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 desde el punto de vista de almacenamiento de datos como para consultas sobre ellos.
- 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 las estructuras de datos arborescentes `BSTree<T>` y `AVLTree<T>`, y la combinación de ellas para la resolución de problemas.
- Contestar adecuadamente a las preguntas que se proponen y en el archivo correspondiente.
- Pasar los respectivos test que se suministran.
- Entrega de todo el material en el repositorio en la fecha acordada, comunicando su finalización a través de la entrada correspondiente en el menú Tareas de la WebCT.

Enunciado

Ejercicio 1. Árbol binario de búsqueda. Uso del árbol binario de búsqueda (BST, Binary Search Tree) para el control de una puerta de acceso a una red.

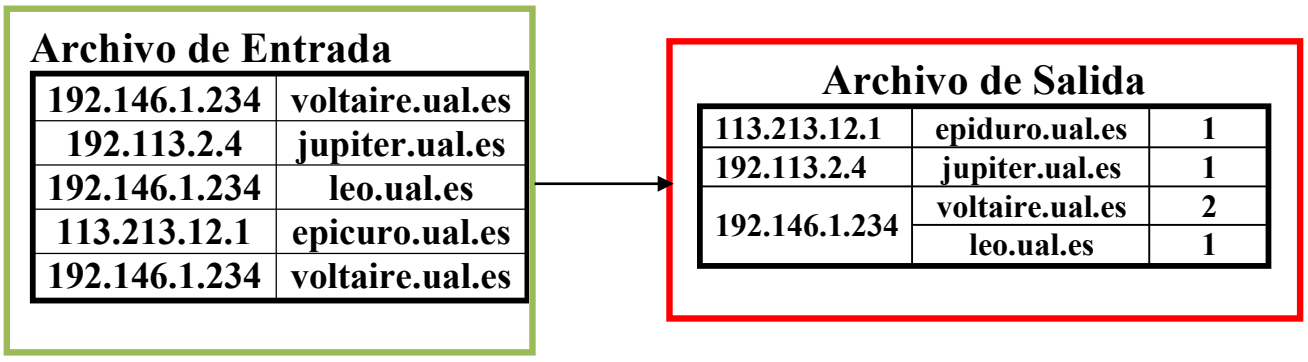
Las direcciones IP identifican de forma unívoca todos y cada uno de los dispositivos (ordenadores, impresoras,…) conectados en Internet. Cada IP tiene asociado un nombre de máquina; por ejemplo, *filabres.ual.es* tiene asociada la dirección IP *150.214.156.2*. Estas direcciones están formadas por cuatro campos que representan partes específicas de Internet, *red.subred.subred.máquina*, que el ordenador trata como una única dirección IP. Esta dirección consta de 32 bits, aunque normalmente se representa en notación decimal, separando los bits en cuatro grupos de 8 bits cada uno, expresando cada campo como un entero decimal (0...255) y separando los campos por un punto. En el ejemplo anterior: *filabres.ual.es* ⇔ *150.214.156.2*. En nuestra implementación se va a suponer que tanto la dirección IP como el nombre de la máquina asociada van a ser de tipo *String*.

Supongamos que se establecen conexiones desde una red a través de una puerta de acceso a otra red y, cada vez que se establece una conexión, la dirección IP del ordenador del usuario se almacena en un archivo junto con el nombre de la máquina. Estas direcciones IP y sus máquinas se pueden recuperar periódicamente para controlar quiénes han utilizado la puerta de acceso (y si ha habido varias máquinas asociadas a una misma dirección IP) y cuántas veces han hecho una conexión.

Como **EJERCICIO** se pide lo siguiente:

- Implementad un programa en Java que lea desde un archivo estas direcciones IP (ver cuestiones de formato en el test correspondiente) y los nombres de las máquinas asociadas, y lo almacene en un **BSTree<T>** (Árbol Binario de Búsqueda). Cada uno de los objetos principales guardarán la dirección IP, el nombre de la máquina (o de las máquinas si hay más de una) y el número de veces que dicho par (dirección IP, nombre máquina) aparecerá en el archivo de entrada. La estructura que se pide para solucionar este ejercicio es un **BSTree<T>** de objetos *DireccionMaquinas* (con una dirección IP como *String* y un **BSTree<String>** con las máquinas que coinciden con dicha dirección así como el número de ocurrencias que aparece dicho par (dirección, máquina)... objeto que llamaremos *MaquinaContador*)).
- Según se lee cada dirección IP y nombre de máquina del archivo, el programa debe comprobar si dicho par está ya en el **BSTree<T>**. Si lo está, se incrementa en 1 el contador asociado a la máquina; en caso contrario, se inserta en el **BSTree<T>** asociado a *DireccionMaquinas* y en el **BSTree<T>** asociado a *MaquinaContador*. Es decir, un **BSTree<T>** de direcciones de **BSTree<T>** de máquinas (cada una con su contador correspondiente).
- Una vez que todas las direcciones IP y nombres de máquina han sido leídas del archivo de entrada, se generará como resultado un archivo de salida en el que cada línea se indique la dirección IP (ordenadas por dirección), lista de nombre máquinas, así como el valor de sus contadores (ordenados por máquinas).

Es decir, el resultado del ejercicio debe ser un archivo de salida donde aparezca la dirección IP, el nombre de máquinas asociadas a dicha IP y el número de conexiones realizadas, tal y como se muestra en la siguiente figura.



En primer lugar, y para comprobar la correcta realización de este ejercicio, se proporcionará el **test** que deberá ser validado correctamente.

Adicionalmente se deberá implementar métodos que respondan a las siguientes consultas:

- Devolver el número de máquinas con un determinado valor de contador proporcionado como parámetro.
- Devolver el valor de contador del par (direcciónIP, máquina).
- Devolver la suma de los contadores asociados a una dirección, así como la suma global, que deberá coincidir con el número de 3-uplas en el conjunto de entradas.

Se ha proporcionado todo lo relativo a la implementación del **BSTree<T>**, por lo que se pide que se **EXPLIQUE** en un documento (memoria) toda la implementación suministrada, prestando especial atención a funciones principales de dicha estructura de datos como: *add*, *remove*, *clear*, *contains*, *isEmpty* y al uso propuesto del iterador asociado. Razone y justifique adecuadamente cada una de las ventajas e inconvenientes del uso de esta estructura comparada con el uso de estructuras lineales.

La memoria consistirá en un archivo PDF con nombre **practica02_ejercicio01** y se almacenará en una carpeta denominada **Memorias** en la carpeta de Practicas de cada alumno. El archivo deberá tener un formato en el que se expliquen la **estructura de datos** del **BSTree<T>** y todas sus **operaciones** (*nombre*, *parámetros de entrada y salida)*, junto con una breve descripción de ***qué*** hace dicho método (su funcionalidad).

Ejercicio 2. Árbol binario de búsqueda AVL. Gestión de ciudades donde empresas de software desarrollan sus proyectos utilizando un AVLTree.

Como ya hemos estudiado en la Práctica 01, disponemos un conjunto de datos para gestionarla de forma eficiente mediante el uso de estructura de datos adecuada. Estos datos están relacionados con las empresas de software que desarrollan proyectos en determinados lugares donde tienen las sedes de dichos proyectos. Para este ejercicio tendremos un nuevo archivo de entrada "**nuevasEmpresasProyectosCiudades.txt**" (atención al formato de entrada).

Como **EJERCICIO** se pide lo siguiente: implementad un programa en Java que lea desde un archivo de entrada la anterior lista de **Empresa_Software -> Proyecto_Software -> Ciudad_Donde_Se_Desarrolla**, almacenando los datos en una estructura de datos arborescente.

- Según se lee cada línea del archivo (**Empresa_Software -> Proyecto_Software -> Ciudad_Donde_Se_Desarrolla**), el programa debe comprobar si dicha 3-upla está ya en el contenedor. Si lo está, no hacer nada pues será una línea repetida; en caso contrario, se inserta en la estructura de datos.
- 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 **AVLTree<T>**, se generará como resultado un archivo de salida y una serie de listados que deben seguir las instrucciones dictadas en el test correspondiente.

Como sugerencia para la realización del ejercicio, podemos plantear una estructura de datos basada en **AVLTree<T>**. De forma genérica podría plantearse lo siguiente (AVLTree de AVLTrees de AVLTrees)

AVLTree(Empresas, **AVLTree**(Proyectos, **AVLTree**(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:

- 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 se deberá pasar correctamente.

Adicionalmente, se ha proporcionado todo lo relativo a la implementación del **AVLTree<T>**, por lo que se pide que se **EXPLIQUE** en un documento (memoria) toda implementación suministrada, prestando especial atención a funciones principales de dicha estructura de datos como: *add*, *remove*, *clear*, *contains*, *isEmpty*, así como al uso propuesto del iterador. También se pide, en dicha memoria, que se enumeren las ventajas e inconvenientes de la resolución de este problema mediante el uso de estructuras arbóreas (**AVLTree<T>**) en lugar de colecciones lineales (**ArrayList<T>**) como se hizo la práctica 1 (Ejercicio 02). Razone y justifique, adecuadamente, cada una de las ventajas e inconvenientes enumerados en el archivo PDF con nombre **practica02_ejercicio02**, almacenándose en la carpeta **Memorias**.

Ejercicio 3. Árbol binario de búsqueda AVL. Corrector ortográfico.

Para finalizar la práctica, vamos a implementar un *sencillo corrector ortográfico*, es decir, un programa que lea frases (introducidas desde teclado) y compruebe su ortografía a través de un diccionario personalizado de palabras (que está almacenado en un archivo y se carga al empezar el programa, volviéndose a guardar en disco al finalizar la misma). El diccionario lo vamos a implementar mediante un ABB equilibrado tipo **AVLTree<T>**. El proceso de nuestro programa se resume en las siguientes líneas:

1. Al ejecutarse el programa, el árbol diccionario leerá desde archivo la lista de palabras actuales que lo componen.
2. El programa analizará, palabra por palabra, una frase que introducirá el usuario, realizando las siguientes acciones:
    a. Si la palabra existe, se le mostrará un OK de conformidad (la palabra está escrita correctamente).
    b. Si la palabra no existe, el programa le indicará este hecho, mostrando una o varias palabras (organizadas según un ranking) ***similares*** a la palabra analizada y contenidas en el diccionario. Ese conjunto de palabras similares podría estar almacenado en una ***PriorityQueue<T>*** (estudiar sus métodos y cómo utilizarlos). Si el usuario no está de acuerdo con la lista de palabras similares, podrá indicarle al programa que añada la nueva palabra al diccionario.
3. Al acabar la sesión, todo el árbol que contiene el diccionario se guardará en archivo (siguiendo un orden lexicográfico).

En primer lugar y como viene siendo habitual para comprobar la correcta realización de este

ejercicio se proporcionará el **test** (de las operaciones que serán necesarias) que se deberá pasar correctamente.

El criterio de ***similitud*** entre pares de palabras que vamos a seguir en este ejercicio se basará en la conocida *distancia de Levenshtein* (http://es.wikipedia.org/wiki/Distancia_de_Levenshtein). En este punto surgen algunas cuestiones que tendréis que pensar antes de implementar la solución:

- ¿Cuál es la idea básica de este método de similitud?.
- ¿Cómo utilizaremos esta distancia de edición para recuperar del AVL las *n* palabras más similares?.
- ¿Qué pasaría si el diccionario lo implementásemos mediante un **BSTree<T>**, en lugar de un **AVLTree<T>**?

Tal y como hemos hecho hasta ahora, las respuestas a estas preguntas se entregarán en un archivo con el siguiente nombre **practica02_ejercicio03.pdf** en la carpeta **Memorias**.

NOTA: Las cuestiones que no consten de forma explícita en el test, se considerarán ejercicios adicionales (no obligatorios).