

PRÁCTICA 2

Uso de árboles binarios de búsqueda en Java

Objetivos

- Repasar los árboles binarios de búsqueda (ABB) para la resolución de diferentes problemas. Tanto desde el punto de vista para el almacenamiento de datos como para las consultas sobre ellos.
- Aprender a utilizar la composición de estructuras de datos arbóreas en la resolución de problemas.
- Utilizar los ABB balanceados AVL para la resolución de problemas, comprobando lo apropiado de su uso en determinadas situaciones.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos arbóreas `BSTree` y `AVLTree`, 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.

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

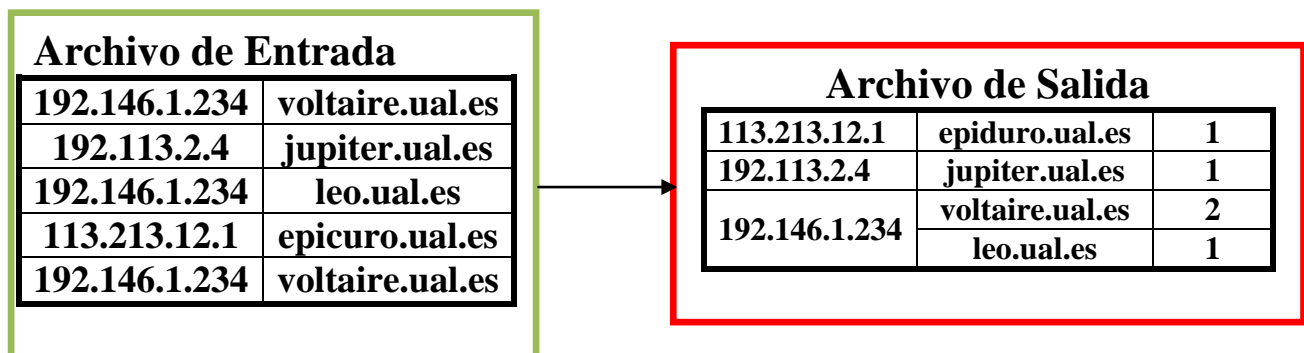
Las direcciones IP identifican de forma unívoca todos y cada uno de los dispositivos (ordenadores, impresoras,...) conectados en Internet. Cada IP tiene asociado un nombre de máquina; por ejemplo, *filabres.ual.es* tiene asociada la dirección IP *150.214.156.2*. Estas direcciones están formadas por cuatro 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* \Leftrightarrow *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:

- Implementar un programa en Java que lea desde un archivo estas direcciones IP y los nombres de las máquinas asociadas, y lo almacene en un **BSTree** (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** de objetos *DireccionMaquinas* (con una dirección IP como *String* y un **BSTree** con las máquinas (*String*) que coinciden con dicha dirección y el número de ocurrencias que aparece dicho par (dirección, máquina), que le 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**. Si lo está, se incrementa en 1 el contador asociado a la máquina; en caso contrario, se inserta en el **BSTree** asociado a *DireccionMaquinas* y en el **BSTree** asociado a *MaquinaContador*. Es decir, un **BSTree** de direcciones de **BSTrees** de máquinas (cada una con su contador).
- 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 donde en cada línea se indique la dirección IP (ordenadas por dirección), lista de nombre máquinas y sus contadores (ordenados por máquinas).

Es decir, el resultado del ejercicio debe ser un archivo de salida donde aparezca la dirección IP, el número de máquinas que tenidos 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 se deberá pasar correctamente. También es interesante destacar que la salida relativa (llamando a la función `ToString()`) al archivo de entrada más extenso del texto que se propone en la práctica debe ser la que se muestra a continuación.


```
((113.213.12.1, [[epi.ual.es, 1]
[epicuro.ual.es, 1]])
(150.214.156.17, [[almirez.ual.es, 3]
[antonio.ual.es, 1]
[rosa.ual.es, 1]])
(150.214.156.2, [[calido.ual.es, 1]
[filabres.ual.es, 1]
[pedro.ual.es, 1]])
(150.214.156.32, [[alboran.ual.es, 2]
[antonio.ual.es, 1]])
(150.214.194.195, [[nevada.ugr.es, 3]])
(150.214.20.25, [[nacho.ugr.es, 1]
[veleta.ugr.es, 1]])
(192.113.2.4, [[jupiter.ual.es, 1]])
(192.146.1.233, [[pascal.ual.es, 2]
[poisson.ual.es, 1]])
(192.146.1.234, [[leo.ual.es, 2]
[pedro.ual.es, 2]
[voltaire.ual.es, 3]])
(192.146.1.244, [[ant.ual.es, 1]
[antonio.ual.es, 1]])
(92.140.12.255, [[rosa.ual.es, 2]])
(97.100.7.155, [[antonio.ual.es, 4]]))
```


Adicionalmente se deberán responder a las siguientes consultas:


- Devolver el número de máquinas con un determinado valor de contador dado como parámetro.
- Devolver los pares (direcciónIP, máquina) que tienen un determinado valor de contador dado como parámetro.
- Devolver el valor del contador de un determinado par (direcciónIP, máquina), en el caso de que no exista debe devolver -1.

Se ha proporcionado todo lo relativo a la implementación del **BSTree**, 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*, y al iterador *iterator* del tipo *TreeIterator*. Razone y justifique adecuadamente cada una de las ventajas e inconvenientes enumerados.

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

 org.eda1.practica02.ejercicio01.MaquinaContador
<ul style="list-style-type: none"> maquina: String contador: int
<ul style="list-style-type: none"> MaquinaContador(maquina: String, contador: int) MaquinaContador(maquina: String) MaquinaContador() setMaquina(maq: String): void getMaquina(): String setContador(c: int): void getContador(): int incrementarContador(): void compareTo(otraMaquinaContador: Object): int equals(obj: Object): boolean toString(): String

 org.eda1.practica02.ejercicio01.DireccionMaquinas
<ul style="list-style-type: none"> direccion: String maquinas: BSTree<MaquinaContador>
<ul style="list-style-type: none"> DireccionMaquinas() DireccionMaquinas(direccion: String) DireccionMaquinas(direccion: String, maquina: String) DireccionMaquinas(direccion: String, maquina: String, contador: int) setDireccion(dir: String): void getDireccion(): String getMaquinas(): BSTree<MaquinaContador> addMaquina(mc: MaquinaContador): boolean addMaquinaWithFind(mc: MaquinaContador): boolean compareTo(otraDireccionMaquina: Object): int equals(obj: Object): boolean toString(): String

 org.eda1.practica02.ejercicio01.ProcesarDirecciones
<ul style="list-style-type: none"> treeDirecciones: BSTree<DireccionMaquinas>
<ul style="list-style-type: none"> ProcesarDirecciones() ProcesarDirecciones(treeDirecciones: BSTree<DireccionMaquinas>) cargarArchivo(archivo: String): BSTree<DireccionMaquinas> guardarDireccionesIncidencias(archivo: String): void addDireccionMaquina(direccion: String, maquina: String): boolean addDireccionMaquinaWithFind(direccion: String, maquina: String): boolean maquinasConContador(contador: int): int direccionMaquinasConContador(contador: int): String contadorDeDireccionMaquina(direccion: String, maquina: String): int

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 de información para gestionarla en una estructura de datos. Ésta está relacionada 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**”.

Como **EJERCICIO** se pide lo siguiente: implementar un programa en Java que lea desde un archivo de entrada la anterior lista de **Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**, almacenando en una estructura de datos basada en **AVLTree**.

- Según se lee cada línea del archivo (**Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**), el programa debe comprobar si dicha tripleta 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**, se generará como resultado un archivo de salida y un listado que debe coincidir con el que se proporciona en el test correspondiente.

Como sugerencia para la realización del ejercicio, podemos plantear una estructura de datos basada en **AVLTree**. 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 haciendo uso, donde proceda, del iterador asociado a **AVLTree**:

- Devolver las empresas que tienen su sede en la ciudad Miami.
- Devolver los *proyectos* con sede en Washington.
- ¿En cuántas ciudades diferentes se desarrollan proyectos de **Apple**?
- Devolver las ciudades, donde desarrollándose el proyecto *PowerPoint*, se desarrollan a su vez proyectos de otras empresas distintas de *Micrsoft*.
- Devolver los proyectos de la empresa *Oracle*, que tienen alguna sede (ciudad) común, indicando también el nombre de dicha ciudad. Haced lo mismo para la empresa *Microsoft*.

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**, 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*, y al iterador *iterator* del tipo *TreeIterator*. 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**) en lugar de colecciones lineales (**ArrayList**) 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_AVLTree**, almacenándose en la carpeta **Memorias**.

Para finalizar, se debe explicar cómo se incluiría en nuestra estructura de datos la siguiente clase denominada *CiudadDirecciones*, para que se pudieran considerar diferentes direcciones en una misma ciudad donde se puede desarrollar un mismo proyecto software. Además, se debe de describir (similar a pseudocódigo) cómo se implementaría la consulta que devolvería las empresas que tienen proyectos en ciudades europeas, indicando también cuáles son y en qué dirección se alojan. Todo ello, en el archivo PDF con nombre **practica02_ejercicio02_CiudadDirecciones**, almacenándose en la carpeta **Memorias**.

```
package org.edal.practica02.ejercicio02;
import org.edal.estructurasdedatos.AVLTree;
public class CiudadDirecciones implements Comparable {
    public String ciudad;
    String pais;
    String continente;
    AVLTree<String> direcciones = new AVLTree<String>();

    public CiudadDirecciones(String ciudad, String pais, String continente) {
        this.ciudad = ciudad;
        this.pais = pais;
        this.continente = continente;
    }

    public void setCiudad(String ciudad) { this.ciudad = ciudad; }




    public String getCiudad() { return ciudad; }

    public boolean addDireccion(String direccion) { ... }

    public AVLTree<String> getDirecciones() { return this.direcciones; }

    public int compareTo(Object otraCiudadDirecciones) { ... }

    // Otras funciones adicionales
}
```

<p> org.eda1.practica02.ejercicio02.ProyectoCiudades</p> <ul style="list-style-type: none"> proyecto: String ciudades: AVLTree<String> ProyectoCiudades() ProyectoCiudades(proy: String) setProyecto(proy: String): void getProyecto(): String addCiudad(ciudad: String): boolean getCiudades(): AVLTree<String> compareTo(otroProyectoCiudades: Object): int size(): int 	<p> org.eda1.practica02.ejercicio02.EmpresaProyectos</p> <ul style="list-style-type: none"> empresa: String proyectosCiudades: AVLTree<ProyectoCiudades> EmpresaProyectos() EmpresaProyectos(empr: String) addProyectoCiudad(proyecto: String, ciudad: String): boolean addProyectoCiudadWithFind(proyecto: String, ciudad: String): boolean setEmpresa(empr: String): void getEmpresa(): String getProyectosCiudades(): AVLTree<ProyectoCiudades> compareTo(otraEmpresaProyectos: Object): int size(): int
<p> org.eda1.practica02.ejercicio02.ProcesarDatos</p> <ul style="list-style-type: none"> <u>cargarArchivo(archivo: String): AVLTree<EmpresaProyectos></u> <u>addEmpresaProyectoCiudad(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String, proyecto: String, ciudad: String): boolean</u> <u>addEmpresaProyectoCiudadWithFind(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String, proyecto: String, ciudad: String): boolean</u> <u>mostrarEmpresasProyectosCiudades(listaEmpresas: AVLTree<EmpresaProyectos>): void</u> <u>guardarEmpresasProyectosCiudades(listaEmpresas: AVLTree<EmpresaProyectos>, archivo: String): void</u> <u>numeroProyectosEmpresa(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String): int</u> <u>numeroCiudadesProyecto(listaEmpresas: AVLTree<EmpresaProyectos>, proyecto: String): int</u> <u>numeroCiudadesEmpresa(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String): int</u> <u>devolverEmpresasProyectosCiudades(listaEmpresas: AVLTree<EmpresaProyectos>): String</u> <u>devolverEmpresasCiudad(listaEmpresas: AVLTree<EmpresaProyectos>, ciudad: String): ArrayList<String></u> <u>devolverProyectosCiudad(listaEmpresas: AVLTree<EmpresaProyectos>, ciudad: String): ArrayList<String></u> <u>devolverCiudadesEmpresa(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String): ArrayList<String></u> <u>devolverCiudadesProyectoEmpresa(listaEmpresas: AVLTree<EmpresaProyectos>, proyecto: String, empresa: String): ArrayList<String></u> <u>devolverEmpresaParesProyectoCiudadesComunes(listaEmpresas: AVLTree<EmpresaProyectos>, empresa: String): ArrayList<String></u> 	

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 AVLTree**. 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 usuario podrá introducir desde teclado una frase, el programa deberá analizar la frase, palabra por palabra, 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 **java.util.PriorityQueue** (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 parejas 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 (*edit distance*, como también se le conoce a la distancia de Levenshtein) para recuperar del AVL las n palabras más similares?.
- ¿Qué pasaría si el diccionario lo implementásemos mediante un **BSTree**, en lugar de con un **AVLTree**?

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**.

org.eda1.practica02.ejercicio03.CorrectorOrtografico
treePalabras: AVLTree<String>
CorrectorOrtografico()
CorrectorOrtografico(treePalabras: AVLTree<String>)
cargarDiccionario(archivo: String): AVLTree<String>
guardarDiccionario(archivo: String): void
minimum(a: int, b: int, c: int): int
computeLevenshteinDistance(str1: String, str2: String): int
computeLevenshteinDistance(str1: char[], str2: char[]): int
listaSugerencias(n: int, s: String): ArrayList<String>
addPalabra(palabra: String): void
containsPalabra(palabra: String): boolean
size(): int
add(palabra: String): boolean
find(palabra: String): boolean

org.eda1.practica02.ejercicio03.DistanceComparator
compare(x: PalabraDistancia, y: PalabraDistancia): int

org.eda1.practica02.ejercicio03.Principal
main(args: String[]): void
LeerFrase(): ArrayList<String>

org.eda1.practica02.ejercicio03.PalabraDistancia
palabra: String
distancia: int
PalabraDistancia(pal: String, dis: int)
PalabraDistancia(pal: String)
PalabraDistancia()
setPalabra(pal: String): void
getPalabra(): String
setfrecuencia(dis: int): void
getDistancia(): int
compareTo(other: Object): int
equals(obj: Object): boolean
toString(): String