PRÁCTICA 2

Uso de Árboles Binarios de Búsqueda en Java

Objetivos

- Utilizar los Árboles Binarios de Búsqueda (ABB) para la resolución de diferentes problemas, tanto para el almacenamiento de datos como para la implementación eficiente de algoritmos de manipulación.
- Aprender a utilizar la composición de estructuras de datos arborescentes en la resolución de problemas.
- Utilizar los ABB balanceados (AVL) para la resolución de problemas, comprobando las ventajas de su uso en determinadas situaciones.

Requisitos

Para superar esta práctica se debe realizar lo que se indica a continuación:

- Dominar el uso de las estructuras de datos arborescentes `BSTree<T>` y `AVLTree<T>`, así como la combinación de ellas para la resolución de problemas.
- Saber contestar adecuadamente a las preguntas que se proponen.
- Pasar los respectivos test que se suministran.

Enunciado - Parte 1

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.

Como **EJERCICIO** se pide lo siguiente: Hay que implementar un programa en Java que lea desde un archivo de entrada la anterior lista formada por 3-uplas: **Empresa - Proyecto - Ciudad**, almacenando los datos en una combinación de estructuras de datos arborescentes.

- Según se lee cada línea del archivo (**Empresa - Proyecto - Ciudad**), el programa debe comprobar si dicha 3-upla ya está en el contenedor. Si lo está, no hay que hacer nada ya que se trata de una línea repetida; en caso contrario, se insertará en la estructura de datos combinada.
- Una vez que todas las líneas (tripletas) han sido leídas del archivo de entrada y almacenadas en memoria en la estructura de datos basada en **Árboles Binarios de Búsqueda**, se implementará una serie de métodos de consulta que detallaremos a continuación.

Como sugerencia para la realización del ejercicio, podemos plantear una estructura de datos

basada en la combinación de estructuras de datos arborescentes: **BSTree<T>** y **AVLTree<T>**. De forma genérica podría plantearse lo siguiente (árboles de árboles de árboles):

AVLTree(Empresas, **AVLTree**(Proyectos, **BSTree**(Ciudades)))

Además, una vez en organizados los datos en la ED indicada anteriormente, se debe responder (con la implementación de la correspondiente función) a las siguientes consultas (ya contempladas en el test):

- Devolver las empresas que tienen su sede en una ciudad determinada.
- Devolver los *proyectos* con sede en una ciudad especificada como parámetro de entrada.
- ¿En cuántas ciudades diferentes se desarrollan proyectos de **una empresa determinada**?.
- ¿En cuántas ciudades diferentes se desarrolla **un determinado proyecto**?
- ¿Cuántos proyectos está desarrollando una empresa?

En primer lugar, y para comprobar la correcta realización de este ejercicio, se proporcionará el **test** que, como ya sabemos, se deberá pasar correctamente.

Adicionalmente, se ha proporcionado todo lo relativo a la implementación de los contenedores **BSTree<T>** y **AVLTree<T>**, por lo que resulta **INTERESANTE** que redactéis un documento (memoria) basado en detalles de la implementaciones proporcionadas, prestando especial atención a los métodos principales de dichas estructuras de datos como: *add()*, *remove()*, *clear()*, *contains()*, *isEmpty()*...*find()*, así como al uso propuesto de los iteradores. También se pide que penséis en las ventajas e inconvenientes de la resolución de este problema mediante el uso de estructuras arborescentes en lugar de colecciones lineales (**ArrayList<T>**). ¿Qué hubiese cambiado si nos centramos en la implementación? ¿Y en eficiencia?

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

Enunciado - Parte 2

AVLTree<T>: Uso de Árboles Binarios de Búsqueda Equilibrados para la gestión de una biblioteca.

En este ejercicio vamos a implementar un programa sencillo, y algo especial, de gestión de una biblioteca (o librería). Nuestra biblioteca está interesada en implementar, en un futuro cercano, un sistema de recomendación basado en similitud. De forma muy breve, podemos definir un sistema de recomendación como un producto software inteligente que personaliza la oferta que se le ofrece a un usuario. En este caso, vamos a sentar las bases de esta herramienta futura permitiendo el cálculo del grado de similitud entre dos usuarios en función de los libros que han leído (o, simplemente, prestado…).

Vamos a comentar algunos detalles importantes relativos a las clases *Libro*, *Usuario* y *Biblioteca*.

Clase Biblioteca

Nuestra biblioteca va a tener como identificador único (clave) su nombre (bibliotecaID), así como dos colecciones de libros y usuarios que, como podéis observar, se tratan de colecciones arborescentes tipo AVL (árboles binarios de búsqueda "bien" equilibrados). Del conjunto de operaciones que definen el comportamiento de esta clase, destacamos:

- *prestarLibro(String usuarioID, String libroID)*: se le presta, si es posible, el libro identificado por el parámetro *libroID* al usuario con clave *usuarioID*. No será posible si el libro ya está prestado o, simplemente, no está registrado el usuario o el libro.
- *devolverLibro(String ususarioID, String libroID)*: el usuario devuelve el libro, suponiendo que lo tenía prestado.
- *getLibrosPrestados()*: simplemente, devuelve el conjunto de libros que están prestados en la actualidad.
- *getUsuariosPrestamo(String libroID)*: devuelve el conjunto de usuarios que han leído el libro con clave *libroID*.
- *getLibrosPrestados(String usuarioID)*: devuelve el conjunto de libros que ha prestado un usuario con clave *usuarioID* (existen dos versiones de este métodos, el segundo acepta una referencia de tipo Usuario como parámetro de entrada).
- *getDistancia(String usuarioID1, String usuarioID2)*: devuelve la distancia entre dos usuarios. El cálculo de la distancia entre dos usuarios se basa en la suma de las distancias entre los libros que ambos han leído. El cómputo de la distancia entre dos libros se detalla en la clase *Libro*.
- *getSimilares(String usuarioID)*: devuelve el conjunto de usuarios, distintos a *usuarioID*, ordenados según el criterio de distancia ascendente (es decir, los primeros usuarios que aparecen en el conjunto serán los que más se "parezcan" al usuario con identificador *usuarioID*). Aquí tenemos que tener claro la relación entre similitud y distancia: dos objetos son más similares cuanto menos distantes son.

Clase Libro

El atributo clave es el libroID. El atributo *usuarioPrestamo* lo vamos a utilizar para indicar quién ha reservado el libro en cuestión. Si *usuarioPrestamo == null,* sabremos entonces que el libro no está prestado; en caso contrario, tendrá la referencia al objeto *Usuario* que tiene prestado el libro. Por otro lado, como vamos a obtener los valores de distancias entre libros, funcionalidad clave para nuestro futuro sistema de recomendación, va a ser necesario almacenar las palabras que contiene. Pero claro, en lugar de almacenar el conjunto completo de palabras, resultaría ineficiente, vamos a almacenar pares (palabra, frecuencia). Como hasta ahora no sabemos hacer uso de la clase *Pair<K,V>* (declarada en *edaAuxiliar*) vamos a resolver el problema creando la clase *PalabraFrecuencia*.

Ahora vamos a ver cómo podemos calcular la distancia entre dos libros. En primer lugar, hay que destacar que similitud y distancia son términos relacionados e inversamente proporcionales. A mayor distancia entre dos objetos, menor similitud, y viceversa. Para simplificar el problema, vamos a definir el grado de similitud entre dos libros como la distancia euclídea entre los vectores formados por la frecuencia de las palabras comunes. Si el valor de distancia es igual a 0 significa que, para nosotros, ambos objetos son iguales (no hay distancia entre ellos).

Siguiendo el ejemplo que veréis en el test, sean *libro1* y *libro2* dos referencias a objetos de tipo *Libro*, tal que:

libro1 → *[adios <2>, cruel <1>, mundo <1>]*
libro2 → *[adios <1>, cruel<1>, hola <1>, maravilloso <1>, mundo <2>]*

> **NOTA**: Esto significa que en el *libro1* aparece dos veces la palabra *adios*, 1 vez la palabra *cruel* y 1 la palabra *mundo*. De igual forma, en el *libro2* aparecería 2 veces la palabra *mundo* y 1 el resto de palabras.

La distancia entre ambos libros se calcularía de la siguiente forma:

$$distancia(libro1,\ libro2)\ =\ \sqrt{(2-1)^2\ +\ (1-1)^{\ 2} + (1-2)^2} = \sqrt{2}$$

> **NOTA:** Observad que únicamente tenemos en cuenta las palabras comunes, es decir, *adiós*, *cruel* y mundo. Para el cómputo de la distancia tendremos en cuenta los valores de *frecuencia* de dichas palabras comunes...

Clase Usuario

Además del identificador de usuario (su nombre), en la clase *Usuario* vamos a ir almacenando todos y cada uno de los libros que el usuario ha ido prestando a lo largo del tiempo (un histórico de libros leídos). Como podéis comprobar, se ha seleccionado un *AVLTree<Book>* como objeto contenedor del conjunto de libros prestados. Y una duda que surge aquí: ¿qué pasaría si el usuario reserva dos veces el mismo libro? ¿Qué cambios habría que hacer en la declaración de la clase si fuese importante tener en cuenta esta posibilidad?

Para más detalles de implementación, consultad el test, así como la base del programa que se encuentra en el repositorio de la asignatura.

Enunciado - Parte 3

AVLTreePair<K,V>: Árboles de pares (clave, valor) basado en AVL.

En este ejercicio vamos a implementar nuestra propia versión de mapa basado en árbol binario de búsqueda (al que llamaremos AVLTreePair<K,V>). De forma muy breve, podemos ver un mapa como un conjunto de pares (clave, valor), donde cada par va a representar un objeto con identificador *clave* (que debe ser Comparable<K>), asociado a un conjunto de atributos representados por el parámetro *valor* (puede ser un tipo simple, una referencia a un objeto o una combinación de estructuras…).

Por ejemplo, la estructura AVLTreePair<String, ArrayList<Double>> representaría un conjunto de pares con clave de tipo String y como valor tendría una referencia a un contenedor de tipo ArrayList<T>. Esta estructura podríamos utilizarla para representar, por ejemplo, a un conjunto de estudiantes y sus notas de prácticas. Si la clave es compuesta, entonces la clave sería una referencia a un objeto como, por ejemplo, AVLTreePair<Estudiante, ArrayList<Double>>.

Internamente, AVLTreePair<K,V> encapsula a un árbol AVL de pares, implementando un nuevo conjunto de métodos que le dan un comportamiento especial a este tipo de contenedor.

Es decir:

AVLTreePair<K,V> → AVLTree<Pair<K,V>> + métodos

El conjunto de operaciones (métodos) más interesantes que va a implementar nuestro árbol de pares es el siguiente:

Método put(): boolean ← put(K clave, V valor)

Este método es equivalente al método add() de la estructura AVLTree<T>. Inserta el par (clave, valor) con la siguiente funcionalidad: Si la clave no existe, inserta el par y devuelve true; en caso contrario, sustituye el valor antiguo por el nuevo y devuelve false.

Por ejemplo: Suponiendo que tenemos la estructura AVLTreePair<String, Integer) arbol:

```
arbol.put(4, 8);
        syso(arbol) → <4,8>
arbol.put(4, 9);
        syso(arbol) → <4,9>
arbol.put(4, null);
        syso(arbol) → <4,null>
```

Método get(): V ← get(K clave)

De alguna forma, podríamos decir que este método es equivalente al método find(), devolviendo: (1) si la clave existe, devuelve el valor asociado; (2) si la clave no existe, devuelve null.

Siguiendo con el ejemplo anterior:

```
arbol.put(7, 9);
        arbol.get(7) → 9
        arbol.get(10) → null
```

Método keySet(): ArrayList<K> ← keySet()

Devuelve en un contenedor ArrayList<K> el conjunto de claves existentes en la estructura arborescente.

Método values(): ArrayList<V> ← values()

Devuelve en un contenedor ArrayList<K> el conjunto de valores existentes en la estructura arborescente.

Método entrySet(): ArrayList<Pair<K,V>> ← entrySet()

Devuelve en un contenedor ArrayList<Pair<K,V>> el conjunto de pares (clave, valor) existentes en la estructura arborescente.

El secreto para entender cómo funciona este árbol de pares está en la implementación de la clase Pair<K,V> (algunos ya la conocéis). La cabecera de esta clase es:

```
public class Pair <K extends Comparable<K>, V> implements Map.Entry<K, V>, Comparable<Pair<K,V>>{
…
}
```

Indicando los siguientes aspectos fundamentales:

- La clave (K) debe ser Comparable<K>.
- Implementa el protocolo Map.Entry<K,V> (esto no es relevante por ahora).
- Implementa la interfaz Comparable<Pair<K,V>>.

Al adoptar el protocolo (o la interfaz) Comparable<Pair<K,V>> tenemos que especificar un comportamiento (implementar) al método compareTo():

```
@Override
public int compareTo(Pair<K,V> o) {
        return this.key.compareTo(o.key);
}
```

Que, como podéis observar, delega la comparación de dos pares en la comparación de sus claves...ignorando los valores asociados a dichas claves. Intuitivamente, podemos ver que cuando inserto un par en un árbol, lo que hace el algoritmo es comparar la clave del nuevo par con la clave del nodo raíz y decidir: (1) si es menor (desciende al subárbol izquierdo); (2) si es mayor (desciende al subárbol derecho); y (3) si es igual (sustituyendo el valor).

Como campo de aplicación, vamos a utilizar nuestro mapa como estructura base para la gestión de las puntuaciones que obtienen los sujetos de un sistema (con identificador *sujetoID*) en diferentes pruebas a las que han sido sometidos. Como requisito del problema, es necesario almacenar el nombre de dichas pruebas para su posterior identificación. Cada prueba, que podrá realizarse en diferentes momentos, consta de un conjunto de ítems, donde a cada ítem se le asigna una puntuación numérica. El tipo de ítem o su orden no es relevante, de manera que a cada identificador de prueba se le asociará una secuencia de valores numéricos.

Así pues, la clase `Sujeto` va a constar de un identificador (su nombre) y de una estructura arborescente de pares con clave de tipo String (identificador de la prueba), asociada a una referencia a un objeto de tipo `ArrayList<Double>`. Una vez cargados los datos en memoria, el conjunto de operaciones básicas consiste, básicamente, en la obtención del número de ítems evaluados, así como la obtención de la puntuación máxima (en general o en una prueba específica).

Para más detalles de implementación, consultad el test, así como la base del programa que se encuentra en el repositorio de la asignatura.