

**TPE Colección de libros**

SEGUNDA etapa

TUDAI | Programación 3 | 06-06-2018

Autor

Magalí Molina (maguimarchena@gmail.com)

Institución



# **Introducción**

## Primera etapa

El objetivo es realizar las clases y métodos necesarios para construir una colección de libros partiendo de un archivo csv. La colección debe tener un índice organizado por géneros para facilitar la búsqueda de libros por los distintos géneros a los que están asociados. El resultado de estas búsquedas debe ser volcado en un nuevo archivo csv.

Luego de la implementación se debe analizar distintas métricas de la performance al trabajar con archivos de 20, 1.000, 100.000 y 1.000.000 de libros respectivamente.

## Segunda etapa

En la segunda parte, el sistema se dedica a analizar las estadísticas de búsqueda. Como entrada, los archivos csv contienen en cada línea un grupo de géneros que se buscó en ese órden. La información debe volcarse en un grafo dirigido con peso que muestre todas las relaciones de búsqueda entre los géneros.

Los servicios deben indicar una cantidad de géneros más buscados luego de uno inicial, todos los géneros buscados luego de uno inicial y, por último, un grafo de los géneros vinculados (en un ciclo) al género inicial.

Se repiten las pruebas y análisis de métricas con los distintos archivos.

**PRIMERA ETAPA**

**Desarrollo**

Elección de estructuras

Las 3 estructuras posibles para la implementación tienen un costo de operaciones distinto, específicamente en sus métodos para insertar elementos, ordenarlos y acceder al elemento contenido en una posición (índice) ó nodo.

LinkedList y ArrayList de Java tienen una complejidad O(1) para agregar un elemento al final con el método add(object), mientras que la implementación de un Árbol Binario de Búsqueda cuenta con una complejidad O(log n) para la inserción. La ventaja del ABB es que la inserción se realiza de forma ordenada por lo que no requiere un ordenamiento posterior.

Pasando a la complejidad de ordenamiento y búsqueda, ArrayList tiene O(n log n) para ordenamiento (sort()), O(1) para búsquedas del tipo contains() que indica la existencia del elemento en la lista y finalmente O(n) para búsquedas en las que no se conoce el índice del elemento, realizadas con iterator; LinkedList con O(n) para ordenar y O(n/2) para buscar, mientras que ABB tiene O(n log n) en búsqueda y no requiere ordenamiento.

ArrayList tiene ventaja sobre ABB en inserción, e inversamente, ABB tiene una clara ventaja en ordenamiento ya que se realiza conjuntamente con la inserción. Si bien el ordenamiento consume una porción importante de tiempo, para este caso sólo se realiza una vez al final de la carga, por lo que no tiene gran influencia en la elección ya que lo más importante para tener en cuenta es la búsqueda, que se realizará en múltiples oportunidades

Tanto para la colección como para el índice, elegí usar ArrayList por los motivos antes mencionados y además por la facilidad de implementación que brinda al ser una clase de Java, probada y mejorada, con una amplia variedad de métodos ya establecidos.

Estructura utilizada

Se implementó una clase principal Library (paquete library) que contiene la lista de libros, el índice y un comparador de nombres de género como atributos. En los métodos se destacan addBook () que se llama en el loop de carga y getBooksByGender () que recibe el nombre del género del que se desean obtener los libros y devuelve una lista con los mismos.

En el paquete components se encuentran las clases que forman la librería, Book, Gender e Index. Se accede a sus atributos y métodos a través de la librería.

Book contiene los atributos title, autor, pages y gender y sólo getters y setters como métodos. En Gender lo más importante es la lista de libros que almacena los libros de ese género y el método addBook () que suma libros a dicha colección. Index reúne una lista de géneros y los métodos para crear un género y obtener su lista de libros. También tiene una lista son sólo los nombres de los géneros que facilita checkear si el género ya se encuentra en el índice.

En el paquete fileReader se encuentra el lector de archivos que se utiliza como elemento de acceso a todo el sistema, sólo se puede crear una colección y hacer búsquedas creando primero un reader y usándolo para levantar información de un archivo y volcarla en la librería. A su vez, el paquete fileWritter contiene la clase que genera el archivo de salida luego de la búsqueda.

Acompañan los paquetes útil y counter que contienen el timer y el contador de iteraciones para las pruebas, así como también el paquete comparators con los comparadores usados para ordenamiento.

Todo el proceso de testeo se realiza a través del main ubicado en la clase App, donde se crean distintos readers, uno por cada tamaño de archivo y se les asigna luego el archivo correspondiente para iniciar la tarea de carga de estructura y posteriormente la búsqueda.

Pruebas de ejecución

Se realizaron 2 tipos de pruebas, de tiempos y de cantidad de iteraciones.

En las pruebas de tiempo, implementadas con la clase Timer provista por la cátedra, se ve claramente la influencia del tamaño del archivo en el tiempo total de ejecución.

Se realizaron dos sets de pruebas, uno para la carga de la estructura y otro para la búsqueda de libros por género. En ambos sets se llevan a cabo cuatro pruebas, una por cada tamaño de archivo, con 5 repeticiones. [[1]](#footnote-1)

Para un tamaño de archivo de 20, los tiempos son de 0.008 segundos para la carga y de 0.001 segundos para la búsqueda, en promedio, con máximos de 0.018 s. carga y 0.0016 s. búsqueda y mínimos de 0.0014 s. carga y 0.00013 s. búsqueda.

Para tamaño 1.000 los valores no aumentan en gran medida, mientras que para tamaño 100.000 se observa un leve incremento. El aumento más importante se ve, como se esperaba, en el tamaño 1.000.000.

**Pruebas de carga:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Carga | | | |
|  | 20 | 1.000 | 100.000 | 1.000.000 |
| valor 1 | 0,017559 | 0,052151 | 1,023500 | 45,082704 |
| valor 2 | 0,001771 | 0,059814 | 1,048894 | 0,506631 |
| valor 3 | 0,016936 | 0,048873 | 1,038633 | 2,600351 |
| valor 4 | 0,001908 | 0,050718 | 1,029887 | 15,426910 |
| valor 5 | 0,001504 | 0,049638 | 0,102081 | 15,734213 |
| **Max** | 0,017559 | 0,059814 | 1,048894 | 45,082704 |
| **Min** | 0,001504 | 0,048873 | 0,102081 | 0,506631 |
| **Promedio** | 0,007936 | 0,052239 | 0,848599 | 15,870162 |

**Pruebas de búsquedas de libros:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Búsqueda | | | |
|  | 20 | 1.000 | 100.000 | 1.000.000 |
| valor 1 | 0,001326 | 0,002254 | 0,021949 | 0,815325 |
| valor 2 | 0,001594 | 0,002526 | 0,022869 | 0,531739 |
| valor 3 | 0,001348 | 0,000241 | 0,021300 | 0,139799 |
| valor 4 | 0,000126 | 0,001971 | 0,020351 | 1,125213 |
| valor 5 | 0,000136 | 0,002173 | 0,021607 | 0,625386 |
| **Max** | 0,001594 | 0,002526 | 0,022869 | 1,125213 |
| **Min** | 0,000126 | 0,000241 | 0,020351 | 0,139799 |
| **Promedio** | 0,000906 | 0,001833 | 0,021615 | 0,647492 |

Para las pruebas de cantidad de iteraciones, se creó la clase IterationCounter y se realizaron nuevamente dos sets de pruebas, igual a las anteriores, para analizar la carga de la estructura y la búsqueda en los cuatro tamaños de archivo.

Al igual que en las pruebas de tiempo se observa un aumento en la cantidad de iteraciones cuanto más grande es el archivo, pasando de 1827 en carga y 41 en búsqueda para tamaño 20 a 226.624.175 en carga y 138252 en búsqueda para tamaño 1.000.000.

**Pruebas de carga:**

|  |  |  |  |
| --- | --- | --- | --- |
| Carga | | | |
| 20 | 1.000 | 100.000 | 1.000.000 |
| 1827 | 229217 | 22617619 | 226624175 |

**Pruebas de búsqueda de libros:**

|  |  |  |  |
| --- | --- | --- | --- |
| Búsqueda | | | |
| 20 | 1.000 | 100.000 | 1.000.000 |
| 41 | 189 | 13498 | 138252 |

**SEGUNDA ETAPA**

**Desarrollo**

Estructura

Para esta etapa se implementó una estructura de Grafo dirigido con peso de la forma:

Graph {

**private** List<Vertex> vertices;

**private** List<Edge> edges;

}

La lista vertices contiene los vértices ó nodos del grafo, en este caso, los Géneros, mientras que la lista edges contiene las aristas que comunican cada vértice con otro.

Vertex {

**private** String label;

**private** List<Edge> edges;

**private** String state;

**private** Vertex parent;

}

La clase Vertex contiene un nombre (label), una lista de las aristas en las que ese vértice es origen (source), un vértice padre y un String con el estado para realizar el dfs.

Edge {

**protected** **static** **final** **int** ***DEFAULT\_WEIGHT*** = 1;

**protected** Integer weight;

**private** Vertex source;

**private** Vertex destination;

}

Por su parte, la clase Edge contiene una constante con el peso default (1), un int de peso, un vértice origen (source) y un vértice destino (destination).

Las clases pertenecientes al grafo pueden encontrarse en el paquete graph.

En la clase Graph, los métodos permiten añadir vértices individualmente y a través de una lista, mientras que las aristas sólo pueden ser agregadas de a una ú obtenidas en base a los vértices (getEdgesFromVertices(List<Vertex>)).

Mediante otros métodos se retornan ambas listas y un vértice ó arista en particular, buscando por el nombre del vértice ó, en el caso de la arista, buscando si existe una arista entre 2 vértices dados. También contiene métodos de ordenamiento para las listas.

Los métodos más importantes son los que permiten el funcionamiento de los servicios:

* public List<String> getMostSearchedAfter (int q, String label)

retorna una lista con los géneros pedidos luego de analizar las aristas del género inicial y contando sólo hasta la cantidad dada (q).

* public List<String> getAllAfter (String label)

retorna una lista con todos los géneros buscados luego del inicial, accediendo a métodos internos que realizan dfs.

* public Graph cycle\_getAllConected (String label)

retorna el grafo que forman los géneros presentes en un ciclo encontrado, con el género inicial como raíz ó principio, accede a métodos internos que buscan este ciclo y construyen el grafo final en base a una lista de todos los géneros del ciclo.

En la clase Edge, además de los getters usuales, se encuentra un método para agregar peso a la arista, usado en la construcción del grafo.

En la clase Vertex, el método más destacado es el que permite la creación de la lista de adyacentes en base a las aristas que contiene el vértice y sus respectivos vértices destino.

La clase App, reúne los métodos para levantar la información de los archivos csv y generar el grafo, además de los 3 servicios necesarios, que acceden a los métodos diseñados para este fin en el grafo.

Finalmente, la clase Test contiene el main y las pruebas a realizar. Esta clase y App se encuentran en el paquete app. Las utilidades para las pruebas, la lectura de archivos y los comparadores de ordenamiento están en el paquete util.

La estructura implementada se basa en ArrayList de Java, cuya complejidad de métodos ya fué analizada en la primera etapa.

Pruebas de ejecución

Al igual que en la primera etapa, se realizaron 2 tipos de pruebas, de tiempos y de cantidad de iteraciones.

Pruebas de tiempo, implementadas con la clase Timer provista por la cátedra, y las de cantidad de iteraciones con una clase creada para tal fin.

Nuevamente se realizaron dos sets de pruebas, uno para la carga de la estructura y otro para la ejecución de los servicios. En ambos sets se llevan a cabo cuatro pruebas, una por cada tamaño de archivo. Las de tiempo se realizan con 5 repeticiones cada una. En el set de servicios, se realiza una prueba por cada servicio separado. [[2]](#footnote-2)

Como se esperaba, ambas métricas son afectadas de forma directa por el tamaño del archivo utilizado, aumentando cada vez más.

A continuación, se detalla en tablas y gráficos los resultados obtenidos.

Tiempo

Se observa el claro aumento de tiempo de ejecución con el tamaño del archivo, al igual que en la primera etapa, por tratarse del mismo tipo de estructura.

Carga:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Carga | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor 1 | 0,08249297 | 0,13090744 | 1,489272925 | 20,053541490 |
| valor 2 | 0,00880505 | 0,09386696 | 1,624356481 | 16,430867047 |
| valor 3 | 0,01125505 | 0,09269267 | 1,543395220 | 16,135972271 |
| valor 4 | 0,00946072 | 0,08818026 | 1,368894835 | 16,140869263 |
| valor 5 | 0,00951686 | 0,08452397 | 1,441581140 | 15,967078885 |
|  |  |  |  |  |
| **máx** | 0,08249297 | 0,130907 | 1,624356 | 20,053541 |
| **min** | 0,00880505 | 0,084524 | 1,368895 | 15,967079 |
| **promedio** | 0,02430613 | 0,098034 | 1,493500 | 16,945666 |

Por el lado de los servicios, no se observa influencia del tamaño del archivo, se mantienen los tiempos relativamente estables entre un tamaño y otro.

Si se observan diferencias al buscar inicialmente con un género que aparece al principio de la lista, al medio ó al final.

Servicio 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 1 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor 1 | 0,001429069 | 0,001154968 | 0,001639777 | 0,001532914 |
| valor 2 | 0,002286390 | 0,001993572 | 0,001536536 | 0,001851088 |
| valor 3 | 0,001303490 | 0,001770790 | 0,001461671 | 0,001813052 |
| valor 4 | 0,001820901 | 0,001623475 | 0,001867389 | 0,001668757 |
| valor 5 | 0,002417061 | 0,001839617 | 0,001807015 | 0,001730339 |
|  |  |  |  |  |
| **máx.** | 0,002417061 | 0,001993572 | 0,001867389 | 0,001851088 |
| **min** | 0,001303490 | 0,001154968 | 0,001461671 | 0,001532914 |
| **promedio** | 0,001851382 | 0,001676484 | 0,001662478 | 0,001719230 |

Servicio 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 2 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor 1 | 0,001154968 | 0,005498928 | 0,002165641 | 0,003188388 |
| valor 2 | 0,002334690 | 0,003339324 | 0,004312564 | 0,002897382 |
| valor 3 | 0,002288201 | 0,003419623 | 0,002446382 | 0,003323627 |
| valor 4 | 0,002097417 | 0,002911871 | 0,002240505 | 0,002575584 |
| valor 5 | 0,005445194 | 0,002603961 | 0,003322420 | 0,003551844 |
|  |  |  |  |  |
| **máx.** | 0,005445194 | 0,005498928 | 0,004312564 | 0,003551844 |
| **min** | 0,001154968 | 0,002603961 | 0,002165641 | 0,002575584 |
| **promedio** | 0,002664094 | 0,003554741 | 0,002897502 | 0,003107365 |

Servicio 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 3 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor 1 | 0,001066217 | 0,002037042 | 0,000953316 | 0,000817473 |
| valor 2 | 0,001979083 | 0,002049117 | 0,001791921 | 0,001230436 |
| valor 3 | 0,001395863 | 0,002338916 | 0,001702566 | 0,000959958 |
| valor 4 | 0,001820901 | 0,001455031 | 0,000839208 | 0,000711817 |
| valor 5 | 0,001954543 | 0,001333677 | 0,002267070 | 0,000800568 |
|  |  |  |  |  |
| **máx.** | 0,001979083 | 0,002338916 | 0,002267070 | 0,001230436 |
| **min** | 0,001066217 | 0,001333677 | 0,000839208 | 0,000711817 |
| **promedio** | 0,001643321 | 0,001842757 | 0,001510816 | 0,000904050 |

Iteraciones

En las iteraciones se repiten los resultados de tiempo, aumenta considerablemente la carga de estructura al variar el tamaño y no así para los servicios.

Carga:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Carga | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor | 4613 | 1974202 | 256002366 | 1715806697 |

En el servicio 1, se probó con el mismo género en un principio, pero al notar que se repetía le número obtenido en 2 tamaños distintos de archivo, se probó con un género diferente en el tamaño 4. El número final no varía demasiado, pero se observa la diferencia.

Servicio 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 1 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor | 36 | 43 | 43 | 45 |

En los 2 servicios restantes, se observa diferencia entre el tamaño 1 y los demás, pero estos últimos presentan resultados similares.

Servicio 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 2 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor | 147 | 2756 | 3234 | 3234 |

Servicio 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Servicio 3 | | | |
|  | tamaño 1 | tamaño 2 | tamaño 3 | tamaño 4 |
| valor | 474 | 2397 | 3582 | 2790 |

**Conclusiones**

Primera etapa

Como puede verse en las tablas y gráficos, ambos procesos son afectados por el tamaño del archivo. La carga de la estructura representa una tarea compleja que requiere más tiempo, sin embargo, este proceso se realizará sólo en escasas ocasiones para aumentar la colección mientras que las búsquedas serán repetidas múltiples veces y no requieren más que un par de segundos para generar el archivo de salida en colecciones muy grandes y apenas unas milésimas para colecciones más acotadas.

Segunda etapa

Al igual que en la primera, la carga de la estructura es la que más tiempo consume. Pero nuevamente, esta tarea debe realizarse sólo en determinadas ocasiones al cargar estadísticas de búsqueda. Una vez organizada esta información, la búsqueda de datos específicos se realizará múltiples veces y esta no depende de la cantidad de información sino de la porción que se desee analizar. Cuanto más “conectado” esté el género en cuestión, más tardará el servicio en devolver los datos. Aún así, en ninguna prueba de servicios se llegó ni de forma cercana al segundo, manteniendo los tiempos de ejecución siempre en milisegundos.

1. Aclaración: el archivo de tamaño 1.000.000 no se encuentra en el repositorio de github ya que el mismo no permitió la subida debido al peso del archivo. [↑](#footnote-ref-1)
2. Aclaración: el archivo de mayor tamaño no se encuentra en el repositorio de github ya que el mismo no permitió la subida debido al peso del archivo. [↑](#footnote-ref-2)