hashing**, que hace a `MyBetterMap` más eficiente.

## 10.1 Hashing

Para mejorar el desempeño de `MyLinearMap`, escribiremos una nueva clase, llamada `MyBetterMap`, que contiene una colección de objetos `MyLinearMap`. Esta clase divide las claves entre los mapas embebidos, así que el número de entradas en cada mapa es más pequeño, lo que hace más rápido a `findEntry` y los métodos que dependen de él.

Aquí está el principio de la definición de la clase:

```
public class MyBetterMap<K, V> implements Map<K, V> {

    protected List<MyLinearMap<K, V>> maps;

    public MyBetterMap(int k) {
        makeMaps(k);
    }

    protected void makeMaps(int k) {
```

```
        maps = new ArrayList<MyLinearMap<K, V>>(k);
        for (int i=0; i<k; i++) {
            maps.add(new MyLinearMap<K, V>());
        }
    }
}
```

La variable de instancia, `maps`, es una colección de objetos `MyLinearMap`. El constructor toma un parámetro, `k`, que determina cuántos mapas usar, al menos inicialmente. Luego `makeMaps` crea los mapas embebidos y los guarda en una `ArrayList`.

Ahora, la clave para hacer que esto funcione es que necesitamos alguna forma de ver una clave y decidir en cuál de los mapas embebidos debería ir. Cuando ponemos (`put`) una nueva clave, elegimos una de los mapas; cuando obtenemos (`get`) la misma clave, tenemos que recordar dónde la pusimos.

Una posibilidad es elegir uno de los submapas al azar y llevar un el control de en dónde pusimos cada clave. ¿Pero cómo llevaríamos el control? Tal vez podríamos usar un `Map` para buscar la clave y encontrar el sub-mapa correcto, pero todo el punto del ejercicio es escribir una implementación eficiente de un `Map`. No podemos asumir que ya tenemos una.

Una mejor aproximación es usar una **función hash**, que toma un `Object`, cualquier `Object`, y devuelve un entero llamado **código hash**. Es importante que si esta función ve el mismo `Object` más de una vez, siempre devuelva el mismo código hash. De esa manera, si usamos el código hash para guardar una clave, obtendremos el mismo código hash cuando la busquemos.

En java, cada `Object` provee un método llamado `hashCode` que calcula una función hash. La implementación del método es diferente para diferentes objetos; veremos un ejemplo pronto.

Aquí está un método auxiliar que elige el sub-mapa correcto para una clave dada:

```
protected MyLinearMap<K, V> chooseMap(Object key) {
    int index = 0;
    if (key != null) {
        index = Math.abs(key.hashCode()) % maps.size();
```

```
    }
    return maps.get(index);
}
```

Si `key` es `null`, elegimos el sub-mapa con el índice 0, de forma arbitraria. De lo contrario usamos `hashCode` para obtener un entero, aplicamos `Math.abs` para asegurarnos que es no negativo, y luego usamos el operador de residuo, `%`, que nos garantiza que el resultado está entre 0 y `maps.size()-1`. Así que `index` siempre es un índice válido en `maps`. Luego `chooseMap` devuelve una referencia al mapa que eligió.

Usamos `chooseMap` tanto en `put` como en `get`, para que cuando busquemos una clave, obtengamos el mismo mapa que elegimos cuando agregamos la clave. Por lo menos, deberíamos — explicaré después por qué esto podría no funcionar.

Aquí está mi implementación de `put` y `get`:

```
public V put(K key, V value) {
  MyLinearMap<K, V> map = chooseMap(key);
    return map.put(key, value);
}

public V get(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.get(key);
}
```

Bastante simple, ¿verdad? En ambos métodos, usamos `chooseMap` para encontrar el sub-mapa correcto y luego invocamos un método en el sub-mapa. Así es como funciona, ahora pensemos en el desempeño.

Si hay $n$ entradas distribuidas entre $k$ sub-mapas, habrán $n/k$ entradas por mapa, en promedio. Cuando busquemos una clave, tendremos que calcular su código hash, lo que toma algún tiempo, entonces podemos buscar el sub-mapa correspondiente.

Porque las listas de entradas en `MyBetterMap` son $k$ veces más cortas que la lista de entradas en `MyLinearMap`, esperaríamos que la búsqueda sea $k$ veces más rápida. Pero el tiempo de ejecución aún es proporcional a $n$, así que `MyBetterMap` todavía es lineal. En el siguiente ejercicio, verás cómo podemos arreglar eso.

## 10.2 ¿Cómo funciona el hashing?

El requisito fundamental para una función hash es que el mismo objeto debería producir el mismo código hash cada vez. Para objetos inmutables, eso es relativamente sencillo. Para objetos con un estado mutable, tenemos que pensar un poco más.

Como un ejemplo de un objeto inmutable, definiré una clase llamada `SillyString` que encapsula a una `String`:

```
public class SillyString {
    private final String innerString;

    public SillyString(String innerString) {
        this.innerString = innerString;
    }

    public String toString() {
        return innerString;
    }
```

Esta clase no es muy útil, y es por eso que se llama `SillyString` (cadena tonta), pero la usaré para mostrar cómo una clase puede definir su propia función hash:

```
    @Override
    public boolean equals(Object other) {
        return this.toString().equals(other.toString());
    }

    @Override
    public int hashCode() {
        int total = 0;
        for (int i=0; i<innerString.length(); i++) {
            total += innerString.charAt(i);
        }
        return total;
    }
```

Nota que `SillyString` sobrescribe tanto `equals` como `hashCode`. Esto es importante. Para funcionar correctamente, `equals` tiene que ser consistente con `hashCode`, lo que significa que si dos objetos son considerados iguales — es decir, `equals` devuelve `true` — deberían tener el mismo código hash. Pero este requerimiento sólo funciona en una vía; si dos objetos tienen el mismo hash code, no necesariamente tienen que ser iguales.

`equals` funciona invocando `toString`, que devuelve `innerString`. Así que dos objetos `SillyString` son iguales si sus variables de instancia `innerString` son iguales.

`hashCode` funciona iterando por los caracteres de la cadena (`String`) y sumándolos. Cuando sumas un carácter a un `int` Java convierte el carácter a un entero usando su punto de código Unicode. No necesitas saber nada sobre Unicode para entender este ejemplo, pero si sientes curiosidad, puedes leer más en http://thinkdast.com/codepoint.

Esta función hash satisface el requisito: si dos objetos `SillyString` contienen cadenas embebidas que son iguales, obtendrán el mismo código hash.

Esto funciona correctamente, pero podría no tener un buen desempeño, porque devuelve el mismo código hash para muchas cadenas diferentes. Si dos cadenas contienen las mismas letras en cualquier orden, tendrán el mismo código hash. E incluso si no contienen las mismas letras, podrían producir el mismo total, como `"ac"` y `"bb"`.

Si muchos objetos tienen el mismo código hash, terminan en el mismo sub-mapa. Si algunos sub-mapas tienen más entradas que otros, el incremento en velocidad cuando tenemos $k$ mapas podría ser mucho menor que $k$. Así que una de las metas de una función hash es ser uniforme; es decir, debería ser igualmente probable que producirá un valor dentro del rango. Puedes leer más sobre el diseño de buenas funciones hash en http://thinkdast.com/hash.

## 10.3 Hashing y mutación

Las cadenas (`String`s) son inmutables y `SillyString` también es inmutable porque `innerString` se declara como `final`. Una vez crees una `SillyString`, no puedes hacer que `innerString` haga referencia a una `String` diferente, y

no puedes modificar la `String` a la que se refiere. Por lo tanto, siempre tendrá el mismo código hash.

Pero veamos lo que ocurre con un objeto mutable. Aquí está una definición para `SillyArray`, que es idéntica a `SillyString`, excepto que usa un arreglo de caracteres en lugar de una `String`:

```
public class SillyArray {
    private final char[] array;

    public SillyArray(char[] array) {
        this.array = array;
    }

    public String toString() {
        return Arrays.toString(array);
    }

    @Override
    public boolean equals(Object other) {
        return this.toString().equals(other.toString());
    }

    @Override
    public int hashCode() {
        int total = 0;
        for (int i=0; i<array.length; i++) {
            total += array[i];
        }
        System.out.println(total);
        return total;
    }
```

`SillyArray` también provee `setChar`, lo que hace posible modificar los caracteres en el arreglo:

```
public void setChar(int i, char c) {
    this.array[i] = c;
}
```

Ahora supón que creamos un `SillyArray` y lo agregamos a un mapa:

```
SillyArray array1 = new SillyArray("Word1".toCharArray());
map.put(array1, 1);
```

El código hash para este arreglo es 461. Ahora si modificamos el contenido del arreglo y luego tratamos de ubicarlo, así:

```
array1.setChar(0, 'C');
Integer value = map.get(array1);
```

El código hash tras la mutación es 441. Con un código hash diferente, hay una probabilidad alta de que buscaremos en el sub-mapa equivocado. En ese caso, no encontraremos la clave, incluso si está en el mapa. Y eso es malo.

En general, es peligroso usar objetos mutables como claves en estructuras de datos que usan hashing, lo que incluye a `MyBetterMap` y a `HashMap`. Si puedes garantizar que las claves no se modificarán mientras están en el mapa, o que cualquier cambio no afectará el código hash, podría ser aceptable. Pero probablemente es una buena idea evitarlo.

## 10.4 Ejercicio 8

En este ejercicio, finalizarás la implementación de `MyBetterMap`. En el repositorio para este libro, encontrarás los archivos de código fuente para este ejercicio:

- `MyLinearMap.java` contiene nuestra solución al ejercicio anterior, sobre la que construiremos en este ejercicio.
- `MyBetterMap.java` contiene el código del capítulo anterior con algunos métodos que tendrás que llenar.
- `MyHashMap.java` contiene un boceto de una tabla hash que crece cuando se necesita, la cual completarás.
- `MyLinearMapTest.java` contiene los tests unitarios para `MyLinearMap`.
- `MyBetterMapTest.java` contiene los tests unitarios para `MyBetterMap`.

- `MyHashMapTest.java` contiene los tests unitarios para `MyHashMap`.
- `Profiler.java` contiene código para medir y graficar el tiempo de ejecución versus el tamaño del problema.
- `ProfileMapPut.java` contiene código que perfile el método `Map.put`.

Como ya es costumbre, ejecutarás `ant build` para compilar los archivos de código fuente. Luego ejecuta `ant MyBetterMapTest`. Varios tests deberían fallar, ¡porque aún tienes trabajo por hacer!

Revisa la implementación de `put` y `get` del capítulo anterior. Luego llena el cuerpo de `containsKey`. PISTA: usa `chooseMap`. Ejecuta `ant MyBetterMapTest` de nuevo y confirma que pasa `testContainsKey`.

Llena el cuerpo de `containsValue`. PISTA: *no* uses `chooseMap`. Ejecuta `ant MyBetterMapTest` de nuevo y confirma que pasa `testContainsValue`. Nota que tenemos que trabajar más para encontrar un valor que para encontrar una clave.

Como `put` y `get`, esta implementación de `containsKey` es lineal, porque tiene que buscar uno de los sub-mapas embebidos. En el siguiente capítulo, veremos cómo podemos mejorar esta implementación aún más.

# Capítulo 11

# HashMap

En el capítulo anterior, escribimos una implementación de la interfaz `Map` que usa hashing. Esperamos que esta versión sea más rápida, porque las listas en las busca son más cortas, pero el orden de crecimiento todavía es lineal.

Si hay $n$ entradas y $k$ sub-mapas, el tamaño de los sub-mapas es $n/k$ en promedio, que aún es proporcional a $n$. Pero si incrementamos $k$ junto con $n$, podemos limitar el tamaño de $n/k$.

Por ejemplo, supón que duplicamos $k$ cada vez que $n$ excede $k$; en ese caso el número de entradas por mapa sería menor a 1 en promedio, y prácticamente menor que 10 todo el tiempo, siempre que la función hash distribuya las claves razonablemente bien.

Si el número de entradas por sub-mapa es constante, podemos buscar en un único sub-mapa en tiempo constante. Y calcular la función hash generalmente requiere tiempo constante (podría depender del tamaño de la clave, pero no depende del número de claves). Eso hace a los métodos principales de `Map`,`put` y `get`, de tiempo constante.

En el siguiente ejercicio, verás los detalles.

## 11.1 Ejercicio 9

En `MyHashMap.java`, proveo el código base de una tabla hash que crece cuando se necesita. Aquí está el principio de la definición:

```
public class MyHashMap<K, V> extends MyBetterMap<K, V> implements Map<K, V> {

    // número promedio de entradas por sub-mapa antes de recodificar (rehash)
    private static final double FACTOR = 1.0;

    @Override
    public V put(K key, V value) {
        V oldValue = super.put(key, value);

        // comprueba si el número de elementos por sub-mapa excede el umbral
        if (size() > maps.size() * FACTOR) {
            rehash();
        }
        return oldValue;
    }
}
```

`MyHashMap` extiende `MyBetterMap`, así que hereda los métodos definidos ahí. El único método que sobrescribe es `put` que llama a `put` en la superclase — es decir, llama a la versión de `put` en `MyBetterMap` — y luego comprueba si tienen que recodificar (rehash). La llamada a `size` devuelve el número total de entradas, $n$. La llamada a `maps.size` devuelve el número de mapas embebidos, $k$.

La contante `FACTOR`, que se conoce como **factor de carga**, determina el número máximo de entradas por sub-mapa, en promedio. Si `n > k * FACTOR`, significa que `n/k > FACTOR`, lo que implica que el número de entradas por sub-mapa excede el umbral, así que llamamos a `rehash`.

Ejecuta `ant build` para compilar los archivos de código fuente. Luego ejecuta `ant MyHashMapTest`. Debería fallar porque la implementación de `rehash` lanza una excepción. Tu trabajo es completarlo.

Llena el cuerpo de `rehash` para recopilar las entradas en la tabla, redimensionar la tabla y luego poner las entradas de regreso. Proveo dos métodos que podrían servirte: `MyBetterMap.makeMaps` y `MyLinearMap.getEntries`. Tu solución debería duplicar el número de mapas, $k$, cada vez que es llamada.

## 11.2 Análisis de MyHashMap

Si el número de entradas en el sub-mapa más grande es proporcional a $n/k$, y $k$ crece en proporción a $n$, varios de los métodos principales de `MyBetterMap` se vuelven de tiempo constante:

```
public boolean containsKey(Object target) {
    MyLinearMap<K, V> map = chooseMap(target);
    return map.containsKey(target);
}

public V get(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.get(key);
}

public V remove(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.remove(key);
}
```

Cada método codifica (hashes) un clave (key), lo que es de tiempo constante y luego invoca un método en un sub-mapa, que también es de tiempo constante.

Hasta el momento, vamos bien. Pero el otro método principal, `put`, es un poco más difícil de analizar. Cuando no tenemos que recodificar, es de tiempo constante, pero cuando tenemos que hacerlo, es lineal. En ese sentido, es similar a `ArrayList.add`, que analizamos en la Sección 3.2.

Por la misma razón, `MyHashMap.put` resulta ser de tiempo constante si promediamos una serie de invocaciones De nuevo, el argumento se base en el análisis amortizado (ver la Sección 3.2).

Supón que el número inicial de sub-mapas, $k$, es 2, y que el factor de carga es 1. Ahora veamos cuánto trabajo requiere poner (`put`) una serie de claves. Como "unidad de trabajo" básica, contaremos el número de veces que tenemos que codificar una clave y agregarla a un sub-mapa.

La primera vez que llamamos a `put` requiere 1 unidad de trabajo. La segunda vez también toma 1 unidad. La tercera vez tenemos que recodificar, así que

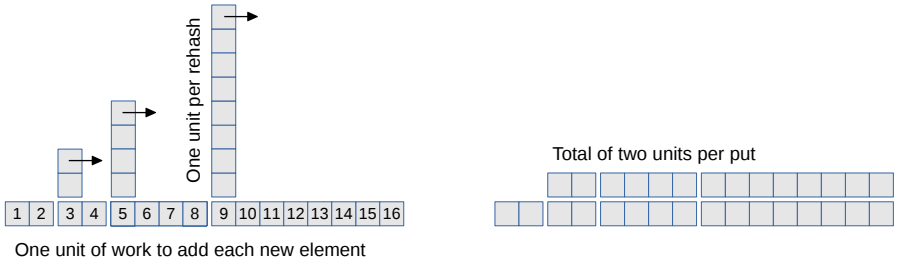


Figura 11.1: Representación del trabajo realizado para agregar elementos a una tabla hash.

toma 2 unidades recodificar las llaves existentes y 1 unidad recodificar la nueva clave.

Ahora el tamaño de la tabla hash es 4, así que la siguiente vez que llamemos a `put`, toma 1 unidad de trabajo. Pero la siguiente vez tenemos que recodificar, lo que toma 4 unidades para recodificar las llaves existentes y 1 unidad para recodificar la nueva clave.

La figura 11.1 muestra el patrón, con la carga normal de trabajo de codificar una nueva clave en la parte inferior y el trabajo extra de recodificar se muestra como una torre.

Como la flecha sugiere, si derribamos las torres, cada una llenará el espacio antes de la siguiente torre. El resultado es una altura uniforme de 2 unidades, que muestra que el trabajo promedio por `put` es de alrededor de 2 unidades. Y eso significa que `put` es de tiempo constante en promedio.

Este diagrama también muestra por qué es importante duplicar el número de sub-mapas, $k$, cuando recodificamos. Si solamente sumamos a $k$ en lugar de multiplicar, las torres estarían demasiado cercanas entre sí y comenzarían a apilarse. Y eso no sería de tiempo constante.

## 11.3 Limitaciones

Hemos mostrado que `containsKey`, `get`, y `remove` son de tiempo constante y que `put` es de tiempo constante en promedio. Deberíamos tomarnos un minuto para apreciar cuán notable es eso. El rendimiento de estas operaciones es prácticamente el mismo sin importar qué tan grande sea la tabla hash. Bueno, en cierto modo.

Recuerda que nuestro análisis se basa un un modelo simple de cálculo donde cada "unidad de trabajo" toma la misma cantidad de tiempo. En la vida real, las computadoras son más complicadas. En particular, usualmente son más rápidas cuando trabajan con estructuras de datos lo suficientemente pequeñas para caber en la caché, un poco más lentas si la estructura de datos no cabe en caché pero sí en la memoria; y *mucho* más lentas si la estructura no cabe en la memoria.

Otra limitación de esta implementación es que el hashing no nos ayuda si nos dan un valor en lugar de una clave: `containsValue` es lineal porque tiene que buscar todos los sub-mapas. Y no hay una forma particularmente eficiente de buscar un valor y encontrar la clave (o posiblemente, claves) correspondiente.

Y hay una limitación más: algunos de los métodos que eran de tiempo constante en `MyLinearMap` se han vuelto lineales. Por ejemplo:

```
public void clear() {
    for (int i=0; i<maps.size(); i++) {
        maps.get(i).clear();
    }
}
```

`clear` tiene que limpiar todos los sub-mapas, y el número de sub-mapas es proporcional a $n$, así que es lineal. Afortunadamente, esta operación no se usa muy a menudo, así que para la mayoría de aplicaciones este sacrificio es aceptable.

## 11.4 Perfilado de `MyHashMap`

Antes de continuar, deberíamos comprobar si `MyHashMap.put` realmente es de tiempo constante.

Ejecuta `ant build` para compilar los archivos de código fuente. Luego ejecuta `ant ProfileMapPut`. Este programa mide el tiempo de ejecución de `HashMap.put` (proporcionado por Java) con un rango de tamaños de problemas y grafica el tiempo de ejecución versus el tamaño del problema en una escala log-log. Si esta operación es de tiempo constante, el tiempo total para $n$ operaciones debería ser lineal, así que el resultado debería ser una línea recta con pendiente de 1. Cuando ejecuté este código, la pendiente estimada era cercana a 1, lo que es consistente con nuestro análisis. Deberías obtener algo similar.

Modifica `ProfileMapPut.java` para perfilar tu implementación, `MyHashMap`, en lugar del `HashMap` de Java. Ejecuta el perfilador de nuevo y observa si la pendiente es cercana a 1. Podrías tener que ajustar `startN` y `endMillis` para encontrar un rango de tamaños del problema donde los tiempos de ejecución sean mayores a unos pocos milisegundos, pero no más de unos cuantos miles.

Cuando ejecuté este código, me sorprendí: la pendiente era alrededor de 1.7, lo que sugiere que esta implementación no es de tiempo constante después de todo. Contiene un "bug de rendimiento'.

Antes de leer la siguiente sección, deberías rastrear el error, arreglarlo y confirmar que `put` es de tiempo constante, como se esperaba.

## 11.5 Arreglando `MyHashMap`

El problema con `MyHashMap` está en `size`, que se heredó de `MyBetterMap`:

```
public int size() {
    int total = 0;
    for (MyLinearMap<K, V> map: maps) {
        total += map.size();
    }
    return total;
}
```

Para calcular el tamaño total tiene que iterar por todos los sub-mapas. Dado que incrementamos el número de submapas, $k$, conforme el número de entradas, $n$, se incrementa, $k$ es proporcional a $n$, así que `size` es lineal.

Y eso hace que `put` también sea lineal, porque usa a `size`:

```
    public V put(K key, V value) {
        V oldValue = super.put(key, value);

        if (size() > maps.size() * FACTOR) {
            rehash();
        }
        return oldValue;
    }
```

¡Todo lo que hicimos para hacer a `put` de tiempo constante es un desperdicio si `size` es lineal!

Afortunadamente, hay una solución simple, y la hemos visto antes: tenemos que llevar la cuenta del número de entradas en una variable de instancia y actualizarla cada vez que llamemos a un método que la cambie.

Encontrarás esta solución en el repositorio para este libro, en `MyFixedHashMap.java`. Aquí está el principio de la definición de la clase:

```
public class MyFixedHashMap<K, V> extends MyHashMap<K, V> implements Map<K, V>

    private int size = 0;

    public void clear() {
        super.clear();
        size = 0;
    }
```

En lugar de modificar `MyHashMap`, defino una nueva clase que la extiende. Esta clase agrega una nueva variable de instancia, `size`, que se inicializa en cero.

Actualizar `clear` es directo; invocamos a `clear` en la superclase (lo que limpia los sub-mapas), y luego actualizamos `size`.

Actualizar `remove` y `put` es un poco más difícil porque cuando invocamos el método en la superclase, no podemos decir si el tamaño del sub-mapa cambió. Así es como superé esta limitante:

```
    public V remove(Object key) {
        MyLinearMap<K, V> map = chooseMap(key);
```

```
        size -= map.size();
        V oldValue = map.remove(key);
        size += map.size();
        return oldValue;
    }
```

`remove` usa a `chooseMap` para encontrar el sub-mapa correcto, luego sustrae el tamaño (size) del sub-mapa. Invoca a `remove` en el sub-mapa, que puede o no cambiar el tamaño del sub-mapa, dependiendo de si encuentra la clave. Pero de cualquier forma, agregamos el nuevo tamaño del sub-mapa a `size`, así que el valor final de `size` es correcto.

La versión reescrita de `put` es similar:

```
    public V put(K key, V value) {
        MyLinearMap<K, V> map = chooseMap(key);
        size -= map.size();
        V oldValue = map.put(key, value);
        size += map.size();

        if (size() > maps.size() * FACTOR) {
            size = 0;
            rehash();
        }
        return oldValue;
    }
```

Tenemos el mismo problema aquí: cuando invocamos a `put` en el sub-mapa, no sabemos si se agregó una nueva entrada. Así que usamos la misma solución, sustrayendo el tamaño (size) antiguo y luego sumando el nuevo.

Ahora la implementación del método `size` es simple:

```
    public int size() {
        return size;
    }
```

Y esta es claramente de tiempo constante.

Cuando perfilé esta solución, encontré que el tiempo total para poner $n$ claves es proporcional a $n$, lo que significa que cada `put` es de tiempo constante, como debe ser.

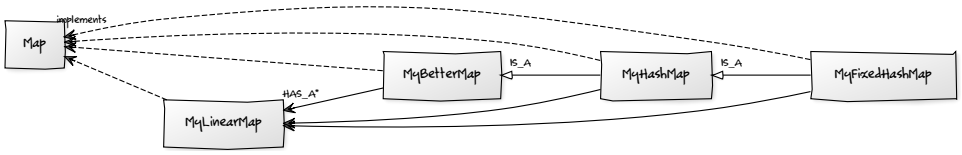


Figura 11.2: Diagrama UML para las clases de este capítulo.

## 11.6 Diagramas de clases UML

Un desafío de trabajar con el código en este capítulo es que tenemos varias clases que dependen una de la otra. Aquí están algunas de las relaciones entre las clases:

- `MyLinearMap` contiene una `LinkedList` e implementa `Map`.
- `MyBetterMap` contiene muchos objetos `MyLinearMap` e implementa `Map`.
- `MyHashMap` extiende a `MyBetterMap`, así que también contiene objetos `MyLinearMap` e implementa `Map`.
- `MyFixedHashMap` extiende a `MyHashMap` e implementa `Map`.

Para llevar el control de relaciones como estas, los ingenieros de software a menudo usan **diagramas de clases UML**. UML significa Unified Modeling Language (Lenguaje de Modelado Unificado) (véase http://thinkdast.com/uml). Un "diagrama de clase" es uno de varios estándares gráficos definidos por UML.

En un diagrama de clases, cada clase se representa con una caja y las relaciones entre clases se representan con flechas. La figura 11.2 muestra un diagrama de clases UML para las clases del ejercicio anterior, generado usando la herramienta en línea yUML en http://yuml.me/.

Las relaciones diferentes se representan con flechas diferentes:

- Las flechas con una punta sólida indican una relación TIENE-UN(A). Por ejemplo, cada instancia de `MyBetterMap` contiene múltiples instancias de `MyLinearMap`, así que se conectan con una flecha sólida.

- Las flechas con una punta en blanco y una línea sólida indican relaciones ES-UN(A). Por ejemplo, `MyHashMap` extiende a `MyBetterMap`, así que se conectan con una flecha ES-UN(A).
- Las flechas con una punta en blanco y una línea punteada indican que una clase implementa una interfaz; en este diagrama, cada clase implementa `Map`.

Los diagramas de clase UML proveen una forma concisa de representar un montón de información sobre una colección de clases. Usualmente se usan durante las fases de diseño para comunicar diseños alternativos, durante las fases de implementación para mantener un mapa mental compartido del proyecto y durante el despliegue para documentar el diseño.

# Capítulo 12

# TreeMap

Este capítulo presenta el árbol binario de búsqueda, que es una implementación eficiente de la interfaz `Map` que es particularmente útil si queremos mantener los elementos ordenados.

## 12.1 ¿Qué tiene de malo el hashing?

En este momento, ya deberías estar familiarizado con la interfaz `Map` y la implementación `HashMap` provista por Java. Y al construir tu propio `Map` usando una tabla hash, deberías entender cómo funciona `HashMap` y por qué esperamos que sus métodos principales sean de tiempo lineal.

Debido a su desempeño, `HashMap` es ampliamente usado, pero no es la única implementación de `Map`. Hay una pocas razones por las que podrías querer otra implementación:

1. El hashing puede ser lento, así que aunque las operaciones de `HashMap` sean de tiempo constante, la "constante" podría ser grande.

2. El hashing funciona bien si la función hash distribuye las llaves de forma equitativa entre los sub-mapas. Pero el diseño de buenas funciones hash no es fácil y si demasiadas claves terminan en el mismo sub-mapa, el desempeño de `HashMap` podría ser pobre.

3. Las llaves en una tabla hash no se guardan en ningún orden particular; de hecho, el orden podría cambiar cuando la tabla crezca y las llaves sean recodificadas (rehashed). Para algunas aplicaciones, es necesario, o por lo menos útil, mantener las clave en orden.

Es difícil resolver todos estos problemas a la vez, pero Java provee una implementación llamada `TreeMap` que se casi lo consigue:

1. No usa una función hash, así que evita el costo del hashing y la dificultad de elegir una función hash.

2. Dentro del `TreeMap`, las claves se guardan en un **árbol binario de búsqueda**, lo que posibilita recorrer las claves, en orden, en tiempo lineal.

3. El tiempo de ejecución de los métodos principales es proporcional a $\log n$, que no tan bueno como tiempo constante, pero todavía es muy bueno.

En la siguiente sección, explicaré cómo funcionan los árboles binarios de búsqueda y usarás uno para implementar un `Map`. Durante el viaje, analizaremos el desempeño de los métodos principales de un mapa cuando se implementan con un árbol.

## 12.2 Árbol binario de búsqueda

Un árbol binario de búsqueda (ABB) es un árbol donde cada nodo (node) contiene una clave y cada `node` tiene la "propiedad ABB":

1. Si `node` tiene un hijo izquierdo, todas las claves en el subárbol izquierdo deben ser menores que la clave en `node`.

2. Si `node` tiene un hijo derecho, todas las claves en el subárbol derecho deben ser mayores que la clave en el `node`.

La figura 12.1 muestra un árbol de enteros que tiene esta propiedad. Esta figura es de la página de Wikipedia sobre árboles binarios de búsqueda en http://thinkdast.com/bst, que podrías encontrar útil mientras trabajas en este ejercicio.

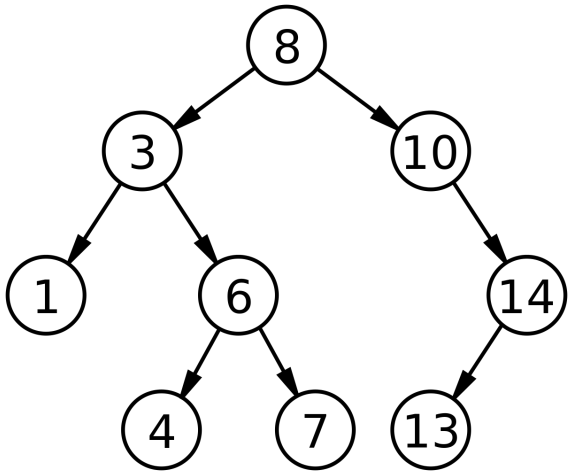


Figura 12.1: Ejemplo de un árbol binario de búsqueda.

La clave en la raíz es 8 y puedes confirmar que todas las claves a la izquierda de la raíz son menores que 8 y todas las claves a la derecha son mayores. También puedes verificar que los otros nodos tienen esta propiedad.

Buscar una clave en un árbol binario de búsqueda es rápido porque no tenemos que buscar en el árbol entero. Comenzando en la raíz, podemos usar el siguiente algoritmo:

1. Compara la clave (key) que estás buscando, `target`, a la clave en el nodo actual. Si son iguales, has terminado.

2. Si `target` es menor que la clave actual, busca en el árbol izquierdo. Si no hay uno, `target` no está en el árbol.

3. Si `target` es mayor que la clave actual, busca en el árbol derecho. Si no hay uno, `target` no está en el árbol.

En cada nivel del árbol, tienes que buscar sólo en un hijo. Por ejemplo, si buscar `target = 4` en el diagrama anterior, comienzas en la raíz, que contiene la clave `8`. Porque `target` es menor que `8`, te vas a la izquierda. Porque `target` es mayor que `3` te vas a la derecha. Porque `target` es menor que `6`, te vas a la izquierda. Y entonces encuentras la clave que estabas buscando.

En este ejemplo, toma cuatro comparaciones encontrar el objetivo (target), aunque el árbol contiene nueve claves. En general, el número de comparaciones es proporcional a la altura del árbol, no al número de claves en el árbol.

Entonces, ¿qué podemos decir sobre la relación entre la altura del árbol, `h`, y el número de nodos, $n$? Comenzando con un número bajo e incrementándolo gradualmente:

- Si `h=1`, el árbol sólo contiene un nodo, así que `n=1`.
- Si `h=2`, podemos agregar dos nodos más, para un total de `n=3`.
- Si `h=3`, podemos agregar hasta cuatro nodos más, para un total de `n=7`.
- Si `h=4`, podemos agregar hasta ocho nodos más, para un total de `n=15`.

Por ahora puedes ver el patrón. Si numeramos los niveles del árbol desde `1` hasta `h`, el nivel con índice `i` puede tener hasta $2^{i-1}$ nodos. Y el número total de nodos en `h` niveles es $2^h - 1$. Si tenemos

$$n = 2^h - 1$$

podemos calcular el logaritmo base 2 de ambos lados:

$$log_2 n \approx h$$

lo que significa que la altura del árbol es proporcional a $\log n$, si el árbol está lleno; es decir, si cada nivel contiene el máximo número de nodos.

Así que esperamos poder buscar una clave en un árbol binario de búsqueda en un tiempo proporcional a $\log n$. Esto es cierto si el árbol está lleno e incluso si el árbol está solo parcialmente lleno. Pero no siempre es cierto, como veremos.

Un algoritmo que toma tiempo proporcional a $\log n$ es llamado "logarítmico" o de "tiempo logarítmico", y pertenece a la orden de crecimiento $O(\log n)$.

## 12.3 Ejercicio 10

Para este ejercicio escribirás una implementación de la interfaz `Map` usando un árbol binario de búsqueda.

Aquí está el principio de una implementación, llamada `MyTreeMap`:

```
public class MyTreeMap<K, V> implements Map<K, V> {

    private int size = 0;
    private Node root = null;
```

Las variables de instancia son `size`, que lleva el control del número de claves, y `root`, que es una referencia al nodo raíz del árbol. Cuando el árbol está vacío, `root` es `null` y `size` es 0.

Aquí está la definición de `Node`, que se define dentro de `MyTreeMap`:

```
    protected class Node {
        public K key;
        public V value;
        public Node left = null;
        public Node right = null;

        public Node(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
```

Cada nodo contiene un par clave-valor (key-value) y referencias a dos nodos hijos `left` (izquierdo) y `right` (derecho). Cualquiera o ambos de los nodos hijos pueden ser `null`.

Algunos de los métodos de `Map` son fáciles de implementar, como `size` y `clear`:

```
    public int size() {
        return size;
    }

    public void clear() {
        size = 0;
        root = null;
    }
```

`size` es claramente de tiempo constante.

`clear` pareciera ser de tiempo constante, pero considera esto: cuando `root` se modifica a `null`, el recolector de basura reclama la memoria de los nodos en el árbol, lo que toma tiempo lineal. ¿Deberíamos contar el trabajo que realiza el recolector de basura? Pienso que sí.

En la siguiente sección, llenarás algunos de los otros métodos, incluyendo los más importantes, `get` y `put`.

## 12.4 Implementación de un TreeMap

En el repositorio para este libro, encontrarás estos archivos de código fuente:

- `MyTreeMap.java` contiene el código de la sección anterior con un boceto para los métodos faltantes.
- `MyTreeMapTest.java` contiene los tests unitarios para `MyTreeMap`.

Ejecuta `ant build` para compilar los archivos de código fuente. Luego ejecuta `ant MyTreeMapTest`. Varios tests deberían fallar, ¡porque tienes trabajo por hacer!

He provisto bocetos para `get` y `containsKey`. Ambos usan `findNode`, que es un método privado que yo definí; no es parte de la interfaz `Map`. Así es como comienza:

```
private Node findNode(Object target) {
    if (target == null) {
        throw new IllegalArgumentException();
    }

    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) target;

    // TODO: FILL THIS IN!
    return null;
}
```

El parámetro `target` es la clave que estamos buscando. Si `target` es `null`, `findNode` lanza una excepción. Algunas implementaciones de `Map` pueden manejar `null` como una clave, pero en un árbol binario de búsqueda, necesitamos ser capaces de comparar claves, así que lidiar con `null` es problemático. Para mantener las cosas simples, esta implementación no permite `null` como una clave.

Las siguientes líneas muestran cómo podemos comparar `target` a una clave en el árbol. De la firma de `get` y `containsKey`, el compilador considera a `target` como un `Object`. Pero necesitamos poder comparar claves, así que convertimos `target` a un `Comparable<? super K>`, lo que significa que es comparable a una instancia de tipo `K`, o cualquier superclase de `K`. Si no estás familiarizado con este uso del "tipo comodín", puedes leer más al respecto en http://thinkdast.com/gentut.

Afortunadamente, lidiar con el sistema de tipos de Java no es el punto de este ejercicio. Tu trabajo es llenar el resto de `findNode`. Si encuentra un nodo que contiene `target` como una clave, debería devolver el nodo. De lo contrario, debería devolver `null`. Cuando logres que esto funcione, debería pasar los tests para `get` y `containsKey`.

Nota que tu solución debería buscar únicamente una ruta a lo largo del árbol, así que debería tomar tiempo proporcional a la altura del árbol. ¡No deberías buscar en todo el árbol!

Tu siguiente tarea es llenar `containsValue`. Para comenzar, he provisto un método auxiliar, `equals`, que compara `target` y una clave dada. Nota que los valores en el árbol (a diferencia de las claves) no son necesariamente comparables, así que no podemos usar `compareTo`; tenemos que invocar `equals` en el `target`.

A diferencia de tu solución anterior para `findNode`, tu solución para `containsValue` *tiene* que buscar en el árbol completo, así que su tiempo de ejecución es proporcional al número de claves, $n$, no a la altura del árbol, `h`.

El siguiente método que deberías llenar es `put`. He provisto un código inicial que maneja los casos más simples:

```
public V put(K key, V value) {
    if (key == null) {
```

```
            throw new IllegalArgumentException();
        }
        if (root == null) {
            root = new Node(key, value);
            size++;
            return null;
        }
        return putHelper(root, key, value);
    }

    private V putHelper(Node node, K key, V value) {
        // TODO: Fill this in.
    }
```

Si tratas de poner `null` como una clave, `put` lanza una excepción.

Si el árbol está vacío, `put` crea un nuevo nodo e inicializa la variable de instancia `root`.

De otra forma, llama a `putHelper`, que es un método privado que yo definí; no es parte de la interfaz `Map`.

Llena `putHelper` para que busque en el árbol y:

1. Si `key` ya está en el árbol, reemplace el valor antiguo con el nuevo, y devuelva el valor antiguo.
2. Si `key` no está en el árbol, cree un nuevo nodo, encuentre el lugar correcto para agregarlo y devuelva `null`.

Tu implementación de `put` debería tomar tiempo proporcional a la altura del árbol, $h$, no al número de elementos, $n$. Idealmente debería buscar en el árbol solo una vez, pero si encuentras más fácil buscar dos veces, puedes hacerlo; será más lento, pero no cambia el orden de crecimiento.

Finalmente, deberías llenar el cuerpo de `keySet`. De acuerdo con la documentación en http://thinkdast.com/mapkeyset, este método debería devolver un `Set` que itere por las claves en orden; es decir, en orden ascendente de acuerdo con el método `compareTo`. La implementación `HashSet` de un `Set`, que usamos en la Sección 8.3, no mantiene el orden de las claves,

pero la implementación `LinkedHashSet` sí lo hace. Puedes leer sobre ella en http://thinkdast.com/linkedhashset.

He provisto un boceto de `keySet` que crea y devuelve un `LinkedHashSet`:

```
public Set<K> keySet() {
    Set<K> set = new LinkedHashSet<K>();
    return set;
}
```

Deberías finalizar este método para que agregue las claves del árbol al `set` en orden ascendente. PISTA: podrías querer escribir un método auxiliar; podrías querer hacerlo recursivo; y podrías querer leer sobre el recorrido en orden de un árbol en http://thinkdast.com/inorder.

Cuando hayas finalizado, deberían pasar todos los tests. En el siguiente capítulo, presentaré mis soluciones y comprobaré el desempeño de los métodos principales.

# Capítulo 13

# Árbol binario de búsqueda

Este capítulo presenta las soluciones al ejercicio anterior, luego prueba el desempeño del mapa basado en un árbol. Presento un problema con la implementación y explico cómo el `TreeMap` de Java lo resuelve.

## 13.1 Un `MyTreeMap` simple

En el ejercicio anterior te dí un boceto de `MyTreeMap` y te pedí que llenaras los métodos faltantes. Ahora presentaré una solución, comenzando con `findNode`:

```
private Node findNode(Object target) {
    // algunas implementaciones pueden tratar a null como clave, no esta
    if (target == null) {
            throw new IllegalArgumentException();
    }

    // algo para hacer feliz al compilador
    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) target;

    // la búsqueda como tal
    Node node = root;
    while (node != null) {
```

```
        int cmp = k.compareTo(node.key);
        if (cmp < 0)
            node = node.left;
        else if (cmp > 0)
            node = node.right;
        else
            return node;
    }
    return null;
}
```

`findNode` es un método privado usado por `containsKey` y `get`; no es parte de la interfaz `Map` El parámetro `target` es la clave que buscamos. Expliqué la primera parte de este método en el ejercicio previo:

- En esta implementación, `null` no es un valor legal para una clave.
- Antes de que podamos invocar a `compareTo` en `target`, tenemos que convertirlo a algún tipo de `Comparable`. El "tipo comodín" usado aquí es tan permisivo como resulta posible; es decir, funciona con cualquier tipo que implemente `Comparable` y cuyo método `compareTo` acepte `K` o cualquier supertipo de `K`.

Después de todo eso, la búsqueda como tal es relativamente simple. Inicializamos una variable de repetición `node` que se refiera al nodo raíz. En cada repetición del bucle, comparamos el objetivo (target) con `node.key`. Si el objetivo es menor que la clave actual, nos movemos al hijo izquierdo. Si es mayor, nos movemos el hijo derecho. Y si son iguales, devolvemos el nodo actual.

Si llegamos a la parte inferior del árbol sin encontrar el objetivo, concluimos que no está en el árbol y devolvemos `null`.

## 13.2 Búsqueda de valores

Como expliqué en el ejercicio previo, el tiempo de ejecución de `findNode` es proporcional a la altura del árbol, no al número de nodos, porque no tenemos que buscar en el árbol completo. Pero para `containsValue`, tenemos que

buscar los valores, no las claves; la propiedad ABB no aplica a los valores, así que tenemos que buscar en el árbol completo.

Mi solución es recursiva:

```
public boolean containsValue(Object target) {
    return containsValueHelper(root, target);
}

private boolean containsValueHelper(Node node, Object target) {
    if (node == null) {
        return false;
    }
    if (equals(target, node.value)) {
        return true;
    }
    if (containsValueHelper(node.left, target)) {
        return true;
    }
    if (containsValueHelper(node.right, target)) {
        return true;
    }
    return false;
}
```

`containsValue` toma el valor objetivo como un parámetro e invoca inmediatamente a `containsValueHelper`, pasando la raíz del árbol como un parámetro adicional.

Así es cómo funciona `containsValueHelper`:

- La primera instrucción `if` comprueba el caso base de la recursión. Si `node` es `null`, eso significa que hemos llegado hasta la parte inferior del árbol sin encontrar el `target`, así que deberíamos devolver `false`. Nota que esto sólo implica que el objetivo no aparecía en una ruta del árbol; aún es posible que sea encontrado en otra.
- El segundo caso comprueba si hemos encontrado lo que estamos buscando. De ser así, devolveríamos `true`. De otra manera, tenemos que continuar.

- El tercer caso realiza una llamada recursiva a la búsqueda para `target` en el subárbol izquierdo. Si lo encontramos, podemos devolver `true` inmediatamente, sin tener que buscar en el subárbol derecho. De lo contrario, seguimos.
- El cuarto caso busca en el subárbol derecho. De nuevo, si encontramos lo que estamos buscando, devolvemos `true`. De lo contrario, tras haber buscado en todo el árbol, devolvemos `false`.

Este método "visita" cada nodo en el árbol, así que toma tiempo proporcional al número de nodos.

## 13.3 Implementación de `put`

El método `put` es un poco más complicado que `get` porque tiene que lidiar con dos casos: (1) si la clave dada ya está en el árbol, la reemplaza y devuelve el valor antiguo; (2) de lo contrario tiene que agregar un nuevo nodo al árbol, en el lugar correcto.

En el ejercicio anterior, proveía este código inicial:

```
public V put(K key, V value) {
    if (key == null) {
        throw new IllegalArgumentException();
    }
    if (root == null) {
        root = new Node(key, value);
        size++;
        return null;
    }
    return putHelper(root, key, value);
}
```

Y te pedí que llenaras `putHelper`. Aquí está mi solución:

```
private V putHelper(Node node, K key, V value) {
    Comparable<? super K> k = (Comparable<? super K>) key;
    int cmp = k.compareTo(node.key);
```

```
    if (cmp < 0) {
        if (node.left == null) {
            node.left = new Node(key, value);
            size++;
            return null;
        } else {
            return putHelper(node.left, key, value);
        }
    }
    if (cmp > 0) {
        if (node.right == null) {
            node.right = new Node(key, value);
            size++;
            return null;
        } else {
            return putHelper(node.right, key, value);
        }
    }
    V oldValue = node.value;
    node.value = value;
    return oldValue;
}
```

El primer parámetro, `node`, es inicialmente la raíz del árbol, pero cada vez que hacemos una llamada recursiva, se refiere un subárbol diferente. Como en `get`, usamos el método `compareTo` para decidir qué ruta seguir en el árbol. Si `cmp < 0`, la clave que estamos agregando es menor que `node.key`, así que queremos buscar en el subárbol izquierdo. Aquí hay dos casos:

- Si el subárbol izquierdo está vacío, es decir, si `node.left` es `null`, hecmos alcanzado la parte inferior del árbol sin encontrar a `key`. En este punto, sabemos que `key` no está en el árbol y sabemos dónde debería ir. Así que creamos un nuevo nodo y lo agregamos como el hijo izquierdo de `node`.
- De lo contarior, hacemos una llamada recursiva para buscar en el subárbol izquierdo.

Si `cmp > 0`, la clave que estamos agregando es mayor que `node.key`, así que queremos buscar en el subárbol derecho. Y se tienen los mismo dos casos

que en la rama anterior. Finalmente, si `cmp == 0`, encontramos la clave en el árbol, así que la reemplazamos y devolvemos el valor antiguo.

Escribí este método recursivamente para hacerlo más legible, pero debería ser sencillo reescribirlo de forma iterativa, lo que podrías querer hacer como un ejercicio.

## 13.4 Recorrido en orden

El último método que te pedí escribir es `keySet`, que devuelve un `Set` que contiene las claves del árbol en orden ascendente. En otras implementaciones de `Map`, las claves devueltas por `keySet` no están en ningún orden particular, pero una de las características de la implementación de árbol es que es simple y eficiente ordenar las claves. Así que deberíamos aprovechar eso.

Aquí está mi solución:

```
public Set<K> keySet() {
    Set<K> set = new LinkedHashSet<K>();
    addInOrder(root, set);
    return set;
}

private void addInOrder(Node node, Set<K> set) {
    if (node == null) return;
    addInOrder(node.left, set);
    set.add(node.key);
    addInOrder(node.right, set);
}
```

En `keySet`, creamos un `LinkedHashSet`, que es una implementación de `Set` que mantiene los elementos en orden (a diferencia de la mayoría de las demás implmentaciones de `Set`). Luego llamamos a `addInOrder` para recorrer el árbol.

El primer parámetro, `node`, es inicialmente la raíz del árbol, pero como deberías esperar por ahora, la usamos para navegar por el árbol de forma recursiva. `addInOrder` realiza un clásico "recorrido en orden" del árbol.

Si el `node` es `null`, significa que el subárbol está vacío, así que nos salimos sin agregar nada al `set`. De lo contrario:

1. Recorre el subárbol izquierdo en orden.
2. Agrega `node.key`.
3. Navega por el subárbol derecho en orden.

Recuerda que la propiedad ABB garantiza que todos los nodos en el subárbol izquierdo son menores que `node.key`, y todos los nodos en el subárbol derecho son mayores. Así que sabemos que `node.key` ha sido agregado en el orden correcto.

Al aplicar el mismo argumento recursivamente, sabemos que los elementos del subárbol izquierdo están en orden, así como los elementos del subárbol derecho. Y el caso base es correcto: si el subárbol está vacío, no se agregan claves. Así que podemos concluir que este método agrega todas las claves en el orden correcto.

Porque este método visita cada nodo en el árbol, como `containsValue`, toma tiempo proporcional a $n$.

## 13.5 Los métodos logarítmicos

En `MyTreeMap`, los métodos `get` y `put` toman tiempo proporcional a la altura del árbol, $h$. En el ejercicio anterior, mostramos que si el árbol está lleno — si cada nivel del árbol contiene el máximo número de nodos — la altura del árbol es proporcional a $\log n$.

Y afirmé que `get` y `put` son logarítmicos; es decir, toman tiempo proporcional a $\log n$. Pero para la mayoría de aplicaciones, no hay garantías que el árbol esté llenon. En general, la forma del árbol depende de las claves y el orden en que se agreguen.

Para ver cómo esto funciona en la práctica, probaremos nuestra implementación con dos conjuntos de datos de ejemplo: una lista de cadenas al azar y una lista de marcas de tiempo (timestamps) en orden ascendente.

Aquí está el código que genera cadenas al azar:

```
Map<String, Integer> map = new MyTreeMap<String, Integer>();

for (int i=0; i<n; i++) {
    String uuid = UUID.randomUUID().toString();
    map.put(uuid, 0);
}
```

`UUID` es una clase en el paquete `java.util` que puede generar un "identificador universalmente único" al azar. Los UUIDs son útiles para una variedad de aplicaciones, pero en este ejemplo tomamos ventaja de una forma fácil para generar cadenas al azar.

Ejecuté este código con `n=16384` y medí el tiempo de ejecución y la altura del árbol final. Aquí está la salida:

```
Time in milliseconds = 151
Final size of MyTreeMap = 16384
log base 2 of size of MyTreeMap = 14.0
Final height of MyTreeMap = 33
```

Incluí "log base 2 of size of MyTreeMap" (logaritmo base 2 del tamaño de MyTreeMap) para ver qué tan alto sería el árbol si estuviera lleno. El resultado indica que un árbol lleno con altura 14 contendría 16,384 nodos.

El árbol real de cadenas al azar tiene altura 33, que es sustancialmente mayor a la altura teórica mínima, pero no está mal. Para encontrar una clave en un colección de 16,348, sólo tenemos que hacer 33 comparaciones. Con respecto a una búsqueda lineal, es casi 500 veces más rápida.

Este desempeño es típico para cadenas al azar u otras claves que no agregan en ningún orden particular. La altura final del árbol podría ser 2-3 veces el mínimo teórico, pero todavía es proporcional a $\log n$, que es mucho menor que $n$. De hecho, $\log n$ crece tan lento conforme $n$ se incrementa, que puede ser difícil distinguir tiempo logarítmico de tiempo constante en la práctica.

Sin embargo, los árboles binarios de búsqueda no siempre se comportan tan bien. Veamos lo que sucede cuando agregamos claves en orden ascendente. Aquí está un ejemplo que mide marcas de tiempo en nanosegundos y las usa como claves:

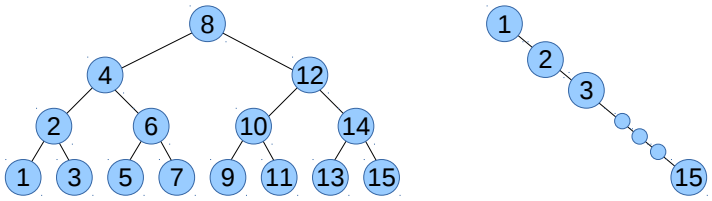


Figura 13.1: Árboles binarios de búsqueda, balanceado (izquierda) y desbalanceado (derecha).

```
MyTreeMap<String, Integer> map = new MyTreeMap<String, Integer>();

for (int i=0; i<n; i++) {
    String timestamp = Long.toString(System.nanoTime());
    map.put(timestamp, 0);
}
```

`System.nanoTime` devuelve un entero de tipo `long` que indica el tiempo transcurrido en nanosegundos. Cada vez que lo llamamos, obtenemos un número más grande. Cuando convertimos estas marcas de tiempo a cadenas, aparecen en orden alfabético ascendente.

Y veamos lo que ocurre cuando lo ejecutamos:

```
Time in milliseconds = 1158
Final size of MyTreeMap = 16384
log base 2 of size of MyTreeMap = 14.0
Final height of MyTreeMap = 16384
```

El tiempo de ejecución es más de siete veces más largo que el del caso anterior. Si te preguntas por qué, mira la altura final del árbol: ¡16384!

Si piensas en cómo funciona `put`, puedes entender lo que ocurre. Cada vez que agregamos una nueva clave, es mayor que todas las demás claves en el árbol, así que siempre elegimos el subárbol derecho y siempre agregamos el nuevo nodo como el hijo derecho del nodo más a la dercha. El resultado es un árbol "desbalanceado" que sólo contiene hijos a la derecha.

La altura de este árbol es proporcional a $n$, no $\log n$, así que el desempeño de `get` y `put` es lineal, no logarítmico.

La figura 13.1 muestra un ejemplo de un árbol balanceado y un árbol desbalanceado. En el árbol balanceado, la altura es 4 y el número total de nodos es $2^4 - 1 = 15$. En el árbol desbalanceado con el mismo número de nodos, la altura es 15.

## 13.6 Árboles auto-balanceables

Hay dos posibles soluciones para este problema:

- Puedes evitar agregar claves al `Map` en orden. Pero esto no siempre es posible.
- Puedes hacer un árbol que haga un mejor trabajo al manejar claves si se da el caso que están en orden.

La segunda solución es mejor y hay varias maneras de hacerlo. La más común es modificar `put` para que detecte cuando el árbol está comenzando a desbalancearse y, de ser así, reacomode los nodos. Los árboles con esta habilidad se llaman "auto-balanceables". Los árboles auto-balanceables comunes incluye el árbol AVL ("AVL" son las iniciales de sus inventores) y el árbol rojo-negro, que es el que el `TreeMap` de Java usa.

En nuestro código de ejemplo, si reemplazamos `MyTreeMap` con el `TreeMap` de Java, los tiempos de ejecución son casi iguales para cadenas aleatorias y para marcas de tiempo. De hecho, las marcas de tiempo se ejecutan incluso más rápido, aunque están en orde, probablemente porque necesitan menos tiempo para codificarse (hash).

En resumen, un árbol binario de búsqueda puede implementar `get` y `put` en tiempo logarítmico, pero solo si las claves se agregan en un orden que mantenga al árbol lo suficientemente balanceado. Los árboles auto-balanceables previenen este problema realizando trabajo adicional cada vez que se agrega una nueva clave.

Puedes leer más sobre árboles auto-balanceables en http://thinkdast.com/balancing.

## 13.7 Un ejercicio más

En el ejercicio anterior no tuviste que implementar `remove`, pero podrías querer intentarlo. Si remueves un nodo de la mitad del árbol, debes reacomodar los nodos restantes para restaurar la propiedad ABB. Probablemente puedas deducir cómo hacer eso por tu cuenta, o puedes leer la explicación en http://thinkdast.com/bstdel.

Remover un nodo y rebalancear un árbol son operaciones similares: si haces este ejercicio, tendrás una mejor idea de cómo funcionan los árboles autobalanceables.

# Capítulo 14

# Persistencia

En los siguientes ejercicios regresaremos a la construcción de un motor de búsqueda web. Para recordar, los componentes de un motor de búsqueda son:

- Rastrear: Necesitaremos un programa que pueda descargar una página, interpretarla y extraer el texto y cualquier enlace a otras páginas.
- Indexar: Necesitaremos un índice que haga posible buscar un término de búsqueda y encontrar las páginas que lo contienen.
- Recuperar: Y necesitaremos una forma de recolectar los resultados del Index e identificar las páginas que son más relevantes para los términos de búsqueda.

Si hiciste el Ejercicio 8.3, implementaste un índice utilizando mapas de Java. En este ejercicio, vamos a revisar el indexador y haremos una nueva versión que guarde los resultados en una base de datos.

Si hiciste el Ejercicio 7.4, construiste un rastreador que siga el primer enlace que encuentra. En el siguiente ejercicio, haremos una versión más general que guarde cada enlace que encuentra en una cola y los explore en orden.

Y luego, finalmente, trabajaremos en el problema de la recuperación.

En estos ejercicios, proveo menos código de inicio, y tendrás que hacer más decisiones de diseño. Estos ejercicios también son más abiertos. Sugeriré algunas metas mínimas que deberías tratar de alcanzar, pero hay muchas formas de lograr mejores resultados si quieres desafiarte a ti mismo.

Ahora, comencemos con la nueva versión del indexador.

## 14.1 Redis

La versión anterior del indexador guarda el índice en dos estructuras de datos: un `TermCounter` que mapea términos de búsqueda con el númerro de veces que aparecen en una página web y un `Index` que mapea un término de búsqueda con el conjunto de páginas donde aparece.

Estas estructuras de datos se guardan en la memoria de un programa de Java en ejecución, lo que significa que cuando el programa se detiene, el índice se pierde. A los datos guardados sólo en la memoria de un programa en ejecución se les llama "volátiles", porque se vaporizan cuando el programa finaliza.

Los datos que persisten después que el programa que los creó finaliza son llamados "persistentes". En general, los archivos guardados en un sistema de archivos son persistentes, así como los datos guardados en bases de datos.

Una forma simple de hacer un dato persistente es guardarlo en un archivo. Antes que el programa finalice, podría traducir sus estructuras de datos en un formato como JSON (http://thinkdast.com/json) y luego escribirlos en un archivo. Cuando inicia de nuevo, podría leer el archivo y reconstruir las estructuras de datos.

Pero hay varios problemas con esta solución:

1. Leer y escribir estructuras de datos de gran tamaño (como un índice Web) sería lento.

2. La estructura de datos completa podría no caber en la memoria de un programa individual en ejecución.

3. Si un programa finaliza de forma inesperada (por ejemplo, debido a un apagón) cualquier cambio hecho desde que el programa se inició por última vez se perdería.

Una mejor alternativa es usar una base de datos que provee almacenamiento persistente y la habilidad de leer y escribir partes de la base de datos sin tener que leer y escribir todo.

Hay muchos tipos de sistemas de adminitración de bases de datos (DBMS) que proveen diferentes características. Puedes leer una introducción en http://thinkdast.com/database.

La base de datos que recomiendo para este ejercicio es Redis, que provee estructuras de datos persistentes que son similares a las estructuras de datos de Java. Específicamente, provee:

- Lista de cadenas, similar a la `List` de Java.
- Hashes, similar al `Map` de Java.
- Conjunto de cadenas, similar al `Set` de Java.

Redis en una "base de datos clave-valor", que significa que las estructuras de datos que contiene (los valores) se identifican con cadenas únicas (las claves). Una clave en Redis juega el mismo rol que una referencia en Java: identifica un objeto. Veremos algunos ejemplos pronto.

## 14.2 Clientes y servidores de Redis

Redis usualmente se ejecuta como un servicio remoto; de hecho, el nombre significa "REmote DIctionary Server" ("Servicio de DIccionario REmoto"). Para usar Redis, tienes que ejecutar el servidor Redis en algún lugar y conectarte a él utilizando un cliente Redis. Hay muchas formas de configurar un servidor y muchos clientes que puedes usar. Para este ejercicio, recomiendo:

1. En lugar de instalar y ejecutar el servidor tú mismo, considera usar un servicio como RedisToGo (http://thinkdast.com/redistogo), que ejecuta Redis en la nube. Ellos ofrecen un plan gratuito con suficientes recursos para este ejercicio.
2. Para el cliente recomiendo Jedis, que es una biblioteca de Java que provee clases y métodos para trabajar con Redis.

Aquí hay instrucciones detalladas para ayudarte a comenzar:

- Crear una cuenta en RedisToGo, en http://thinkdast.com/redissign, y selecciona el plan que quieres (probablemente el plan gratuito para comenzar).
- Crea una "instancia", que es una máquina virtual ejecutando el servidor Redis. Si haces clic en la pestaña "Instancias", deberías ver tu nueva instnacia, identificada con un nombre de host y un número de puerto. Por ejemplo, yo tengo una instancia llamada "dory-10534".
- Haz clic en el nombre de la instancia para ir a la página de configuración. Anota la URL cerca de la página superior de la página, que se ve algo así:

```
redis://redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534
```

Esta URL contiene el nombre de host del servidor, `dory.redistogo.com`, el número de puerto, `10534`, y la constraseña que necesitarás para conectarte al servidor, que es la cadena larga de letras y números en el medio. Necesitarás esta información para el siguiente paso.

## 14.3 Hacer un índice respaldado por Redis

En el repositorio para esta libro, encontrarás el código fuente para este ejercicio:

- `JedisMaker.java` contiene código de ejemplo para conectarse a un servidor Redis y ejecutar algunos métodos de Jedis.
- `JedisIndex.java` contiene código inicial para este ejercicio.
- `JedisIndexTest.java` contiene código de pruebas para `JedisIndex`.
- `WikiFetcher.java` contiene el código que vimos en los ejercicios anteriores para ler páginas web e interpretarlas usando jsoup.

También necesitarás estos archivos, en los que trabajaste en ejercicios anteriores:

- `Index.java` implementa un índice usando estructuras de datos de Java.
- `TermCounter.java` representa un mapa de los términos con sus frecuencias.
- `WikiNodeIterable.java` itera por los nodos en un árbol DOM producido por jsoup.

Si tienes versiones funcionales de estos archivos, puedes usarlas para este ejercicio. Si no hiciste los ejercicios anteriores, o si no confías en tus soluciones, puedes copiar mis soluciones desde el directorio `solutions`.

El primer paso es usar Jedis para conectarte a tu servidor Redis. `RedisMaker.java` muestra cómo hacer esto. Este archivo lee información sobre tu servidor Redis de un archivo, se conecta a él e inicia sesión usando tu contraseña, luego devuelve un objeto `Jedis` que puedes usar para realizar operaciones en Redis.

Si abres `JedisMaker.java`, deberías ver la clase `JedisMaker`, que es una clase auxiliar que provee un método estático, `make`, la cual crea un objeto `Jedis` object. Una vez este objeto se ha autenticado, puedes usarlo para comunicarte con tu base de datos Redis.

`JedisMaker` lee información sobre tu servidor redis de un archivo llamado `redis_url.txt`, que debes colocar en el directorio `src/resources`:

- Usa un editor de texto para crear y editar `ThinkDataStructures/code/src/resources/r`
- Pega la URL de tu servidor. Si estás usando RedisToGo, la URL se verá así:

  `redis://redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534`

Porque este archivo contiene la contraseña de tu servidor Redis, no deberías ponerlo en un repositorio publico. Para ayudarte a prevenir hacerlo por accidente, el repositorio contiene un archivo `.gitignore` que hace más difícil (pero no imposible) poner este archivo en tu repo.

Ahora ejecuta `ant build` para compilar los archivos de código fuente y `ant JedisMaker` para ejecutar el código de ejemplo en `main`:

```
    public static void main(String[] args) {

        Jedis jedis = make();

        // String
        jedis.set("mykey", "myvalue");
        String value = jedis.get("mykey");
        System.out.println("Got value: " + value);

        // Set
        jedis.sadd("myset", "element1", "element2", "element3");
        System.out.println("element2 is member: " +
                           jedis.sismember("myset", "element2"));

        // List
        jedis.rpush("mylist", "element1", "element2", "element3");
        System.out.println("element at index 1: " +
                           jedis.lindex("mylist", 1));

        // Hash
        jedis.hset("myhash", "word1", Integer.toString(2));
        jedis.hincrBy("myhash", "word2", 1);
        System.out.println("frequency of word1: " +
                           jedis.hget("myhash", "word1"));
        System.out.println("frequency of word1: " +
                            jedis.hget("myhash", "word2"));

        jedis.close();
    }
```

Este ejemplo demuestra los tipos de datos y métodos que probablemente uses en este ejercicio. Cuando lo ejecutes, la salida debería ser:

```
Got value: myvalue
element2 is member: true
element at index 1: element2
frequency of word1: 2
frequency of word2: 1
```

En la siguiente sección, explicaré cómo funciona el código.

## 14.4 Tipos de datos de Redis

Redis es básicamente un mapa de las claves, que son cadenas, a valores, que pueden ser uno de varios tipos de datos. El tipo de datos más básico en Redis es una *string*. Escribiré los tipos de Redis en cursiva para distinguirlos de los tipos de Java.

Para agregar una *string* a la base de datos, usaremos `jedis.set`, que es similar a `Map.put`; los parámetros son la nueva clave y el valor correspondiente. Para buscar una clave y obtener su valor, usamos `jedis.get`:

```
jedis.set("mykey", "myvalue");
String value = jedis.get("mykey");
```

En este ejemplo, la clave es `"mykey"` y el valor es `"myvalue"`.

Redis provee una estructura *set*, similar a un `Set<String>` de Java. Para añadir elementos a un *set* de Redis, eliges una clave para identificar el *set* y luego usas `jedis.sadd`:

```
jedis.sadd("myset", "element1", "element2", "element3");
boolean flag = jedis.sismember("myset", "element2");
```

No tienes que crear el *set* en un paso separado. Si no existe, Redis lo crea. En este caso, crea un *set* llamado `myset` que contiene tres elementos.

El método `jedis.sismember` comprueba si un elemento está en un *set*. Agregar elementos y comprobar la membresía son operaciones de tiempo constante.

Redis también provee una estructura *list* similar a una `List<String>` de Java. El método `jedis.rpush` agrega elementos al final (lado derecho) de una *list*:

```
jedis.rpush("mylist", "element1", "element2", "element3");
String element = jedis.lindex("mylist", 1);
```

De nuevo, no tienes que crear la estrucutura antes de empeza a agregar elementos. Este ejemplo crea una *list* llamada “mylist” que contiene tres elementos.

El método `jedis.lindex` toma un índice entero y devuelve el elemento indicado de una *list*. Agregar y acceder elementos son operaciones de tiempo constante.

Finalmente, Redis provee una estructura *hash*, similar a un `Map<String, String>` de Java. El método `jedis.hset` añade una nueva entrada al *hash*:

```
jedis.hset("myhash", "word1", Integer.toString(2));
String value = jedis.hget("myhash", "word1");
```

Este ejemplo crea un *hash* llamado `myhash` que contiene una entrada, la cual mapea la clave `word1` con el valor `"2"`.

Las claves y los valores son *string*s, así que si queremos guardar un `Integer`, tenemos que convertirlo a una `String` antes de llamar a `hset`. Y cuando busquemos el valor usando `hget`, el resultado es un `String`, así que puede que tengamos que convertirlo de vuelta a un `Integer`.

Trabajar con los *hash*es de Redis puede ser confuso, porque usamos una clave para identificar el *hash* que queremos, y luego otra clave para identificar un valor en el *hash*. En el contexto de Redis, la segunda clave es llamada un "campo", que podría ayudarnos a mantener las cosas en orden. Así que una "clave" como `myhash` identifica un *hash* particular, y luego un "campo" como `word1` identifica un valor en el *hash*.

Para muchas aplicaciones, los valores en un *hash* de Redis son enteror, por lo que Redis provee unos cuantos métodos especiales, como `hincrby`, que tratan a los valores como números:

```
jedis.hincrBy("myhash", "word2", 1);
```

Este método accede a `myhash`, obtiene el valor actual asociado con `word2` (o 0 si no existe), lo incrementa en 1, y escribe el resultado de regreso en el *hash*.

Actualizar, obtener e incrementar entradas en un *hash* son operaciones de tiempo constante.

Puedes leer más sobre los tipos de datos de Redis en http://thinkdast.com/redistypes.

## 14.5 Ejercicio 11

En este punto, ya tienes la información que necesitas para hacer un índice de búsqueda web que guarde los resultados en una base de datos Redis.

Ahora ejecuta `ant JedisIndexTest`. Debería fallar, ¡porque tienes algo de trabajo por hacer!

`JedisIndexTest` prueba estos métodos:

- `JedisIndex`, que es el constructor que toma un objeto `Jedis` como parámetro.
- `indexPage`, que agrega una página Web al índice; toma una URL de tipo `String` y un objeto `Elements` de jsoup que contiene los elementos de la página que deberían ser indexados.
- `getCounts`, que toma un término de búsqueda y devuelve un `Map<String, Integer>` que mapea cada URL que contiene el término de búsqueda con el número de veces que éste aparece en dicha página.

Aquí está un ejemplo de cómo usar estos métodos:

```
WikiFetcher wf = new WikiFetcher();
String url1 =
    "http://en.wikipedia.org/wiki/Java_(programming_language)";
Elements paragraphs = wf.readWikipedia(url1);

Jedis jedis = JedisMaker.make();
JedisIndex index = new JedisIndex(jedis);
index.indexPage(url1, paragraphs);
Map<String, Integer> map = index.getCounts("the");
```

Si buscamos `url1` en el resultado, `map`, deberíamos obtener 339, que es el número de veces que la palabra "the" aparece en la página de Wikipedia para Java (es decir, en la versión que guardamos).

Si indexamos la misma página de nuevo, los nuevos resultados deberían reemplazar a los antiguos.

Una sugerencia para traducir estructuras de datos de Java a Redis: recuerda que cada objeto en una base de datos Redis se identifica con una clave única, que es una *string*. Si tienes dos clases de objetos en la misma base de datos, podrías querer agregar un prefijo a las claves para distinguirlos. Por ejemplo, en nuestra solución, tenemos dos clases de objetos:

- Definimos un `URLSet` como un *set* de Redis que contiene las URLs que contienen un término de búsqueda dado. La clave para cada `URLSet` comienza con `"URLSet:"`, así que para obtener las URLs que contienen la palabra "the", accedemos al *set* con la clave `"URLSet:the"`.

- Definimos un `TermCounter` como un *hash* de Redis que mapea cada término que aparece en una página con el número de veces que aparece. La clave para cada `TermCounter` comienza con `"TermCounter:"` y finaliza con la URL de la página que buscamos.

En mi implementación, hay un `URLSet` para cada término y un `TermCounter` para cada página indexada. Proveo dos métodos auxiliares, `urlSetKey` y `termCounterKey`, para juntar estas claves.

## 14.6 Más sugerencias si las quieres

Llegados a este punto tienes toda la información que necesitas para hacer el ejercicio, así que puedes iniciar si estás listo. Pero tengo unas cuantas sugerencias que podrías querer leer primero:

- Para este ejercicio proveo menos guía que para los ejercicios precios. Tendrás que tomar algunas decisiones de diseño, en particular, tendrás que ingeniártelas para dividir el problema en partes que puedes probar una por una, y luego ensamblar las piezas en una solución completa. Si tratas de escribir todo de una sola vez, sin probar fragmentos pequeños, la depuración podría tomarte un largo tiempo.

- Uno de los desafíos de trabajar con datos persistentes es que son persistentes. Las estructuras guardadas en la base de datos podrían cambiar cada vez que ejecutes el programa. Si arruinas algo en la base de datos, tendrás que arreglarlo o comenzar de cero antes de proceder. Para ayudarte a mantener las cosas bajo control, he provisto métodos llamados `deleteURLSets`, `deleteTermCounters`, y `deleteAllKeys`, que puedes usar para limpiar la base de datos y comenzar de nuevo. También puedes usar `printIndex` para imprimir los contenidos de un índice.

- Cada vez que invoques un método de `Jedis`, tu cliente envía un mensaje al servidor, luego el servidor realiza la acción que le pediste y envía un mensaje de vuelta. Si realizas muchas operaciones pequeñas, probablemente se demore mucho. Puedes mejorar el desempeño al agrupar un serie de operaciones en una `Transaction`.

Por ejemplo, aquí está una versión simple de `deleteAllKeys`:

```
public void deleteAllKeys() {
    Set<String> keys = jedis.keys("*");
    for (String key: keys) {
        jedis.del(key);
    }
}
```

Cada vez que invocas a `del` se requiere un viaje de ida y vuelta del cliente al servidor. Si el índice contiene más que unas cuantas páginas, este método utilizará una larga cantidad de tiempo para ejecutarse. Podemos acelerarlo con un objeto `Transaction`:

```
public void deleteAllKeys() {
    Set<String> keys = jedis.keys("*");
    Transaction t = jedis.multi();
    for (String key: keys) {
        t.del(key);
    }
    t.exec();
}
```

`jedis.multi` devuelve un objeto `Transaction`, que provee todos los métodos de un objeto `Jedis`. Pero cuando invocas un método en una `Transaction`, la operación no se ejecuta de inmediato y no se comunica con el servidor. En su lugar, almacena un lote de operaciones hasta que invocas `exec`. Luego envía todas las operaciones almacenadas al servidor al mismo tiempo, lo que usualmente es mucho más rápido.

## 14.7 Unos cuantos tips de diseño

Ahora sí *realmente* tienes todas la información que necesitas; deberías comenzar a trabajar en el ejercico. Pero si te atascas, o si de verdad no sabes cómo comenzar, puedes regresar por más pistas.

**No leas lo siguientes hasta que hayas ejecutado el código de pruebas, probado algunos comandos básicos de Redis y escrito unos cuantos métodos en `JedisIndex.java`.**

OK, si realmente estás atascado, aquí están algunos métodos en los que podrías querer trabajar:

```
/**
 * Agrega una URL al set asociado con el término.
 */
public void add(String term, TermCounter tc) {}

/**
 * Busca un término de búsqeuda y devuelve un set de URLs.
 */
public Set<String> getURLs(String term) {}

/**
 * Devuelve el número de veces que el término dado
 * aparece en la URL proporcionada.
 */
public Integer getCount(String url, String term) {}

/**
 * Agrega los contenidos del TermCounter a Redis.
 */
public List<Object> pushTermCounterToRedis(TermCounter tc) {}
```

Estos son los métodos que usé en mi solución, pero ciertamente no son la única manera de dividir el trabajo. Así que por favor toma estas sugerencias si te sirven, pero ignóralas si no.

Para cada método, considera escribir los tests primero. Cuando decides cómo probar un método, muchas veces se te ocurren ideas para escribirlo.

¡Buena suerte!

# Capítulo 15

# Rastreo de Wikipedia

En este capítulo, presento una solución al ejercicio previo y analizo el desempeño de los algoritmos de indexación Web. Luego construimos un rastreador Web simple.

## 15.1 El indexador respaldado por Redis

En mi solución, guardamos dos clases de estructuras en Redis:

- Para cada término de búsqueda, tenemos un `URLSet`, que es un *set* de Redis de URLs que contienen el término de búsqueda.
- Para cada URL, tenemos un `TermCounter`, que es un *hash* de Redis que mapea cada término de búsqueda con el número de veces que aparece.

Discutimos estos tipos de datos en el capítulo previo. También puedes leer sobre la estructuras de Redis en http://thinkdast.com/redistypes

En `JedisIndex`, proveo métodos que toman un término de búsqueda y devuelven la clave Redis de su `URLSet`:

```
private String urlSetKey(String term) {
    return "URLSet:" + term;
}
```

Y un método que toma una URL y devuelve la clave Redis de su `TermCounter`:

```
private String termCounterKey(String url) {
    return "TermCounter:" + url;
}
```

Aquí está la implementación de `indexPage`, que toma una URL y un objeto which takes a URL and a `Elements` de jsoup que contiene el árbol DOM de los párrafos que queremos indexar:

```
public void indexPage(String url, Elements paragraphs) {
    System.out.println("Indexing " + url);

    // make a TermCounter and count the terms in the paragraphs
    TermCounter tc = new TermCounter(url);
    tc.processElements(paragraphs);

    // push the contents of the TermCounter to Redis
    pushTermCounterToRedis(tc);
}
```

Para indexar una página,

1. Creamos un `TermCounter` de Java para los contenidos de la página, usando el código del ejercicio anterior.

2. Agregamos los contenidos del `TermCounter` a Redis.

Aquí está el nuevo código que agrega un `TermCounter` a Redis:

```
public List<Object> pushTermCounterToRedis(TermCounter tc) {
    Transaction t = jedis.multi();

    String url = tc.getLabel();
    String hashname = termCounterKey(url);

    // si esta página ya fue indexada, borra el hash anterior
    t.del(hashname);

    // para cada término de búsqueda, agrega una entrada en el TermCounter
```

```
    // y un nuevo miembro del índice
    for (String term: tc.keySet()) {
        Integer count = tc.get(term);
        t.hset(hashname, term, count.toString());
        t.sadd(urlSetKey(term), url);
    }
    List<Object> res = t.exec();
    return res;
}
```

Este método usa una `Transaction` para recolectar las operaciones y enviarlos al servidor a la vez, que es mucho más rápido que enviar una serie de operaciones pequeñas.

El método itera por los término del `TermCounter`. Para cada uno

1. Encuentra o crea un `TermCounter` en Redis, luego agrega un campo para el nuevo término.

2. Encuentra o crea un `URLSet` en Redis, luego agrega la URL actual.

Si la página ya ha sido indexada, borramos su `TermCounter` antes de agregar los nuevos contenidos..

Eso es todo lo relacionado con la indexación de nuevas páginas.

La segunda parte del ejercicio te pedía escribir `getCounts`, que toma un término de búsqueda y devuelve un mapa de cada URL donde el término aparece con el número de veces que aparece ahí. Aquí está mi solución:

```
    public Map<String, Integer> getCounts(String term) {
        Map<String, Integer> map = new HashMap<String, Integer>();
        Set<String> urls = getURLs(term);
        for (String url: urls) {
            Integer count = getCount(url, term);
            map.put(url, count);
        }
        return map;
    }
```

Este método usa dos métodos auxiliares:

- `getURLs` toma un término de búsqueda y devuelve el Set de URLs donde el término aparece.

- `getCount` toma una URL y un término y devuelve el número de veces que el término aparece en la URL dada.

Aquí están las implementaciones:

```
public Set<String> getURLs(String term) {
    Set<String> set = jedis.smembers(urlSetKey(term));
    return set;
}

public Integer getCount(String url, String term) {
    String redisKey = termCounterKey(url);
    String count = jedis.hget(redisKey, term);
    return new Integer(count);
}
```

Debido a la manera en que diseñamos el índice, estos métodos son simples y eficientes.

## 15.2 Análisis de la búsqueda

Supón que hemos indexado $N$ páginas y descubierto $M$ términos de búsqueda únicos. ¿Cuánto tiempo requerirá la búsqueda de un término de búsqueda? Piensa en tu respuesta antes de continuar.

Para buscar un término de búsqueda, ejecutamos `getCounts`, que

1. Crea un mapa.

2. Ejecuta `getURLs` para obtener un Set de URLs.

3. Para cada URL en el Set, ejecuta `getCount` y agrega una entrada a la `HashMap`.

`getURLs` toma tiempo proporcinal al número de URLs que contiene el término de búsqueda. Para términos raros, podría ser un número pequeño, pero para términos comunes podría ser tan grande como $N$.

Dentro del bucle, ejecutamos `getCount`, que encuentra un `TermCounter` en Redis, busca un término y agrega una entrada a un HashMap. Estos son todas operaciones de tiempo constante, así que la complejidad global de `getCounts` es $O(N)$ en el peor caso. Sin embargo, en la práctica el tiempo de ejecución es proporcional al número de páginas que contienen el término, que normalmente es mucho menor que $N$.

Este algoritmo es tan eficiente como puede ser, en términos de la complejidad algorítmica, pero es muy lento porque envía muchas operaciones pequeñas a Redis. Puedes hacerlo más rápido usando una `Transaction`. Podrías querer hacerlo como ejercicio, o puedes ver mi solución in `RedisIndex.java`.

## 15.3 Análisis de la indexación

Usando las estructuras de datos que diseñamos, ¿cuánto tiempo tomaría indexar una página? De nuevo, piensa en tu respuesta antes de continuar.

Para indexar una página, recorremos su árbol DOM, encontramos todos los objetos `TextNode`, y separamos las cadenas en términos de búsqueda. Todo eso toma tiempo proporcional al número de palabras en la página.

Para cada término, incrementamos un contador en un HashMap, que es una operación de tiempo constante. Así que hacer el `TermCounter` toma tiempo proporcional al número de palabras en la página.

Agregar el `TermCounter` a Redis requiere borrar un `TermCounter`, que es lineal en el número de términos únicos. Luego para cada término tenemos que

1. Agregar un elemento a un `URLSet`, y
2. Agrega un elemento a un `TermCounter` de Redis.

Ambas son operaciones de tiempo constante, así que el tiempo total para agregar el `TermCounter` es lineal en el número de términos de búsqueda únicos.

En resumen, hacer el `TermCounter` es proporcional al número de palabras en la página. Agregar el `TermCounter` a Redis es proporcional al número de términos únicos.

Dado que el número de palabras en la página usualmente excede el número de términos de búsqueda únicos, la complejidad global es proporcional al número de palabras en la página. En teoría una página podría contener todos los términos de búsqueda en el índice, así que el peor escenario de desempeño es $O(M)$, pero no esperamos ver el peor escenario en la práctica.

Este análisis sugiere una forma de mejorar el desempeño: deberíamos probablemente evitar indexar palabras muy comunes. Primero, toman mucho tiempo y espacio, porque aparecen en cada `URLSet` y `TermCounter`. Además, no son muy útiles porque no ayudan a identificar páginas relevantes.

La mayoría de motores de búsqueda evitan indexar palabras comunes, que se conocen en este contexto como palabras vacías o stop words (http://thinkdast.com/stopword).

## 15.4 Recorrido de grafos

Si hiciste el ejercicio "Llegar a la Filosofía" en el Capítulo 7, ya tienes un programa que lee una páginas de Wikipedia, encuentra el primer enlace, usa el enlace para cargar la siguiente páginas y repite el proceso. Este programa es un tipo especializado de rastreador, pero cuando las personas dicen "rastreador Web" (o Web crawler), usualmente se refieren a un programa que

- Carga una página inicial e indexa los contenidos,
- Encuentra todos los enlaces en la página y agrega las URLs enlazadas a una colección y
- Navega por la colección, carga las páginas, las indexa y agrega nuevas URLs.
- Si encuentra una URL que ya ha sido indexada, la omite.

Puedes pensar en la Web como un grafo donde cada página es un nodo y cada enlace es una arista dirigida de un nodo a otro. Si no estás familiarizado con los grafos, puedes leer sobre ellos en http://thinkdast.com/graph.

Comenzando con un nodo origen, un rastreador recorre este grafo visitando cada nodo alcanzable una vez.

La colección que usamos para guardar las URLs determina qué clase de recorrido realiza el rastreador:

- Si es una cola primero en entrar, primero en salir (PEPs), el rastreador realiza un recorrido en anchura.
- Si es una pila último en entrar, primero en salir (UEPS), el rastreador realiza un recorrido en profundidad.
- De forma más general, los elementos en la colección podrían ser priorizados. Por ejemplo, podríamos querer dar una prioridad mayor a las páginas que no han sido indexadas por un largo tiempo.

Puedes leer más sobre el recorrido de grafos en http://thinkdast.com/graphtrav.

## 15.5 Ejercicio 12

Ahora es el momento de escribir el rastreador. En el repositorio para este libro, encontrarás los archivos de código fuente para este ejercicio:

- `WikiCrawler.java`, que contiene código inicial para tu rastreador.
- `WikiCrawlerTest.java`, que contiene código de pruebas para `WikiCrawler`.
- `JedisIndex.java`, que es mi solución al ejercicio previo.

También necesitarás algunas de las clases auxiliares que hemos usado en ejercicios previos:

- `JedisMaker.java`
- `WikiFetcher.java`

- `TermCounter.java`
- `WikiNodeIterable.java`

Antes de ejecutar `JedisMaker`, debes proveer un archivo con información sobre tu servidor Redis. Si hiciste esto en el ejercicio previo, deberías estar listo. De lo contrario puedes encontrar instrucciones en la Sección 14.3.

Ejecuta `ant build` para compilar el código fuente, luego ejecuta `ant JedisMaker` para asegurarte que está configurado para conectarse a tu servidor Redis.

Ahora ejecuta `ant WikiCrawlerTest`. Debería fallar, ¡porque tienes trabajo por hacer!

Aquí está el principio de la clase `WikiCrawler` que proveí:

```
public class WikiCrawler {

    public final String source;
    private JedisIndex index;
    private Queue<String> queue = new LinkedList<String>();
    final static WikiFetcher wf = new WikiFetcher();

    public WikiCrawler(String source, JedisIndex index) {
        this.source = source;
        this.index = index;
        queue.offer(source);
    }

    public int queueSize() {
        return queue.size();
    }
}
```

Las variables de instancia son

- `source` es la URL desde donde comenzamos el rastreo.
- `index` es el `JedisIndex` adonde van los resultados.

- `queue` es una `LinkedList` donde llevamos el control de las URLs que hemos descubierto pero aún no hemos indexado.
- `wf` es el `WikiFetcher` que usaremos para leer e interpretar páginas Web.

Tu trabajo es llenar `crawl`. Aquí está el prototipo:

```
public String crawl(boolean testing) throws IOException {}
```

El parámetro `testing` será `true` cuando este método sea llamado desde `WikiCrawlerTest` y debería ser `false` en caso contrario.

Cuando `testing` es `true`, el método `crawl` debería:

- Elegir y remover una URL de la cola en orden PEPS.
- Leer los contenidos de la página usando `WikiFetcher.readWikipedia`, que lee copias en caché de páginas incluidad en el repositorio para propósitos de pruebas (para evitar problemas si la versión de Wikipedia cambia).
- Debería indexar pagínas sin importar si ya han sido indexadas.
- Debería encontrar todos los enlaces internos en la página y agregarlos a la cola en el orden en que aparecen. Los "enlaces internos" son enlaces a otras páginas de Wikipedia.
- Y debería devolver la URL de la página que indexó.

Cuando `testing` es `false`, este método debería:

- Elegir y remover una URL de la cola en orden PEPS.
- Si la URL ya está indexada, no debería indexarla de nuevo, y debería devolver `null`.
- De lo contrarior debería leer los contenidos de la página usando `WikiFetcher.fetchWikipe` que lee el contenido actual de la Web.
- Luego debería indexar la página, agregar los enlaces a la cola y devolver la URL de la página que indexó.

`WikiCrawlerTest` carga la cola con alrededor de 200 enlaces y luego invoca a `crawl` tres veces. Después de cada invoación, comprueba el valor de retorno y la nueva longitud de la cola.

Cuando tu rastreador trabaje como se especificó, este test debería pasarse. ¡Buena suerte!

# Capítulo 16

# Búsqueda booleana

En este capítulo presento una solución al ejercicio previo. Luego escribirás código para combinar múltiples resultados de búsqueda y ordenarlos de acuerdo a su relevancia con respecto a los términos de búsqueda.

## 16.1 Solución del rastreador

Primero, revisemos nuestra solución al ejercicio anterior. Proveí un boceto de un `WikiCrawler`; tu trabajo era llenar `crawl`. Como recordatorio, aquí están los campos de la clase `WikiCrawler`:

```
public class WikiCrawler {
    // keeps track of where we started
    private final String source;

    // the index where the results go
    private JedisIndex index;

    // queue of URLs to be indexed
    private Queue<String> queue = new LinkedList<String>();

    // fetcher used to get pages from Wikipedia
    final static WikiFetcher wf = new WikiFetcher();
}
```

Cuando creamos un `WikiCrawler`, proveemos `source` e `index`. Inicialmente, `queue` contiene solo un elemento, `source`.

Nota que la implementación de `queue` es una `LinkedList`, así que podemos agregar elementos al final — y removerlos del principio — en tiempo constante. Al asignar un objeto `LinkedList` a una variable de tipo `Queue`, nos limitamos a usuar los métodos en la interfaz `Queue`; específicamente, usaremos `offer` para agregar elementos y `poll` para removerlos.

Aquí está mi implementación de `WikiCrawler.crawl`:

```
public String crawl(boolean testing) throws IOException {
    if (queue.isEmpty()) {
        return null;
    }
    String url = queue.poll();
    System.out.println("Crawling " + url);

    if (testing==false && index.isIndexed(url)) {
        System.out.println("Already indexed.");
        return null;
    }

    Elements paragraphs;
    if (testing) {
        paragraphs = wf.readWikipedia(url);
    } else {
        paragraphs = wf.fetchWikipedia(url);
    }
    index.indexPage(url, paragraphs);
    queueInternalLinks(paragraphs);
    return url;
}
```

La mayoría de la complejidad de este método está ahí para hacerlo más fácil de probar. Aquí está la lógica:

- Si la cola (queue) está vacía, devuelve `null` para indicar que no indexó una página.

- De lo contrario remueve y guarda la siguiente URL de la cola.
- Si la URL ya ha sido indexada, `crawl` no la indexa de nuevo, a menos que esté en modo de pruebas.
- A continuación lee los contenidos de la página: si está en modo de pruebas, los lee de un archivo; de lo contrario, los lee de la Web.
- Indexa la página.
- Intepreta la página y agrega los enlaces internos a la cola.
- Finalmente, devuelve la URL de la página que indexó.

Presenté una implementación de `Index.indexPage` en la Sección 15.1. Así que el único método nuevo es `WikiCrawler.queueInternalLinks`.

Escribí dos versiones de este método con diferentes parámetros: uno toma un objeto `Elements` conteniendo un árbol DOM por párrafo; el otro toma un objeto `Element` que contiene un único párrafo.

La primera versión solo itera por los párrafos. La segunda versión es la que hace realmente el trabajo.

```
void queueInternalLinks(Elements paragraphs) {
    for (Element paragraph: paragraphs) {
        queueInternalLinks(paragraph);
    }
}

private void queueInternalLinks(Element paragraph) {
    Elements elts = paragraph.select("a[href]");
    for (Element elt: elts) {
        String relURL = elt.attr("href");

        if (relURL.startsWith("/wiki/")) {
            String absURL = elt.attr("abs:href");
            queue.offer(absURL);
        }
    }
}
```

Para determinar si un enlace es "interno," comprobamos si la URL comienza con "/wiki/". Esto podría incluir algunas páginas que no queremos indexa, como meta-páginas sobre Wikipedia. Y podría exluir algunas páginas que queremos, como enlaces a páginas en idiomas diferentes al inglés. Pero esta prueba simple es lo suficientemente buena para empezar.

Eso es todo. Este ejercició no tenía mucho material nuevo; era mayormente para combinar todas las piezas del rompecabezas.

## 16.2 Recuperación de información

La siguiente fase de este proyecto es implementar una herramienta de búsqueda. Las partes que necesitaremos incluir son:

1. Una interfaz donde los usuarios puedan proveer términos de búsqueda y ver los resultados.
2. Un mecanismo de búsqueda que tome cada término de búsqueda y devuelva las páginas que lo contienen.
3. Mecanismos para combinar los resultados de búsqueda de múltiples términos de búsqueda.
4. Algoritmos para clasificar y ordenar los resultados de búsqueda.

El término general para procesos como este es "recuperación de información", sobre el que puedes más en `http://thinkdast.com/infret`.

En este ejercicio, nos enfocaremos en los pasos 3 y 4. Ya hemos construido una versión simple del 2. Si estás interesado en construir aplicaciones Web, podrías considerar trabajar en el paso 1.

## 16.3 Búsqueda boolena

La mayoría de los motores de búsqueda pueden realizar "búsquedas booleanas", lo que significa que puedes combinar los resultados de múltiples términos de búsqueda usando lógica booleana. Por ejemplo:

- La búsqueda "java AND programación" podría devolver sólo las páginas que contienen ambos términos de búsqueda: "java" y "programación".

- "java OR programación" podría devolver páginas que contienen cualquiera de los términos, pero no necesariamente ambos.

- "java -indonesia" podría devolver las páginas que contienen "java" y no contienen "indonesia".

Las expresiones como estas que contienen términos de búsqueda y operadores se conocen como "consultas".

Cuando se aplican a resultados de búsqueda, los operadores booleanos `AND`, `OR`, y `-` corresponden a las operaciones de conjuntos `intersection` (intersección), `union` (unión), y `difference` (diferencia). Por ejemplo, supón:

- `s1` es el conjunto de páginas que contienen "java",

- `s2` es el conjunto de páginas que contienen "programación", y

- `s3` es el conjunto de páginas que contienen "indonesia".

En ese caso:

- La intersección de `s1` y `s2` es el conjunto de páginas que contienen "java" AND "programación".

- La unión de `s1` y `s2` es el conjunto de páginas que contienen "java" OR "programación".

- La diferencia de `s1` y `s2` es el conjunto de páginas que contienen "java" y no "indonesia".

En la siguiente sección escribirás un método para implementar estas operaciones.

## 16.4 Ejercicio 13

En el repositorio para este libro encontrarás los archivos de código fuente para este ejercicio:

- `WikiSearch.java`, que define un objeto que contiene resultados de búsqueda y realiza operaciones en ellos.
- `WikiSearchTest.java`, que contiene código de pruebas para `WikiSearch`.
- `Card.java`, que demuestra cómo usar el método `sort` en `java.util.Collections`.

También encontrarás algunas de las clases auxiliares que hemos usado en los ejercicios anteriores.

Aquí está el principio de la definición de la clase `WikiSearch`:

```
public class WikiSearch {

    // map from URLs that contain the term(s) to relevance score
    private Map<String, Integer> map;

    public WikiSearch(Map<String, Integer> map) {
        this.map = map;
    }

    public Integer getRelevance(String url) {
        Integer relevance = map.get(url);
        return relevance==null ? 0: relevance;
    }
}
```

Un objeto `WikiSearch` contiene un mapap de URLs con su puntaje de relevancia. En el contexto de la recuperación de información, un "puntaje de relevancia" que pretende indicar qué tan bien una página satisface las necesidades de un usuario como se infieren de la consulta. Hay muchas formas de construir un puntaje de relevancia, pero la mayoría de ellos están basadas en la "frecuencia de término", que es el número de veces que los términos de búsqueda aparecen en la página. Un puntaje de relevancia común es llamado

TF-IDF, que significa "frecuencia de término – frecuencia inversa de documento"("term frequency – inverse document frequency"). Puedes leer más sobre al respecto en http://thinkdast.com/tfidf.

Tendrás la opción de implementar TF-IDF más tarde, pero comenzaremos con algo incluso más simple, TF:

- Si una consulta contiene un único término de búsqueda, la relevancia de una página es su frecuencia de término, es decir, el número de veces que el término aparece en la página.
- Para consultas con múltiples términos, la relevancia de una página es la suma de las frecuencias de término; es decir, el número total de veces que cualquiera de los términos de búsqueda aparece.

Ahora estás listo para comenzar el ejercicio. Ejecuta `ant build` para compilar los archivos fuentes, luego ejecuta `ant WikiSearchTest`. Como es usual, debería fallar, porque tienes trabajo por hacer.

En `WikiSearch.java`, llena el cuerpo de `and`, `or`, y `minus` para que pasen los tests relevantes. No tienes que preocuparte por `testSort` todavía.

Puedes ejecutar `WikiSearchTest` sin usar Jedis porque no depende del índice en tu base de datos Redis. Pero si quieres ejecutar una consulta contra tu índice, tienes que proveer un archivo con información sobre tu servidor Redis. Mira la Sección 14.3 para más detalles.

Ejecuta `ant JedisMaker` para asegurarte que está configurado para conectarse a tu servidor Redis. Luego ejecuta `WikiSearch`, que imprime los resultados de tres consultas (los términos de búsqueda están en inglés):

- "java"
- "programming"
- "java AND programming"

Inicialmente los resultados no estarán en ningun orden particular, porque `WikiSearch.sort` está incompleto.

Llena el cuerpo de `sort` para que los resultados se devuelvan en orden ascendente de relevancia. Sugiero que uses el método `sort` provisto por `java.util.Collections`, que ordena cualquier clase de `List`. Puedes leer la documentación en http://thinkdast.com/collections.

Hay dos versiones de `sort`:

- La versión con un parámetro toma una lista y ordena los elementos usando el método `compareTo`, así que los elementos tienen que ser `Comparable`.
- La versión con dos parámetros toma una lista de cualquier tipo de objeto y un `Comparator`, que es un objeto que provee un método `compare` que compara elementos.

Si no estás familiarizado con las interfaces `Comparable` y `Comparator`, las explicaré en la siguiente sección.

## 16.5 Comparable y Comparator

El repositorio para este libro incluye `Card.java`, que demuestra dos formas de ordenar una lista de objetos `Card`. Aquí está el principio de la definición de la clase:

```
public class Card implements Comparable<Card> {

    private final int rank;
    private final int suit;

    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
    }
```

Un objeto `Card` tiene dos campos enteros, `rank` y `suit`. `Card` implementa `Comparable<Card>`, lo que significa que provee `compareTo`:

```
public int compareTo(Card that) {
    if (this.suit < that.suit) {
        return -1;
    }
    if (this.suit > that.suit) {
        return 1;
    }
    if (this.rank < that.rank) {
        return -1;
    }
    if (this.rank > that.rank) {
        return 1;
    }
    return 0;
}
```

La especificación de `compareTo` indica que debería devolver un número negativo si `this` se considera menor que `that`, un número positivo si se considera mayor y 0 si se consideran iguales.

Si usas la versión con un parámetro de `Collections.sort`, se usa el método `compareTo` provisto por los elementos para ordenarlos. Para demostrarlo, podemos hacer una lista de 52 cartas como esta:

```
public static List<Card> makeDeck() {
    List<Card> cards = new ArrayList<Card>();
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            Card card = new Card(rank, suit);
            cards.add(card);
        }
    }
    return cards;
}
```

Y ordenarla de esta forma:

```
    Collections.sort(cards);
```

Esta versión de `sort` pone los elementos en lo que se llama su "orden natural" porque lo determinan los mismos objetos.

Pero es posible imponer un orden diferente al proveer un objeto `Comparator`. Por ejemplo, el orden natural de los objetos `Card` trate los Ases como las cartas de menor rango, pero en algunos juegos tienen el rango más alto. Podemos definir un `Comparator` que considere a los "Ases con un rango alto", como este:

```
Comparator<Card> comparator = new Comparator<Card>() {
    @Override
    public int compare(Card card1, Card card2) {
        if (card1.getSuit() < card2.getSuit()) {
            return -1;
        }
        if (card1.getSuit() > card2.getSuit()) {
            return 1;
        }
        int rank1 = getRankAceHigh(card1);
        int rank2 = getRankAceHigh(card2);

        if (rank1 < rank2) {
            return -1;
        }
        if (rank1 > rank2) {
            return 1;
        }
        return 0;
    }

    private int getRankAceHigh(Card card) {
        int rank = card.getRank();
        if (rank == 1) {
            return 14;
        } else {
            return rank;
        }
    }
};
```

Este código define una clase anónima que implementa `compare`, como se requiere. Luego crea una intancia de la clase recién definida e innombrada. Si no estás familiarizado con las clases anónimas en java, puedes leer sobre ellas en http://thinkdast.com/anonclass.

Al usar este `Comparator`, podemos invocar `sort` así:

```
Collections.sort(cards, comparator);
```

En este ordenamiento, el As de Espadas (Ace of Spades) es considerado como el de mayor rango en el mazo; el dos de Tréboles (two of Clubs) es el menor.

El código en esta sección está en `Card.java` si quieres experiementar con él. Como un ejercicio podrías querer escribir un comparador que ordene por `rank` primero y luego por `suit`, así todos loa Ases estarían juntos y todos los dos, etc.

## 16.6 Extensiones

Si logras que funcione una versión básica de este ejercicio, podrías querer trabajar en estos ejercicios opcionales:

- Lee sobre TF-IDF en http://thinkdast.com/tfidf e impleméntalo. Podrías tener que modificar `JavaIndex` para calcular frecuencias de documento; es decir, el número total de veces que cada término aparece en todas las páginas en el índice.
- Para consultas con más de un término de búsqueda, la relevancia total para cada página es actualmente la suma de la relevancia para cada término. Piensa sobre cuándo esta versión simple podría no funcionar bien, y prueba algunas alternativas.
- Construye una interfaz de usuario que permita a los usuarios ingresar consultas con operadores booleanos. Interpreta las consultas, genera los resultados, luego ordénalos por relevacia y muestra las URLs con los puntajes más altos. Considera generar "fragmentos" que muestren dónde aparecieron los términos de búsqueda en la página. Si quieres hacer una aplicación Web para tu interfaz de usuario, considera usar Heroku como una opción simple para desarrollar y desplegar aplicaciones Web usando Java. Ver http://thinkdast.com/heroku.

# Capítulo 17

# Ordenamiento

Los departamentos de ciencias de la computación tienen una obsesión poco saludable con los algoritmos de ordenamiento. Con base en la cantida de tiempo que los estudiantes de CC dedican al tema, podrías pensar que la elección de un algoritmo de ordenamiento es la piedra fundamental de la ingeniería de software moderna. Por supuesto, la realidad que los desarrolladores de software pueden pasar años, o incluso carreras, sin pensar sobre cómo funciona el ordenamiento. Para casi todas las aplicaciones, ellos utilizan cualquier algoritmo de propósito general que provea el lenguaje o las bibliotecas que usan. Y usualmente eso está bien.

Así que si te saltas este capítulo y no aprendes nada sobre algoritmos de ordenamiento todavía puedes ser un excelente desarrollador. Pero hay unas cuantas razones por las que podrías querer hacerlo de todas maneras:

1. Aunque hay muchos algoritmos de propósito general que funcionan bien para la vasta mayoría de aplicaciones, hay dos algoritmos de propósito espcial sobre los que podrías querer saber: ordenamiento radix y el ordenamiento por montículos acotado.

2. Un algoritmo de ordenamiento, el ordenamiento por mezcla, es un excelente ejemplo para la enseñanza porque demuesra una estrategia importante y útil para el diseño de algoritmos, llamada “divide-conquista-pega”. También, cuando analizamos su rendimiento, aprenderás sobre

una orden de crecimiento que no hemos visto antes, **linearítmica**. Finalmente, algunos de los algoritmos más ampliamente utilizados son híbridos que incluyen elementos del ordenamiento por mezcla.

3. Una razón más para aprender sobre algoritmos de ordenamiento es que a los entrevistadores técnicos les fascina preguntar sobre ellos. Si quieres que te contraten, ayuda si puedes demostrar formación cultural en CC.

Así, en este capítulo analizaremos el ordenamiento por inserción, implementarás el ordenamiento por mezcla, te contaré sobre el ordenamiento radix y escribirás una versión simple del ordenamiento por montículos acotado.

## 17.1 Ordenamiento por inserción

Comenzaremos con el ordenamiento por inserción,mayormente porque es sencillo de describir e implementar. No es muy eficiente, pero tiene algunas características que lo redimen, como veremos.

En lugar de explicar el algoritmo aquí, sugiero que leas la página de Wikipedia sobre el ordenamiento por inserción en http://thinkdast.com/insertsort, que incluye pseudocódigo y ejemplos animados. Regresa cuando hayas captado la idea general.

Aquí está una implementación del ordenamiento por inserción en Java:

```
public class ListSorter<T> {

    public void insertionSort(List<T> list, Comparator<T> comparator) {

        for (int i=1; i < list.size(); i++) {
            T elt_i = list.get(i);
            int j = i;
            while (j > 0) {
                T elt_j = list.get(j-1);
                if (comparator.compare(elt_i, elt_j) >= 0) {
                    break;
                }
                list.set(j, elt_j);
```

```
                j--;
            }
            list.set(j, elt_i);
        }
    }
}
```

Defino una clase, `ListSorter`, como un contenedor para algoritmos de ordenamiento. Al usar el parámetro de tipo, `T`, podemos escribir métodos que funcionen en listas que contengan cualquier tipo de objeto.

`insertionSort` toma dos parámetros, una `List` de cualquier tipo y un `Comparator` que sabe cómo comparar dos objetos de tipo `T`. Éste ordena la lista "en sitio", lo que significa que modifica la lista existente y no tiene que reservar nuevo espacio.

El siguiente ejemplo muestra cómo llamar a este método con una `List` de objetos `Integer`:

```
List<Integer> list = new ArrayList<Integer>(
    Arrays.asList(3, 5, 1, 4, 2));

Comparator<Integer> comparator = new Comparator<Integer>() {
    @Override
    public int compare(Integer elt1, Integer elt2) {
        return elt1.compareTo(elt2);
    }
};

ListSorter<Integer> sorter = new ListSorter<Integer>();
sorter.insertionSort(list, comparator);
System.out.println(list);
```

`insertionSort` tienen dos bucles anidados, así que podrías asumir que su tiempo de ejecución es cuadrático. En este caso, eso es correcto, pero antes de saltar a esa conclusión, tienes que comprobar que el número de veces que se ejecuta cada bucle es proporcional a $n$, el tamaño del arreglo.

El bucle externo itera desde 1 hasta `list.size()`, así que es lineal con respecto al tamaño de la lista, $n$. El bucle interno itera desde `i` hasta 0, así que también

es lineal con respecto a $n$. Por lo tanto, el número total de veces que el bucle interno se ejecuta es cuadrático.

Si no estás seguro de ello, aquí está el argumento:

- La primera vez, $i = 1$ y el bucle interior se ejecuta al menos una vez.
- La segunda vez, $i = 2$ y el bucle interior se ejecua al menos dos veces.
- La última vez, $i = n - 1$ y el bucle interior se ejecuta al menos $n - 1$ veces.

Así que el número total de veces que el bucle interior se ejecuta es la suma de la serie $1, 2, \ldots, n - 1$, que es $n(n - 1)/2$. Y el término principal de esa expresión (el que tiene el exponente más alto) es $n^2$.

En el peor escenario, el ordenamiento por inserción es cuadrático. Sin embargo:

1. Si los elementos ya están ordenados, o casi, el ordenamiento por inserción es lineal. Específicamente, si cada elemento no está a más de $k$ posiciones de donde debería estar, el bucle interior nunca se ejecuta más de $k$ veces, y el tiempo de ejecución total es $O(kn)$.
2. Porque la implementación es simple, la sobrecarga es menor, es decir, aunque el tiempo de ejecución es $an^2$, el coeficiente del término principal, $a$, probablemente es pequeño.

Así que si sabemos que el arreglo está casi ordenado, o no es muy grande, el ordenamiento por inserción podría ser una buena elección. Pero para arreglos más grandes, podemos hacerlo mejor. De hecho, mucho mejor.

## 17.2 Ejercicio 14

El ordenamiento por mezcla (mergesort) es uno de varios algoritmos de ordenamiento cuyo tiempo de ejecución es mejor que el cuadrático. De nuevo, en lugar de explicar el algoritmo aquí, sugiero que leas sobre él en Wikipedia en http://thinkdast.com/mergesort. Una vez captes la idea, regresa y puedes probar su comprensión escribiendo una implementación.

En el repositorio para este libro, encontrarás los archivos de código fuente para este ejercicio:

- `ListSorter.java`
- `ListSorterTest.java`

Ejecuta `ant build` para compilar los archivos de código fuente, luego ejecuta `ant ListSorterTest`. Como es lo usual, debería fallar, porque tienes trabajo por hacer.

En `ListSorter.java`, he provisto un boceto de dos métodos, `mergeSortInPlace` y `mergeSort`:

```
public void mergeSortInPlace(List<T> list, Comparator<T> comparator) {
    List<T> sorted = mergeSortHelper(list, comparator);
    list.clear();
    list.addAll(sorted);
}

private List<T> mergeSort(List<T> list, Comparator<T> comparator) {
   // TODO: fill this in!
   return null;
}
```

Estos dos métodos hacen lo mismo pero proveen interfaces diferentes. `mergeSort` toma una lista y devuelve una nueva lista con los mismos elementos ordenados de forma ascendente. `mergeSortInPlace` es un método `void` que modifica una lista existente.

Tu trabajo es llenar `mergeSort`. Antes de escribir una versión completamente recursiva del ordenamiento por mezcla, comienza con algo como esto:

1. Divide la lista en dos.
2. Ordena (sort) las mitades usando `Collections.sort` o `insertionSort`.
3. Combina (merge) las mitades ordenadas en una lista completa ordenada.

Esto te dará una oportunidad de depurar el código para combinar sin lidiar con la complejidad de un método recursivo.

A continuación, agrega un caso base (er http://thinkdast.com/basecase). Si te dan una lista con solo un elemento, puedes devolverla inmediatamente,

dado que ya está ordenada, en cierto modo. O si la longitud de la lista está por debajo de algún umbral, podrías ordenarla usando `Collections.sort` o `insertionSort`. Prueba el caso base antes de continuar.

Finalmente, modifica tu solución para que realice dos llamadas recursivas para ordenar las mitades del arreglo. Cuando logres que funcionen, deberían pasar `testMergeSort` y `testMergeSortInPlace`.

## 17.3 Análisis del ordenamiento por mezcla

Para clasificar el tiempo de ejecución del ordenamiento por mezcla, sirve pensar en términos de niveles de recursión y de cuánto trabajo se realiza en cada nivel. Supón que comenzamos con una lista que contiene $n$ elementos. Aquí están los pasos del algoritmo:

1. Crea dos nuevos arreglos y copia la mitad de los elementos en cada uno.
2. Ordena las dos mitades.
3. Combina las mitades.

La figura 17.1 muestra estos pasos.

El primer paso copia cada uno de los elementos una vez, así que es lineal. El tercer paso también copia cada elemento una vez, así que también es lineal. Ahora necesitamos determinar la complejidad del paso 2. Para hacerlo, ayuda una mirada a una parte diferente de cálculo, que muestra los niveles de la recursión, como en la figura 17.2.

En el nivel superior, tenemos 1 lista con $n$ elementos. Por simplicidad, asumamos que $n$ es una potencia de 2. En el siguiente nivel hay 2 listas con $n/2$ elementos. Luego 4 listas con $n/4$ elementos, y así sucesivamente hasta que tenemos $n$ con 1 elemento.

En cada nivel tenemos un total de $n$ elementos. Hacia abajo, tenemos que dividir (split) los arreglos por la mitad, lo que toma tiempo proporcional a $n$ en cada nivel. Hacia arriba, tenemos que combinar (merge) un total de $n$ elementos, lo que también es lineal.

Si el número de niveles es $h$, la cantidad total de trabajo para el algoritmo es $O(nh)$. Así que, ¿cuántos niveles hay? Hay dos formas de pensar sobre eso:

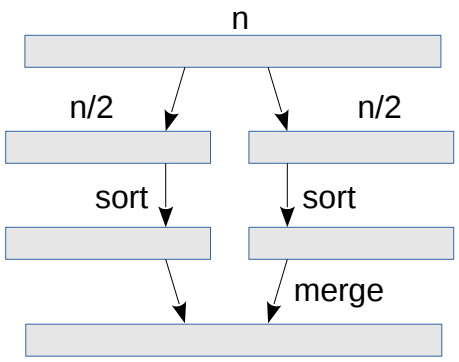


Figura 17.1: Representación del ordenamiento por mezcla mostrando un nivel de recursión.

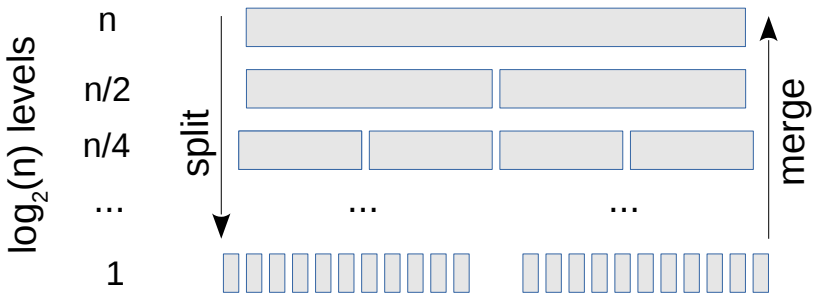


Figura 17.2: Representación del ordenamiento por mezcla mostrando todos los niveles de recursión.

1. ¿Cuántas veces tenemos que dividir $n$ por la mitad para obtener 1?

2. O, ¿cuántas veces tenemos que duplicar 1 para obtener $n$?

Otra forma de formular la segunda pregunta es "¿Qué potencia de 2 es $n$?"

$2^h = n$

Aplicando $\log_2$ a ambos lados se tiene:

$h = \log_2 n$

Así que el tiempo total es $O(n \log n)$. No me preocupé por escribir la base del logaritmo porque logaritmos con diferentes bases difieren por un factor constante, así que todos los logaritmos están en la misma orden de crecimiento.

Los algoritmos en $O(n \log n)$ a veces son llamados "linearítmicos", pero la mayoría de personas solo dicen "n log n".

Resulta que $O(n \log n)$ es el límite inferior teórico para los algoritmos de ordenamiento que trabajan comparando elementos entre sí. Eso significa que no hay un "ordenamiento por comparación" cuya orden de crrecimiento sea mejor que $n \log n$. Véase http://thinkdast.com/compsort.

Pero como veremos en la siguiente sección, ¡hay ordenamientos de no comparación que toman tiempo lineal!

## 17.4 Ordenamiento radix

Durante la Campaña Presidencial de Estados Unidos en 2008, al candidato Barack Obama se le pidió que realizara un análisis de algoritmos improvisado cuando visitó Google. El director ejecutivo Eric Schmidt en broma le preguntó sobre "la forma más eficiente de ordenar un millón de enteros de 32 bits." Al parecer a Obama lo habían puesto sobre aviso, porque rápidamente replicó, "Creo que el ordenamiento de burbuja sería la forma equivocada de hacerlo." Puedes ver el video en http://thinkdast.com/obama.

Obama tenía razón. el ordenamiento de burbuja es conceptualmente más sencillo pero su tiempo de ejecución es cuadrático; e incluso entre los algoritmos

de ordenamiento cuadráticos, su desemepeño no es muy bueno. Véase http://thinkdast.com/bubble.

La respuesta que Schmidt probablemente buscaba es "ordenamiento radix", que es un algoritmo de ordenamiento de **no comparación** que funciona si el tamaño de los elementos está acotado, como un entero de 32-bits o una cadena de 20 caracteres.

Para ver cómo funciona esto, imagina que tienes una pila de fichas donde cada ficha contiene una palabra de tres letras. Así es como podrías ordenar las fichas:

1. Realiza un pase por las fichas y divídelas en contenedores con base en la primer letra. Así que las palabras que comienzan con `a` deberían ir en un contenedor, seguidas por las palabras que comienzan con `b`, y así sucesivamente.

2. Divide cada contenedor de nuevo con base en la segunda letra. Así que las palabras que comienzan con `aa` irían juntas, seguidas por las palabras que comienzan con `ab`, y así. Por supeuesto, no todos los contenedores se llenarían, pero eso está bien.

3. Divide cada contenedor de nuevo con base en la tercera letra.

Llegados a este punto cada contenedor contiene un elemento y los contenedores están ordenados de forma ascendente. La figura 17.3 muestra un ejemplo con palabras de tres letras.

La fila superior muestra las palabras desordenadas. La segunda fila muestra cómo quedan los contenedores tras el primer pase. Las palabras en cada contenedor comienzan con la misma letra.

Tras el segundo pase, las palabras en cada contenedor comienzan con las mismas dos leras. Tras el tercer pase, solo puede haber una palabra en cada contenedor, y los contenedores están en orden.

Durante cada pase, iteramos por los elementos y los agregamos a contenedores. Siempre que los contenedores permitan agregar en tiempo constante, cada pase es lineal.

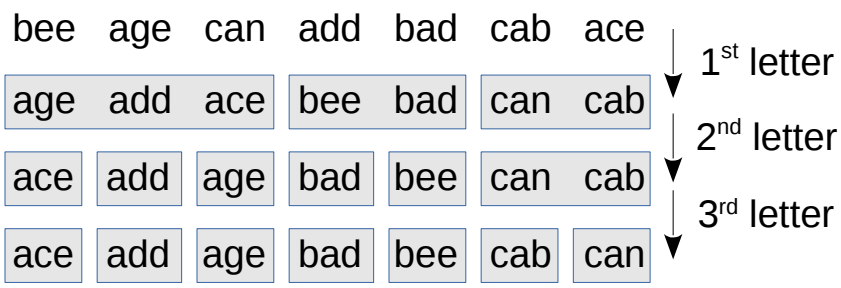


Figura 17.3: Ejemplo del ordenamiento radix con palabras de tres letras.

El número de pases, al que llamaré $w$, dependen del "ancho" ("width") de las palabras, pero no depende del número de palabras, $n$. Así que el orden de crecimiento es $O(wn)$, que es lineal con respecto a $n$.

Hay muchas variantes del ordenamiento radix, y muchas formas de implementar cada una. Puedes leer más sobre ellas en http://thinkdast.com/radix. Como un ejercicio opcional, considera escribir una versión del ordenamiento radix.

## 17.5 Ordenamiento por montículos

Además del ordenamiento radix, que se aplica cuando las cosas que quieres ordenar tienen un tamaño fijo, hay otro ordenamiento de propósitos especiales con el que podrías encontrarte: ordenamiento por montículos acotados. El ordenamiento por montículos acotados es útil si estás trabajando con un conjunto de datos muy grande y quieres reportar el "Top 10" o el "Top k" para algún valor $k$ mucho más pequeño que $n$.

Por ejemplo, supón que estás monitoreando un servicio Web que maneja mil millones de transacciones por día. Al final de cada día, quieres reportar las $k$ transacciones más grandes (o las más lentas, o cualquier otro superlativo). Una opción es guardar todas las transacciones, ordenarlas al final del día, y seleccionar el top $k$. Eso tomaría tiempo proporcional a $n \log n$, y sería muy lento porque probablemente mil millones de transacciones no quepan en la memoria de un programa. Tendríamos que usar un algoritmo de memoria

externa. Puedes leer sobre el ordenamiento externo en http://thinkdast.com/extsort.

Al usar un montículo acotado, ¡podemos obtener mejores resultados! Así es como procederemos:

1. Explicaré el ordenamiento por montículos (no acotado)
2. Lo implementarás.
3. Explicaré el ordenamiento por montículos acotado y lo analizaré.

Para entender el ordenamiento por montículos, tienes que entender qué es un montículo, que es una estructura de datos similar a un árbol binario de búsqueda (ABB). Aquí están las diferencias:

- En un ABB, cada nodo, `x`, tiene la "propiedad ABB": todos los nodos en el subárbol izquierdo de `x` son menores que `x` y todos los nodos en el subárbol derecho son mayores que `x`.
- En un montículo, cada nodo, `x`, tiene la "propiedad montículo": todos los nodos en ámbos subárboles de `x` son mayores que `x`.
- Los montículos son como ABBs balanceados; cuando agregas o remueves elementos, ellas realizan algún trabajo adicional para volver a balancear el árbol. Como resultado, pueden ser implementadas de forma eficiente usando un arrreglo de elementos.

El elemento más pequeño en un montículo siempre está en la raíz, así que podemos encontrarlo en tiempo constante. Agregar y remover elementos de un montículo toma tiempo proporcional a la altura del árbol $h$. Y como el montículo siempre está balanceado, $h$ es proporcional a $\log n$. Puedes leer más sobre los montículos en http://thinkdast.com/heap.

La `PriorityQueue` de Java se implementa usando un montículo. `PriorityQueue` provee los métodos especificados en la interfaz `Queue`, incluyendo `offer` y `poll`:

- `offer`: Agrega un elemento a la cola, actualizando el montículo para que cada nodo tenga la "propiedad montículo". Toma un tiempo de $\log n$.

- `poll`: Remueve el elemento más pequeño en la cola de la raíz y actualiza el montículo. Toma un tiempo de $\log n$.

Dada una `PriorityQueue`, puedes ordenar fácilmente una colección de $n$ elementos de la siguiente manera:

1. Agrega todos los elementos de la colección a una `PriorityQueue` usando `offer`.

2. Remueve los elementos de la cola usando `poll` y agrégalos a una `List`.

Porque `poll` devuelve el elemento más pequeño que queda en la cola, los elementos se agregan a la `List` en orden ascendente. A esta forma de ordenar se la llama **ordenamiento por montículos** (véase http://thinkdast.com/heapsort).

Agregar $n$ elementos a la cola toma un tiempo de $n \log n$. Al igual que remover $n$ elementos. Así que el tiempo de ejecución para el ordenamiento por montículos es $O(n \log n)$.

En el repositorio para este libro, en `ListSorter.java` encontrarás el boceto de un método llamado `heapSort`. Llénalo y luego ejecuta `ant ListSorterTest` para confirmar que funciona.

## 17.6 Montículo acotado

Un montículo acotado es un montículo que está limitado a contener como máximo $k$ elementos. Si tienes $n$ elementos, puedes llevar el control de los $k$ elementos mayores así:

Inicialmente, el montículo está vacío. Para cada elemento, `x`:

- Caso 1: Si el montículo no está lleno, agrega a `x` al montículo.

- Caso 2: Si el montículo está lleno, compara `x` con el elemento *menor* del montículo. Si `x` es más pequeño, no puede ser uno de los $k$ elementos mayores, así que puedes descartarlo.

- Caso 3: Si el montículo está lleno y `x` es mayor que el elemento menor en el montículo, remueve el elemento menor del montículo y agrega a `x`.

Al usar un montículo con el elemento menor en la parte superior, podemos llevar el control de los $k$ elementos mayores. Analicemos el desempeño de este algoritmo. Para cada elemento, realizamos una de las siguientes operaciones:

- Caso 1: Agregar un elemento al montículo es $O(\log k)$.
- Caso 2: Encontrar el elemento más pequeño en el montículo es $O(1)$.
- Caso 3: Remover el elemento más pequeño es $O(\log k)$. Agregar `x` también es $O(\log k)$.

En el peor escenario, si los elementos aparecen en orden ascendente, siempre ejecutamos el caso 3. De ser así, el tiempo total para procesar $n$ elementos es $O(n \log k)$, que es lineal con respecto a $n$.

En `ListSorter.java` encontrarás el boceto de un método llamado `topK` que toma una `List`, un `Comparator`, y un entero $k$. Debería devolver los $k$ elementos mayores en la `List` en orden ascendente. Llénalo y luego ejecuta `ant ListSorterTest` para confirmar que funciona.

## 17.7 Complejidad espacial

Hasta este momento hemos hablando mucho sobre análisis del tiempo de ejecución, pero para muchos algoritmos también nos preocupa el espacio. Por ejemplo, una de las desventajas del ordenamiento por mezcla es que copia todos los datos. En nuestra implementación, la cantidad total de espacio que se reserva es $O(n \log n)$. Con una implementación un poco más ingeniosa, puedes reducir el requerimiento de espacio a $O(n)$.

Por el contrario, el ordenamiento por inserción no copia los datos porque ordena los elementos en sitio. Usa variables temporales para comparar dos elementos a la vez y usa unas cuantas variables locales adicionales. Pero su uso del espacio no depende de $n$.

Nuestra implementación del ordenamiento por montículos crea una nueva `PriorityQueue` para guardar los elementos, así que es espacio es $O(n)$; pero

si se te permitiera ordenar la lista en sitio, podrías ejecutar el ordenamiento por montículos con un espacio de $O(1)$.

Uno de los beneficios del algoritmo de ordenamiento por montículos acotados que acabas de implementar es que sólo necesita espacio proporcional a $k$ (el número de elemento que queremos conservar), y $k$ a menudo es mucho menor que $n$.

Los desarrolladores de software tienden a prestar más atención al tiempo de ejecución que al espacio, y para muchas aplicaciones, eso es lo apropiado. Pero para conjuntos de datos grandes, el espacio puede ser igual de importante o incluso más. Por ejemplo:

1. Si un conjunto de datos no cabe en la memoria de un programa, el tiempo de ejecución muchas veces se incrementa dramáticamente, o puede que ni siquiera se ejecute. Si eliges un algoritmo que necesita menos espacio, y eso hace posible que el cálculo quepa en la memoria, podría ejecutarse mucho más rápido. En el mismo sentido, un programa que usa menos espacio podría hacer mejor uso de la caché de la CPU y ejecutarse más rápido (véase http://thinkdast.com/cache).

2. En un servidor que ejecuta muchos programas al mismo tiempo, si puedes reducir el espacio que cada programa necesita, podrías ser capaz de ejecutar más programas en el mismo servidor, lo que reduce los costos de hardware y energía.

Así que esas son algunas razones por las que deberías saber al menos un poquito sobre las necesidades de espacio de los algoritmos.

# Índice