EDA (ETS de Ingeniería Informática). Academic Year 2020-2021

Lab 3 - Part 1. The Search Engine Map of a Digital Library: implementation, evaluation and use

Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València

1 Objectives

The final goal of this lab is to apply to the design of a given application the Java concepts for the *Map* data structure studied in Topic 3 of the course, so as to develop a better understanding of this data structure and the conditions that need to be held in order to use it efficiently. Specifically, after this lab the student should be able to implement, evaluate and reuse the Java *Map* hierarchy in the Search Engine of a Digital Library.

2 Problem description

According to the first definition given by the Oxford Dictionary, the term "index" means ...

> (in a book or set of books) an alphabetical list of names, subjects, etc. with reference to the pages on which they are mentioned.

From ancient times to nowadays, kings as Ashurbanipal (7th century BCE), librarians as Callimachus of Cyrene (3rd century BCE), intellectuals as Ibn al-Nadim (10th century CE), ordinary people and "Mr. Google" have been using quite different types of indexes for the very same and sole reason: to avoid linearly searching for a specific information in every single one of the documents of a collection, the simplest form of tracing and recovering the required information.

Despite the data's order of magnitude and the tools these people handle have deeply changed over the years, the solution to the problem still remains the same: adding an index to the search engine. Since, by definition, an index maps back from terms to the locations in the documents of the collection in which they occur, if the queries are index terms, the index will provide a direct access to these documents. This is why the use of an index always minimizes the time required to find information and the amount of information which must be traced.

It is important to point out now that to gain the speed benefits of using an index at searching (retrieval) time, an index has to be built beforehand. So, the search engine performs two main tasks: indexing and searching. The indexing stage builds the index by scanning the contents of all the collection documents and it is performed by the so called indexer (person or machine). In the search stage, when looking a specific index term up, only the index is referenced. Note that since the collection is indexed only once, the additional computer storage required to store the index, as well as the considerable increase in the time required to build it, are trade off for the time saved during the searching stage.

In this first part of the lab the student will face the implementation, use and evaluation of the search engine of a Digital library, the characteristics of which are the following:

- Its 76 books are in plain text (`.txt`) files and available in PoliformaT (compressed folder `TXT.zip`). Their titles are in the file `lista.txt` and the first ten of them are in the `lista10.txt` file, so the libraries they make up shall be hereinafter referred to as "`lista`" and "`lista10`"
- Its index terms are only those words in the library books made up of letters of the Spanish alphabet, but in lower case. There is a total of 105985 terms, out of which the first 22310 are in the first ten books.

  These index terms are obtained during the indexing stage: first, the lexical analysis of every line of every library book takes place; second, every word detected is converted to lower case.
- It has a *Full* index, i.e. an index in which an entry is composed of a term and the list that records **both** the titles and the line numbers of those library books in which such term appears; each element in the list is known as *posting*, so the list is called *postings* list. Furthermore, note that the frequency of occurrence of an index term in a given digital library is the size of its paired *postings* list. For example, Figure 1 shows

the *postings* list of size 15 paired with the index term `criterios` in the digital library `lista10`: its first element (`Acceso-Abierto, 391`) indicates that the term `criterios` appears for the first time at line 391 of the library book `Acceso-Abierto.txt`, while its 15th and last element (`capitalismo_cog, 3370`) indicates that the term appears for the last time at line 3370 of the book `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
]
```

Figure 1: *Postings list* for index term `criterios` in the Digital Library `lista10`

The data structure that implements this index is a *Map*. Each key of this *Map* is an index term of the system and its corresponding value is the *postings* list associated to that key. As already mentioned, this *Map* is built on the fly while processing the library books to detect the terms of its index. Specifically, once an index term `t` has been detected, the algorithm that is applied follows the "standard recipe" for building a *Map* `m`, regardless of the type of its keys and values:

(a) `IF m.recuperar(t) == null`, create/initialize the value to be paired with key `t`; `ELSE`, update the current value paired with the existing key `t`, i.e. the resulting value of `m.recuperar(t)`.

(b) Put the resulting key-value pair into the *Map* `m`.

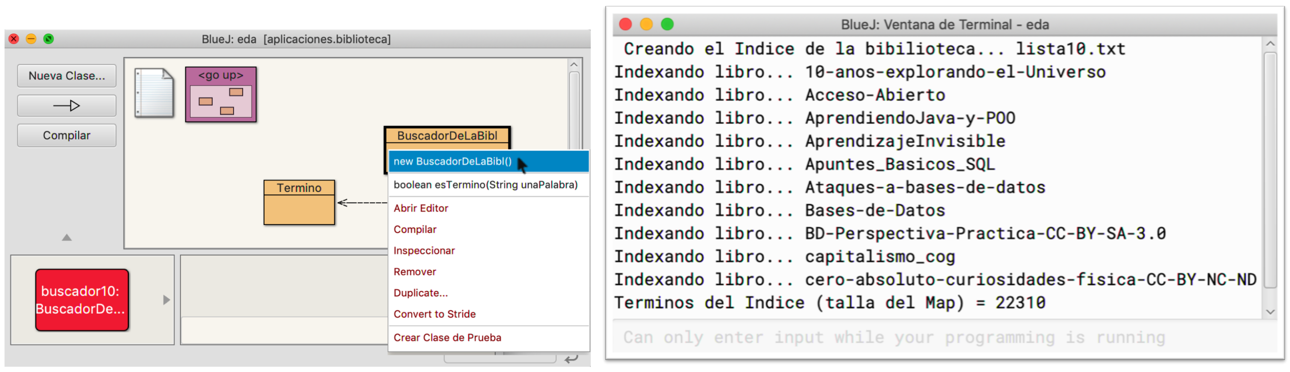
The Digital Library (DL) classes

Following the previous statements, a Java application has been implemented with the following classes:

- `Termino`, the class which represents a term of the DL search engine index and, thus, it is the key class of the `Map` that implements such an index. So, `Termino HAS` ...
  - `A String termino` that stores the word associated to the index term.
  - `AN int valorHash` that stores the hash value associated to the index term.
  - `AN int baseHashCode` that stores the base of the polynomial hash function used to compute the `valorHash` of the index term.
  - `A hashCode` method that overrides the one in `Object` in order to calculate the `valorHash` of the index term by using a polynomial function in base `baseHashCode`.
  - `AN equals` method that overrides the one in `Object` to provide consistent behavior, namely that any two index terms viewed as "`equals`" must have the same hash value.
- `BuscadorDeLaBibl`, the class which represents the DL search engine. So, `BuscadorDeLaBibl HAS` ...
  - `AN` inner `class Posting` that represents an element of the *postings* list associated to a DL index term. So `Posting HAS` a pair of fields: `A String tituloLibro` representing the title of a library book and `AN int lineaLibro` representing a line number of `tituloLibro`.

    It is quite obvious then that, each `Map` value, i.e. the *postings* list paired with an index term, can be represented by one `ListaConPI` of `Indices`.
  - `A Map<Termino, ListaConPI<BuscadorDeLaBibl.Posting>> index`, that represents the *Full* index of the DL and it is implemented as a Separate-Chaining Hash Table (`TablaHash` class).
  - `A` constructor method that, basically, builds the `Map index` from an estimation of its maximum size (`maxTerminos`), the list name of the library books (`listaLibros`) and the set of separators to be used during the lexical analysis of its books (`separadores`). Specifically, the constructor invokes the execution of the `indexarLibro` method in order to update the `Map index` with the terms (keys) and *postings* (values) that occur in each book (`fichLibro`) of the DL.

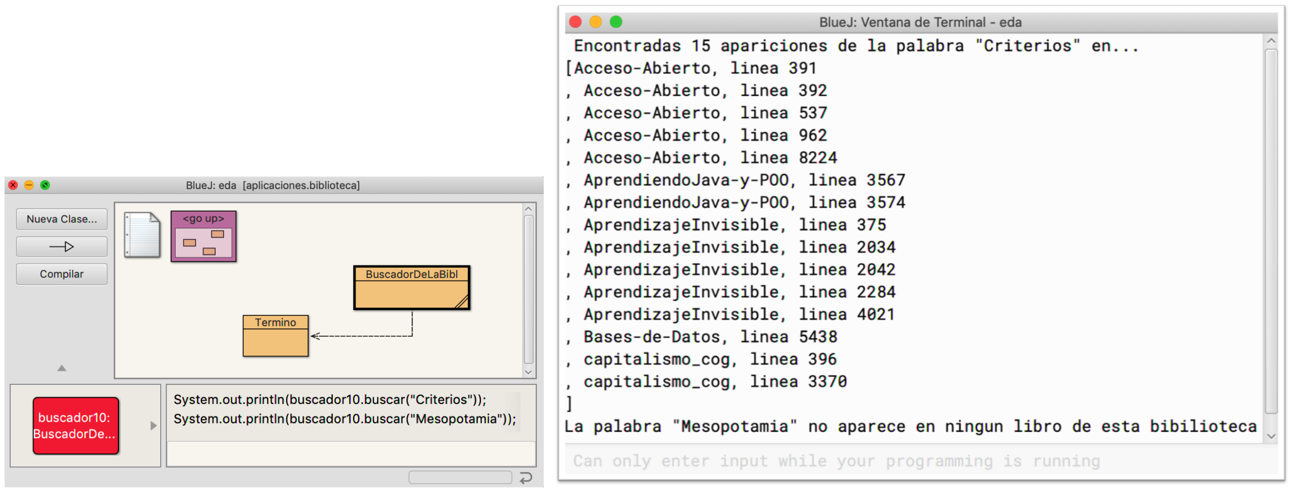
As an example, Figure 2 shows the summary of the indexing process that will be displayed at the *BlueJ* terminal window when creating the search engine of the DL `lista10` (object `buscador10`) on the object bench -by right-clicking the class icon and executing its constructor from the class popup menu.



```
 Creando el Indice de la bibilioteca... lista10.txt
Indexando libro... 10-anos-explorando-el-Universo
Indexando libro... Acceso-Abierto
Indexando libro... AprendiendoJava-y-POO
Indexando libro... AprendizajeInvisible
Indexando libro... Apuntes_Basicos_SQL
Indexando libro... Ataques-a-bases-de-datos
Indexando libro... Bases-de-Datos
Indexando libro... BD-Perspectiva-Practica-CC-BY-SA-3.0
Indexando libro... capitalismo_cog
Indexando libro... cero-absoluto-curiosidades-fisica-CC-BY-NC-ND
Terminos del Indice (talla del Map) = 22310
 Can only enter input while your programming is running
```

Figure 2: Building the Search Engine and Index of the DL `lista10`

– AN accessor (getter) method `buscar` that implements the retrieval stage of the search engine, that is to say looking up a given word (`String`) `p` in the search engine index, i.e. in the `Map index`. Specifically, `buscar` returns the frequency of occurrence of the index term associated with `p` and, if applicable, the list that records the titles and line numbers of those library books in which such a term appears, i.e. its *postings* list; for example, Figure 3 shows the state of the *BlueJ* terminal window when the index of `buscador10` is searched for the words “Criterios” and “Mesopotamia” -by calling the method `buscar` from the *BlueJ* code pad.


BlueJ: eda [aplicaciones.biblioteca]
Nueva Clase...
Compilar
<go up>
BuscadorDeLaBibl
Termino
buscador10: BuscadorDe...
BlueJ: Ventana de Terminal - eda

```
System.out.println(buscador10.buscar("Criterios"));
System.out.println(buscador10.buscar("Mesopotamia"));
```

```
 Encontradas 15 apariciones de la palabra "Criterios" en...
[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
]
La palabra "Mesopotamia" no aparece en ningun libro de esta bibilioteca
 Can only enter input while your programming is running
```

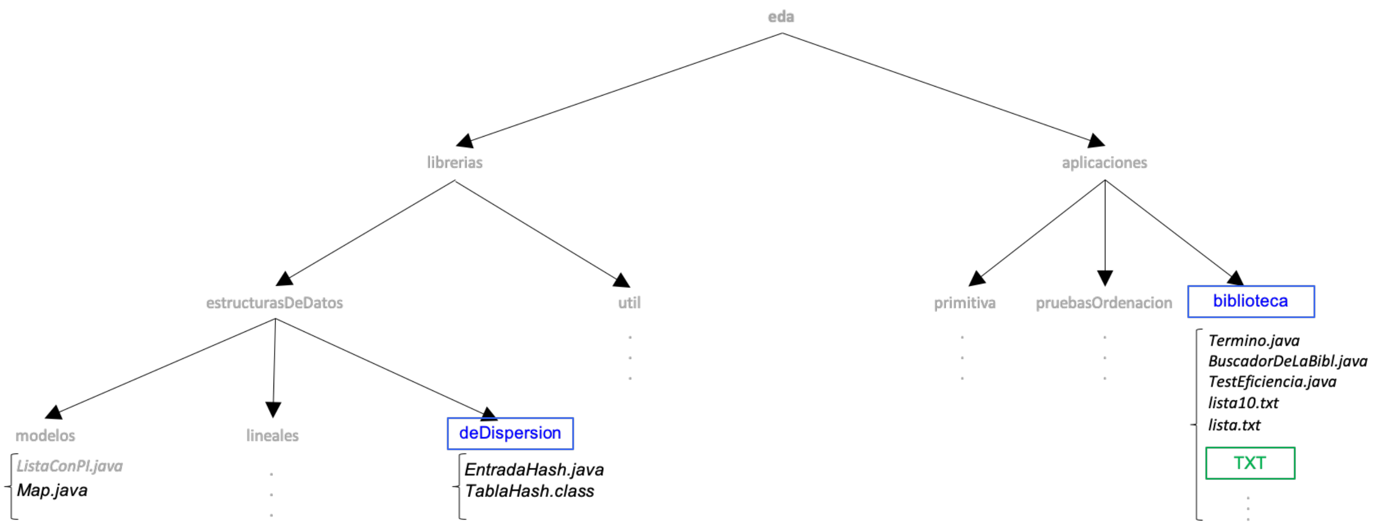
Figure 3: Looking up the words “Criterios” and “Mesopotamia” in the Index of the DL `lista10`

3 Lab activities

Before doing the activities described below in this section, the structure of packages and files of the student *BlueJ eda* project must be updated by carrying out the following steps:

- Start *BlueJ* and open the *eda* project.
- Open the *aplicaciones* package of the project and create within it a new package called *biblioteca*; it will contain the application’s classes of this lab (`Termino` and `BuscadorDeLaBibl`) and the program to evaluate its efficiency (`TestEficiencia`).
- Open the *librerias.estructurasDeDatos* package of the project and create within it a new package called *deDispersion*; it will contain the classes needed to implement the Separate-Chaining Hash Table to be used in this lab.
- ***Quit BlueJ* by selecting *Quit* from the *Project* menu.**

- Download the classes available in *PoliformaT* into their corresponding folders as shown in the following figure.



  NOTE: Remember to unzip `TXT.zip` on your computer to find the 76 books (`.txt` files) of the DL "`lista`".

- (Re)Start *BlueJ* and open the *eda* project.

- Open the package *librerias.estructurasDeDatos.modelos* of the project and compile the class `Map`. This done, close the package by selecting *Close* from the *Project* menu.

- Open the package *deDispersion* and compile the class `EntradaHash`. This done, close the package by selecting *Close* from the *Project* menu.

3.1 Completing and testing the `Termino` class

In order to carry out this activity the student must do the following steps:

- Within *aplicaciones.biblioteca* package of the *BlueJ eda* project, open the `Termino` class.

- Complete the code of `hashCode` method of the class, that overrides the one in `Object` to calculate the `valorHash` of `this Termino`. To do so, write an efficient implementation of the following `n`th-degree polynomial hash function in base `this.baseHashCode`, where `n` is the length of `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)
```

  Note that using the Horner's rule avoids to compute this polynomial directly and so the use of `Math.pow` method.

  Also note that, because each the `Termino` object "caches its `hashCode`" into the instance variable `valorHash`, the proposed implementation of the `hashCode` method turns out to be very efficient: the value of the polynomial in `baseHashCode` of `this Termino` is fully computed ONLY on the first call to the method, i.e. when `valorHash == 0`, whereas subsequent calls to the method will simply return the "cached" value of `valorHash` that was set on the first call.

- Complete the body of the `equals` method of the class, that overrides the one in `Object` to provide consistent behavior in the most efficient way possible, namely by using the `String` class `equals` method ONLY when `this` and `other Termino` have the same `valorHash`.

- Test the correctness of the `Termino` class by typing into the *BlueJ* code pad the combination of Java expressions needed for (1) creating the `Termino` objects corresponding to each one of the words and bases listed in the table below, (2) applying `hashCode` and `equals` methods of the class to these `Termino` objects, and (3) checking whether the results of said methods are as expected.

Word	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

- Execute the constructor and `buscar` methods of `BuscadorDeLaBibl` class as shown in Figures 2 and 3 and check that their results are the same as those displayed at the *BlueJ* terminal windows of said figures.

3.2 Completing and testing `hapax` method of `BuscadorDeLaBibl` class

In order to carry out this activity the student must do the following steps:

- Within `BuscadorDeLaBibl` class, complete the body of `hapax()` method that returns a `ListaConPI` with those index terms that occur only once in the books of a DL, the so-called *hapax legomena*, or `null` if no such terms exist.

- Test the correctness of the `hapax` method as follows:
  - Create the object `buscador10`, i.e. the search engine of the DL `lista10`, as described in Section 2 of this bulletin (see Figure 2).
  - Type into the *BlueJ* code pad the necessary stataments or expressions to
    a) Store the `ListaConPI` that returns the `hapax` method, i.e. the *hapax legomena* in `lista10`, into a variable called `hapax10`.
    b) Check that the `hapax10` list has 10126 elements and that its first, third and last elements are, respectively, `"traté"`, `"monopolizar"` and `"estantes"`.

    TIP: given its high size, an easy way to get the last element of the list `hapax10` is to convert it into a `String` object and use the `String` class methods `lastIndexOf(",")` and `substring` to manipulate it.

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()` que se implemente 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`.