ESTRUCTURAS DE DATOS Y ALGORITMOS I
Grado en Ingeniería Informática
Convocatoria Ordinaria - Enero 2022

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 (3p)

1. (1.5p) 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-Warshal*) 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 dichas matrices, obtened razonadamente los caminos mínimos que unen: (1) el vértice 0 con el vértice 5; (2) el vértice 5 con el vértice 0; y (3) el vértice 4 con el vértice 3.

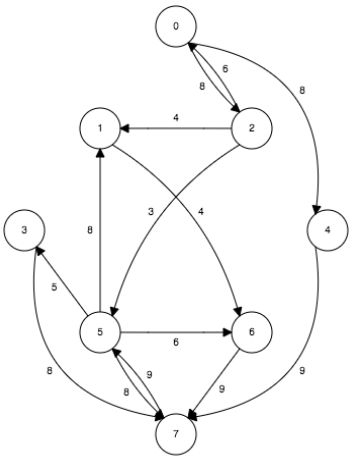

Figura 1. Grafo

	0	1	2	3	4	5	6	7
0	INF	12	8	16	8	11	16	17
1	INF	INF	INF	27	INF	22	4	13
2	6	4	INF	8	14	3	8	11
3	INF	25	INF	INF	INF	17	23	8
4	INF	26	INF	23	INF	18	24	9
5	INF	8	INF	5	INF	INF	6	8
6	INF	26	INF	23	INF	18	INF	9
7	INF	17	INF	14	INF	9	15	INF

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

Figura 2. Matrices de coste y de traza

2. (1.5p) Técnica de direccionamiento abierto para la resolución de colisiones en Tablas Hash.

   a) Fundamento, expresión general y casos particulares.
   b) Partiendo de una tabla hash, de tamaño N=11, con direccionamiento abierto (inicialmente vacía), con función hash principal $H_1(x) = x\%11$, realice la inserción de los valores de clave 3, 14, 25, 17 y 1, utilizando como método de resolución de colisiones la técnica de doble hash, siendo la función de dispersión secundaria $H_2(x) = 7 - x \% 7$.

PRÁCTICA (7p)

3. (2.5p) Basándonos en la Parte 02 de la Práctica 03, en el **Anexo** podemos ver una definición alternativa (y más simplificada) de la clase **GestionArticulos**. Como puede observar, en esta versión hacemos uso de una estructura de datos asociativa, llamada *datos*, en la que vamos a almacenar un conjunto de pares *(autorID, lista_de_articulos)*, donde cada *artículo* se representa como un conjunto de pares *(articuloID, palabras)*. Tened en cuenta que la colección de palabras que componen un artículo se va a almacenar en una estructura de datos lineal, de forma secuencial según el orden de aparición, existiendo la posibilidad obvia de que existan palabras repetidas. A partir de la estructura de datos anidada principal, *datos*, vamos a implementar el método *getPalabrasAutor(String)*. Este método, que hará uso únicamente de las variables locales *result* y *articulos* (observad detenidamente la implementación parcial que se propone en el **Anexo**), devolverá un mapa formado por el conjunto de pares *(palabra, frecuencia)* asociado con el autor que se indica como

parámetro de entrada (*autorID*). Como única restricción adicional se indica que, en el caso de que el usuario no exista, se devolverá el mapa vacío.

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", "w03", "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()*) obtendremos la siguiente cadena:

```
{autor01={articulo01_01=[w01, w02, w03], articulo01_02=[w01, w02, w03, w01], articulo01_03=[w03, w02, w01]},
 autor02={articulo02_01=[w01, w02, w03], articulo02_02=[w01, w02, w03, w01]},
 autor03={articulo03_01=[w03, w03, w03]}}
```

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

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

4. (2.5p) Basándonos en la Parte 03 de la Práctica 03, en el **Anexo** podemos ver la definición de la clase **GestionHashTag**. En este caso, la estructura de datos principal, llamada *datos*, consiste en un mapa basado en Hash formado por el conjunto de pares *(hashTag, palabras)*, donde hashTag denota referencias a objetos de tipo HashTag (ver esqueleto y detalles importantes de la clase HashTag especificada en el **Anexo**). Tened en cuenta que la colección de palabras de los mensajes que contienen un hashtag determinado se va a almacenar en una estructura de datos lineal, de forma secuencial según el orden de aparición, existiendo la posibilidad obvia de que existan palabras repetidas. A partir de la estructura de datos principal, *datos*, vamos a implementar el método *toStringOrdered()*. Este método, que hará uso únicamente de las variables locales *aux* y *result* (observad detenidamente la implementación parcial que se propone en el **Anexo**), devolverá una cadena de caracteres mostrando el conjunto ordenado formado por todos los pares (país, frecuencia), donde país denota el identificador de país y frecuencia representa el número de palabras diferentes asociado con cada país (independientemente del identificador de hashtag utilizado). Como se podrá observar en el siguiente ejemplo, el conjunto estará ordenado por orden ascendente, teniendo como clave única el identificador de país.

   Por ejemplo. Supongamos que insertamos los siguientes datos:

```
GestionHashTags gestion = new GestionHashTags();
gestion.add("#ht01", "pais01", "w01", "w02", "w03", "w04", "w04", "w04");
gestion.add("#ht01", "pais01", "w05", "w06");
gestion.add("#ht01", "pais02", "w01", "w02", "w03");
gestion.add("#ht01", "pais02", "w04");
gestion.add("#ht02", "pais01", "w01", "w02", "w03");
gestion.add("#ht02", "pais03", "w01");
```

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

```
{ht01 <pais01>=[w01, w02, w03, w04, w04, w04, w05, w06],
 ht01 <pais02>=[w01, w02, w03, w04],
 ht02 <pais03>=[w01],
 ht02 <pais01>=[w01, w02, w03]}
```

   En este caso, la salida del método *toStringOrdered()* deberá ser:

```
gestion.toStringOrdered() → {pais01=6, pais02=4, pais03=1}
```

5. (2p) Basándonos en la Parte 03 de la Práctica 02, presentamos esta versión de la clase **AVLTreePair<K,V>** especificada en el **Anexo**. Como podemos observar, este árbol de alto nivel encapsula un árbol equilibrado de pares, en concreto:

```
private AVLTree<Pair<K,V>> tree = new AVLTree<Pair<K,V>>();
```

Observad con detenimiento los detalles de las clases auxiliares **AVLTree<T>** y **Pair<K,V>** que se detallan en el **Anexo**. Tal y como están declaradas las clases, y sin entrar en detalles de implementación de los distintos métodos que suponemos que están definidos y sin errores, *Eclipse* nos informaría de un error de compilación importante. ¿En qué consiste dicho error? Una vez identificado y resuelto el error, detalla en Java una versión eficiente del método *toTreeMap()* perteneciente a la clase **AVLTreePair<K,V>** (tened en cuenta la declaración especificada en el **Anexo**). Simplemente, el método vuelca los datos del árbol de pares en un mapa basado en árbol binario de búsqueda.

Por ejemplo. Supongamos que insertamos los siguientes datos:

```
AVLTreePair<Integer, Integer> arbol = new AVLTreePair<>();
arbol.put(1, 1);
arbol.put(2, 2);
arbol.put(3, 3);
```

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

```
[<1, 1>, <2, 2>, <3, 3>]
```

En este caso, la salida del método *toTreeMap()* deberá ser:

```
arbol.toTreeMap() → {1=1, 2=2, 3=3}
```

ANEXO

```
public class GestionArticulos {
        private TreeMap<String, TreeMap<String, ArrayList<String>>> 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<>();
                TreeMap<String, ArrayList<String>> articulos = //…
                    //…
                return result;
        }
        @Override
        public String toString() {
                return this.datos.toString();
        }
}
```

```
public class GestionHashTags {
        private HashMap<HashTag, ArrayList<String>> datos = new HashMap<>();
        public void add(String hashTagID, String paisID, String...palabras) {...}
        public String toStringOrdered() {
                TreeMap<String, TreeSet<String>> aux = new TreeMap<>();
                String result = "{";
                    //…
                return result + "}";
        }
        @Override
        public String toString() {
                return this.datos.toString();
        }
}
```

```java
public class HashTag {
    private String hashTagID;
    private String paisID;
    public HashTag(String hashTagID, String pais) {...}
    public String getPaisID() {...}
    public String getHashTagID() {...}
    @Override
    public int hashCode() {
        return Objects.hash(this.hashTagID, this.paisID);
    }
    @Override
    public boolean equals(Object o) {
        return this.hashCode() == ((HashTag) o).hashCode();
    }
    @Override
    public String toString() {
        return this.hashTagID + " <" + this.paisID + ">";
    }
}
```

```java
public class AVLTreePair<K extends Comparable<K>,V> {
    private AVLTree<Pair<K,V>> tree = new AVLTree<Pair<K,V>>();

    public boolean put(K key, V value) {...}
    public V get(K key) {...}
    public boolean containsKey(K key) {...}
    public void clear() {...}
    public boolean isEmpty() {...}
    public int size() {...}
    public ArrayList<K> keySet(){...}
    public ArrayList<V> values(){...}
    public ArrayList<Pair<K,V>> entrySet(){...}
    @Override
    public String toString() {...}
    public TreeMap<K,V> toTreeMap(){
            //…
    }
}
```

```java
public class Pair <K ,V> {
        private final K key;
        private V value;
        public Pair(K key, V value) {...}
        public K getKey() {...}
        public V getValue() {...}
        public V setValue(V value) {...}
        @Override
        public String toString() {
          return "<" + key + ", " + value + ">";
        }
}
```

```java
public class AVLTree<T extends Comparable<T>>
                                implements Iterable<T>{

    private static class AVLNode<T> {
            public T nodeValue;
            public AVLNode<T> left, right;
            public int height;
            public AVLNode(T item) {..}
    }
    private AVLNode<T> root;
    private int treeSize;

    public AVLTree() {...}
    private void deleteTree(AVLNode<T> current) {...}
    public T find(T item) {}
    private AVLNode<T> findNode(T item) {...}
    public boolean add(T item) {...}
    public boolean remove(T item) {...}
    //….
    public void clear() {...}
    @Override
    public boolean contains(T item) {...}
    public boolean isEmpty() {...}
    public int size() {...}
    public ArrayList<T> toArray() {...}
    @Override
    public String toString() {...}
     @Override
    public Iterator<T> iterator() {...}
}
```