Estructuras de Datos y Algoritmos I
Grado en Ingeniería Informática, Curso 2º

ACTIVIDAD 03

Tablas. Aplicación de TreeSet<T> y TreeMap<K,V>

Objetivos

- Estudiar en profundidad los TreeSet<T> y TreeMap<K,V> como estructuras de datos asociativas.

Requerimientos

Para superar esta actividad se debe realizar lo siguiente:

- Desarrollar la solución a problemas sencillos, haciendo uso de `TreeSet<T>` o `TreeMap<K,V>`.

Enunciado

En esta actividad vamos a complementar lo estudiado en las clases de teoría, con la utilización de `estructuras asociativas` en ejemplos concretos.

- `TreeSet()` ⇒ Construye un nuevo conjunto vacío, ordenado según el orden natural de los elementos.
- `TreeSet(Comparator c)` ⇒ Construye un nuevo conjunto vacío, ordenado según el `Comparator` especificado.
- `boolean add(Object o)` ⇒ Añade el elemento `o` al conjunto si no está presente.
- `void clear()` ⇒ Elimina todos los elementos del conjunto.
- `Object clone()` ⇒ Devuelve una copia de esta instancia `TreeSet<T>`.
- `boolean contains(Object o)` ⇒ Devuelve true si el conjunto contiene el elemento `o`.
- `Object first()` y `Object last()` ⇒ Devuelve el primer y último elemento que se encuentra actualmente en este conjunto ordenado.
- `boolean remove(Object o)` ⇒ Elimina el elemento del conjunto si éste está presente.
- `int size()` ⇒ Devuelve el número de elementos del conjunto.

- `TreeMap()` ⇒ Construye un nuevo mapa vacío, ordenado según el orden natural de la clave.
- `TreeMap(Comparator c)` ⇒ Construye un nuevo mapa vacío, ordenado según el `Comparator` especificado.
- `TreeMap(Map m)` ⇒ Construye un nuevo mapa que contiene los mismos pares (correspondencias) que el mapa `m`, y ordenando con el orden natural de la clave.
- `void clear()` ⇒ Elimina todos los pares (correspondencias) clave-valor del `TreeMap`.
- `Object clone()` ⇒ Devuelve una copia de esta instancia `TreeMap`.
- `boolean containsKey(Object key)` ⇒ Devuelve true si este mapa contiene un par clave-valor (correspondencia) para la clave `key` especificada.
- `boolean containsValue(Object value)` ⇒ Devuelve true si este mapa hace corresponder una o más claves con el valor `value` especificado.
- `Object firstKey()` y `Object lastKey()` ⇒ Devuelve la primera y la última clave que este mapa ordenado contenga actualmente.
- `Object get(Object key)` ⇒ Devuelve el valor correspondiente con la clave `key` dada, en este mapa.
- `Object put(Object key, Object value)` ⇒ Asocia en el mapa el valor especificado con la clave dada por `key` y el valor dado por `value`.
- `boolean remove(Object key)` ⇒ Elimina del `TreeMap<K,C>` el par clave-valor (correspondencia) asociado a la clave dada `key`, si éste está presente.
- `int size()` ⇒ Devuelve el número de pares clave-valor (correspondencias) que hay en el mapa.
- `Collection values()` ⇒ Devuelve una vista, en forma de colección, de los valores contenidos en el mapa.

Ejercicio 1. Implementación de un sencillo corrector ortográfico (`SpellChecker`) utilizando `TreeSet<T>`.

Una de las características más útiles de los procesadores de textos modernos es la corrección ortográfica. Para ello, suele ser una tarea común escanear un documento para detectar posibles faltas de ortografía. Decimos *posible faltas de ortografía* ya que el documento puede contener palabras que sean legales pero que no se encuentren en un diccionario. Por ejemplo, *iterator* y *postorden* han sido detectados como que no se encuentran en el diccionario que utiliza el procesador de textos (Microsoft Word)…aunque en esta asignatura se hace un uso extensivo de estos términos.

El problema que se plantea en esta actividad, en general, es que dado un diccionario (`dictionary.txt`) y un documento (`document.txt`), mostrar todas las palabras del documento que no se encuentren en el diccionario (objetivo del método `compare()`), junto con algunas funcionalidades adicionales (que se indicarán en la propuesta de clase `SpellChecker`). Para ello vamos a presentar unas sencillas restricciones del problema a resolver, como son:

- El diccionario se compone únicamente de palabras en minúsculas, una por línea (sin definiciones).
- Las palabras en el documento están separadas entre sí por, al menos, un carácter no alfabético (como un símbolo de puntuación, un espacio en blanco, o un marcador de fin de línea).
- El archivo del diccionario está en orden alfabético.
- El archivo del documento, que puede estar en también orden alfabético, se almacenará en memoria (junto con el diccionario) durante el proceso de comparación (`compare()`).

Ejercicio 2. Implementar un sencillo Tesauro (Thesaurus) utilizando `TreeMap<K,V>` y `TreeSet<T>`.

Como sabemos, un tesauro es sencillamente un diccionario de sinónimos. El problema que queremos resolver en esta actividad consiste en realizar una serie de actividades sobre un archivo de sinónimos, en el que cada línea consiste en una ***palabra base*** seguida de una ***lista de sinónimos*** (separados por espacios en blanco). Por ejemplo, construir el tesauro (almacenado en un `TreeMap` de `TreeSet`) en base a dicho archivo (`Thesaurus.txt`), añadir sinónimos al tesauro (`add()`); eliminar sinónimos del tesauro (`remove()`); actualizar lista de sinónimos (`update()`); decir si una palabra es sinónimo de una palabra base (`isSynonymousOf()`); si una palabra es sinónimo de cualquier palabra de tesauro (`isSynonymous()`); si una palabra base tiene sinónimos (`hasSynonymous()`); obtener la lista de sinónimos de una palabra base (`getSynonymous()`), etc. Además, el archivo de sinónimos estará en orden alfabético a excepción de la palabra base. Tened en cuenta este hecho y pensad cómo lo podríamos resolver, dada la propiedad de los TreeSet<T>.

Formato del archivo de entrada:

$Palabra_{Base1}$: {$sinonimo_1$, $sinonimo_2$,…, $sinonimo_{n1}$}
$palabra_{Base2}$: {$sinonimo_1$, $sinonimo_2$,…, $sinonimo_{n2}$}
…
$palabra_{Basem}$: {$sinonimo_1$, $sinonimo_2$,…,$sinonimo_{nm}$}

Es posible la existencia de comentarios dentro del archivo de entrada. Estos comentarios se identifican con el carácter @ al comienzo de la línea y, como es obvio, serán ignorados.