PRÁCTICA 03

Utilización de Estructuras de Datos Asociativas en Java

Objetivos

- Estudiar los conjuntos y los mapas basados en árboles y tablas hash para la resolución de diferentes problemas (desde los puntos de vista de almacenamiento y manipulación de datos).
- Aprender a utilizar la composición de estructuras de datos asociativas en la resolución de problemas.

Requisitos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos asociativas *TreeSet<T>*, *TreeMap<K, V>, HashSet<T> y HashMap<K,V>*, así como la combinación de ellas para la resolución de problemas.
- Conocer cómo realizar consultas eficientes sobre estructuras de datos asociativas, así como la mejor forma de plantearlas en problemas sencillos.

Enunciado

Parte 01. Gestión de ciudades donde empresas de desarrollo implementan productos software.

Disponemos de un conjunto de datos que queremos gestionar de forma eficiente mediante el uso de estructuras de datos adecuadas (ver archivo de texto datos.txt y prestad atención al formato). Estos datos están relacionados con un conjunto de empresas que desarrollan proyectos software en determinados lugares donde se ubican las sedes de dichos proyectos….

Os suena este problema, ¿verdad?

Se trata del caso que hemos resuelto en la primera parte de la práctica 02. Como ya sabéis, en aquella ocasión tuvimos que implementar 3 clases: *Proyecto*, *Empresa* y *GestionEmpresas*. En esta ocasión vamos a plantear el problema haciendo uso de estructuras de datos asociadas "anidadas", en concreto:

TreeMap<String, TreeMap<String, TreeSet<String>>> datos

Donde la primera clave corresponde con el identificador de empresa, la segunda con el identificador de proyecto y, asociado a cada par, tendremos un conjunto de ciudades. ¿Somos capaces de dibujar la estructura? Esta propuesta basada en la anidación de estructuras la podemos "leer" de la siguiente manera:

La estructura (que la hemos llamado datos) está compuesta por un conjunto de pares (clave, valor), donde la clave es *empresaID* (de tipo *String*) y como valor tiene asociado un conjunto de proyectos. ¿Cómo se representan los proyectos? Pues, como bien se muestra en la definición de la estructura, proyectos sería otro conjunto de pares (clave, valor) donde, en esta ocasión, la clave es proyectoID (de tipo *String*), teniendo como valor un conjunto ordenado de ciudades (cada ciudad de tipo String).

Al tratarse de objetos con clave única (empresaID, proyectoID, ciudad) se va a poder resolver el problema sin necesidad de crear más clases que la que encapsula la estructura de datos principal. Por lo tanto, el problema se reduce drásticamente teniendo que implementar únicamente la clase *GestionEmpresas*.

Para más detalles, consultad el test (esta vez hay un único subtest), así como la base del programa que se adjunta en el repositorio de la asignatura.

Parte 02. Gestión de un repositorio de artículos científicos: recuperación automática de palabras clave.

Según el diccionario de la RAE (Real Academia Española), *Repositorio* se define como "lugar donde se almacena algo". Y suena lógico pensar que si queremos almacenar cosas, objetos, es porque queremos recuperarlos, y todo de la forma más rápida posible. Pero para eso tendremos que almacenarlos siguiendo un esquema de organización óptimo que dé lugar a un proceso de manipulación eficiente. Esta es la idea.

En esta práctica vamos a trabajar con un tipo especial de repositorio en el que una Institución (como la UAL, por ejemplo) va a poder almacenar, organizar y proveer un mecanismo de gestión digital de artículos científicos con la única idea de favorecer una difusión basada en contenido entre la comunidad científica internacional. Una de las características fundamentales del prototipo de sistema software que vamos a implementar es su orientación hacia los autores, es decir, más que implementar un sistema simple de gestión de recuperación de artículos, nuestro sistema software va a basarse en el contenido de dichos artículos para caracterizar a los autores (investigadores) según un conjunto de palabras clave. Otras operaciones interesantes consisten en la obtención del conjunto de artículos (sus títulos) que ha publicado un autor o bien la obtención del conjunto de autores con los que ha trabajado (coautores).

Pero, ¿qué es y qué va a ser para nosotros un artículo científico? En la guía para la redacción de artículos científicos publicados por la UNESCO (podéis consultar en la web), un artículo científico se define como un informe escrito y publicado que describe resultados originales de una investigación. La publicación es uno de los métodos inherentes al trabajo científico. Lo que se investiga y no se escribe, o se escribe y no se publica, equivale a que no se investiga. Y resaltamos la siguiente frase, ya que lo mismo ocurre en programación:

"Se escribe para el mundo, no para uno mismo"

La estructura básica de un artículo científico es la siguiente:

- Título (articuloID)
- Autor/autores
- Resumen (abstract)

- Palabras clave (del artículo)
- Contenido
- Bibliografía

Aunque, para nosotros, la estructura básica va a ser la que se propone a continuación:
- Título (articuloID)
- Contenido (texto)

Como veremos más adelante, la lista de autores la vamos a gestionar de forma externa y no como un conjunto de atributos de la clase que modela un artículo.

Y, en segundo lugar, ¿qué es y qué va a ser para nosotros una palabra clave? En este contexto, una palabra clave (*keyword*) es un término significativo (compuesto por una o más palabras) que describe, en esencia, de qué trata el artículo. Estas palabras claves son las que los investigadores suelen utilizar para buscar artículos en motores de búsqueda. Así pues, la elección de una o varias palabras clave va a ser una tarea esencial.

Como se ha comentado anteriormente, nuestro sistema va a estar orientado al autor. Esto implica que las palabras clave de los artículos las vamos a ignorar y vamos a centrarnos en la obtención (de forma automática) de palabras clave a partir de textos. Así pues, para nosotros, una palabra clave va a ser una palabra frecuente, obtenida de forma automática a partir del conjunto de artículos publicados por dicho autor. Una palabra se considera frecuente si su frecuencia (número de ocurrencias...las veces que aparece en el texto) es mayor o igual que el valor de un parámetro establecido por el usuario. A la hora de contar palabras para determinar si son frecuentes, o no, eliminaremos aquellas que se definen como "palabras vacías" (*stopwords*), que son palabras que carecen de significado y que, por tanto, no van a servir para caracterizar a un autor según su trabajo. Ejemplos de *stopwords*: determinantes, preposiciones, pronombres, … (ver clase Auxiliar.java).

La generación automática de palabras clave a partir de textos es un problema muy interesante que pertenece a un área de la Inteligencia Artificial conocida como Procesamiento del Lenguaje Natural (NLP - acrónimo del inglés *Natural Language Processing*) con amplio uso en tareas de análisis de datos, monitorización de redes sociales, marketing, inteligencia de negocio, y un largo etcétera...

Resumiendo: nuestra aplicación va a gestionar un conjunto de repositorios en los que se almacenan artículos, escritos por autores, de manera que:

N repositorios
1 repositorio → N artículos
1 artículo → 1 repositorio
1 autor → N artículos
1 artículo → N autores

Empezando por la relación más anidada, autor y artículos forman un par formado por la clave=artículo y valor=colección de autores. Y, como nuestro sistema va a almacenar una colección de artículos, la estructura base podría ser:

TreeMap<Articulo, colección de autores>

El autor va a estar identificado por su nombre, es decir, su nombre será su clave (*autorID, de* tipo *String*) y, por lo tanto, no puede haber dos autores diferentes con el mismo nombre.

Por otro lado, un artículo va a estar identificado por su título (*articuloID*) y va a tener asociado un texto (de tipo *String*), de manera que:

1 artículo → articuloID + texto

Entonces, en un principio, la estructura podría ser:

TreeMap<Articulo, String>

El tipo de datos *Articulo* sería una clase que encapsula dos atributos: (1) su clave (articuloID) y un texto de tipo String (o bien una colección de cadenas de textos).

Sin embargo, vamos a tener en cuenta los siguientes aspectos:

- De los artículos (su texto) nos interesan únicamente las palabras, no su estructura, y nos interesa ya que, a partir de ellas, vamos a obtener las palabras clave (palabras frecuentes) que caracterizan a los autores.
- Hay que tener en cuenta que en la fase de conteo se ignoran cuenta las palabras vacías (*stopWords*).
- Habrá que pensar que este proceso de conteo deberá combinar, en una estructura, todos los pares (palabra, frecuencia) asociados con todos y cada uno de los artículos que ha publicado un autor.

Así pues, parece interesante que el texto no se represente con un simple String, sino que el proceso de inserción de una palabra en la estructura, y su conteo, se haga de forma simultánea. Una posible propuesta podría ser que el atributo texto se modele como una colección de pares (palabra, frecuencia). ¿Qué tal resulta esta propuesta desde el punto de vista de eficiencia? La estructura texto podría ser:

texto → *TreeMap<palabra, frecuencia>*

Entonces, el tipo *Articulo* estaría formado por los atributos:

- articuloID (su clave)
- TreeMap<String, Integer> (clave palabra y valor frecuencia)

Y, por último, tenemos que tener en cuenta que vamos a gestionar un conjunto de **repositorios** (identificados únicamente por su clave, repositorioID, de tipo *String*). Cada artículo va a pertenecer a un repositorio (o varios).

Así pues, la estructura definitiva sería una colección pares (clave, valor), donde clave sería *repositorioID* y valor estaría formado, a su vez, por una colección de pares (clave, valor), donde clave es el *Articulo* (clave no predefinida) y valor sería un conjunto de autores. La idea de crear la clase *Articulo* es debido a que se trata de una entidad con unos atributos y un comportamiento muy determinado.

Nuestra estructura de datos sería:

TreeMap<String, TreeMap<Articulo, TreeSet<String>>>

donde Articulo se define como el conjunto de atributos:

- articuloID
- TreeMap<String, Integer> (conjunto ordenado de pares <palabra, frecuencia>)

Obviamente, la clase Articulo debe implementar la interfaz *Comparable<Articulo>*. ¿O no?

Para más detalles, consultad el conjunto de tests, así como la base del programa que se adjunta en el repositorio de la asignatura.

¿Qué solución alternativa proponéis?

Parte 03. Gestión de *HashTags*

¿Qué es un *hashtag*? Hoy en día es difícil que alguien no conozca este término y en qué ámbitos se utiliza. Según la RAE, se aconseja utilizar el término etiqueta, aunque se admite que es muy común el uso del anglicismo. Básicamente, un *hashtag* consiste en una secuencia de caracteres precedido por el carácter # (numeral o almohadilla) y sirve para etiquetar o identificar un mensaje en las redes sociales. Aunque está mayormente asociado con *Twitter*, se utiliza también en *Instagram*, *Facebook* y otras redes sociales. El uso de esta etiqueta es variado aunque, de forma muy resumida, se va a utilizar como palabra clave para indicar el tema (contexto) del mensaje que está publicando. De esta forma, a partir de estos índices se podrán agrupar mensajes, procesarlos, analizarlos y clasificarlos con muy diferentes propósitos como, por ejemplo, aplicar técnicas de "análisis de sentimientos". Estas técnicas pertenecen al campo de la inteligencia artificial y, básicamente, parten del conjunto de mensajes asociados con una etiqueta determinada (cuyo tamaño puede ser enorme), para determinar, de forma automática, cuál es el sentimiento (opinión o impresión general) que se puede inferir a partir del contenido de los mensajes asociados con la etiqueta: positivo, negativo…

Aquí surge un área de gran interés del que podríamos estar hablando durante mucho tiempo aunque, a falta de este recurso tan esencial, vamos a simplificar el problema. Así pues, ¿qué va a ser para nosotros un *hashtag* y en qué contexto lo vamos a utilizar? La idea consiste en asociar una etiqueta con el conjunto de mensajes que la ha incluido. Los mensajes (con un número de caracteres muy limitado) se clonarán para asociarlos a cada etiqueta que incluya. Vamos a suponer que hemos procesado el conjunto de mensajes de origen, extraído los identificadores de etiqueta utilizados y vamos a tener muy en cuenta que nos interesa conocer el país de origen de los mensajes. Vamos a pensar también que hemos traducido, tanto etiquetas como mensajes, en caso de que estén escritos en distintos idiomas.

Así pues, vamos a elegir una colección que nos permita asociar una etiqueta (hashtag) con un conjunto de mensajes. Para almacenar de forma compacta y asociar etiquetas con mensajes vamos a elegir una colección mapa basada en tabla hash (*HashMap<K,V>*), donde la clave será el tipo *HashTag* y valor será una referencia a una colección de objetos de tipo *String*. La elección de la colección principal no tiene nada que ver con el término inglés HashTag...es una simple coincidencia.

HashMap<HashTag, ArrayList<String>> datos

Nuestro tipo **HashTag** va a tener los siguientes atributos:

- hashTagID: su identificador, de tipo String. Si originalmente va precedido con el símbolo #, se elimina.
- paisID: identificador del país de origen de la etiqueta que estamos analizando (es = España, it = Italia, …).
- isTrendingTopic: valor booleano que nos va a indicar si la etiqueta ha sido tendencia en el país de origen durante un intervalo de tiempo determinado.

Como se ha indicado anteriormente, el país de origen también va a ser un atributo clave. Es decir, voy a poder tener dos etiquetas con el mismo identificador de etiqueta, pero con identificador de país diferente. Dicho de otra manera, el par (*hashTagID*, *paisID*) serán los atributos clave del tipo *HashTag*.

De esta forma: *(#SaveTheWorld, es, true)* y *(#SaveTheWorld, it, true)* son etiquetas diferentes y, por lo tanto, van a poder tener asociadas una colección diferente de mensajes.

Como sabemos que vamos a utilizar el tipo *HashTag* como clave en una estructura *HashMap<K,V>* , vamos a tener en cuenta que *HashMap<K,V>* **NO** es una estructura ordenada y, por lo tanto, *HashTag* no tiene por qué implementar la interfaz *Comparable<T>* (que si lo hace, no pasa nada). Sin embargo, ¿cómo le vamos a decir a Java que nuestra clave está compuesta por el par *<hashTagID, paisID>*?

En este caso, en lugar de implementar el método *compareTo()* para decirle a Java cuál es la lógica de comparación de elementos para su posterior ordenación, le vamos a decir cómo debe obtener el valor hash asociado a la clave ya que Java utiliza este valor hash para indexar la estructura de base. Y para ello existe el método *hashCode()*.

Hasta ahora siempre hemos dicho que el método *equals()* debe ser coherente con el método *compareTo()*, de manera que nuestra implementación, simplificada al máximo, se basaba en el hecho de que dos objetos son iguales sii *compareTo() = 0*. En este caso, *equals()* no va a ser coherente con *compareTo()*, sino con *hashCode()*, de manera que dos objetos serán iguales sii tienen, exactamente, el mismo valor hash.

Para más detalles de implementación, consultad la base del programa que se adjunta en el repositorio de la asignatura. Como veréis, en este caso no existe ningún test de prueba JUnit, sino que añadimos un test por clase especificado en el método main()

correspondiente. Observad que las restricciones (*assertEquals()*, *assertTrue()*...) las hemos incluido en el método principal sin diferencia aparente. Sin embargo, ahora ya no existe el "semáforo verde" → en este caso, su equivalente será el mensaje **TODO OK!!!!!**. Si alguna restricción no se cumple, se lanzará una excepción, rompiendo la ejecución normal del programa. Si todo va bien, el programa terminará mostrando este mensaje en Consola.

Por lo tanto, semáforo verde equivale a: