ESTRUCTURAS DE DATOS Y ALGORITMOS I
Grado en Ingeniería Informática
Convocatoria Ordinaria - Enero 2023

Nombre:	**DNI:**

INSTRUCCIONES

- **Es obligatorio la inclusión de comentarios claros, concisos y concretos que justifiquen los aspectos más relevantes de las respuestas planteadas.**
- **Deberá siempre destacar la solución final, indicando en todo momento el proceso que se ha seguido para su consecución.**
- **Es obligatorio incluir comentarios en los algoritmos que se implementen indicando qué hacen los bloques de código más relevantes.**
- **Implementar siempre la alternativa más eficiente y con la estructura de datos más apropiada. De lo contrario, no se considerará correcta la solución.**
- **Debe utilizarse exclusivamente bolígrafo para responder al examen; no utilizar lápiz ni tinta de color rojo.**

TEORÍA (2p) (Elige uno de los dos ejercicios propuestos)

1. (2p) Algoritmo de Floyd.
   a. Fundamento del algoritmo de *Floyd* para la obtención de caminos mínimos entre todos los pares de vértices de un grafo.
   b. Supongamos que ejecutamos el algoritmo de *Floyd* (también conocido como algoritmo de *Floyd-Warshall*) sobre el grafo que se especifica en la Figura 1, obteniendo las matrices de coste y de traza especificadas en la Figura 2. A partir de las figuras, determina, razonadamente: (1) ¿qué tipo de grafo es?; (2) el camino mínimo que une el vértice 7 con el vértice 6.

	0	1	2	3	4	5	6	7
0		9		2	9			
1	9		4	8		2	8	
2		4			9	1		
3	2	8				1		8
4	9		9				8	8
5		2	1	1				1
6		8			8			
7				8	8	1		

Figura 1. Grafo

Cost Table

	0	1	2	3	4	5	6	7
0	INF	5	4	2	9	3	13	4
1	5	INF	3	3	11	2	8	3
2	4	3	INF	2	9	1	11	2
3	2	3	2	INF	10	1	11	2
4	9	11	9	10	INF	9	8	8
5	3	2	1	1	9	INF	10	1
6	13	8	11	11	8	10	INF	11
7	4	3	2	2	8	1	11	INF

Path Table

	0	1	2	3	4	5	6	7
0	-1	5	5	0	0	3	1	5
1	3	-1	5	5	7	1	1	5
2	3	5	-1	5	2	2	1	5
3	3	5	5	-1	7	3	1	5
4	4	5	4	5	-1	7	4	4
5	3	5	5	5	7	-1	1	5
6	3	6	5	5	6	1	-1	5
7	3	5	5	5	7	7	1	-1

Figura 2. Matrices de coste y de traza

2. (2p) Sea T el árbol binario de búsqueda bien equilibrado (tipo AVL) que se muestra en la Figura 3.

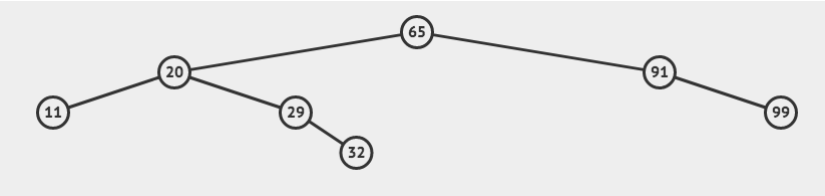

Figura 3. AVL

   a. ¿Es T realmente un árbol AVL? Razona la respuesta a partir del concepto de factor de equilibrio de un nodo.
   b. En caso de que la respuesta a la pregunta anterior sea afirmativa (T es AVL), ¿cómo quedaría el árbol tras eliminar el valor **91**? Razona detalladamente la respuesta.

PRÁCTICA (8p)

3. (3p) Basándonos en la Parte 02 de la Práctica 03, en el **Anexo** podemos ver una definición alternativa (y más simplificada) de la clase

GestionRepositorios a la que llamaremos ***GestionArticulos***. Como puede observar, en esta versión hacemos uso de una estructura de datos asociativa anidada, llamada *datos*, en la que se va a almacenar un conjunto de pares *(autorID, lista_de_articulos)*, donde cada *artículo* se representa con un conjunto de pares *(articuloID, palabrasFrecuencias)*, siendo *palabrasFrecuencias* una colección de pares donde se almacenan las distintas palabras que aparecen en el artículo, así como su frecuencia (el número de veces que se repiten). A partir de la estructura de datos anidada principal, *datos*, vamos a implementar el método ***getPalabrasAutor(String)***. Este método devolverá un mapa formado por la combinación de todos y cada uno de los conjuntos de pares *(palabra, frecuencia)* asociados con el autor cuya clave se indica como parámetro de entrada (*autorID*).

Por ejemplo. Supongamos que insertamos los siguientes datos:

```
GestionArticulos gestion = new GestionArticulos();
gestion.add("autor01", "articulo01_01", "w01", "w02", "w03");
gestion.add("autor02", "articulo02_01", "w01", "w02", "w03");
gestion.add("autor03", "articulo03_01", "w03", "w03", "w03");
gestion.add("autor01", "articulo01_02", "w01", "w02", "w04", "w01");
gestion.add("autor02", "articulo02_02", "w01", "w02", "w03", "w01");
gestion.add("autor01", "articulo01_03", "w03", "w02", "w01");
```

Tras mostrar el contenido de la estructura en Consola *(gestion.toString()*) obtenemos la siguiente cadena:

```
{autor01={articulo01_01={w01=1, w02=1, w03=1},
          articulo01_02={w01=2, w02=1, w04=1},
          articulo01_03={w01=1, w02=1, w03=1}},
 autor02={articulo02_01={w01=1, w02=1, w03=1},
          articulo02_02={w01=2, w02=1, w03=1}},
 autor03={articulo03_01={w03=3}}}
```

En este caso, la salida del método *getPalabrasAutor(String)* debe ser:

```
gestion.getPalabrasAutor("autor01") → {w01=4, w02=3, w03=2, w04=1}
gestion.getPalabrasAutor("autor02") → {w01=3, w02=2, w03=2}
gestion.getPalabrasAutor("autor03") → {w03=3}
gestion.getPalabrasAutor("autor33") → {}
```

4. (2.5p) Vamos a observar con detenimiento los detalles de la clase **MySimpleBSTree**<T> especificada en el **Anexo**. En particular, vamos a analizar la implementación del método **getNumLeaves()** que, supuestamente, devuelve el número de hojas que tiene un árbol binario de búsqueda determinado. Para saber si el método es correcto (o no), comprobando si la salida esperada es igual que la salida real, vamos a hacer un seguimiento exhaustivo (tal y como se ha estudiado en clase) partiendo del árbol que se genera tras la inserción de la secuencia de datos [10, 6, 15, 3, 20, 17, 23]. En caso de obtener una salida real distinta a la salida esperada, corrige el método y demuestra que funciona correctamente.

5. (2.5p) Basándonos en la Parte 03 de la Práctica 02, en el **Anexo** podemos ver una definición alternativa de la clase **User**. En este caso, la estructura de datos principal, llamada *dispositivos*, consiste en una colección basada en árbol binario de búsqueda básico de pares con clave String (id del dispositivo) y valor ArrayList<String> (colección de palabras enviadas por el usuario haciendo uso del dispositivo especificado como clave). Vamos a implementar el método *add()* teniendo en cuenta: (1) la restricción especificada en el Anexo; y (2) en la colección de palabras no se permiten palabras repetidas.

Por ejemplo, supongamos que insertamos los siguientes datos:

```
User user = new User("user01");
user.add("dev01", "w01", "w02", "w01", "w02");
user.add("dev01", "w01", "w02", "w01");
user.add("dev02", "w01", "w01", "w02");
user.add("dev02", "w01", "w02", "w02");
user.add("dev03", "w01", "w01", "w01");
user.add("dev03", "w02", "w02", "w02");
```

La salida que se muestra en Consola debe ser la siguiente:

System.*out*.println(user) → user01=<dev01 <[w01, w02]> dev02 <[w01, w02]> dev03 <[w01, w02]> >

ANEXO

```
public class GestionArticulos {
        private TreeMap<String, TreeMap<String, TreeMap<String, Integer>>> datos = new TreeMap<>();
        public void add(String autorID, String articuloID, String...palabras) {...}
        public TreeMap<String, Integer> getPalabrasAutor(String autorID){
                TreeMap<String, Integer> result = new TreeMap<>();
              //...
                return result;
        }
        @Override
        public String toString() {
                return this.datos.toString();
        }
}
```

```
public class User implements Comparable<User>, Iterable<String> {
    private String userID;
    private MySimpleBSTree<Pair<String, ArrayList<String>>> dispositivos;
    public User(String userID) {...}
    public void clear() {...}
    public void add(String dispositivoID, String...palabras) {
        //1 única variable local
        //…
     }
    @Override
     public String toString() {
        return this.userID + "=<" + this.dispositivos.toString() + ">";
     }
    @Override public boolean equals(Object o) {...}
    @Override public int compareTo(User o) {...}
    @Override public Iterator<String> iterator() {...}
}
```

```
public class Pair <K extends Comparable<K>,V> implements Comparable<Pair<K,V>>{
        private final K key;
        private V value;
        public Pair(K key, V value) {...}
        public K getKey() {...}
        public V getValue() {...}
        public void setValue(V value) {...}
        @Override
        public String toString() {
                return key + " <" + value + ">";
        }
        @Override
        public boolean equals(Object o) {
                Pair<?,?> p = (Pair<?,?>)o;
                return this.key.equals(p.key);
        }
        @Override
        public int compareTo(Pair<K,V> o) {
                return this.key.compareTo(o.key);
        }
}
```

```
public class MySimpleBSTree<T extends Comparable<T>> implements Iterable<T>{
    private static class BSTNode<T> {
            private T nodeValue;
            private BSTNode<T> left, right;
            public BSTNode(T value) {
                    nodeValue = value;
                    left = null;
                    right = null;
            }
    }
    private BSTNode<T> root;
    public MySimpleBSTree() {...
    public void clear() {...}
    private void clear(BSTNode<T> current) {...}
    public boolean add(T item) {...}
    private BSTNode<T> add(BSTNode<T> current, T item) {...}
    public T find(T item) {...}
    private T find(BSTNode<T> current, T item) {...}
    private String toString(BSTNode<T> current) {...}
    public String displayTree() {...}
    private String displayTree(BSTNode<T> current, int level) {...}
    public ArrayList<T> toArray(){...}
    private void toArray(BSTNode<T> current, ArrayList<T> a) {...}
    @Override public String toString() {...}
    @Override public Iterator<T> iterator() {...}

    public int getNumLeaves() {
            return getNumLeaves(this.root);
    }

    private int getNumLeaves(BSTNode<T> current) {
            if (current == null) return 0;
            if (current.left  == null || current.right == null) return 1;
            return getNumLeaves(current.left) + getNumLeaves(current.right);
    }
}
```