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

ACTIVIDAD 4

Tablas. Aplicación de TreeSet y TreeMap

Objetivos

- Aprender a utilizar las tablas hash como estructura de datos y conocer sus características principales.
- Estudiar los TreeSet y TreeMap como estructuras de datos asociativas.
- Conocer las características diferenciadoras entre los Tree* y Hash*.

Requerimientos

Para superar esta actividad se debe realizar lo siguiente:

- Dominar cómo implementar una tabla hash y el tratamiento de colisiones.
- Desarrollar la solución a problemas sencillos, haciendo uso de `TreeSet` y `TreeMap`.

Enunciado

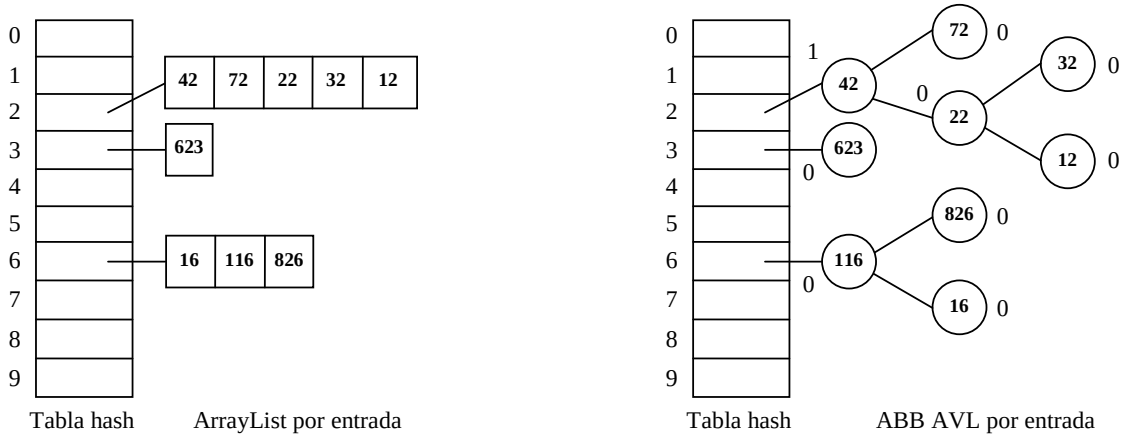
En esta actividad vamos a complementar lo estudiado en las clases de teoría, con la utilización de `tablas hash`, `TreeSet` y `TreeMap` en ejemplos concretos.

- `TreeSet()` ⇒ Construye un nuevo conjunto vacio, ordenado según el orden natural de los elementos.
- `TreeSet(Comparator c)` ⇒ Construye un nuevo conjunto vacio, 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`.
- `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.
- `Iterator iterator()` ⇒ Devuelve un `iterador` para los elementos de este conjunto.
- `boolean remove(Object o)` ⇒ Elimina el elemento o del conjunto si éste está presente.
- `int size()` ⇒ Devuelve el número de elementos del conjunto.

- `TreeMap()` ⇒ Construye un nuevo mapa vacio, ordenado según el orden natural de la clave.
- `TreeMap(Comparator c)` ⇒ Construye un nuevo mapa vacio, 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` el par clave-valor (correspondencia) asociado a la clave dada `key`, si éste está presente.
- `int size()` ⇒ Devuelve el nº 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.

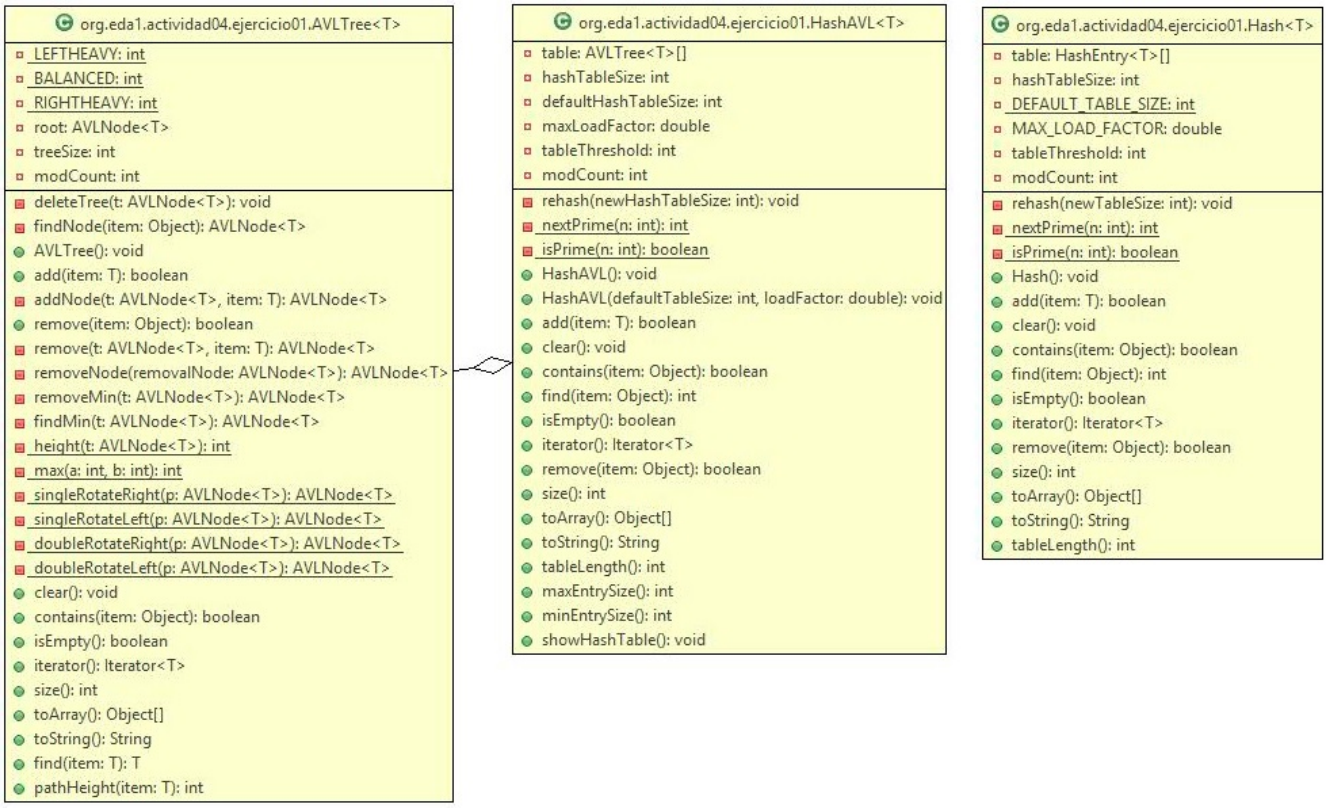
Ejercicio 1. Hemos estudiado en teoría las *tablas hash*, y se ha proporcionado, en los archivos `Hash.java` y su correspondiente `Principal.java`, la implementación de una tabla hash con encadenamiento (tabla hash encadenada) para que se estudie y comprenda su funcionamiento. Ahora se proporciona una nueva implementación de *tabla hash con encadenamiento*, en la que se utiliza un `ArrayList` para cada entrada de la tabla hash (`HashArrayList.java`).

En esta actividad se pide estudiar `HashArrayList.java` y a continuación implementar en Java una tabla hash para almacenar un conjunto de elementos genéricos (`T`), en la que cada entrada de la nueva tabla hash (`HashAVL.java`) en lugar de ser `ArrayList` de elementos; éstas serán ABB AVL (`AVLTree.java`), extendiendo dicha implementación si es necesario. Las siguientes figuras muestran ejemplos de los esquemas hash que indican cómo deben ser las estructuras de datos para esta actividad.



Recordar que el *factor de carga* (α) indica la relación entre el número de elementos almacenados y el tamaño de la tabla, ***α = n/max***. Y para el caso de una tabla hash con encadenamiento nos indica el nivel de ocupación de la tabla. Si α = 1 ⇒ (n = max) hay igual número de elementos que de entradas en la tabla hash; si α < 1 ⇒ hay menos elementos que entradas, y si α > 1 ⇒ hay más elementos que entradas. Es decir, conforme aumenta α (> 1) la tabla hash crece a lo ancho (en el número de elementos de la ED encadenada) y en nuestros test utilizaremos valores de α (5 y 100) para este fin.

Una vez implementado todo debe comprobar su correcto funcionamiento con el **test** correspondiente. Explique de forma razonada toda la implementación de código realizada.


org.eda1.actividad04.ejercicio01.AVLTree<T>
LEFTHEAVY: int
BALANCED: int
RIGHTHEAVY: int
root: AVLNode<T>
treeSize: int
modCount: int
deleteTree(t: AVLNode<T>): void
findNode(item: Object): AVLNode<T>
AVLTree(): void
add(item: T): boolean
addNode(t: AVLNode<T>, item: T): AVLNode<T>
remove(item: Object): boolean
remove(t: AVLNode<T>, item: T): AVLNode<T>
removeNode(removalNode: AVLNode<T>): AVLNode<T>
removeMin(t: AVLNode<T>): AVLNode<T>
findMin(t: AVLNode<T>): AVLNode<T>
height(t: AVLNode<T>): int
max(a: int, b: int): int
singleRotateRight(p: AVLNode<T>): AVLNode<T>
singleRotateLeft(p: AVLNode<T>): AVLNode<T>
doubleRotateRight(p: AVLNode<T>): AVLNode<T>
doubleRotateLeft(p: AVLNode<T>): AVLNode<T>
clear(): void
contains(item: Object): boolean
isEmpty(): boolean
iterator(): Iterator<T>
size(): int
toArray(): Object[]
toString(): String
find(item: T): T
pathHeight(item: T): int
org.eda1.actividad04.ejercicio01.HashAVL<T>
table: AVLTree<T>[]
hashTableSize: int
defaultHashTableSize: int
maxLoadFactor: double
tableThreshold: int
modCount: int
rehash(newHashTableSize: int): void
nextPrime(n: int): int
isPrime(n: int): boolean
HashAVL(): void
HashAVL(defaultTableSize: int, loadFactor: double): void
add(item: T): boolean
clear(): void
contains(item: Object): boolean
find(item: Object): int
isEmpty(): boolean
iterator(): Iterator<T>
remove(item: Object): boolean
size(): int
toArray(): Object[]
toString(): String
tableLength(): int
maxEntrySize(): int
minEntrySize(): int
showHashTable(): void
org.eda1.actividad04.ejercicio01.Hash<T>
table: HashEntry<T>[]
hashTableSize: int
DEFAULT_TABLE_SIZE: int
MAX_LOAD_FACTOR: double
tableThreshold: int
modCount: int
rehash(newTableSize: int): void
nextPrime(n: int): int
isPrime(n: int): boolean
Hash(): void
add(item: T): boolean
clear(): void
contains(item: Object): boolean
find(item: Object): int
isEmpty(): boolean
iterator(): Iterator<T>
remove(item: Object): boolean
size(): int
toArray(): Object[]
toString(): String
tableLength(): int

Ejercicio 2. Implementar un sencillo corrector ortográfico (`SpellCheckeer`) utilizando `TreeSet`.

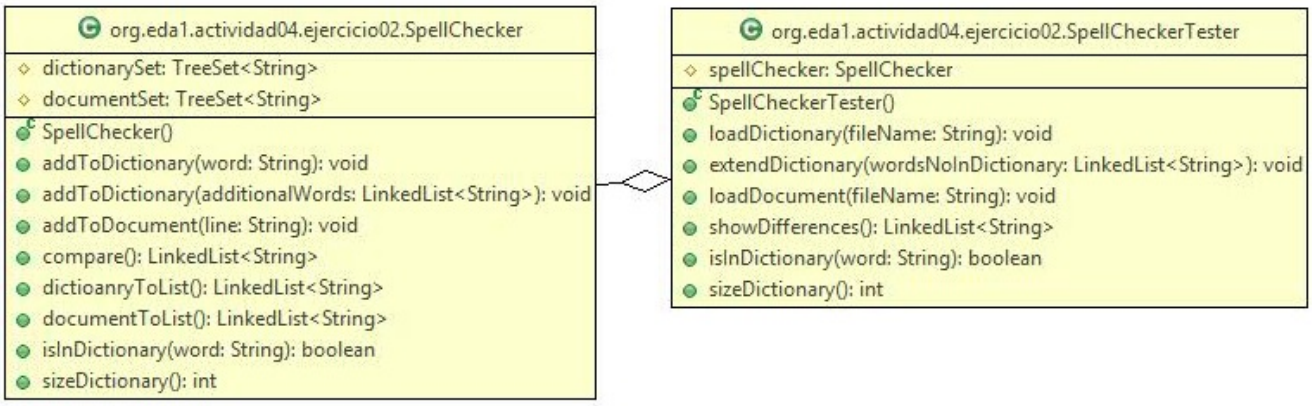
Una de las características más útiles de los procesadores de textos modernos es la corrección ortográfica, para ello, suele ser una tarea común escanear un documento para detectar posibles faltas de ortografía. Decimos *posible faltas de ortografía* ya que el documento puede contener palabras que sean 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), y que se emplea para escribir esta actividad.

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 encuentran en el diccionario (`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 que:

- El diccionario se compone únicamente de palabras 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()`).

A nivel orientativo para resolver el ejercicio y el test asociado al mismo se plantea la siguiente clase.

```
public class SpellChecker {
   protected TreeSet<String> dictionarySet;
   protected TreeSet<String> documentSet;
   public SpellChecker() { … }
   public void addToDictionary(String word) { … }
   public void addToDictionary(LinkedList<String> additionalWords) { … }
   public void addToDocument(String line) { … }
   public LinkedList<String> compare() { … }
   public LinkedList<String> dictionaryToList() { … }
   public LinkedList<String> documentToList() { … }
   public boolean isInDictionary(String word) { … }
   public int sizeDictionary() { … }
}
```



Ejercicio 3. Implementar un sencillo tesauro (Thesaurus) utilizando `TreeMap` y `TreeSet`.

Como sabemos, un tesauro es sencillamente un diccionario de sinónimos. El problema que queremos resolver en esta actividad es que dado un archivo de sinónimos, teniendo en cada línea una *palabra base* y después una *lista de sinónimos* separados por espacios en blanco, se van a realizar una serie de operaciones sobre él. 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 y la lista de sinónimos asociados a una palabra base no, aunque al cargarlo en memoria sí debería estarlo al utilizar para ello un `TreeSet`.

A nivel orientativo para resolver el ejercicio y el test asociado al mismo se plantea la siguiente clase.

```
public class Thesaurus {
   protected TreeMap<String, TreeSet<String>> thesaurusMap;

   public Thesaurus() { … }
   public void add(String line) { … }
   public void add(String word, String synonym) { … }
   public TreeSet remove(String word) { … }
   public boolean remove(String word, String synonym) { … }
   public TreeSet<String> update(String word, LinkedList<String> synonyms) { … }
   public int size() { … }
   public TreeSet<String> getSynonymous(String word) { … }
   public int size(String word) { … }
   public boolean isSynonymousOf(String word, String synonym) { … }
   public boolean isSynonymous(String synonym) { … }
   public boolean hasSynonymous(String word) { … }
   public String showThesaurus() { … }
}
```