EDA (ETS de Ingeniería Informática). Curso Académico 2020-2021

Práctica 3 - Parte 1. El Map de Términos del Buscador una Biblioteca Digital: implementación, evaluación y uso

Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València

1. Objetivo

El objetivo final de esta práctica es que el alumno aplique al diseño de una aplicación concreta los conceptos Java sobre la EDA *Map* estudiados en el Tema 3 de esta asignatura, a fin de que obtenga una mejor comprensión de esta EDA y de las condiciones que deben cumplirse para poder usarla eficientemente. En concreto, al acabar esta práctica el alumno deberá ser capaz de implementar, evaluar y reutilizar la jerarquía Java del *Map* que usa el Buscador de una Biblioteca Digital.

2. Descripción del problema

De acuerdo con la segunda definición que aparece en el diccionario de la Real Academia de la Lengua Española (RAE), el término "índice" (del latín *index*) significa...

> En un libro u otra publicación, lista ordenada de los capítulos, artículos, materias, voces, etc., en él contenidos, con indicación del lugar donde aparecen.

Desde la antigüedad hasta nuestros días, reyes como Ashurbanipal (siglo VII a.C.), bibliotecarios como Calímaco de Cirene (siglo III a.C.), intelectuales como Ibn al-Nadim (siglo X d.C.), gente normal y corriente y "Mr. Google" vienen utilizando muy variados tipos de índices por una misma y única razón: evitar la forma más simple de búsqueda (y recuperación) de determinada información en una colección de documentos, esto es, buscar dicha información en, uno tras otro, todos sus documentos (búsqueda lineal).

A pesar de que el orden de magnitud de las colecciones de documentos y las herramientas empleadas para manejarlas han cambiado significativamente a lo largo de los años, la solución del problema sigue siendo la misma: añadir un índice al buscador (motor de búsqueda o *search engine* en inglés). Dado que, por definición, un índice asocia términos a los lugares de los documentos en los que aparecen, si se consulta uno de sus términos, el índice provee un acceso directo a dichos documentos. Esta es la razón por la que el uso de un índice siempre minimiza no solo el tiempo requerido para encontrar la información que se busca sino también la cantidad de información que debe ser analizada para encontrarla.

Obviamente, para conseguir la rapidez que proporciona un índice a la hora de buscar información **antes** hay que construirlo. Así que, en realidad, las operaciones básicas que realiza el buscador de una colección de documentos son dos: indexar, o construir su índice, y buscar. Para construir el índice de términos se escanea el contenido de todos y cada uno de los documentos de la colección, mientras que para buscar una determinada información en ella solo se consulta su índice; dado que solo se indexa una vez la colección pero se realizan un gran número de búsquedas sobre su índice, el elevado coste de indexar se amortiza más que suficientemente.

En esta primera parte de la práctica el estudiante implementará, usará y evaluará el buscador de una biblioteca Digital con las siguientes características:

- Los 76 libros que la componen están almacenados en formato texto (ficheros `.txt`) y disponibles en PoliformaT (carpeta comprimida `TXT.zip`). Sus títulos figuran en el fichero `lista.txt` y los `10` primeros de ellos en el fichero `lista10.txt`, por lo que las bibiliotecas que conforman se denominarán en adelante, respectivamente, "`lista`" y "`lista10`".

- Los términos de su índice son aquellas palabras de sus libros compuestas exclusivamente por letras minúsculas del alfabeto castellano. En sus libros hay un total de 105985, y solo 22310 en los `10` primeros.

  Estos términos se obtienen al construir el índice, tras realizar el análisis léxico de cada una de las líneas de los libros de la biblioteca y pasar a minúsculas todas las palabras que contienen.

- Su índice es **C**ompleto, i.e. es un índice en el que cada término tiene asociada una lista en la que aparecen registrados los títulos **Y** las líneas de los libros donde aparece. Cada elemento de esta lista se suele

denominar *posting*, por lo que, a su vez, la lista de apariciones de un término se conoce como su lista de *postings*; además, nótese, la frecuencia de aparición del término `t` en los libros de la bibilioteca es la talla de su lista de *postings*. Por ejemplo, en la figura 1 aparece la lista de *postings* de talla 15 asociada al término `criterios` en los libros de la biblioteca `lista10`: su primer *posting* (`Acceso-Abierto, 391`) indica que `criterios` aparece por primera vez en la línea 391 del libro `Acceso-Abierto.txt`, mientras que el decimoquinto y último (`capitalismo_cog, 3370`) indica que la última aparición del término se da en la línea 3370 del libro `capitalismo_cog.txt`.

```
[Acceso-Abierto, linea 391
, Acceso-Abierto, linea 392
, Acceso-Abierto, linea 537
, Acceso-Abierto, linea 962
, Acceso-Abierto, linea 8224
, AprendiendoJava-y-POO, linea 3567
, AprendiendoJava-y-POO, linea 3574
, AprendizajeInvisible, linea 375
, AprendizajeInvisible, linea 2034
, AprendizajeInvisible, linea 2042
, AprendizajeInvisible, linea 2284
, AprendizajeInvisible, linea 4021
, Bases-de-Datos, linea 5438
, capitalismo_cog, linea 396
, capitalismo_cog, linea 3370
]
```

Figura 1: Lista de *postings* asociada al término `criterios` en la biblioteca Digital `lista10`

La estructura de datos que representa este índice es un *Map* en el que cada clave es un término del índice y su valor asociado es la lista de *postings* correspondiente a dicho término. Como ya se ha comentado, este *Map* se construye al tiempo que se procesan los libros de la biblioteca para obtener los términos de su índice; en concreto, una vez detectado un término `t`, el algoritmo que se aplica es el "clásico" que se usa para construir cualquier *Map* `m`, independientemente de los tipos de sus claves y valores:

(a) `SI m.recuperar(t) == null`, crear una nueva entrada de `m` con clave `t` y su valor asociado; `SINO`, actualizar el valor asociado a la entrada de clave `t`, el resultado de `m.recuperar(t)`.

(b) Insertar en `m` la entrada de clave `t` y su valor asociado.

Las clases de una Bibiblioteca Digital (BD)

En base a la descripción del problema realizada, se ha diseñado una aplicación Java cuyas clases son:

- `Termino`, la clase que representa un término del índice del buscador de la BD y, por tanto, la clase de la clave del `Map` que implementa dicho índice mediante una Tabla *Hash*. Por ello, un `Termino TIENE`...
  - `UN String termino` que almacena la palabra asociada al término.
  - `UN int valorHash` que almacena el valor *Hash* del término.
  - `UNA int baseHashCode` que almacena la base de la función de dispersión polinomial que se usa para calcular el `valorHash` del término.
  - `UN` método `hashCode` que sobrescribe el de `Object` para calcular el `valorHash` del término según una función polinomial con base `baseHashCode`.
  - `UN` método `equals` que sobrescribe el de `Object` para garantizar que cualquier par de términos que sean "`equals`" tengan el mismo valor *Hash* (consistencia de `equals` con `hashCode`).
- `BuscadorDeLaBibl`, la clase que representa al buscador de la BD y que, por ello, `TIENE`...
  - `UNA` clase interna `Posting`, que representa un elemento de la lista de *postings* asociada a un término de la BD. Así, un `Posting TIENE UN String tituloLibro` y `UNA int lineaLibro` que representan, respectivamente, el título de un libro de la BD y un número de línea de `libro`.

    Obviamente, una lista de *postings* se puede representar mediante un `ListaConPI` de `Posting`s.
  - `UN Map<Termino, ListaConPI<BuscadorDeLaBibl.Posting>> index`, implementado mediante una Tabla `Hash` Enlazada, que representa el índice Completo del buscador de la BD.
  - `UN` constructor que, básicamente, construye el índice `Map index` a partir de la estimación del máximo número de términos que puede tener el índice (`maxTerminos`), la lista de libros que componen la BD (`listaLibros`) y el conjunto de separadores que se usarán durante el análisis léxico de sus libros

(`separadores`). Este constructor invoca al método `indexarLibro` para actualizar el `Map index` con los términos (claves) y *postings* asociados (valores) que aparecen en cada uno de los libros de la BD.

A modo de ejemplo, la Figura 2 muestra el resumen del proceso de indexación que aparece en la ventana de terminal de *BlueJ* al crear en su *Object Bench* el buscador de la BD `lista10`, i.e. el objeto `buscador10`.

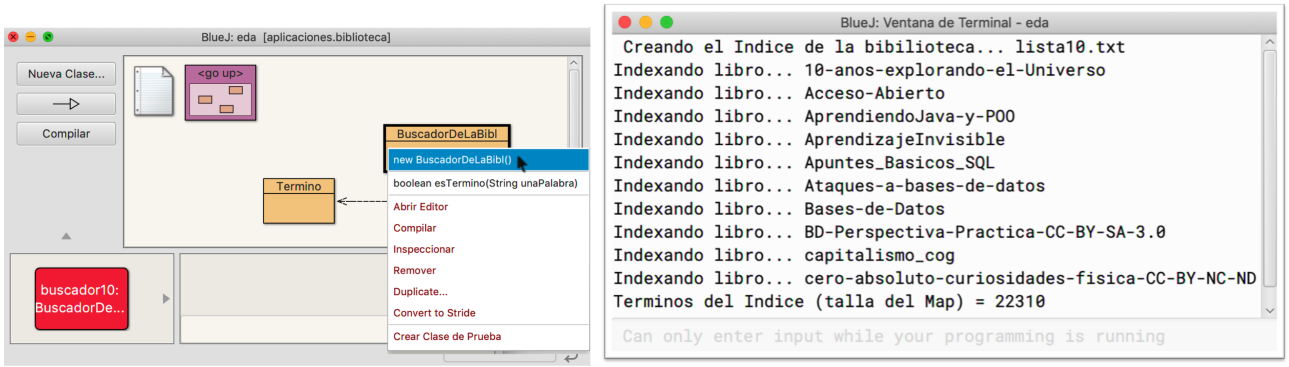


Figura 2: Resultados de crear el Buscador e Índice de la BD `lista10`

- `UN` método consultor `buscar` que, como su nombre indica, implementa la búsqueda de una palabra `String p` en el índice del buscador, i.e. en el `Map index`. Específicamente, `buscar` devuelve la frecuencia de aparición del término asociado a `p` y, en su caso, el listado que contiene los títulos y líneas de los libros de la BD en los que aparece dicho término, i.e. su lista de *postings*; por ejemplo, la Figura 3 muestra el estado de la ventana de terminal de *BlueJ* al `buscar` las palabras "Criterios" y "Mesopotamia" en el índice de `buscador10` desde su *Code Pad*.

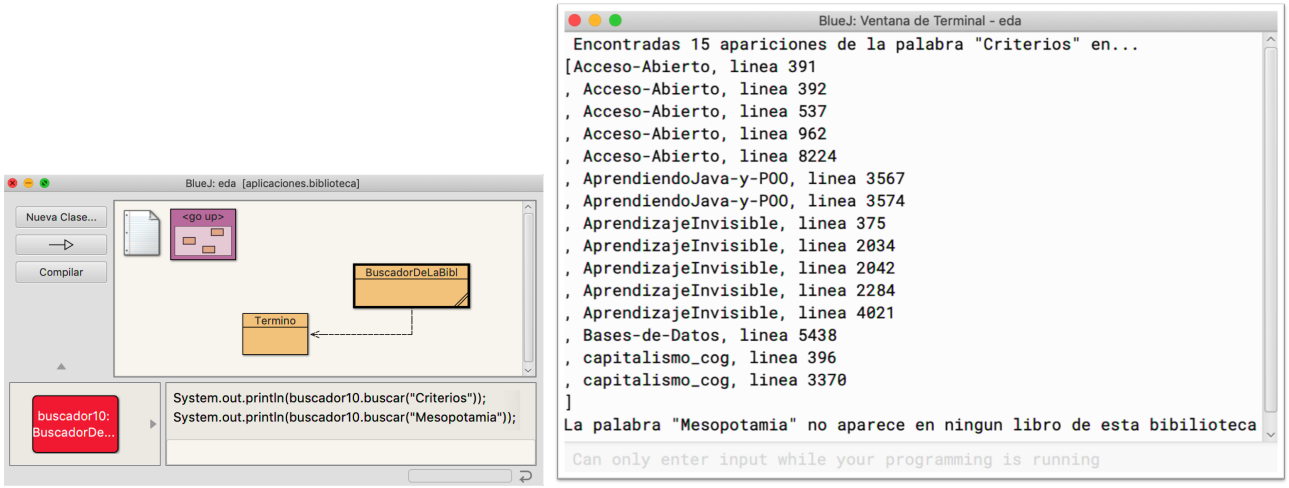


Figura 3: Resultados de `buscar` "Criterios" y "Mesopotamia" en el Índice de la BD `lista10`

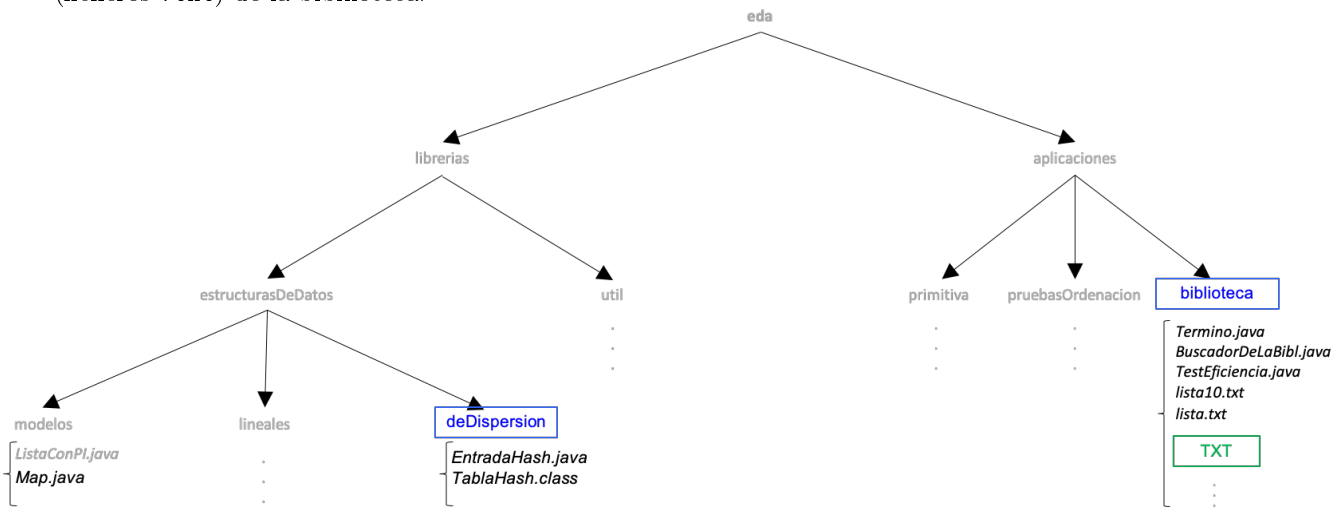
3. Actividades a realizar

Antes de llevar a cabo las actividades que se proponen en este apartado, es necesario que el alumno actualice la estructura de paquetes y ficheros de su proyecto *Bluej eda* siguiendo los pasos que se indican a continuación.

- Abrir su proyecto *BlueJ eda*.
- Abrir el paquete *aplicaciones* del proyecto y crear en él un nuevo paquete de nombre *biblioteca*; este nuevo paquete contendrá tanto las clases de la aplicación a desarrollar en esta práctica (`Termino` y `BuscadorDeLaBibl`) como el programa que permitirá evaluar su eficiencia (`TestEficiencia`).
- Abrir el paquete *librerias/estructurasDeDatos* del proyecto y crear en él un nuevo paquete de nombre *deDispersion*; este nuevo paquete contendrá las clases que requiere la implementación de la Tabla *Hash* Enlazada que usa la aplicación de la práctica.
- Salir de *BlueJ* **seleccionando la opción *Salir* de la pestaña *Proyecto***.

- Descargar los ficheros disponibles en *PoliformaT* en sus correspondientes directorios, tal y como se muestra en la siguiente figura.

  ATENCIÓN: descomprimir `TXT.zip` para obtener la carpeta `TXT` que contiene los 76 libros "digitales" (ficheros `.txt`) de la bibilioteca.



- Abrir otra vez el proyecto *eda*.
- Compilar la clase `Map` del paquete *librerias.estructurasDeDatos.modelos* del proyecto; luego, se aconseja cerrar el paquete seleccionando la opción *Cerrar* de la pestaña *Proyecto*.
- Compilar la clase `EntradaHash` del paquete *librerias.estructurasDeDatos.deDispersion* del proyecto; luego, se aconseja cerrar el paquete *deDispersion* seleccionando la opción *Cerrar* de la pestaña *Proyecto*.

3.1. Completar el código de la clase `Termino` y validarlo

Tras acceder al paquete *aplicaciones.biblioteca* de su proyecto *eda*, el alumno debe:

- Completar el código del método `hashCode` de la clase, que sobrescribe al de `Object` para obtener el `valorHash` de un (`this`) `Termino`. Para ello, implementar eficientemente -usando la regla de Horner, SIN usar método alguno de la clase `Math`- la siguiente función de dispersión polinomial de base `baseHashCode`, en la que `n` es la longitud de `this.termino`.

```
this.valorHash = this.termino.charAt(0) * this.baseHashCode^(n - 1) +
                 this.termino.charAt(1) * this.baseHashCode^(n - 2) + ... +
                 this.termino.charAt(n - 1)
```

  Cabe observar que, gracias a que la clase `Termino` tiene el atributo `valorHash`, la implementación del método `hashCode` propuesta resulta muy eficiente: el valor de la función polinomial de `baseHashCode` para `this Termino` solo se calcula la primera vez que se invoca al método, i.e. cuando `valorHash == 0`, mientras que en las siguientes invocaciones el método devolverá directamente el `valorHash` calculado en la primera.

- Completar el código del método `equals` de la clase, que sobrescribe al de `Object` para lograr la consistencia con el método `hashCode` de la forma más eficiente posible. Para ello, invocar en su cuerpo al método `equals` de la clase `String` ÚNICAMENTE si `this` y `otro Termino` tienen el mismo `valorHash`.
- Comprobar la corrección de la clase `Termino`. En concreto, debe usar el *Code Pad* de *BlueJ* para crear los `Terminos` asociados a las palabras y bases de la siguiente tabla, aplicarles los métodos `hashCode` y `equals` diseñados y comprobar que los resultados son los esperados.

Palabra	Trivial (1)	McKenzie (4)	String (31)
saco	422	9419	3522362
asco	422	8555	3003422
noreste	768	602277	2127397360
enteros	768	564879	-1591951684
cronista	867	2239905	2118401189
cortinas	867	2232087	-452686651

- Ejecutar los métodos constructor y `buscar` de la clase `BuscadorDeLaBibl` tal y como se describió en el apartado 2 de este boletín (ver Figuras 2 y 3) y comprobar que los resultados de los procesos de indexación y búsqueda que aparecen en la ventana de terminal de *BlueJ* coinciden con los de dichas figuras.

3.2. Completar el método `hapax` de la clase `BuscadorDeLaBibl` y validarlo

En esta actividad el alumno debe:

- Completar el cuerpo del método `hapax()` de la clase `BuscadorDeLaBibl`, que devuelve una `ListaConPI` con aquellos términos del índice de una BD que aparecen solo una vez en sus libros, los llamados *hapax legomena* de la BD, o `null` si no existe ninguno.
- Comprobar la corrección del método `hapax()` implementado realizando las siguientes acciones:
  - Crear el objeto `buscador10`, i.e. el buscador de la BD `lista10`, tal y como se describió en el apartado 2 de este boletín (ver Figura 2).
  - Escribir en el *Code Pad* de *BlueJ* las instrucciones o expresiones Java que permitan...
    a) Asignar la `ListaConPI` que devuelve el método `hapax()`, i.e. los *hapax legomena* de `lista10`, a una variable llamada `hapax10`.
    b) Comprobar que la lista `hapax10` tiene 10.126 elementos y que el primero, tercero y último de ellos son, respectivamente, `"traté"`, `"monopolizar"` y `"estantes"`.

    NOTA: dado lo elevado de su talla, se recomienda obtener el último elemento de la lista `hapax10` convirtiéndola en un `String` y usando los métodos `lastIndexOf(",")` y `substring` de la clase `String`.

3.3. Analizar la eficiencia del (Índice del) Buscador de la BD `lista10`

Dado que el `Map index` se implementa mediante una Tabla *Hash* **Enlazada** con factor de carga por defecto **0.75** y sin *Rehashing*, su eficiencia viene dada por el factor de carga (`fc`) de la Tabla que lo implementa, i.e. por la longitud media de sus cubetas: solo si es menor o igual que 0.75, la Tabla es eficiente. A su vez, el `fc` de la Tabla depende de dos factores:

- La efectividad del método `hashCode` implementado en la clase `Termino`, puesto que cuanto mejor disperse menores serán el número de colisiones que se produzcan y la longitud media de las cubetas de la Tabla.
- La estimación de la talla que, como máximo, va a tener el `Map`, puesto que es la que determina el número de cubetas de las que dispone la Tabla para dispersar sus Entradas y, con ello, su `fc`.

Por tanto, el análisis de la eficiencia del `Map index` de la BD `lista10` que debe realizar el alumno en esta actividad consiste en determinar experimentalmente cuál es el "mejor" método `hashCode()` que se puede implementar en la clase `Termino` y cuál es la "mejor" estimación de la talla que, como máximo, debe tener el `Map`. Para ello, dispone del programa `TestEficiencia` que, como puede observarse en su código, ...

- Toma como argumentos, en este orden, una de las bases (31, 1 o 4) que se pueden usar para implementar el método `hashCode()` de la clase `Termino` y un código que indica si la Tabla que implementa el `Map index` efectúa o no la operación de *Rehashing* (`"CON"` o `"SIN"`).
- Para cada par de argumentos, construye tres `TablasHash` con el mismo método `hashCode()` pero con una talla máxima estimada distinta: 22310, el número exacto de `Terminos` del `Map`; 11155, aproximadamente la mitad de la cifra anterior; 112, la centésima parte (aproximadamente) de la cifra anterior.
- Para cada `TablasHash` construida, muestra en la ventana de terminal de *BlueJ* los siguientes valores: su `fc`; la desviación típica de la longitud de sus cubetas; el coste promedio de localizar una de sus claves, calculado a partir del número de colisiones que se producen al localizar sus 22310 claves.

  Así mismo, obtiene el histograma de ocupación de la tabla y lo guarda en un fichero de texto del directorio *aplicaciones/biblioteca/**res*** con un nombre que corresponde a los datos empleados para generarlo. Por ejemplo, si se ejecuta `TestEficiencia` con argumentos `"31"` y `"SIN"`, los nombres de los ficheros con los histogramas de ocupación de las tres tablas construidas son `histoB31(112).txt`, `histoB31(11155).txt` e `histoB31(22310).txt`.

  NOTA: para dibujar un histograma de ocupación, por ejemplo el que contiene el fichero `histoB31(112).txt`, se puede usar el comando `gnuplot>plot "histoB31(112).txt" using 1:2 with boxes`.

En resumen, para realizar esta actividad el alumno debe hacer lo siguiente:

(a) Ejecutar el programa `TestEficiencia` con tres pares distintos de argumentos: (`"31"`, `"SIN"`), (`"1"`, `"SIN"`) y (`"4"`, `"SIN"`).

(b) Analizar los resultados obtenidos para establecer cuál es la Tabla que puede implementar con mayor eficiencia el `Map index` de la BD `lista10`.